БОЛЬШОЙ
самоучитель

Delphi XE3

Рубанцев Валерий



Валерий Рубанцев

Большой самоучитель Delphi XE3



Эта книга – условно-бесплатная, Ознакомившись с её содержанием, Вы должны либо стереть книгу, либо оплатить её стоимость.

Стоимость книги и формы оплаты Вы найдёте на сайте:

www.rvGames.de/buy.htm

Все права защищены. Никакая часть этой книги не может быть воспроизведена в любой форме без письменного разрешения правообладателей.

Автор книги не несёт ответственности за возможный вред от использования информации, составляющей содержание книги и приложений.

Copyright 2012-2013 *Валерий Рубанцев*

Лилия Рубанцева

BEGIN

Эта книга адресована тем читателям, которые имеют небольшой опыт программирования. В первую очередь – школьникам и студентам, умеющим обращаться с компьютером. Содержание книги не требует знания каких-либо других языков программирования, а внимательное отношение к многочисленным примерам поможет читателю быстро освоить самые необходимые типы данных, операторы, функции и другие элементы языка программирования *Delphi*.

Книга состоит из *трёх* логически и дидактически связанных частей. В **первой части** ясно и доступно излагаются все необходимые конструкции языка *Delphi*. Представленный на **уроках** теоретический материал иллюстрируется многочисленными примерами, а потом дополнительно закрепляется на **семинарах**, где все программы подробно обсуждаются и комментируются, что облегчает понимание заложенных в них алгоритмов и приёмов программирования. Отличительной особенностью книги является то, что читатели уже с первых уроков будут создавать настоящие программы и смогут решать собственные задачи.

На уроках и семинарах детально рассматривается работа с текстовыми файлами, строками, массивами, множествами, записями, классами, встроенными процедурами и функциями. Также в первой части книги приводятся основные приёмы работы в среде программирования *Delphi XE3*.

Изучение материала **второй** и **третьей** частей книги позволит расширить и углубить знания и умения по языку программирования *Delphi*, приобщит к разработке более сложных алгоритмов и написанию полнофункциональных программ. Здесь подробно рассматриваются элементы интерфейса (формы, канва, сетки, кнопки) и конструирование из них графического интерфейса программ. Все описания сопровождаются многочисленными примерами - как этюдами, так и *полноценными* игровыми программами: *Тест космонавта, Mind Game, Tetris, Флип-Флоп, XorGame, Logos, Японский кросворд, Цветные линии*, *Жизнь* и другими.

Почти все проекты сопровождаются исчерпывающим *анализом* игр и головоломок. Поиск *выигрышных стратегий* и разработка *эффективных алгоритмов* - вот главное содержание и достоинство представленного здесь

материала, поскольку в большинстве книг по программированию читателям сразу же предъявляется *готовое* решение проблемы, из которого невозможно (или, по крайней мере, весьма затруднительно) понять, откуда у него ноги растут и где собака зарыта.

С моей точки зрения, гораздо полезнее объяснить именно процесс *предварительных* исследований проблемы, чем описать реализацию полученного решения на каком-либо языке программирования, так как это исключительно дело техники и обычно не вызывает затруднений даже у начинающих программистов.

По ходу проработки материала книги читатели научатся использовать в программах многие компоненты *Delphi*, работать с файлами, графикой, звуком и музыкой, создавать такие элементы любых современных игр, как заставки, таблицы рекордов, справочную систему, а – самое главное – находить эффективные алгоритмы для своих программ.

Валерий Рубанцев

Обозначения, принятые в книге:



Дополнение, примечание



Предложение, ненавязчивое требование



Предупреждение



Задание для *самостоятельного* решения



Папка с *исходным кодом* приложения

Все исходные коды находятся в папках *_Source2006 (Turbo Delphi)* и *_Source (Delphi XE3)* .

Официальный сайт книги: www.rvGames.de

Здесь вы можете скачать и другие книги автора.

Оглавление

Большой самоучитель Delphi XE3 2
BEGIN 4
Оглавление 7
Delphi forever 13

Часть I. Уроки *Delphi* 15

Урок 1. Паскальная история 17
Turbo Pascal 18
Object Pascal 19
Delphi 20
Бесплатные версии Object Pascal 22
Семинар 1. Экскурсия по *Delphi*, или ОКОНчательный диагноз 23
Главное окно 24
Окно Менеджера проекта (Project Manager) 27
Окно Структуры проекта (Structure) 27
Рабочее окно 28
Страница приветствия (Стартовая страница, Welcome Page) 28
Окно Конструктора формы 30
Окно Редактора кода (Code Editor) 31
Окно Инспектора объектов (Object Inspector) 32
Окно/Палитра компонентов (Tool Palette) 32

Урок 2. Грамматика *Delphi* 46
Числа 47
Выражения 47
Идентификаторы 48
Комментарии 53
Программы и алгоритмы 55
Семинар 2. С чего начать?, или Проект, откройся! 57
Как сохранить файлы проекта на диске 60
Как загрузить проект с диска 64
Как закрыть проект 66

Урок 3. Переменные и константы 68
Константы 68
Именованные константы 69
Переменные 71
Глобальные и локальные переменные и константы 73
Типизированные константы 74
Как работать с компонентами 75
Семинар 3. «Здравствуй, мир!», или Первое приложение… усилий 78
Самостоятельная работа 85

Урок 4. Процедуры и функции, или Разделяй и властвуй 86
Объявление процедуры 87
Объявление функции 88
Описание процедуры 88

Описание функции 89
Вызов процедур и функций 90
Локальные и глобальные подпрограммы 91
Формальные и фактические параметры 91
Общие приёмы работы с окнами/панелями 94
Семинар 4. Контрольное взвешивание, или Веское приложение 97
Последний штрих 103

Урок 5. Рекурсия, или Сказочка про белого бычка 107
Как не заблудиться в коде, или Секреты навигации 111
Семинар 5. Всё познаётся в сравнении, или Вычисляем НОК и НОД 116
Наибольший общий делитель 116
Наименьшее общее кратное 120

Урок 6. Простые типы данных 121
Целые типы 121
Вещественные типы (Действительные типы) 122
Перечисляемый тип 124
Логический тип 125
Интервальный тип (тип-диапазон, ограниченный тип) 125
Символьные типы 126
Функции и процедуры для порядковых типов данных 127
Как добавить новый модуль или новую форму к проекту? 128
Семинар 6. Форма-заставка 133
Искусственное дыхание, или Оживляем форму-заставку 140
Бледная личность, или Уходящая натура 142

Урок 7. Операции и выражения 146
Операции 146
Выражения 153
Семинар 7. Одно- и многооконные приложения, или Сколько кожи на человеке 155
Однодокументный интерфейс 155
Самостоятельная работа 160
Многодокументный интерфейс 161
ЧВС «Фенечки, или "Остероиды" 163

Урок 8. Циклюем циклы 178
Оператор *FOR* 178
Оператор *WHILE* 182
Оператор *REPEAT* 185
Вложенные циклы 186
Досрочный выход из цикла 188
Выводы 193
Семинар 8. Циклотрон, или Истории Древнего мира 194
Римские числа 194
Египетские пирамиды 197

Урок 9. Массивы, или Как разложить всё по полочкам 203
Одномерные массивы символов 206
Типы индексов 206
Операции с массивами 208
Многомерные массивы 210

Семинар 9. Латинские квадраты, или Квадратно-гнездовой посев чисел 213
Самостоятельная работа 216

Урок 10. Строки, или Жестокие игры со словами 217
Короткие строки 217
Длинные *ANSI*-строки 221
Длинные строки *Юникода* 223
Другие типы строк 223
Мешанина из строк разных типов 224
Простые операции со строками 225
Семинар 10. Игра *Виселица*, или *Hangman* 227
Происхождение игры 228
Правила игры 228
Варианты игры 231
Пример игры 233
Стратегии игры 235
Программа, помогающая играть в *Виселицу*, - *HangmanHelper* 239
Самостоятельная работа 253
Дополнительный материал 253

Урок 11. Текстовые файлы, или FILEгранная работа со строками 259
Пифагоровы штаны 265
Функции для работы с файлами 266
Компоненты для работы с файлами 267
Семинар 11. Фракционирование, или Слова мелкого и крупного помола 268

Урок 12. Процедуры и функции для работы со строками 273
Семинар 12. Палиндромы, или Арабский взгляд на русские слова 279
Палиндромная программа 280
Самостоятельная работа 283

Урок 13. Множества 285
Операции над множествами 287
Операторы отношения 287
Сколько элементов в множестве? 289
Приоритет выполнения операций 289
Диаграммы Венна 289
Семинар 13. Разнобуквица, или SETка для слов 294
Супернаборщик 299
Самостоятельная работа 302
Наборщик 303
Самостоятельная работа 306
Анаграммы 307
Самостоятельная работа 309

Урок 14. Условный оператор *IF* 310
Семинар 14. Игра *Угадай число*, или Бинарный поиск 313
Правила игры 313
Стратегия игры 313
Программа для игры в *100* 314

Урок 15. Оператор выбора CASE, или *Выбирай, но осторожно. Но выбирай!* 316

Семинар 15. Лототрон, или Как помочь старушкам 319
Самостоятельная работа 323

Урок 16. Записи, или Рекорды в программировании 324
Оператор над записями *With* 328
Семинар 16. 21, или Игра в карты по-научному 330
Ближе к коду! 332
Подведём итоги 341
Самостоятельная работа 341

Урок 17. Трёхголовое ООП, или КЛАССный руководитель 342
Классы и объекты 343
Поля, свойства и методы 344
Конструктор и деструктор 346
Принципы ООП 347
Области видимости и «доступность» членов класса 355
Выводы 355
Семинар 17. Игра в КЛАССики, или Опять бОЧКОвое ЛОТО 357
Самостоятельная работа 373
ООПять лото! 374

Часть II. Практикум 379

Занятие 1. Неформальные формы 381
Шейпинг-класс, или Форменное безобразие 381
Колоритная фигура, или Долой серость! 425
Проба пера, или *Mind Game* - гимнастика для ума 434

Занятие 2. *Тетрис*, или Великолепная семёрка 472
Новый Пигмалион, или Картинная форма 473
Умелые руки, или Кнопки – своими руками 479
Тетрис *a la Delphi* 485
Хитрая механика, или Алгоритм – душа программы 501
Родословная тетриса 510
Художественные п(р)омыслы 512

Занятие 3. Викторина, или *Все флаги в гости будут к нам* 516

Занятие 4. Тест космонавта, или Форма как ракетоноситель 531
Стресс как порча жизни 531

Занятие 5. ДОСКОнально о сетке, или Занимательное полеводство 550
Поле в клеточку, или Компонент *TDrawGrid* 550
Игры-оборотни 560
XorGame, или Как обнулить матрицу 560
Другие игры-оборотни 658
Чёт и нечет 660
Ножки вверх!, или Головоломка с бокалами 663

Занятие 6. Живописная основа, или класс TCanvas 676
Пуантилизм, или Ставим точки 676

Геометрические фантазии, или Методы *TCanvas* 684
Заливная рыба, или метод *FloodFill* 707
Чините перья, или класс *TPen* 711
Наше вам с кисточкой, или класс *TBrush* 729
Траектории мысли, или Paths 740
Картинки с выставки, или Художественные методы канвы 747
Великий комбинатор, или Свойство *CopyMode* 764
Наш аниматограф 772
Движущиеся линии, или «Классная» программа 774
AVIация, или Клепаем клипы 795

Занятие 7. Компот из компонентов, или «А мне всегда чего-то не хватает…» 796
Просто кнопка, или *Как пройти в библиотеку?* 796
Копай с умом!, или Пятнадцать человек на сундук мертвеца 802

Часть III. Факультатив 842

Факультатив 1. Фейсом к интерфейсу, или *Дельфи, открой личико!* 844
Горячие клавиши ИСР 886

Факультатив 2. Практические занятия, или Анатомия приложения 890
Файл проекта 892
Файлы формы 894
Файлы модулей 898
Файлы ресурсов 898
Параметры проекта 910
Как нам обустроить *Delphi* 913

Факультатив 3. Формы и окна 923
Элементы стандартного окна 923
Свойства формы 926
События формы 934
Методы формы 937

Факультатив 4. Японский кроссворд, или Восхождение на Фудзияму мысли 941
Просветление, или Что придумали японцы 941
Правила хорошего моветона 943
Решительный момент, или Берём быка за бока 948
Кибернетическое убийство времени, или Компьютер-поводырь 954
Решительный штурм, или Как расколоть японские кроссворды 957
Занимательная механика, или За кулисами формы 963
Каждому сверчку по шестку, или Нумеруем клетки 978
Мотаем на ус, или Наш задачник 986
Делайте с нами, или Тонкие штучки на компьютере 1002
Компьютерная педагогика, или Пускай работает Иван 1008
Мандрагора, или Зри в корень! 1020
Магическая кнопка, или Первые аплодисменты 1027
Мастерская Самоделкина, или Наши весёлые картинки 1035
«Шлите апельсины бочками», или «Ждите ответа!» 1039
«А напоследок я скажу…» 1040
Генеалогическое древо, или Японская родня 1043

Факультатив 5. Игра-головоломка *Logos* 1046
Правила игры 1046
Программируем от достигнутого 1048
Всплывающее меню 1061
«Отзывчивые» кнопки 1063
Веб спешит на помощь 1078
Управляющие кнопки 1081
Всплывающее меню 1081
Шиворот-навыворот и задом наперёд 1083
Учимся лениться 1090
«С любимыми не расставайтесь…» 1095

Факультатив 6. Цветные линии, или Критическая масса мыслей 1096
Мир *Цветных линий* 1096
Нить Ариадны, или Как выбраться из лабиринта 1108
Большая стройка, или «Линейное» программирование 1116
«Требую продолжения банкета!», или Где начало того конца, которым оканчивается начало? 1165

Факультатив 7. «Что наша «Жизнь»? – Игра!», или Клеточный автомат Джона Конвея 1186
Законы *Жизни* 1186
Как построить вселенную, или Даём жизни! 1187

Факультатив 8. Бонусы, или Хитрая головоломка – хитори 1221

Факультатив 9. *FireMonkey*, или Стильный интерфейс 1225
Приложения *FireMonkey* 1227
Компоненты *FireMonkey* 1231
Привет, *FireMonkey*! 1237

Happy **END.**, или Карлсоны всегда возвращаются 1267

Литература 1268

Delphi forever

Старый друг лучше новых двух.

Русская пословица

Alte Liebe rostet nicht.

Немецкая пословица



Все примеры и программы для этой книги написаны в *Turbo Delphi 2006* и в *Delphi XE3*. Это значит, что они без всяких изменений могут быть загружены в любую версию *Delphi*, начиная с *Turbo Delphi 2006* и заканчивая самой последней - *Delphi XE3*. Если же вы хотите использовать только код, а не готовые файлы приложения, то большинство программ будет прекрасно работать во всех версиях, начиная с четвёртой (код придётся копировать вручную). Более того, вы можете бесплатно скачать программу *Lazarus* и самостоятельно создавать интерфейс и набирать код. Это долго, но гораздо более полезно, чем питаться «консервами». Таким образом, проблем со средой разработки программ и в принципе быть не может. Что касается самого языка программирования *Delphi*, то ничего мощнее и удобнее вы не найдёте. С его помощью вы можете писать любые программы – как для себя, так и для семьи, школы и даже коммерческие трёхмерные игры на основе *DirectX 9*. Здесь результат больше зависит от программиста, чем от языка программирования, поскольку все современные языки заведомо мощнее, чем вам может потребоваться.

Если вы хотите получить от программирования не только пользу, но и удовольствие, выбирайте *Delphi*. Здесь вполне уместно вставить рекламный довод: «При всём богатстве выбора другой альтернативы нет». Из всех современных универсальных языков программирования высокого уровня он наиболее «гуманный», а потому наименее машинный. Тут с ним не могут сравниться ни *C/C++,* ни *C#*, ни бейсик, ни другие, более экзотические языки. Поэтому программист может полностью сосредоточиться на логике алгоритма, а не на его подгонке к особенностям процессора или причудливым конструкциям самого языка программирования. Кроме относительной простоты, у *Delphi* есть и другие достоинства. Он универсальный, то есть позволяет решать практически любые задачи: от квадратных

уравнений до создания современных трёхмерных игр. Скомпилированные программы очень небольшие по размеру и очень быстрые, при этом они не требуют для своей работы установки дополнительных программ и библиотек. *Delphi* помогает очень быстро и удобно разрабатывать приложения с очень красивым графическим интерфейсом.

Часть I. Уроки *Delphi*

Железный конь идёт на смену крестьянской лошадке!

Из эпохальной речи
товарища Бендера на митинге

Нельзя объять необъятного.

Козьма Прутков

Первая часть книги целиком посвящена изучению основ языка *Delphi*. В ней вы найдёте все необходимые сведения о таких объектах языка, как идентификаторы, константы и переменные, простые и структурные типы данных, процедуры и функции, операторы и операции. Также вы познакомитесь с классами и объектно-ориентированным программированием (ООП), без которого невозможно себе представить разработку современных программ.

На **уроках** вы будете изучать язык программирования *Delphi* и приёмы работы в среде программирования *Delphi XE3*. Здесь вы напишете короткие программы-примеры, на которых легче усвоить пройденный материал.

После каждого урока добро пожаловать на **семинар**. На каждом из них вы напишете одну программу (иногда и больше), которая подробно обсуждается и комментируется. Именно на семинарах вы сможете укрепить свои знания, полученные на уроках.

Самый лучший способ научиться чему-либо – это практика, поэтому исходный текст программ (особенно небольших) полезнее набирать самостоятельно с клавиатуры, а не загружать уже готовые файлы с диска. Так вы не только быстрее освоите *Редактор кода*, но и научитесь находить и исправлять собственные ошибки. Но даже когда текст набран верно и программа работает безукоризненно, пробуйте изменить её и – анализируйте результаты: ведь наверняка ваша цель не в том, чтобы повторять чужие программы, а в том, чтобы писать свои собственные.

Недаром эпиграфом к первой части книги выбран известный афоризм Козьмы Пруткова – в одной книге невозможно рассказать ни обо всех возможностях языка *Delphi*, ни проиллюстрировать примерами все особенности применения конструкций языка. Однако примеров (простых и не очень) и в этой части книги очень много, а во второй и третьей частях вы найдёте и большие, вполне законченные программы, изучая код которых вы сможете писать собственные, очень сложные программы.

Урок 1. Паскальная история

Язык до Киева доведёт.

Из путеводителя по Украине

Первый **паскаль** (тогда ещё *Паскаль*) появился в 1970 году. Его автор – профессор Швейцарского института информатики Никлаус Вирт – назвал его в честь великого французского учёного 17-го века Блеза Паскаля, который, кроме прочего, изобрёл автоматическое устройство для суммирования чисел.

Никлаус Вирт «причастен» к созданию и других языков программирования – *Эйлер*, *Модула*, *Модула-2*, *Оберон*.

Хотя паскаль был предназначен в первую очередь для обучения программированию, он довольно быстро стал популярным и среди программистов, поскольку позволял выразить алгоритмы в понятной и удобной форме. К другим достоинствам паскаля можно отнести:

- Полную поддержку структурного программирования и структур данных.

- Минимально необходимый, но достаточный набор типов: простые переменные, массивы, файлы, множества, записи, указатели.

- Он позволяет создавать из встроенных новые типы данных. Обоснованно считается, что правильный выбор стандартных типов данных или удачная разработка новых типов данных не менее важны при написании программы, чем составление хорошего алгоритма. И паскаль предоставляет все средства для создания адекватных структур данных, что даёт программисту возможность формулировать более простые и эффективные алгоритмы.

- Надёжность программ, хотя это и заставляет программиста писать немало «лишнего» кода. Например, все переменные, константы и другие объекты программы должны быть описаны в соответствующих разделах модуля или подпрограммы, хотя в языке *C* объявлять новые переменные разрешается даже внутри операторов. Нет в паскале и «хитроумных» конструкций, которыми изобилует язык *C*. Здесь не приветствуются «трюки», в программе на паскале код должен быть понятен и недвусмыслен.

- Высокий уровень языка, что увеличивает производительность работы программиста, так как ему не приходится заниматься описанием мелких операций и он может полностью сосредоточиться на алгоритме.

- Язык довольно лёгок в изучении и удобен в использовании как для профессионалов, так и для любителей.

- Это универсальный язык, который позволяет решать прикладные задачи и разрабатывать системные программы.

Итак, Никлаус Вирт наделил свой язык программирования только самыми необходимыми возможностями, но программистов много и они решают разные задачи, поэтому стали появляться расширения языка паскаль, которые затрудняли переносимость программ с одного компьютера на другой. Поэтому в 1982 году был «законодательно» установлен стандарт языка паскаль ISO7180.

Turbo Pascal

Адам познал Еву, жену свою; и она родила Каина...;
И познал Каин жену свою, и она родила Еноха;
У Еноха родился Ирад;
Ирад родил Мехиаеля;
Мехиаель родил Мафусала;
Мафусал родил Ламеха.

Библейское

Однако язык продолжал развиваться уже независимо от воли и желания самого создателя.

В 1983 году на рынке появился **Turbo Pascal 1.0** американской фирмы *Borland International* для CP/M и компьютеров 8086.

В 1985 году вышла новая версия - **Turbo Pascal 3.0**. Он был совмещён с быстрым компилятором и редактором кода, что обеспечивало пользователя всем необходимым для быстрой разработки программ под операционную среду *MS-DOS*. Именно эти качества и позволили паскалю «проникнуть» в школы и институты, где его изучали как первый язык программирования.

С выходом версии **Turbo Pascal 5.5** пакет «обзавёлся» встроенным *отладчиком*, который ещё больше ускорил разработку программ и привлёк на свою сторону немало профессиональных программистов. Был «дополнен» и язык паскаль (настолько, что *Turbo Pascal* считается не расширением, а другим языком программирования, основанной на паскале), например, он умел работать с графикой и большими программами, разбитыми на модули, впервые появилась поддержка объектно-ориентированного программирования (ООП), к которому сам Вирт относился с большой долей иронии.

В 1992 году была выпущена последняя версия пакета - **Turbo Pascal 7.0**, которая до сих пор используется в школах для обучения программированию. Он имеет многооконный редактор, обширную систему помощи, текстовый редактор с выделением элементов языка разным цветом, позволяет включать в код вставки на ассемблере, что делает программу очень быстрой. В этой версии были полностью реализованы возможности ООП, появился типизированный адресный оператор @, открытые массивы и строки. Всё вместе это позволило программистам эффективно решать новые задачи.

Turbo Pascal 7.0 был предназначен для любительского программирования под MS-DOS, а вышедшая одновременно с ней версия **Borland Pascal 7.0** – для профессионалов, она позволяла писать программы для работы в ОС *Window 3.1*.

Object Pascal

Седьмая версия оказалась последней для языка *Turbo Pascal*. После него фирма *Borland* начала выпускать интегрированную среду быстрой разработки программ **Delphi**, основой которой стало объектно-ориентированное расширение языка *Turbo Pascal*, получившее название **Object Pascal**.

Первая версия *Delphi* позволяла разрабатывать программы под старую 16-разрядную платформу *Windows*. Начиная со *второй* версии, программы компилируются под 32-разрядную ОС *Windows*.

Object Pascal в сравнении с языком *Turbo Pascal* получил новые возможности:

- Появились *интерфейсы*, которые напоминают классы, они были введены для поддержки *COM* технологии фирмы *Microsoft*.

- Добавлена *перегрузка* процедур и функций с помощью ключевого слова *overload*, перегрузка операторов, динамические массивы, ключевое слово *object* было заменено словом *class*, чтобы устранить двусмысленность между описанием объекта и самим объектом. В *Turbo Pascal* были объекты, но не было классов, описывающих эти объекты.

- *Визуальное* программирование превратило разработку интерфейса пользователя в простое и весёлое занятие: теперь все элементы интерфейса можно собирать из компонентов и сразу же оценивать результаты. Программисты, знакомые с библиотекой *Turbo Vision* для *Turbo Pascal*, предназначенной для этих же целей, легко поймут разницу!

Delphi

Из двух языков программирования выбирай Delphi.

Новая русская пословица

Если тебе скажут, что C# лучше Delphi,
не верь ушам своим!

Программист Козьма Прутков

Начиная с *седьмой* версии, фирма *Borland* стала называть язык программирования *Object Pascal* точно так же, как и саму ИСР – **Delphi**. Кстати говоря, именно седьмая версия очень популярна у программистов и сейчас - следующие всё дальше и дальше уходят от полюбившегося всем паскаля.

ИСР, а затем и язык программирования *Delphi* были названы в честь древнегреческого города Дельфы, где находился храм Аполлона и дельфийский оракул (вот он-то и повлиял на выбор названия). Этот храм украшал заставку к четвёртой версии *Delphi* (Рис. У1.1).

Версия *8* была способна компилировать программы исключительно для платформы *.NET*. Оказалась крайне неудачной и практически не использовалась.

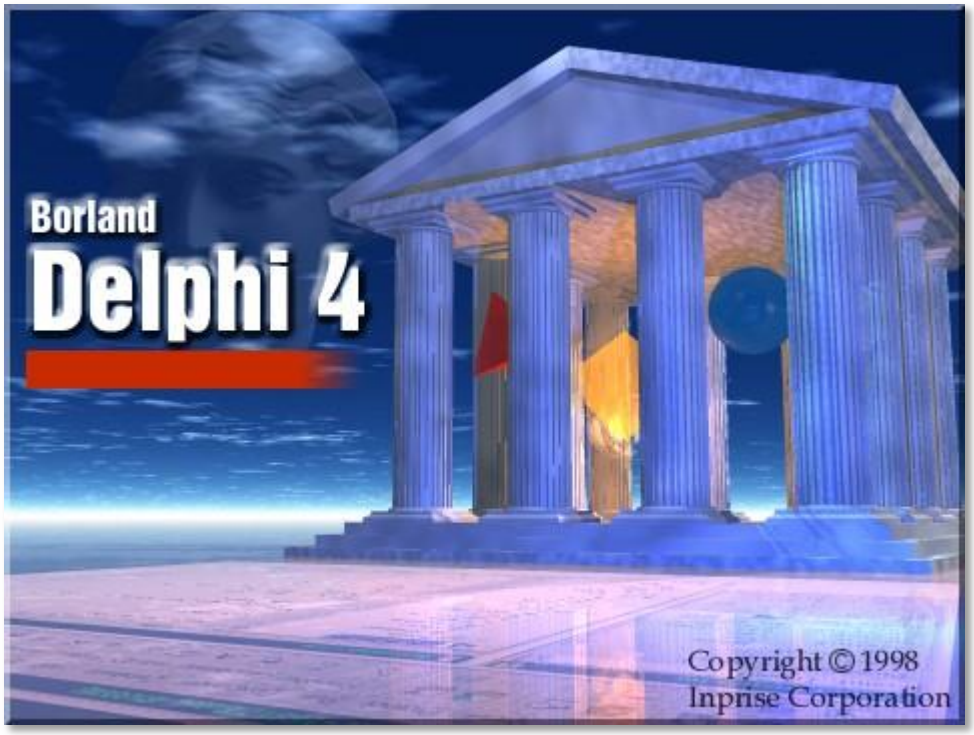


Рис. У1.1. Храм Аполлона на заставке *Delphi 4*

Следующие версии *Delphi* обозначались годом выхода (в коммерческих целях – не текущим, а следующим) и могли создавать приложения как для *Windows*, так и для платформы *.NET*: **Borland Delphi 2005** (по счёту это 9-я версия), **Borland Delphi 2006** (10-я версия, компания *CodeGear*), наряду с ней была также выпущена «облегченная» версия - **Turbo Delphi 2006**. Затем последовали **Delphi 2009** *и* **Delphi 2010** (*Embarcadero RAD Studio 2010*), после чего компания *Embarcadero* запустила новую линейку *Delphi – XE*.

Delphi XE, или *Delphi 2011* была выпущена в 2010 году, **Delphi XE2** - в 2011 и последняя на сегодняшний день версия **Delphi XE3** - в 2012. Она поддерживает 64-битную ОС *Windows*, а также ОС *Mac OS* и *iOS*, так что теперь на *Delphi* можно разрабатывать *многоплатформенные* приложения.

Delphi и на сегодняшний день - это один из лучших пакетов для быстрой разработки программ. Он содержит все основные элементы современных языков программирования: модули, классы и обобщенные классы, перегрузку операций, интерфейсы, исключения, сборку мусора, а также некоторые средства параллельного программирования. Транслятор с языка *Delphi* на машинный является *компилятором*, который и сам работает очень быстро и даёт очень быстрые приложения.

Поскольку *Delphi* - наследник славного паскаля, исходный код программ, написанных на нём, очень лёгок для понимания, в чём вы скоро сами сможете убедиться!

Бесплатные версии Object Pascal

Все коммерческие версии *Delphi*, понятное дело, стоят недёшево и вряд ли доступны начинающим программистам, однако не перевелись ещё робин-гуды: вы можете свободно скачать и пользоваться компиляторами языка *Object Pascal*, например, **Free Pascal** (*Free Pascal Compiler*), **TopSpeed Pascal** (версия языка *Turbo Pascal*), **TMT Pascal**, ***Virtual Pascal*** и **GNU Pascal**. В первую очередь можно рекомендовать **Lazarus** – аналог среды разработки программ *Delphi*, который в качестве компилятора использует *Free Pascal* (Рис. У1.2).

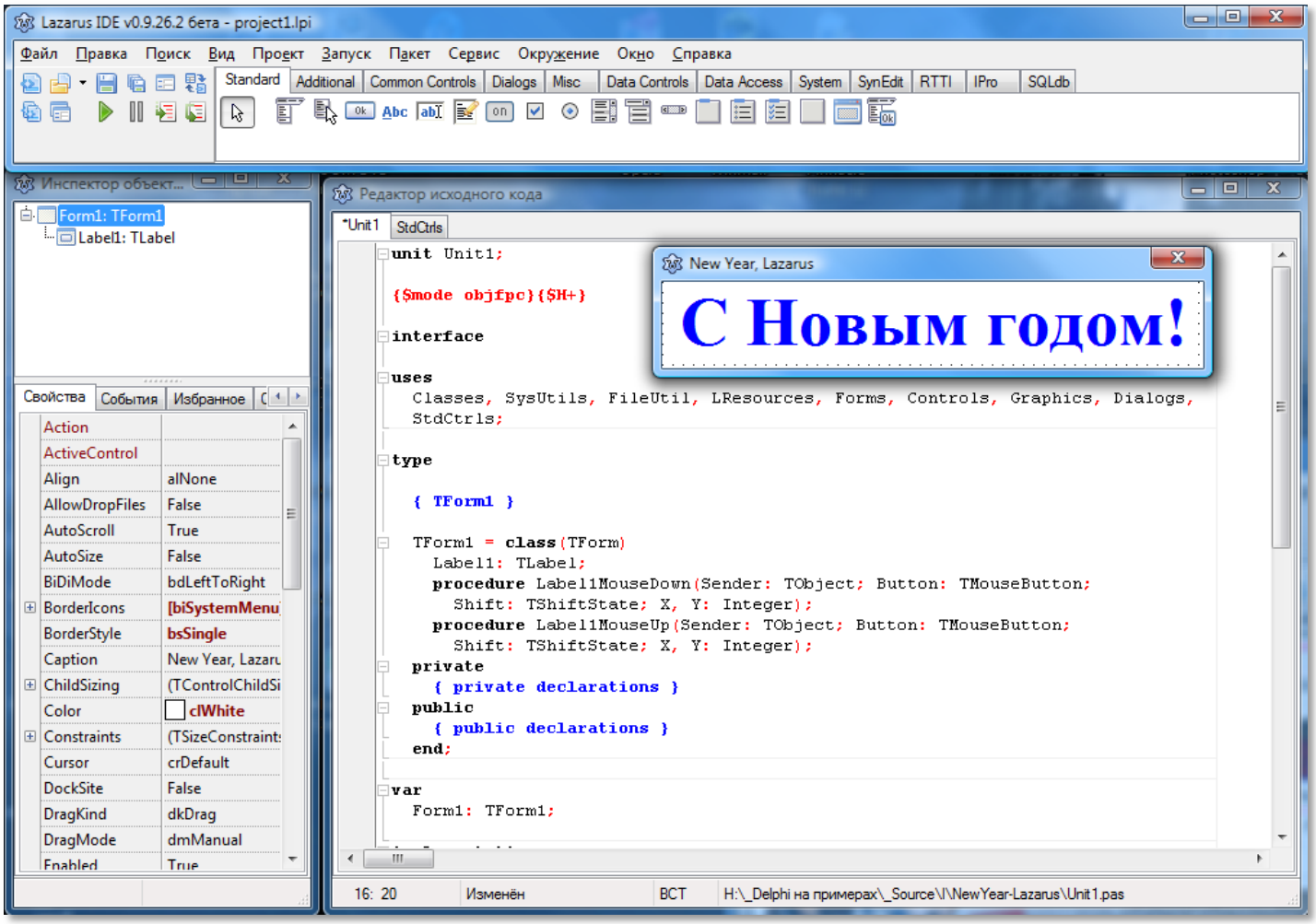


Рис. У1.2. *Lazarus* поздравляет вас с праздником!

С его помощью вы сможете писать программы не только для *Windows*, но и для других операционных систем - *GNU/Linux, Mac OS X, FreeBSD, WinCE*.

Семинар 1. Экскурсия по *Delphi*, или ОКОНчательный диагноз

Коль выгонят в окно, так я влечу в другое.

А.Крылов, басня *Муха и Пчела*

Выгляни в окошко - дам тебе горошка!

Сказочная завлекалочка

На первом семинаре мы бегло рассмотрим основные элементы интерфейса *Delphi XE3*. Более подробно вы познакомитесь с ними на Факультативе 1.

Интегрированная среда разработки программ (ИСР) *Delphi* представляет собой настоящий «окономастас»: весь рабочий стол - в окнах. Сразу в них не разобраться, поэтому мы совершим короткую экскурсию по *Delphi*, нескромно заглядывая во все открытые окна.

Сразу после загрузки *Delphi XE3* вы увидите такую картину:

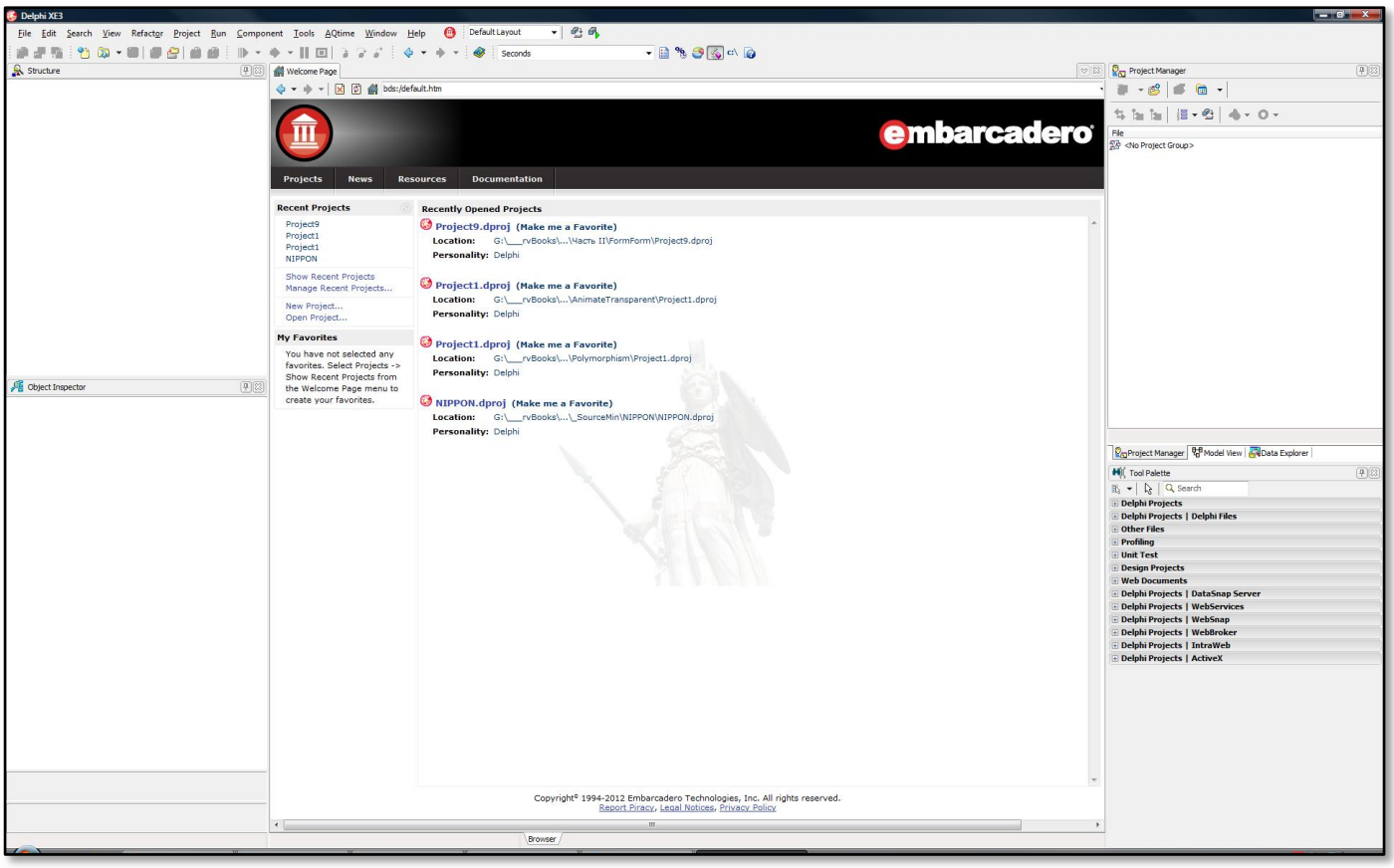


Рис. С1.1. Общий вид ИСР *Delphi XE3*

Над всеми другими окнами «возвышается» *Главное окно*. На нём размещаются *Главное меню* программы и *Панели инструментов* с кнопками быстрого доступа.

В левой и правой частях экрана располагаются различные панели, а центральную часть занимает *Рабочее окно* ИСР.

Главное окно

Главное окно состоит из нескольких частей - *Главного меню* и нескольких *Панелей инструментов*.

Главное меню (Main Menu)

Находится сразу под заголовком *Главного окна* и содержит несколько подменю с множеством команд.

File
- загрузка и сохранение файлов.

- **New:**
  VCL Forms Application – Delphi – начинает новый проект со стандартными элементами управления.

  FireMonkey Desktop Application – Delphi – начинает новый проект с элементами управления *FireMonkey*.

  VCL Form – Delphi – добавляет новую пустую стандартную форму к проекту.

  FireMonkey Form – Delphi – добавляет новую пустую форму *FireMonkey* к проекту.

  Unit – Delphi – добавляет модуль к проекту.

- **Open** – открывает стандартное диалоговое окно загрузки файлов.

- **Open Project** - загружает существующий проект *Delphi*.

- **Reopen** – открывает список с названиями последних проектов и модулей.

- **Save** – записывает на диск текущий файл (формы и модуля), находящийся в окне *Редактора кода* под текущим именем.

- **Save As** – позволяет ввести в диалоговом окне новое имя для текущего файла и сохранить его на диске.

- **Save Project As** – записывает текущий проект под новым именем.

- **Save All** – записывает все открытые файлы проекта.

- **Close** – закрывает текущий проект и все связанные с ним файлы.

- **Close All** – закрывает все открытые файлы.

Edit
– редактирование кода и компонентов на форме.

Search
– поиск и замена последовательности символов (текста) в исходном коде.

View
– работа с окнами и панелями ИСР.

Project
– работа с файлами проекта.

Run
– запуск и отладка программы в среде *Delphi*:

Help
– вызов справки помощи.

Обратите внимание на то, что самые «популярные» команды *Главного меню* имеют клавиатурные сокращения. Например, отправить в буфер обмена выделенный фрагмент кода (или компонент) можно, нажав одновременно клавиши *Ctrl* и *C*, а вставить в позицию курсора (или на форму, если

речь идёт о компоненте) - клавишами *Ctrl* и *V*. Безусловно, это гораздо удобнее, чем каждый раз бродить по просторам меню. Другая возможность повышения комфортности работы в ИСР – использовать кнопки на *Панелях инструментов*.

Панели инструментов (Toolbars)

Находятся под *Главным меню* в левой части *Главного окна* и содержат кнопки, дублирующие наиболее часто используемые команды *Главного меню*.

Standard – стандартная:



 - команда **New Items** очень часто применяется для присоединения новой формы или модуля к проекту.

 - аналог команды **File > Reopen**. Позволяет быстро открывать сохранённые на диске проекты. Нажмите стрелочку справа от папки, чтобы раскрыть список последних проектов и файлов.
 - аналог команды **File > Save** (быстрые клавиши - **Ctrl+S**).

 - аналог команды **File > Save all** (быстрые клавиши – **Shift + Ctrl+S**).

 - аналог команды **File > Open Project** (**Ctrl+F11**). Быстрое открытие сохранённого на диске проекта.

View - просмотр:



 - аналог команды **View > Unit** (**Ctrl+F12**). Выводит на экран список всех модулей приложения, в котором можно выбрать нужный для отображения в окне *Редактора кода*.

 - аналог команды **View > Form** (**Shift+F12**). Выводит на экран список всех форм приложения. Выбранная форма появляется в окне *Конструктора формы*.

 - аналог команды **View > Toggle Form/Unit** (**F12**). Переключается между окном *Редактора кода* и окном *Конструктора формы* (если в проекте несколько форм, то сначала выберите нужную предыдущей командой).

Debug – отладка:



 - аналог команды **Run > Run** (**F9**). Используется очень часто для запуска текущего проекта.

 - аналог команды **Run > Run Without Debugging** (**Shift+Ctrl+F9**). Запуск текущего проекта без отладки.

Окно Менеджера проекта (Project Manager)

Расположено в правом верхнем углу ИСР и показывает все файлы текущего проекта.

Менеджер проекта облегчает работу с файлами проекта. Например, после нажатия правой кнопки мыши на строчке *Unit1.pas* (главный модуль проекта), на экране появится контекстное меню.

В нём вы можете сохранить файл модуля на диске (команды *Save* или *Save As...*), переименовать его (*Rename*) и удалить из проекта (*Remove From Project*). Команда *Открыть* (*Open*) действует аналогично клавише *F12*, то есть открывает окно *Конструктора формы* (если курсор находится в строке *Unit1.dfm*) или окно *Редактора кода* (если в строке *Unit1.pas*).

Окно Структуры проекта (Structure)

Находится слева вверху. Оно отображает иерархическую структуру компонентов на форме (когда открыто окно *Конструктора формы*) или структуру кода программы (когда открыто окно *Редактора кода*).

Рабочее окно

Рабочее окно представляет собой панель с вкладками. Первоначально вкладка только одна – *Welcome Page*, но после открытия нового проекта или после загрузки существующего количество вкладок увеличивается:



Чтобы открыть нужное окно, достаточно просто щёлкнуть по вкладке с его именем.

В режимах конструирования формы и редактирования кода под *Рабочим окном*, в строке состояния также появляются три вкладки:



Поскольку окна *Конструктора формы* и *Редактора кода* не могут присутствовать на экране одновременно, то приходится очень часто переключаться между ними. Сделать это можно несколькими способами:

- нажмите горячую клавишу **F12**;
- нажмите кнопку быстрого доступа 

в *Главном меню*;
- нажмите соответствующую вкладку в строке состояния 

.

Страница приветствия (Стартовая страница, Welcome Page)

При первом запуске *Delphi XE3* в *Рабочее окно* загружается страница *Welcome Page* (Рис. С1.2). В её верхней части находится список последних проектов, над которыми вы работали - *Recent Projects*. Он позволяет очень удобно и быстро возвращаться к работе над приложениями.

Ниже вы найдёте кнопки *New Project*... и *Open Project*..., с помощью которых можно начать новый проект или открыть существующий, если он отсутствует в списке *Recent Projects*.

В правой части *Страницы приветствия* располагается **Список часто открываемых проектов** (*Recently Opened Projects*).

По сути, *Стартовая страница* представляет собой браузер, подобный *Internet Explorer*, поэтому непосредственно из неё вы можете перейти к чтению новостей на сайте фирмы *Embarcadero* или открыть любую другую страницу (если ваш компьютер имеет доступ к Интернету). Вот, например, как выглядит статья о языке *Delphi* на сайте *Wikipedia*, загруженная в *Рабочее окно* ИСР (Рис. С1.3).

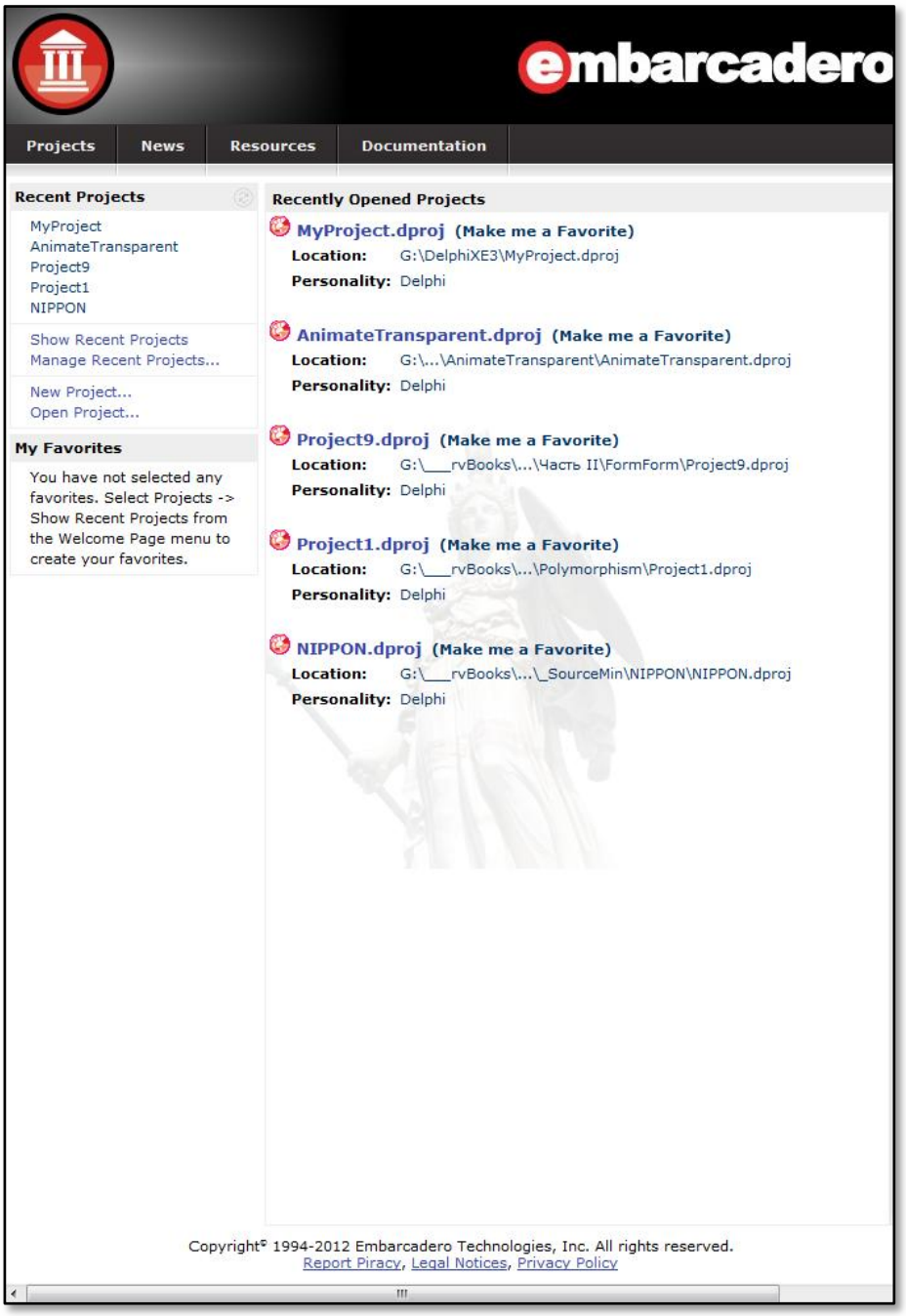


Рис. С1.2. Стартовая страница

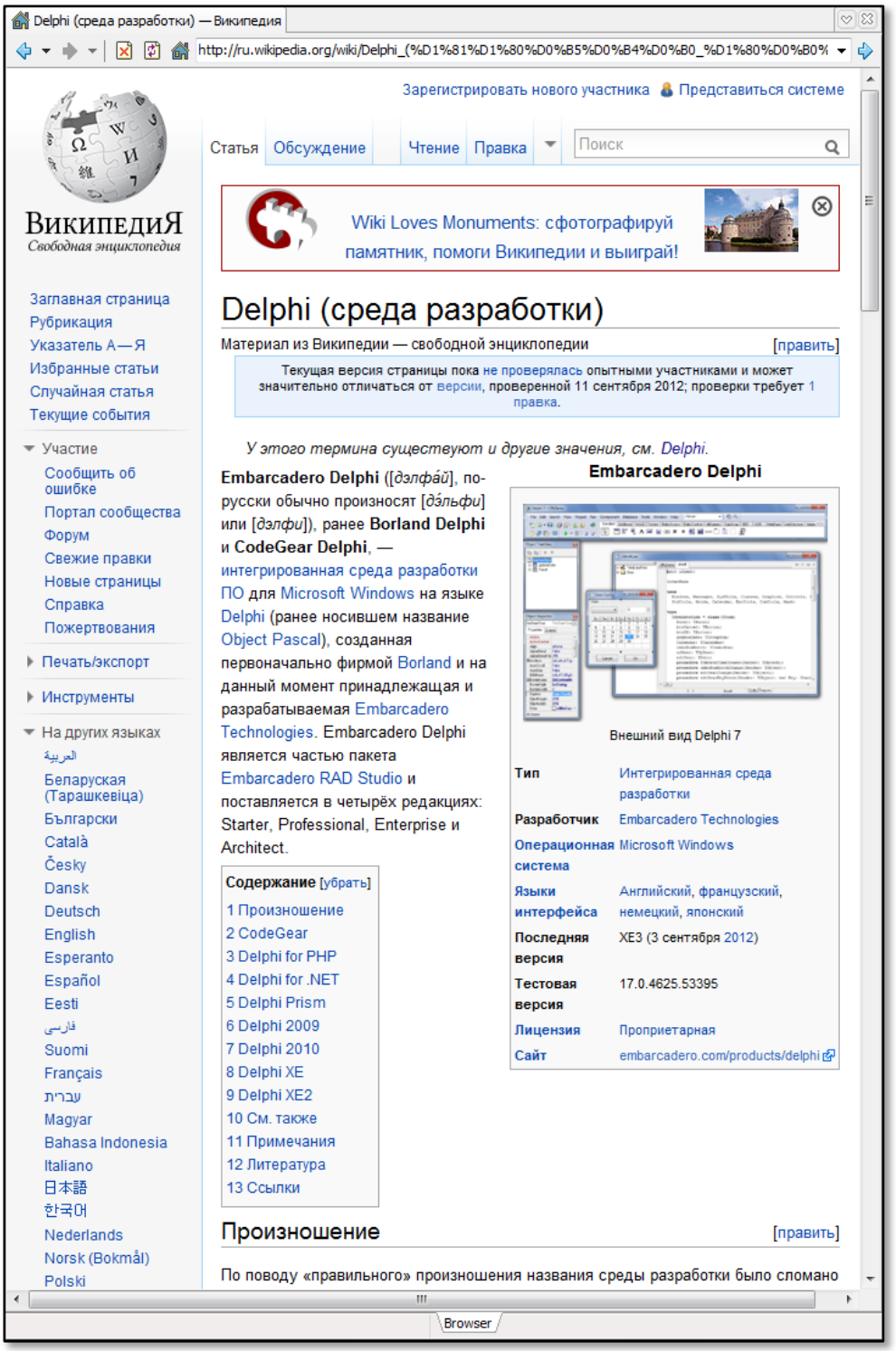


Рис. С1.3. Читаем в *Delphi* о *Delphi*!

Окно Конструктора формы

Именно в этом окне выполняется *визуальное проектирование* формы вашего приложения. Здесь вы можете задать ей необходимые размеры, расставить на ней нужные для работы программы компоненты.

Окно Редактора кода (Code Editor)

В окне **Редактора кода** вы можете создавать и редактировать код программы (Рис. С1.4).

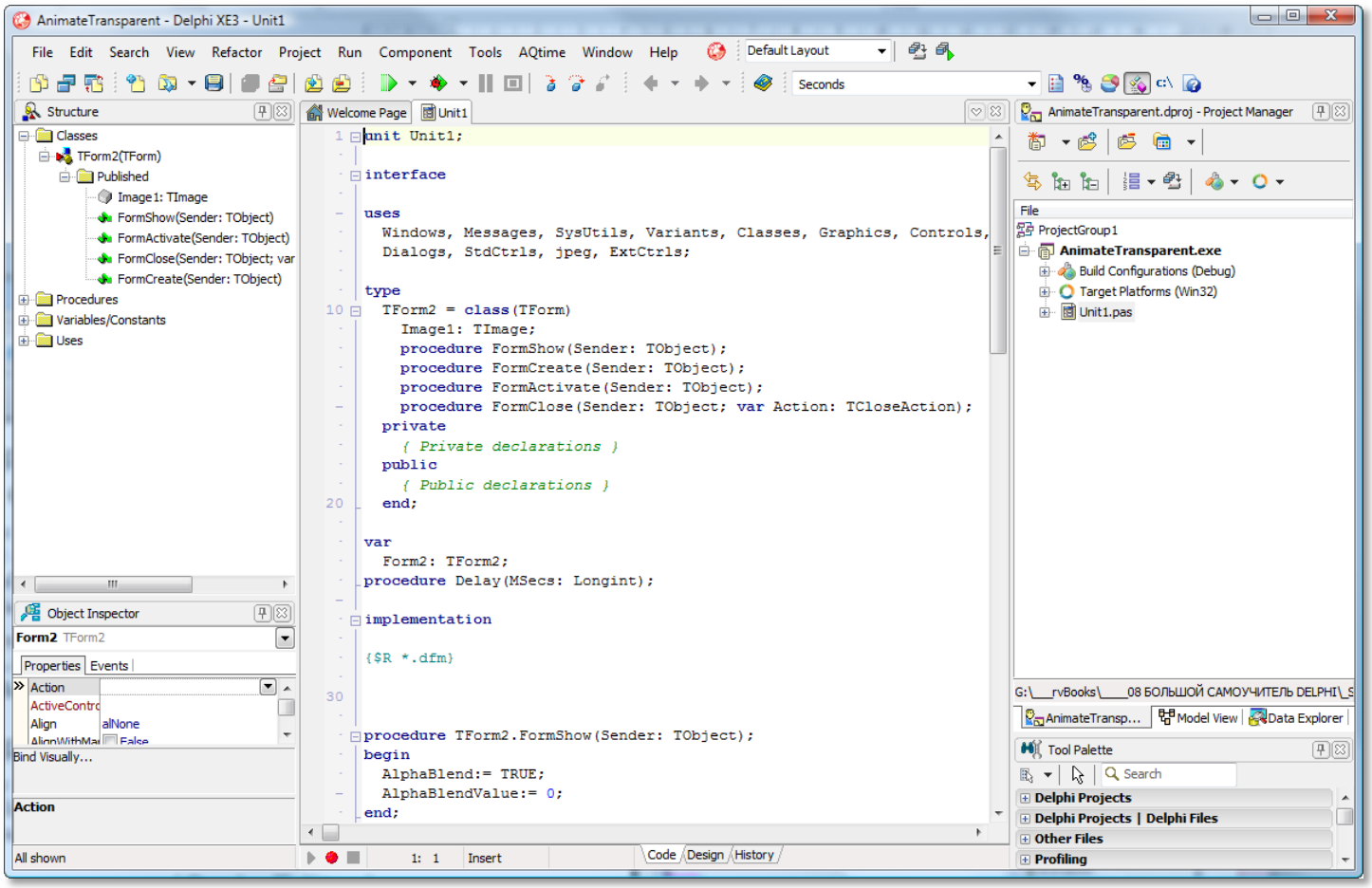


Рис. С1.4. Окно *Редактора кода*

Редактор кода представляет собой обычный текстовый редактор. Сначала в нём открыта для редактирования только одна страница – с исходным текстом (заготовкой) модуля формы *Form1* разрабатываемого приложения. Его название *Unit1.pas* находится в заголовке окна и на вкладке *Рабочего окна*.

В редактор можно загрузить и другие текстовые файлы, например, файл проекта или других модулей проекта (выбрать нужный можно, выполнив команду *Главного меню* **View > Units...** 

, нажав клавиши **Ctrl+F12** или кнопку).

Окно Инспектора объектов (Object Inspector)

Окно **Инспектора объектов** находится под панелью *Структуры проекта* и имеет две страницы (вкладки) - *Properties* и *Events*. Каждая из них разделена на две части вертикальной линией, которую можно передвигать мышкой. Чтобы открыть нужную страницу, щёлкните, как обычно, на её ярлычке.

На странице **Properties** можно задавать *свойства* компонентов, установленных на форме (а также и самой формы). В начале работы над проектом в окне *Инспектора объектов* виден только список свойств и их значения для формы. Если вы добавите на форму какой-либо компонент, то перечень его свойств появится, если выделить его (щёлкните на нём мышкой) или выбрать название этого компонента в открывающемся списке, находящемся сразу под заголовком окна.

На странице **Events** вы найдёте все *события*, на которые реагирует текущий компонент.

Окно/Палитра компонентов (Tool Palette)

Панель инструментов (*Tool Palette*) находится в правом нижнем углу ИСР, под окном *Менеджера проектов*.



В *Delphi XE3* называется *Палитрой инструментов*, но мы будем также использовать и её прежнее название – *Палитра компонентов*, - которое более точно определяет её назначение.

Сразу после загрузки ИСР вы увидите в ней список проектов, которые можно создавать в *Delphi* (Рис. С1.5, слева). Щёлкните на строчке *Delphi Projects*, чтобы развернуть список доступных приложений (Рис. С1.5, справа). Чтобы открыть список компонентов, щёлкните по строке *VCL Forms Application*. В *Панели инструментов* появится список компонентов (Рис. С1.6), а также одновременно откроется новый проект.

Впрочем, обычно сначала создают новый проект, а *Палитра инструментов* появляется самостоятельно в режиме конструирования формы. Эта панель содержит большое количество компонентов, разбитых на функциональные группы, для каждой из которых имеется отдельная вкладка (Рис. С1.6).

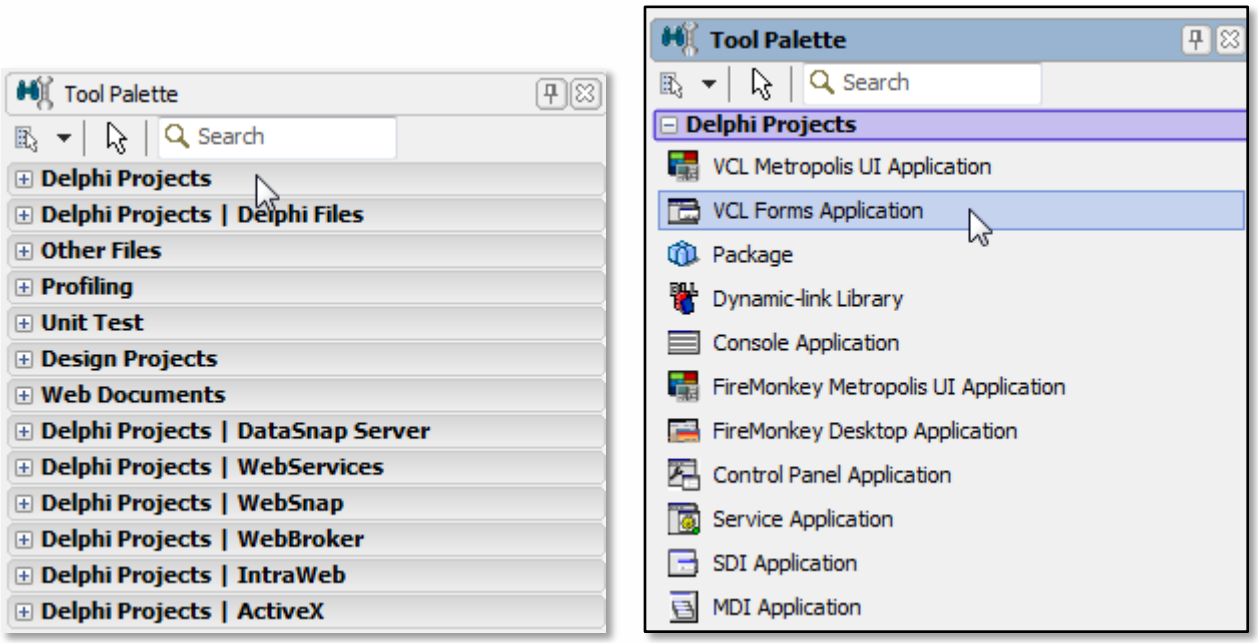


Рис. С1.5. *Палитра инструментов* после запуска *Delphi*

Чтобы выбрать нужную страницу компонентов, щёлкните по значку с плюсиком ⊞ или просто на заголовке страницы, при этом страница раскрывается, показывая список (значки и названия) всех своих компонентов (Рис. С1.7).

Повторный щелчок сворачивает страницу – очень удобно!

Если вся *Панель компонентов* на экране не помещается, то навигация по ней осуществляется с помощью полосы прокрутки, которая находится справа.

Так как страниц с компонентами очень много, иногда бывает удобнее найти нужную и быстро перейти к ней с помощью списка, который раскрывается при нажатии на значок *Show palette categories* (Рис. С1.8).



Страницы вместе с компонентами можно перемещать вверх и вниз в пределах панели, ухватившись мышкой за заголовок. При этом безразлично, раскрыта страница или свёрнута.



Рис. С1.6. Полный список компонентов

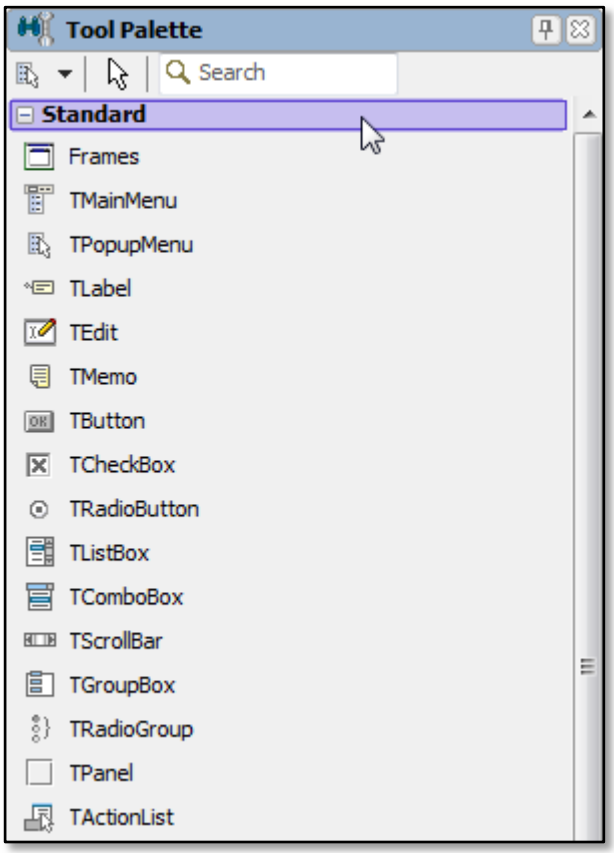


Рис. С1.7. Открытая страница *Standard*

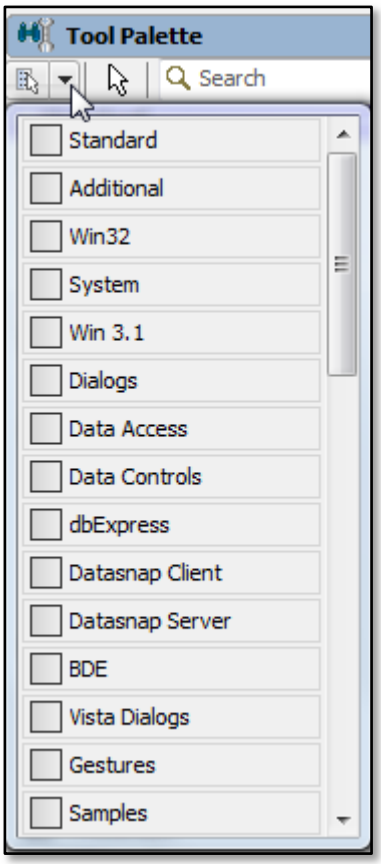


Рис. С1.8. Список компонентов по категориям

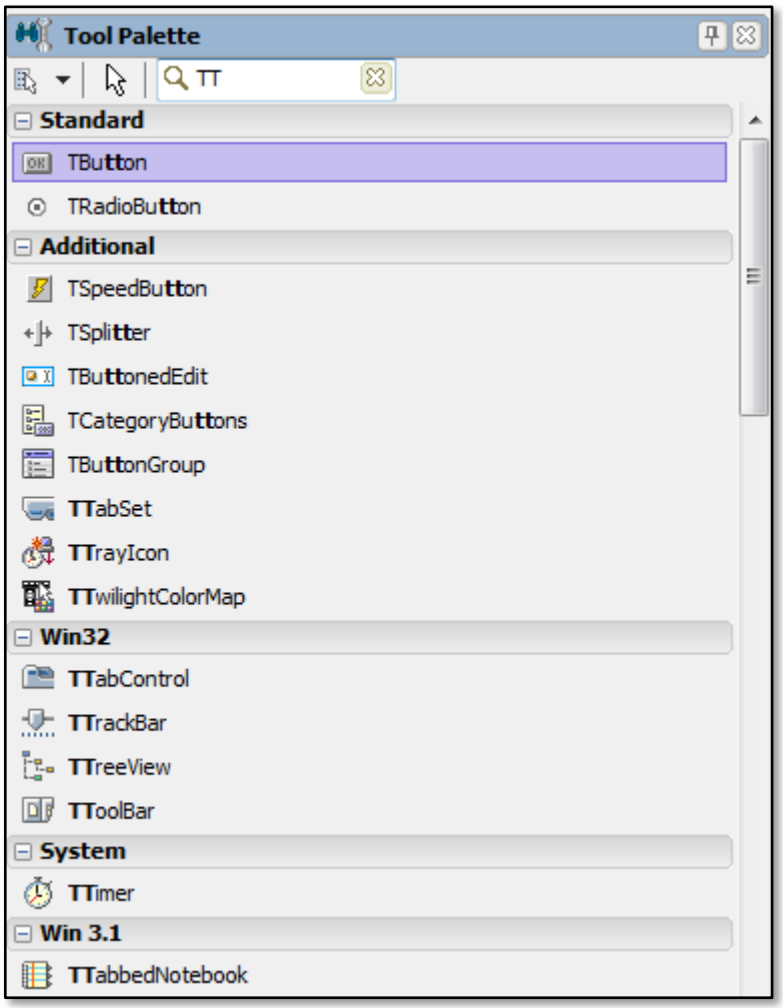


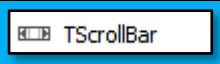
Рис. С1.9. Ищем нужный компонент

Так как компонентов огромное множество, то очень нелегко найти нужный компонент. Но если вам известно его название, то напечатайте его в окне поиска (Рис. С1.9), и в *Палитре компонентов* вы получите весь список подходящих компонентов (он автоматически обновляется по мере ввода новых букв). Щёлкните по названию компонента, и он появится на форме.

В *Палитре компонентов* включает несколько десятков страниц, из которых нам понадобятся только следующие.

Standard

– **стандартные** компоненты; наиболее часто используются в приложениях (Рис. С1.7).

	**фрейм**.
	**главное меню** программы.
	**контекстное меню**, появляющееся на экране при щелчке правой кнопкой мыши.
	**метка**; используется для вывода надписей на форму.
	**текстовое поле**, однострочный текстовый редактор. Позволяет вводить данные в работающей программе.
	простой **многострочный текстовый редактор**.
	стандартная **кнопка**.
	**флажок**; позволяет установить или отменить связанное с ним свойство. *В группе* может быть выбрано *любое* количество флажков.
	**переключатель**; одиночный действует, как флажок, но в *группе* может быть выбран только *один* переключатель.
	**список строк**.
	**комбинированный список** (текстовое поле и список).
	**полоса прокрутки**, **скроллер**.
	**контейнер для флажков**.
	**контейнер для переключателей**.
	**панель**, служит контейнером для размещения на ней других компонентов. Может быть использована для создания панелей инструментов и строки состояния.
	невизуальный компонент, создаёт список событий.

Additional

– **дополнительные** компоненты, многие из которых также используются очень часто (Рис. С1.10).

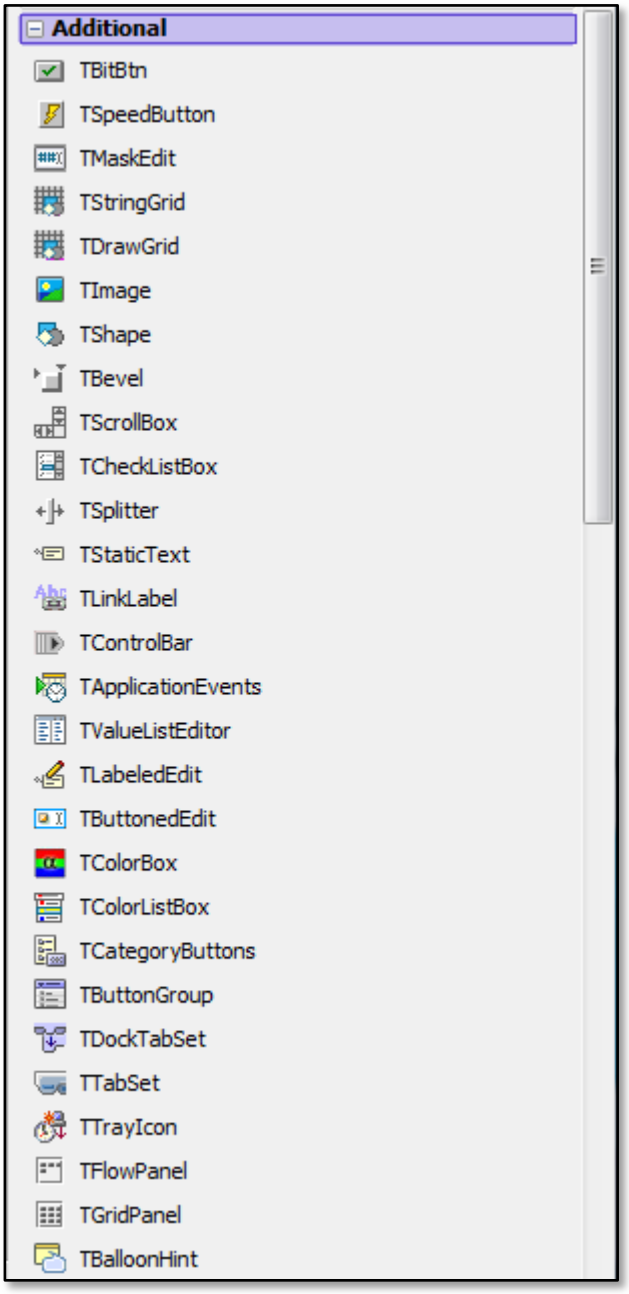
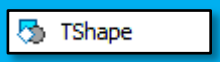
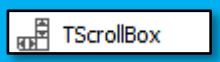
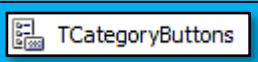


Рис. С1.10. Палитра *Additional*

TBitBtn	**кнопка** с растровой картинкой.
TSpeedButton	**кнопка** с растровой картинкой. Может выполнять функции флажка и переключателя, годится для создания панелей инструментов. Наиболее часто используется в программах, описанных в этой книге.

	аналог компонента **Edit**, но позволяет организовать ввод данных по образцу (телефонный номер, дата…).
	**сетка** для вывода **строк** в виде таблицы.
	**сетка** для вывода **данных** в виде таблицы. Главный компонент всех наших программ. Подробно рассматривается в одной из следующих глав.
	основа для вывода на экран **графических изображений**.
	простейшие **геометрические фигуры** (примитивы): эллипсы и прямоугольники.
	**рамка**.
	**контейнер** для других компонентов на форме, к которому автоматически добавляются полосы прокрутки, если размеры компонентов больше размеров контейнера.
	**список с флажками**.
	**разделитель** между двумя компонентами; позволяет изменять их размеры во время выполнения программы (примером может служить разделительная «черта» в *Инспекторе объектов*).
	аналог компонента **Label**.
	**плавающая панель** инструментов.
	**группа кнопок** категорий.
	компонент, похожий на предыдущий, но не поддерживает категории.

Win32

– компоненты 32-разрядного интерфейса (Рис. С1.11).

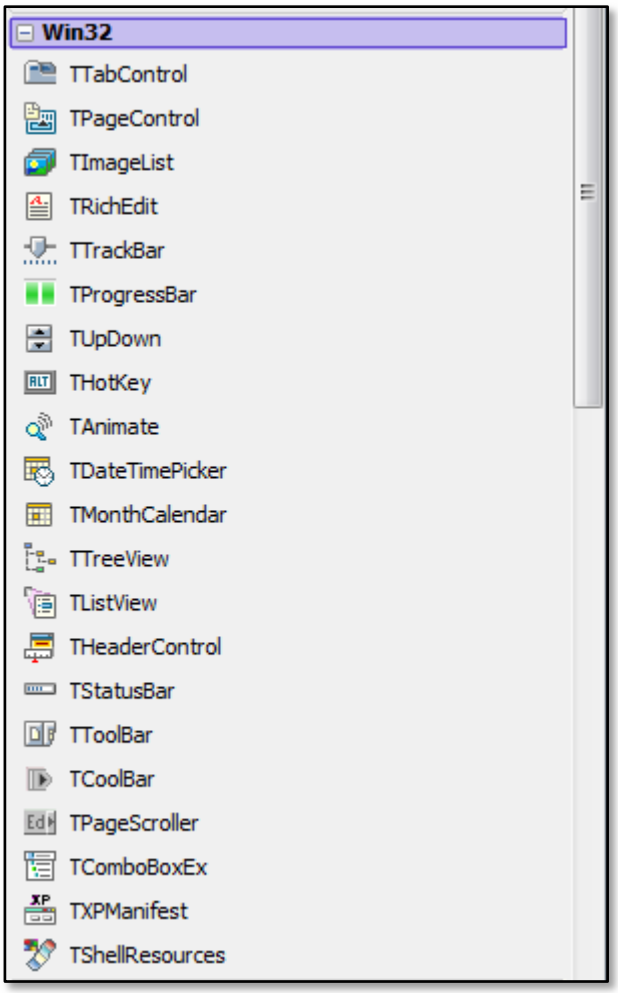


Рис. С1.11. Палитра *Win32*

TTabControl	**блокнот** со страницами (как *Рабочее окно Delphi*).
TPageControl	многостраничное **диалоговое окно**.
TImageList	**набор картинок** одинакового размера для вывода графики на компоненты. Годится также для хранения картинок, используемых в программе.
TRichEdit	полноценный *текстовый редактор* для работы с файлами в формате *rtf* (*Rich Text Format*).
TTrackBar	**ползунок** для установки числовых значений.
TProgressBar	**указатель** выполнения какого-либо процесса.
TUpDown	компонент для дискретного изменения числовых значений.

THotKey	назначает **горячую клавишу** для любого компонента.
TAnimate	**проигрыватель** AVI-файлов.
TDateTimePicker	**календарь и часы**.
TMonthCalendar	**календарь** на один месяц.
TTreeView	древовидный (иерархический) **список данных** (как *Проводник Windows*).
TListView	выводит **список данных** в колонки.
TStatusBar	**строка состояния**. Широко используется в программах для вывода подсказок и меняющейся информации.
TToolBar	**панель инструментов**. Удобный контейнер для кнопок одинакового размера.
TCoolBar	**плавающая панель инструментов**.

System

– **системные** управляющие компоненты (Рис. С1.12).

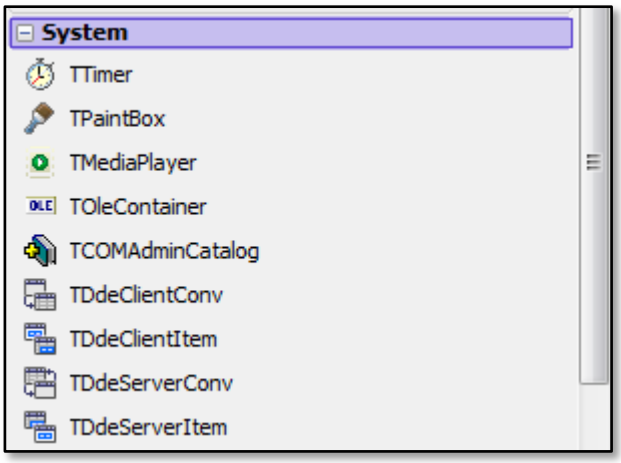


Рис. С1.12. Палитра *System*

TTimer	**таймер**; используется для отсчёта времени и анимации.
TPaintBox	прямоугольная область на форме **для вывода графики**.
TMediaPlayer	**проигрыватель мультимедиа-файлов.**
TOleContainer	**контейнер для OLE-объектов.**

Win 3.1

– компоненты для создания интерфейса в стиле **Win 3.1** (Рис. С1.13). Их основное назначение – сохранить совместимость с программами, написанными на ранних версиях *Delphi*. Многие из этих компонентов имеют 32-разрядные аналоги, которые и следует использовать при написании приложений.

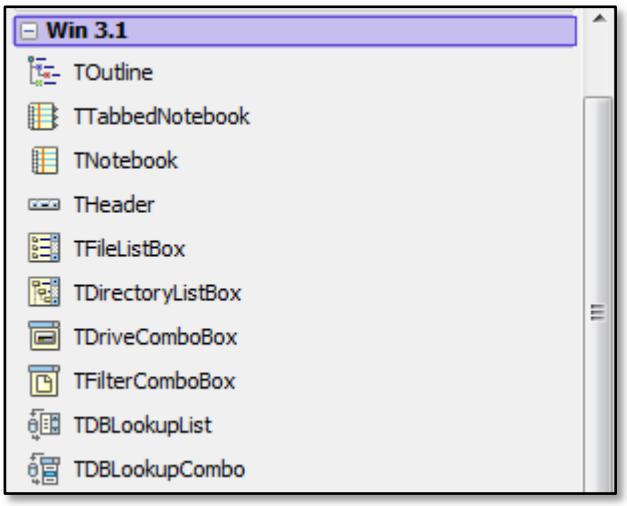


Рис. С1.13. Палитра *Win 3.1*

TOutline	аналог *TreeView*.
TTabbedNotebook	заменён компонентом *PageControl*.
TNotebook	заменён компонентом *PageControl*.
THeader	аналог компонента *HeaderControl*.
TFileListBox	**список файлов** в текущей директории (папке).
TDirectoryListBox	**список директорий** (папок) текущего диска.
TDriveComboBox	**список доступных дисков.**
TFilterComboBox	позволяет задать **фильтр** или **маску** для вывода списка файлов.

Dialogs

– невизуальные компоненты; вызывают стандартные **диалоговые окна** *Windows* (Рис. С1.14).

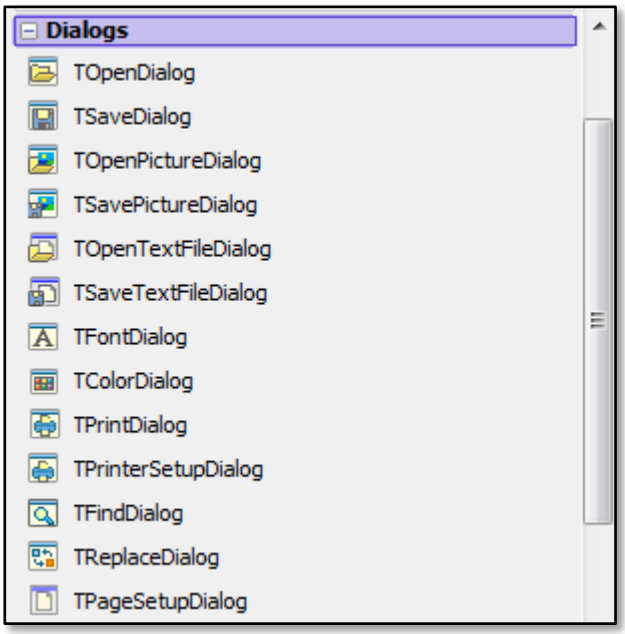


Рис. С1.14. Палитра *Dialogs*

TOpenDialog	**диалоговое окно открытия файлов.**
TSaveDialog	**диалоговое окно сохранения файлов.**
TOpenPictureDialog	**диалоговое окно открытия графических файлов**. Отличается от *OpenDialog* тем, что имеет область предварительного просмотра картинок.
TSavePictureDialog	**диалоговое окно сохранения графических файлов**. Имеет область предварительного просмотра картинок.
TFontDialog	**диалоговое окно выбора шрифта.**
TColorDialog	**диалоговое окно выбора цвета.**
TPrintDialog	**диалоговое окно печати на принтере.**
TPrinterSetupDialog	**диалоговое окно настройки принтера.**
TFindDialog	**диалоговое окно поиска строки в тексте.**
TReplaceDialog	**диалоговое окно замены строки в тексте.**
TPageSetupDialog	**диалоговое окно настройки параметров страницы** для печати на принтере.

Samples

– компоненты-**примеры** (Рис. С1.15).

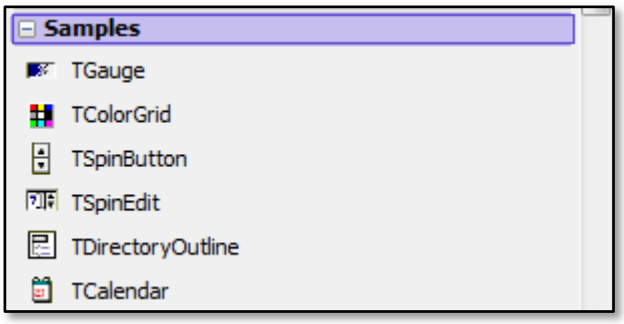


Рис. С1.15. Палитра *Samples*

TGauge	**указатель завершённости процесса.**
TColorGrid	**палитра для выбора цвета.**
TSpinButton	**дискретное изменение значения переменной.**
TSpinEdit	*SpinButton*, объединённый с **однострочным редактором**. Позволяет более удобно изменять значения числовых переменных.
TDirectoryOutline	**список директорий (папок).**
TCalendar	**календарь.**

Остальные компоненты в наших программах не используются.



Кроме *встроенных* компонентов, имеется множество компонентов других производителей. Некоторые из них могут быть очень полезны при разработке программ, так как дополняют и улучшают компоненты *Delphi*. Впрочем, злоупотреблять ими не следует, так как при переходе на новую версию *Delphi* вы не сможете работать с файлами программ, использующими компоненты для прежних версий. В этом случае отдавайте предпочтение тем компонентам, которые поставляются с исходным кодом, чтобы их можно было переустановить, либо компонентам от серьёзных производителей, которые оперативно выпускают их обновлённые версии.

Палитра компонентов имеет своё *контекстное меню*, которое появляется, как обычно, при нажатии правой кнопки мыши в области палитры и помогает настроить *Палитру компонентов* (Рис. С1.16).

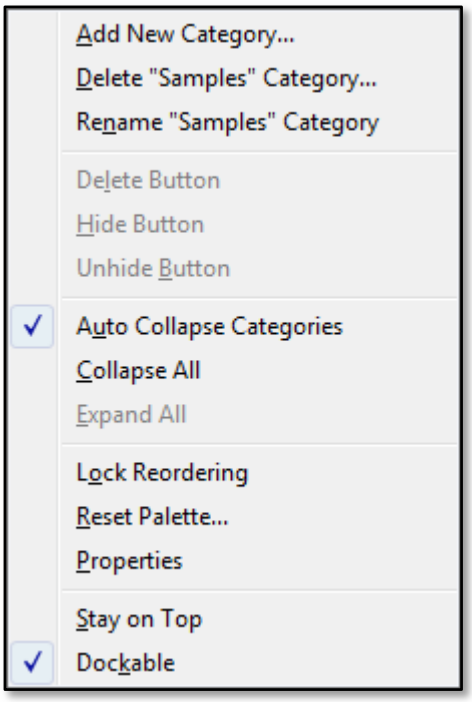


Рис. С1.16. Контекстное меню *Палитры компонентов*

Команда **Add New Category…** позволяет *добавить* новую страницу в конец *Палитры компонентов*. Сначала она будет пустая, но вы можете перетащить на неё компоненты с тех страниц, на которых вам обычно бывают нужны только некоторые компоненты.

Команда **Delete…**, наоборот, *удаляет* выбранную страницу вместе с её компонентами.

Команда **Rename…** позволяет *переименовать* выбранную страницу.

При установленном флажке **Auto Collapse Categories**, будет раскрыта только одна страница. При переходе на другую страницу предыдущая сворачивается, что экономит место на экране.

Команды **Collapse All** и **Expand All** *сворачивают* и *раскрывают* все страницы палитры, соответственно.

Установленный флажок **Lock Ordering** запрещает изменять порядок страниц в палитре.

Команда **Reset Palette…** удаляет все пользовательские настройки палитры. Все страницы *Палитры компонентов* появятся в первозданном виде, то есть точно такими, какими они были при первом запуске программы. Естественно, все ваши труды по благоустройству палитры пойдут насмарку.

Команда **Properties** открывает диалоговое окно *Options*, в котором можно настроить по своему вкусу внешний вид *Палитры компонентов*: размер значков компонентов (*Button Size*), окраску элементов палитры (*Color Schemes*), формат надписей на ней.

Установленный флажок **Stay on Top** всегда располагает *Панель компонентов* поверх других окон ИСР.

И последний флажок, **Dockable**, будучи включен, позволяет перетаскивать палитру по экрану и стыковать её с другими палитрами.

Урок 2. Грамматика *Delphi*

Аз да буки, а там и науки.

Русская пословица

- Скажите, а когда я закончу
курсы иностранных языков,
я смогу общаться с англичанами?
- Нет, но вы сможете разговаривать
по-английски с другими людьми,
которые посещали наши курсы!

Анекдот о пользе изучения языков

Программа на языке *Delphi* в окне *Редактора кода* напоминает обычный текст, состоящий из слов, чисел и различных знаков. И это естественно, ведь программист набирает его на своей клавиатуре и не может напечатать никакой «загогулины» и «отсебятины». Зато почти все символы на клавишах используются в программах:

- **Все строчные** и **прописные буквы** как латинского, так и русского алфавита, а также **знак подчёркивания** _ и **пробел**.

- **Все цифры**.

- **Специальные символы** # $ & ' ( ) * + , - . / : ; < = > @ [ ] ^ { }. Некоторые пары этих символов: (* (. *) .) .. // := <= >= <> имеют собственное значение. Например, круглые скобки с точкой заменяют квадратные скобки, а со звёздочкой – фигурные. Такая замена может потребоваться, если на клавиатуре отсутствуют эти символы.

Известно, что обезьяна, нажимая на клавиши печатной машинки (а теперь и компьютера), может написать *Войну и мир*. Ценность такой возможности, конечно, невелика, потому что Лев Толстой успел это сделать раньше, но вероятность плагиата всё равно остаётся. Таким образом, наука допускает, что, взяв в руки палку, обезьяна стала человеком, а взяв клавиатуру, сможет стать писателем. Анализ текстов знаменитой детективной писательницы Дарьи Донцовой раскрыл секрет её творчества: на неё работает весь обезьяний питомник Московского зоопарка.

Вначале было слово.

Библейское

Осталось правильно расставить буквы.

Леонид Якубович из Поля чудес

Числа

Начнём мы с того, что из цифр составим **числа**. Они могут быть **целыми** и **действительными** (числа с плавающей точкой, вещественные). Отличительной чертой действительных чисел является *десятичная точка* (а не запятая, как в школьной математике!). Если число отрицательное, то перед ним ставится минус. Перед положительным числом можно поставить плюс, но обычно этого не делают. Примеры чисел в *Delphi*:

```
1    +27   865   5470 - положительные целые числа
-3   -87 -555   -9536 - отрицательные целые числа
0.9    2.476 72.0005 - положительные действительные числа
-0.7 -21.43 -573.9999 - отрицательные действительные числа
```

Целые и действительные числа называют также *числовыми константами* (целыми и действительными, соответственно).

Целые числа иногда записывают и в 16-ричном виде с помощью цифр 0..9 и латинских букв A..F (или a..f). Перед таким числом должен стоять символ $:

```
$0A   $20   $FFFFFFFF
```

Действительные числа могут записываться и в *научной нотации*:

`1.5e8` (`1.5E8`), что составляет 150 000 000 (догадайтесь, что за число!).

Выражения

Из числовых констант и арифметических знаков можно составить **выражения**:

```
1 + 2
2 * 2
3 * 4 + 5 - 6/2
-+12
```

```
-2.67 * 3.14 + -3/2 - 7.01
```

Значение числовых (и других) константных выражений вычисляются транслятором при компиляции программы и подставляются вместо них, поэтому константные выражения не могут содержать переменных (ведь их значения могут изменяться!), а также вызовов функций (за редким исключением).

При вычислении значения выражения учитывается приоритет операций – точно так же, как и в математике. Сначала слева направо выполняются операции умножения и деления, затем сложения и вычитания. Изменить порядок вычисления можно с помощью круглых скобок (не квадратных и не фигурных!) – в полном согласии с законами арифметики:

```
(2 + 4 - 1) / 5 * 11
(3.1415 - 1.0) / 7.78 * (45 - 7)
```

Между числами и знаками операций может быть любое число пробелов, но можно печатать их и вплотную друг к другу (особенно это полезно делать, чтобы испортить зрение и откосить от армии).

Выражения могут состоять не только из числовых констант (и всех других), но из идентификаторов переменных, функций и процедур.

Идентификаторы

Как вы судно назовёте,
так оно и поплывёт.

Ной

Если мы хотим обратиться к человеку, то должны знать его имя (в особо ответственных случаях – полное ФИО). То же самое и в *Delphi* – редкий объект обходится без имени, которое также называется **идентификатором**.

Формально идентификатором может служить любая последовательность букв и цифр, начинающаяся с буквы или знака подчёркивания. Длина идентификатора не ограничена, но транслятор учитывает только первые 255 символов (трудно даже представить себе имя такой длины!). Таким образом,

```
_
i
```

```
abcd
qwerty
i_j
g56
o8j90
```

это правильные имена для любых объектов программы. Но они годятся только в качестве кодов к замкам, потому что не имеют никакого смысла, доступного разумению.



Delphi не делает различий между строчными и прописными буквами, в отличие, например, от языка *C*. Поэтому идентификаторы

delphi
Delphi
DELPHI
DeLpHi

обозначают один и тот же объект!

Как правильно выбрать имя

- Не беспокойся, Козлодоев...
- Козодоев!
- Козлодоев! Буду бить аккуратно, но сильно!

Диалог Лёлика и Гены из *Бриллиантовой руки*

Самое важное правило составления имён
заключается в том, чтобы это правило было.
Если Вы называете переменные по сиюминутной прихоти,
то не только дойдёте до странных имён,
которые потом и не вспомните,
но и потратите массу энергии
на выдумывание «умных» имён,
вместо того, чтобы придумывать умные алгоритмы.

Брюс Мак-Кинни. *Крепкий орешек Visual Basic 4*

Пространная цитата из Брюса Мак-Кинни, конечно, не случайна – она должна наставить нас на путь истинный при наречении новорожденного элемента программы.

Delphi даёт всем вновь созданным объектам маловразумительные имена:

```
Project1
Form1
Unit1
```

Если вы добавите на форму кнопку, то она будет называться, конечно, *Button1* (Рис. У2.1). Сможете ли вы или какой-нибудь другой, более проницательный человек, догадаться, что будет, если нажать на эту кнопку? – Вряд ли, ведь и на самой кнопке тоже написано *Button1*!

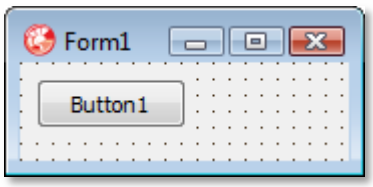


Рис. У2.1. Кнопка, на которой написано, что это кнопка!

И не надейтесь, что со второй кнопкой будет лучше, *Delphi* назовёт её – «никогда не догадаетесь!», - как восклицала в подобных таинственных случаях Регина Дубовицкая, – *Button2*.

Поскольку *Delphi* даёт нам полную свободу словотворчества, мы можем в качестве идентификаторов использовать *любые* слова, которые знаем или можем придумать.

Этим не следует злоупотреблять, но это так. Запрещено использовать не по прямому назначению только **зарезервированные слова**:

and	**else**	**inherited**	**packed**	**then**
array	**end**	**initialization**	**procedure**	**threadvar**
as	**except**	**inline**	**program**	**to**
asm	**exports**	**interface**	**property**	**try**
begin	**file**	**is**	**raise**	**type**
case	**final**	**label**	**record**	**unit**
class	**finalization**	**library**	**repeat**	**unsafe**
const	**finally**	**mod**	**resourcestring**	**until**
constructor	**for**	**nil**	**sealed**	**uses**
destructor	**function**	**not**	**set**	**var**
dispinterface	**goto**	**object**	**shl**	**while**
div	**if**	**of**	**shr**	**with**
do	**implementation**	**or**	**static**	**xor**
downto	**in**	**out**	**string**	

Не следует применять всуе и некоторые другие слова, например, директивы. Такие ограничения не должны вас огорчать: мало ли в русском языке других слов!

Идентификаторы по-венгерски

Ох, я в Венгрии по рынку похожу,
На венгерских на румынок погляжу…

В Высоцкий. *Инструкция перед поездкой за рубеж*

Плохие идентификаторы, как и плохие имена, затрудняют жизнь людей и программ. Чтобы облегчить выбор сотрудник фирмы *Майкрософт* Чарльз Силош предложил весьма сложную систему составления имён объектов программы. Как известно, венгерский язык очень сложный, но «венгерской» эту систему прозвали не поэтому, просто Чарльз Силош по происхождению венгр.

Существует немало разновидностей этой системы, но мы остановимся на самой простой и внятной из них. Итак, «по-венгерски» любой идентификатор составляют из двух частей:

- **префикса**, записанного строчными буквами, который указывает на тип объекта, и

- **спецификатора**, начинающегося с прописной буквы.

Например, префикс **i** обозначает *целый* тип. Тогда координаты точки на экране можно хранить в двух переменных:

```
Var
  iX, iY: integer;   или
  iXCoord, iYCoord: integer;   или
  iCoordX, iCoordY: integer;
```

Если имеется в виду не точка, а мяч, то идентификатор легко приспособить и к этому случаю:

```
Var
  iXBall, iYBall: integer;
```

Префикс может также отражать *назначение* объекта:

c (*counter*) – счётчик;
f (*flag*) - переменная логического типа;

p (*pointer*) – указатель;
n, num (*number*) – число, количество чего-либо;
cur (*current*) – текущее значение;
tmp (*temporal*) – временное значение.

Вы можете выработать свою *систему наименования*, но некоторые распространённые префиксы желательно при этом учитывать.

b, flg	переменная логического типа **boolean**
b	переменная целого типа **byte**
i	переменная целого типа **integer**
g_	глобальная переменная
r	переменная вещественного типа **single**, **double**
a, arr	массив (**array**)
h	описатель (**handle**)
x,y,z	координаты точки
dx, dy	приращение координат
T	названия типов
F	названия полей
C	названия классов (стиль языка C++)
id, idx	индекс, идентификатор
frm	форма
cbo	комбинированный список
lst	список
txt	текстовое поле
cmd	командная кнопка типа **TButton**
sbt	командная кнопка типа **TSpeedButton**
sb	полоса прокрутки типа **TScrollBar**
ico, pic, img, bmp	рисунок
iml	список рисунков типа **TImageList**
mnu	меню
tmr	таймер типа **TTimer**
opt	переключатель

Этот список вы можете продолжить сами – место в таблице предусмотрительно оставлено!

Глобальные константы обычно записывают прописными буквами, разделяя отдельные слова знаком подчёркивания:

```
Const
  MAX_WIDTH= 20;
  MAX_HEIGHT= 24;
```

Ну, вот, теперь вы можете «обзывать» объекты строго по-научному!

Идентификаторы по-русски

Сердцеведением и мудрым познанием жизни
отзовётся слово британца;
легким щёголем блеснёт и разлетится
недолговечное слово француза;
затейливо придумает своё, не всякому доступное,
умно-худощавое слово немец;
но нет слова, которое было бы так замашисто,
бойко, так вырвалось бы из-под самого сердца,
так бы кипело и живо трепетало,
как метко сказанное русское слово.

Гоголь. *Мёртвые души*

В защиту русских слов: их разрешено использовать не только в комментариях, но и - весьма опрометчиво! - для именования переменных и прочих элементов программы. Это значит, что вы можете написать не только `fence:= 1`, но и `забор:= 1`. И даже то, что потом пишут на заборе (я имею в виду надпись *Осторожно - злая собака!*). Обязательно так и делайте, и тогда ни на одном вражеском компьютере исходный текст вашей программы работать не станет, а постоянное переключение раскладки клавиатуры с английской на отечественную сильно укрепит ваше материалистическое мировоззрение и повысит меткость русского слова.

Комментарии

Как птицу видно по полёту, так и хорошего программиста – по комментариям в программе. Их должно быть много, но они должны описывать алгоритм, а не элементарные действия, которые и без того всем понятны. Например, комментарий

```
i:= i+1;  // увеличиваем значение переменной i на единицу
```

ничего не объясняет, а просто пересказывает действие оператора сложения.

Комментарий

```
c:= c+1;  // увеличиваем значение счётчика слов
```

уже значительно лучше, поскольку мы понимаем, что в этом месте кода подсчитывается количество слов, а не абстрактных объектов.

Когда вы только пишете программу, вам и без комментариев всё в ней понятно. Но если вам понадобится модифицировать программу, написанную несколько месяцев назад, вот тогда и начнутся проблемы: без комментариев придётся восстанавливать весь алгоритм заново, а это иногда бывает труднее, чем с начала придумать новый. Не меньше трудностей вы создадите и своим друзьям-товарищам, если поделитесь с ними своей программой. Вряд ли им доставит удовольствие многочасовой разбор «немого» кода. Ещё больше вы сможете насолить им и себе, если будете называть объекты малопонятными именами. Вывод такой: не жалейте времени на комментирование программы. Вам и самому будет легко разобраться в коде и другие программисты будут вспоминать вас с благодарностью всю оставшуюся жизнь.

А вот транслятору ваши комментарии ни к чему, он просто игнорирует их при компиляции программы, так что от ваших пояснений выполняемый код не увеличится ни на один байт.

Обычно комментарии занимают целиком всю строку либо остаток строки:

```
// увеличиваем значение счётчика слов
c:= c+1;
c:= c+1; // увеличиваем значение счётчика слов
```

В этом случае они начинаются двумя слешами, за которыми и следует собственно комментарий. В самом комментарии разрешается использовать любые символы.

Реже комментарии располагают внутри кода (в любом его месте), тогда они обрамляется фигурными скобками:

```
c:= {увеличиваем значение счётчика слов} c+1;
```

Вместо фигурных скобок можно использовать круглые скобки со звёздочками:

```
c:= (* увеличиваем значение счётчика слов *) c+1;
```

Второй способ можно применять и для комментариев внутри комментариев, записанных с помощью фигурных скобок:

```
{
      // увеличиваем значение счётчика слов
      c:= c+1;
      c:= c+1; // увеличиваем значение счётчика слов
      c:= {увеличиваем значение счётчика слов} c+1;
      c:= (* увеличиваем значение счётчика слов *) c+1;
}
```

Если вам при отладке программы понадобится закомментировать блок кода, то достаточно заключить его в фигурные скобки. С одностррчными комментариями всё будет нормально, но комментарии с фигурными скобками будут вызывать коллизии. Обратите внимание на третью строку кода: закрывающая фигурная скобка завершает комментарий, и весь следующий код оказывается раскомментированным, что приведёт к ошибке, как только компилятор наткнётся на вторую закрывающую фигурную скобку.

Программы и алгоритмы

Программа – это последовательность инструкций (команд), которые выполняют заданный алгоритм. Для обычного человека программа – просто непонятный текст, разбитый на отдельные строки. В каждой строке обычно располагается одна команды (вы можете нашпиговать её множеством команд, но это только усложнит понимание программы). К сожалению, компилятор не понимает человеческого языка (иначе было бы достаточно комментариев к программе), поэтому людям пришлось придумывать языки программирования, которые одинаково плохо понятны как людям, так и компьютерам (не думаете же вы, что процессор понимает паскаль или бейсик; он «говорит» на своём, родном машинном языке). Программу, написанную, например, на *Delphi* транслятор переводит в машинный код, который затем и выполняется процессором.

Для нас же важно то, что программа – это не любой код, пусть даже и правильный с точки зрения транслятора, а только тот, который решает определённую задачу. И решает он её с помощью алгоритма, переведённого на язык программирования. Таким образом, чтобы написать программу, недостаточно знать грамматику, орфографию и пунктуацию языка программирования. Выучить их как раз не очень сложно, поэтому параллельно с изучение языка *Delphi* мы будем заниматься составлением алгоритма для каждого примера и приложения. Без этого умения вы всю жизнь будете играть гаммы, вместо того, чтобы писать симфонии.

В общем случае **алгоритм** – это набор правил, которые однозначно определяют содержание и последовательность выполнения действий для решения поставленной задачи. Не следует думать, что алгоритмы нужны только в программировании. Люди и сами ежедневно выполняют какие-либо алгоритмы. Например, при лепке и варке пельменей вы пользуетесь рецептом, в котором написано: возьми то-то и то-то, сделай так-то и так-то. Если алгоритм-рецепт хороший, а вы прилежный исполнитель, то и пельмени выйдут на славу. Но обычно пельмени получаются комом, поэтому рано или поздно приходится покупать компьютер и жениться.



Множеством алгоритмов нас нагружают в школе и в жизни как таковой: типовые задачи по химии и физике, правила решения квадратных уравнений и написания сочинений, правила дорожного движения и этикета. Так что, выполняя все алгоритмы, принятые в обществе, вы станете примерным гражданином, чтущим уголовный кодекс. В противном случае – человеческим фактором или компьютерным гением.

Семинар 2. С чего начать?, или Проект, откройся!

Лёгким движением руки
брюки превращаются в … элегантные шорты.

Незабываемый показ мод в исполнении Козодоева

Если в прежних версиях *Delphi* новый проект открывался автоматически при загрузке ИСР, то теперь вам придётся делать это самостоятельно. Для этого имеется несколько способов.

Чтобы **начать новый проект**, нажмите кнопку **New Items** и в открывшемся диалоговом окне выберите *VCL Forms Application* (Рис. С2.1).

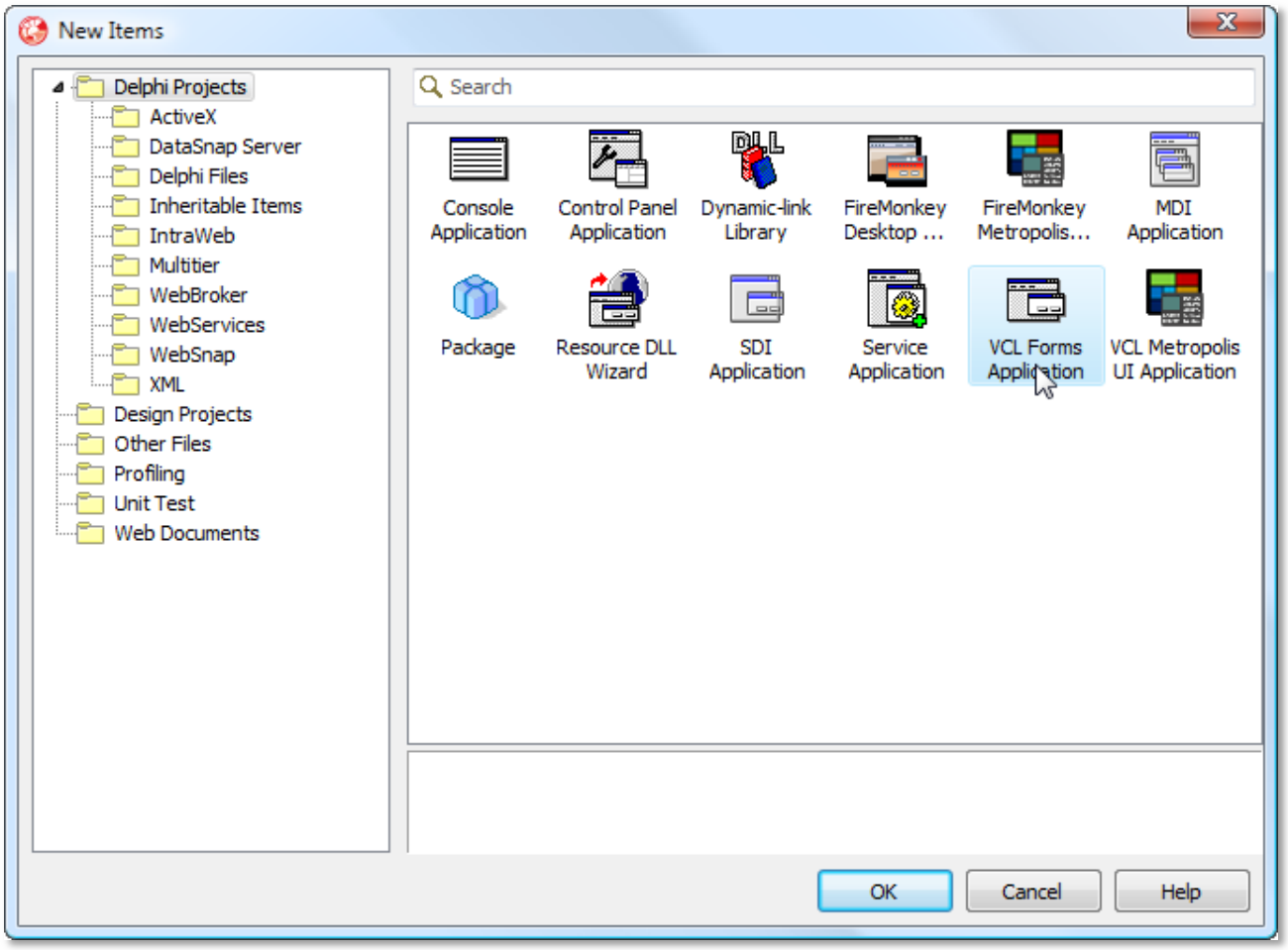


Рис. С2.1. Начинаем новый проект кнопкой *New Items*!

Открыть новый проект можно также из *Панели инструментов* (Рис. С2.2).

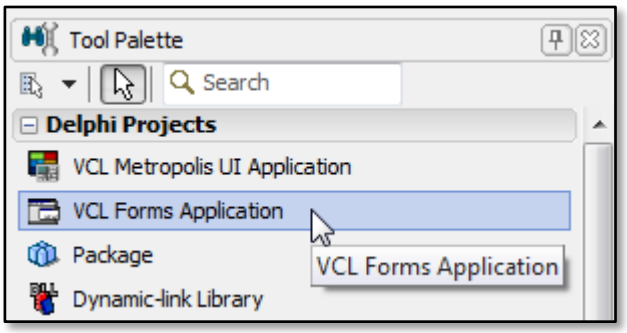


Рис. С2.2. Начинаем новый проект из *Панели инструментов*!

Из окна *Менеджера проекта* (Рис. С2.3).

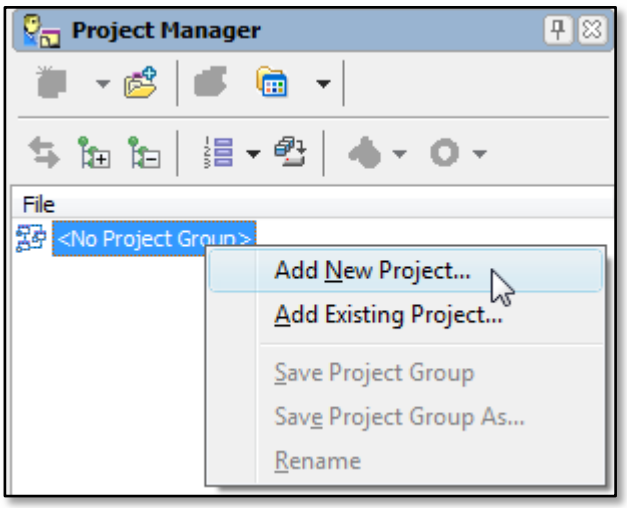


Рис. С2.3. Начинаем новый проект из *Менеджера проекта*!

Из *Окна приветствия*, нажав строчку **New Project...** (Рис. С2.4).

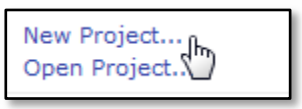


Рис. С2.4. Начинаем новый проект из *Окна приветствия*!

И из *Главного меню*, выполнив команду **Project > Add New Project...** (Рис. С2.5)

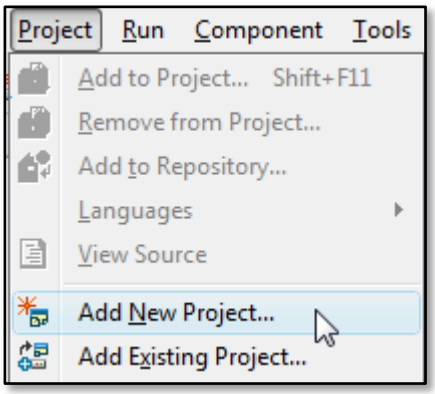


Рис. С2.5. Начинаем новый проект из *меню Project*!

или **File > New > VCL Forms Application – Delphi for Win32** (Рис. С2.6).

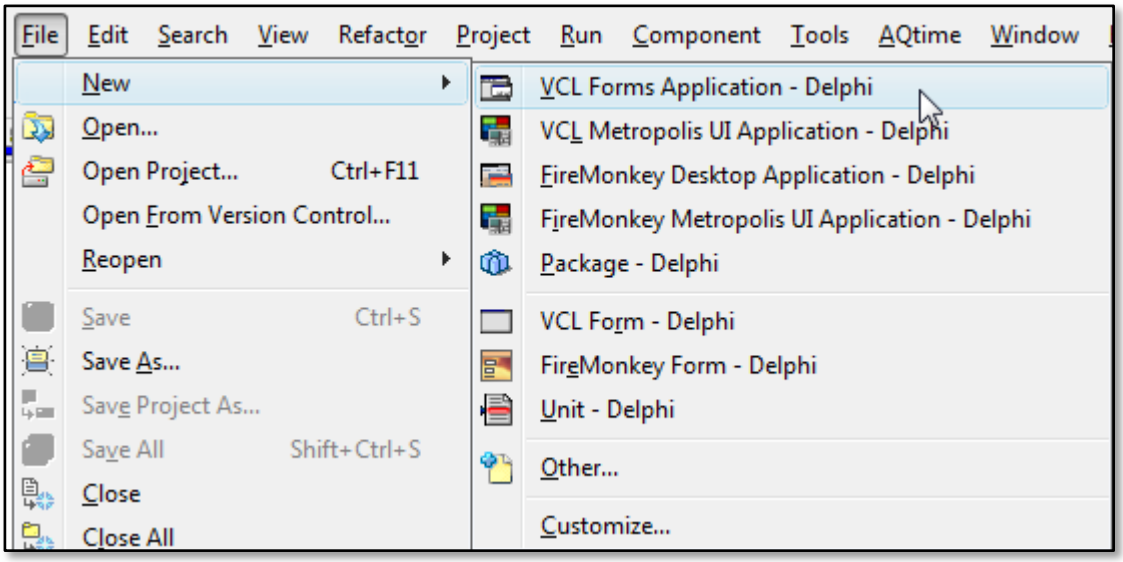


Рис. С2.6. Начинаем новый проект из *меню File*!

В *Рабочем окне* появится новая закладка – **Unit1** и главная форма приложения (Рис. С2.7).

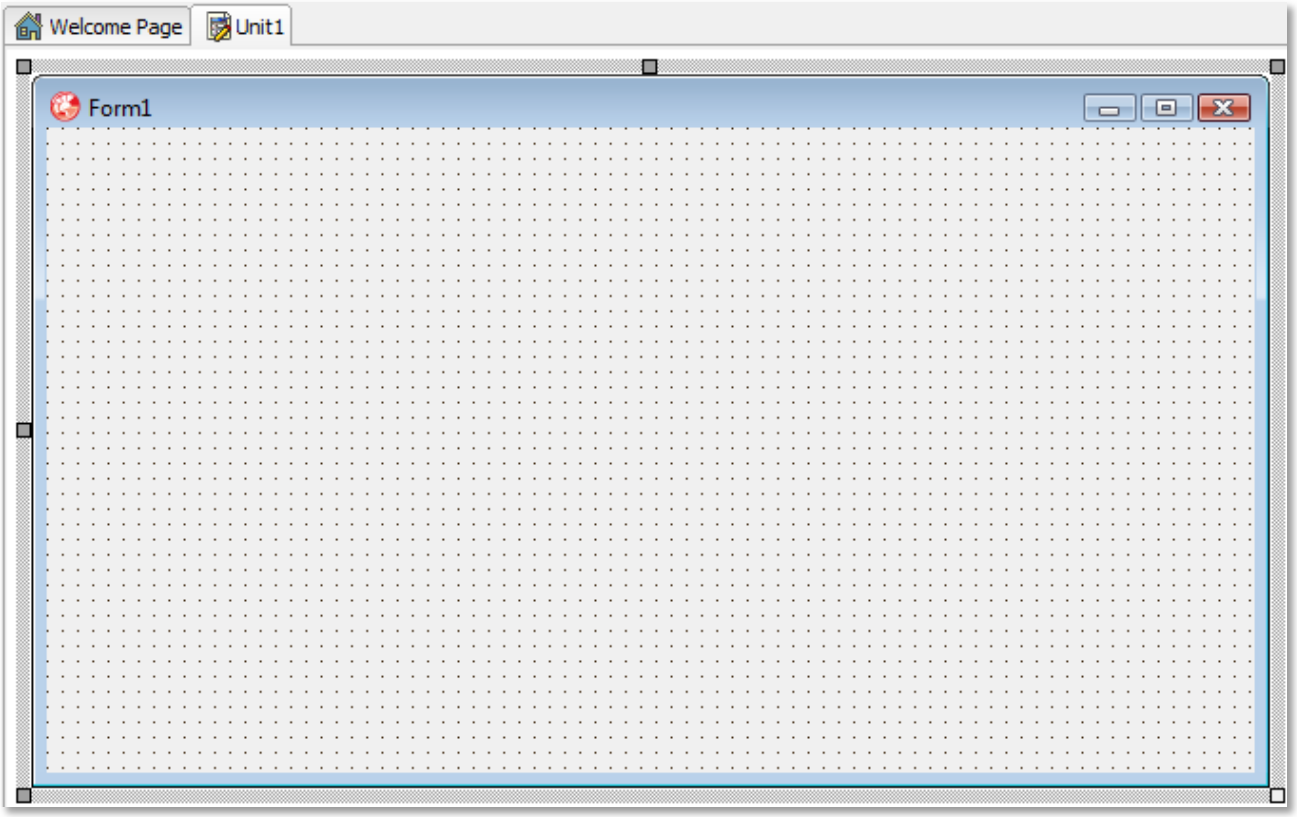


Рис. С2.7. С открытием!

И хотя вы ещё ничего не успели сделать, ваш проект уже содержит пустую форму **Form1** (она находится в окне *Конструктора формы*), модуль формы **Unit1.pas** (в окне *Редактора кода*) и ещё несколько файлов..

Как сохранить файлы проекта на диске

Сохраняйся!

Программист Козьма Прутков

Так как с этого и следует начинать работу над новым проектом, то создайте для него на диске *отдельную папку*. Затем нажмите кнопку - аналог команды **File > Save all** (быстрые клавиши – **Shift + Ctrl + S**) на стандартной панели и в открывшемся диалоговом окне перейдите в эту папку.

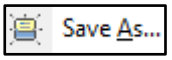


Новую папку можно создать и непосредственно из диалогового окна, воспользовавшись услугами кнопки 

.

Вместо названия модуля по умолчанию *Unit1* выберите более подходящее, например *MyUnit*, и сохраните его на диске. Тут же проделайте эту операцию и файлом проекта (назовём его *MyProject*).



Записать новый проект можно и из меню. Командой **File >** 

сохраните модуль, а командой **File >** 

- файл проекта.

Менеджер проекта также облегчает работу с файлами проекта. Например, после нажатия правой кнопки мыши на строчке *Unit1.pas*, на экране появится контекстное меню (Рис. С2.8).

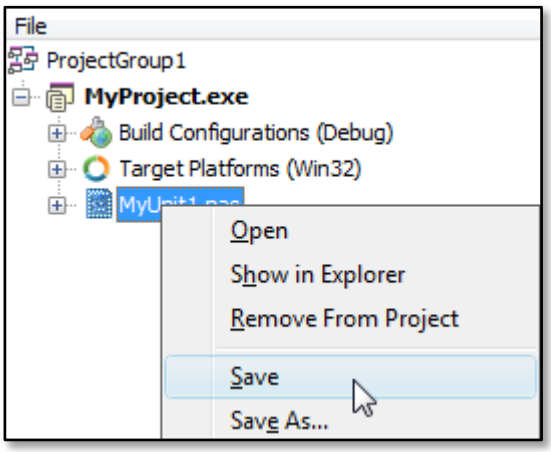


Рис. С2.8. Контекстное меню *Менеджера проекта*

В нём вы можете *сохранить* файл модуля на диске (команды **Save** или **Save As...**), *переименовать* его (**Rename**) и *удалить* из проекта (**Remove From Project**).



Имейте в виду, что *Delphi* не позволяет присваивать имя проекта файлу модуля, хотя они и имеют разные расширения.



Любой проект на *Delphi* включает в себя довольно много файлов. Чтобы они не перемешались с другими файлами, их нужно сохранять в *отдельной* папке, желательно в корневом каталоге диска и под «прозрачным» названием, иначе потом вам придётся долго искать её среди прочих папок с ничего не значащими именами типа «абвгд» или «ёклмн». Можно предварить название папки символом подчёркивания «_», тогда она будет находиться в начале списка папок на диске. Закончив работу над проектом, вы можете сте-

реть этот символ, папку заархивировать и убрать на вечное хранение в папку для всех ваших проектов на *Delphi*. Такой подход позволит вам всегда быстро находить необходимые файлы.



Переключаться между окнами *Редактора кода* и *Конструктора формы* удобно с помощью кнопки  на *Панели инструментов*.

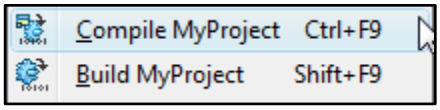
Выбрать любой модуль проекта для отображения в окне *Редактора кода* можно из списка, который появляется после нажатия на кнопку . Для перехода с одной страницы на другую в *Редакторе кода* достаточно щёлкнуть мышкой на табуляторе с названием модуля.

После того как вы сохранили файлы проекта под другими именами, они будут обновлены и в заголовках *Главного окна* и окна *Редактора кода*. А заголовок формы отражает значение её свойства *Caption*, которое можно установить в *Инспекторе объектов*. Обычно это название программы и номер её версии. Само же название формы задаётся свойством *Name* в том же *Инспекторе объектов*. Если значение свойства *Caption* не установлено, то именно название формы и появится в её заголовке (Рис. С2.9).



После набора очередной порции кода, сохраняйте его текущее состояние в файле. Для этого нажмите кнопку , если вы хотите записать **все** открытые файлы проекта, или кнопку , если изменения коснулись только **отдельного** модуля.

Никогда не набирайте текст программы подолгу, минут через 5-10 сохраняйте изменения на диске! Особенно важно сохранять результаты работы перед отладочным запуском программы из среды разработки. Программа вполне может зависнуть, и всё, что нажито непосильным трудом, придётся набирать ещё раз.

При большом риске зависания программы её лучше скомпилировать одной из команд меню *Project* 

, а потом запустить из *Проводника* или файлового менеджера выполняемый файл приложения.

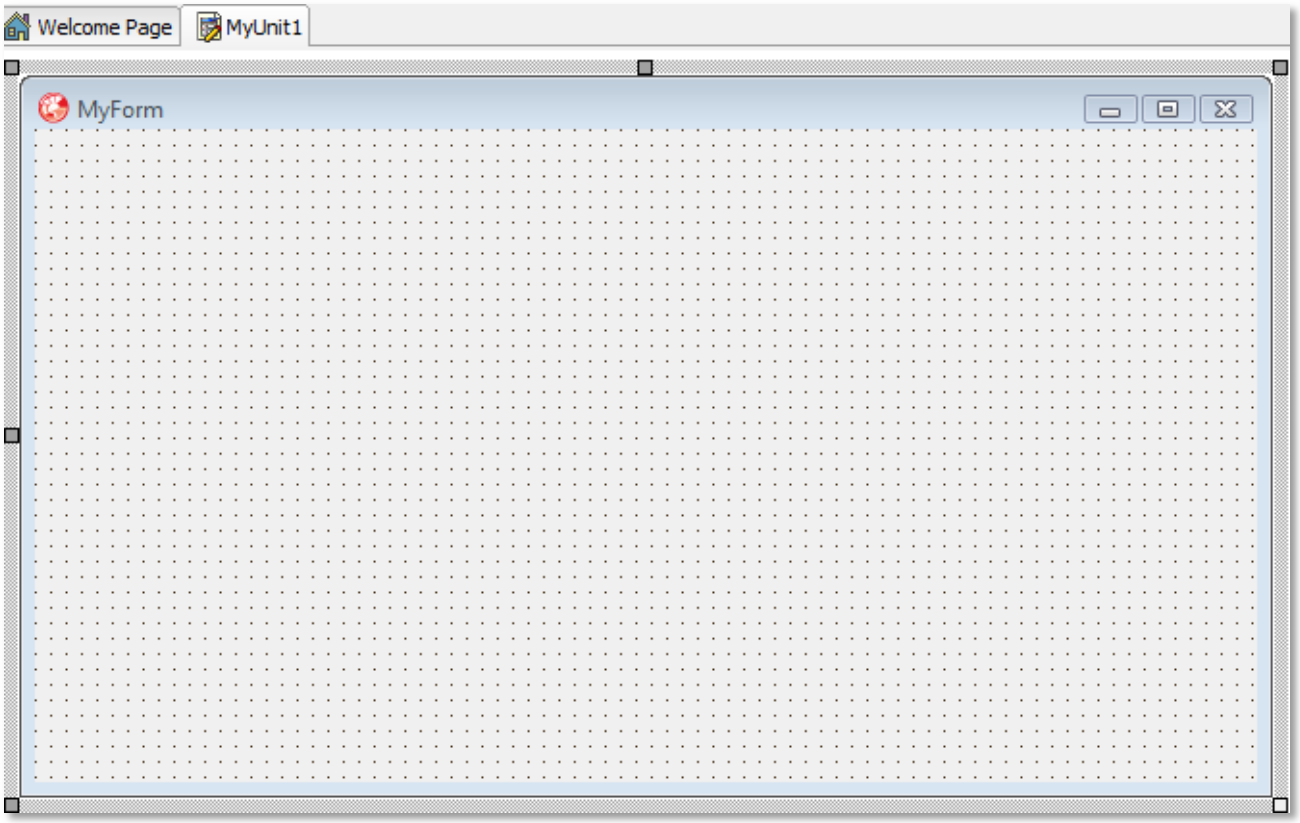


Рис. С2.9. Приложение обзавелось новыми именами

Как это ни удивительно, но вы можете уже сейчас *запустить* ваш проект. Нажмите кнопку  на *Панели инструментов*, и через некоторое время на экране появится форма, представляющая собой стандартное окно *Windows* (Рис. С2.10), которое можно передвигать, сворачивать, разворачивать, изменять его размеры и закрывать (в *Панели задач Windows* появится также название вашего проекта со значком из файла ресурсов проекта). Больше ничего вы от неё не добьётесь, поэтому выполните именно последнюю из доступных команд, чтобы вернуться в среду программирования *Delphi*.

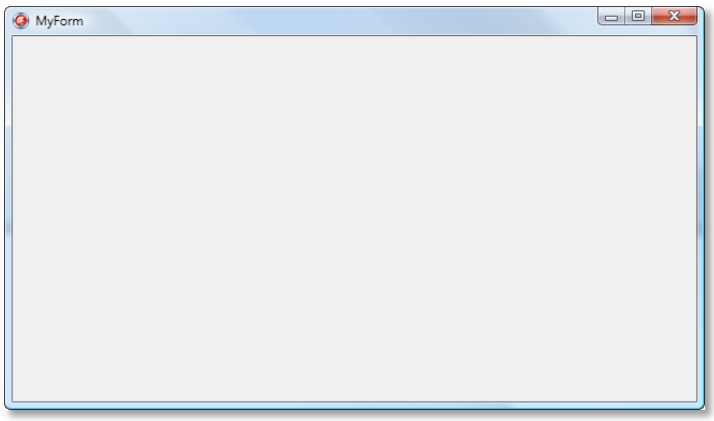


Рис. С2.10. Окно приложения

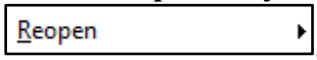
Если вас терзает любопытство, загляните в папку с проектом – в ней появился файл **MyProject.exe** (а именно в папке *Win32 > Debug*), который обычно и называют программой. И пусть вы не написали до сих пор ни одной строчки кода, но уже получили самую настоящую программу с выбранным вами названием. Вы можете запустить её из *Проводника Windows* или файлового менеджера, дважды щёлкнув по названию (или значку) мышкой. Конечно, ничего нового она не сделает по сравнению с первым стартом, но её можно скопировать на диск и запустить на любом другом компьютере – она не требует для своей работы никаких других файлов. Это большое преимущество программ, написанных на *Delphi*, – они исполняются без нудной установки на компьютер (впрочем, вы можете легко испортить людям жизнь, добавив к готовой программе инсталлятор со всякими вопросами, предупреждениями и просьбами).

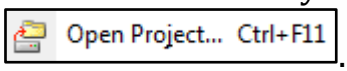


Исходный код программы находится в папке **Пример**.

Как загрузить проект с диска

Сохранённый на диске проект вы можете вновь **открыть** в *Delphi*.

Особенно удобна для этого кнопка **Open**  на *Панели инструментов*. Нажмите стрелочку справа от папки, чтобы раскрыть список последних проектов и файлов, и выберите нужный. Так же действует и команда *Главного меню* **File >** 

. Если нажать на саму кнопку, то она выполнит команду *Главного меню* **File >** 

.

Нажав строчку **Open Project...** на *Странице приветствия* (Рис. С2.11) или кнопку  (**Ctrl+F11**) на *Панели инструментов*,

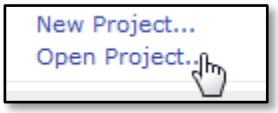


Рис. С2.11. Открываем проект на *Стартовой странице*

вы попадёте в диалоговое окно выбора папки проекта (Рис. С2.12).

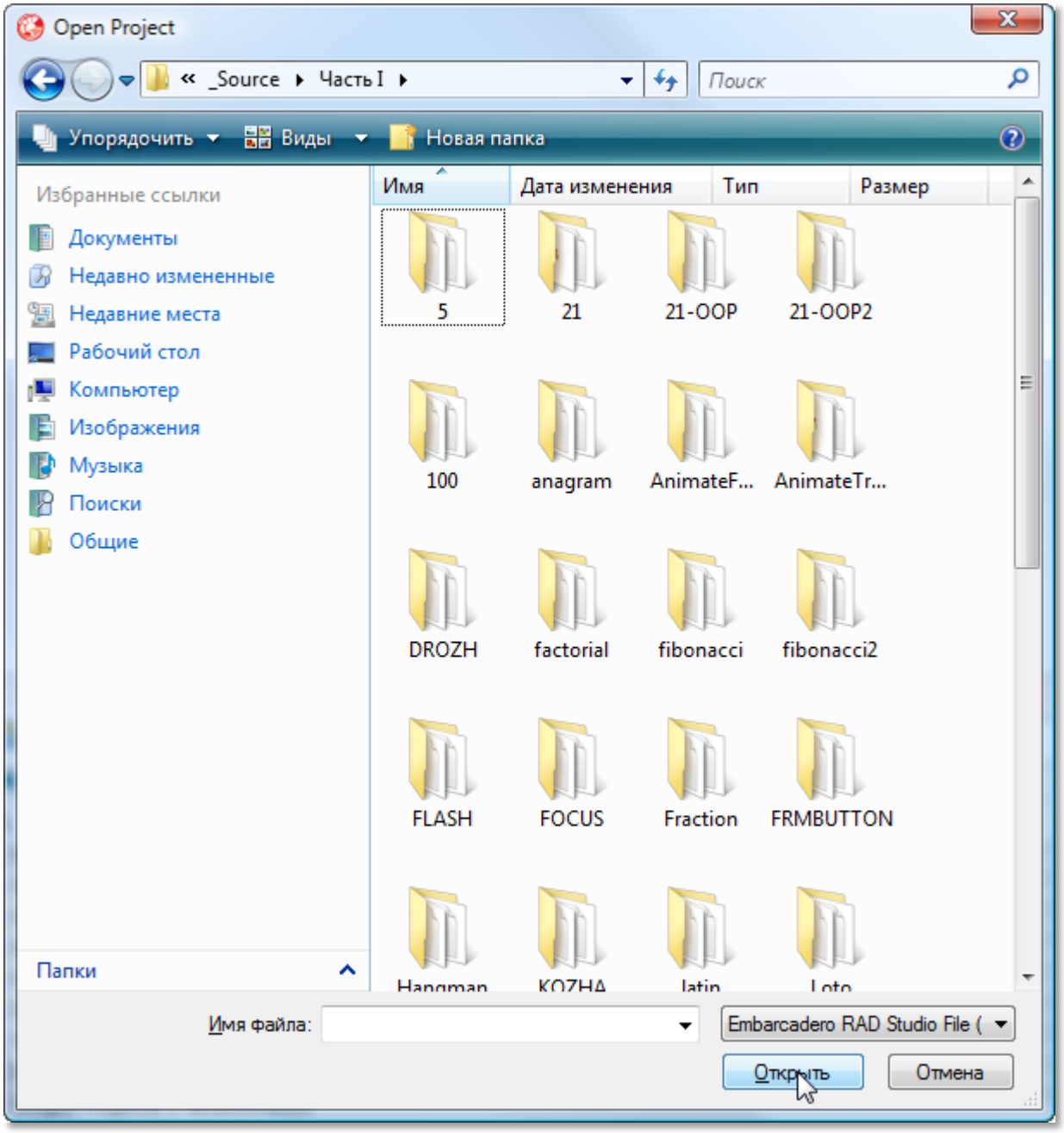
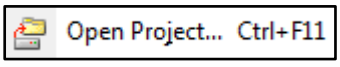


Рис. С2.12. Папки с проектами на диске

Команда *Главного меню* **File >** 

также открывает это диалоговое окно.

Чтобы **открыть** проект, нужно загрузить один из файлов с расширением *.dpr* или *.dproj*, которые находятся в папке с проектом.

Как закрыть проект

Рано или поздно, работу над проектом придётся закончить и заняться менее приятными делами.

Можно закрывать каждый файл по отдельности (сначала его нужно вывести в окно *Редактора кода*), нажимая строку в *Главном меню* **File >** Close. Если в файле остались несохранённые изменения, то *Delphi* напомнит вам об этом (Рис. С2.13).

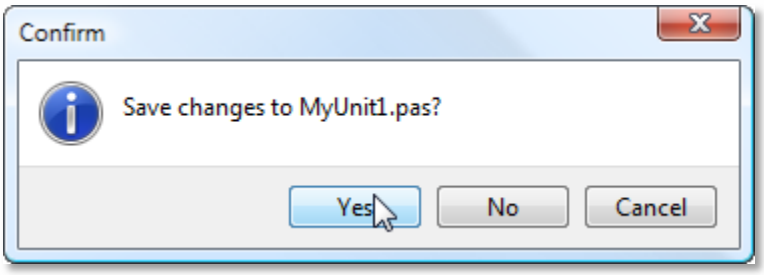


Рис. С2.13. Не торопитесь!

Разом закрывает все файлы проекта команда **File >** Close All. В этом случае вы можете получить другое предупреждение (Рис. С2.14).

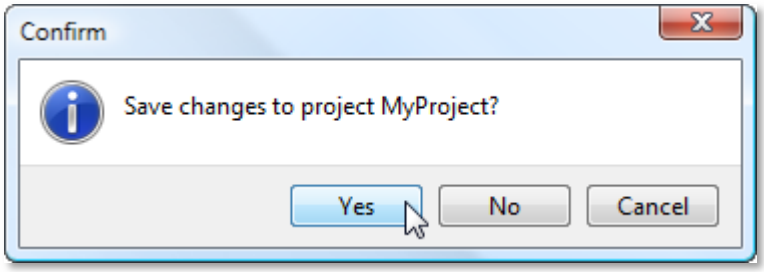


Рис. С2.14. Сохраняйте изменения!

Естественно, если у вас на то нет веских причин, следует нажать кнопку *Yes*.

После этих действий вы попадёте на *Страницу приветствия* и сможете открыть любой проект.

Если же вы полностью и окончательно утолили жажду программирования, то закройте *Delphi* командой **File >** Exit или, как и всякое другое прило-

жение, нажав крестик в правом верхнем углу *Главного окна Delphi* (Рис. С2.15).



Рис. С2.15. Тут и *Delphi* конец!

Урок 3. Переменные и константы

- Константин, что в переводе с греческого значит „постоянный“!

Стандартное представление главного героя
Покровских ворот

Сердце красавицы склонно к измене
И к перемене, как ветер мая.

Дж. Верди. *Риголетто*

Константы

Константы – это параметры (объекты) программы, которые *не могут изменять* своё *значение* в ходе выполнения программы.

Как мы уже знаем, все числа являются **числовыми константами**. Из чисел, знаков операций, идентификаторов некоторых стандартных функций и круглых скобок можно составлять **константные выражения**:

```
123
' Z '
Cos(3.14)
'string1 '
Chr(255)
```

Точно так же из любых символов мы можем составить **символьные** (один любой символ) и **строковые константы** (любое количество любых символов, в том числе и ни одного). Такие константы заключаются в *одинарные* (не двойные!) кавычки:

'G' 'R' '7' 'Ж' 'Э' – символьные константы
'' – пустая строка, в которой нет ни одного символа
'Delphi' 'G' 'Constant' 'Строковая константа' - строковые константы

Строковые и символьные константы можно соединять знаком +:

```
'Delphi'   + ' - '   +  '  это наш выбор! '
```

Значением этого выражения будет строка «Delphi - это наш выбор!»

`'205'` `'14.007'` `'2 + 2'` - тоже строковые константы! Их значениями являются именно строки, а не числа, поэтому значение выражение '2 + 2' вовсе не 4!

Если в строковой константе имеется одиночная кавычка, то её следует повторить дважды.

Кроме того, существуют **логические константы** *True* (соответствует значению *истина*), *False* (*ложь*), а также константа **nil** (*0*, используется как значение указателей).

Такие явные константы и константные выражения можно использовать в любых операторах и для этого их не нужно объявлять.

Именованные константы

Именованные константы **описываются** в разделе объявления констант модуля или подпрограммы. Он начинается зарезервированным словом **const**, за которым следуют объявления констант, состоящие из *имени*, *знака равенства* и *значения* константы. Завершается объявление точкой с запятой:

```
Const
  Имя= значение;
```

Здесь *значение* – это константа или константное выражение.

Например:

```
const
  num= 123;
  pi= 3.1415926;
  pi2= pi * 2;
  e12= 2.1728 / 2;
  str= 'string1';
  Flag= TRUE;
  Ch= 's';
```

Именованные константы получают своё значение при объявлении и потом изменить его не могут.

Константы принято описывать *раньше* переменных, так как в их объявлениях часто эти константы используются.

Разделов объявления констант и переменных может быть сколько угодно:

```
const
  num= 123;
  pi= 3.1415926;
var s: single;
  c1: char;
const c: integer= $ffff;
```

Вы должны были обратить внимание на то, что тип констант разный, но нигде не указан. Тогда каким же образом компилятор определяет его? – По виду константы или константного выражения! Действительные числа должны быть записаны с десятичной точкой или в научной нотации. Все остальные числа считаются целыми (десятичными и 16-ричными). Символьные (один символ) и строковые (любое количество символов) константы должны быть заключены в одиночные кавычки. Логические должны иметь значение *TRUE* или *FALSE*.

При компиляции программы значения всех именованных констант вычисляются и подставляются в исполняемом коде вместо их идентификаторов. Это немного увеличивает скорость работы программы, но главная цель применения именованных (а не явных) констант в том, чтобы сделать исходный код более понятным и облегчить модификацию программы в случае необходимости.

Пусть нам нужно написать программу, играющую в шашки. Доска имеет 8 вертикалей и 8 горизонталей. Если эти числа использовать в исходном коде, то эти восьмёрки невозможно будет отличить от других восьмёрок, которые тоже могут оказаться в коде. Да и вообще число 8 никак не указывает на то, что это число вертикалей на шахматной доске. Пусть, несмотря на это, нам удалось написать шашечную программу. Но - аппетит приходит во время еды: мы решили взяться за 100-клеточные шашки, а там доска побольше: 10 горизонталей и столько же вертикалей. Придётся просматривать (и очень внимательно!) весь код и исправлять восьмёрки на десятки.

Конечно, лучше сразу объявить две константы:

```
Const
  MAX_ROW = 8;
  MAX_COL = 8;
```

Их же следует использовать и в коде программы, тогда при переходе от русских шашек к международным достаточно будет изменить только 2 числа в самом начале программы, где их отыскать совсем нетрудно:

```
  MAX_ROW = 10;
  MAX_COL = 10;
```



Надеюсь, вы даже и не подумали, что после этого программа сумеет играть в международные шашки! Речь идёт только о размере доски, алгоритм игры изменится до неузнаваемости, но это уже ваша забота.

Переменные

Переменные – это параметры (объекты) программы, которые *могут* изменять своё значение в ходе выполнения программы.

Физически переменная представляет собой область памяти (её размер зависит от типа переменной; в этих ячейках памяти хранится значение переменной), связанную с её идентификатором.

Как и все другие объекты программы, до первого использования переменные должны быть объявлены. Переменные описываются в разделе объявления переменных модуля или подпрограммы. Он начинается зарезервированным словом **var** (от слова *variable - переменная*), за которым следуют объявления переменных.

При **объявлении** переменной указывается её *имя*, которое отделяется *двоеточием* от *типа*. Каждое объявление завершается *точкой с запятой*:

```
Var
  I,i,n: integer;
  Flag: boolean:
  Ch: char;
  X,y,z: single;
```

Переменные одного типа можно перечислять через запятую с последующим указанием их общего типа.

Типом переменной может быть:

- стандартный тип *Delphi* (*integer, single, char*...);
- - ранее определённый *пользователем* тип:

```
Type
  TNumber= 0..10;

Var
  Num: TNumber;
```

- - непосредственно вводимый при объявлении тип:

```
  Var
    Num: 0..10;
```

Хорошенько запомните: после объявления переменной её значение *не определено*. При выполнении программы для каждой переменной выделяется память, но в этих ячейках может оказаться всё что угодно, поэтому значение переменной будет меняться от одного запуска программы к другому. Это может привести к ошибкам: в некоторых случаях программа будет работать правильно, в других нет. Это значит, что все переменные должны быть *инициализированы*, то есть им должно быть присвоено какое-либо значение (конечно, не при объявлении):

```
var s: single;
begin
  s:= 3.1415926;
. . .
```

Никогда не используйте неинициализированные переменные!

Впрочем, *Delphi* следит за вами! После отладочного запуска программы проверяйте содержимое окна сообщений (Рис. УЗ.1).

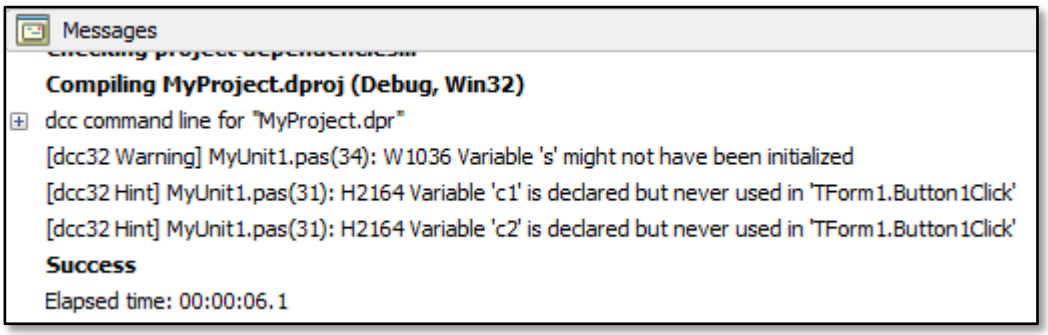


Рис. УЗ.1. Компилятор сообщает об ошибке!

В третьей строке компилятор *предупреждает* (такое легкомысленное отношение к переменным ошибкой не считается, но использовать её не следует!), что вы забыли задать значение переменной *s*.



Глобальные переменные инициализируются *автоматически* значением по умолчанию. Для числовых переменных - *нуль*, для строковых – *пустая строка*. Инициализация *локальных* переменных целиком лежит на совести программиста.

Два других сообщения – *подсказки*: вы объявили переменные, то не использовали их. Возможно, они лишние и их следует удалить.

```
const
```

```
  num= 123;
  pi= 3.1415926;
  Flag= TRUE;

var s: single;

begin
  s:= num;
  s:= pi;
  s:= flag;
. . .
```

Не забывайте также о **совместимости типов** переменных и присваиваемых им значений. В данном примере *действительной* переменной s можно присвоить значение целого или действительного типа, но никак не логического или строкового! В этом случае компилятор огорчит вас сообщением об ошибке и наотрез откажется запускать «дефективную» программу (Рис. У3.2).

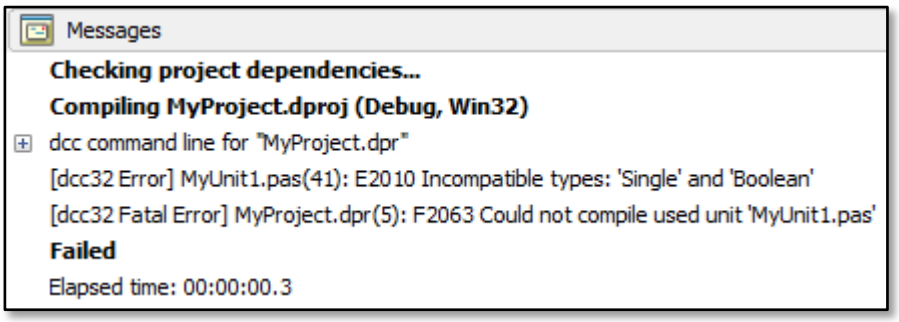


Рис. У3.2. А это уже серьёзная ошибка!

Глобальные и локальные переменные и константы

В программах используют глобальные и локальные переменные. По крайней мере, одна глобальная переменная существует помимо вашей воли, это переменная главной формы приложения:

```
var
  Form1: TForm1;
```

Мы сравним оба вида переменных, чтобы вы могли осознанно их выбирать в собственных программах.

Глобальные переменные объявляются в секции *interface* модуля. **Локальные** – в *процедурах* и *функциях*.

Глобальные переменные доступны (видимы) *во всех частях программы*, в том числе и в подпрограммах. **Локальные** переменные доступны только в той *подпрограмме*, в которой они объявлены.

Глобальные переменные живут «*вечно*» (точнее – пока не закрыта программа), для них выделяется постоянное место в памяти. **Локальные** переменные создаются *только на время работы подпрограммы*, после чего уничтожаются, а память, занимаемая ими, передаётся в общее пользование.

Глобальные переменные можно *инициализировать* (задавать им начальные значения) при объявлении. Делается это так:

```
Var
  Имя : тип = значение;
```

Здесь *значение* – это константа или константное выражение *того же типа*, что и переменная.

Например:

```
Var
  Seven: integer = 7;
  Pi: double= 3.1415926;
```

Локальные переменные инициализировать при объявлении *нельзя*.

Теперь вам, наверное, будет нетрудно сделать выбор в пользу глобальных и локальных переменных. Объявляйте *глобальные* переменные только тогда, когда вы планируете использовать их в разных подпрограммах или разных модулях, во всех остальных случаях используйте *локальные* переменные.

Глобальными и локальными могут быть и *именованные константы*. И те, и другие инициализируются при объявлении, в остальном они мало чем отличаются от соответствующих переменных, разве что глобальные константы используются в программах значительно чаще, чем локальные.

Типизированные константы

Типизированные константы также объявляются в разделе констант **const** и почти так же, как глобальные переменные:

```
const
  Имя : тип = значение;
```

Например:

```
const
  Seven: integer = 7;
  Pi: double= 3.1415926;
```

Как работать с компонентами

Из компонентов, как из кубиков, собирается вся визуальная (и не только) часть приложения.

Чтобы установить компонент на форме, просто дважды щёлкните на его значке в *Палитре компонентов*. Он появится в центре формы с установленными по умолчанию размерами (и другими свойствами). Его легко переместить на любое место, взявшись за него мышкой. Размеры визуальных компонентов можно изменить, потянув за один из «узелков», появляющихся по контуру компонента, когда он выделен (Рис. УЗ.3).



Рис. УЗ.3. Активный компонент имеет узелки

Вид *курсора* изменяется, когда он находится на габаритном узелке компонента, теперь можно переместить его влево или вправо (а также вверх-вниз), изменяя тем самым ширину компонента

Аналогично устанавливаются компоненты и на другой контейнер, например, компонент *TPanel*. Достаточно выделить контейнер и дважды щёлкнуть на значке компонента. Другой способ «расселения» компонентов таков: выделите значок компонента в палитре, щёлкнув на нём, а затем в нужном месте формы (или другого контейнера) нажмите левую кнопку мышки и растягивайте рамку компонента до нужных размеров. Как только вы отпустите мышкину кнопку, компонент появится на форме с заданными размерами (если вы поленитесь «тащиться» по форме, то размеры компонента будут установлены по умолчанию).

Чтобы установить на форме несколько одинаковых компонентов, есть два способа.

1. Дважды щёлкайте по значку компонента в *Палитре компонентов*. В этом случае все компоненты будут появляться в середине формы, перекрывая друг друга.

2. Нажмите клавишу *Shift*, а затем выделите компонент в *Палитре* (щёлкните на нём). Вокруг значка возникнет синяя рамка. Теперь щёлкайте мышкой в нужном месте формы. С каждым щелчком на контейнере будет появляться новый компонент. Дабы пресечь этот очаровательный процесс, выберите другой компонент или нажмите стрелку слева от значков компонентов.



Снять выделение с компонента можно, нажав кнопку *Deselect the current Tool Palette item* .

Палитра компонентов появляется, когда открыто окно *Конструктора формы*, а не *Редактора кода*!

На форме имеется сетка из точек – для более точной установки компонентов (Рис. У3.4).

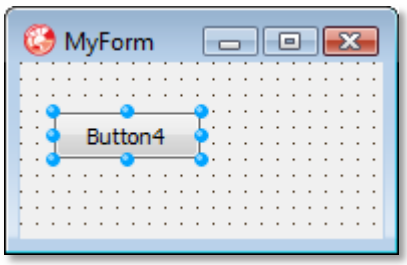


Рис. У3.4. Сетка для компонентов

Размеры сетки и привязку к ней вы можете задать в диалоговом окне **Options** (Рис. У3.5).

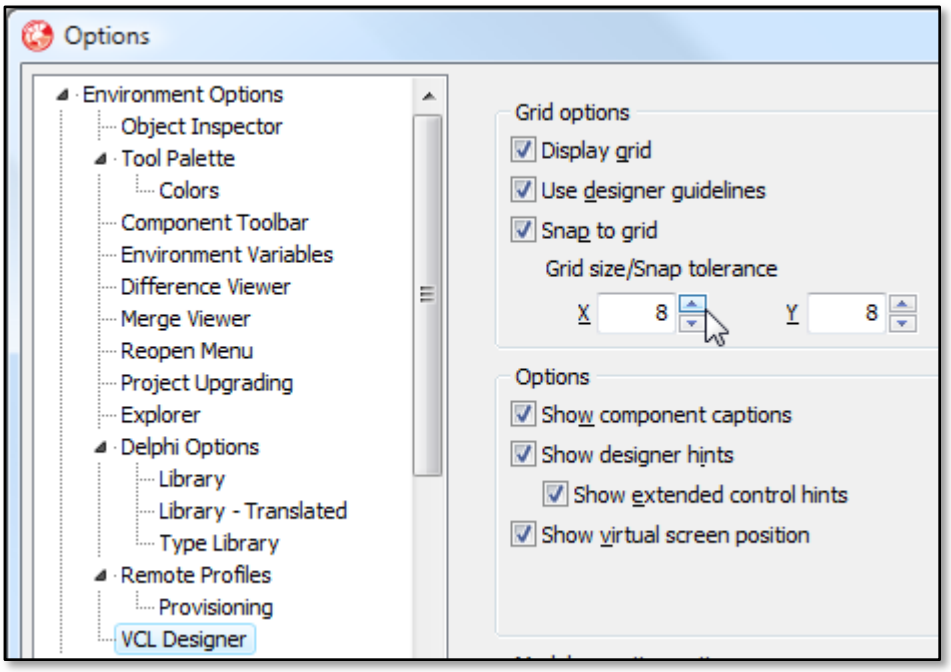


Рис. У3.5. Настраиваем размеры сетки

Оно появляется после выполнения команды *Главного меню* **Tools > Options...**

Если привязка к сетке включена (установлен флажок *Snap to grid*), то при перемещении компонентов мышкой они будут выравниваться по узелкам сетки. Оптимальные размеры сетки – 4 x 4 пикселя.

Для задания *точных* размеров и координат компонента пользуйтесь *Инспектором объектов*.

Семинар 3. «Здравствуй, мир!», или Первое приложение… усилий

Удобно устроившись в среде *Delphi*, мы просто обязаны бороться с гиподинамией мысли и упадническими настроениями. И тут как нельзя более кстати придутся умственно-физиологические упражнения по программированию без отягощений.

На предыдущем семинаре нам уже довелось «создать» *простейшее* приложение – оно само собой возникает при открытии нового проекта и состоит из «пустой» формы (на самом деле у неё есть все основные элементы стандартного окна *Windows*) и заготовки модуля формы. Это приложение можно запустить, и оно будет работать. Правда, на экране появится только прямоугольное окно, которое можно двигать, сворачивать, изменять его размеры и закрывать. Можно сказать, что простейшее приложение имеет свойства, совершенно необходимые любому приложению *Windows*, но ничего «лишнего».

Чтобы написать по-настоящему полезное приложение, нужно **разместить на форме компоненты**, обеспечивающие взаимодействие программы с пользователем (интерфейс), **установить их свойства** и **написать процедуры-обработчики**, которые будут выполняться при возникновении тех или иных событий в приложении (функциональная часть программы). Визуальная часть интерфейса приложения создаётся в *Конструкторе формы*, а его функциональная часть – в *Редакторе кода*.

В компьютерной литературе принято начинать программирование с создания приложения, выводящего на экран надпись *Hello, world!*. Мы не станем нарушать эту похвальную традицию, а только «русифицируем» сообщение. Давайте поздравим всех пользователей программы с нашим любимым праздником.

Запустите *Delphi* и начните новый проект.

В окне *Конструктора формы* появится унылая серая форма. Выкрасим её в «зимний», белый цвет. Для этого нажмите в *Инспекторе объектов* в строке *Color* кнопочку со стрелкой. Откроется список со стандартными цветами *Delphi* и *Windows*. Слева от названия цвета в этом списке имеется и образец (Рис. С3.1).

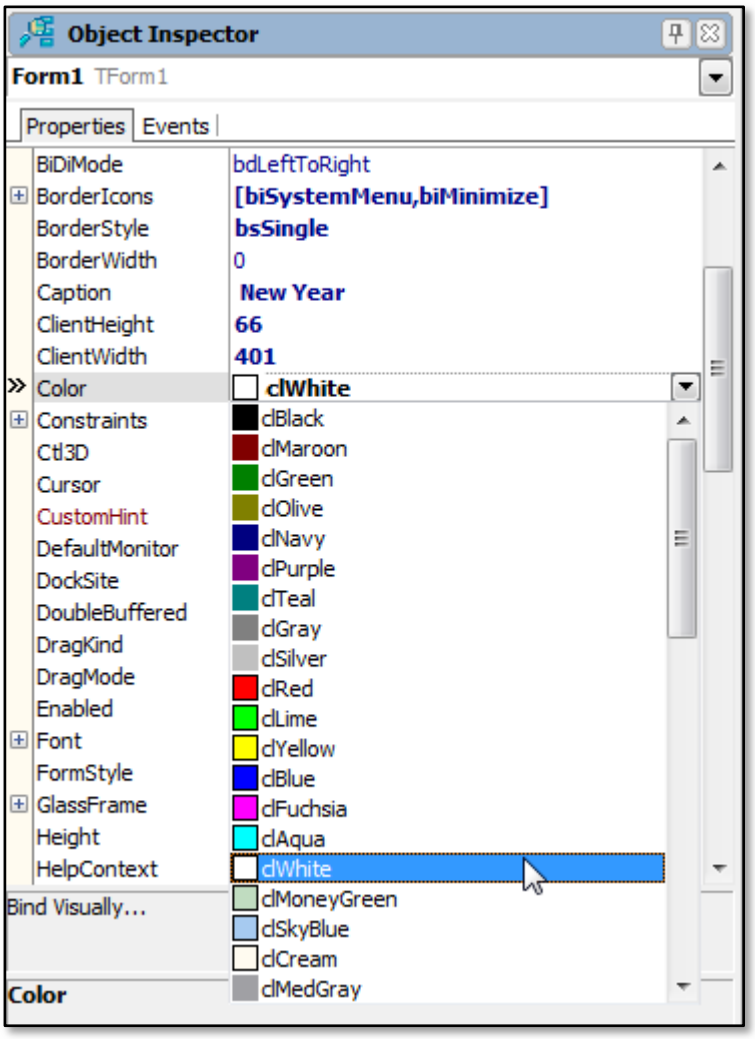
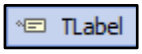


Рис. С3.1. Выбираем цвет из списка

Щёлкните мышкой на строке с белым цветом *(clWhite)* – и форма станет снежно-белой. Точно так же устанавливается цвет любого визуального компонента.

Вывести *надпись* на форму можно разными способами. Лучше всего воспользоваться компонентом *метка* **TLabel**, специально для этого предназначенным. Найдите в *Палитре компонентов* значок 

(он находится на странице *Standard*) и щёлкните на нём два раза. В середине формы появится этот компонент с надписью *Label1*. Измените в *Инспекторе объектов* значение свойства **Caption** этого компонента на *С Новым годом!* (просто введите эту строку в текстовом окошке справа от названия свойства) (Рис. С3.2).

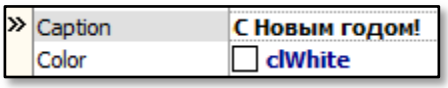


Рис. С3.2. Пишем надпись

Поздравление получилось слишком невзрачным, поэтому щёлкните на окошечке свойства метки **Font**, а затем на кнопке с тремя точками. Откроется диалоговое окно выбора свойств шрифта. Размер задайте побольше, например 40 (такого значения нет в списке размеров, поэтому введите нужное значение в окошечке над списком). Выберите шрифт *Times New Roman* (или любой другой по вкусу), установите атрибут *полужирный* и синий цвет. Теперь надпись имеет вполне праздничный вид. Оттащите её мышкой в левый верхний угол формы и размеры формы уменьшите так, чтобы не оставалось слишком много свободного места на форме (ухватите форму за нижний правый угол и перемещайте его вверх влево) (Рис. С3.3).



Рис. С3.3. Надпись готова!

Сохраните проект на диске в новой папке **NewYear** и запустите программу кнопкой  на *Панели инструментов*. Программа будет скомпилирована, и на экране появится форма с нашим приветствием (Рис. С3.4).



Рис. С3.4. Новогодний привет!

Как видите, она мало чем отличается от той, что мы видели в окне *Конструктора формы* (исчезла разметочная сетка и изменилась пиктограмма). С этим окном вы можете делать всё, что можно делать с любыми окнами *Windows*. И это при том, что мы не написали ещё ни одной строчки кода!

Закройте приложение - оно не предназначено для долгого любования. Попробуем сделать его «прикольным». Прикол будет незатейливым: при нажатии кнопки мыши на надписи она будет заменяться другой (предлагаю взять её из «негритянских» рассказов юмориста Лукинского), а при отпускании восстанавливаться.

Выделите на форме компонент *Label1* (просто щёлкните на нём мышкой) и в *Инспекторе объектов* перейдите на страницу **Events** (щёлкните на табуляторе сверху). Откроется список событий, на которые может реагировать наш компонент. Дважды щёлкните в окошке *OnMouseDown*, и в нём появится название процедуры, которая будет обрабатывать нажатие кнопки мыши на надписи (Рис. С3.5).

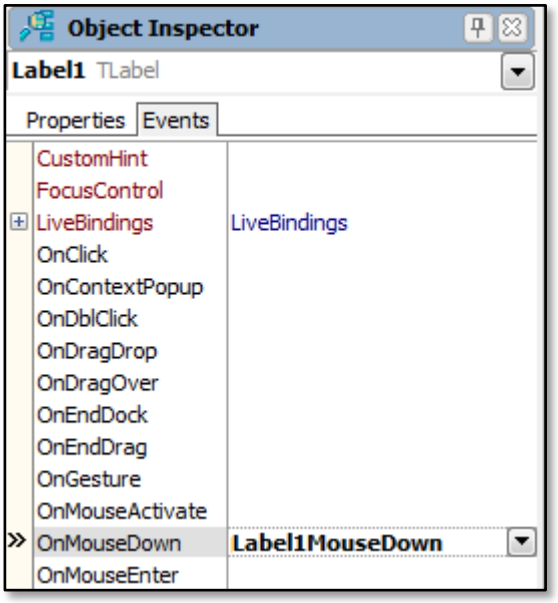


Рис. С3.5. События метки

В окне *Редактора кода* вы увидите заготовку для этой процедуры:

```
procedure TForm1.Label1MouseDown(Sender: TObject; Button: TMouse-
Button;
  Shift: TShiftState; X, Y: Integer);
begin

end;
```

Конечно, ничего из задуманного нами она делать не будет. Посему добавим ещё две строки кода:

```
procedure TForm1.Label1MouseDown(Sender: TObject; Button: TMouse-
Button;
  Shift: TShiftState; X, Y: Integer);
begin
  Label1.Font.Color:= clRed;
  Label1.Caption:= 'Пошёл на фиг!'
end;
```

В первой строке мы устанавливаем красный цвет шрифта, чтобы показать серьёзность наших намерений, во второй – задаётся суровая проза жизни.

Чтобы ввести эти строчки, наберите название компонента **с точкой** – *Label1.*, подождите, пока появится список со свойствами этого компонента, выберите нужный и нажмите клавишу *Enter* (или дважды щёлкните на нём мышкой) (Рис. С3.6).

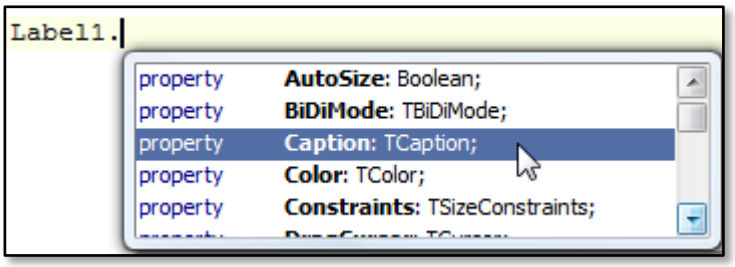


Рис. С3.6. Подсказка в действии

Действуя аналогично, создайте заготовку процедуры, которая будет обрабатывать **отпускание кнопки мыши**:

```
procedure TForm1.Label1MouseUp(Sender: TObject; Button: TMouseBut-
ton;
  Shift: TShiftState; X, Y: Integer);
begin

end;
```

Чтобы не переписывать весь код заново, скопируйте его в заготовку второй процедуры.



Чтобы скопировать фрагмент текста, сначала выделите его. Для этого установите мышкой курсор в начало первой строки, нажмите левую кнопку мышки и ведите её в конец последней строки. Теперь отпустите и снова нажмите левую кнопку мыши на выделенном фрагменте кода и, удерживая клавишу **Ctrl**, переместите курсор в нужное место. Если не нажать клавишу **Ctrl**, то фрагмент **будет перемещён** из одного места на другое.

Выделить часть текста можно и клавишами со стрелками, удерживая клавишу **Shift**.

Осталось слегка подредактировать текст:

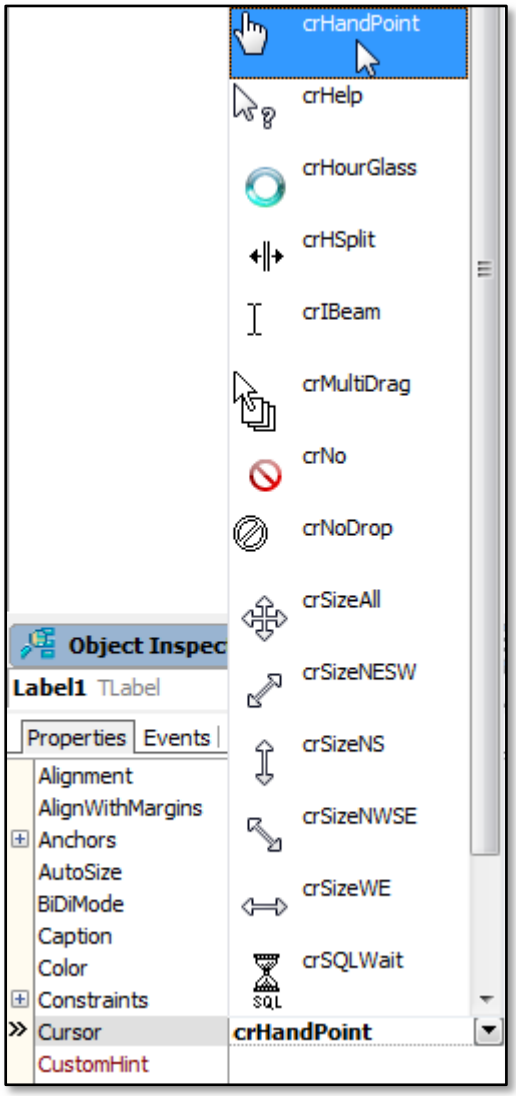
```
procedure TForm1.Label1MouseUp(Sender: TObject; Button: TMouseBut-
ton;
  Shift: TShiftState; X, Y: Integer);
begin
  Label1.Font.Color:= clBlue;
```

```
  Label1.Caption:= 'С Новым годом!'
end;
```

Опять запустите программу. Появится всё то же окно с банальной уже надписью. Зато теперь вы можете «поприкалываться», нажимая кнопку мыши на ней (Рис. С3.7).



Рис. С3.7. Безобразие!



Согласитесь, жить стало веселей. И развлекаемся мы с помощью своих же ручек! А если они у вас шаловливые, то можно заготовить на форме множество таких фраз и подсунуть их своим товарищам, близким и родственникам.

Вот только как они узнают о ваших злонамеренных кознях? Значит, их нужно подвести к необдуманным поступкам. И *Delphi* предоставляет немало средств начинающим «хулиганам». Во-первых, вы можете установить для каждого визуального компонента свой собственный курсор, который будет появляться, когда мышка бегает по нему. В *Инспекторе объектов* откройте список доступных курсоров и выберите нужный (Рис. С3.8).

Рис. С3.8. Выбираем курсор для метки

Он будет явно указывать, что на этом компоненте нужно нажать кнопку мыши. Впрочем, вы не должны обольщаться: не все поймут вас правильно.

Тогда переходите к более сильнодействующим средствам. В свойстве **Hint** метки напишите *Нажми кнопку*, а свойство **ShowHint** установите в *True* (Рис. С3.9).

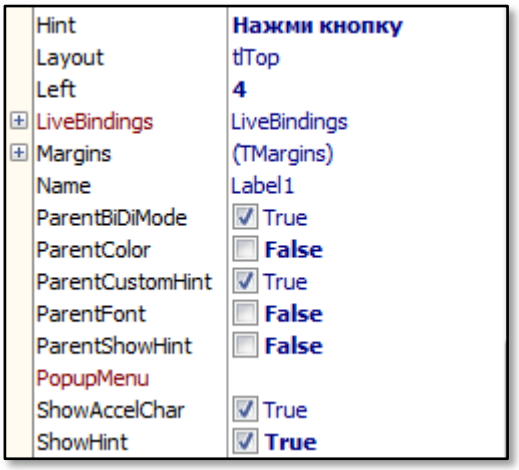


Рис. С3.9. Подсказка для курсора

Эта подсказка появится, когда мышка замешкается на компоненте. Если и это не поможет, то оставьте свои затеи до лучших времён, когда вы изучите такие возможности *Delphi*, которые позволят вам подшучивать над кем угодно в самой обидной форме (если вы, конечно, так и не найдёте себе более достойных занятий). Пока же допишите пару строк кода, улучшающих действие нашей программы:

```
procedure TForm1.Label1MouseDown(Sender: TObject; Button: TMouse-
Button;
  Shift: TShiftState; X, Y: Integer);
begin
  Label1.Font.Color:= clRed;
  Label1.Caption:= 'Пошёл на фиг!';
  Label1.Hint:= 'Отпусти - больно же!'
end;

procedure TForm1.Label1MouseUp(Sender: TObject; Button: TMouseBut-
ton;
  Shift: TShiftState; X, Y: Integer);
begin
  Label1.Font.Color:= clBlue;
  Label1.Caption:= 'С Новым годом!';
  Label1.Hint:= 'Нажми кнопку!'
end;
```

Lazarus тоже годится для милых розыгрышей (Рис. С3.10).



Рис. С3.10. Новогодний привет от *Lazarus*‘а



Исходный код программы находится в папках **NewYear** и **NewYear-Lazarus**.

Самостоятельная работа



Поскольку это была наша первая серьёзная программа, то названия компонентов не изменялись – чтобы не вводить вас в заблуждение, если вы захотите создать программу с нуля. Замените стандартные имена для формы и метки более осмысленными. И в следующих примерах не забывайте делать то же самое!

Урок 4. Процедуры и функции, или Разделяй и властвуй

- Милая, на сколько кусков разрезать тортик,
на четыре или на шесть?
- Режь на четыре, шесть я не съем!

Анекдотичный пример женской логики

- Резать к чёртовой матери!

Решительная фраза врачихи
из к/ф *Покровские ворота*

Во всех современных языках программирования большие программы делятся на отдельные части (*модули*), которые не зависят друг от друга, поэтому их легче отлаживать, чем программу необъятных размеров. Другое преимущество модульного программирования состоит в реализации важнейшего принципа: «Никогда не пишите дважды один и тот же код!». Отлаженные модули могут быть легко присоединены к любой программе и будут работать там «без шума и пыли».

В отдельные модули обычно помещают процедуры и функции общего назначения, которые могут потребоваться во многих программах. Например, процедура *Sound* в наших программах воспроизводит звуковые файлы, функция *GetRandom* генерирует случайные числа в заданном интервале… У вас по мере написания программ также накопится множество таких полезных подпрограмм, которые есть смысл сгруппировать вместе и перенести в отдельный модуль.

Каждый класс (о классах речь пойдёт дальше) тоже желательно упрятать в отдельный модуль – как для удобства отладки и повторного использования, так и для сокрытия деталей реализации класса от посторонней публики.

Модульный принцип позволяет разрабатывать сложную программу по частям, что гораздо эффективнее и быстрее, чем «монолитное» приложение. Этот же принцип действует и на уровне каждого модуля, который и сам сложен из отдельных «кирпичей» программного кода - подпрограммам, которые в *Delphi* называются **процедурами** и **функциями**.

Подпрограмма – *это часть программы, выполняющая некоторую последовательность действий (операторов) для решения какой-либо частной*

задачи. Например, подпрограммы могут воспроизводить звук, генерировать числа, вычислять значения, делать паузу в программе – да что угодно! Обычно подпрограмма служит для разбивки большого куска кода на отдельные части, удобные для написания и отладки, а также для повторного их использования в разных частях программы. По структуре подпрограммы очень похожи на программы в миниатюре.

Примером модульного подхода может служить книга: она состоит из отдельных частей (томов), разбитых на главы. Сами главы разбиты на параграфы, абзацы, строки. А любой механизм состоит из отдельных узлов, а те, в свою очередь, - из деталей. И даже венец природы являет собой лучший образец модульного мышления Создателя нашего: человек – это совокупность отдельных органов (пищеварения, дыхания, кровообращения, слуха и других), что и позволяет лечить болезни узким специалистам (увы, не всегда успешно именно из-за своей узкой специализации).

Процедуры и функции очень похожи друг на друга, поэтому мы будем изучать их параллельно. Кстати говоря, в языке С процедур нет – только функции. Здесь роль процедур играют функции, не возвращающие значений. Именно этим принципиально отличаются процедуры и функции: процедура только выполняет свои операторы, а функция ещё и возвращает некоторое значение (вычисленное в ней) в ту точку программного кода, откуда она была вызвана.

Прежде чем использовать подпрограммы, их нужно объявить в разделе *interface* модуля (после объявления переменных) и описать в разделе *implementation*. Объявление представляет собой заголовок процедуры или функции, которые, естественно отличаются друг от друга.

Объявление процедуры

Заголовок процедуры состоит из зарезервированного слова **procedure**, за которым следует *имя* процедуры, а затем в круглых скобках через точку с запятой перечисляются *формальные параметры* с указанием их типа:

```
procedure имя (список формальных параметров);
```

Формальные параметры могут и отсутствовать, тогда заголовок процедуры будет ещё короче:

```
procedure имя;
```

Гораздо чаще встречаются процедуры с параметрами, поскольку они позволяют легко настраивать процедуру на конкретные действия, определяемые значениями параметров.

Объявление функции

Заголовок функции состоит из зарезервированного слова **function**, за которым следует *имя* функции, а затем в круглых скобках через точку с запятой перечисляются *формальные параметры* с указанием их типа. В отличие от процедуры функция *возвращает значение*. *Тип* этого значения и самой функции указывается после двоеточия:

```
function имя (список формальных параметров): тип функции;
```

Формальные параметры могут и отсутствовать, тогда заголовок функции будет короче:

```
function имя: тип функции;
```

Описание процедуры

Описание процедуры начинается с её заголовка, после чего могут идти (некоторые из них или все могут отсутствовать) точно такие же *разделы*, что и в основной программе: объявления локальных *констант* (*const*), *типов* (*type*), *переменных* (*var*), *меток* (*label*):

```
procedure Sound(name: PChar);
var s: PChar;
begin
  //сформировать имя файла:
  s:= PChar('wav\'+ name + '.wav');
  sndPlaySound(s, SND_ASYNC or SND_FILENAME);
end;
```

Или

```
procedure TfrmMain.LoadFile; // ← Заголовок процедуры
var                          // ← Объявления локальных переменных
  s : String;
  F: TextFile;
  i,j: integer;
  s1,s2,s3,s4,s5,s6,s7: string;
  n,m:integer;
begin       // ← Операторные скобки
```

```
. . .          // ← Тело процедуры

end;           // ← Конец процедуры
```

После объявления констант и переменных в *операторных скобках* **begin – end;** помещается *тело* процедуры, которое представляет собой операторы, выполняющие действия, для которых, собственно, процедура и предназначена.

Описание функции

Описание функции ничем не отличается от описания процедуры, но в теле функции возвращаемое значение должно быть присвоено *идентификатору* функции или предопределённой *переменной* **result**.



В последних версиях *Delphi* возвращать значение можно с помощью процедуры **exit(значение)**.

Для примера напишем функцию, вычисляющую факториал числа *n*.

```
function fact(n: integer): integer; // ← Заголовок функции
var                  // ← Объявления локальных переменных
  f,i: integer;
begin
  f:= n;
  for i:= n-1 downto 2 do f:= f*i;

  fact:= f;  // ← Возвращаемое значение
  //или
  //result:= f;
  //или
  //exit(f);
end;           // ← Конец функции
```



Если вы не вернули значение функции ни одним из описанных способов, то её значение будет **неопределённым**!

В процедурах и функциях могут быть объявлены и описаны другие процедуры и функции, которые являются локальными, то есть могут использоваться только там, где они объявлены.

Вызов процедур и функций

Для вызова процедуры из основной программы или другой процедуры (функции) необходимо записать оператор, составленный из *имени* процедуры (функции) и списка *фактических параметров*, которые должны совпадать по количеству и типу с формальными параметрами.

При вызове *процедуры* ей передаются указанные параметры, которые становятся локальными переменными процедуры, к ним добавляются локальные переменные самой процедуры, а затем выполняются операторы, записанные в теле процедуры. Как только тело процедуры выполнено, управление передаётся тому оператору программы, который следует за вызовом процедуры.

Например, для попискивания вредного персонажа игры мы можем в нужном месте обратиться за помощью к нашей процедуре **`Sound(name: String);`** по извлечению звуков:

```
Sound('pisk');
```

Естественно, вместо формального параметра *name*, мы должны указать имя реально существующего звукового файла на диске, который и содержит нужное нам попискивание.

Вызов *функции*, если возвращаемое ею значение не используется, ничем не отличается от вызова процедуры:

```
fact(10);
```

В этом случае функция вычислит факториал числа 10, но оно будет проигнорировано программой. Обычно поступают более культурно, и работа функции не пропадает даром. Для этого возвращаемое значение присваивают переменной того же типа (или совместимого), что и функция:

```
Var n: integer;

. . .

n:= fact(10);
```

После вызова функции переменная *n* примет значение 10! (посчитайте сами, сколько это будет!).

Вызов функции называют также *обращением* к функции.

Фактические параметры перечисляются через запятую.

Локальные и глобальные подпрограммы

Все переменные, константы, типы, функции и переменные, объявленные в подпрограмме, являются **локальными** (то есть *местными*). Они «видны» только в подпрограмме и доступны только в её теле. После завершения вызова подпрограммы (все операторы выполнены) они прекращают своё существование.

Если в подпрограмме используются *глобальные* идентификаторы, то тут имеется одна тонкость: если в подпрограмме объявлен точно такой же идентификатор, что и в главной программе (например, две переменные имеют одно и то же имя), то в подпрограмме будет использоваться именно *локальный* элемент, а глобальный подпрограмма не увидит вообще и даже знать о нём ничего не будет!

Формальные и фактические параметры

Формальные параметры записываются в *заголовке* подпрограммы. Они определяют количество, последовательность и тип параметров, которые передаются в подпрограмму и в ней используются. Для каждого формального параметра указывается его *имя* и - через *двоеточие* – *тип*. Друг от друга в списке параметры отделяются *точкой с запятой*. Параметры одного типа можно перечислять через *запятую*, а затем указывать их общий тип.

Все формальные параметры можно разделить на:

1. *параметры-значения;*

2. *параметры-переменные;*

Параметры-значения

Все формальные параметры, которые перечислены в заголовке подпрограммы подобным образом

```
Procedure one(a,b,c: integer);
Var n: integer;
Begin
  n:= a * b*c;
```

```
End;
```

называются **параметрами-значениями**. В данной процедуре это *a,b,c*. Такой способ передачи параметров в процедуру называется **передачей по значению**.

Фактическим параметром, соответствующим формальному, может быть константа, переменная или выражение того же типа, что и формальный параметр.

Обращение к процедуре one может быть таким:

```
const WIDTH=10;
var height: integer;
 . . .
height:= 20;

one(WIDTH, height, 2+3);
```

Если процедура имеет параметры-значения, то для них создаются *локальные переменные* с именем и типом формальных параметров, после чего в них последовательно копируются значения фактических параметров.

Первый формальный параметр *a* станет локальной переменной *a* целого типа со значением первого фактического параметра *WIDTH*, то есть 10. Второй – локальной переменной *b* со значением переменной *height*, то есть 20. И, наконец, третий - локальной переменной *c* со значение 5.

После этого выполняется тело процедуры:

```
n:= 10 * 20 * 5; (n:= a * b * c;)
```

Так как *n* это локальная переменная, то толку от неё не будет никакого – она умрёт сразу после вычисления значения. В данном случае можно использовать глобальную переменную *n*, тогда после выполнения процедуры *one* ей будет присвоено значение 1000.



Если имена глобальной и локальной переменных совпадают, то в процедуре будет использована *локальная*, поэтому мы должны закомментировать строчку

```
//Var n: integer;
```



Хотелось бы предостеречь от использования *глобальных* переменных в процедурах. Для отладки и использования подпрограмм го-

раздо удобнее и надёжнее включать все параметры в список при её вызове. Так легче следить за соответствием типов формальных и фактических параметров, а также мы добиться определённой автономности подпрограмм. Подпрограммы, как и классы, позволяют скрывать детали реализации от пользователя, которого должен интересовать результат действия подпрограммы, а не сам код, выполняющий эти действия.

Так как в процедуру передаются только *значения* переменных, то соответствующие им локальные переменные не могут изменить *их* значения. Например, в процедуре *one* мы присвоим переменной *b* значение 1000:

```
b:= 1000;
```

При этом переменная **height**, *значение* которой копируется в переменную *b*, не изменится, то есть останется равным 20. Это главное преимущество передачи параметров по значению: подпрограмма получает в своё распоряжение не переменную, а только её значение, поэтому ни одна переменная не пострадает!

Но есть и *недостатки*. Для хранения локальных переменных требуется дополнительная память, а на их создание и копирование в них значений передаваемых параметров нужно время. Впрочем, для простых типов данных эти затраты очень невелики, а вот для структур, классов, массивов такие действия могут обернуться большими потерями производительности. В некоторых случаях может оказаться недостатком и неспособность процедур изменять значение внешней переменной (ещё недавно мы причислили эту особенность процедур с параметрами-значениями к достоинствам, но такова жизнь: у каждой медали две стороны). Конечно, можно воспользоваться глобальными переменными, но их лучше не упоминать всуе. Но выход всё-таки есть!

Параметры-переменные

Достаточно в списке формальных параметров перед идентификатором поставить ключевое слово **var**, как мы получаем эту переменную в полное своё распоряжение и можем изменять её значение точно так же, как если бы она была глобальной.

```
Procedure two(a,b,c: integer; var n: integer);
Begin
  n:= a * b * c;
End;
```

Слово **var** действует только в пределах одной группы переменных, то есть до ближайшей точки с запятой.

В заголовке процедуры формальному параметру *n* предшествует слово **var**, значит это *параметр-переменная*. Такой способ передачи параметров называется **передачей по ссылке**. Для параметров-переменных локальные переменные не создаются, а вместо значения фактического параметра передаётся адрес переменной, по которому она хранится в памяти. В этом случае в процедуре мы имеем дело именно с той переменной, которая используется при вызове процедуры как фактический параметр, поэтому все изменения формальной переменной n отразятся и на фактическом параметре.
Например, после вызова процедуры

```
Var res: integer;
. . .

two(WIDTH, height, 2+3, res);
```

переменная n:= a * b * c; примет значение 1000. Но этот идентификатор *n* – всего лишь псевдоним переменной *res*, то есть это значение будет присвоено именно переменной *res*. Итак, в данном случае *n* и *res* – одна и та же переменная, хотя в процедуре она и называется иначе (но вы можете назвать их и одинаково).

Этот способ передачи параметров обычно используется для передачи в процедуру переменных большого размера – записи, классы, массивы. Такой способ гораздо быстрее и экономичнее, поскольку не нужно выделять под переменные память и копировать туда их значения. В процедуру передаются только их адреса, а это очень быстрая операция. Но, как говорится, недостатки – это продолжение наших достоинств: можно случайно изменить в процедуре значение переменной, а потом долго искать причину неверной работы программы.

Общие приёмы работы с окнами/панелями

В *Delphi* все окна жёстко скреплены друг с другом, в отличие от предыдущих версий *Delphi*, где каждое окно жило своей собственной жизнью. Но вы легко можете вернуться в прошлое, изменив раскладку интерфейса. Для этого в панели **Desktop** (находится в строке Главного меню), в раскрывающемся списке выберите строку *Classic Unlocked* (Рис. У4.1).

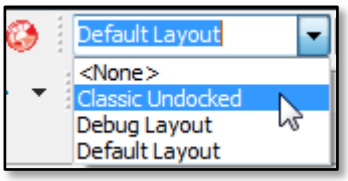


Рис. У4.1. Освобождаем окна

Теперь вы можете перемещать каждое окно по своему усмотрению. Однако *Главное окно* лучше не трогать. А уж если вы его неосторожно сдвинули, то просто дважды щёлкните на заголовке, и оно вернётся на своё законное место. Остальные окна можно передвигать, раскрывать (максимизировать), сворачивать (минимизировать) и закрывать – как и любые другие окна *Windows*.

По умолчанию раскладка интерфейса - **Default Layout**. Чтобы переключиться на неё, выберите в списке соответствующую строку (Рис. У4.2).

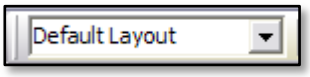


Рис. У4.2. Раскладка ИСР

Удалить какое-либо окно с экрана можно, нажав кнопку с крестиком в заголовке этого окна. Если через некоторое время окно вам вновь понадобится, то вы можете вернуть его из *Главного меню* **View** (Рис. У4.3).

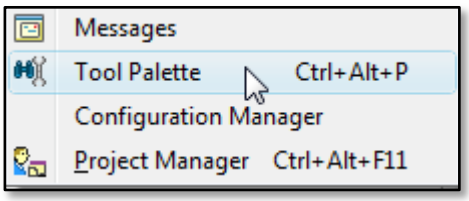


Рис. У4.3. Возвращаеи *Палитру инструментов* на место

Окна в *Delphi* имеют ещё одну кнопку – с кнопкой (извините за тавтологию!), которая делает окно скользящим – оно самостоятельно задвигается за границу экрана, оставляя только закладку 

как напоминание о себе. При этом вертикальная кнопка становится горизонтальной - . Теперь достаточно навести курсор на закладку, как окно послушно выдвигается на своё место. Уберите курсор – и окно вновь исчезает с экрана, освобождая место. Чтобы лишить окно способности постоянно убегать с экрана, вторично щёлкните по кнопке , она снова займёт вертикальное положение .

Окна в *Delphi* являются *прикрепляемыми* (*Dock-окна*). Чтобы изменить местоположение какого-либо окна на экране, возьмите его за заголовок мышкой и передвиньте на другое место. Окно автоматически присоединится сбоку или сверху (снизу) к соседнему окну.

Вы можете не только изменять взаимное положение окон, но и их *размеры*. Для этого установите курсор на вертикальной или горизонтальной границе окна, курсор изменит свой вид на, соответственно,  или . Теперь передвигайте границу окна в нужную сторону.

Семинар 4. Контрольное взвешивание, или Веское приложение

Хорошего человека должно быть много.

Коварное заблуждение

Поставим перед собой более серьёзную задачу, чем на предыдущем семинаре: поможем себе и своим близким приблизиться к идеалу. Пока только в смысле веса.

Некоторые, так называемые, специалисты считают, что здоровый человек должен иметь идеальный вес, который можно определить по формуле

P = (3 x A - 450 + B) x 0,25 + 45 - для мужчин,
P = (3 x A - 450 + B) x 0,225 + 40,4 - для женщин,

где под загадочными буквами **A** и **B** скрываются *рост* (в сантиметрах) и *возраст* (в годах).

Попробуем согласиться с ними и напишем чрезвычайно полезную для дома и семьи программу, которая позволит вам и всем окружающим оценить свои материальные (опять же только в смысле веса) достоинства.

Какой же должна быть программа, основанная на этих незамысловатых формулах? Ясно, что главную роль в ней должна играть **функция**, вычисляющая идеальный вес по заданным параметрам – полу «подопытного индивидуума», его возрасту и росту. С написанием функции проблем не будет, ведь достаточно перевести на «дельфийский» язык обычные математические выражения. Чтобы функция «знала» исходные величины, введём переменные, в которых будем хранить текущие значения возраста и роста.

Нам нужно позаботиться о том, чтобы пользователь мог быстро и удобно передавать в программу необходимые величины. Так как пол может быть либо мужской, либо женский, то логично поставить два переключателя: нажмёшь один – выключится другой, и наоборот. И в программе нетрудно будет узнать, какой переключатель включен в настоящий момент.

Возраст и рост могут принимать большое количество последовательных целых значений. Их можно вводить по-разному. Выберем наиболее удобный способ. Поставим два скроллера (он же ползунок, он же полоса прокрутки), которые обычно используются для прокручивания окон. Перемещая движок, мы сможем установить нужное значение, которое будем

выводить на экран с помощью хорошо нам уже известного компонента – метки.

Вот, пожалуй, и вся премудрость. Осталось только воплотить мечту в реальность.

Откроем новый проект.

В заголовке формы напишем *Вес*, то есть свойству **Caption** присвоим значение 'Вес'.

Из свойств **BorderIcons** мы оставим только *biSystemMenu* и *biMinimize*, а для **BorderStyle** выберем *bsSingle*, чтобы размеры формы нельзя было изменять в работающей программе.

Для выбора пола «клиента» установим компонент **RadioGroup** со страницы *Standard Палитры компонентов* (щёлкните дважды на значке TGroupBox). Он позволяет очень быстро сформировать набор переключателей. В его рамке выведем соответствующую надпись жирным шрифтом (**Caption** = ' Пол ', **Font.Style** = [fsBold]), установим форму курсора в виде руки (**Cursor** = crHandPoint) – прозрачный намёк на то, что здесь нужно приложить руки. Дважды щёлкните на окошке свойства **Items** и в раскрывшемся редакторе **String List Editor** наберите две строки (Рис. С4.1).

```
- мужской
- женский
```

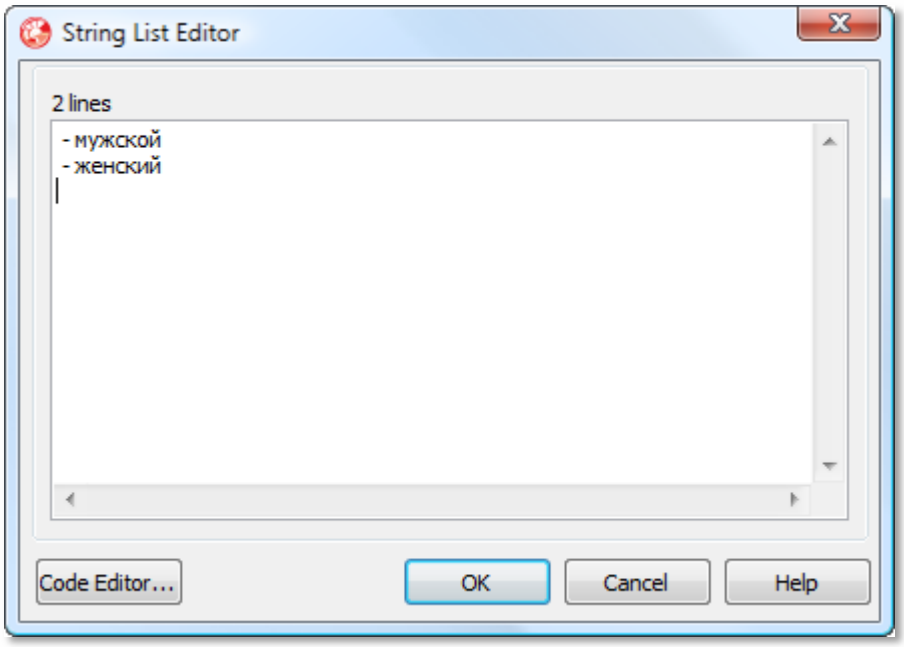


Рис. С4.1. Редактор строк

Нажмите кнопку **ОК** – и у компонента появятся два переключателя с этими надписями. Свойство **ItemIndex** показывает, какой из переключателей выбран (первоначально его значение равно **0** – по умолчанию пол предполагается мужской; суфражистки, феминистки и подкаблучники могут изменить последовательность строк или установить **ItemIndex=1**).

Чтобы придать интерфейсу элегантный вид, установим на форму компонент **Panel** TPanel, а уже на нём разместим два **скроллера** TScrollBar. Первый (**Name** = *sbAge*) будет отвечать за *возраст*, второй (**Name** = *sbRost*) - за *рост*. Выберем по изложенным выше причинам для обоих ползунков курсор в виде руки (**Cursor** = *crHandPoint*). Дополнительно сделаем подсказки (**Hint** = *'Возраст'* и **Hint** = *'Рост'*, **ShowHint** = *True*). Будем считать, что *возраст* изменяется в пределах от 0 до 100 (**Min** = 0, **Max** = 100, значение по умолчанию **Position** = 20), а *рост* от 60 до 220 (**Min** = 60, **Max** = 220, значение по умолчанию **Position** = 160).

Справа от скроллеров разместим две **метки** (**lblAge** и **lblRost**), в которые будем выводить текущие значения возраста и веса. Шрифт установим покрупнее (**Font.Height** = -16), потолще (**Font.Style** = [fsBold]) и поярче (**Font.Color** = clRed). В каждой метке напишем значения по умолчанию (**Caption** = '20' и **Caption** = '160'). Чтобы числа всегда были расположены строго друг под другом, установите **Alignment** = *taRightJustify* и **AutoSize** = *False*. Осталось справа от них поместить две метки, поясняющие назначение скроллеров. Забегая немного вперёд, мы должны увидеть такую картину (Рис. С4.2).

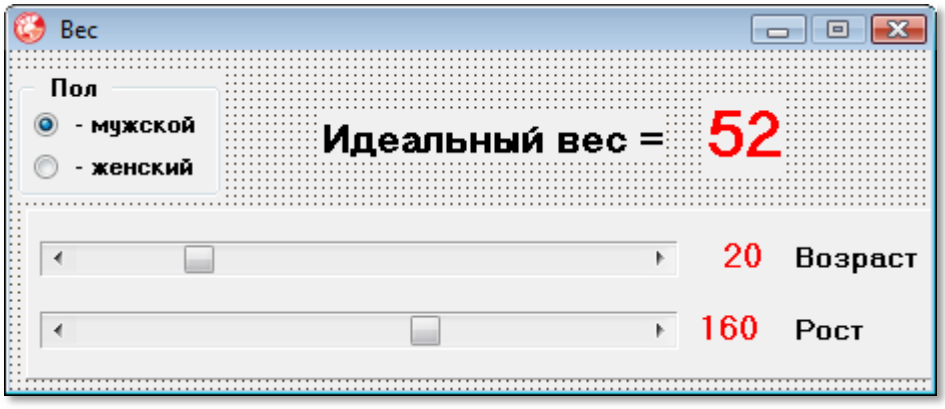


Рис. С4.2. Весовой интерфейс

Осталось самое «главное» - установить на форме пару **меток** для вывода вычисленного значения идеального веса:

Метка **lblVes**: **Caption** = '52', **Font.Color** = clRed, **Font.Height** = -32, **Font.Style** = [fsBold]

Метка **Label2**: **Caption** = 'Идеальный вес =', **Font.Height** = -19, **Font.Style** = [fsBold].

С визуальной частью интерфейса мы справились неплохо, можно браться за функциональную.

Как мы уже выяснили, нам потребуются две *переменные* для хранения введённых значений возраста и веса. Объявим их сразу перед разделом *implementation*:

```
var
  Form1: TForm1;
  age, rost: integer;
```

Рядышком объявим *функцию*, которая по «учёным» формулам будет вычислять вес:

```
 private
    { Private declarations }
    function Ves: single;
```

Что до самой формулы, то исключительно вручную наберите её код:

```
//ВЫЧИСЛИТЬ ИДЕАЛЬНЫЙ ВЕС
function TForm1.Ves: single;
begin
  if RadioGroup1.ItemIndex= 0 then //мужской пол
    Result:= (rost*3 - 450 + age) * 0.25 + 45
  else                                //женский пол
    Result:= (rost*3 - 450 + age) * 0.225 + 40.4;
  //скорректировать отрицательные значения:
  if Result< 0 then Result:= 4;
end;
```

Действует-злодействует (сказочное выражение) она так. По номеру нажатого переключателя (отсчёт начинается с нуля!) определяется пол «персонажа», в соответствии с которым выбирается одна из двух формул. Последняя строчка заменяет отрицательное значение на более осмысленное. Это вызвано тем, что данные формулы не универсальны, то есть действительны только для разумных сочетаний параметров, поэтому идеальный вес Кощея Бессмертного или дядьки Черномора по ней вычислять нельзя.

Мы поступим правильно, если при запуске программы сразу же выведем значение веса по установкам скроллеров (в *Инспекторе объектов* перей-

дите на страницу **Events** и дважды щёлкните на окошечке события **OnCreate**, а затем в заготовке процедуры обработки напишите следующие строки):

```
//СОЗДАТЬ ФОРМУ
procedure TForm1.FormCreate(Sender: TObject);
begin
  //снять значения со скроллеров:
  age:= sbAge.Position;
  Rost:= sbRost.Position;
  //вычислить вес:
  lblVes.Caption:= inttostr(round(ves))
end;
```

Конечно, можно сразу же во всех метках вывести согласованные значения, но если вы потом измените одно из них, то либо придётся исправлять и вес, либо при запуске программы вес не будет соответствовать положениям движков. Ничего страшного в этом нет – вряд ли вы угадаете возраст и рост пользователя, он всё равно будет двигать ползунки, - но лучше приучить себя всё доводить до ума.

Запустив эту программу, «клиент» будет вводить свои данные, то есть пол, возраст и вес. Чтобы не заставлять его нажимать после этого кнопку для вычисления веса, будем «публиковать» его сразу после каждого изменения параметров. Например, пользователь сменил пол (к счастью, только в программе), и тут же выводится новое значение веса:

```
//ВЫБРАТЬ ПОЛ
procedure TForm1.RadioGroup1Click(Sender: TObject);
begin
  //вычислить вес:
  lblVes.Caption:= inttostr(round(ves))
end;
```

Аналогично поступаем и при изменении положения ползунка скроллера:

```
//ЗАДАТЬ ВОЗРАСТ
procedure TForm1.sbAgeChange(Sender: TObject);
begin
  //возраст:
  age:= sbAge.Position;
  //вывести значение на экран:
  lblAge.Caption:= inttostr(age);
  //вычислить вес:
  lblVes.Caption:= inttostr(round(ves))
```

```
end;

//ЗАДАТЬ РОСТ
procedure TForm1.sbRostChange(Sender: TObject);
begin
  //рост:
  Rost:= sbRost.Position;
  //вывести значение на экран:
  lblRost.Caption:= inttostr(Rost);
  //вычислить вес:
  lblVes.Caption:= inttostr(round(ves))
end;
```

Значения возраста и роста определяем по положению ползунка (свойство **Position** компонента *ScrollBar*), вычисляем вес, округляем его до целого числа (формулы, естественно, не настолько точны, чтобы рассчитывать ещё и граммы), которое преобразуем в строку и выводим на экран.

Перед тем как предлагать эту программу другим людям, очень полезно испытать её на себе. Запустите программу, подвигайте ползунки, оцените все достоинства и недостатки интерфейса (Рис. С4.3). И – непременно! – постарайтесь её улучшить.

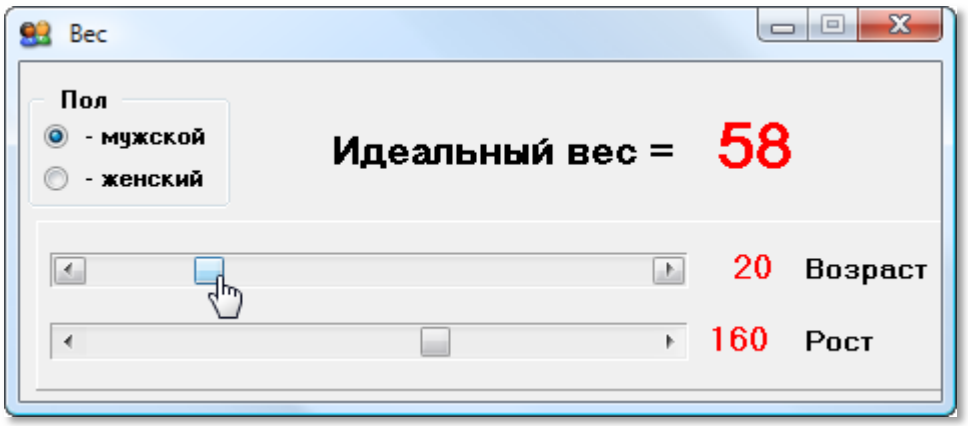


Рис. С4.3. Вес взят!



Исходный код программы находится в папке **Ves**.

Последний штрих

Вы можете продолжить свои исследования рода человеческого с помощью компьютера. Немного подправив нашу программу *Вес*, вы легко определите **жирность** тела любого гражданина (или гражданки).

Дотошная наука выяснила, что тело молодых здоровых мужчин содержит около 15%, а тело женщин – 22% жира. Жирность произвольно выбранного тела приблизительно оценивается по формуле:

Ж = (Вес - Р) : Вес х 100 + 15 - для мужчин,
Ж = (Вес - Р) : Вес х 100 + 22 - для женщин,

где **Р** – идеальный вес, который определяется по предыдущим формулам.

Не будем целиком переписывать всю программу *Вес*, а адаптируем её к «новым реалиям». Для этого создадим новую папку **Zhir**, в которую перепишем файлы формы и программы (*Главное меню* **File > Save as...,** затем **File > Save Project as...).** В файле модуля формы добавим ещё одну переменную – для хранения веса тела:

```
var
  Form1: TForm1;
  age, rost, ves: integer;
```

Также нужно исправить название функции – теперь мы будем вычислять не вес тела, а его *жирность*:

```
  private
    { Private declarations }
    function Zhir: single;
```

Основательно поработаем и над самой *функцией*:

```
//ВЫЧИСЛИТЬ ЖИРНОСТЬ ТЕЛА
function TForm1.Zhir: single;
var
  v: single;
begin
  if RadioGroup1.ItemIndex= 0 then begin //мужской пол
    v:= (rost*3 - 450 + age) * 0.25 + 45;
    Result:= (ves - v) / ves * 100 + 15 end
  else begin                              //женский пол
```

```
    v:= (rost*3 - 450 + age) * 0.225 + 40.4;
    Result:= (ves - v) / ves * 100 + 22 end;
  //скорректировать значения:
  if Result< 0 then Result:= 0;
  if Result> 100 then Result:= 100;
end;
```

Сначала, как и раньше, мы находим идеальный вес тела, а затем его жирность. Так как она вычисляется в процентах, то следует выправить курьёзные значения.

На форму «придётся» добавить ещё один **скроллер** – для ввода истинного веса тела:

```
Name = sbVes
Cursor = crHandPoint
Hint = 'Вес'
Max = 200
Min = 3
Position = 60
ShowHint = True
```

и **метку**:

```
Name = lblVes
Alignment = taRightJustify
AutoSize = False
Caption = '100'
Font.Color = clRed
Font.Height = -16
Font.Style = [fsBold]
```

Остальные изменения внесите сами, поглядывая на картинку (Рис. С4.4).

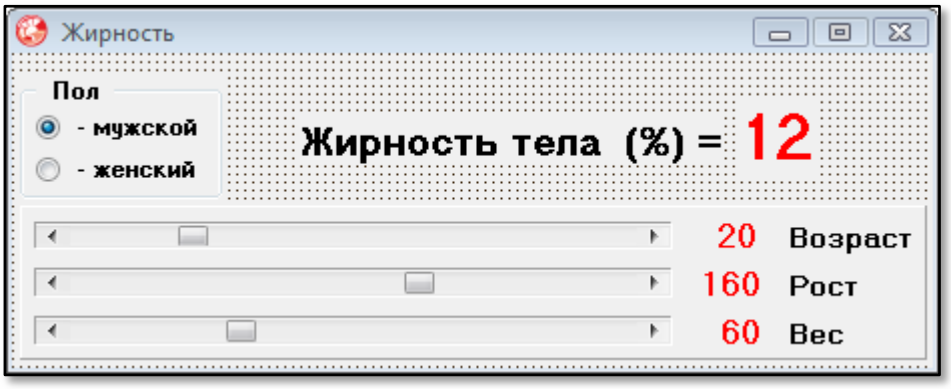


Рис. С4.4. «Жирный» интерфейс

Не забудьте прежнюю метку *lblVes* переименовать в **lblZhir**.

Осталось подправить код модуля (**выделенные** строки):

```
//СОЗДАТЬ ФОРМУ
procedure TForm1.FormCreate(Sender: TObject);
begin
  //снять значения со скроллеров:
  age:= sbAge.Position;
  Rost:= sbRost.Position;
  Ves:= sbVes.Position;
  //вычислить жирность:
  lblZhir.Caption:= inttostr(round(zhir))
end;

//ВЫБРАТЬ ПОЛ
procedure TForm1.RadioGroup1Click(Sender: TObject);
begin
  //вычислить жирность:
  lblZhir.Caption:= inttostr(round(zhir))
end;

//ЗАДАТЬ ВОЗРАСТ
procedure TForm1.sbAgeChange(Sender: TObject);
begin
  //возраст:
  age:= sbAge.Position;
  //вывести значение на экран:
  lblAge.Caption:= inttostr(age);
  //вычислить жирность:
  lblZhir.Caption:= inttostr(round(zhir))
end;

//ЗАДАТЬ РОСТ
procedure TForm1.sbRostChange(Sender: TObject);
begin
  //рост:
```

```
  Rost:= sbRost.Position;
  //вывести значение на экран:
  lblRost.Caption:= inttostr(Rost);
  //вычислить жирность:
  lblZhir.Caption:= inttostr(round(zhir))
end;

//ЗАДАТЬ ВЕС
procedure TForm1.sbVesChange(Sender: TObject);
begin
  //вес:
  ves:= sbVes.Position;
  //вывести значение на экран:
  lblVes.Caption:= inttostr(Ves);
  //вычислить жирность:
  lblZhir.Caption:= inttostr(round(zhir))
end;
```

Протестируйте программу на ком только сможете. Наверное, это доставит и вам, и всем испытуемым массу удовольствия и запомнится на всю жизнь.



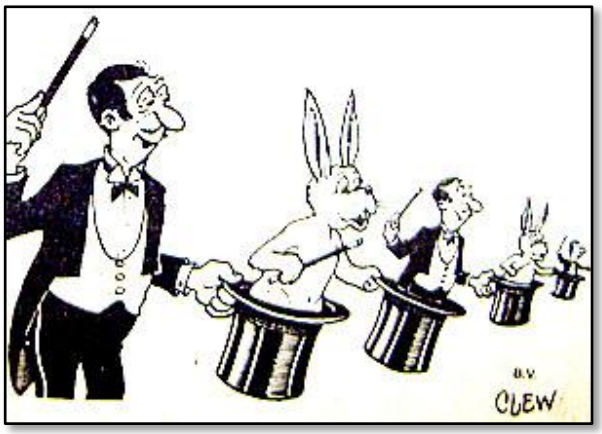
Исходный код программы находится в папке **Zhir**.

Если для вас это настоящая докука - из элементарных формул делать не очень простые программы, то дельфопланеризм – не ваша стихия, постарайтесь найти себя в других областях нескучных развлечений.

Урок 5. Рекурсия, или Сказочка про белого бычка

- Сказать ли тебе сказку про белого бычка?
- Скажи.
- Ты скажи, да я скажи,
да сказать ли тебе сказку про белого бычка?
- Скажи.
- Ты скажи, да я скажи, да чего у вас будет,
да докуль это будет!
Сказать ли тебе сказку про белого бычка?

Докучная сказка

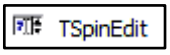


Юмористическая рекурсия

В теле процедур и функций могут быть записаны идентификаторы других процедур и функций, то есть из одной подпрограммы можно вызвать другую подпрограмму. Но наиболее интересный случай таких вызовов – это когда подпрограмма вызывает саму себя. Этот приём программирования называется **рекурсией**.

Хорошим примером рекурсии в литературе могут служить докучные сказки, которые находчивый русский народ придумал для того, чтобы изводить ими своих врагов, поскольку они никогда не заканчиваются (и те, и другие). Такую рекурсию называют *бесконечной*. В программах она вызывает «зависание» компьютера до тех пор, пока не будут полностью исчерпаны его ресурсы. Естественно, такое поведение приложения совершенно недопустимо, поэтому для рекурсивных подпрограмм необходимо предусмотреть *условие выхода* из рекурсии.

Попробуем запрограммировать другую докучную сказку – про то, как поп убил собаку.

Для гуманизации интерфейса установим на форме компонент 

(*Палитра Samples*), который будет ограничивать число повторений сказки. После нажатия на кнопку **btnPop** *Сказочка*, мы попадём в обработчик события:

```
procedure TForm1.btnPopClick(Sender: TObject);
var
  n: integer;
begin
  n:= speIter.Value;
  pop(n);
  listbox1.TopIndex:=listbox1.Items.Count-1;
end;
```

Здесь мы просто указываем программе, сколько раз хотим услышать эту душещипательную историю. Это и будет тот счётчик повторений, который избавит нас от бесконечной рекурсии.

А вот и сама сказочка:

```
procedure pop(n: integer);
begin
  if n <= 0 then exit;
  Form1.listbox1.Items.Add('У попа была собака, он её любил,');
  Form1.listbox1.Items.Add('Она съела кусок мяса, он её убил.');
  Form1.listbox1.Items.Add('В землю закопал и надпись написал:');
  Form1.listbox1.Items.Add('');
  pop(n-1);
end;
```

В первой строке как раз и находится обязательное условие, гарантирующее завершение сказочки. Как только параметр *n* станет равным нулю, процедура закончится. Но до этого она будет напечатана 1 раз, а затем вызовет себя

```
pop(n-1);
```

для повторения сказки на 1 раз меньше. Таким образом, процедура будет вызвана 5 раз (если вы не изменили начальное значение):

```
pop(4);
pop(3);
pop(2);
pop(1);
pop(0);
```

На последнем вызове сработает первая строка, и страдания читателя сих милых строк закончатся (Рис. У5.1).

Очень важно разобраться в механизме «самовызова» подпрограмм, так как рекурсивные процедуры обычно записываются очень коротко, но при этом выполняют множество действий, последовательность которых не так-то просто понять.

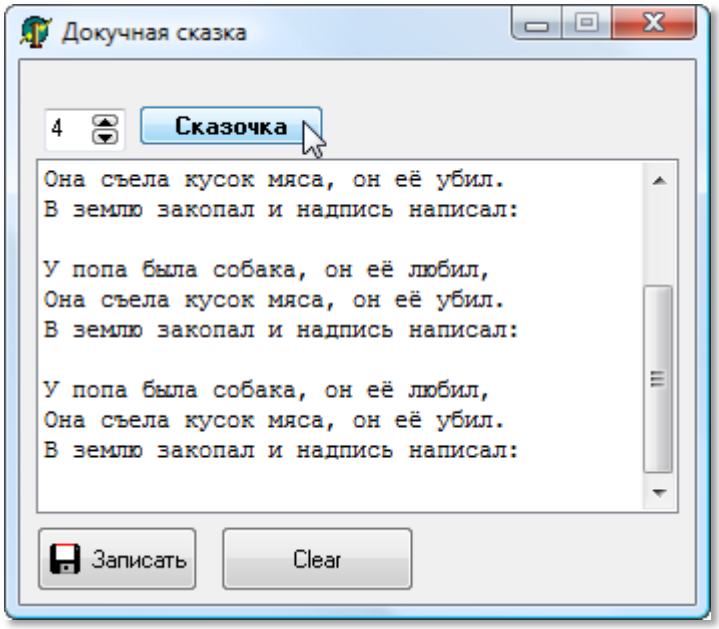


Рис. У5.1. Тут и сказочке конец!



Исходный код программы находится в папке **pop**.

Перейдём теперь к несказочной рекурсии – к классическому примеру **вычисления факториала**.

Интерфейс программы можно позаимствовать у предыдущего примера, с небольшими исправлениями (Рис. У5.2).

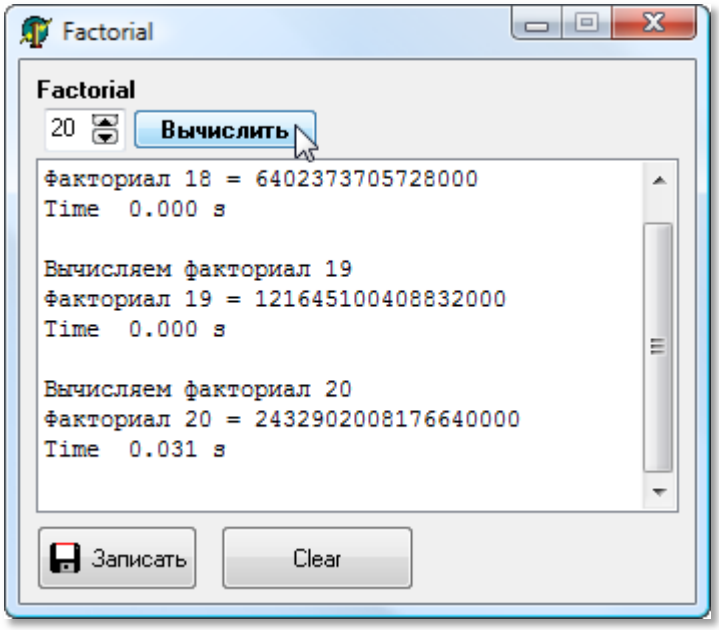


Рис. У5.2. Вычисляем факториалы

Здесь всё также начинается с задания количества повторов в окошечке *Factorial*, это как раз и будет то число, для которого мы будем вычислять факториал. Дополнительно засекаем время работы функции **factorial**, хотя на современных компьютерах это всегда будет 0.

Чтобы расширить допустимые пределы для результата вычислений, выберем целый тип *int64*.

```
//Рекурсивное вычисление факториала
procedure TForm1.btnFactorialClick(Sender: TObject);
var
  time,time0: single;
    n: integer;
    //k: longword;
    k: int64;
  s: string;
begin

  time0:=GetTickCount(); //засечь начало
  time:=time0;

  n:= speFactorial.Value;

  listbox1.items.add('Вычисляем факториал ' + inttostr(n));
  k:=factorial(n);
  listbox1.items.add('Факториал ' + inttostr(n) + ' = ' + int-
tostr(k));

  time0:=(GetTickCount()-time0)/1000;
  str(time0:6:3,s);
  listbox1.items.add('Time ' + s + ' s');
```

```
  listbox1.items.add('');

  listbox1.TopIndex:=listbox1.Items.Count-1;
end;
```

А вот как просто *рекурсивно* вычисляется факториал – всего 2 строки кода:

```
function factorial (n: integer): int64;
begin
  if n = 0 then factorial:= 1
  else factorial:= factorial(n-1)*n;
end;
```

Сравните её с той функцией, что мы рассматривали ранее. Обратите внимание на первую строку, где проверяется условие завершения рекурсии!



Исходный код программы находится в папке **factorial**.

Рекурсивные подпрограммы выглядят очень красиво, «профессионально», однако не стоит ими злоупотреблять – практически всегда можно решить задачу и без рекурсии (но не всегда это решение будет проще). С помощью рекурсии следует решать задачи, в которых требуется многократно выполнить сходные действия. При этом должно существовать нерекурсивное решение для самого простого из этих действий, которое и будет служить условием завершения рекурсии. Часто рекурсию используют для решения комбинаторных задач с полным или частичным перебором: обход лабиринта, расстановка ферзей, генерирование перестановок чисел, головоломка *Ханойские башни*.

Как не заблудиться в коде, или Секреты навигации

Если вы потерялись,
встречайтесь у фонтана!

Первое правило всех паганелей

Пока код программы занимает несколько строчек, достаточно прокрутить весь текст, чтобы перейти к нужной строке, но обычно программы состоят из многих сотен и тысяч строк. И тут уже не обойтись без **навигации**. Так как это одна из наиболее «употребляемых» операций, то есть смысл рассказать о ней подробнее.

В окне *Структуры проекта* находится список всех объектов модуля, выбранного в *Рабочем окне.* Здесь вы сможете легко найти все константы, переменные, типы, процедуры и функции, объявленные в модуле, и быстро перейти к ним.

Если проект содержит большое количество компонентов и/или очень длинный исходный код, то из окна *Структуры проекта* вы сможете быстро переместиться к нужному компоненту или фрагменту исходного кода. Например, в режиме конструирования формы *одиночный щелчок* на названии компонента выделяет его на форме (Рис. У5.3).

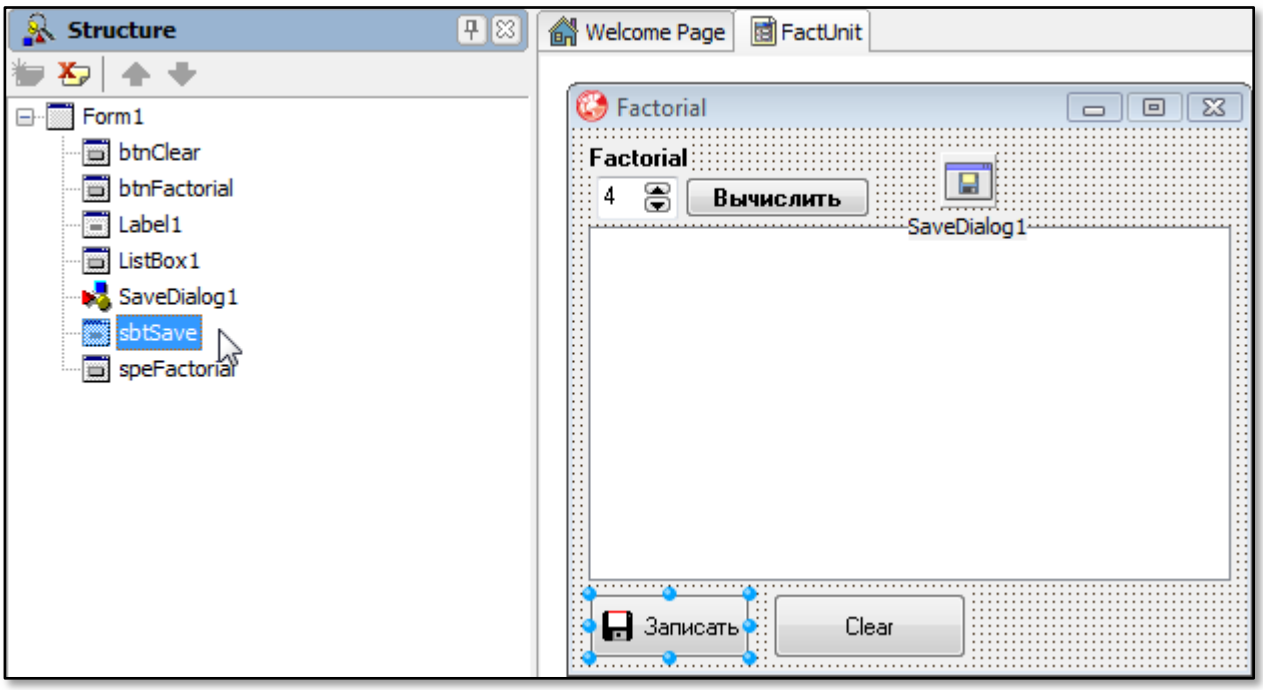


Рис. У5.3. Одним щелчком мыши можно выделить кнопку на форме

Обратите внимание, что в окне *Инспектора объектов* этот компонент становится активным, и вы можете изменять его свойства.

Установите курсор на объявлении процедуры, функции, метода в разделе *interface* вашего модуля и нажмите клавиши **Ctrl+Shift+Up** или **Ctrl+Shift+Down** (клавиши со стрелками). Вы тут же окажетесь в разделе **implementation**, где находится их код. И наоборот, повторное нажатие этих комбинаций клавиш перенесёт вас в мгновение ока в раздел *interface*, на объявление процедуры.

При наведении курсора на любой объект исходного кода рядом появляется информация о нём (Рис. У5.4).

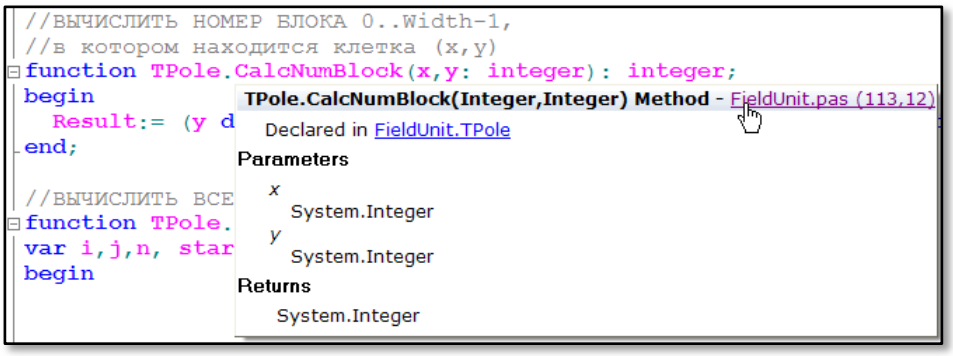


Рис. У5.4. Подсказка

Щёлкнув по ссылке, вы перейдёте к его объявлению.

В любой строке текста можно **установить закладку**. Для этого следует выполнить команду контекстного меню **Toggle Bookmarks** и щёлкнуть на любой из десяти закладок или нажать комбинацию клавиш **Ctrl + Shift + 0..9** (цифра обозначает номер закладки). Слева от строки текста с курсором появится зелёная книжка. В *Delphi* все закладки сохраняются вместе с файлом и затем восстанавливаются при повторной загрузке файла!

Для перехода к нужной закладке используйте команду контекстного меню **Goto Bookmarks** или клавиши **Ctrl + 0..9**.

Убрать закладку можно той же командой **Toggle Bookmarks**.

Если у вас хорошая память, то попробуйте запомнить *номер* нужной строки (он, конечно, может и измениться, если вы добавите код). Вы вернётесь к ней по команде *Главного меню* **Search > Go to Line Number** (или **Alt+G**), указав в диалоговом окне номер строки (Рис. У5.5).

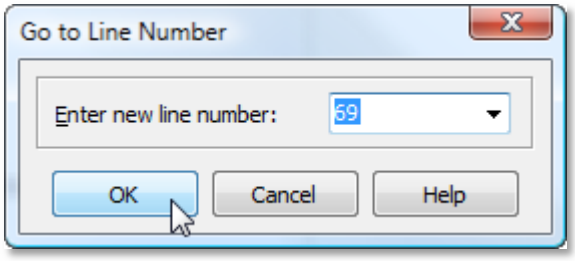


Рис. У5.5. Переходим в заданную строку

Ещё больше возможностей предоставляет команда *Главного меню* **Search > Find** (она же **Ctrl+F**). Установите курсор на нужный идентификатор (или любое слово) и выполните указанную команду. Под окном *Редактора кода* появится диалог, в текстовом окне которого будет ваше слово (можно и просто набрать нужное слово) (Рис. У5.6). Нажмите кнопку со стрелкой, и

курсор переместится на «первое попавшееся» указанное слово в тексте. Нажимайте клавишу *F3* для перехода к следующей «находке». Кнопки со стрелками перебирают все найденные в тексте слова вверх или вниз от текущего (оно выделено чёрным фоном, а все остальные «находки» - оранжевым).

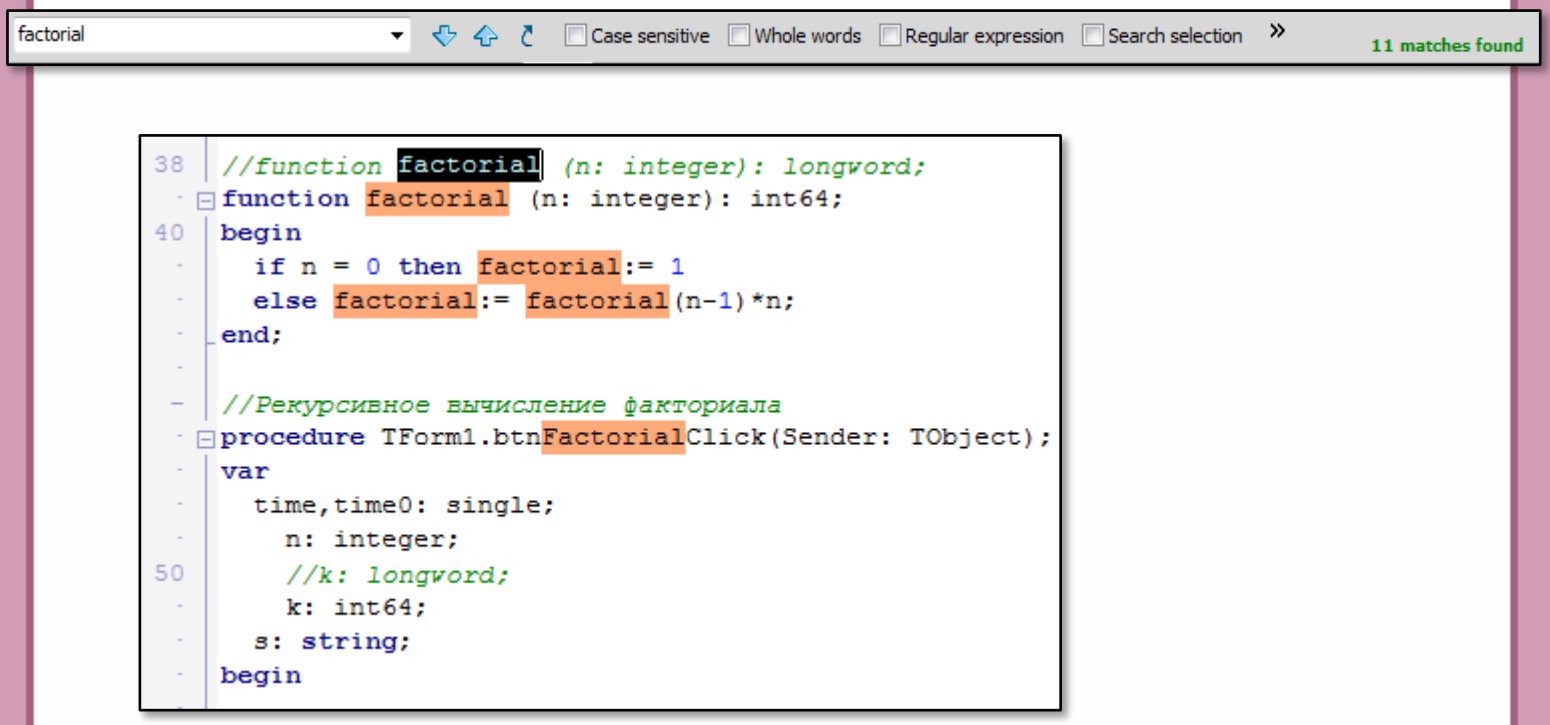


Рис. У5.6. Тотальный поиск

Все введённые слова запоминаются, поэтому для повторного поиска достаточно выбрать нужное слово в раскрывающемся списке, который появляется, если нажать на кнопку со стрелкой справа от окна ввода текста.

И ещё одна возможность перехода через альпы кодовых строк. Установите курсор на названии процедуры в разделе *implementation*, вызовите контекстное меню *Редактора кода* и выполните команду **Find Declaration** (Рис. У5.7), чтобы оказаться в разделе *interface*, где находится объявление данной процедуры (Рис. У5.8).

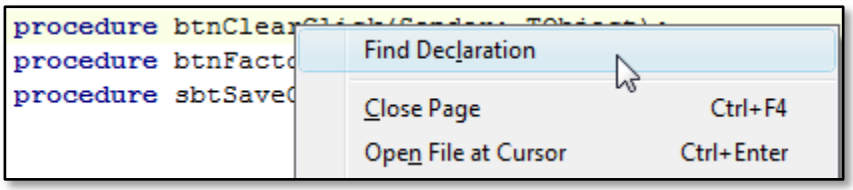


Рис. У5.7. Переходим к процедуре

```
procedure TForm1.btnClearClick(Sender: TObject);
begin
  listbox1.Clear;
end;
```

Рис. У5.8. Можно работать!

И только попробуйте теперь заблудиться!

Семинар 5. Всё познаётся в сравнении, или Вычисляем НОК и НОД

Иван кивает на Петра,
а Петр кивает на Ивана.

Русская рекурсивная пословица

Для сравнения итерационных и рекурсивных процедур напишем две родственные программы из школьной жизни – для вычисления **НОД** (*наибольшего общего делителя* двух чисел) и **НОК** (*наименьшего общего кратного*). Они должны быть вам хорошо известны (если это не так, сходите, наконец, в школу!).

Наибольший общий делитель

Так как компьютер производит вычисления очень быстро, для более точного сравнения скоростей будем повторять вычисления 1 миллион раз (но вы можете задать и другое число, побольше, если никуда не торопитесь). Для определения времени нам потребуются две подпрограммы:

```
var
  Form1: TForm1;
  time,time0: single;

procedure TimeOn;
begin
  time0:=GetTickCount(); //засечь начало
end;

function TimeOut: string;
var
  s: string;
begin
  time:=(GetTickCount()-time0)/1000;
  str(time:6:3,s);
  Result:= s;
end;
```

Для запуска соответствующих подпрограммм установим две кнопки: **btnIter** с надписью *ИТЕРАЦИИ* и **sbtRecurs** с надписью *РЕКУРСИЯ*. Первая, как следует из названия, запускает итерационный цикл, вторая – рекурсивный:

```
procedure TForm1.btnIterClick(Sender: TObject);
var n1,n2: integer;
  i: integer;
begin
  n1:= strtoint(txtNum1.Text);
  n2:= strtoint(txtNum2.Text);
  TimeOn;
  for i:= 1 to strtoint(txtNumLoop.Text) do NOD(n1,n2);
  lblTime.Caption:= TimeOut;
  lblRes.Caption:= inttostr(NOD(n1,n2));
end;

procedure TForm1.sbtRecursClick(Sender: TObject);
var n1,n2: integer;
  i: integer;
begin
  n1:= strtoint(txtNum1.Text);
  n2:= strtoint(txtNum2.Text);
  TimeOn;
  for i:= 1 to strtoint(txtNumLoop.Text) do NODR(n1,n2);
  lblTime.Caption:= TimeOut;
  lblRes.Caption:= inttostr(NODR(n1,n2));
end;
```

Сами функции для вычисления **НОД** достаточно просты:

```
//Вычисление наибольшего общего делителя
//двух чисел типа integer

//Итерационное вычисление НОД
function NOD(n1,n2: integer): integer;
//n1 >= n2 >= 0;
var temp: integer;
begin
  if n1 < n2 then swap(n1,n2);
  if n2=0 then result:= n1
  else begin
    while n2>0 do begin
      temp:= n1 mod n2;
      n1:=n2;
      n2:= temp;
    end;
    result:=n1;
  end;
end;

//Рекурсивное вычисление НОД
function NODR(n1,n2: integer): integer;
begin
  if n1 < n2 then result:=NODR(n2,n1)
  else
    if n2<=0 then result:= n1
```

```
    else result:= NODR(n2,n1 mod n2);
end;
```

Найдите в функции **NODR** условие выхода из рекурсии!

Алгоритм, реализованный в функции **NODR**, кажется очень простым, но это обманчивая простота, поэтому рассмотрим его подробнее. Для определённости будем считать, что нужно найти **НОД** чисел **n1**=15 и **n2**=21.

1. Для вычислений важно, чтобы первое число было не меньше второго. Если это не так, то в строке

```
if n1 < n2 then result:=NODR(n2,n1)
```

числа переставляются, и функция **NODR** вызывается с правильным порядком аргументов. Для наших чисел *n1 < n2*, поэтому будет вызвана функция NODR(n2,n1), то есть NODR(21,15). Важно понять, что как числа ни переставляй, первое из них – это **n1**, а второе - **n2**, поэтому в самой функции n1=21, а n2=15!

Теперь условие *n1 >= n2* точно выполняется.

2. В случае, если второе число меньше нуля или равно нулю, **НОД** равен первому числу и вычисления заканчиваются:

```
if n2<=0 then result:= n1
```

Возможно, это условие будет верным и сразу, а если нет, то будет выполняться строка

```
result:= NODR(n2,n1 mod n2);
```

Мы видим, что большим числом становится *n2* (которое было меньшим в предыдущем обращении к функции), а меньшим – остаток от деления большего числа на меньшее (очевидно, что *n2* не может быть меньше остатка).

Для наших чисел условие **n2<=0** (15 <=0) неверное, так что будет исполняться строка

```
result:= NODR(15, 21 mod 15); или
result:= NODR(15, 6);
```

и вычисления продолжатся с начала п.2.

3. Так как **n2 (15)** больше нуля, то ещё раз будет выполнена строка

```
result:= NODR(n2,n1 mod n2); или
```

```
result:= NODR(6, 15 mod 6); или
result:= NODR(6, 3);
```

Опять переходим к п.2, а затем к п.3:

```
result:= NODR(n2,n1 mod n2); или
result:= NODR(3, 6 mod 3); или
result:= NODR(3, 0);
```

Снова переходим к п.2. Теперь условие

```
if n2<=0 then result:= n1
```

выполнено и функция возвращает **НОД** – число 3.

Как видите, параметр n2 рано или поздно станет равен нулю, и рекурсия неизбежно закончится в строке

```
if n2<=0 then result:= n1,
```

которая и является условием завершения рекурсии.

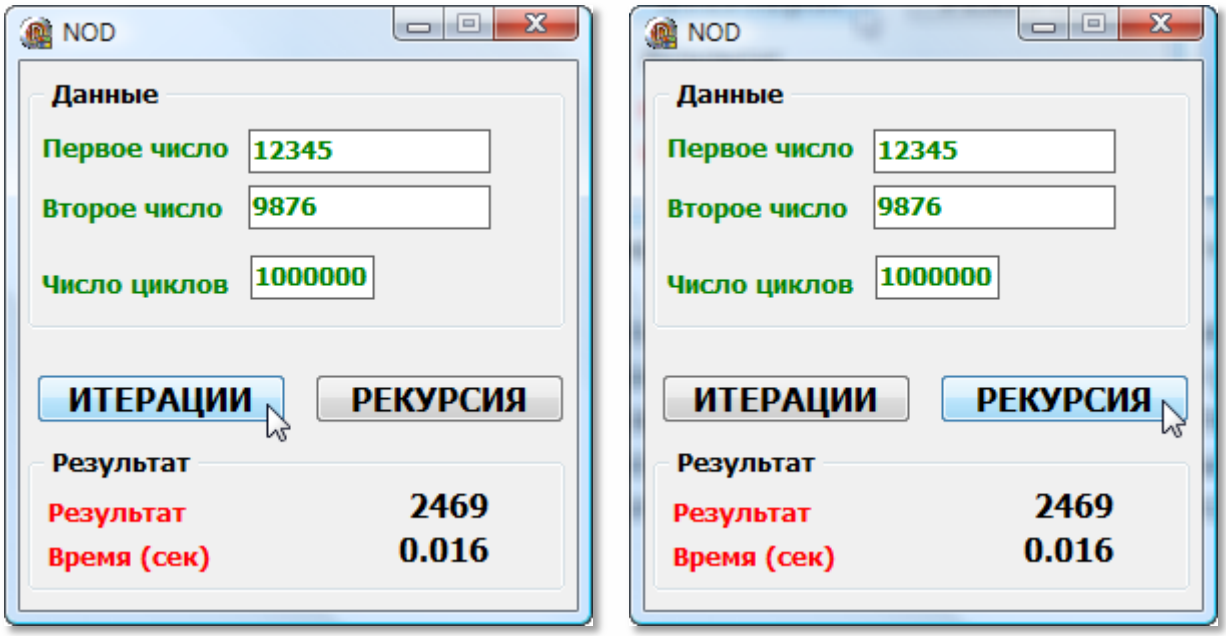


Рис. С5.1. «Итерации» - «Рекурсия»: счёт 1 : 1!

Для более точного сравнения итерационного и рекурсивного методов, понажимайте на кнопки по нескольку раз, потому что в системе *Windows* одновременно выполняется множество процессов, так что время может несколько отличаться в разных «попытках». Более того, результаты зависят и от чисел, для которых вычисляется **НОД** (Рис. С5.1).



Исходный код программы находится в папке **NOD**.

Наименьшее общее кратное

На самом деле **НОК** просто вычисляется через **НОД** по такой формуле:

```
НОК = n1 * n2 div НОД (n1,n2),
```

так что нам достаточно дополнить предыдущую программу двумя функциями, которые реализуют эту формулу:

```
//Итерационное вычисление НОК
function NOK(n1,n2: integer): integer;
var temp: integer;
begin
  temp:= n1 * n2;
  if temp = 0
  then result:= 0
  else result:= temp div NOD(n1,n2);
end;

//Рекурсивное вычисление НОК
function NOKR(n1,n2: integer): integer;
var temp: integer;
begin
  temp:= n1 * n2;
  if temp = 0
  then result:= 0
  else result:= temp div NODR(n1,n2);
end;
```

Казалось бы, и здесь скорость вычислений должна быть одинакова, но рекурсивный способ всё-таки уверенно выигрывает (Рис. С5.2).

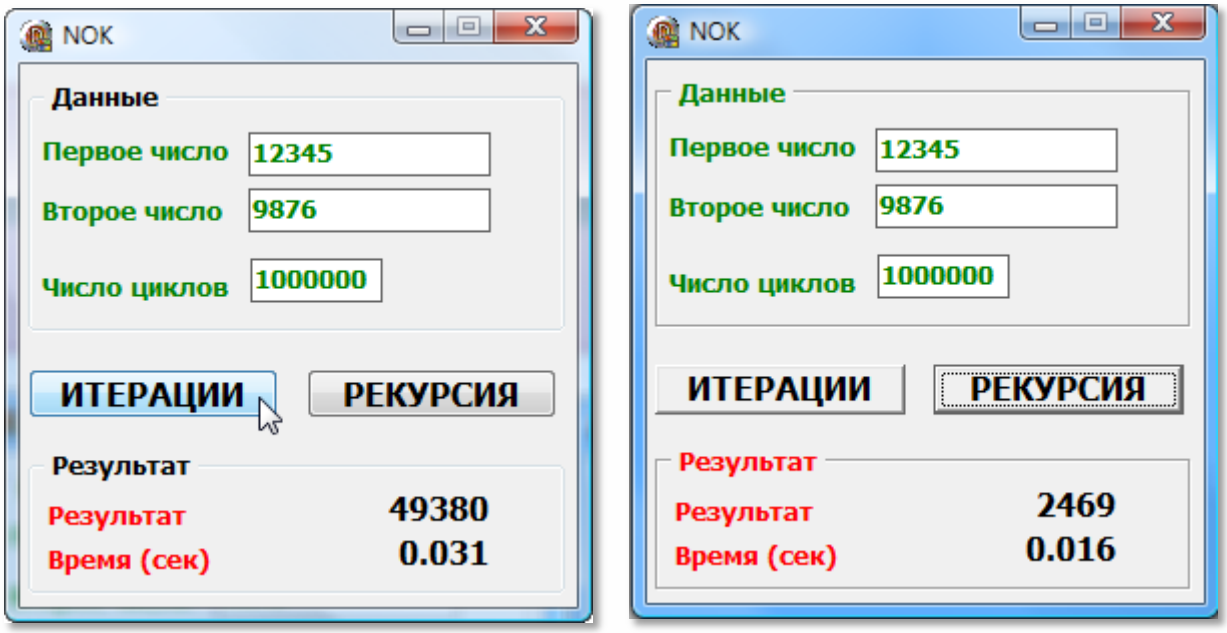


Рис. С5.2. А в этом «матче» «Рекурсия» выиграла! Счёт 2 : 1!



Исходный код программы находится в папке **NOK**.

Урок 6. Простые типы данных

Будьте проще, и к вам потянутся люди!

Кредо Глеба Жеглова

Название «простые типы данных» обусловлено тем, что они не содержат «в себе» никаких других типов данных.

Все простые типы данных, кроме вещественных, называют также **порядковыми**, так как они определяют упорядоченный набор значений.

Целые типы

Самые «востребованные» типы данных! Можно смело утверждать, что ни одна программа не обходится без них. Именно этим и объясняется такое обилие целых типов данных в *Delphi*.

Как следует из названия, они предназначены для хранения целых чисел. Большинство типов могут хранить числа со знаками, но типы **byte** и **Longint** - только неотрицательные числа. Другое принципиальное отличие целых типов заключается в максимальной и минимальной величине чисел, которые они содержат. Естественно, чем эти значения больше, тем больше памяти потребуется для их хранения. Кроме того, операции с «родными» целыми типами – **Integer** и **cardinal** - *Delphi* работает быстрее всего. Таким образом, при выборе целого типа данных для своих переменных вы должны руководствоваться именно этими соображениями. Если коротко: всегда выбирайте типы **Integer** и **cardinal**, если у вас нет веских аргументов в пользу других типов данных.

Итак, выбираем тип данных по душе:

Тип	Диапазон значений	Объём памяти
Shortint	-128..127	1 байт
Byte	0..255	1 байт
SmallInt	-32768..32767	2 байта
Word	0..65535	2 байта
Integer	-2147483648..2147483647	4 байта
LongInt	-2147483648..2147483647	4 байта
Cardina l	0..4294967295	4 байта
LongWord	0..4294967295	4 байта
Int64	$-2^{63}..2^{63}-1$	8 байтов

Объявление переменных *целого* типа:

```
Var i,j: integer;
  C: cardinal;
  B: byte;
  W: word;
```

Значения переменным целого типа иногда удобнее задавать не десятичными, а 16-ричными числами:

```
B:= 255; или
B:= $FF;

w:= 2009; или
w:= $000007D9;
```

Особенно такой способ уместен для типов **Byte** и **Word**, которые часто используют для манипуляций с битами.

Большое преимущество целых типов перед вещественными состоит не только в том, что процессор быстрее их «переваривает», но прежде всего в том, что они всегда хранят **точные** значения, а вещественные – приближённые.

Вещественные типы (Действительные типы)

Значение переменных вещественного типа всегда **приблизительные**, поэтому их нельзя проверять на равенство!

Тип	Диапазон значений	Точность	Объём памяти
Real48	-2.9×10^{-39} ..1.7×10^{38}	11-12 цифр	6 байт
Single	-1.5×10^{-45} ..3.4×10^{38}	7-8 цифр	4 байта
Double = Real	-5.0×10^{-324} ..1.7×10^{308}	15-16 цифр	8 байтов
Extended	-3.6×10^{-4951} ..1.1×10^{4932}	19-20 цифр	10 байтов
Comp	$-2\times10^{63}+1$ ..$2\times10^{63}-1$	19-20 цифр	8 байтов
Currency	-922337203685477.5808 .. 922337203685477.5807	19-20 цифр	8 байтов

Тип **Real48** оставлен только для совместимости с предыдущими версиями *Delphi*, поэтому его не следует использовать.

Типы **Comp** и **Currency** используются только в денежных операциях.

Обычно для расчётов хватает точности типа **Single**, но вычисления производятся быстрее с типом **Real** (он же **Double**).

Для записи вещественных чисел используют 2 способа. Первый – как в школе, но вместо десятичной запятой ставится точка:

```
1.5
-3.14
```

Для очень больших и очень маленьких чисел больше подходит второй способ. Сначала записывается однозначное число с десятичной точкой, затем буква E (или e) и целое число, которое равно степени десятки. Например,

1.05e2 это 1.05×10^2 **или** 1.05 x 100 = 105

-3.14e-5 это -3.14×10^{-5} **или** -3.14 x 0,00001 = -0.0000314

Объявление переменных *вещественного* типа:

```
Var a,b: Single;
  C: Real;
```

Вещественные числа часто используются для «технических» расчётов, а при необходимости их можно округлить до целых чисел (тип результата - **int64**) с помощью функций **trunc** и **round**.

Целым переменным нельзя присваивать действительные значения!

Функция **trunc** просто отсекает дробную часть числа:

```
trunc(2.1728)  = 2
trunc(-2.1728)) = -2
```

Функция **round** округляет действительное число до ближайшего целого:

```
round(2.1728)  = 2
round(2.7)  = 3
round(2)  = 2
round(-2.1728)) = -2
```

Чтобы было легче понять, как действует функция **round**, выразим её через **trunc**:

```
round(x) =  trunc(x + 0.5), если x>=0
round(x) =  trunc(x - 0.5), если x < 0
```

Перечисляемый тип

Определяется списком идентификаторов, которые перечисляются через запятую в круглых скобках. Поскольку этот список формирует сам программист, он должен объявить новый тип. Например, тип **Autos** может содержать марки любимых автомобилей:

```
Type
  Autos = (Mazda, Toyota, Opel, BMW, Volkswagen);
```

Далее мы можем объявить *переменную* этого типа:

```
Var
  MyAutos: autos;
```

Но можно и сразу объявить переменную перечисляемого типа вот так:

```
Var
  MyAutos: (Mazda, Toyota, Opel, BMW, Volkswagen);
```

Значением переменной **MyAutos** могут быть *Mazda, Toyota, Opel, BMW, Volkswagen* и только они! Если вы купили (или богатый папа подарил) мерседес, то вам придётся подправить объявление типа, а вот с демократичным фольксвагеном не будет никаких проблем:

```
MyAutos:= Volkswagen;
```

Присвоили переменной нужное значение – и на дорогу с чистой совестью.

Функция **Ord** возвращает порядковый номер идентификатора (его числовое значение) в списке:

```
ord(Mazda) → 0
ord(Volkswagen) → 4
```

Будьте внимательны: нумерация начинается с *нуля*!

При сравнении переменных учитываются значения соответствующих идентификаторов:

```
Mazda < Toyota < Opel < BMW < Volkswagen
```

Идентификаторы в перечисляемом типе не могут быть использованы для наименования других переменных!

Логический тип

Это перечисляемый тип, который содержит только 2 элемента: **TRUE** и **FALSE**, поэтому логические переменные могут принимать только эти 2 значения. Для их хранения достаточно 1 байта.

Как для перечисляемого типа

```
FALSE < TRUE
Ord(FALSE) = 0
Ord(TRUE) = 1
succ(FALSE) = TRUE
pred(TRUE) = FALSE
```

Объявление переменных *логического* типа:

```
var a,b,c: boolean;
```

Тип **ByteBool** эквивалентен типу **boolean**. **WordBool** занимает в памяти 2 байта, а **LongBool** – 4. Эти логические типы используются крайне редко.

Интервальный тип (тип-диапазон, ограниченный тип)

Определяет программист на основе любого простого порядкового типа. Так как значения всех порядковых типов данных упорядочены, то для этого достаточно указать границы диапазона значений.

Объявление интервальных типов:

```
Type
  hours = 0..23;
  letters=  'a'..'z';
```

Как видите, нужно указать интервал значений, определяемый двумя константами – минимальным и максимальным значением, - которые разделяются двумя точками.

Переменные этих типов объявляются, как обычно:

```
Var
  Hour: hours;
  Letter: letters;
```

Если встреча запланирована на 21 час, мы можем запомнить это событие так:

```
Hour:= 21;
```

Но не пытайтесь встретиться в 24 часа, компилятор выдаст сообщение об ошибке. Таким образом, интервальные типы не только экономят память для хранения значений переменных, но и дополнительно контролируют ваши действия!

Символьные типы

Основной символьный тип в *Delphi* – **Char**. Он определяет набор символов *Юникода* с кодами 0.. 65535, поэтому для хранения значений использует два байта. В последних версиях *Delphi* тип **Char** эквивалентен типу *WideChar* (прежде это был тип *AnsiChar*).

Код любого символа можно узнать с помощью функции **ord**:

```
ord('1') = 49
ord('S')= 83
ord('Ы')= 219
```

Символы *сравниваются* по их кодам, поэтому 'Ы' > ' S '.

Функция **chr**, наоборот, возвращает символ с заданным кодом:

```
chr(49) = '1'
chr(83)='S'
chr(219)='Ы'
```

Тип **AnsiChar** определяет набор символов *ASCII* с кодами 0..255, поэтому для хранения значений достаточно одного байта. Первые 128 символов этих типов всегда одинаковы, а остальные зависят от локальных настроек компьютера. Например, если у вас установлена русская версия *Windows*, то это символы русского алфавита.

Объявление переменных *символьного* типа:

```
var a,b,c: char;
```

Некоторые языки содержат значительно больше знаков, чем оставшиеся 128 символов, поэтому для их хранения в памяти требуется 2 байта. Тип **WideChar** содержит 65535 символов, что вполне достаточно даже для китайских иероглифов.

Функции и процедуры для порядковых типов данных

Для порядковых типов есть несколько очень полезных функций и процедур, некоторыми из которых мы уже пользовались.

Pred – функция, которая возвращает предыдущее значение выражения порядкового типа, указанного в скобках:

```
Pred(2) = 1
Pred('2') = '1'
Pred(True) = False
Pred(Volkswagen) = BMW
```

Succ - функция, которая возвращает следующее значение:

```
Succ(1) =2
Succ('1') = '2'
Succ(False) = True
Succ(BMW) = Volkswagen
```

Легко заметить, что действие этой функции противоположно действию функции *Pred*, и наоборот.

Ord - функция, которая возвращает порядковый номер значения в списке:

```
Ord('A') = 65
Ord(True) = 1
Ord(Volkswagen) = 4
```

Low - функция, которая возвращает минимальное значение указанного типа данных:

```
Low(Integer) = -2147483648
Low(Boolean) = False
Low(Char) = #0 (символ с кодом 0)
```

High - функция, которая возвращает максимальное значение указанного типа данных:

```
High(Integer) = 2147483647
High(Boolean) = True
High(Char) = (символ с кодом #$FFFF)
```

С помощью этих двух функций вы легко сможете определять диапазон значений любого порядкового типа.

Ну, и ещё две процедуры, с которыми мы уже знакомы:

Dec – функция *декремента*; уменьшает значение переменной типа *integer* на единицу:

```
Var n: integer;
n:= 2;
```

После выполнения функции *Dec(n)* значение переменной *n* станет равным 1.
Можно уменьшить значение переменной на *любое* целое число:

```
Dec(n, 3) → n= -1
```

По своему действию эта функция эквивалентна оператору присваивания:

```
n:= n - 1; или
n:= n - 3;
```

но выполняется быстрее.

Inc – процедура *инкремента*; увеличивает значение на единицу.

```
Var n: integer;
n:= 2;
```

После выполнения функции *Inc(n)* значение переменной n станет равным 3.
Можно увеличить значение переменной на *любое* целое число:

```
Inc(n, 3) → n= 5
```

По своему действию эта функция эквивалентна оператору присваивания:

```
n:= n + 1; или
n:= n + 3;
```

но выполняется быстрее.

Как добавить новый модуль или новую форму к проекту?

Если проект большой, то удобнее разбить его на несколько отдельных частей, состоящих из модуля и формы (или только модуля).

Чтобы **добавить новый модуль к проекту**, нажмите кнопку **New Items** на *Панели инструментов* и в диалоговом окне выберите **Delphi Projects > Delphi Files > Unit** (Рис. У6.1).

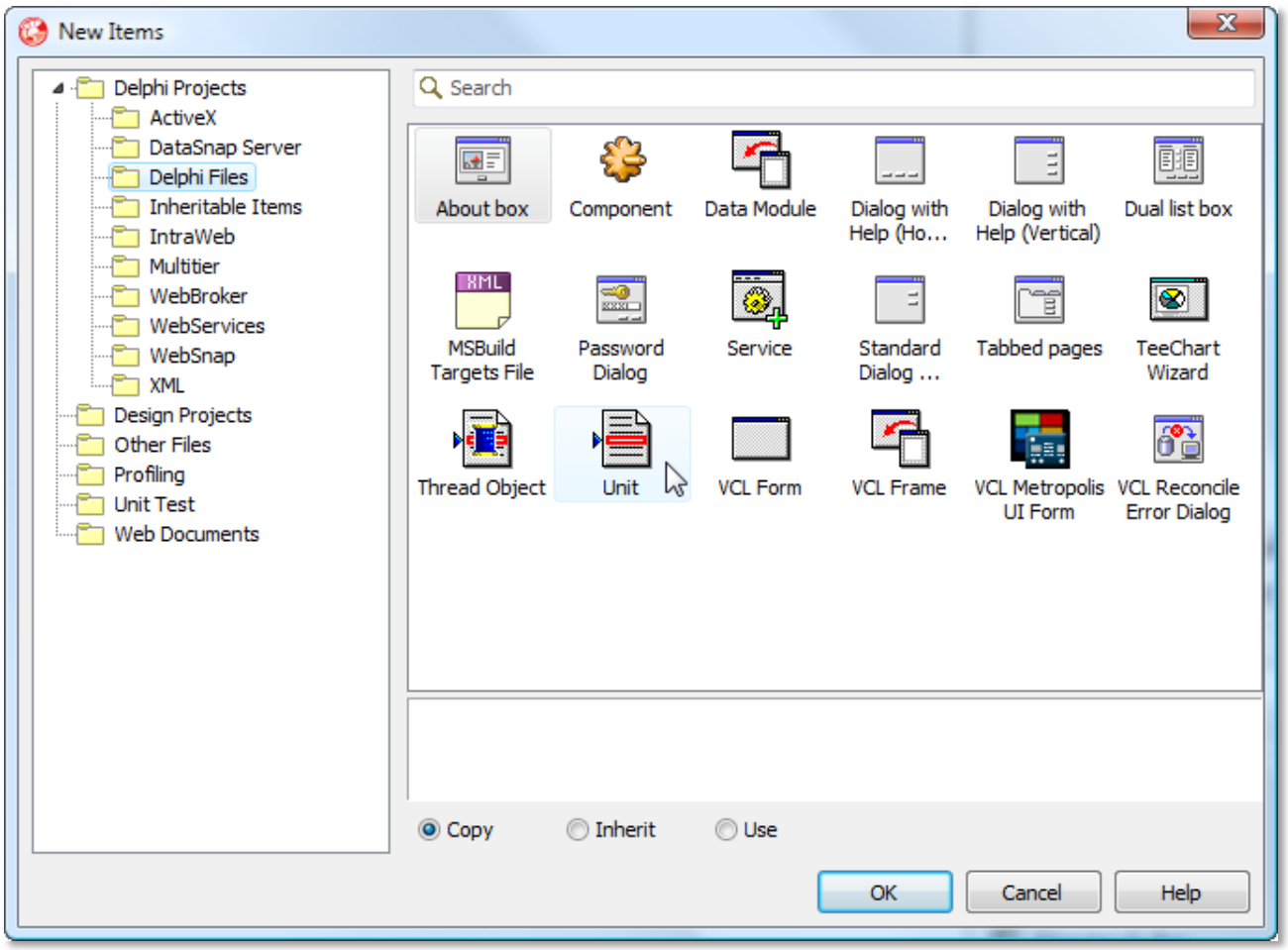


Рис. У6.1. Добавляем модуль

То же самое можно сделать командой *Главного меню* **File > New > Unit – Delphi** (Рис. У6.2).

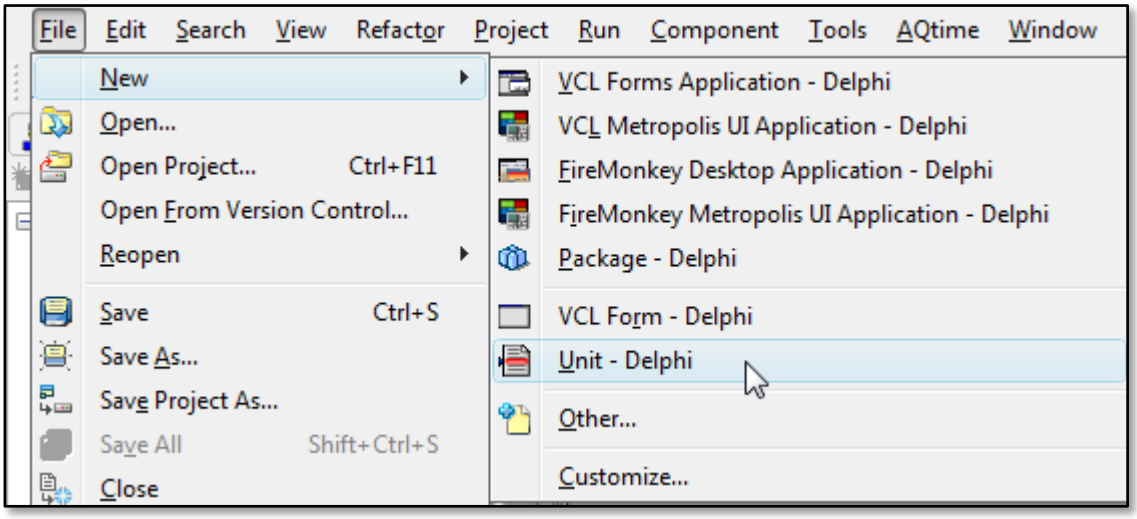


Рис. У6.2. Добавляем модуль из меню *File*

Ещё быстрее можно добавить модуль из окна *Менеджера проекта* (Рис. У6.3).

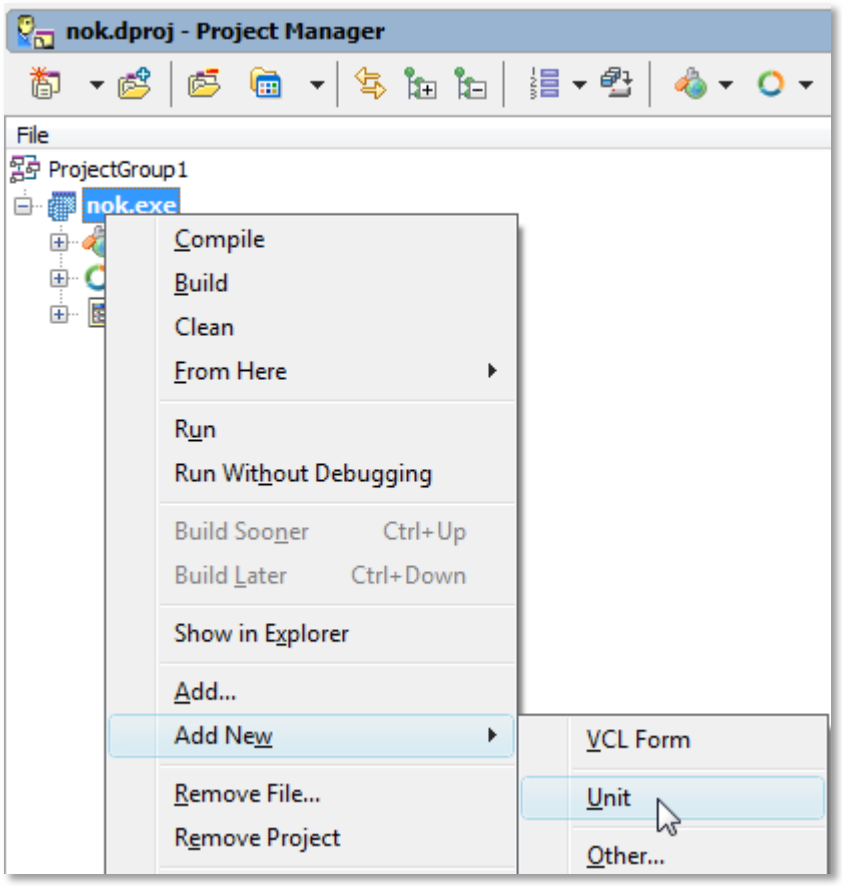


Рис. У6.3. Добавляем модуль из *Менеджера проекта*

Чтобы **добавить новую форму к проекту**, нажмите кнопку **New Items** на *Панели инструментов* и в диалоговом окне выберите **Delphi Projects > Delphi Files > VCL Form** (Рис. У6.4).

То же самое можно сделать командой *Главного меню* **File > New > VCL Form – Delphi** (Рис. У6.5).

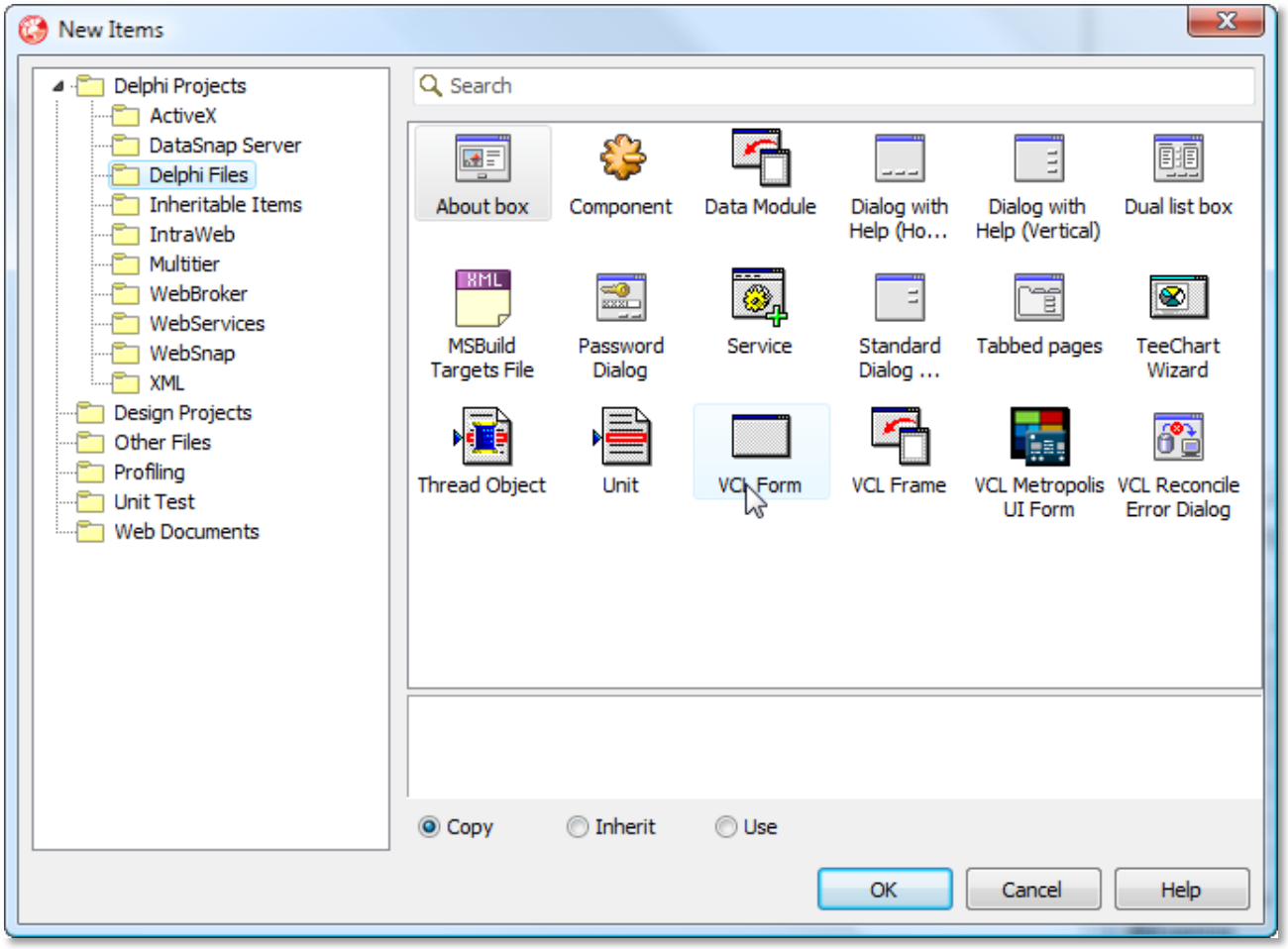


Рис. У6.4. Добавляем форму

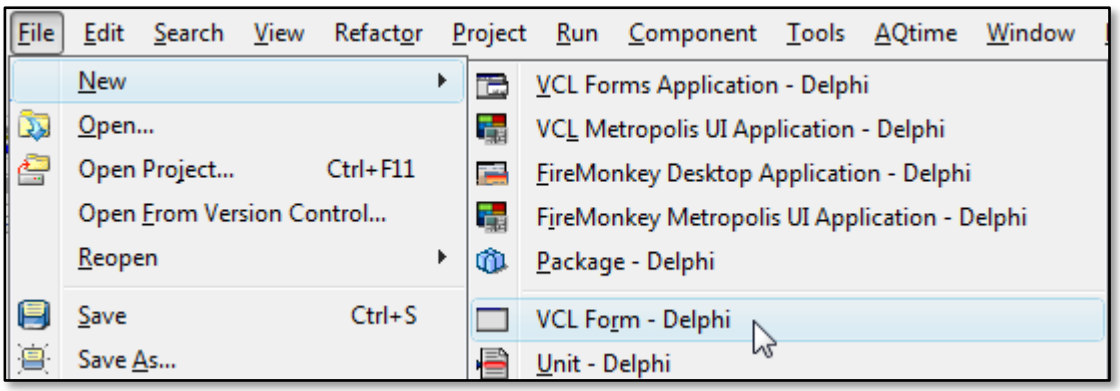


Рис. У6.5. Добавляем форму из меню *File*

Ещё быстрее можно добавить форму из окна *Менеджера проекта* (Рис. У6.6).

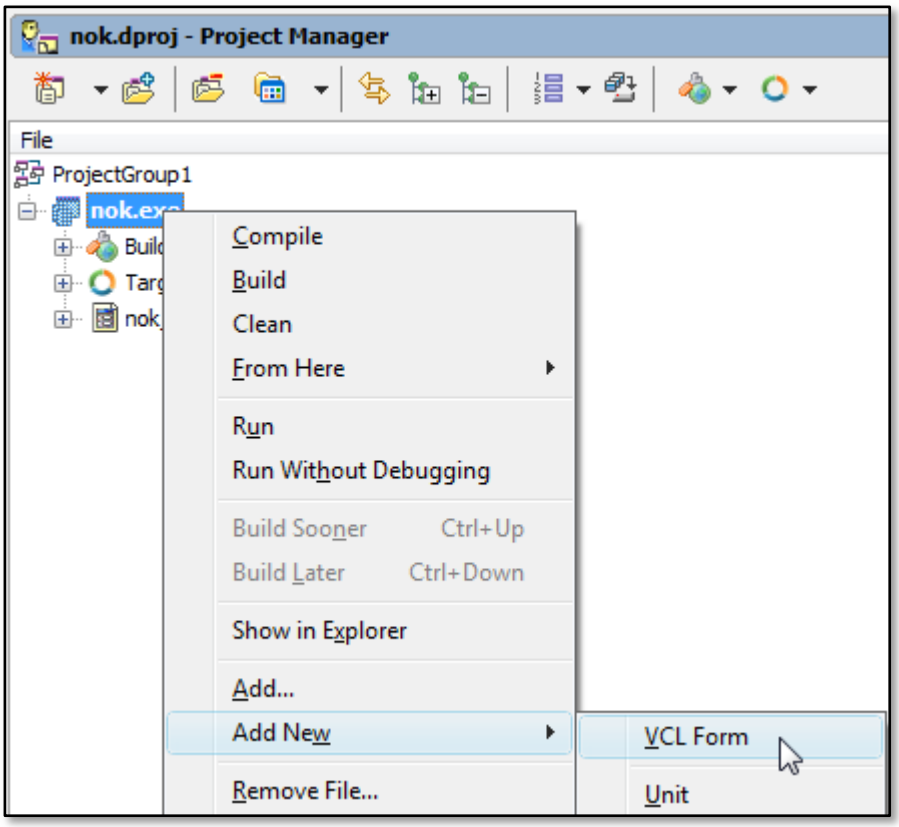


Рис. У6.6. Добавляем форму из меню *Менеджера проекта*

Семинар 6. Форма-заставка

Форма-заставка (**splash screen**) часто используется при загрузке больших программ для того, чтобы развлечь пользователя интересной картинкой, сообщить ему полезную информацию о программе, об адресах и сайтах производителя, наконец, призвать к регистрации программы. Вы, конечно, много раз видели такие заставки – они появляются при установке программ и запуске *MS Word, Corel Draw, Adobe Photoshop* или *Delphi* (Рис. С6.1).

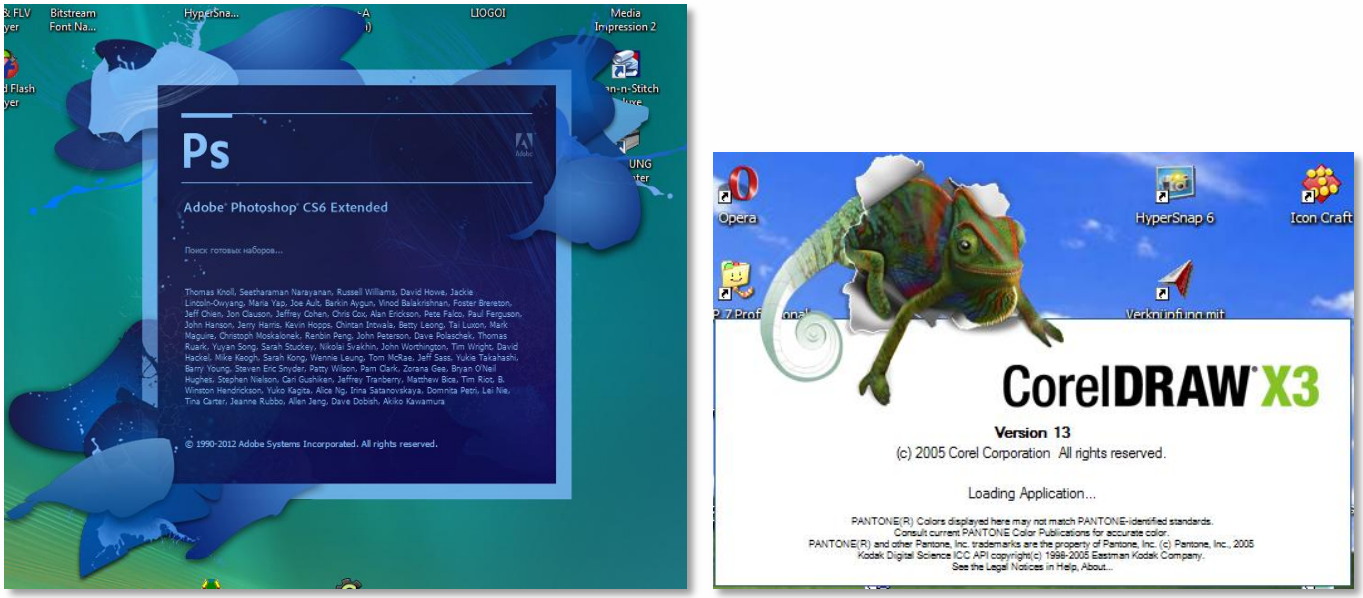


Рис. С6.1. Заставки в популярных программах

Вы и сами можете включить в свой проект заставку ничуть не хуже любой «фирменной».

Возьмём за основу нашего проекта познавательную программу, с помощью которой мы успешно вычисляли жирность человеческого тела. Загрузите её в *Delphi* и скопируйте все файлы в новую папку **Plot** – на этот раз мы займёмся расчётом плотности человеческого тела (на самом деле – удельного веса, но для практических нужд всё едино).

Она близка к плотности воды (1 г на куб. см) и зависит от соотношения количества жира (плотность 0,890 г на куб. см) и мускулов (плотность 1,100 г на куб. см). Формула для подсчёта средней плотности человеческого тела такова:

p = (8,90 x zh + 11 x (100 - zh)) г/см^3.

Легко видеть, что в интерфейсе предыдущей программы нужно сделать минимальные изменения:

Свойство **Caption** формы - *Плотность тела*
Caption метки **Label2** – *Плотность тела (г/куб.см) =*
Name lblZhir – *lblPlot*, а её **Caption** – *1.05*.
Остальные элементы на форме – те же, что и раньше (Рис. С6.2).

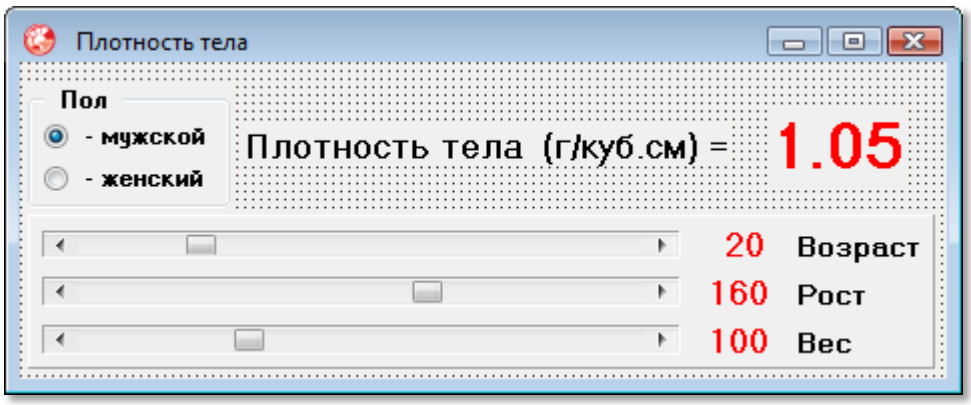


Рис. С6.2. «Плотный» интерфейс

Естественно, придётся поработать и с кодом модуля формы. В ле **private** нужно объявить процедуру, вычисляющую плотность тела:

```
  private
    { Private declarations }
    procedure Plot;
```

Внимательный читатель должен был заметить, что до этого нужные значения вычислялись в функции и затем возвращались в вызывающую её процедуру. Но ещё более внимательный читатель, безусловно, обратил внимание и на то, что дальше они использовались совершенно одинаково – просто выводились на экран. Приходилось одни и те же действия повторять в каждой процедуре. На этом можно сэкономить – будем показывать вычисленное значение там же, где мы его вычисляем. И так как теперь возвращать ничего не нужно, то функцию следует превратить в процедуру:

```
//ВЫЧИСЛИТЬ ПЛОТНОСТЬ ТЕЛА
procedure TForm1.Plot;
var
   v, zh, p: single;
begin
  //вычислить жирность тела:
  if RadioGroup1.ItemIndex= 0 then begin //- мужской пол
    v:= (rost*3 - 450 + age) * 0.25 + 45;
    zh:= (ves - v) / ves * 100 + 15 end
  else begin                              //- женский пол
```

```
    v:= (rost*3 - 450 + age) * 0.225 + 40.4;
    zh:= (ves - v) / ves * 100 + 22 end;

  //скорректировать значения:
  if zh< 0 then zh:= 0;
  if zh> 100 then zh:= 100;

  //вычислить плотность тела:
  p:= 8.90 * zh + 11 * (100 - zh);
  p:= p / 1000;

  lblPlot.Caption:= copy(floattostr(p),1,4);
end;
```

Вам нужно только добавить выделенные строки, чтобы получить достойный результат. Математике вас, конечно, учить не надо, а касательно последней строки можно кое-что сказать: в ней число p преобразуется в строку, из которой выделяются только первые 4 знака, чтобы избавиться от длинной вереницы совершенно бесполезных цифр.

По понятным причинам (беда, если вы их не поняли!) все строчки

```
//вычислить жирность:
lblZhir.Caption:= inttostr(round(zhir))
```

следует заменить строчками:

```
//вычислить плотность:
Plot
```

Запускайте на радость ближним своим новое приложение и проверяйте, утонут ли они в воде (пресной), как и подобает культурному человеку, или будут плавать на поверхности - подобно пресловутому батончику *Milky Way*. Только не удивляйтесь, что никакой заставки вы не увидели – её и нет.

Если жажда новизны ещё не оставила вас, то давайте продолжим мудрить в области программирования. Прежде всего, добавьте к проекту новую пустую форму (о том, как это сделать, говорилось уже много раз), назовите ее **frmSplash** (или как угодно по-другому) и сохраните проект на диске.

У сплаш-формы свойство **BorderStyle** нужно установить в *bsNone*, чтобы убрать полосу заголовка вместе с кнопками. Размеры и оформление формы могут быть любыми. Неплохо смотрится подходящая картинка в рамке, для которой годится компонент **Panel** (найдите его в *Палитре компонентов*!) со свойствами:

```
Left = 0
Top = 0
```

```
Align = alClient
BevelInner = bvLowered
Color = clSilver
```

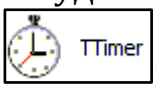
и компонент **Bevel** (для горизонтальной полоски под рисунком) со страницы **Additional**:

```
Left = 0
Height = 2
Style = bsRaised
```

На панель поместите компонент **Image**, а в него загрузите заранее приготовленную картинку. Для прояснения ваших намерений подпишите картинку (компонент **Label**). В итоге в окне *Конструктора формы* вы увидите такую заставку для вашей программы (Рис. С6.3).



Рис. С6.3. Заставка в *Конструкторе формы*

Так как большинство ваших программ будет загружаться настолько быстро, что заставка успеет только мелькнуть на экране, то нам следует позаботиться о продлении её бренного существования. Отмерять срок её жизни удобно с помощью компонента **Timer**. Щёлкните дважды на его значке 

в *Палитре компонентов*, и он появится на форме.

Ещё одна особенность форм-заставок состоит в том, что их нужно создавать самостоятельно, а не автоматически. Поэтому выполните команду *Главного меню* **Project > Options...** и в диалоговом окне **Project Options**, на странице **Forms** выделите название формы-заставки в левом списке и, нажав кнопку 

, перенесите его в правый список **Available Forms** (Рис.

С6.4). Ваша форма не будет создаваться автоматически при запуске программы, теперь это ваша забота.

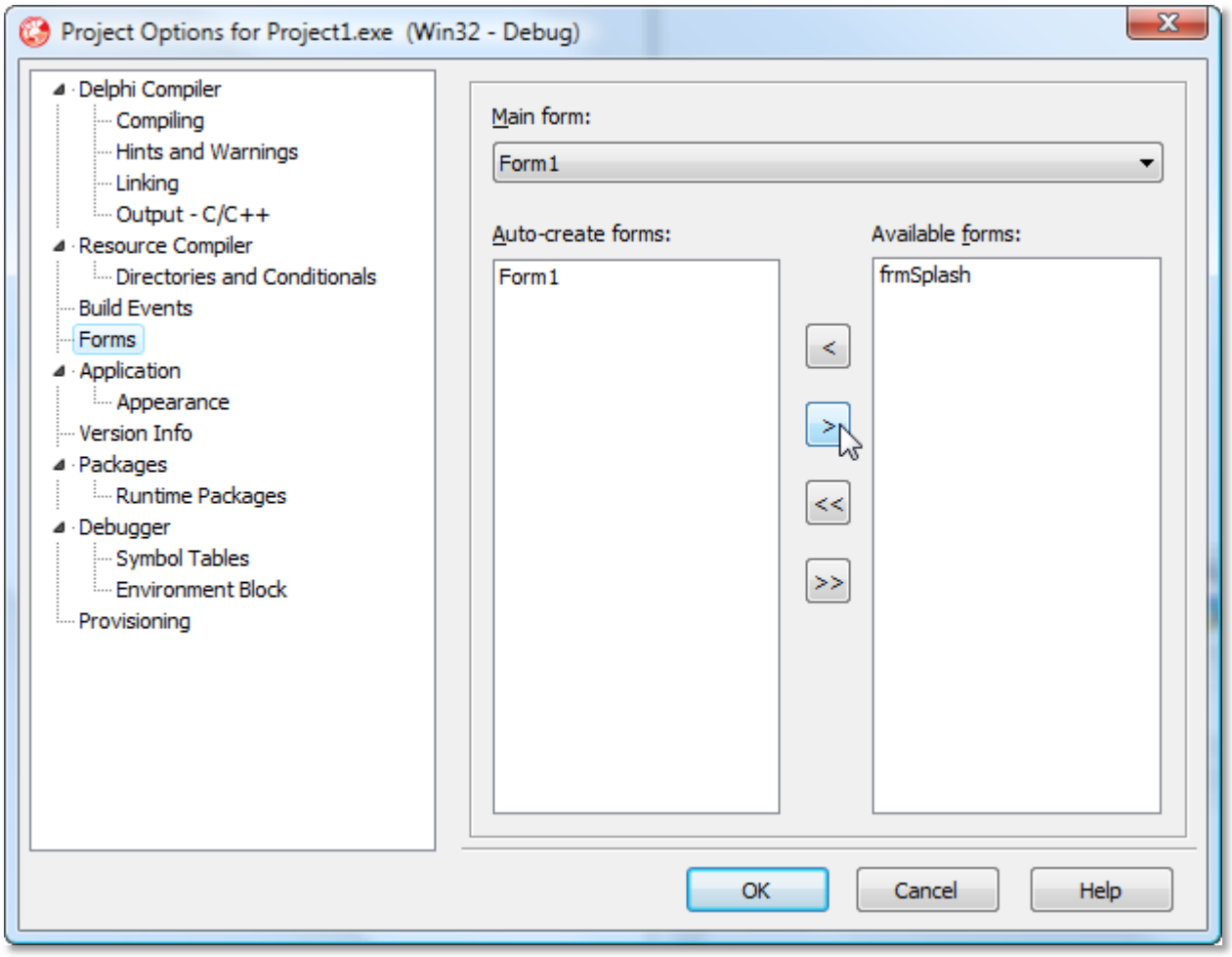


Рис. С6.4. Переносим заставку

А где вы должны сделать это? При создании главной формы? – Но заставка должна появиться до того! И вот тот редкий случай, когда придётся «лезть» в файл проекта. Выполните команду *Главного меню* **Project > View Source**, файл проекта откроется в окне *Редактора кода*. Допишите в него выделенные строчки:

```
program Project1;

uses
  Forms,
  Unit1 in 'Unit1.pas' {Form1},
  Unit2 in 'Unit2.pas' {frmSplash};

{$R *.RES}

begin
  frmSplash := TfrmSplash.Create(Application);
```

```
  frmSplash.Show;
  frmSplash.Update;
  Application.Initialize;
  Application.CreateForm(TForm1, Form1);
  frmSplash.Hide;
  frmSplash.Free;
  Application.Run;
end.
```

Смысл их действия таков. Создаём форму-заставку и отрисовываем её, чтобы она появилась на экране. Затем выполняем стандартные операции по инициализации приложения и, когда всё готово к его запуску, убираем форму-заставку с экрана и выгружаем её из памяти компьютера (и из своей собственной).

Мы должны учесть ещё два обстоятельства. Во-первых, нужно поместить заставку в центре экрана (установите свойство **Position** в *poDesktopCenter*), во-вторых, нужно задержать её на экране, так как обычное приложение загружается очень быстро. Для этого нужно написать такой код в модуле формы-заставки:

```
unit Unit2;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
Dialogs, ExtCtrls, StdCtrls;
```

```
type
  TfrmSplash = class(TForm)
    Panel1: TPanel;
    Bevel1: TBevel;
    Label1: TLabel;
    Image1: TImage;
    Timer1: TTimer;
    procedure Timer1Timer(Sender: TObject);
    procedure FormCloseQuery(Sender: TObject; var CanClose: Boole-
an);
    procedure FormHide(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  frmSplash: TfrmSplash;

implementation

{$R *.DFM}
```

```
//ВЫКЛЮЧИТЬ ТАЙМЕР, ЕСЛИ ВРЕМЯ ЗАКОНЧИЛОСЬ
procedure TfrmSplash.Timer1Timer(Sender: TObject);
begin
  Timer1.Enabled:= False
end;


//МОЖНО ЛИ ЗАКРЫТЬ ФОРМУ?
procedure TfrmSplash.FormCloseQuery(Sender: TObject; var CanClose:
Boolean);
begin
  CanClose:= Timer1.Enabled = False
end;
//УБРАТЬ ФОРМУ, КОГДА КОНЧИТСЯ ВРЕМЯ
procedure TfrmSplash.FormHide(Sender: TObject);
begin
  repeat
    Application.ProcessMessages;
  until frmSplash.CloseQuery
end;
end.
```

Обязательно увеличьте значение свойства **Interval** таймера до 4000-5000 мс (в *Инспекторе объектов*). В этом случае форма-заставка просуществует на экране 4 или 5 секунд. Поначалу свойство **Enabled** таймера равно *True*, и он включится при появлении заставки на экране. После отсчёта установленного интервала времени таймер выключится, и свойство **Enabled** примет значение *False*.

За этим внимательно следит процедура **FormHide**, обрабатывающая событие **OnHide**, вызываемое при выполнении процедуры **frmSplash.Hide.** В ней проверяется, можно ли закрыть форму-заставку, а «добро» на это даёт процедура **TfrmSplash.FormCloseQuery**, которая ждёт, пока таймер отработает заданное время.

Вот теперь ваше приложение обязательно предъявит пользователю свою «визитную карточку»!



Исходный код программы находится в папке **PLOT**.

Искусственное дыхание, или Оживляем форму-заставку

Эй, товарищ, больше жизни!

Вратарская песня

Раз уж мы хотим, чтобы заставка наилучшим образом «презентовала» нашу программу, то давайте добавим ей привлекательности. Пусть она появляется не вдруг, а частями (товарищу Бендеру, конечно, такое решение вряд ли пришлось бы по вкусу, он ведь предпочитал всё сразу!), а для этого воспользуемся функцией *Windows API* **AnimateWindow**, которая «анимирует» окно при появлении на экране и при исчезновении с него:

```
BOOL AnimateWindow(
    HWND hwnd,
    DWORD dwTime,
    DWORD dwFlags
);
```

hwnd – дескриптор окна, в нашем случае для всех окон приложения это **Handle**.
dwTime – продолжительность эффекта в миллисекундах.
dwFlags – константа, определяющая вид эффекта (может состоять из одного или нескольких флагов, соединённых логическим оператором **OR**):

AW_HIDE – если указан этот флаг, то окно **исчезает** с экрана, если нет, то **появляется**;

Эффекты анимации:

AW_SLIDE – окно выдвигается по направлению, задаваемому флагом направления анимации (кроме AW_CENTER). Используется по умолчанию. Если флаг не задан, то изображение не выскальзывает, а раскручивается;

AW_BLEND – окно проявляется или угасает.

Направление анимации:

AW_CENTER - окно появляется из центра. Флаги направления не действуют;

AW_HOR_POSITIVE – окно появляется слева направо. Не действует с AW_CENTER и AW_BLEND;

AW_HOR_NEGATIVE – окно появляется справа налево. Не действует с AW_CENTER и AW_BLEND;

AW_VER_POSITIVE – окно появляется сверху вниз. Не действует с AW_CENTER и AW_BLEND;

AW_VER_NEGATIVE – окно появляется снизу вверх. Не действует с AW_CENTER и AW_BLEND.



Чтобы показать окно без эффектов анимации, можно воспользоваться другой функцией:

BOOL ShowWindow(HWND *hWnd*, int *nCmdShow*);

Эффект анимации уместно использовать при появлении окна на экране (событие **OnShow**) и при исчезновении с оного (событие **OnClose** или **OnHide**). Пусть у нас «эффектно» появляется и исчезает окно-заставка .
В текст модуля формы **Unit2** добавим процедуру:

```
//ВЫВЕСТИ ОКНО НА ЭКРАН С ЭФФЕКТАМИ АНИМАЦИИ
procedure TfrmSplash.FormShow(Sender: TObject);
begin
  AnimateWindow(Handle, DURATION, AW_CENTER);
  //AnimateWindow(Handle, DURATION, AW_BLEND);
  //AnimateWindow(Handle, DURATION, AW_SLIDE);
  //AnimateWindow(Handle, DURATION, AW_HOR_POSITIVE or AW_SLIDE);
  //AnimateWindow(Handle, DURATION, AW_HOR_POSITIVE);
  //AnimateWindow(Handle, DURATION, AW_HOR_NEGATIVE or
AW_ACTIVATE);
  //AnimateWindow(Handle, DURATION, AW_VER_POSITIVE);
  //AnimateWindow(Handle, DURATION, AW_VER_NEGATIVE);
end;
```

В данном случае окно будет расти из центра к краям – мы используем функцию **AnimateWindow(Handle, DURATION, AW_CENTER),** но вы можете поэкспериментировать и с другими аргументами. Например, раскомментируйте строчки (по очереди), запускайте программу и наблюдайте за появлением формы-заставки на экране.

Но прежде добавьте ещё немного кода в модуль:

```
Implementation
 const DURATION: integer=1000;
//СОЗДАТЬ ОКНО В ЦЕНТРЕ ЭКРАНА
procedure TfrmSplash.FormCreate(Sender: TObject);
begin
  left:= (screen.Width-width) div 2;
  top:= (screen.Height-height) div 2;
end;
```

Подберите значение константы **DURATION** по своему вкусу, а затем допишите выделенную строку в процедуру, которая убирает форму с экрана, - теперь она будет исчезать гораздо эффектней, стягиваясь («коллапсируя») к центру:

```
//УБРАТЬ ФОРМУ, КОГДА КОНЧИТСЯ ВРЕМЯ
```

```
procedure TfrmSplash.FormHide(Sender: TObject);
begin
  repeat
    Application.ProcessMessages;
  until frmSplash.CloseQuery;
  AnimateWindow(Handle, DURATION, AW_CENTER or AW_HIDE)
end;
```



Аналогично вы можете обойтись и с главным окном программы, если не считаете, что это будет перебор.



Исходный код программы находится в папке **AnimateForm**.

Бледная личность, или Уходящая натура

А вы можете исчезать и
появляться не так внезапно?
А то у меня голова идет кругом.
- Хорошо, - сказал Кот и исчез –
на этот раз очень медленно.
Первым исчез кончик его хвоста,
а последней - улыбка;
она долго парила в воздухе,
когда все остальное уже пропало.

Льюис Кэрролл.
Приключения Алисы в стране чудес

В «старших» версиях *Delphi* у формы появилось одно замечательное свойство – **AlphaBlendValue,** -, которое вполне может послужить основой для анимации формы.

Значение этого свойства, равное **0**, делает окно абсолютно прозрачным, а максимальное значение **255** – абсолютно непрозрачным. Все промежуточные значения этого свойства придают окну частичную прозрачность, то есть сквозь него будут «просвечивать» окна, находящиеся ниже (Рис. С6.5).



Рис. С6.5. *AlphaBlendValue* = 50 = 125 = 200

Полупрозрачные окна хороши уже сами по себе и их вполне можно использовать в программах для вывода окон справки, настройки, помощи и тому подобных, но мы пойдём ещё дальше и при появлении формы на экране будем постепенно уменьшать её (не)прозрачность от 0 до 255.

Форма появляется на экране совершенно *прозрачной*:

```
procedure TForm2.FormShow(Sender: TObject);
begin
  AlphaBlend:= TRUE;
  AlphaBlendValue:= 0;
end;
```

Обратите внимание, что мы должны установить флажок *AlphaBlend*, иначе свойство **AlphaBlendValue** не будет действовать!

Затем в цикле мы плавно уменьшаем прозрачность нового окна (если вы хотите оставить окно слегка непрозрачным, чтобы добавить шарма, то вычтите в выделенной строке некоторое число из значения **AlphaBlendValue**):

```
procedure TForm2.FormActivate(Sender: TObject);
var i,n: integer;
begin
  n:= 100;
  for I := 0 to n do begin
```

```
      AlphaBlendValue:= 255 * i div n;//-100
      delay(10);
  end;
end;
```

В процедуре **delay** цикл делает небольшую паузу, чтобы эффект анимации стал доступен человеческому глазу:

```
//ЗАДЕРЖКА
procedure Delay(MSecs: Longint);
var
  FirstTickCount, Now: Longint;
begin
  FirstTickCount := GetTickCount;
  repeat
    Application.ProcessMessages;
    Now := GetTickCount;
  until (Now - FirstTickCount >= MSecs) or (Now < FirstTickCount);
end;
```

С точностью до наоборот при закрытии программы мы стираем окно с экрана (Рис. С6.6):

```
procedure TForm2.FormClose(Sender: TObject; var Action: TCloseAc-
tion);
var i,n: integer;
begin
  n:= 100;
  for I := n downto 0 do begin
     AlphaBlendValue:= 255 * i div n;
     delay(10);
  end;
end;
```

Правда, в отличие от Чеширского Кота от него не останется даже улыбки. Впрочем, может быть, она останется у вас – как результат хорошо сделанной работы.

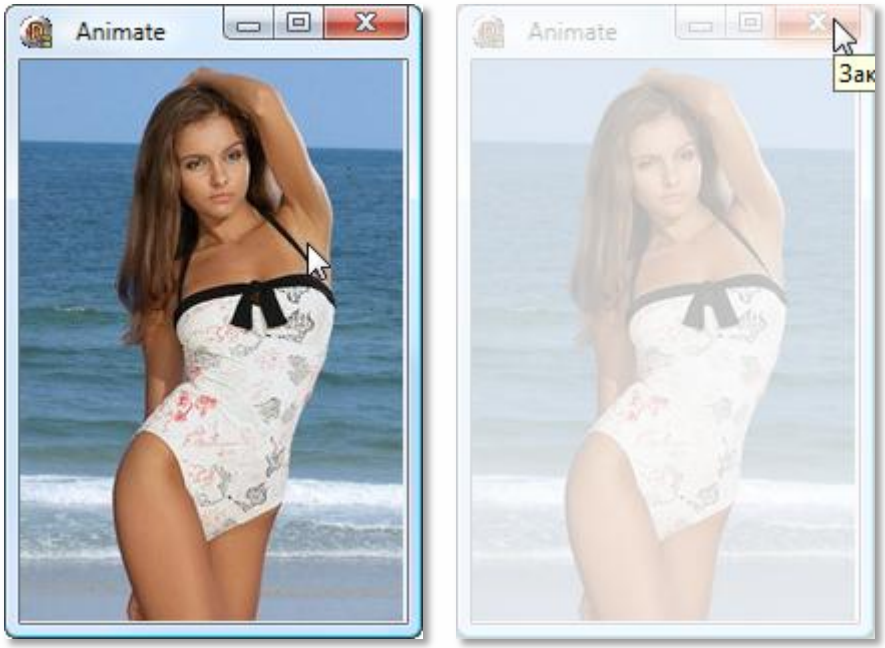


Рис. С6.6. Форма растворяется



Исходный код программы находится в папке **AnimateTransparent**.

Урок 7. Операции и выражения

- Доктор, операцию делать будем?
- Да, ладно, пускай живёт!

Анекдот о самой гуманной профессии

Операции

Операции - это действия, которые применяют к объектам программы (операндам) – переменным, константам, обращениям к процедурам и функциям.

Знаки операций (**операторы**) предназначены для обозначения этих действий. Это могут быть особые символы: **+ - * / < > =,** а также зарезервированные слова: **not**, **div**, **mod**, **or**. Как видите, некоторые из них вам хорошо известны по урокам математики.



При использовании зарезервированного слова в качестве знака операции он должен быть отделён от операндов хотя бы одним пробелом!

Операции делятся на **унарные** (одноместные) и **бинарные** (двуместные). В унарных операциях участвуют знак операции и следующий за ним операнд:

```
-2
+variant
```

В большинстве операций участвуют 2 операнда, и знак операции записывается между ними. Такие операции называются бинарными:

```
2 + 5
2 * 2
a * sin(alpha)
```

Для каждого типа данных существует свой набор операций, поэтому операнды должны быть одного типа или совместимых типов. Нельзя складывать числа со строками или сравнивать их между собой!

На этом уроке мы рассмотрим только **операции с простыми типами данных**. Операции с массивами, строками, записями, классами будут рассмотрены на уроках, посвящённых этим структурированным типам данных.

Арифметические операции с целыми числами

Результат всех операций, кроме деления, имеет *целый* тип.

Пусть

```
Var n,m: integer;
. . .
n:=3;
m:=2;
```

+ - унарный плюс. Ставится перед операндом и никак на него не влияет.

- - унарный минус. Ставится перед операндом и заменяет его знак противоположным.

+ - сложение. Результат операции равен сумме двух операндов:

```
m + n → 5
```

- - вычитание. Результат операции равен разности двух операндов:

```
m – n → –1
```

* - **умножение**. Результат операции равен произведению двух операндов:

```
m * n → 6
```

/ - деление. Результат операции равен частному от деления первого операнда на второй и имеет вещественный тип (**extended**):

```
m / n → 0,666666666666667
```



На ноль делить нельзя!

div – целочисленное деление. Результат операции равен частному от деления первого операнда на второй:

```
m div n → 0
n div m → 1
```

Если сравнить целочисленное деление с делением вещественных чисел, то

```
m div n = m / n,
```

округлённому до ближайшего целого в сторону нуля.

mod – остаток от целочисленного деления (**модуль**, деление по модулю). Результат операции равен остатку от деления первого операнда на второй:

```
m mod n → 2
n mod m → 1
```



Оператор **mod** часто используют для генерации периодически повторяющейся последовательности целых чисел. Например, нам нужна такая последовательность: 0,1,2,3,4,5,6, 0,1,2,3,4,5,6, …

Введём 2 переменные:
Var n, m: integer;
. . .
Обнулим счётчик:
n:= 0;

//циклически выполняющийся код:
m:= n mod 7;
inc(n);

Переменная **m** будет последовательно принимать нужные нам значения!

Арифметические операции с вещественными числами

Результат всех операций имеет вещественный тип, если хотя бы один из операндов вещественного типа (один из них может быть и целым).

С вещественными числами допустимы те же операции, что и с целыми числами, за исключением **div** и **mod**.

Операции отношения

Предназначены для **сравнения** двух операндов совместимых типов (но целые и вещественные числа **могут** сравниваться между собой). **Результат** всегда имеет логический тип и равен *TRUE*, если указанное соотношение выполняется, и *FALSE* в противном случае.

```
Var x, y: integer;
. . .
X:= 7;
Y:= 13;
```

= - равно. Результат равен *TRUE*, если операнды равны, и *FALSE*, если не равны:

```
7 = 13 → FALSE
7 = 2 + 5 → TRUE
```

< - меньше. Результат равен *TRUE*, если первый операнд меньше второго, и *FALSE*, если больше или равен:

```
7 < 13 → TRUE
7 <  3 → FALSE
7 <  2 + 5 → FALSE
```

<= - меньше или равно (не больше). Результат равен *TRUE*, если первый операнд меньше второго или равен ему, и *FALSE*, если больше:

```
7 <= 13 → TRUE
7 <=  3 → FALSE
7 <=  2 + 5 → TRUE
```

> - больше. Результат равен *TRUE*, если первый операнд больше второго, и *FALSE*, если меньше или равен:

```
7 > 13 → FALSE
7 >  3 → TRUE
7 >  2 + 5 → FALSE
```

>= - больше или равно (не меньше). Результат равен *TRUE*, если первый операнд больше второго или равен ему, и *FALSE*, если меньше:

```
7 >= 13 → FALSE
7 >=  3 → TRUE
```

```
7 >=  2 + 5 → TRUE
```

<> - не равно. Результат равен *TRUE*, если операнды не равны, и *FALSE*, если равны:

```
7 <> 13 → TRUE
7 <> 2 + 5 → FALSE
```

Логические операции

Применяются к **логическому** типу данных, и **результат** также имеет логический тип. Операция логического отрицания унарная, остальные две – бинарные.

NOT - оператор логического отрицания **НЕ**:

```
Not False → TRUE
Not True  → FALSE
```

По этим примерам видно, что в результате выполнения этой операции истинное выражение становится ложным, и наоборот:

```
Var x, y: integer;
. . .
X:= 7;
Y:= 13;

Not (7 > 13) → TRUE
Not (7 < 13) → FALSE
```

AND - логическая операция **И** (**конъюнкция**).
Результат операции тогда и только тогда будет истинным, если истинны одновременно оба операнда:

```
(7 >  13) and (7 < 13)  → FALSE
(7 <> 13) and (7 < 13)  → TRUE
```

OR - логическая операция **ИЛИ** (**дизъюнкция**).
Результат операции будет истинным, если хотя бы один операнд истинен:

```
(7 > 13)  or (7 < 13)  → TRUE
(7 <> 13) or (7 < 13)  → TRUE
```

XOR - логическая операция **ИСКЛЮЧАЮЩЕЕ ИЛИ** (**исключающая дизъюнкция**).

Результат операции тогда и только тогда будет истинным, если один операнд истинен, а второй ложен, во всех остальных случаях результат будет ложным:

```
(7 > 13)  xor (7 < 13)   → TRUE
(7 <> 13) xor (7 < 13)   → FALSE
```



В *Delphi* по умолчанию логические выражения вычисляются по **укороченному способу**. Это значит, что вычисления проводятся до тех пор, пока не станет заведомо известно значение всего выражения, после чего вычисления прекращаются. Например, рассмотрим выражение

```
(7 > 13) and (7 < 13)   → FALSE
```

Значение первого операнда равно *FALSE*, поэтому и результат вычисления всего операнда равен *FALSE*, даже если второй операнд истинен. Поэтому значение второго операнда вообще не вычисляется. Этот приём не только сокращает время вычислений, но и помогает избегать ошибок в программе:

```
If (n <> 0) and (m div n = 10) then  m:= m +1;
```

В данном примере сначала проверяется, не равен ли делитель нулю. Если это так, то дальнейшие вычисления не производятся. Если же используется **полное вычисление** логического результата (включается директивой компилятора **{$B+}**), то при делении на ноль возникнет ошибка.

Логические поразрядные операции (операции с битами)

Применяются к отдельным битам **целых чисел**.

NOT – унарная операция **инверсии** всех битов числа. Все нулевые биты становятся единичными, и наоборот. Так, число 7 (типа **byte**) 00000111 превратится в 11111000, то есть в 248.

AND – побитовая логическая операция **И** двух чисел. Если в одном и том же разряде в обоих числах стоят единицы, то и в результате будет единица, во всех остальных случаях – ноль:

```
7 and 13 =   00000111 and
             00001101
         →   00000101 = 5
```

OR – побитовая логическая операция **ИЛИ** двух чисел. Если в одном и том же разряде хотя бы в одном числе стоит единица, то и в результате будет единица, а ноль будет только тогда, когда в обоих числах стоят нули:

```
7 or 13 =    00000111 or
             00001101
         →   00001111 = 15
```

XOR – побитовая логическая операция **ИСКЛЮЧАЮЩЕЕ ИЛИ** двух чисел. Если в одном и том же разряде в числах стоят разные биты (0 и 1 или 1 и 0), то врезультате будет единица, если одинаковые – ноль:

```
7 xor 13 =   00000111 xor
             00001101
         →   00001010 = 10
```



Если операцию **xor** выполнить 2 раза, то результат будет равен первому операнду:

```
7 xor 13  xor 13 = 7;
```

SHL – **сдвиг** всех битов первого операнда **влево** на число разрядов, равное второму операнду. Освободившиеся справа разряды заполняются нулями.

Например, результат выполнения операции **7 shl 2** равен 28. Чтобы разобраться, как происходит сдвиг, нужно представить себе первое число в двоичном виде.

7 – это 111. Сдвигаем влево на два разряда и получаем число 11100, или в десятичном виде 28. Таким образом, сдвиг влево на 1 разряд равносилен умножению первого операнда на 2, на 2 разряда – на 4, на 3 разряда – на 8, и так далее. Поскольку сдвиг выполняется быстрее умножения, то для умножения числа на степень двойки лучше использовать сдвиг влево.

SHR – **сдвиг** всех битов первого операнда **вправо** на число разрядов, равное второму операнду. Освободившиеся слева разряды заполняются нулями. Опять сдвинем биты числа 7 на 2 разряда, но уже вправо:

111 → 001, то есть получим единицу. Таким образом, сдвиг вправо на 1 разряд равносилен целочисленному делению первого операнда на 2.

Приоритеты операций

Список операторов по уменьшению их *приоритета*:

Вычисления в круглых скобках
Вычисления функций
- + унарные
NOT
*** / DIV MOD AND SHL SHR**
+ - OR XOR
= < > <= >= <>

Операции с более высоким приоритетом выполняются раньше операций, приоритет которых ниже. Если приоритет операций одинаков, то они выполняются последовательно слева направо.

Выражения

Представляют собой формулы для вычисления значения, составленные с помощью знаков операций, операндов и скобок, то есть это одна или больше операций, объединённых операторами.

В частном случае это могут быть просто отдельные переменные и обращения к процедурам и **функциям**:

```
var
  n: integer;
. . .
n;

procedure TForm1.FormCreate(Sender: TObject);
begin
  newGame();
end;
```

Последовательность выполнения действий в *Delphi*, как и в школьной математике, определяется старшинством (приоритетом) операций и может

быть изменена с помощью круглых скобок (но не квадратных и не фигурных!).

При формировании логических выражений **каждую логическую операцию следует заключать в скобки**:

```
var
  b: boolean;
…
b:=(7 <> 13) and (7 < 13);
```

При такой записи

```
b:=7 <> 13 and 7 < 13;
```

компилятор выдаст ошибку несоответствия типов, поскольку порядок выполнения операций будет совсем не тот, на который вы могли рассчитывать:

```
b:=7 <> (13 and 7) < 13;
```

То есть `b:= (7 <> 5) < 13;` и далее `b:= TRUE < 13;`

Так как мы не имеем права сравнивать логический тип с целым, то и получаем по полной программе!

Семинар 7. Одно- и многооконные приложения, или Сколько кожи на человеке

Многие приложения имеют несколько дополнительных окон (форм) – для показа справки помощи, ведения протокола, загрузки файлов с диска и по другим причинам. По отношению к главному окну программы они могут вести себя по-разному. Как – с этим мы и разберёмся на семинаре.

Однодокументный интерфейс

Иначе – **SDI** (**Single Document Interface**). Все проекты *Delphi* по умолчанию имеют именно такой интерфейс (свойство **FormStyle** равно *fsNormal*). Однодокументные приложения содержат одну главную форму и любое количество дочерних, которые могут запускаться как из главной формы, так и из других дочерних. При этом дочерние формы могут располагаться в любом месте экрана и перекрывать главную форму.

Главной обычно является та форма, которая создаётся первой, но в диалоговом окне **Project Options** (открывается по команде *Главного меню* **Project > Options...)** вы можете перепоручить эту роль любой другой форме проекта. Дочерние формы создаются автоматически при старте приложения или программно (список тех и других форм также можно отредактировать в диалоговом окне **Project Options**). Если дочерняя форма выводится на экран методом **ShowModal**, то нельзя перейти на другие окна приложения, пока она не закрыта. При сворачивании главного окна закрываются и все дочерние.

Формы в **SDI**-приложении связаны между собой через данные и элементы управления, для чего они должны быть объявлены в модуле той формы, в которой есть ссылка на элементы (компоненты, переменные и так далее) другой формы. Рассмотрим это на примере.

Кожа да кости!

Требование Славы Зайцева к моделям

Не лезьте из кожи!

Царевна Лягушка

Продолжим составлять познавательные программы. Давайте вычислим площадь поверхности человеческого тела (кожи). С точки зрения геомет-

рии, фигура человека весьма «вычурна», поэтому не существует точных формул для определения площади тела. Мы воспользуемся формулой Бойде, которая позволяет приближённо вычислить нужную величину (не пугайтесь - формула заковыристая):

$$S = (P \times 1000)^{(\lg(1/P)+35{,}75)\,/\,53{,}2} \times H^{0{,}3} : 3118{,}2 \quad (м^2),$$

где **P** – вес человека в *кг*, **H** – его рост в *см*.

Так как для вычисления площади тела нужно знать рост и вес человека, то за основу нашей программы возьмём предыдущее наше творение, убрав из него всё лишнее. После чего форма программы будет выглядеть вот так (Рис. С7.1).

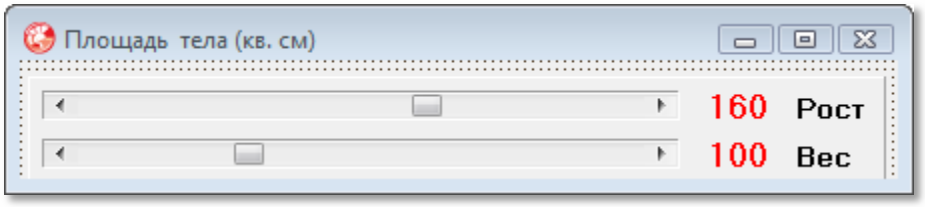


Рис. С7.1. «Кожистый» интерфейс

Несколько исправлений придётся внести и в модуль формы. Сотрите все ссылки на ненужные элементы, измените название процедуры и добавьте модуль **math**, без которого невозможно провести вычисления по нашей страшной формуле:

```
unit Unit1;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
Dialogs,  StdCtrls, ExtCtrls, math;

type
  TForm1 = class(TForm)
    Panel1: TPanel;
    lblRost: TLabel;
    Label4: TLabel;
    lblVes: TLabel;
    Label5: TLabel;
    sbRost: TScrollBar;
    sbVes: TScrollBar;
    procedure sbVesChange(Sender: TObject);
    procedure sbRostChange(Sender: TObject);
    procedure FormShow(Sender: TObject);
```

```
  private
    { Private declarations }
    procedure Kozha;

  public
    { Public declarations }

  end;

var
  Form1: TForm1;
  rost, ves: integer;
```

Вычислять площадь тела мы будем в процедуре:

```
//ВЫЧИСЛИТЬ ПЛОЩАДЬ ТЕЛА
procedure TForm1.Kozha;
var
   s: Extended;
begin
  s:= (log10(1 / ves)+ 35.75) / 53.2;

  //вычислить площадь тела:
  s:= power((ves * 1000), s) *  power(rost, 0.3) / 0.31182;
  Form2.lblKozha.Caption:= Format('%5.0f', [s])
end;
```

Переводим формулу Бойде на математический диалект дельфийского языка и выводим на экран площадь тела в квадратных сантиметрах (отбрасываем десятичные знаки с помощью функции **Format**): как говорил Удав, «А в попугаях я гораздо длиннее!».

Добавим к главной форме одну дочернюю (форма-заставка нам здесь не понадобится, поэтому исключите её из проекта, а из модуля проекта удалите все строчки, где она упоминается). Делаем это, как обычно, а самой форме придаём нужный вид (не забудьте записать новый проект в папку **KOZHA**!) (Рис. С7.2).

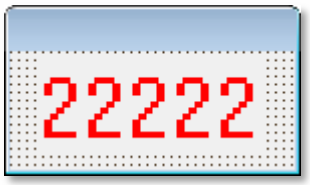


Рис. С7.2. Дочерняя форма

Убираем кнопки из заголовка - в свойстве **BorderIcons** все значения устанавливаем в *False*: *BorderIcons = [].* Оставляем только заголовок (**BorderStyle** = *bsDialog* , **Caption** = ' '), чтобы форму можно было передвинуть на удобное место. Так как эта форма всегда должна быть на виду, задаём значение **FormStyle** = *fsStayOnTop*.

Вычисленную величину выводим в метке **lblKozha**:

```
Font.Color = clRed
Font.Height = -40
Font.Style = [fsBold].
```

Подправляем процедуры, доставшиеся нам в наследство:

```
//ЗАДАТЬ РОСТ
procedure TForm1.sbRostChange(Sender: TObject);
begin
  //рост:
  Rost:= sbRost.Position;

  //вывести значение на экран:
  lblRost.Caption:= inttostr(Rost);

  //вычислить площадь кожи:
  Kozha
end;

//ЗАДАТЬ ВЕС
procedure TForm1.sbVesChange(Sender: TObject);
begin
  //вес:
  ves:= sbVes.Position;

  //вывести значение на экран:
  lblVes.Caption:= inttostr(Ves);

  //вычислить площадь кожи:
  Kozha
end;
```

Процедуру **FormCreate** заменим процедурой **FormShow**, так как теперь нам нужно дождаться, пока будет создана и дочерняя форма, иначе мы не сможем вывести значение площади кожи, вычисленное по установленным движкам.

```
procedure TForm1.FormShow(Sender: TObject);
begin
  //снять значения со скроллеров:
  Rost:= sbRost.Position;
  Ves:= sbVes.Position;
```

```
  //вычислить площадь кожи:
  Kozha;

  //выводим главную форму в центре экрана:
  left:= (screen.width - width) div 2;
  top:= (screen.height - height) div 2;
  //а дочернюю - справа от неё:
  Form2.Left:= left+width;
  Form2.Top:= top;
  Form2.Show;

end;
```

Если вы теперь запустите программу, то на экране возникнет такое сообщение (Рис. С7.3).

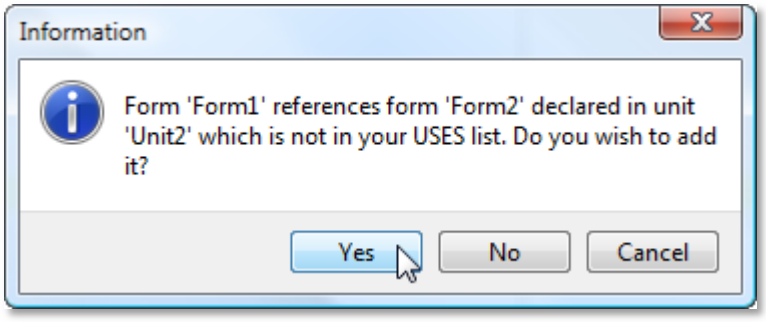


Рис. С7.3. Что-то мы забыли!

Форма Form1 ссылается на форму Form2, объявленную в модуле Unit2, которого нет в списке USES. Хотите ли вы добавить его? Смело нажимайте кнопку **Yes**, и *Delphi* «позаботится» о вас, дописав в модуль главной формы выделенную строку (конечно, этот «текст» вы можете впечатать и сами, не обременяя *Delphi*):

```
 . . .

implementation
uses Unit2;

{$R *.DFM}

. . .
```

А вот сейчас жмите на **RUN** с полной силой – ожидания не обманут вас: на экране во всей красе предстанут *обе* формы (Рис. С7.4).

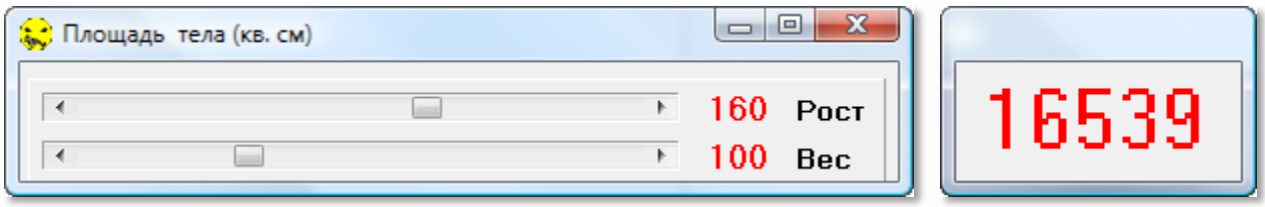


Рис. С7.4. Двухформенное приложение

С их помощью вы сможете вычислить площади всех доступных вам тел, к вящему удовольствию их «владельцев» (если они ещё не потеряли вкус к жизни после первых трёх экспериментов).



Исходный код программы находится в папке **KOZHA**.

Самостоятельная работа



1. Посчитайте, сколько татуировок сможет разместить на себе Владимир Винокур, если средняя площадь татуировки равна 100 квадратным сантиметрам.

2. Интерфейс **SDI** наиболее «популярен» в приложениях (да и сама среда *Delphi* являет собой пример такого приложения). Все программы в этой книге также используют только однодокументный интерфейс.

   Для облегчения разработки подобных приложений в **Репозитории** (страница **Delphi Projects (**Рис. С7.5, слева) имеется шаблон со всеми атрибутами современного интерфейса **SDI** (Рис. С7.5, справа).

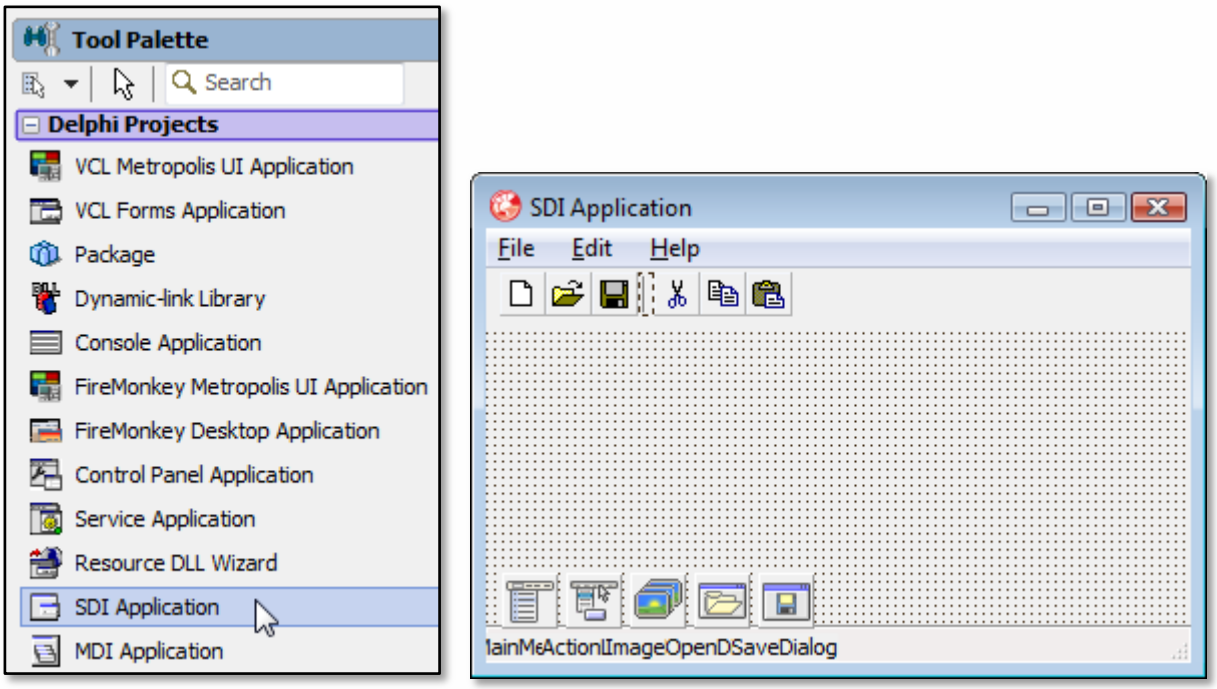


Рис. С7.5. Создание однодокументного интерфейса из шаблона

Многодокументный интерфейс

В оригинале – **Multi Document Interface**, **MDI**. Приложение с этим интерфейсом имеет одну главную форму (её свойство **FormStyle** должно быть установлено в *fsMDIForm*) и любое количество дочерних (со свойством **FormStyle =** *fsMDIChild*), которые, в отличие от интерфейса **SDI**, могут располагаться только в клиентской области главного окна. При сворачивании главного окна, сворачиваются и все дочерние. Примером подобного приложения может служить *Редактор ресурсов* **Image Editor**.

Шаблон многодокументного приложения можно взять в **Репозитории** (Рис. С7.6).

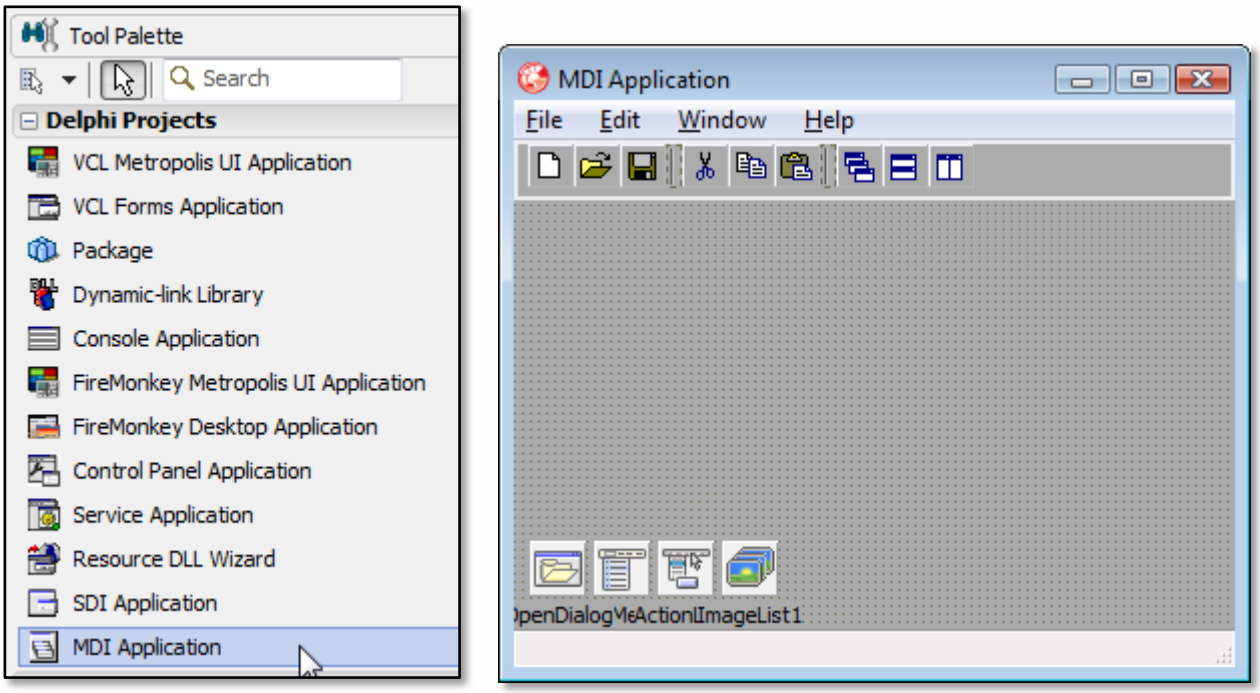


Рис. С7.6. Создание однодокументного интерфейса из шаблона

и изменить его в соответствии с потребностями вашей программы.

При программировании игр интерфейс **MDI** встречается крайне редко, поэтому не будем на нём «тормозить» своё внимание.

ЧВС «Фенечки, или "Остероиды"

Делу время, потехе час.

Наш режим дня

Бросая в воду камешки, смотри на круги,
ими образуемые; иначе такое бросание
будет пустою забавою.

Козьма Прутков

Подыстощив свою нервную систему серьёзными мероприятиями, мы должны подумать и о нравственном здоровье нации в своём лице, то есть развлечься невинными пустяками, - впрочем, тоже не без пользы. Давайте напишем малюсенькие программульки, которые смогут послужить основой для больших приколов (за более вредными советами обращайтесь к их папе – писателю Григорию Остеру).



Теперь уж мало кто об этом помнит, но в стародавние времена в школах время от времени проводили интеллектуальные игрища, которые назывались ЧВС, то есть *Час Весёлых Состязаний*. Это была некая школьная интерпретация популярного тогда КВН (его первого пришествия, а не того, который мы имеем удовольствие наблюдать сейчас). Присутствовали там, конечно, и шутки-прибаутки, но ученики не чурались и более серьёзных заданий из школьной программы.

До свиданья, форма дорогая!

Откройте новый проект и сохраните его в папке. Поместите на форму метку и установите для неё размер шрифта покрупнее (Рис. С7.7).



Рис. С7.7. Форменная метка

Чтобы форма всегда была видна, но не мешала обзору, задайте ей перцу:

```
  AutoSize = True
  BorderStyle = bsNone
```

Добавьте к описанию типа формы две выделенные строки:

```
  public
    { Public declarations }

  protected
     Procedure LastFocus(var Mess : TMessage) ;
sage  WM_ACTIVATE;
```

Теперь мы сможем обрабатывать сообщение *Windows* **WM_ACTIVATE**, которое окно получает при потере и получении фокуса ввода (не активная/активная форма). Если значение флага **fActive** в сообщении **WM_ACTIVATE** равно *WA_INACTIVE*, то форма становится неактивной (теряет фокус), в противном случае (флаг **fActive** равен *WA_ACTIVE* или *WA_CLICKACTIVE*) – активной (получает фокус). Мы художественно оформим эти события: в метку **label1** будем выводить соответствующие надписи:

```
//ФОРМА ПОЛУЧАЕТ ИЛИ ТЕРЯЕТ ФОКУС
Procedure TForm1.LastFocus(var Mess : TMessage) ;
var
  s: string;
  cl: TColor;
Begin
  IF  Mess.wParam = WA_INACTIVE
       Then Begin  cl:= clBlue; s:= 'Чао!' end
  Else Begin  cl:= clRed; s:= 'Привет!'end;
  label1.Color:= cl;
  label1.Caption:= s;
  Inherited;
End;
```

Естественно, вы можете «побезобразничать» и по-своему, как говорится, в меру своей испорченности.

Поскольку у нашей формы нет «ни окон, ни дверей», то будем закрывать её двойным щелчком на метке (почему не по форме?):

```
//ЗАКРЫТЬ ФОРМУ
procedure TForm1.Label1DblClick(Sender: TObject);
begin
  close
end;
```

Запустите программу – она порадует вас словом «Привет!». Ради этого стоило постараться! Но только вы щёлкните мышкой на каком-нибудь другом окне, как наша форма – тоже вежливо! – попрощается с нами (Рис. С7.8).



Рис. С7.8. Вежливое приложение

Что называется, культуру – в массы! Ну, разве не прелесть, эта мулечка? То есть фенечка… А ведь есть и другие!



Исходный код программы находится в папке **FOCUS**.

О кнопках системных замолвите слово…

Есть и другая возможность заставить форму «огрызнуться» на действия неискушённого пользователя. Аналогично первому случаю допишите пару строк:

```
  protected

     Procedure WMGetSysCommand(var Message : TMessage); message
WM_SYSCOMMAND;
```

В этот раз мы будем обрабатывать сообщение *Windows* **WM_SYSCOMMAND**, которое ОС посылает окну при различных манипуляциях с ним. По значению параметра **uCmdType** этого сообщения мы можем судить о том, какая из системных кнопок была нажата:

```
 //ОБРАБОТКА НАЖАТИЯ КНОПОК НА ЗАГОЛОВКЕ ФОРМЫ
Procedure TForm1.WMGetSysCommand(var Message : TMessage) ;
var s: string;
Begin
  s:='';
  case Message.wParam of
    SC_CLOSE: s:= 'Не закрывай меня!';
    SC_MINIMIZE: s:= 'Не сворачивай меня!'; //== SC_ICON
    SC_MAXIMIZE: s:= 'Не разворачивай меня!'; //== SC_ZOOM
  end;

  if s<> '' Then showmessage(s)

  Else Inherited;

End;
```

А уж как вы сумеете прокомментировать действия пользователя, зависит только от вашего остроумия!

Закрыть форму можно, просто щёлкнув на ней:

```
//ЗАКРЫТЬ ФОРМУ
procedure TForm1.FormClick(Sender: TObject);
begin
  close
end;
```

Конечно, картинка на экране будет без всяких там прикрас (Рис. С7.9).

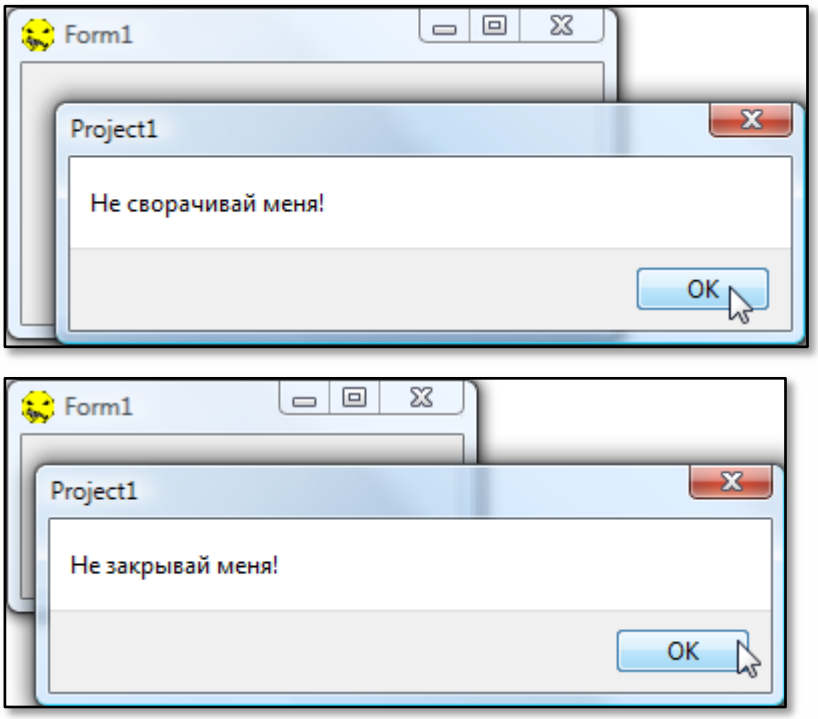


Рис. С7.9. Разговорчивая форма



Подумайте сами, как сделать её симпатичнее!



Исходный код программы находится в папке **FRMBUTTON**.

"МЕНЮатюра"

А вот и третья возможность пошкодить в своей программе. Теперь жертвой наших экспериментов падёт системное меню формы. Как вы помните, оно появляется при нажатии правой кнопки мыши на заголовке формы. Об этом мы там и напишем (**Caption** = *Нажмите правую кнопку мышки на заголовке*) (Рис. С7.10).

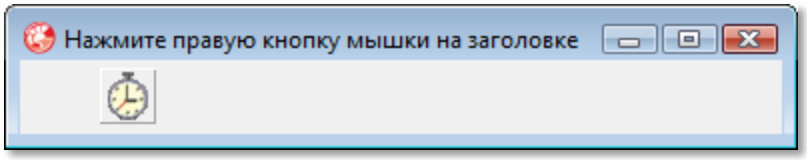


Рис. С7.10. «Менюатюрное» приложение

Чтобы на форме не оставалось пустот, установите её свойство **AutoSize** в *True*.

Поместите на форму метку **Label1** - для вывода смехотворных сообщений. Её свойства:

```
Left = 0,     Top = 0
Caption = '                                                    '
Font.Height = -32,     Font.Style = [fsBold]
```

Пригодится нам и таймер **Timer1** – для смены этих «мессажей»:

```
Enabled = False
Interval = 3000
```

Чтобы добавить новые пункты в системное (и не только) меню, воспользуемся функцией *Windows* **API AppendMenu**, а «отлавливать» нажатие новых пунктов меню будем, обрабатывая уже известное нам сообщение *Windows* **WM_SYSCOMMAND**. Поэтому объявим процедуры **WMSysCommand**, **OnMyMenu** и **OnMyMenu2** в описании типа формы, а также ты *SC_MyMenuItem* и *SC_MyMenuItem2*, нужные при вызове функции **AppendMenu** (значения *0.. WM_USER-1* зарезервированы для системных сообщений).

```
unit Unit1;

interface
```

```
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
Dialogs, StdCtrls, ExtCtrls;

 type
  TForm1 = class(TForm)
    Label1: TLabel;
    Timer1: TTimer;
    procedure FormCreate(Sender: TObject);
    procedure OnMyMenu;
    procedure OnMyMenu2;
    procedure Timer1Timer(Sender: TObject);

  private
    { Private declarations }
    procedure WMSysCommand(var Msg: TWMSysCommand); message
WM_SYSCOMMAND;

  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation

{$R *.DFM}

const
  SC_MyMenuItem = WM_USER + 1;
  SC_MyMenuItem2 = WM_USER + 2;
```

Дописать новые пункты в меню удобнее всего при запуске программы. Сначала проведём горизонтальную разделительную черту (сепаратор), а затем добавим пару команд (подобным образом можно добавить и ещё несколько):

```
//СОЗДАТЬ ФОРМУ
procedure TForm1.FormCreate(Sender: TObject);
begin

  //добавляем новые пункты в системное меню приложения:
  AppendMenu(GetSystemMenu(Handle, FALSE), MF_SEPARATOR, 0, '');
  AppendMenu(GetSystemMenu(Handle, FALSE), MF_STRING,
SC_MyMenuItem, 'Рекламная пауза');
  AppendMenu(GetSystemMenu(Handle, FALSE), MF_STRING,
SC_MyMenuItem2, 'Это конец!');

end;
```

Если значение параметра *uCmdType* в сообщении **WM_SYSCOMMAND** равно одной из наших констант, значит, выбран «наш» пункт системного меню:

```
//ОБРАБОТКА НАЖАТИЯ СИСТЕМНЫХ КНОПОК
procedure TForm1.WMSysCommand(var Msg: TWMSysCommand);
begin

  // перехват события выбора новых пунктов меню:
  if Msg.CmdType = SC_MyMenuItem then OnMyMenu
  else if Msg.CmdType = SC_MyMenuItem2 then OnMyMenu2
  else inherited;

end;
```

Если выбран пункт меню со строкой *Рекламная пауза*, то мы выводим на метке красную «рекламную» надпись и запускаем таймер:

```
//ВЫПОЛНИТЬ КОМАНДУ МЕНЮ 1
procedure TForm1.OnMyMenu;
Begin
  Label1.Font.Color:= clRed;
  Label1.Caption:= 'Тут могла бы быть ваша реклама!';
  Timer1.Enabled:= True;
End;
```

А если «щёлкнута» строчка *Это конец!,* то подтверждаем догадку пользователя:

```
//ВЫПОЛНИТЬ КОМАНДУ МЕНЮ 2
procedure TForm1.OnMyMenu2;
Begin
  Label1.Font.Color:= clRed;
  Label1.Caption:= 'Вы правы!';
  Timer1.Enabled:= True;
End;
```

Через 3 секунды программа попадёт в процедуру **Timer1Timer**, где приложение будет закрыто, если исполняется вторая команда меню, иначе надписи будут последовательно сменять друг друга:

```
//ТАЙМЕР
procedure TForm1.Timer1Timer(Sender: TObject);
begin
  if Label1.Caption= 'Вы правы!' then Close;
  if Label1.Font.Color= clRed then begin
    Label1.Font.Color:= clBlue;
    Label1.Caption:= 'А её тут нет!'; end
  else  begin
    Label1.Font.Color:= clBlack;
    Label1.Caption:= 'Пока!';
```

```
    Timer1.Enabled:= False;
  end;
end;
```

Следуя вашим указаниям на заголовке формы, отчаянный пользователь откроет системное меню, а в нём – при известном любопытстве – выберет один из ваших пунктов (Рис. С7.11). А дальше – будет весело!

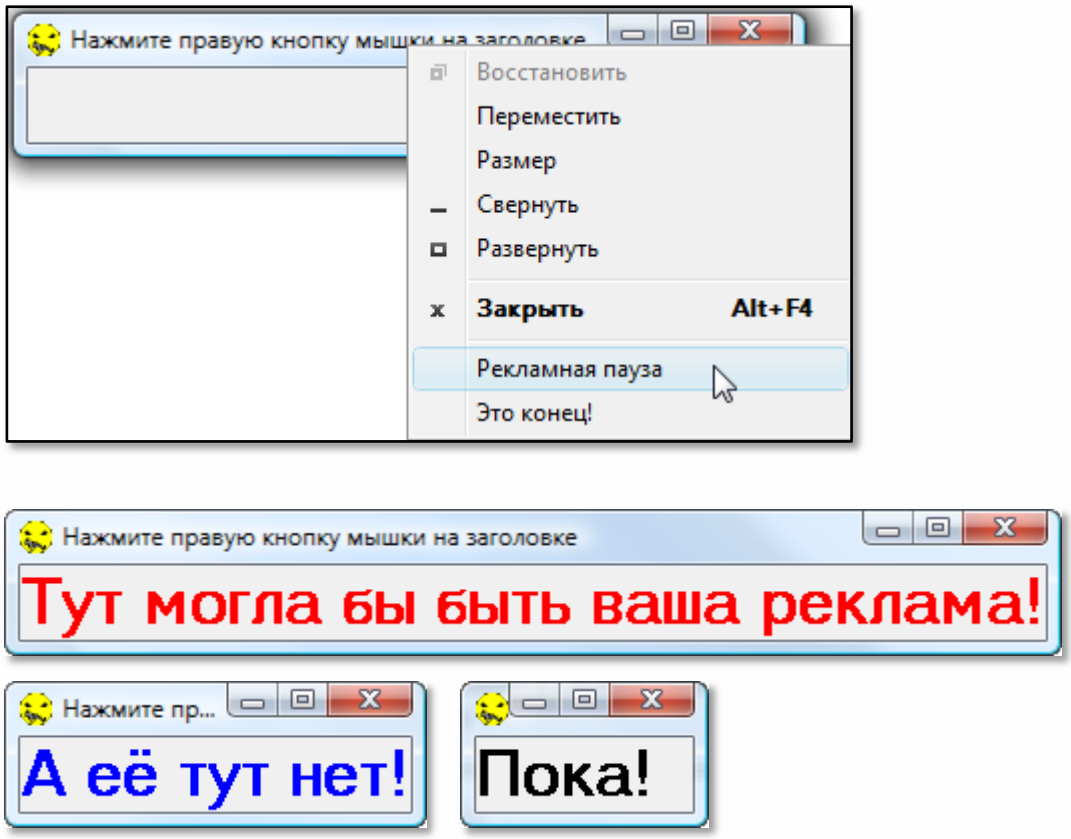


Рис. С7.11. «Системный» разговор!



Исходный код программы находится в папке **SYSMENU**.

"Прикольная" форма

Наверное, вы слышали (и только!) о таком розыгрыше советских времён. К металлическому рублю припаивают гвоздь, а затем всю эту конструкцию вбивают в пол и скрытно ждут наивного искателя сокровищ. Рано или поздно он появляется. Ничего не подозревающий «счастливчик» пытается поднять ценную находку, да не тут то было – рубль невозможно ни сдвинуть, ни сковырнуть. Остаётся только с досадой смотреть на него… (Ещё более мерзкий вариант этого розыгрыша: прибить гвоздями к полу бо-

тинки товарища. Заодно вы сможете проверить, насколько у него развито чувство юмора, - и вообще, товарищ ли он вам.)

А почему бы нам не сыграть эту шутку средствами *Delphi*? Суть подвоха: при запуске программы на экране появляется совершенно обычная форма, но она так привязана к своему месту рождения, что и не сдвинешь (как судно на приколе). Изменить размеры формы тоже не удастся – сколько ни тяни её за углы и стороны! Более того, на каждую попытку покуситься на её суверенитет форма откликается отборными русскими выражениями (их отбор зависит от вас).

Главную роль в нашей программе исполнит процедура **WMPosChange**. Она обрабатывает сообщение *Windows* **WM_WINDOWPOSCHANGING**, которое окно получает при изменении его размеров и положения. В ролях второго плана – процедуры **WMSize** и **WMSizeChange**, также связанные с сообщениями операционной системы.

```
unit Unit1;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
Dialogs, StdCtrls, ExtCtrls;

type
  TForm1 = class(TForm)
    Label1: TLabel;
    Timer1: TTimer;
    procedure Timer1Timer(Sender: TObject);

  private
    { Private declarations }
    procedure WMPosChange (var Message: TWMWINDOWPOSCHANGING);
      message WM_WINDOWPOSCHANGING;
    procedure WMSize (var Message: TWMSIZE);
      message WM_SIZE;
    procedure WMSizeChange (var Message: TWMGETMINMAXINFO);
      message WM_GETMINMAXINFO;

  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation
```

```
{$R *.DFM}
```

Не трогая свойства новой формы, установите на ней метку **Label1**:

```
Font.Color = clRed
Font.Height = -32
Font.Style = [fsBold]
```

и таймер **Timer1** (Рис. С7.12):

```
Enabled = False
Interval = 500
OnTimer = Timer1Timer
```

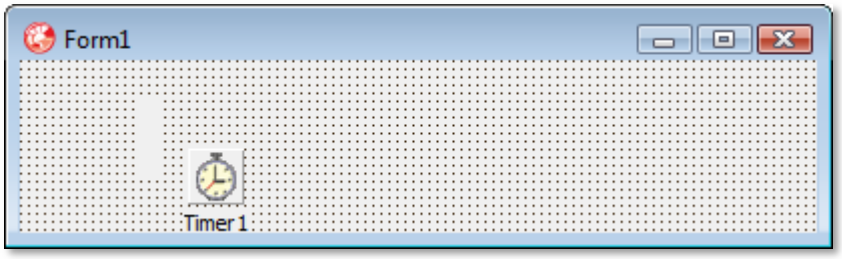


Рис. С7.12. «Прикольный» интерфейс

Чтобы запретить изменение положения и размеров формы, перехватываем сообщение *Windows* **WM_WINDOWPOSCHANGING** и добавляем флаги *SWP_NOMOVE* и *SWP_NOSIZE*:

```
//ЗАПРЕТИТЬ ПЕРЕМЕЩЕНИЕ ФОРМЫ И ИЗМЕНЕНИЕ ЕЁ РАЗМЕРОВ
procedure TForm1.WMPosChange (var Message: TWMWINDOWPOSCHANGING);
begin
  PWindowPos(TMessage(Message).lParam).Flags:=
    PWindowPos(TMessage(Message).lParam).Flags or SWP_NOMOVE or
SWP_NOSIZE;
end;
```

Теперь форма будет стоять на экране, как корабль – на якоре.

Стираем надпись на метке, которая появляется при старте программы:

```
procedure TForm1.WMSize(var Message: TWMSIZE);
begin
  if Message.SizeType <> 0 then
    Label1.Caption:= ''
end;
```

Следующая процедура будет отзываться (или **об**зываться) на попытку сдвинуть форму или изменить её размеры:

```
//ОТВЕТ НА ПОПЫТКУ ИЗМЕНИТЬ СТАТУС ФОРМЫ
procedure TForm1.WMSizeChange (var Message: TWMGETMINMAXINFO);
begin
  if Message.Msg <> 0 then
    Label1.Caption:= 'Что вы меня дёргаете?';
  Timer1.Enabled:= True;
end;
```

Таймер необходим для того, чтобы через некоторое время стирать надпись:

```
//СТЕРЕТЬ НАДПИСЬ
procedure TForm1.Timer1Timer(Sender: TObject);
begin
  Label1.Caption:= '';
  Timer1.Enabled:= False;
end;
```

После старта программы форма будет выглядеть совершенно обычно, но как только пользователь зацепит её мышкой, она отзовётся раздражённым сообщением (Рис. С7.13).

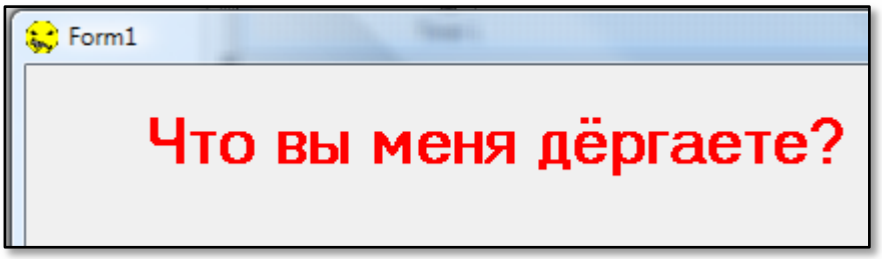


Рис. С7.13. Раздражённая форма



Исходный код программы находится в папке **PRIKOL**.

"Форменная" лихорадка

Можно поступить и иначе – заставить форму беспрерывно «ёрзать» по экрану. Сделать это проще простого. На «свежей» форме установите **таймер**, задайте *Interval* 200-400 мс и напишите код для обработки запуска таймера:

```
unit Unit1;

interface
```

```
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
Dialogs, ExtCtrls;

type
  TForm1 = class(TForm)
    Timer1: TTimer;
    procedure Timer1Timer(Sender: TObject);

  private
    { Private declarations }

  public
    { Public declarations }

  end;

var
  Form1: TForm1;
  rele: boolean= true;

implementation

{$R *.DFM}

//ПЕРЕМЕЩЕНИЕ ФОРМЫ
procedure TForm1.Timer1Timer(Sender: TObject);
begin
  //переключить направление перемещения формы:
  rele:= not rele;

  if rele then begin
     Form1.Left:= Form1.Left+ 5;
     Form1.Top:= Form1.Top+ 5 end
  else begin
     Form1.Left:= Form1.Left- 5;
     Form1.Top:= Form1.Top- 5 end
end;
```

После появления на экране форма будет дрожать как осиновый лист. Для *себя* предусмотрим возможность её успокоения:

```
//ВЫКЛЮЧИТЬ ТАЙМЕР
procedure TForm1.FormDblClick(Sender: TObject);
begin
  Timer1.Enabled:= False
end;

end.
```

При желании можете заставить форму перемещаться по замысловатой траектории - и пользователь неминуемо впадёт в отчаяние.



Исходный код программы находится в папке **DROZH**.

Форма-маяк

Заголовок активной формы имеет яркий цвет, неактивной – блеклый. Если попеременно окрашивать его в эти цвета, то форма будет мигать, как маячок в ночи. Сделать это очень просто, воспользовавшись услугами функции *Windows* **API FlashWindow(HWND: hWnd, bInvert : BOOLEAN)**. Если при её вызове флаг **bInvert** установлен, то форма будет мигать, если сброшен – вернётся в то состояние, в котором она пребывала до начала подмигивания.

Возьмите чистую форму (не правда ли, похоже на кулинарный рецепт), добавьте 1 штуку **таймера** и допишите код:

```
unit Unit1;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
Dialogs, ExtCtrls;

type
  TForm1 = class(TForm)
    Timer1: TTimer;
    procedure Timer1Timer(Sender: TObject);
    procedure FormClick(Sender: TObject);

  private
    { Private declarations }

  public
    { Public declarations }

  end;

var
  Form1: TForm1;

implementation

{$R *.DFM}

//МИГАНИЕ ЗАГОЛОВКА ФОРМЫ
procedure TForm1.Timer1Timer(Sender: TObject);
begin
  FlashWindow(Handle, true)
end;
```

```
//ВЫКЛЮЧИТЬ МИГАНИЕ
procedure TForm1.FormClick(Sender: TObject);
begin
  FlashWindow(Handle, false);
  Timer1.Enabled:= false;
end;

end.
```

После запуска программы заголовок формы будет отчаянно мигать. Но если вы щёлкнете по форме, она прекратит этот нудный процесс.



Исходный код программы находится в папке **FLASH**.

Светоформа

Ещё легче превратить форму в светофор - чтобы она переливалась всеми цветами радуги. Достаточно немного изменить процедуру, обрабатывающую срабатывание таймера:

```
unit Unit1;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
Dialogs, ExtCtrls;

type
  TForm1 = class(TForm)
    Timer1: TTimer;
    procedure Timer1Timer(Sender: TObject);

  private
    { Private declarations }

  public
    { Public declarations }

  end;

var
  Form1: TForm1;

implementation

{$R *.DFM}

//МИГАНИЕ ФОРМЫ
procedure TForm1.Timer1Timer(Sender: TObject);
begin
```

```
  Randomize;
  Form1.Color:= rgb(random(256),random(256),random(256));
  if Form1.visible = true then Form1.Hide
  else Form1.Show
end;
end.
```

После старта программы форма будет то исчезать, то вновь появляться на экране – каждый раз в другой «одёжке» (Рис. С7.14).

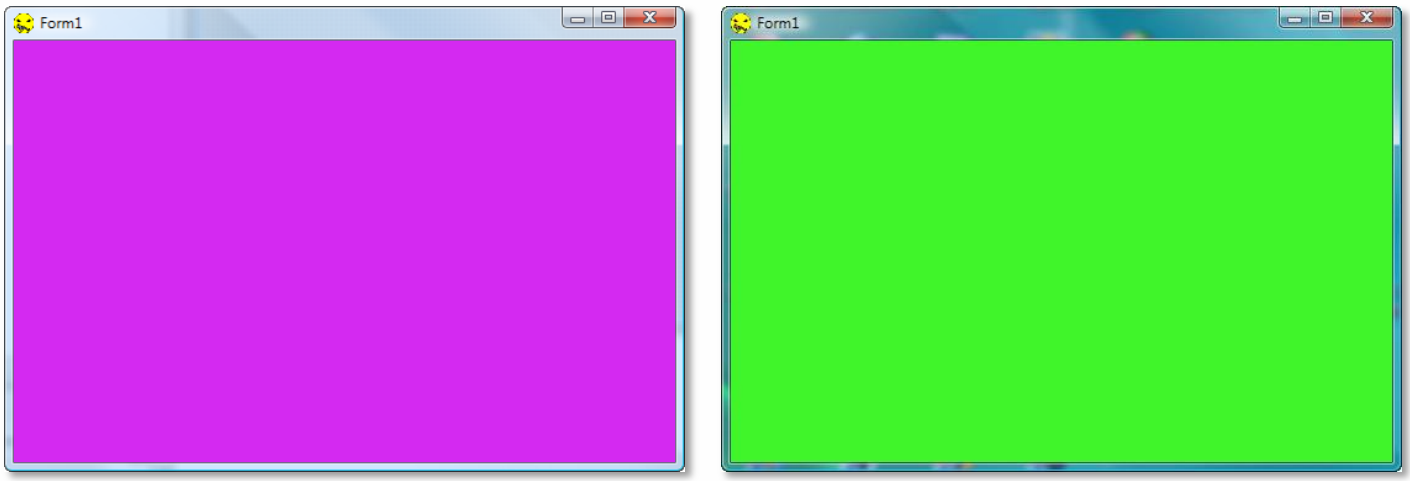


Рис. С7.14. Разноцветная форма



Исходный код программы находится в папке **SVETOFOR**.

На этом потешный час заканчивается. Далее в этой книге вы найдёте фигурные и прозрачные формы, а также немало других программистских артефактов, которые пригодятся вам при организации розыгрышей. Усердно изучая дельфийские премудрости, вы сможете сделать убегающие кнопки, исчезающие меню, возникающие ниоткуда картинки… Да что тут говорить: программируйте – «и хорошее настроение не покинет больше вас»!

Урок 8. Циклюем циклы

Эх, раз, ещё раз,
Ещё много-много раз!

Цыганская песня о циклах

Терпенье и труд всё перетрут.

Пословица

Очень часто в программах встречаются ситуации, когда одни и те же действия нужно повторить многократно. Для таких случаев в языке *Delphi* припасены аж три оператора циклов – **for**, **while** и **repeat**.



Примеры циклов мы можем найти и в повседневной жизни: времена года (циклически сменяют друг друга весна – лето – осень - зима), время суток (утро – день – вечер - ночь), фазы Луны, вращение планет вокруг Солнца и Солнечной системы вокруг центра Галактики, спортивные состязания, учебный год, режим дня, дыхание и кровообращение, часы, биоритмы активности человека – можно даже утверждать, что уникальны именно неповторяющиеся события, а не циклические.

Оператор *FOR*

Цикл **for** встречается в программах чаще других. Его используют, когда число повторений оператора *заранее известно*.

Цикл **for** записывается в одной из форм:

```
for переменная:= начальное значение to     конечное значение do оператор;
for переменная:= начальное значение downto конечное значение do оператор;
```

Оператор (или составной оператор) называется **телом цикла**. Число повторений (они также называются **итерациями**) определяется переменной (управляющей переменной, счётчиком цикла, параметром цикла), которая **автоматически** увеличивается или уменьшается при каждом выполнении тела цикла.

Переменная цикла – это локальная переменная порядкового типа (обычно целого). Она не должна изменяться в теле цикла (да вы и не сможете этого сделать – *Delphi* не позволит!), но её значение может использоваться в выражениях внутри цикла.

Первая форма действует так:

1. Переменной цикла присваивается начальное значение.

2. Текущее значение переменной цикла сравнивается с её конечным значением. Если оно **меньше или равно** конечному значению, то выполняется тело цикла – оператор(ы). Если **больше**, то выполнение цикла заканчивается и управление переходит к следующему за структурой **for** оператору.

3. Если цикл не завершён в п.2, то значение переменной цикла **увеличивается** на 1 и цикл возвращается к п.2.

Понятно, что выполнение цикла **for** рано или поздно закончится. Более того, он может вообще не выполняться ни разу, если начальное значение сразу больше конечного, либо будет выполнен только 1 раз, если начальное значение равно конечному.

Важно помнить, что после завершения цикла значение переменной цикла **не определено**, то есть её нельзя использовать в выражениях!

Вторая форма действует аналогично, за исключением того, что в п.3 значение переменной цикла **уменьшается** на 1. Оператор **for** с **downto** не будет выполнен ни разу, если начальное значение переменной цикла меньше конечного.

Рассмотрим простой цикл, считающий до пяти:

```
procedure _5;
var i: integer;
begin
  for i:= 1 to 5 do      //← Заголовок цикла
    print(inttostr(i)); //← Тело цикла
end;
```

Здесь **i** – переменная цикла (их обычно именуют буквами *i, j, k, l, m, n*).

Начальное значение переменной цикла равно 1, конечное 5. Таким образом, цикл будет выполнен 5 раз, при этом на экране будут напечатаны числа 1 2 3 4 5 (Рис. У8.1).

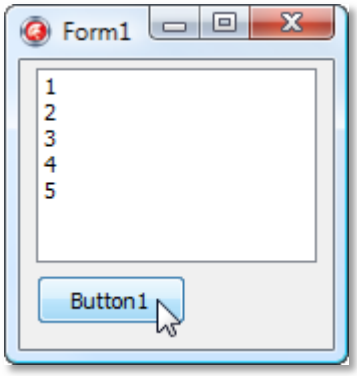


Рис. У8.1. Учимся считать

Оператор *FOR … IN*

В последних версиях *Delphi* появилась новая разновидность оператора цикла *For*. Оператор **For … in** перебирает *все* элементы **коллекции** – это могут быть символы строки, элементы массивов и записей.

```
for переменная in коллекция do оператор;
```



В языке *C#* подобный оператор называется **foreach**.

Переменная цикла должна быть объявлена в том же блоке, что и оператор цикла, а её тип должен быть совместим с типом элементов в коллекции.

Изменять значение переменной цикла внутри цикла нельзя!

Установите на форме текстовое поле **Edit1**, щёлкните по кнопке *Button1* и напишите код в методе **Button1Click**:

```
procedure TForm2.Button1Click(Sender: TObject);
var
  s: string;
  ch: char;
begin
  s:= Edit1.Text;
  for ch in s do
    ListBox1.Items.Add(ch);
end;
```

Здесь мы считываем **строку**, которую ввёл пользователь в текстовое поле, в цикле **for…in** перебираем все символы этой строки и добавляем их в список (Рис. У8.2).

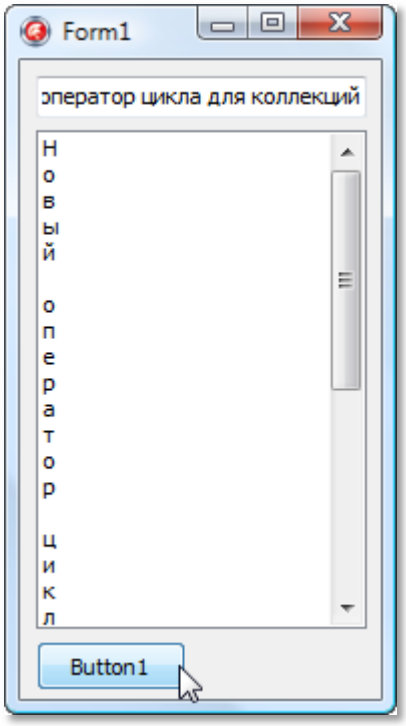


Рис. У8.2. Читаем по буквам

И для этого нам даже не нужно знать длину строки – оператор **for…in** умеет работать самостоятельно!

Очень часто в программах приходится перебирать и элементы **массива**. Дополним нашу программу целочисленным массивом **arr**, в который поместим восемь первых простых чисел:

```
var
  Form2: TForm2;
  arr: array[1..8] of integer   = (2, 3, 5, 7, 11, 13, 17, 19);

implementation

{$R *.dfm}

procedure TForm2.Button1Click(Sender: TObject);
var
  s: string;
  ch: char;
  i: integer;
begin
  s:= Edit1.Text;

  if s <> 'Edit1' then
    for ch in s do
      ListBox1.Items.Add(ch)
  else
    for i in arr do
```

```
      ListBox1.Items.Add(inttostr(i));

end;
```

В метод *Button1Click* добавим проверку строки в текстовом поле. Если пользователь не ввёл строку, то мы печатаем элементы массива (Рис. У8.3), в противном случае - строку.

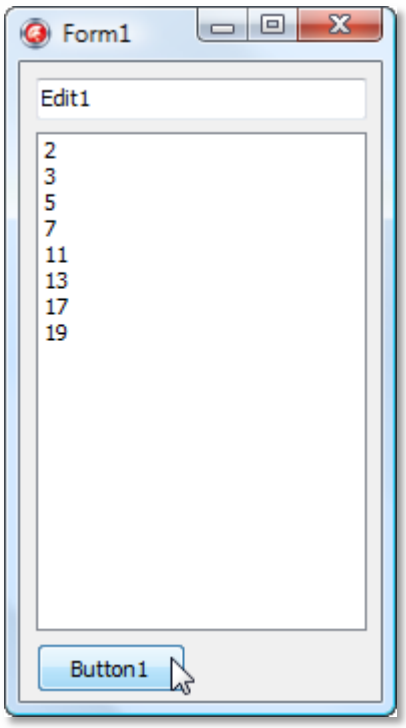


Рис. У8.3. Список простых чисел

Оператор *WHILE*

Любовь – кольцо, а у кольца
Начала нет и нет конца

Песня о зацикливании

Цикл **while** называется также **циклом с предусловием** и имеет следующий вид:

```
while выражение do оператор;
```

Выражение должно быть *логического* типа (*boolean*) и называется также **условием выполнения цикла**.

Работает он так:

1. Вычисляется логическое выражение.

2. Если оно *ложно* (значение равно *FALSE*), то выполнение цикла заканчивается.

3. Если оно *истинно* (значение равно *TRUE*), то выполняется оператор и управление передаётся в п.1.

Поскольку условие проверяется **до** выполнения тела цикла, то операторы могут быть вообще не выполнены ни одного раза – если выражение с самого начала ложно.

Как мы видим, цикл заканчивается, когда условие станет **ложным**. А почему изменяется значение выражения? – Да потому, что изменились значения переменных, входящих в это условие. И это может произойти только в теле цикла. Отсюда грустный вывод: цикл **while** может вообще никогда не закончиться, и программа **зациклится** навечно, что очень часто и бывает!

Пример 1. Найдём все числа Фибоначчи, которые не больше некоторого заданного числа **n**.

Результаты вычислений мы будем выводить в список TListBox **lstProtokol**, задавать нужное нам число от 2 до 99999 в компоненте TSpinEdit **speNum**, а понуждать программу к этому - с помощью кнопки **sbtGo** (Рис. У8.4).

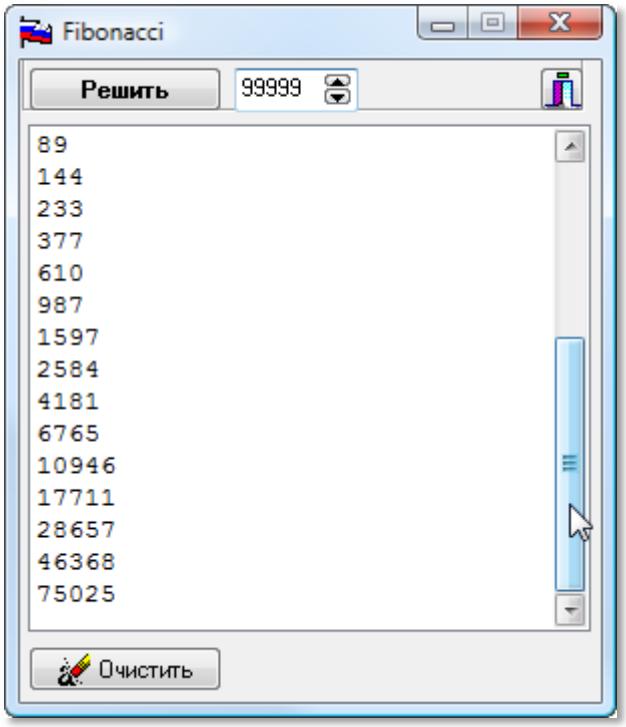


Рис. У8.4. Фибоначчиевый интерфейс

Итак, вводим в однострочный редактор (или нажимаем кнопки со стрелками, если число небольшое) нужное нам число и попадаем в процедуру-обработчик:

```
procedure TfrmMain.sbtGoClick(Sender: TObject);
var n: integer;
begin
  n:= speNum.Value;
  fibo(n);
end;
```

Здесь вызывается процедура, которая и проводит нужные нам расчёты:

```
//РЕШИТЬ ЗАДАЧУ
procedure fibo(n: integer);
var f,f1,f2: integer;
begin
  f:=0;
  f1:=1;
  f2:=1;
  frmMain.lstProtokol.Clear;
  while f2 <= n do begin
    frmMain.lstProtokol.Items.add(inttostr(f2));
    f2:= f1+f;
    f:=f1;
    f1:=f2;
  end;
end;
```



Чтобы понять, как действует в этом примере оператор **while**, уделим немного времени числам Фибоначчи (а эти числа очень интересны!). Нетрудно догадаться, что их открыл Фибоначчи (он же Леонардо Пизанский), средневековый математик, внимательно наблюдавший за размножением кроликов.

Первые два числа равняются 1:

```
f1:=1;
f2:=1;
```

а все последующие равны сумме двух предыдущих, то есть третье число 1 + 1 = 2, четвёртое 1 + 2= 3, пятое 2 + 3 = 5 и так далее, до бесконечности:

```
f2:= f1+f;
f:=f1;
f1:=f2;
```

А цикл **while f2 <= n do** только проверяет, не достигло ли очередное число Фибоначчи заданного нами предела.



Исходный код программы находится в папке **fibonacci**.

Оператор *REPEAT*

Цикл **repeat** называется также **циклом с постусловием** и имеет следующий вид:

```
repeat
  оператор
until выражение;
```

Выражение (условие выполнения цикла) должно быть логического типа.

Оператор **repeat** действует почти так же, как **while**, но с одним важным отличием: условие продолжение цикла проверяется **после** выполнения оператора. Это значит, что цикл обязательно будет исполнен **по крайней мере 1 раз**, даже если логическое выражение истинно с самого начала.

По записи цикла видно, что зарезервированные слова **repeat** и **until** играют роль операторных скобок, поэтому в теле цикла можно просто перечислять операторы - использовать составной оператор нет необходимости (но это и не запрещено). По этой же причине перед **until** точку с запятой разрешается не ставить.

Работает оператор **repeat** так:

1. Выполняется оператор в теле цикла.

2. Вычисляется логическое выражение. Если оно ложно (значение равно *FALSE*), то управление передаётся в п.1. Если оно истинно (значение равно *TRUE*), то выполнение цикла заканчивается.

Теперь мы можем найти и второе различие циклов **while** и **repeat** : первый заканчивается, когда условие не выполняется (значение выражения становится равным *FALSE*), второй – когда условие выполняется (значение выражения становится равным *TRUE*).

Пример 2. Найдём все числа Фибоначчи, которые не больше некоторого заданного числа **n**.

Внешне («интерфейсно») программа ничем не отличается от первого примера, но теперь мы найдём числа Фибоначчи с помощью оператора **repeat**:

```
//РЕШИТЬ ЗАДАЧУ
procedure fibo(n: integer);
var f,f1,f2: integer;
begin
  f:=0;
  f1:=1;
  f2:=1;
  frmMain.lstProtokol.Clear;
  repeat
    frmMain.lstProtokol.Items.add(inttostr(f2));
    f2:= f1+f;
    f:=f1;
    f1:=f2;
  until f2 >= n;
end;
```

Легко заметить, что нам пришлось изменить только небольшую часть кода: заменить один оператор цикла другим и «инвертировать» условие продолжения цикла. Как говорится, лёгким движением руки **while** превращается в элегантный **repeat**!



Исходный код программы находится в папке **fibonacci2**.

Вложенные циклы

Довольно часто внутри одного цикла нужно выполнить другой цикл, например, при поиске нужного элемента в двумерном массиве. Хорошим примером вложенных циклов может быть русская матрёшка: каждая меньшая по размерам кукла целиком посещается внутри большей. При известном мастерстве удаётся создать глубоко вложенные матрёшки:

```
for . . .      //начало первого цикла
begin
  for . . .    //начало второго цикла
  begin
    for . . . //начало третьего цикла
    begin

    end;       //конец третьего цикла
  end;         //конец второго цикла
```

```
end;            //конец первого цикла
```

Как и всегда, отступами выделяйте отдельные циклы! Вложенных циклов может быть любое количество и это могут быть не только циклы **for**, но и **while** и **repeat** в любых сочетаниях.

Пример. В своё время мне пришлось на городской олимпиаде по математике решать такую задачу: Подсчитать, сколько раз пятёрка входит в представление чисел от 1 до 1000 в виде произведения простых чисел: 2, 3, 5, 7, 11,... (например, 24 = 2 * 2 * 2 * 3; 25 = 5 * 5). Мы не на олимпиаде, поэтому пусть задачу решает компьютер, а мы составим для него простенькую программу:

```
//РЕШИТЬ ЗАДАЧУ
procedure TfrmMain.sbtGoClick(Sender: TObject);
var i, n, n5: integer;
begin
  n5:=0;
  for i:= 1 to 1000 do begin //проверяем все заданные числа
    n:= i;
    while TRUE do begin
      if n mod 5 = 0 then begin
        n:= n div 5;
        inc(n5);
      end
      else break;
    end;
  end;
  frmMain.lstProtokol.Items.add(inttostr(n5));
end;
```

Здесь мы легко найдём цикл **for**, в котором перебираются все заданные числа, и вложенный в него бесконечный цикл **while**, который и подсчитывает их делители. Если число делится **нацело** на 5, то увеличиваем счётчик на 1, иначе переходим к проверке следующего числа. Олимпиадный «подвох» этой задачи заключается в том, что полученное после деления на 5 число опять может быть кратно 5, то есть его опять нужно проверить. Если это обстоятельство не учесть, то число пятёрок можно было бы подсчитать мгновенно: 1000 : 5 = 200. Таким образом, в первой тысяче чисел ровно 200 делятся на 5. Ещё 40 делятся на 25 (5*5), 8 – на 125 (5*5*5) и одно – на 625 (5*5*5*5). Зная ответ, и задачу легко решить, а я на олимпиаде намаялся (Рис. У8.5)!

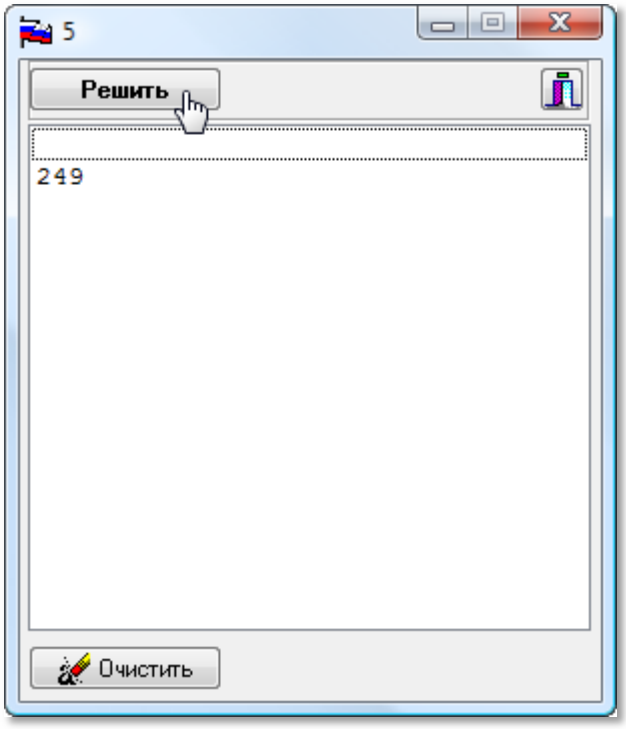


Рис. У8.5. Одно нажатие на кнопку – и олимпиадная задача решена



Исходный код программы находится в папке **5**.

Досрочный выход из цикла

Можно выйти?

Сакраментальный школьный вопрос

Иногда нужно прервать выполнение цикла до его окончания.

Есть несколько способов досрочного завершения циклов. Например, для этого в циклы **while** и **repeat** можно добавить *флаг* (переменную логического типа), который будет сигнализировать об окончании работы цикла. Рассмотрим практически бесполезную процедуру:

```
procedure _cycl;
var
  n,i: integer;
  flag: boolean;
begin
  flag:= FALSE;
  i:=0;
```

```
  while not flag do begin
    i:= i+1;
    //if i > 5 then flag:= TRUE;
    flag:= i > 5;
  end;
  frmMain.lstProtokol.Items.add(inttostr(i));
end;
```

Перед началом цикла мы сбрасываем флаг, и теперь оператор **while** будет выполняться до тех пор, пока значение флага не станет равным *TRUE*. Мы сами должны «поднять флаг» в нужный момент. Например, мы отсчитали больше 5 проходов цикла (переменная **i** приняла значение 6). Нам этого вполне достаточно, что мы и сообщаем оператору **while**, который тут же заканчивается и передаёт управление оператору, следующему за структурой **while**, и мы в окне списка увидим число 6.

Как видите, всё очень просто. Главное – не забыть на каждой итерации увеличивать значение переменной **i**, в противном случае цикл не закончится никогда! Точно так же флаг используется и в цикле **repeat**:

```
  repeat
    i:= i+1;
    flag:= i > 5;
  until flag;
  frmMain.lstProtokol.Items.add(inttostr(i));
```

Процедура *break* и оператор *goto*

Но в *Delphi* есть и более элегантные средства для прерывания любых циклов - достаточно в нужном месте тела цикла записать слова **break** или **goto**.

Слегка изменим последний пример:

```
  repeat
    i:= i+1;
    if i > 5 then break;
  until FALSE;
  frmMain.lstProtokol.Items.add(inttostr(i));
```

Мы избавились от флага, и теперь при наступлении нужного нам события (**i > 5**) выполняется процедура **break**, и цикл заканчивается.

Действие процедуры **break** в цикле **while** рассмотрите самостоятельно, а цикл **for** прерывается ещё проще:

```
  for i:= 1 to 20 do begin
    if i > 5 then break;
  end;
  frmMain.lstProtokol.Items.add(inttostr(i));
```



Пример из школьной жизни. Нам нужно определить, под каким номером записан Ваня Ургант в классном журнале. Как учит нас Президент Медведев, вся документация должна вестись в электронном виде. Значит, классный журнал можно представить себе как массив, в котором записаны все ученики (числом 30 голов). Всё – начинаем перебирать массив:

```
  for i:= 1 to 30 do begin
    if name[i] = 'Ваня Ургант' then begin
      n:= i;
      break;
    end;
  end;
  frmMain.lstProtokol.Items.add(inttostr(n));
```

Как только мы найдём в журнале нашего Ваню, дальше искать смысла нет – Ваня Ургант единственный и неповторимый! В переменной **n** после окончания цикла мы и найдём номер Вани в классном журнале. И хотя фамилия Вани начинается на букву «у», скорее всего это будет номер 1, ведь Ваня даже на Первом первый!

Можно привести и другие примеры – уже из спортивной жизни. Один удачный удар – нокаут, и бой между Николаем Валуевым и Ваней Ургантом преждевременно заканчивается. Такая же досада случилась и у хоккеистов сборной России по хоккею в матче с финнами, когда эти горячие парни забили нам шайбу в дополнительном периоде. Есть и более огорчительные для мужского достоинства примеры, но упоминать их было бы неспортивно!

Вместо **break** вы можете использовать оператор **goto**, который передаёт управление какому-нибудь оператору за пределами цикла. Этот оператор должен быть обозначен **меткой** (иначе получится, как в сказке, *пойди туда, не знаю куда!*), которая должна быть объявлена в процедуре:

```
procedure _cycl;
label vyhod;
var
  i: integer;
  flag: boolean;

  while TRUE do begin
```

```
    i:= i+1;
    if i > 5 then goto vyhod;
  end;
vyhod: frmMain.lstProtokol.Items.add(inttostr(i));
```

В данных примерах действие процедуры **break** и оператора **goto** одинаково – они просто передают управление следующему за циклом оператору, который печатает число выполненных итераций. Однако легко видеть, что оператор **goto** гораздо более дальнобойный – он может передать управление любому оператору процедуры, которому предшествует метка (меток в процедуре может быть сколько угодно). Обычно применение операторов **goto** в программе не приветствуется, поскольку они нарушают логику работы программы, перескакивая в произвольное место кода. Из этого следует, что ими не следует злоупотреблять, но в некоторых случаях они могут и облегчить понимание кода, если проверки выхода из цикла очень сложны. Обычно это бывает при выходе из вложенных циклов.

И последнее: при выходе из цикла с помощью **goto** с программой ничего не случится, но никогда не запрыгивайте извне внутрь цикла!

Процедура *exit*

Процедура **exit** действует более радикально – она досрочно завершает не только цикл, но и всю подпрограмму, в которой этот цикл находится.

Например, если при выполнении цикла **while** сработала процедура **exit**,

```
. . .
  while TRUE do begin
    i:= i+1;
    if i > 5 then exit;
  end;
frmMain.lstProtokol.Items.add(inttostr(i));
end;
```

то число 6 в окне списка напечатано не будет, поскольку мы сразу же «улетим» в запроцедурное пространство. Такие «полные улёты» бывают полезны при выходе из вложенных циклов.

В последних версиях *Delphi* процедура **exit** может возвращать значение того же типа, что и функция пользователя.



Так действует оператор *return* в языках C# и C++.

Если функция *exit* не возвращает никакого значения, то ни круглые скобки, ни аргумент указывать не нужно.

Например, «по старинке» мы можем написать вот такую функцию *GetRandom*, которая возвращает случайное целое число в диапазоне *iLo..iHi*:

```
Function GetRandom(iLo, iHi: Integer): Integer;
begin
  GetRandom := iLo + (Random (iHi - iLo + 1))
end;
```

Или

```
Function GetRandom(iLo, iHi: Integer): Integer;
begin
  Result := iLo + (Random (iHi - iLo + 1))
end;
```

Эту же функцию мы можем обновить, возвращая случайное значение как аргумент функции **exit**:

```
Function GetRandom(iLo, iHi: Integer): Integer;
begin
  exit(iLo + (Random (iHi - iLo + 1)));
end;
```

Естественно, в этой функции нет особого резона возвращать значение с помощью процедуры *exit*. Мы это сделали только для примера. Правильнее использовать процедуру *exit* именно для *досрочного* завершения функции.

Процедура *continue*

Процедура **continue** прерывает только текущую итерацию, и тело цикла выполняется с начала.

Продолжим терзать Ваню. Пусть вместе с ним в классе учатся Цекало, Светлаков, Мартиросян и другие лоботрясы, которые портят успеваемость. Подсчитаем число прилежных учеников в классе:

```
n:=0;
for i:= 1 to 30 do begin
  if uspev[i] = 2 then continue;
  n:= n+1;
end;
frmMain.lstProtokol.Items.add(inttostr(n));
```

Как только мы наткнёмся на двоечника, сразу же переходим к следующему ученику, а троечников, хорошистов и отличников валим в одну кучу. Теперь классный руководитель точно знает, сколько билетов ему заказывать для поездки в Италию. Плюс ещё один билет – для Вани.

Более серьёзные и полезные примеры с процедурой **continue** вы найдёте дальше в этой книге.

Выводы

Используйте оператор **for**, когда число итераций известно заранее, а операторы **while** и **repeat** – если не известно.

Если вы уверены, что цикл может быть выполнен хотя бы 1 раз, используйте **repeat**, в противном случае **while**.

Всегда следите, чтобы условие продолжения цикла неизбежно принимало такое значение, которое завершает цикл, иначе программа зациклится.

Для досрочного выхода из цикла пользуйтесь процедурами **break** и **exit**, а также оператором **goto**.

Семинар 8. Циклотрон, или Истории Древнего мира

Говоря о циклах, можно и зациклиться – так много есть красивых примеров, иллюстрирующих их применение в программировании, поэтому мы напишем всего 2 программы, но если вы будете наблюдательны, то найдёте циклы ещё на многих страницах этой книги.

Римские числа

Все дороги ведут в Рим.

Из ответа на уроке географии

Древние римляне создали огромную, могущественную империю, которую назвали в свою честь, но что касается чисел, то тут они изрядно начудили и придумали столь вычурную систему обозначения чисел буквами, что даже простейшие арифметические вычисления давались им с большим трудом. Существенно облегчили школярам тяготы учёбы пришедшие на смену римским арабские цифры, которые почему-то придумали индийцы.

Римская империя давно пала под натиском варваров, а их вычурные цифры сохранились до сих пор в первозданном виде. К счастью, нам не нужно пользоваться ими на уроках математики, но как украшательство мы можем найти их в книгах для обозначения глав, для подсчёта столетий, царей, съездов и других исторических событий.

Поскольку компьютер не понимает не только римских цифр, но даже и арабских, то напишем простую программу, которая поможет нам разобраться в премудростях римской нумерации.

Установим на форме заветную кнопку **sbtGo** с призывной надписью *Найти*, да и нажмём её (Рис. С8.1):

```
procedure TfrmMain.sbtGoClick(Sender: TObject);
var n: integer;
begin
  n:= speNum.Value;
  rome(n);
end;
```

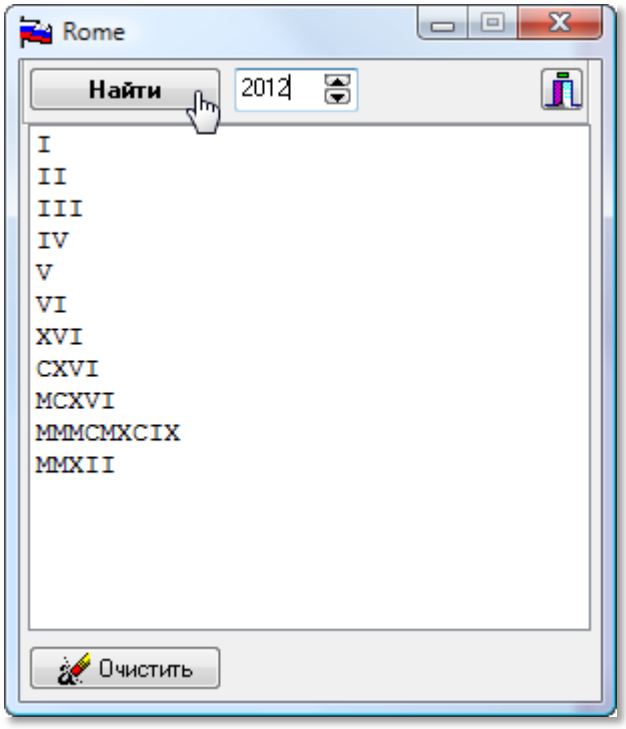


Рис. С8.1. Считаем по-римски!

Но мы не в сказке, и, конечно, ничего не произойдёт, поэтому введём новый тип-запись **TRome**, который будет хранить и само число, и его «римское» представление. Если бы не краткость нашей жизни, мы могли бы просто записать все числа от 1 до 3999 в арабском и римском виде, но поскольку это не так, то схитрим и в массиве **num** сохраним только 13 самых необходимых сочетаний чисел, а остальные придётся вычислять.

```
type
  TRome=record
     an: 1..1000;
     rn: string;
   end;

var
  frmMain: TfrmMain;
  num: array[1..13] of TRome;
```

При запуске программы заполняем массив **num**:

```
procedure TfrmMain.FormCreate(Sender: TObject);
begin
  num[1].an:=1000; num[1].rn:='M';
  num[2].an:=900; num[2].rn:='CM';
  num[3].an:=500; num[3].rn:='D';
  num[4].an:=400; num[4].rn:='CD';
  num[5].an:=100; num[5].rn:='C';
  num[6].an:=90; num[6].rn:='XC';
  num[7].an:=50; num[7].rn:='L';
```

```
  num[8].an:=40; num[8].rn:='XL';
  num[9].an:=10; num[9].rn:='X';
  num[10].an:=9; num[10].rn:='IX';
  num[11].an:=5; num[11].rn:='V';
  num[12].an:=4; num[12].rn:='IV';
  num[13].an:=1; num[13].rn:='I';
end;
```

Легко понять, почему нам понадобились именно эти числа – остальные довольно просто собрать из них, как ожерелье из бусин.

И вот как это делается:

```
//НАЙТИ РИМСКОЕ ЧИСЛО
procedure rome(number: integer);
var n: integer;
  s: string;
begin
  if (number < 1) or (number> 3999) then exit;
  //frmMain.lstProtokol.Clear;

  n:=0;
  s:='';
  while number>0 do begin
    inc(n);
    while num[n].an <= number do begin
      s:= s + num[n].rn;
      number:= number - num[n].an;
    end;
  end;
  frmMain.lstProtokol.Items.add(s);
end;
```

Как мы и договаривались, разумно ограничим себя числом 3999 сверху и 1 снизу (а меньше у римлян и не было), поэтому «сверхъестественные» числа просто игнорируем.

Для подсчёта числа и формирования строки нам потребуются: переменная **n** целого типа и строка **s**. Вся премудрость конвертирования чисел таится в двух циклах **while**. Мы последовательно сравниваем заданное число **number** с числами, которые хранятся в массиве **num**, начиная с тысяч. Внутренний цикл как раз и нужен для того, чтобы определять, сколько в **number** имеется тысяч, сотен, десятков и единиц (все остальные римские числа - 900, 500, 400, 90, 50, 9, 5, 4 - могут быть только в единственном числе). Если текущее значение **number** не меньше этих чисел, то из него число вычитаем, а в строку добавляем буквы, соответствующие этому числу в римской записи. Как только от числа ничего не осталось (это контролирует внешний цикл **while**), преобразование числа заканчивается, и мы

смело можем печатать на экране перевод римского числа на современный европейский математический язык.



Исходный код программы находится в папке **rome**.

Египетские пирамиды

Компьютер – страшная сила!
Программистская банальность

Всё врут календари.
А. С. Грибоедов. *Горе от ума*

Пирамиды - большая загадка для всего человечества, нам, конечно, её не решить даже с помощью компьютера, но в мире есть и другие причудливые пирамиды, - например, числовые!

В довольно увесистой книге *Gehirn Jogging* под номером 147 можно найти вот такую задачу (Рис. С8.2).

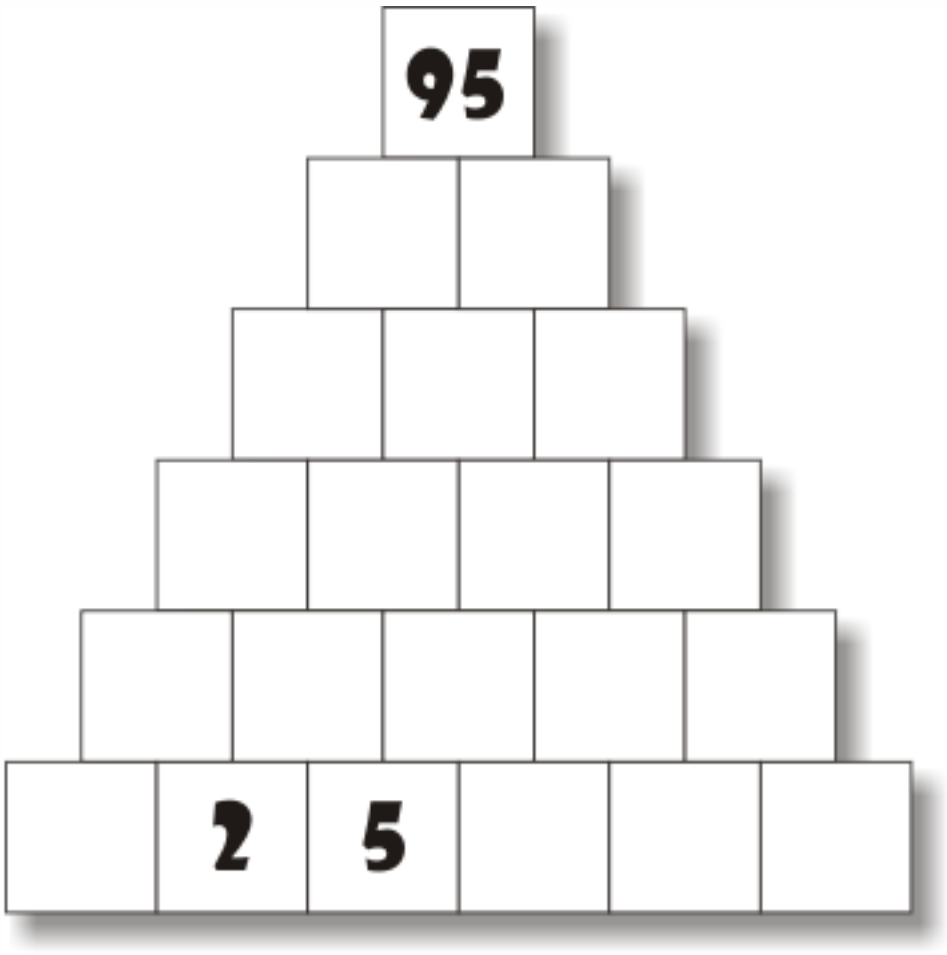


Рис. С8.2. Пирамидальная задача

Условие таково: в основании пирамиды находятся все числа от 1 до 6, найти числа в пустых клетках, если каждое число, опирающееся на два нижних, равно их сумме.

Ясно, что в клетке над двойкой и пятёркой стоит число 2 + 5 = 7 (Рис. С8.3).

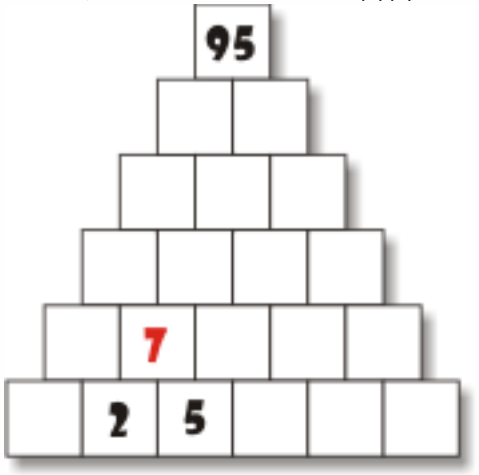


Рис. С8.3. Одно число отгадали!

Что делать? – Либо думать, либо писать программу. Лень – двигатель прогресса, поэтому мы напишем программу, которая решит не только эту задачу, но и все подобные.

Интерфейс собираем, не мудрствуя лукаво, на один раз любой сгодится (Рис. С8.4).

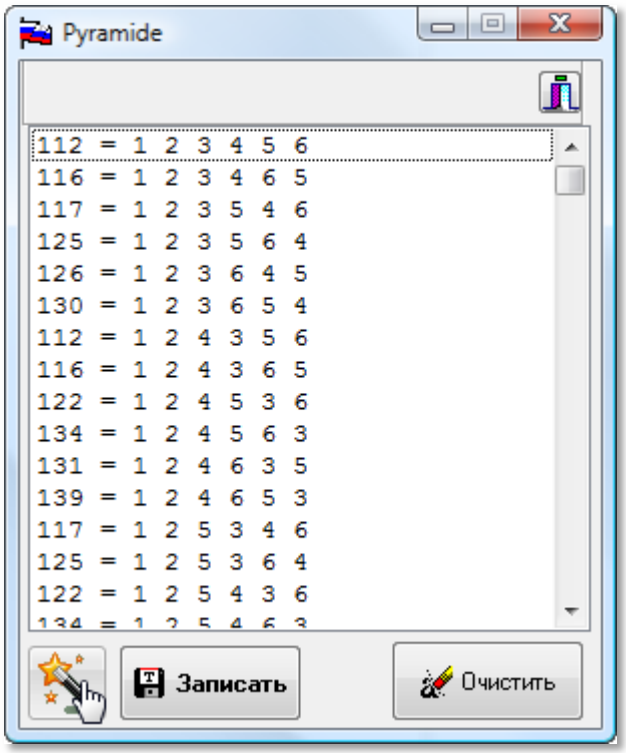


Рис. С8.4. Пирамидальный интерфейс

Результаты наших поисков выводим в список **lstProtokol**. Магическая кнопка **sbtSolve** находит все перестановки чисел 1, 2, 3, 4, 5, 6 в основании пирамиды, а также сумму, которая получается при этом в её вершине. Затем список сохраняем на диске, ублажив курсором кнопку **sbtSaveProtokol**.

Перестановки чисел будем искать грубой силой, без поиска изящных алгоритмов:

```
procedure TfrmMain.sbtSolveClick(Sender: TObject);
var
  a1,a2,a3,a4,a5,a6,f: integer;
  s: string;
begin
  lstProtokol.Items.clear;
  for a1:= 1 to 6 do begin
    for a2:= 1 to 6 do begin
      if a2=a1 then continue;
      for a3:= 1 to 6 do begin
        if (a3=a2) or (a3=a1) then continue;
        for a4:= 1 to 6 do begin
          if  (a4=a3) or (a4=a2) or (a4=a1)then continue;
          for a5:= 1 to 6 do begin
          if (a5=a4) or  (a5=a3) or (a5=a2) or (a5=a1)then conti-
nue;
            for a6:= 1 to 6 do begin
             if  (a6=a5)  or   (a6=a4)  or  (a6=a3)  or  (a6=a2)  or
(a6=a1)then continue;
              f:= a1 + a6 + 5*(a2+a5) + 10*(a3+a4);
              s:='';
              if f < 100  then s:= s + '0';
              s:=  s+  inttostr(f)+'  =  '+  inttostr(a1)+'  '+  int-
tostr(a2)+' '+ inttostr(a3)+' '+ inttostr(a4)+' '+ inttostr(a5)+'
'+ inttostr(a6);
              lstProtokol.Items.add(s);
            end;
          end;
        end;
      end;
    end;
  end;
end;
```

На деле это выглядит так. В основании стоят 6 чисел от единицы до шестёрки, причём числа не повторяются. Значит, выписываем **6 вложенных циклов for** и в каждом проверяем это условие. Если параметр цикла равен уже вышедшему ранее числу, то проверяем следующее (вот и процедура **continue** пригодилась!). Как только мы нашли очередную расстановку

(шесть разных чисел в основании), вычисляем по правилам задачи сумму в вершине пирамиды:

```
f:= a1 + a6 + 5*(a2+a5) + 10*(a3+a4);
```

Хотите верьте, хотите нет, но она именно такая (посчитайте сами!) (Рис. С8.5).

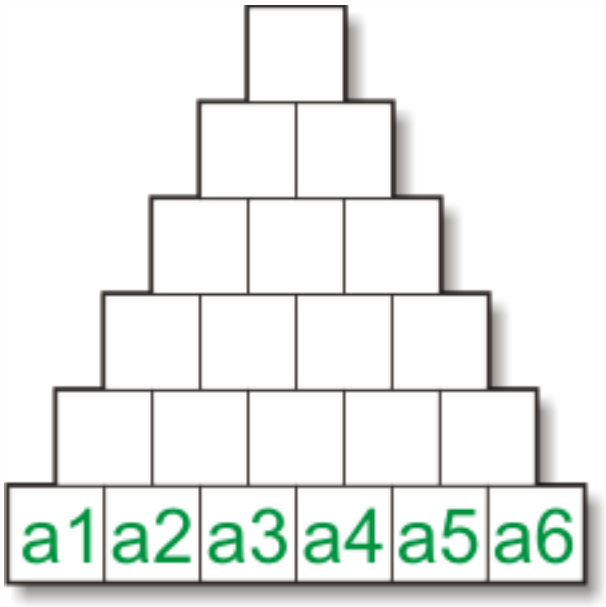


Рис. С8.5. Выводим формулу!

Осталось красиво записать найденный вариант расстановки чисел и сумму в окно протокола.

К сожалению, в этом списке очень неудобно отыскивать нужную нам сумму в вершине. Но мы не будем утруждать себя сортировкой вариантов по возрастанию суммы, а просто запишем список в текстовый файл **protokol.TXT**:

```
//ЗАПИСАТЬ ПРОТОКОЛ НА ДИСК
procedure TfrmMain.sbtSaveProtokolClick(Sender: TObject);
var
  s,fn: string;
  i: integer;
  f: textfile;
begin
  //savedialog1.DefaultExt:='txt';
  savedialog1.Filter:='Solutions (*.txt)|*.TXT';
  savedialog1.FilterIndex:=1;
  s:=extractfilepath(application.exename)+'solutions\';
  savedialog1.InitialDir:= s;
  savedialog1.Title:='Save the solution to the disk';
```

```
  //имя файла:
  savedialog1.filename:='protokol_temp.txt';

  if not savedialog1.Execute then exit;

  //имя конечного файла:
  fn:= savedialog1.filename;
  assignfile(f,fn);

  if FileExists(savedialog1.filename) then
    append(f)        //- добавить информацию в конец файла
  else rewrite(f);  //- перезаписать файл

  //записать протокол:
  for i:=0 to (lstProtokol.Items.Count - 1) do
    writeln (f,lstProtokol.Items.Strings[i]);
  Closefile(f);

  Messagebeep(0)
end;
```

А затем откроем его в текстовом редакторе *Microsoft Word*, отсортируем его и получим список **protokol.doc**, в котором легко найти нужное нам число в вершине пирамиды:

```
095 = 4 2 1 5 3 6
095 = 4 2 5 1 3 6
095 = 4 3 1 5 2 6
095 = 4 3 5 1 2 6
095 = 6 2 1 5 3 4
095 = 6 2 5 1 3 4
095 = 6 3 1 5 2 4
095 = 6 3 5 1 2 4
```

Всего можно насчитать 8 расстановок шести чисел, которые дают сумму **95**, но нас интересуют только те, в которых двойка и пятёрка стоят на втором и третьем местах. И что мы видим: годятся две расстановки:

```
095 = 4 2 5 1 3 6
095 = 6 2 5 1 3 4,
```

значит, и решений у задачи тоже два. Смотрим в ответ – одно (первое, Рис. С8.6). Не верьте **толстым** книгам!

В книге ещё много таких задач, но мы их решили все разом и даже подкузьмили незадачливого немецкого автора, который не удосужился поискать и второе решение (во всех «пирамидальных» задачах книги).

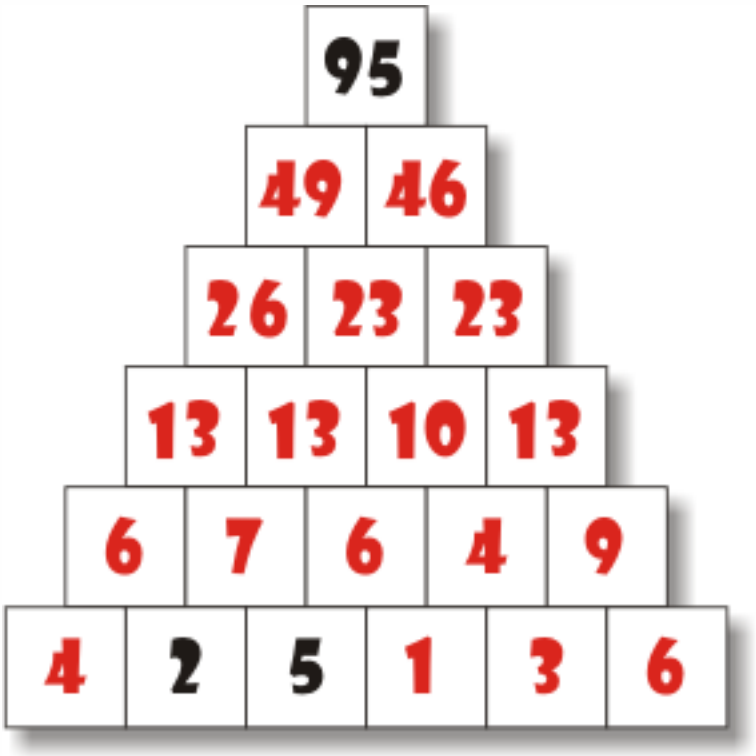


Рис. С8.6. Ответ из книги, «наш» вариант добавьте сами!

И не надейтесь, что я написал всю программу с нуля! Компонент со списком, кнопки, процедуру записи списка в файл я благополучно экспортировал (легально!) из других своих программ. Добавить пришлось только незамысловатую процедуру составления списка вариантов – по времени дольше объяснять, чем «соорудить» такую программу.



Исходный код программы находится в папке **pyramide**.

Урок 9. Массивы, или Как разложить всё по полочкам

Раз ступенька, два ступенька – будет лесенка.
Раз словечко, два словечко - будет песенка.

Как раз песенка

…В чешуе, как жар горя,
Тридцать три богатыря,
Все красавцы удалые,
Великаны молодые.
Все равны, как на подбор,
С ними дядька Черномор.

Пушкин. *Сказка о царе Салтане*

Как Однажды Жак-Звонарь
Городской Сломал Фонарь.

Мнемонический стишок про цвета радуги

Массив – это набор объектов (элементов) *одного и того же типа*, каждый из которых имеет свой номер в этом наборе (*индекс*).



Массивы относятся к структурным типам данных - как множества, записи, классы, файлы и строки.

Объявление переменной-массива имеет вид:

```
var a: array[начальный индекс..конечный индекс] of тип;
```

или более определённо:

```
var a: array[1..10] of integer;
```

Как видите, чтобы объявить массив, используется зарезервированное слово **array**, после которого в квадратных скобках указывается диапазон индексов, затем слово **of** и тип элементов массива. Объявление заканчивается точкой с запятой.

Индекс определяет относительную позицию элемента в массиве.

Начальный и конечный индексы задаются при объявлении массива и **потом не изменяются**. Если в языке *C* начальный индекс всех массивов равен нулю (что часто приводит к ошибкам), то *Delhi* более либерален: вы

сами можете выбирать диапазон индексов, как вам угодно (но конечный индекс должен быть не меньше конечного!).

Например, переменную-массив a можно объявить и так:

```
var a: array[0..9] of integer;
var a: array[-1..8] of integer;
var a: array[2001..2010] of integer;
```

Сам массив от этого не изменится - каждый из этих массивов содержит по 10 целых чисел, - но вы должны помнить, что самый первый элемент массива имеет начальный индекс, а последний – конечный.

Тип массива (а, значит, и его элементов) может быть любым: *char, real, integer, string, array, …*

К каждому элементу массива можно **обратиться** по его индексу. Для этого нужно указать имя массива и индекс нужного элемента в квадратных скобках:

`a[1]` – это переменная типа **integer** (в общем случае – того типа, который указан при объявлении массива); она имеет в массиве индекс 1 (в нашем примере это первый элемент массива, в общем случае – в зависимости от заданного диапазона).

Индекс можно задавать и *выражением*, которое имеет тот же тип, что и индекс (обычно диапазон):

`a[2 + 2*3]` – элемент массива с индексом 8.

С элементом массива можно обращаться точно так же, как и с переменной того же типа:

```
a[1]:= 1;
a[2]:= a[1] + 1;
a[3]:= a[1] * a[2];
```

Кроме непосредственного объявления переменной-массива, можно сначала **объявить тип** со структурой массива, а уже потом - одну или больше переменных этого типа:

```
type
  mas= array[1..10] of integer;
var
  a1,a2,a3: mas;
```

Объявление переменной типа массива можно совмещать с **присвоением ей значений**. Они перечисляются через запятую в круглых скобках:

```
var a: array[1..10] of integer = (1,1,2,3,5,8,13,21,34,55);
```

Вот так просто мы заполнили массив числами Фибоначчи.

В *памяти* компьютера одномерный массив занимает последовательные ячейки (Рис. У9.1).

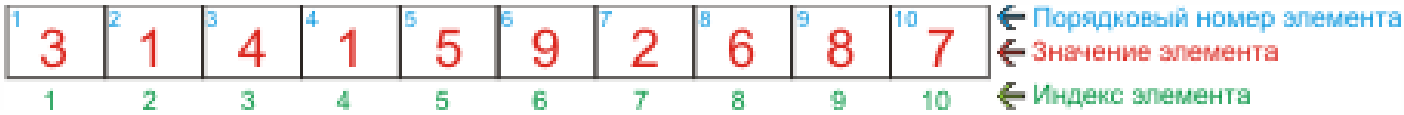


Рис. У9.1. Массив в памяти компьютера

Каждый элемент массива занимает столько ячеек памяти, сколько требуется для хранения значения такого типа. Например, для целого числа типа *Integer* отводится 4 байта, а для символа достаточно и одного.



Массивы – повсюду: патроны в магазине, семечки в подсолнухе, кильки в томате, ученики в классе, слова в словаре, пальцы на руке, цвета в радуге, солдаты в шеренге – как только мы сумеем выделить несколько одинаковых объектов, которые образуют группу, мы тут же можем объявить массив:

```
Type
  отряд= array[1..33] of богатырь;
  колода= array[1..52] of карта;
  словарь= array[1..30000] of string;
  класс= array[1..30] of ученик;
…
```

Хорошим примером одномерного массива могут послужить квартиры в доме, бирки в раздевалке, бочонки русского лото или шары в спортлото – они уже пронумерованы, то есть даже номера элементов (индексов) проставлены!

Теперь нам не нужно помнить имя каждого ученика – мы можем обратить к нему «по индексу», или номеру под которым он записан в журнале (квартирам или биркам вообще никто собственного имени не даёт). Вместо Шурика можно вызвать ученика `класс[13]`, если Шурику опять не повезло и он 13-ый в классном журнале.

Одномерные массивы символов

Одномерные массивы символов имеют некоторые особенности по сравнению с другими типами массивов. Им можно приваивать начальные значения как обычным массивам – каждому элементу в отдельности, так и всем сразу, как это делается для строк:

```
var ch: array[1..6] of char = ('s', 't', 'r', 'i', 'n', 'g');
var ch: array[1..6] of char = ('string');
```

Затем можно присвоить новое значение и каждому элементу массива, и всему массиву разом:

```
ch[1]:= 's'; …
ch:= 'string';
```

Массивы символов можно *сравнивать* между собой, даже если они не идентичных типов и разной длины, и объединять в один массив операцией конкатенации.

Таким образом, с массивом символом можно обращаться и как со строкой фиксированной максимальной длины (в данном случае в переменную нельзя «вогнать» больше 6 символов), и как с обычным массивом.

Типы индексов

В качестве индекса может быть использован не только ограниченный целый тип, как во всех наших примерах, но и любые другие ограниченные типы (поэтому нельзя использовать типы **integer** и **real** – первый слишком большой, второй – вообще бесконечный).

1. **Логический тип:**

   ```
   var al: array[boolean] of integer;
   ```

   Поскольку логический тип имеет всего 2 элемента, то и в массиве их будет тоже 2. Обратиться к ним можно так:

   ```
   al[false]:= 0;
   al[true]:= 1;
   ```

2. **Диапазон:**

   ```
   var ad: array[1..10] of boolean;
   ```

Этот тип индексов встречается чаще всего. Обычно для конечного индекса используют константу

```
const MAX_INDEX= 10;
var ad: array[1.. MAX_INDEX] of boolean;
```

Эту же константу следует записывать и в исходном коде программы при обозначении максимального индекса массива. Такой способ задания диапазона индексов позволяет легко модифицировать программу (достаточно изменить в одном месте значение константы), да и идентификатор *MAX_INDEX* гораздо понятнее, чем просто числовая константа 10.

А такой массив, индексами которого являются все заглавные буквы латинского алфавита

```
var ad: array['A'..'Z'] of Integer;
```

поможет вам подсчитать, сколько раз они встречаются в каком-либо тексте:

```
ad['A']:=0;
```

Затем увеличиваем счётчик каждый раз, когда найдём букву А:

```
inc(ad['A']);
```

Для других букв действуйте аналогично.

3. **Перечисляемый тип** можно употребить и как тип элементов, и как тип индексов:

```
type color= (red, orange, yellow, green, cyan, blue, violet,
black);
  cells= array[1..8,1..8] of color;

var field: cells;
```

Теперь вы можете раскрашивать любое игровое поле (хотя бы шахматную доску) во все цвета радуги:

```
field[1,1]:= red;
field[1,2]:= orange;
```
и так далее.

```
type  cells2= array[color] of boolean;
var clr: cells2;

clr[black]:= false; //- такого цвета в радуге нет
```

```
clr[red]:= true; //- такого цвет в радуге есть
```

Типы индексов можно смешивать:

```
Var field2: array[1..8, color] of boolean;

Field2[2, violet]:= true;
```

4. **Литерный тип** состоит из всех символов *ASCII*:

```
Var Lit: array[char] of byte;
```

Объявив таким образом переменную, мы можем задать коды символов:

```
Lit['0']:= 48;
Lit['1']:= 49;
```

Если вы вдруг переврёте коды символов, то сможете шифровать свои сообщения и смело делиться своими сообщениями с товарищами – а пусть они голову поломают!

Операции с массивами

Одному массиву можно *присвоить* значение другого массива, при условии, что они имеют одинаковый тип. Но тут есть определённая тонкость. Если объявить тип массива, то все переменные этого типа можно присваивать друг другу:

```
type
  mas= array[1..10] of integer;
var
  a1,a2: mas;

a1:= a2;
```

Если переменные объявить в одном списке, то они также будут иметь *одинаковый* тип:

```
var
  a1,a2: array[1..10] of integer;
```

Однако совершенно одинаковые массивы, объявленные раздельно, однотипными не считаются:

```
var
  a1,a2: array[1..10] of integer;
```

```
  a3: array[1..10] of integer;
```

и операция присваивания `a1:= a2;` вызовет ошибку несовместимости типов!

Для всех массивов определены **функции**:

- **length** – возвращает число элементов в массиве (для *a1* - 10)
- **low** – возвращает начальный (минимальный) индекс массива (для *a1* - 1)
- **high** – возвращает конечный (максимальный) индекс массива (для *a1* - 10)

Если присоединить к программе модуль **Math**, то станут доступны функции для **числовых массивов** целого типа и типов *double* и *extended*.

Объявим массив целых чисел и заполним его:

```
var
  a1: array[1..10] of integer;

  for n := 1 to 10 do   a1[n]:= n;
```

Теперь с помощью функции **MaxIntValue** мы легко найдём минимальное и минимальное значение в массиве:

```
MinIntValue(a1) → 1
MaxIntValue(a1) → 10
```

И при этом нам не пришлось перебирать элементы массива!

Функции для массивов вещественного типа работают аналогично:

```
var ad: array[1..10] of double;

for n := 1 to 10 do ad[n]:= n;
```

Аналогично находим

```
MinValue(ad) → 1    - минимальное значение
MaxValue(ad) → 10   - максимальное значение
```

Дополнительно мы можем получить и **сумму** всех элементов массива:

```
Sum(ad) →55
```

Многомерные массивы

Поскольку типом массива может снова быть массив, то мы можем объявлять и массивы массивов, то есть **многомерные** массивы. Чаще других в программах встречаются дву- и трёхмерные массивы:

```
Var
  a2: array[1..10, 1..8] of integer;
  a3: array[1..10, 1..8, 1..2] of integer;
```

или

```
Var
  a2: array[1..10] of array[1..8] of integer;
  a3: array[1..10] of array[1..8] of array[1..2] of integer;
```

Обычно используется первая, более короткая запись.

Иногда удобнее объявить сначала тип нужного массива, а уже потом – переменные этих типов:

```
type
  t2= array[1..10, 1..8] of integer;
  t3: array[1..10, 1..8, 1..2] of integer;
Var
  a2: t2;
  a3: t3;
```

Двумерные массивы также называют **матрицами**. Например, массив **a2** описывает матрицу (таблицу), состоящую из 10 строк и 8 столбцов.

К каждому элементу многомерного массива можно **обратиться** по его индексам, которые указываются в квадратных скобках через запятую:

a2[3, 2] – элемент массива, который находится в третьей строке, во втором столбце.

Как и в случае одномерных массивов, объявление многомерного массива можно сочетать с **присвоением** начальных значений его элементам. При этом значения для каждой размерности заключаются в круглые скобки:

```
var
  a: array[1..4, 1..2] of integer = ((0,1), (2,3), (4,5), (6,7));
```

Например, значение элемента **a[3, 2]** равно 5. Чтобы легче было представить двумерный массив в виде таблицы, запишем объявление массива иначе:

```
var a: array[1..4, 1..2] of integer = ((0,1),
                                       (2,3),
                                       (4,5),
                                       (6,7));
```



С многомерными массивами мы сталкиваемся в школе, это

1. **Таблица умножения:**

   ```
   Var tab: array[2..9, 2..9] of integer;
   ```

   Значения элементов массива – произведение номера столбца на номер строки.

2. **Шахматная доска:**

   ```
   Var field: array['a'..'h', 1..8] of figura;
   ```

   В каждой клетке может стоять чёрная или белая фигура (или пешка), или клетка может быть пустой.

3. **Школьный журнал.**

   Введём константы для максимальных значений:

```
const
  число_предметов = 10;
  число_учеников  = 30;
  число_занятий = 100;
```

Если в вашей школе другие порядки, то подставьте другие значения!

Объявим новые типы:

```
type
  оценка= 1..5;
  школьный_журнал=              array[1..число_предметов,
1..число_учеников, 1..число_занятий] of оценка;
```

и переменную, в которой будет храниться школьный журнал 5[а] класса:

```
var
  журнал5а_класса: школьный_журнал;
. . .
```

Если первый предмет алгебра, Шурик - 13-ый ученик в списке класса и на третьем занятии он получил пятёрку, то можно записать её в школьный журнал:

```
журнал5а_класса[1,13,3]:= 5;
```

Конечно, всю эту безликую цифирь следует заменить нормальными словами, но к массивам это отношения не имеет.

Объявим массив целых чисел:

```
Var
  a2: array[1..5] of array[1..10] of integer;
```

и посмотрим, как он располагается в *памяти* компьютера (Рис. У9.2).

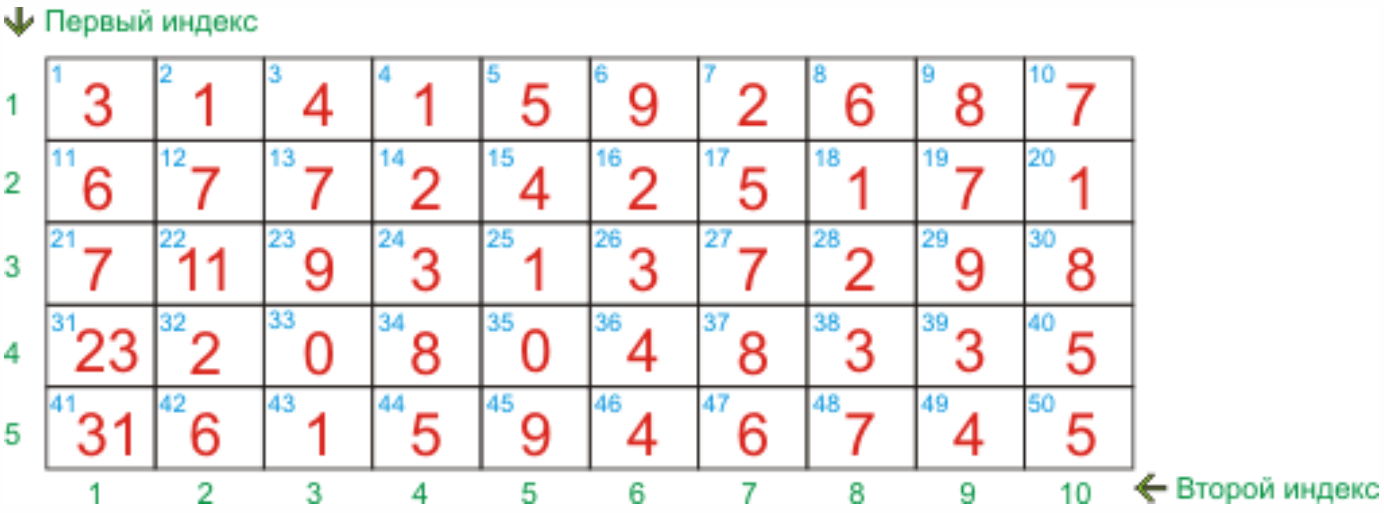


Рис. У9.2. Двумерный массив в виде таблицы

Очень похоже на несколько одномерных массивов, уложенных стопкой, но на самом деле все элементы массива хранятся в памяти не в виде таблицы, а последовательно – как и одномерные массивы (Рис. У9.3).



Рис. У9.3. Двумерный массив в памяти компьютера

Нам же удобнее представлять себе двумерные массивы именно *первым* способом.

Семинар 9. Латинские квадраты, или Квадратно-гнездовой посев чисел

...
сама садик я садила,
сама буду поливать.

В. Маяковский. *Во весь голос*

Латинскими квадратами называют таблицы, в которых каждая строка и каждый столбец содержат по 1 разу все латинские буквы, начиная с A. Общее количество разных букв определяется размерами квадрата. Например, для квадрата 4 x 4 клетки в каждом столбце и строке будут расположены буквы A, B, C, D.

Важно отметить, что в диагоналях буквы могут и повторяться.

Хотя первоначально таблицы заполнялись латинскими буквами (что и дало название квадрату), гораздо удобнее расставлять в клетках числа от 1 до размерности квадрата.

Теорию построения латинских квадратов разработал великий швейцарский учёный Леонард Эйлер. Сейчас латинские квадраты используют при планировании экспериментов (например, при исследовании урожайности различных сортов злаков), в теории кодирования, в секретной связи. Более «легкомысленное» применение латинских квадратов мы найдём в головоломках – сквэрвордах, футошики, диаго и всем известных судоку.

Но нас интересуют сейчас латинские квадраты не сами по себе, а только как таблицы, заполненные числами, то есть двумерные массивы.

Интерфейс программы нам уже хорошо знаком (Рис. С9.1), поэтому сразу нажмём кнопку с волшебной палочкой, которая всё за нас и решит:

```
procedure TfrmMain.sbtSolveClick(Sender: TObject);
begin
  generate(speNum.Value);
end;
```

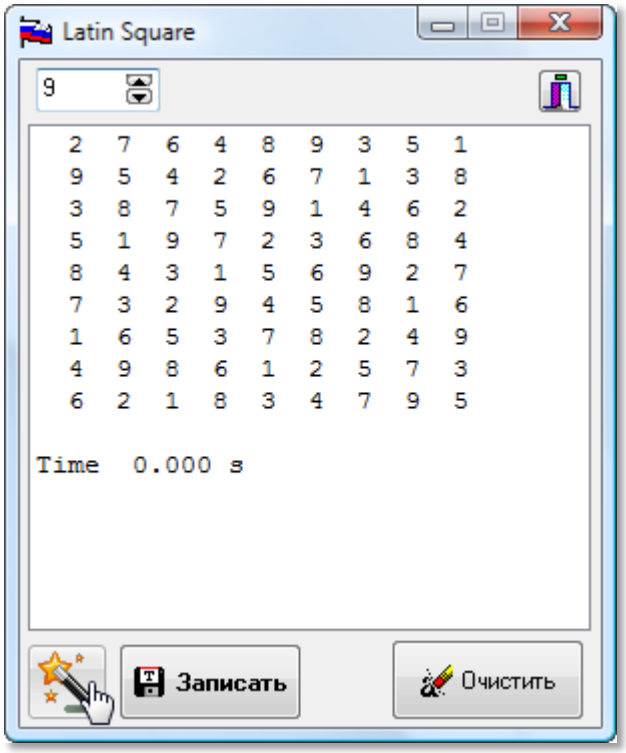


Рис. С9.1. Латинский интерфейс

Конечно, никакое чудо не произойдёт, если его заранее не подготовить:

```
//ГЕНЕРИРОВАТЬ 1 ЛАТИНСКИЙ КВАДРАТ
procedure generate(number: integer);
var i,j,k,n,x: integer;
    B,C,D: array[1..20] of integer;
    s: string;
    time,time0: single;
    field: array[0..20,0..20] of integer;
begin
  Randomize;

  frmMain.lstProtokol.Clear;
  time0:=GetTickCount(); //засечь начало

  for i:= 1 to number do begin
    B[i]:= i; C[i]:= i; D[i]:= i;
  end;

  n:= number;
  //формируем «случайный» столбец:
  for i:= 1 to number do begin
      x:= random(n)+1;
      B[i]:= C[x];
      dec(n);
      for k:= x to n do C[k]:= C[k+1];
  end;
```

```
  n:= number;
  //формируем «случайную» строку:
  for i:= 1 to number do begin
      x:= random(n)+1;
      C[i]:= D[x];
      dec(n);
      for k:= x to n do D[k]:= D[k+1];
  end;

  //составляем случайный латинский квадрат:
  for j:= 1 to number do
    for i:= 1 to number do begin
      n:= B[j]+C[i];
      if n > number then n:= n - number;
      field[j,i]:= n;
    end;

  for  j:= 1 to number do begin
    s:='';
    for i := 1 to number do begin
      if field[i,j] < 10 then s:= s + '  '
      else s:= s + ' ';
      s:= s + inttostr(field[i,j]);
    end;
    frmMain.lstProtokol.Items.Add(s);
  end;

  time:=(GetTickCount()-time0)/1000;
  str(time:6:3,s);
  frmMain.lstProtokol.Items.Add('');
  frmMain.lstProtokol.Items.Add('Time ' + s + ' s');
  application.ProcessMessages;
end;
```

Хитроумный алгоритм, по которому генерируется случайный латинский квадрат, мы разбирать не будем, однако заметим, что для его реализации нам потребовались 3 вспомогательных одномерных массива B, C, D:

```
B,C,D: array[1..20] of integer;
```

Для самого латинского квадрата мы подготовим двумерный массив:

```
field: array[1..20,1..20] of integer;
```

После заполнения и перемешивания массивов **B** и **C** в двух циклах *for* из них составляется латинский квадрат:

```
  for j:= 1 to number do
    for i:= 1 to number do begin
      n:= B[j]+C[i];
```

```
      if n > number then n:= n - number;
      field[j,i]:= n;
    end;
```

Ну, а дальше остаётся только вывести результат в окно списка.

Нажимая волшебную кнопку вновь и вновь, вы будете получать каждый раз новый латинский квадрат, который вы легко сможете сохранить на диске.

Программа благодаря хорошему алгоритму генерирует латинские квадраты практически мгновенно, хотя это не так просто – попробуйте сами составить хотя бы маленький квадрат!



Исходный код программы находится в папке **latin**.

Самостоятельная работа



Наверное, вы обратили внимание на то, что объявления массивов ограничивают максимальный размер квадрата 20-ю клетками. Этого более чем достаточно, но лучше заменить это число константой – чтобы в случае необходимости можно было бы легко модифицировать программу.

Урок 10. Строки, или Жестокие игры со словами

Вначале было слово,
А потом слова, слова, слова…!

Афоризм

Людям привычнее общаться с помощью слов, а не чисел, поэтому для вывода сообщений и другой текстовой информации почти в каждой программе используются строки. О важности этого типа данных говорит и тот факт, что в паскале был только 1 тип строк, а в *Delphi* их огромное количество!



Очень часто мы будем рассматривать строки как одномерный массив символов типа **Char**, но при этом следует помнить, что строки это не разновидность массива, а отдельный тип данных. К ним можно применять операции, отсутствующие у массивов, например, конкатенацию («сложение» строк).

Короткие строки

Тот самый, «паскалевский», тип строк в *Delphi* называется **ShortString** – *короткая строка*. Он используется всё реже и оставлен только для совместимости с более ранними версиями *Delphi*. **Объявляют** короткие строки с помощью зарезервированного слова **string**, после которого, подобно массивам, в квадратных скобках указывают максимальную длину строки в диапазоне от 1 до 255:

```
var sPascal: string[10];
```

Сразу после объявления длина строки *sPascal* равна нулю, хотя в памяти компьютера она *всегда* будет занимать 11 байтов. Почему 11, а не 10, как указано в объявлении?

Дело в том, что, действительно, для хранения каждого символа в памяти компьютера требуется 1 байт, что для 10 символов и составляет 10 байтов. А ещё 1 байт нужен для хранения текущей длины строки. Он имеет *нулевой* индекс в массиве, а символы строки – от 1 до заданной максимальной длины строки. Так как под хранение длины строки отводится 1 байт, то и длина строки не может превышать 255 символов.

Чтобы яснее представить, как хранятся строки в памяти компьютера, обратимся к Рис. У10.1, верхний.

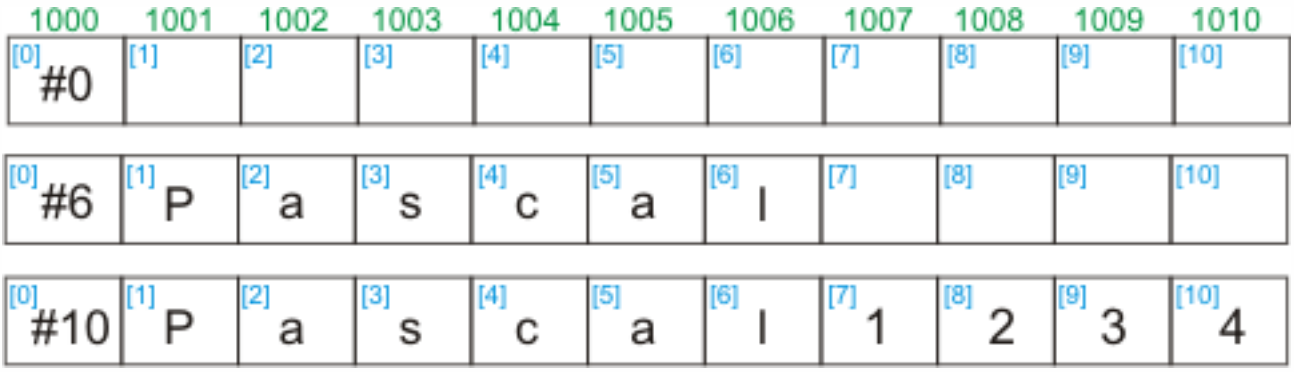


Рис. У10.1. Строки в памяти компьютера

Итак, при запуске программы процессор компьютера выполняет машинный код, соответствующий строке исходного кода `var sPascal: string[10];` При этом создаётся переменная *sPascal*, под неё отводится 11 байтов памяти, начиная, например, с адреса 1000 (конечно, адрес вымышленный, но точный адрес нам и не нужен) и по адресу 1000 заносится число 0 (на самом деле не число, а символ с кодом 0!), соответствующее текущей длине строки, то есть наша строка пустая: `sPascal = ''`.

Присвоим строке какое-нибудь значение, к примеру, *Pascal*:

```
sPascal:='Pascal';
```

Содержимое памяти изменится (Рис. У10.1, средний). Теперь по адресу 1000 находится число 6 (длина строки), а далее - символы строки, которые имеют тип *AnsiChar.*

Если присваивать переменной *sPascal* другие значения, то, соответственно, будет изменяться и содержимое ячеек памяти. Как именно, вы теперь легко можете догадаться. Нас могут заинтересовать только 2 предельных случая. Чтобы *уничтожить* строку (не саму переменную, а только её значение!), достаточно её сделать пустой:

```
sPascal := '';
```

В этом случае строка вернётся в исходное состояние, показанное на рисунке Рис. У10.1, верхний.

Если переменной присвоить строку, длина которой *больше* указанной в объявлении максимальной, например,

```
sPascal:='Pascal123456';
```

то все «лишние» символы будут просто проигнорированы, и в данном случае длина строки будет равна 10, а её значение – *Pascal1234*. Эту ситуацию отражает Рис. У10.1, нижний.

Таким образом, *Delphi* нещадно пресекает все ваши - даже неумышленные – попытки испортить программу. В самом деле, под переменную отводится 10 байтов памяти, а вы желаете втиснуть туда строку, длина которой на 2 символа больше. Значит, остальные символы должны занять следующие 2 байта памяти, в которых уже есть что-то жизненно необходимое для программы, и тем безнадёжно испортить её. А сможете ли вы потом найти ошибку в исходном коде?

Впрочем, если вы ярый последователь экстремального программирования, переходите на язык *C* и вытворяйте с памятью компьютера самые непотребные вещи, о которых здесь даже стыдно упоминать.

Поскольку короткая строка это почти массив символов, то мы легко можем добраться до каждого из них по его *индексу*. Снова обратим свой пылающий взор на Рис. У10.1. На нём синим цветом выделены индексы массива, то есть номера символов в строке. Из него ясно видно, что `sPascal[1] = 'P'`, `sPascal[2] = 'a'`, и так далее до `sPascal[10] = '4'`.

Но вы никогда не узнаете, что же находится дальше - `sPascal[11]`: *Delphi* сочтёт ваше любопытство необоснованным и выдаст ошибку при первой же попытке запустить программу.



Наверное, вы думаете, что значение *sPascal[0]* имеет тип *byte* и вы легко сможете узнать длину строки? – А вот и нет! Оно тоже имеет тип *AnsiChar* – как и символы строки (это значит, что для *Delphi* там находится не число, а символ, код которого равен длине строки). Но, «если нельзя, но очень хочется, то можно», поэтому просто объявите новую переменную типа *byte* (или *integer*)

```
var n: byte;
```

и получите длину строки

```
n:= byte(sPascal[0]);
```

Или так:

```
n:= ord(sPascal[0]);
```

Хотя можно было не стараться, а просто расширить кругозор, найти функцию **Length** и воспользоваться ею:

```
n:= Length(sPascal);
```

Результат будет тем же самым, но программа лишится части своей загадочности, которая так же гипнотически действует на неокрепшие умы, как удав на кролика.

Если мы имеем право считать значение элемента *sPascal[0],* то мы можем и записать туда всякую ерунду. Например, `sPascal[0]:= #255;` (почему не просто 255?) – и длина строки увеличится до 255. Однако попробуйте что-нибудь записать в новую строку: `sPascal[11]:= 'Я';` - всё равно ничего не получится.

Для таких крамольных действий имеется и встроенная процедура **SetLength**`(sPascal, 255);`, которая действует аналогично и столь же бесполезна для коротких строк. Разве что вы захотите отсечь хвост у текущей строки. После выполнения процедуры **SetLength**`(sPascal, 2);`, останутся только 2 первых символа «Ра». Но это па вы можете станцевать и более красиво (см. урок Процедуры и функции для работы со строками)!

Если строка **неполная**, то есть короче максимальной длины, то значения символов за последним символом строки не определены! Обратимся к среднему рисунку. Первые 6 байтов памяти занимает строка *Pascal*, а дальше находятся ещё 4 «пустые» ячейки памяти. Не обольщайтесь, они не пустые, в них что-нибудь да есть!

Если вы присваивали строке значение *Pascal123456*, а потом *Pascal*, то в них останутся символы *1234*, хотя длина строки теперь только 6 символов. В других случаях эти ячейки могут занимать любые символы, поэтому никогда не используйте в своих программах значения символов за пределами присвоенного строке значения!



Если вы создаёте несколько строковых переменных равной длины, то удобнее объявить новые типы строк:

```
Type
   str5 = string[5];
   str10 = string[10];
```

Затем вы можете использовать эти типы для объявления строковых переменных ограниченной длины:

```
Var
  s1, s2: str5;
  s3, s4, s5: str10;
```

Длинные *ANSI*-строки

AnsiString, или **длинные строки**, – основной тип строк в *Delphi* до последнего времени. Они могут содержать до 2^{31} символов и занимать в памяти компьютера до 2 гигабайтов.

Для **объявления** длинных строк также используется зарезервированное слово **AnsiString**, но максимальная длина строки в скобках не указывается, поскольку длина этих строк изменяется *динамически*:

```
var sDelphi: AnsiString;
```

Но главное отличие длинных строк от коротких вовсе не в длине строки. Символы длинной строки, как и короткой, образуют в памяти последовательность символов типа **AnsiChar**, подобную массиву, после которых *Delphi* автоматически добавляет **нулевой символ #0**, который и является **признаком конца строки**. Кроме того, в четырёх байтах перед первым символом строки хранится текущая длина строки (и другая информация).

При изменении строки *Delphi* автоматически уменьшает или увеличивает размер памяти под строку так, чтобы она целиком могла там разместиться. Это значит, что вам не нужно заранее заботиться и высчитывать максимальную длину строки, в то время как при объявлении коротких строк это делать необходимо. Более того, даже пустая короткая строка всегда занимает в памяти заказанное число байтов. Интересный вопрос: сколько памяти занимает пустая длинная строка? Интересный ответ: нисколько, пустые строки в памяти не хранятся. Так что при *объявлении* длинной строки система выделяет только 4 байта для хранения адреса памяти, начиная с которого будет размещаться сама строка, и для пустой строки там будет константа **nil**.



Строки с нулевым символом в конце используются во многих языках программирования, например в *C* и *C++,* а также в *Windows API*, чем и вызвано появление таких строк в *Delphi*.

Таким образом, переменная типа *AnsiString* – это **указатель** на область памяти, в которой хранятся символы строки.

Как мы уже выяснили, при объявлении длинной строки место в памяти для неё не выделяется. Если же присвоить строке значение (не пустую строку!), например,

```
sDelphi:='Delphi';,
```

то в памяти выделяется столько байтов, чтобы поместились все символы заданной строки, плюс ещё 1 байт для нулевого символа в конце строки (не нуля, а нулевого символа типа *AnsiChar*!), плюс 4 байта для хранения длины строки (они занимают в памяти место **перед** первым символом строки), плюс ещё 4 байта для подсчёта ссылок на строку (нам это значение не потребуется) + 4 байта для хранения кодовой страницы.

В картинках это выглядит так (Рис. У10.2).

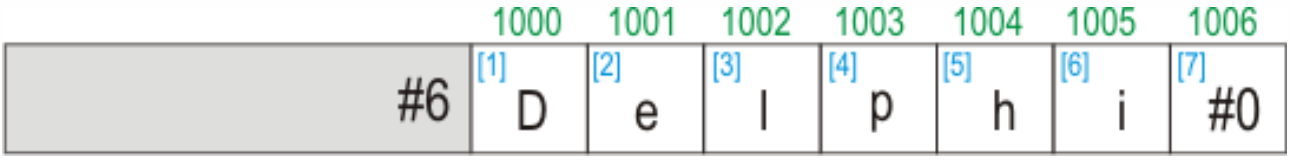


Рис. У10.2. Длинная строка типа *AnsiChar* в памяти компьютера

Очень похоже на короткую строку, к которой добавлен нулевой символ. Поэтому доступ к отдельным символам строки осуществляется точно так же, как и в случае коротких строк – по их индексу в массиве символов. Запомните, что индекс первого символа равен 1!

Таким образом, если

```
Var ch: AnsiChar;,
```

то после присвоения символьной переменной **ch** значения первого символа строки *sDelphi*

```
ch:= sDelphi[1];
```

её значение будет равно *D*, второго символа - *e*, шестого - *i*.

Если задать индекс, превышающий длину слова (в нашем случае 7 и больше), то значение переменной **ch** будет равно нулевому символу.

Длинные строки *Юникода*

Длинные строки Юникода типа **UnicodeString** – основной тип строк в последних версиях языка *Delphi*. Они очень похожи на длинные *ANSI*-строки, но для каждого символа строки в них отводится **2** байта, поэтому они могут содержать до 2^{30} символов (вдвое меньше, чем строки *ANSI*) и занимать в памяти компьютера до 2 гигабайтов.

Для **объявления** длинных строк используются зарезервированные слова **UnicodeString** и **string**:

```
var sDelphi:  string;
    sDelphi2: UnicodeString;
```

Таким образом, ключевое слово **string** – это всего лишь псевдоним более длинного слова **UnicodeString**. Оба эти слова обозначают один и тот же тип данных.

Длинные строки Юникода имеют ту структуру и те же свойства, что и *ANSI*-строки, поэтому мы не будем повторяться.



Объявление глобальных строковых переменных и типизированных констант можно совмещать с их инициализацией, то есть присваивать им начальное значение:

```
var sDelphi: string ='Delphi';
const STR_DELPHI='Delphi';
```

Обратите внимание: нулевой символ после строки указывать не нужно, он будет добавлен автоматически!

Другие типы строк

Строки **WideString** – основной тип строк платформы *.NET* - очень похожи на длинные строки Юникода, но символы имеют тип не **UnicodeChar**, а **WideChar** (эти типы совместимы с типами *string* и *char*). Они занимают 2 байта памяти каждый. Этот тип строк используется исключительно для работы с объектной моделью компонентов для ОС *Microsoft* (*Component Object Model, COM*).

Строки типа **PAnsiChar, PChar** (**PUnicodeChar**) и **PWideChar**– также очень похожи на длинные строки *Delphi* (точнее, наоборот: длинные строки похожи на эти строки), они также заканчиваются нулевым символом, но в

строках этих типов отсчёт индексов ведётся **от нуля**, а не от единицы (вечная головная боль от массивов в языке *C*).

Строки типа **PChar** используются не только в языках *C* и *C++* (нам до этого и дела нет), но и в функциях *Windows API*, без которых обойтись не удастся.

Нам придётся использовать эти невнятные строки только для передачи строковых параметров в чуждые нам функции, поэтому подробно рассматривать их не будем.

Мешанина из строк разных типов

Таким образом, строки типа *string* сочетают в себе все достоинства коротких строк паскаля и строк **PChar** языка *C*, избегая при этом их недостатков. Им можно присваивать значения коротких строк и строк **PChar**:

```
var
  sPascal: string[6];
  sDelphi: string;
  szDelphi: PChar;

  sPascal:= 'Pascal';
  sDelphi:= sPascal;    // значение переменной sDelphi = „Pascal"
  szDelphi:= 'Delphi0';
  sDelphi:= szDelphi;   // значение переменной sDelphi = „Delphi0"
```

Строки **PChar** вреднее строк *Delphi*: они не позволяют присвоить им всуе значения ни коротких, ни длинных строк. Для этого необходимо явно преобразовать их к типу **PChar**:

```
szDelphi:= PChar(string(sPascal));
szDelphi:= PChar(sDelphi);
```

Очень часто в приложениях для вывода сообщений используется функция *Windows API* **Application.MessageBox**:

```
function MessageBox(const Text: PChar;
                    const Caption: PChar;
                    Flags: Integer): Integer;
```

Первые два передаваемых ей параметра – текст сообщения и заголовок окна - имеют тип *PChar*. Если вы передаёте в функцию строковые константы (выражения), то просто укажите их при вызове функции:

```
Application.MessageBox('Мешок пустой!', 'Лото', MB_OK);
```

Если же ваши строки имеют тип *string*, то следует привести их к типу *PChar*:

```
Var
  sMessage: string;
  sTitle: string;

sMessage:= 'Мешок пустой!';
sTitle:= ' Лото';

Application.MessageBox(PChar(sMessage), PChar(sTitle), MB_OK);
```

Впрочем, для таких случаев лучше сразу использовать тип *PChar*:

```
Var
  sMessage: PChar;
  sTitle: PChar;

sMessage:= 'Мешок пустой!';
sTitle:= ' Лото';

Application.MessageBox(sMessage, sTitle, MB_OK);
```

Простые операции со строками

Строки типа **string** можно объединять в одну строку с помощью оператора **+** (оператор сцепления строк; сама операция называется **конкатенацией** строк). В качестве операнда можно использовать не только длинную, но и короткую строку, а также символьные и строковые переменные и константы.

```
Var
  s1, s2, s3: string;

s1:= ' string1';
s2:= ' string2';
s3:= ' s3 = ' + s1 + ' + ' + s2;
```

Значением строки **s3** будет новая строка *s3 = string1 + string2*. Длина строки будет равна длине всех строковых операндов, входящих в выражение. В нашем случае – 22.

Эти и многие другие операции со строками можно выполнить с помощью процедур и функций, рассматриваемых на одном из следующих уроков.

Для сравнения строк используются операторы отношения (< > = <> <= >=). Результат сравнения имеет логический тип, то есть значение *true* или *false*.

Сравнение строк производится слева направо, начиная с первого символа.

Пустая строка всегда меньше строки, имеющей символы:

`'' < 'Delphi'` (то есть значение этого выражения равно *true*, а выражения `'' > 'Delphi'` - *false*).

Если очередные символы двух сравниваемых строк не совпали, то меньше та, код символа которой меньше:

```
'ab' < 'ba'
'ab' < 'ac'
'abc' < 'abde'
```

Если строки имеют разную длину и все символы более короткой строки совпали с соответствующими символами более длинной строки, то меньше первая строка:

```
'abc' < 'abcd'
```

Если строки имеют одинаковую длину и все символы совпали, то строки *равны*:

```
'abc' = 'abc'
'abcd' = 'abcd'
```

Семинар 10. Игра *Виселица*, или *Hangman*

Для более близкого и приятного знакомства со строковыми переменными мы напишем одну занятную и полезную программу, которая поможет нам избегать печальной участи при игре в *Виселицу*. Но прежде мы должны хорошенько разобраться во всех премудростях игры и найти оптимальную стратегию, позволяющую продлить наше бренное существование насколько это возможно. Затем на основе этой стратегии мы разработаем алгоритм, который, по сути, и является программой. Всегда поступайте точно так же «алгоритмично» со всеми своими проблемами - и смело можете отмечать День программиста на 256-ой день каждого календарного года.

Игры на отгадывание – одни из наших первых детских игр, вспомните хотя бы игру *Холодно-горячо*. Повзрослев и возмужав, вы наверняка приобщились и к другим играм на отгадывание – таким, как:

- **Виселица** (нужно отгадать слово или фразу)
- **Быки и коровы** (отгадать число)
- **Мастермайнд** (отгадать комбинацию цветных фишек)
- **Отгадать слово** (его и отгадать)
- **Морской бой** (отгадать расположение кораблей противника)
- **Угадай животное** (отгадать название животного)

Подобные игры развивают творческие способности, логическое мышление, умение правильно ставить вопросы и находить на них ответы - то есть очень полезны в любом возрасте.

В отличие от **игр с полной информацией** (шахматы, шашки, го, отелло...), игры на отгадывание - это типичные **игры с неполной информацией**: игрок не знает, чт**о** именно задумано, и пытается это угадать.

Суть этих игр такова. Один игрок что-то (слово, число, предмет, расстановку, расположение объектов) задумывает, а второй ищет разгадку. Для этого он последовательно задаёт первому игроку по одному вопросу (или делает другой "ход" - в зависимости от вида игры). Тот отвечает, и на основе полученной информации второй игрок задаёт следующий вопрос (высказывает догадку). И так до тех пор, пока либо не будет найден правильный ответ, либо не закончатся попытки.

Часто оба игрока выполняют одновременно роль и загадчика, и отгадчика. В этом случае побеждает тот, который раньше найдёт решение, то есть задаст для этого меньше вопросов или выскажет догадок...

Так как игроку приходится отвечать на вопросы, эти игры называется также **тестовыми**.

Виселица - это одна из наиболее популярных игр среди школьников, студентов и других бездельников. Широко известны также её телевизионные реализации – *Колесо фортуны* - «*Wheel of Fortune*» (у них) и *Поле чудес* (у нас). Они более гуманны: в них никого не вешают, а даже награждают призами, а также подсказывают, какое слово нужно отгадать.

Происхождение игры

Учитывая зверства по ходу игры (медленное повешение, больше похожее на четвертование), можно предположить её британское происхождение. Очевидно, это так и есть, но когда она возникла, не знает никто. Вероятно, в эпоху королевы Виктории.

Что достоверно известно, так это то, что в 1894 году *Alice Bertha Gomme* в книге *The Traditional Games of England, Scotland and Ireland* описала подобную игру под названием *Birds, Beasts and Fishes*. Правила этой игры очень просты: ведущий пишет первую и последнюю буквы названия какого-либо животного, а второй игрок отгадывает его, последовательно называя буквы.

В других источниках игра называется также *Gallows* или *The Game of Hanging.* Немцы называют её так – *Galgenmännchen*, или *Galgenmann*, или просто *Galgen*.

Правила игры

Играют двое. Один загадывает слово, другой отгадывает его, называя каждый раз по одной букве.

Загадывающий игрок (первый) рисует на бумаге столько чёрточек или точек, сколько букв в искомом слове (таким образом, количество букв в слове, или его длина, отгадывающему игроку известны всегда). Например, первый игрок задумал слово «железо», тогда он рисует на листке бумаги такую фигуру:

_ _ _ _ _ _ или . . . . . .



Замечание 1. Обычно в «Виселице» используются те же слова, что и в других словесных играх, то есть имена существительные, нарицательные, в единственном числе (или во множественном, если единственное число отсутствует, - *штаны*, *ножницы*, *шахматы*), в именительном падеже. Специальные термины, ненормативная лексика и тому подобные слова, как правило, не употребляются, но, по договорённости, можно ограничить выбор слов какой-либо темой (это не только облегчает поиск загаданного слова, но и позволяет целенаправленно обогащать словарный запас).

Так как очень коротких и очень длинных слов мало, то в игре используют 4-12-буквенные слова. Более короткие слова употреблять не рекомендуется, более длинные (до 15 букв) можно, но среди них мало интересных.

Вместо слов можно брать какие-либо фразы, пословицы, названия книг, фильмов и так далее. На немецком телевидении была эта модификация *Виселицы*, - но это уже совсем другая игра.

Замечание 2. В классической игре никаких подсказок к загаданному слову не дают (исключая информацию о числе букв, см. также предыдущее замечание). В одном из вариантов первый игрок сразу выписывает первую и последнюю буквы слова. В нашем примере условие задачи выглядело бы так:

Ж _ _ _ _ **О**.

Причём, если эти буквы повторяются в загаданном слове, то они не открываются и их придётся называть ещё раз.

Безусловно, такие подсказки существенно облегчают игру, поэтому этот вариант игры можно рекомендовать начинающим «висельникам».

Другая разновидность подсказок – указание на область применения загаданного слова (см. **Замечание 1**) или его толкование (такой приём используется, например, в капитал-шоу *Поле чудес*). Это может быть разумно, когда загадываются редкие слова или игра используется в учебных целях. Кстати говоря, *Виселица* весьма популярна среди учителей Германии. Особенно при изучении немецкого правописания, иностранных слов, названий стран, столиц и тому подобных слов. Однако педагоги рекомендуют и настаивают на рисовании не бесчеловечного орудия убийства, коим, без

сомнения, является виселица, а какого-нибудь зверька, например, свиньи (хотя по-русски это называется «хрен редьки не слаще»).

Каждый **ход** состоит в том, что **отгадывающий игрок** (второй) называет какую-либо букву, а загадывающий, в зависимости от того, имеется ли эта буква в слове или нет, делает следующее:

- Если названная буква в слове **присутствует**, он пишет её на листке над теми чёрточками, где эта буква находится в слове.

- Если буква встречается в слове **несколько раз**, она должна быть вскрыта везде. Если в нашем примере будет названа буква Е, то ведущий должен написать на листке:

_ **Е** _ **Е** _ _

- Отсутствующие буквы принято писать **под** чёрточками.



Замечание 3. Буквы **Е** и **Ё**, а также **И** и **Й** считаются различными, в отличие от удивительного *Поля чудес* и не менее удивительных современных "кроссвордов"...

- Если названная буква в слове **отсутствует**, ведущий рисует часть виселицы:

```
 +----+
 |/   |
 |    O
 |   /|\
 |   / \
-+-
```

Последовательность такая: основание – столб – перекладина – распорка – верёвка – голова – тело – левая рука – правая рука – левая нога – правая нога.



Замечание 4. Вместо очередной буквы отгадчик может назвать и целое слово. Если оно совпадает с загаданным, то он выигрывает, а игра заканчивается. В противном случае ход считается неверным, и игрок получает следующую часть виселицы. Этот приём используется, например, в «**Поле чудес**».

Замечание 5. Иногда распорка рисуется также и внизу.

Замечание 6. В более жестоком варианте игры, виселица присутствует изначально, наводя страх и ужас на соперника.

Замечание 7. Для противника рисуется не виселица, а решётка тюрьмы:

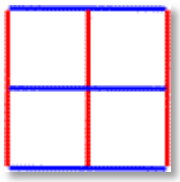


Рис. С10.1. Игровой гуманизм!

А можно просто писать по буквам слово *ВИСЕЛИЦА*.

Игра заканчивается, если

- слово отгадано;
- отгадчик повешен.



Замечание 8. Так как виселица (или заменяющая её фигура) может быть сделана из разного числа составных частей, то и количество неверных ходов может быть разным – от 6 до 12. Конечно, чем больше допускается сделать ошибок, тем легче отгадать слово. Шести ошибочных ходов, безусловно, слишком мало, чтобы рекомендовать этот вариант для серьёзной игры. Вариант с семью ошибочными ходами также не позволяет гарантированно отгадывать любое загаданное слово даже при хорошей игре (именно столько попыток даётся, например, на сайте germany.ru).

Когда сразу открываются первая и последняя буквы слова (особенно "тематического"), количество попыток может быть и меньше - 5-7.

Варианты игры

Кроме уже рассмотренных, имеются ещё такие.

При одновременной игре вдвоём оба игрока пишут на листочке свои задания. В этом случае каждый из них играет две роли – и загадчика, и отгадчика. В остальном правила игры те же, что и в основном варианте. Это значит, что независимо от числа попыток, игра заканчивается вничью, если оба отгадали задуманные слова.

При большом количестве игроков слова загадывают по очереди. Все игроки (исключая загадчика) отгадывают одно и то же слово. Выигрывает тот, кто назовёт его раньше. Нечто подобное мы можем наблюдать в игре *Поле чудес*. Если бы каждый игрок отгадывал слово синхронно с другими, но независимо от них, этот вариант был бы значительно справедливее и интереснее.



Замечание 9. В *Поле чудес* за каждый результативный ход игрок получает очки. Таким образом, тот, кто играет плохо и долго отгадывает слово, имеет больше шансов набрать много очков. Тот же игрок, который отгадывает слово сразу после объяснения ведущего, без угадывания букв, вообще ничего не получит. Не удивительно, что эта передача так популярна в России.

Безразмерная виселица - длина слова не называется, известно только, что для загадывания используются 4-12-буквенные слова. Очевидно, в этом случае слово будет труднее отгадать, поэтому число попыток следует увеличить.

Шаблонная виселица - от классического варианта отличается только тем, что гласные и согласные буквы в задуманном слове обозначаются разными значками. Например, для слова ЖЕЛЕЗО, исходная позиция будет такой:

_·_·_·

Напоминает азбуку Морзе, но только внешне. Дополнительные подсказки должны существенно облегчить отгадывание слов.

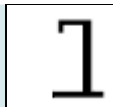
Galgen. Немецкий вариант игры ничем не отличается от нашего (Рис. С10.2).



Рис. С10.2. Немецкая виселица

Пример игры

Продолжим игру, начатую ранее.

Ход 1	 _ _ _ _ _ _ Называем букву **Е**
Ход 2	 _ **Е** _ **Е** _ _ Называем букву **Р**
Ход 3	 _ **Е** _ **Е** _ _ **Р** Называем букву **Л**
Ход 4	 _ **ЕЛЕ** _ _ **Р** Называем букву **А**

Ход 5	 _ ЕЛЕ _ _ РА Называем букву **Н**
Ход 6	 _ ЕЛЕ _ _ РАН Называем букву **С**
Ход 7	 _ ЕЛЕ _ _ РАНС Называем букву **О**
Ход 8	 _ ЕЛЕ _О РАНС Называем букву **Ж**
Ход 9	 ЖЕЛЕ _О РАНС Называем букву **З**
Ход 10	 ЖЕЛЕЗО РАНС
	Слово отгадано!

Стратегии игры

Несмотря на популярность *Виселицы* во всём "цивилизованном" мире, стратегия игры разработана плохо. Попробуем сами выработать **алгоритм** выбора правильного хода.

Стратегия 1. Хаотическая

Принято считать, что *Виселица* - игра на эрудицию, но *Поле чудес* полностью опровергает это мнение. Чаще всего участники этого якобы шоу просто беспорядочно перебирают буквы алфавита. Так как любой алфавит, в том числе и русский, не бесконечен, то слово рано или поздно будет отгадано - даже в том случае, если игрокам оно не известно (это легко определить по их изумлениям и приветам). К счастью, в настоящей *Виселице* число ходов ограничено, так что здесь полностью полагаться на случайные догадки не приходится, посему эту стратегию мы отвергаем.



Замечание 10. Конечно, такая стратегия иногда может привести к быстрой и красивой победе (выигрывают же люди в *Спортлото*!), но в серии игр эта стратегия всегда заканчивается разгромным поражением.

Стратегия 2. Статистическая

Известно, что одни буквы в словах встречаются чаще, другие - реже. Например, для английского языка последовательность букв по убыванию их частоты такая: **e-t-a-o-i-n-s-h-r-d-l-u**. Естественно, подобная последовательность существует и для слов русского языка. Мы её здесь не приводим потому, что средняя частота букв вычисляется по текстам, а в игре используются только существительные да и то в одной форме (см. **Замечание 1**). Совсем не обязательно частота букв в этом случае будет той же, что и для текстов. Например, местоимение **Я** в текстах встречается довольно часто, чего не скажешь о букве **Я** в существительных. Более того, в существительных разной длины (с разным числом букв) и частота букв тоже будет несколько различаться. Так как длина загаданного слова нам точно известна, то лучше воспользоваться статистикой для слов заданной длины.

Главная проблема при игре в компьютерные реализации подобных игр состоит в том, что мы не знаем всего перечня слов, из которого выбирается загаданное. Таким образом, точную статистику получить невозможно, но приличную - вполне вероятно. В качестве основы мы можем выбрать *Толковый словарь русского языка* Ожегова и Шведовой. В нём гарантированно (если это не так, то придётся дополнять его словами и из других словарей)

содержатся все слова, которые могут быть предложены для отгадывания. А также и много других, что будет немного портить нам статистику, но с этим можно мириться (или выбросить из словаря сомнительные слова).

По сути, мы отгадываем не отдельные буквы, а целое слово, поэтому нужно подсчитывать не частоту букв, а частоту слов, содержащих ту или иную букву. На самом деле это не одно и то же. Если одна буква часто встречается в словах по несколько раз, а вторая однократно, то при одной и той же частоте, количество слов с первой буквой будет меньше, чем со второй.

Длина слова	Частота слов, содержащих указанную букву, по убыванию															
4	А	О	Р	Т	Л	К	С	У	Е	И	Н	П	Д	М	В	Б
5	А	О	К	Р	Т	Л	Е	И	Н	С	У	П	М	В	Б	Д
6	А	Р	О	К	И	Е	Н	Т	Л	С	П	У	Д	М	В	Б
7	А	О	К	И	Р	Е	Н	Т	Л	С	П	В	Д	У	М	Б
8	А	О	И	К	Н	Р	Е	Т	С	Л	П	В	Д	М	У	Ь
9	А	О	И	Е	Н	Т	Р	К	С	Л	В	П	Д	М	У	Ь
10	О	И	А	Н	Е	Р	Т	С	К	Л	В	П	Ь	М	Д	У
11	О	И	Н	А	Т	Е	С	Р	В	Л	К	Ь	П	Д	М	У
12	О	И	Н	Т	Е	С	А	Р	В	Л	Ь	К	П	Д	М	З

Итак, стратегия игры такова.

В зависимости от длины загаданного слова последовательно называем буквы из соответствующей строки. Как это ни прискорбно, но мы должны быть солидарны с игроками *Поля чудес*: для 4-9-буквенных слов первой лучше называть именно букву **А**! И только для более длинных слов - **О**.



Замечание 11. Поскольку в словах чаще встречаются гласные, то и начинать игру выгоднее именно с них. В нескольких книгах мне встречались указания - называть гласные в первую очередь. Это имеет смысл при компьютерном отгадывании слова. Если же игрок может полагаться только на себя, то гласные ему помогут значительно меньше, чем согласные. Например, загадано слово **ТРАТА**. Называем букву **А** и получаем ответ: **..А.А.**. Под этот шаблон подходит 43 слова, и попробуйте догадаться, какое слово было задумано! Если же вам повезло отгадать букву **Т**, то ситуация будет гораздо приятнее. Шаблон - **Т..Т.** и всего 9 вариантов, многие из которых легко вспомнятся.

Стратегия 3. Динамическая статистика

Есть в **Стратегии 2** что-то инстинктивное - она никак не учитывает результат хода. Угадали или не угадали мы букву, - но теперь уже не все слова могут быть отгадкой, как в начале игры. Например, мы назвали букву **А**. Если её в слове нет, то и задуманное слово нужно искать только среди тех слов, в которых нет буквы **А**. Аналогично, если буква **А** в слове есть, то мы уже не должны учитывать те слова, в которых её нет. Естественно частота слов, содержащих ту или иную букву, может измениться. Например, отыскивая 4-буквенное слово, мы назвали буквы **А** и **О** и оба раза не угадали. По таблице, следующей мы должны были бы назвать букву **Р**, но, оказывается, что из оставшихся слов больше всего тех, которые содержат букву **Е**. И эта закономерность действует для слов любой длины.

Таким образом, правильнее пересчитывать статистику после каждого хода. Естественно, вручную сделать это невозможно, но для компьютера эта задача - совершенно пустяшная. Поэтому будем полагать, что статистику мы можем обновлять после каждого хода.

Как играть

После того как нам загадали слово, мы должны изо всех слов заданной длины выбрать одно. Если каждый раз называть по одной букве (как вы помните, можно постараться угадать и целое слово, но в компьютерных играх эта возможность не используется), то нам потребуется не меньше ходов, чем букв в загаданном слове (при отсутствии повторяющихся букв!). Так как в *Виселице* учитываются только ошибочные ходы, то каждый раз лучше называть ту букву, которая содержится в наибольшем числе подходящих слов. На первом ходе все слова подходящие, поэтому для 4-9-буквенных слов называем букву **А**, для более длинных - **О**.

После ответа загадчика выбираем все слова, которые могут быть загаданы (с учётом названной буквы - её отсутствием или присутствием в слове и месте (или местах), которое она в искомом слове занимает). Подсчитываем количество слов, содержащих каждую из допустимых букв (то есть ещё не использованных) и выбираем ту букву, которая присутствует в наибольшем числе слов. Называем эту букву. Действуем дальше аналогично - до победы или поражения.

В начале игры, когда слов много, лучшую букву выбрать легко, но в конце игры встречаются ситуации, когда сразу несколько букв имеют равную частоту. Однозначный выбор здесь невозможен. Придётся случайно (или как-то иначе) выбирать из них одну букву. Это значит, что такая стратегия будет давать разные результаты, в зависимости от удачности выбора

лучшей буквы. Иногда приходится полагаться на судьбу несколько раз за игру, так что число плохих ходов при игре по этой стратегии может колебаться, если несколько раз отгадывать одно и то же слово.

Впрочем, как обычно, лучшие результаты даёт объединение усилий компьютера (программы) и игрока (пользователя). На первых ходах, когда подходящих слов много, лучше называть те буквы, которые предлагает компьютер. А когда кандидатов на загаданное слово останется 1-2 десятка, можно самому выбрать из них наиболее вероятные и называть ту букву, которая наиболее часто присутствует в них. Дело в том, что в словаре *Ожегова* заведомо больше слов, чем их используется при загадывании, поэтому вы можете игнорировать сомнительные, хотя формально и подходящие, слова.

Маловероятно, что эта стратегия учитывается при загадывании слов, но, с другой стороны, всем известно, что отгадчики обычно называют достаточно частые буквы. Зная это, загадчик может подбирать такие слова, в которых этих букв нет. Например, загадано 4-буквенное слово. Вы последовательно называете хорошие буквы **АОРТЛКС** (см. таблицу) и после семи попыток не угадываете вообще ни одной буквы! Всего таких "нехороших" слов аж 78 (от **БЕЗЕ** до **ЯЧЕЯ**)! Правда, чем длиннее слово, тем менее вероятна такая ситуация. Для 5-буквенных слов мы получим 48 слов, для 6-буквенных - 9, а среди более длинных таких неудачных слов нет вообще.



Замечание 12. Из этого, кстати говоря, следует, что короткие слова отгадать труднее, чем длинные. Именно по той причине, что в них мало букв (а если они повторяются, так и того меньше (разных, естественно)).

Тем более удивительными выглядят правила *Поля чудес*, когда в супер-игре участнику дают возможность назвать половину букв загаданного слова. В этом случае практически невозможно угадать буквы короткого слова (а если и удастся, так это будут **А** или **О**, от которых проку всё равно не много).

Если ходить не буквами **АОРТЛКС** (**Стратегия 2**), а **АОЕИУЛЯ** (**Стратегия 3**), то всё равно останется 35 слов, в которых нет ни одной из этих букв. Очевидно, что любые другие последовательности могут только ухудшить этот результат. Из этого следует, что при 7 неверных попытках нельзя гарантированно угадать любое 4-буквенное слово. Как я уже говорил, именно столько даётся на сайте *germany.ru*, то есть там играют заведомо нечестно.

Тот же вывод можно сделать и для 5-буквенных слов: какие бы 7 букв вы ни называли, всегда найдутся слова, в которых их нет.

Для 6-буквенных и более длинных слов семи ходов должно быть достаточно.

Вывод простой. Для честной игры число неверных попыток должно быть больше 7 либо длина слов должна быть больше 5 букв.

Чтобы получить шанс выиграть и при "нечестной" игре, иногда следует ходить не лучшей буквой, а случайной. Причём вероятность выбора буквы должна быть прямо пропорциональна её частоте, иначе мы будем выбирать редкие буквы, что опасно.

Программа, помогающая играть в *Виселицу*, - *HangmanHelper*

Нам осталось написать программу, которая будет играть по **Стратегии 3**. Назовём её **HangmanHelper**, то есть помощник висельника.

Мы сосредоточим свои усилия исключительно на реализации разработанного нами алгоритма выбора хода по **Стратегии 3**, что и является сутью программы. Работа с кнопками, текстовыми полями, списками и другими компонентами приложения подробно описана в других программах.

Интерфейс программы - предельно прост (Рис. С10.3).

Сверху находятся *кнопки*:

- Загрузить новый словарь (кнопка **sbtOpenFile**). При запуске программы автоматически загружается словарь Ожегова и Шведовой. Это обычный текстовый файл (в отличие от большинства других подобных программ с абракадаброй вместо словаря, - чтобы никто не мог им воспользоваться), который вы легко можете редактировать. Только имейте в виду, что слова в нём расположены в зависимости от длины, это условие вы нарушать не должны!

- Очистить (кнопка **sbtClear**). Стереть информацию в окне протокола.

- Записать (кнопка **sbtSave**). Если вы хотите сохранить на диске запись партии для последующего анализа, воспользуйтесь этой кнопкой.

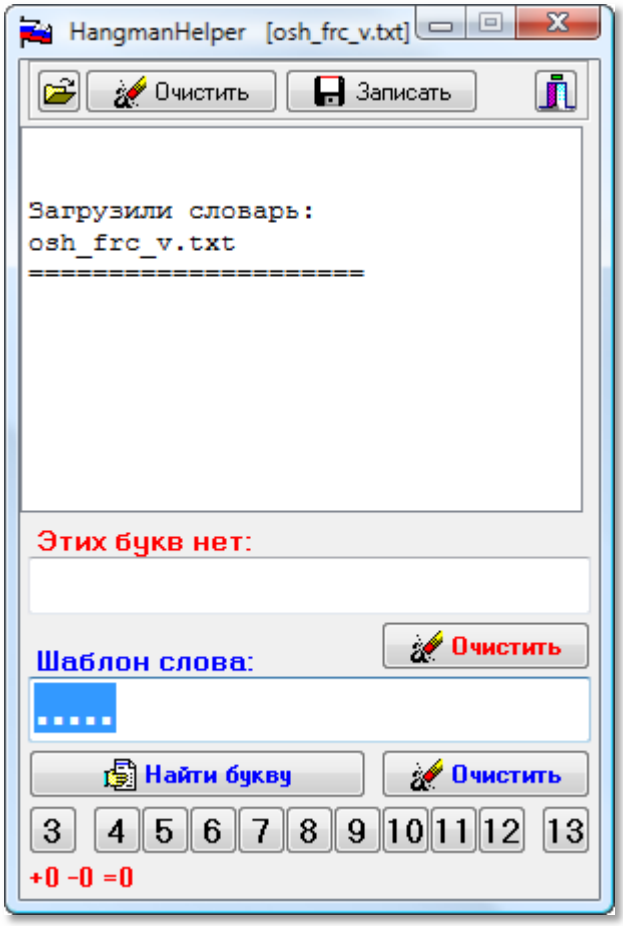


Рис. С10.3. Висельный интерфейс

- Выход из программы (кнопка **sbtClose**)**.** Закрыть приложение.

Под кнопками находится **окно протокола** (список **lstProtokol**). В нём выводится вся необходимая информация: число подходящих слов, список этих слов, количество слов с каждой буквой, лучшие буквы (если их несколько, дополнительно выбирается одна из них) и случайная буква, если вы не хотите ходить лучшей буквой.

Еще ниже находится окно **Этих букв нет:** (текстовое поле **txtLetterMinus**). В него вы должны самостоятельно вписывать те буквы, которых не оказалось в слове (согласно ответу загадчика). Порядок и регистр букв значения не имеют.

Под ним расположено самое главное окно - **Шаблон слова:** (текстовое поле **txtShablon**). В него вы должны - опять же сами - вписывать «правильные» буквы, в нужные позиции.

Кнопка **Найти букву** (**sbtFindLetter**) запускает поиск очередной буквы. Результат поиска отображается в окне протокола.

Две кнопки **Очистить** стирают всю информацию в этих текстовых окнах.

Кнопки **3 - 13** (**btnNumber3.. btnNumber13**) служат для быстрого ввода пустого шаблона.

Строка в самом низу окна программы сообщает число угаданных и не угаданных букв и общее число попыток.

Кодируем алгоритм

Как мы уже договорились, при запуске программы мы сразу же загружаем словарь по умолчанию:

```
////////////////////////////////////////////
//                                        //
//              СОЗДАТЬ ФОРМУ             //
//                                        //
////////////////////////////////////////////
procedure TfrmMain.OnCreate(Sender: TObject);
begin
   {загрузить словарь по умолчанию и запомнить его имя}
   filename:= DefaultFileName;
   loadfile;
   hod:= 0;
   OutHod(0, 0, 0);
end;
```

Название формы в данном примере **frmMain**. Для того чтобы создать пустую заготовку для процедуры-обработчика события *OnCreate* формы, щёлкните по его названию на вкладке *Events* в *Инспекторе объектов*. Затем наберите или скопируйте текст процедуры.

В разделе **interface** объявите *константы*:

```
const
   MAX_LEN_WORD=40;                          //!!! - макс. длина
слов в словаре !!!
   MAX_DICTIONARY=100000;                    //!!! - макс. число
слов в словаре !!!
   DefaultFileName='osh_frc_v.txt';          //словарь по умолча-
нию (путь к словарю и название файла)
   ABC_B='АБВГДЕЁЖЗИЙКЛМНОПРСТУФХЦЧШЩЪЫЬЭЮЯ'; //большие буквы рус-
ского алфавита
   ABC_M='абвгдеёжзийклмнопрстуфхцчшщъыьэюя'; //маленькие буквы
русского алфавита
   CONV_B=ABC_B + ABC_B;
   CONV_M=ABC_M + ABC_B;
```

и *глобальные переменные*:

```
var
  frmMain: TfrmMain;
  NumWords : array[0..MAX_LEN_WORD] of integer; // - число слов
словаре опр. длины
  NAME_PROG : PChar =' HangmanHelper';          // - название про-
граммы
  WordsAll : integer;                           // - всего считано
слов из файла словаря
  MaxLenWord : integer;                         // - максимальная
длина слов в файле
  Spisok : array [1 .. MAX_DICTIONARY] of string;// - массив сло-
варя
  BeginWord : array[0..MAX_LEN_WORD] of integer;// - начальный ин-
декс слов опр. длины в загруженном словаре
  FileName : string='';                         // - имя файла
словаря (= '', если ни один словарь не загружен)
  hod: Integer;
```

Работу с текстовыми файлами мы подробно изучим на следующем уроке, а сейчас нам важно знать, что при загрузке словаря в процедуре **loadfile** заполняется список слов **Spisok**, массив **NumWords** содержит число слов каждой длины, а массив **BeginWord** – начальные индексы в списке, с которых начинаются слова той или иной длины.

Теперь всё готово к игре. Обратите внимание, что сразу после запуска игры в поле шаблона 5 точек (свойство *Text* текстового поля **txtShablon** равно «**.....**»), поэтому если загаданное слово имеет другую длину, нажмите кнопку с соответствующим числом:

```
//
// ЗАПОЛНИТЬ ОКНО ШАБЛОНА СООТВ. ЧИСЛОМ ТОЧЕК
//
procedure TfrmMain.btnNumber4Click(Sender: TObject);
var s: string;
n: integer;
begin
  txtShablon.Clear;
  s:= '.............................';
  n:= (sender as Tcomponent).tag;
  txtShablon.Text:=copy(s,1, n);
  hod:= 0;
  OutHod(0, 0, 0);
end;
```

Так как все эти кнопки действуют одинаково, то нажатие любой их них обрабатывается в одной и той же процедуре (полюбопытствуйте, как это сделано!).

Итак, слово задумано, длина его обозначена точками в окне шаблона. Смело нажимайте на кнопку *Найти букву* - и тут начинается всё самое интересное: мы попадаем в главную процедуру программы, которая и выдаёт прогноз лучшего хода.

```
////////////////////////////////////////////////////////////
//                                                        //
//          НАЙТИ БУКВУ (СЛОВО) -  СТРАТЕГИЯ 3            //
//                                                        //
////////////////////////////////////////////////////////////

procedure TfrmMain.sbtFindLetterClick(Sender: TObject);
var
    s : String;
    i, j, n, m :  Integer;
    shablon: String;    //- шаблон слова
    letterminus, letterplus: String;
    lens: Integer;
    numWord, allWords: Integer;
    time: single;
    flg: boolean;
    //список подходящих слов:
    list: array [1 .. 30000] of string;
    //число букв в найденных словах (число слов, содержащих опр.
букву):
    numLetter: array [1..255] of integer;
    sTmp: String;
    MoveLetter: Integer;
    mm: array [1 .. 300] of integer;

begin
  randomize;
```

Вначале мы должны проверить, загружен ли словарь (мы загрузили его ещё при запуске программы, но проверка наличия словаря не повредит):

```
  if FileName='' then  {не загружен словарь!}
  begin
    application.MessageBox ('Загрузите словарь!', NAME_PROG,
MB_OK);
    exit
  end;
```

Проверяем шаблон слова (точки и отгаданные буквы). Его длина должна быть не менее двух символов и не превышать длину слов в словаре. Если обнаружится ошибка, то дальнейшие поиски загаданного слова будут бесполезны, о чём мы и должны предупредить игрока.

```
  //шаблон слова:
```

```
  shablon:= txtShablon.text;
  //конвертируем его в верхний регистр:
  shablon:= convertb(shablon);
  lens:= Length(shablon);  {его длина}
  //если оно длиннее всех слов в загруженном словаре,
  //то мы его не найдём:
  if lens > maxlenword then begin
    application.MessageBox ('Слишком длинный шаблон
слова!',NAME_PROG, MB_OK);
    exit;
  end;

  if shablon=''then
  begin
    application.MessageBox ('Вы не задали шаблон
слова!',NAME_PROG, MB_OK);
    exit;
  end;
  if length(shablon) < 2 then
  begin
    application.MessageBox ('Слишком короткий шаблон
слова!',NAME_PROG, MB_OK);
    exit;
  end;
```

Безусловно, проверка данных - не самое увлекательное занятие, поэтому вы можете её пропустить и потом долго отыскивать ошибки в правильно работающем алгоритме, который сбили с толку неверные исходные данные!

При поиске подходящих слов мы должны учитывать те буквы, которых в слове нет (это выясняется по результату хода), и те, которые есть (тоже). Строку с отсутствующими буквами мы просто копируем из текстового окна **letterminus**, а строку с угаданными буквами **letterplus** собираем из букв (точки пропускаем) в текстовом поле шаблона.

```
  //отсутствующие буквы:
  letterminus:= txtLetterMinus.Text;
  //конвертируем отсутствующие буквы в верхний регистр:
  letterminus:= convertb(letterminus);
  //и формируем строку из букв, которые уже открыты в шаблоне:
  letterplus:='';
  For n:= 1 To lens do
    if (shablon[n]<> '.') then letterplus:= letterplus + shab-
lon[n];
  letterplus:= convertb(letterplus);
```

Осталось по одному извлекать слова нужной длины из списка и определять, могут ли они быть искомыми или нет.

```
  //просматриваем в списке все слова той же длины, что искомое:
  For i:=beginword[lens] To beginword[lens]+numwords[lens]-1 do
    begin
      //очередное слово:
      s:= Spisok[i]; //s:= copy(Spisok[j],1,lens);
      //конвертируем его в верхний регистр:
      s:= convertb(s);

      //listbox1.Items.Add (s);
      //признак, по которому мы можем определить, что нашли (true)
или не нашли (false) подходящее слово
      flg:= TRUE;
```

Сразу отвергаем слова, в которых имеются отсутствующие буквы.

```
      //если в слове есть отсутствующие буквы, оно не подходит:
      For n:= 1 To lens do
      begin
        if pos(s[n], letterminus)> 0 then
        begin
          flg:= FALSE;
          //listbox1.Items.Add ('!!!!' + Spisok[i]);
          break;
        end
      end;
```

Если в шаблоне есть угаданные буквы, то точно такие же должны быть и в слове-кандидате, причём в тех же самых позициях. Если это не так, то такое слово заведомо не годится.

```
      //если в шаблоне открыта буква,
      //но она не совпадает с буквой в той же позиции очередного
слова,
      //то слово не подходит:
      For n:= 1 To lens do
      begin
        if  (shablon[n]<> '.') and (shablon[n]<> s[n]) then
        begin
          flg:= FALSE;
          //listbox1.Items.Add (Spisok[j]);
          break;
        end;
      end;
```

Труднее понять ещё одно условие: если в шаблоне стоит точка (буква неизвестна), а в очередном слове из списка там стоит уже угаданная буква, то такое слово не подходит. Подсказка: если в слове имеется угаданная буква, то и в шаблоне она должна находиться в той же позиции (либо вы забыли её «открыть», либо в загаданном слове там стоит совсем другая буква).

```
      //если в шаблоне не открыта буква,
      //но буква в той же позиции очередного слова есть среди обя-
зательных букв,
      //то слово не подходит:
      For n:= 1 To lens do
      begin
        if  (shablon[n]= '.') and (pos(s[n], letterplus)<>0)then
        begin
          flg:= FALSE;
          //listbox1.Items.Add (Spisok[i]);
          break;
        end;
      end;
```

Если слово выдержало все испытания, то оно может быть искомым, и мы его заносим в список **List** для дальнейшего использования и выводим на экран для всеобщего обозрения.

```
      //нашли подходящее слово:
      if flg then begin
         //увеличиваем счёткик подходящих слов:
         inc(numWord);
         //заносим его в список подходящих слов:
         list[numWord]:= Spisok[i];
         //и в протокол:
         lstProtokol.Items.Add (Spisok[i]);
          //lstProtokol.TopIndex:=lstProtokol.Items.Count-1
      end
  end;  // for

  //после проверки всего списка слов ыводим информацию в протокол:
  lstProtokol.Items.Add ('----------------------------');
  lstProtokol.Items.Add   ('Всего   найдено   слов:   '   +   int-
tostr(numword));
  lstProtokol.Items.Add ('============================');
```

Так как в переменной **numWord** мы подсчитывали подходящие слова, то можем сделать некоторые выводы. Так, если **numWord=0**, то ни одного подходящего слова не найдено и мы потерпели полное фиаско (то есть опростоволосились). А вот если **numWord=1**, то, наоборот, мы однозначно можем утверждать, что слово найдено и мы игру выиграли!

```
  //нет ни одного подходящего слова:
  if numWord = 0 then
  begin
    lstProtokol.Items.Add ('Подходящих слов нет!');
    lstProtokol.TopIndex:=lstProtokol.Items.Count-1;
    exit;
  end;
  //всего одно подходящее слово:
```

```
  if numWord = 1 then
  begin
    lstProtokol.Items.Add ('Слово отгадано: ' + list[1]);
    lstProtokol.TopIndex:=lstProtokol.Items.Count-1;
    exit;
  end;
```

При всех других значениях **numWord** нам придётся постараться. Сначала мы подсчитываем слова из списка **List**, в которых есть ещё не «называвшиеся» буквы, причём слова подсчитываем отдельно для каждой буквы, и результат заносим в массив *numLetter*. Затем мы просматриваем этот массив и находим букву, которая встречается в большем числе слов:

```
  n:= 0;
  allWords:= 0;
  for i:= 1 to 255 do
  begin
    if n < numLetter[i] then n:= numLetter[i];
    allWords:= allWords + numLetter[i];
  end;

  m:=0;
  for i:= 1 to length(ABC_B) do
    if numLetter[i] =n then begin
      lstProtokol.Items.Add     (ABC_B[i]+     '     '     +     int-
tostr(numLetter[i]));
      inc(m);
      mm[m]:= i;
    end;
```

Возможно, несколько букв покажут лучший результат, тогда случайным образом выбираем одну из них:

```
  //если имеется несколько лучших букв:
  if m > 1 then
  begin
    //то случайно выбираем одну из них:
    m:= random(m)+1;
    lstProtokol.Items.Add ('Ходим буквой: ' + ABC_B[mm[m]]);
  end;
```

Тут всё просто и понятно, но мы знаем, что иногда полезнее назвать не лучшую букву, а случайную. Мы уже знаем число всех подходящих слов для всех букв – **allWords** и легко можем выбрать любое из них:

```
i:= random(allWords)+1;
```

Осталось определить букву, для которой отбиралось это слово:

```
j:= 0; MoveLetter:=0;
 while j< i do  begin
   inc(MoveLetter);
   j:= j+ numLetter[MoveLetter]
 end;
```

На самом деле мы определили не саму букву, а её место в строковой константе

ABC_B='АБВГДЕЁЖЗИЙКЛМНОПРСТУФХЦЧШЩЪЫЬЭЮЯ';

Ну, а саму букву теперь определить проще простого: **ABC_B[MoveLetter]**.

И наконец, в окне протокола процедура выводит результаты всех подсчётов и рекомендуемые буквы – самую «лучшую» и случайную. Игрок выбирает ту, которая ему больше приглянулась, и «называет». Если буква угадана, она открывается в шаблоне, если нет, заносится в окно отсутствующих букв.

Всё – разработанный нами алгоритм выбора лучшего хода переведён на язык *Delphi*, а значит, программа практически написана. Оставшиеся «побочные» процедуры, облегчающие работу пользователя с программой, разберите самостоятельно.

Работа с программой на конкретном примере

Пусть, как и раньше, задумано слово **ЖЕЛЕЗО**. Запускаем программу. Все окна, кроме окна шаблона, пустые. В окне шаблона вы увидите 5 точек. Так как наше слово состоит из шести букв, то просто нажмите кнопку **6**, чтобы в окне шаблона появилась исходная позиция (Рис. С10.4).

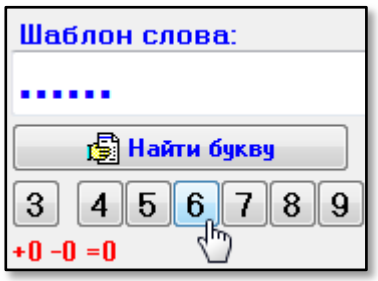


Рис. С10.4. Задумано слово из 6 букв

Нам уже известно, что лучший первый ход для 6-буквенных слов - буква **А**. Можете сразу же записать её в окно отсутствующих (вы же сами выбрали слово **ЖЕЛЕЗО**!), а можете нажать кнопку *Найти букву*. Тогда в окне протокола вы увидите такую информацию (Рис. С10.5, слева).

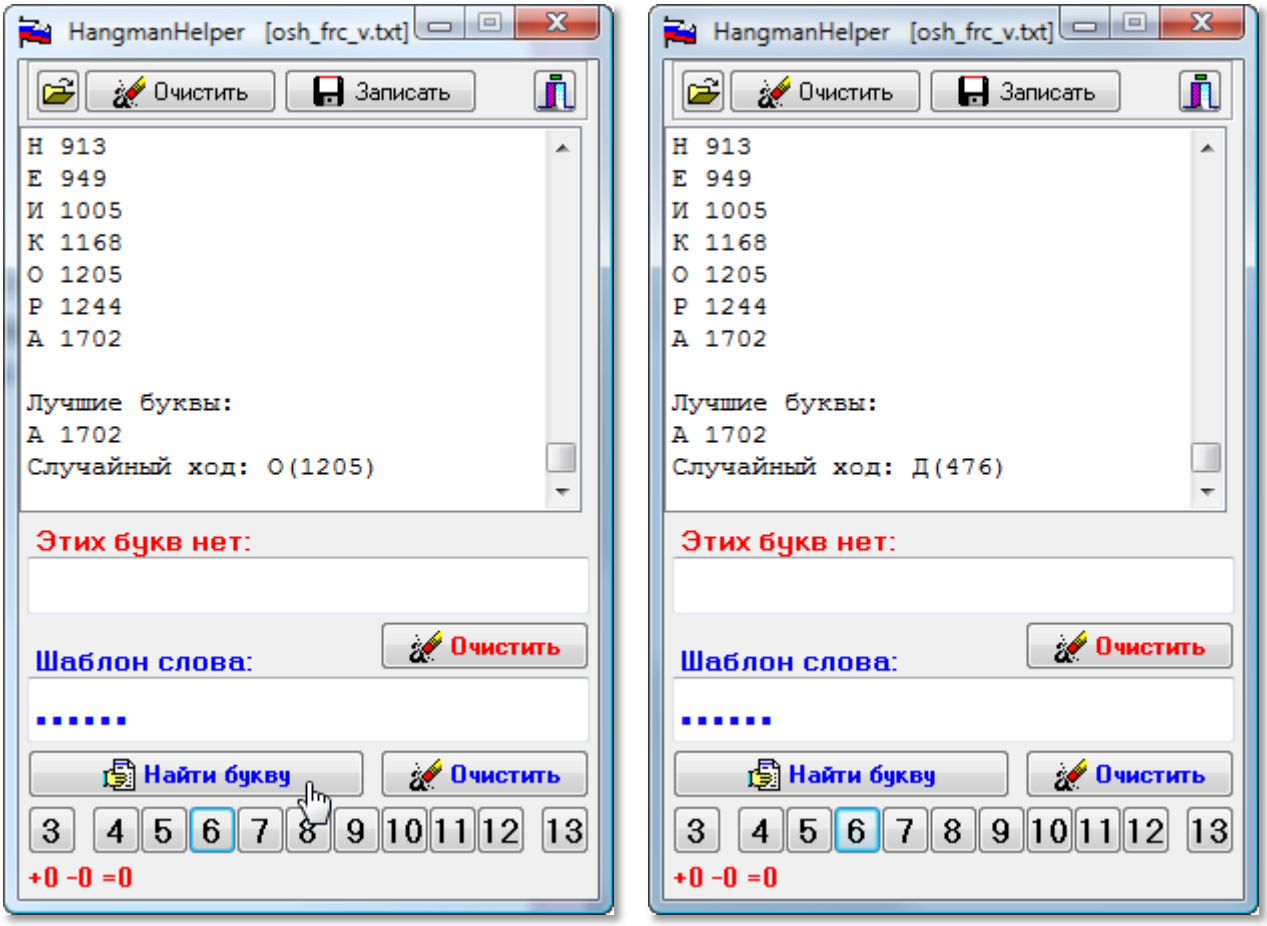


Рис. С10.5. Первый ход

То есть вам будет рекомендовано сходить именно так. Конечно, случайный ход может быть и другим (Рис. С10.5, справа).

Кроме того, в окне приведены данные по количеству слов с каждой буквой - вы можете выбрать букву и самостоятельно.

Итак, буквы **А** в слове не оказалось. Записываем её в отсутствующие и снова нажимаем кнопку *Найти букву* (Рис. С10.6).

В этот раз лучшей буквой оказывается буква **О**. Ходим этой буквой (но вы можете назвать и другую букву, естественно). Загадчик (в данном примере это тоже вы, если помните) отвечает, что эта буква в слове есть и стоит она на последнем месте (так как программа в общем случае не знает, какое слово задумано, она не может автоматически исправлять шаблон). Вы вписываете букву **О** в поле шаблона. Для этого устанавливаете курсор **перед** нужной позицией, выделяете её и щёлкаете левой кнопкой мыши и нажимаете на клавиатуре нужную букву (раскладка может быть и английской).

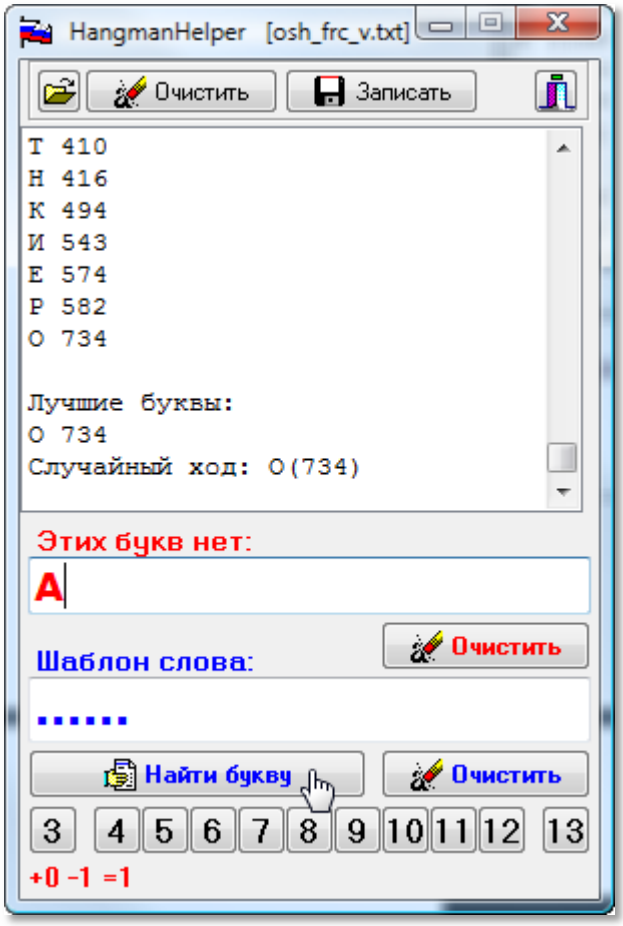


Рис. С10.6. Второй ход

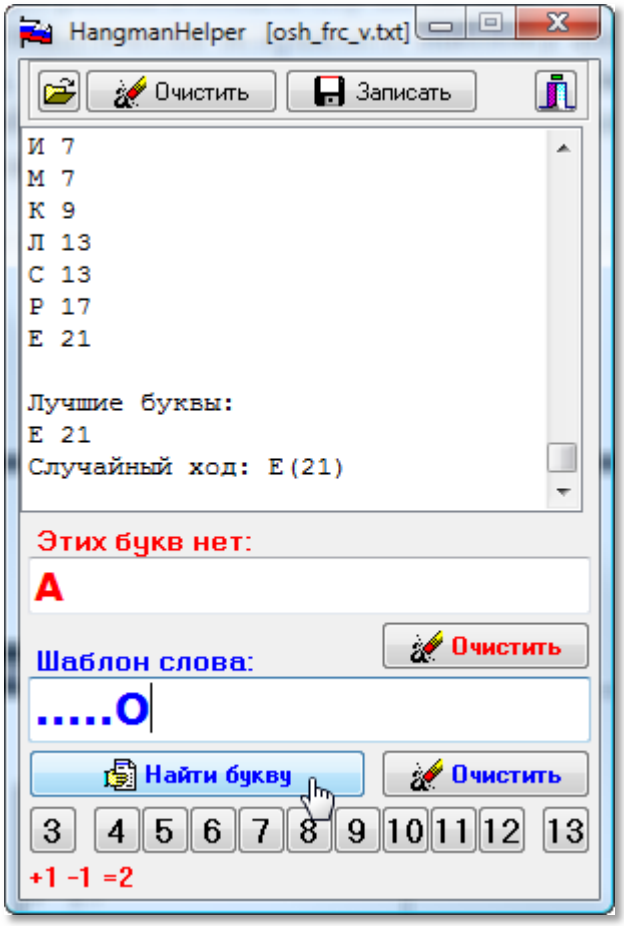


Рис. С10.7. Третий ход

Делаем следующий ход (Рис. С10.7).

Если вы выберете лучшую букву, то загадчик (это вы и есть) должен вписать её в шаблон. Затем вы делаете очередной ход (Рис. С10.8).

Случайный ход на этот раз лучше, но мы этого не знаем и ходим буквой **Р**, которой в слове нет. Кстати, подходящих слов осталось уже совсем немного (Рис. С10.9).

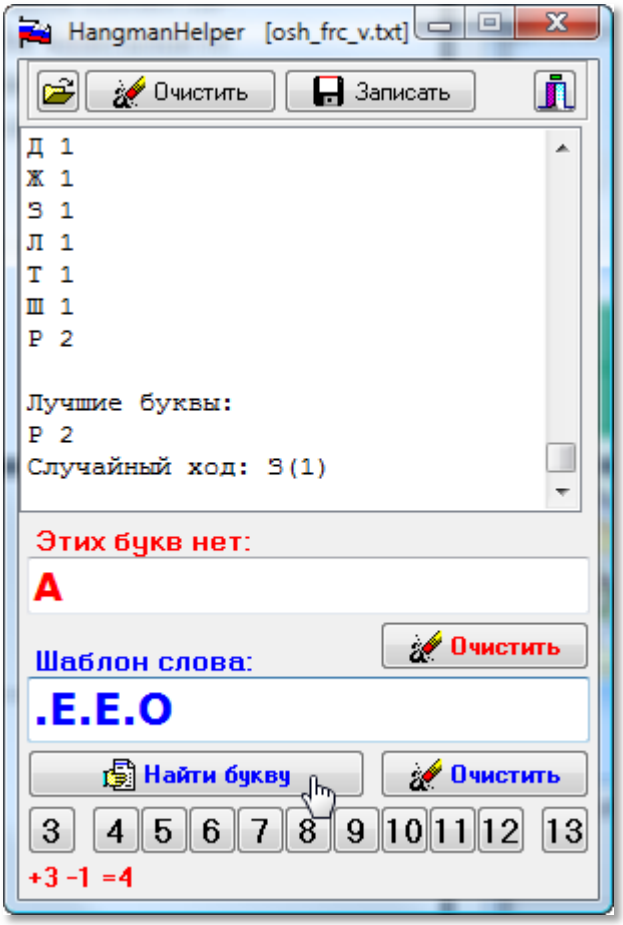


Рис. С10.8. Четвёртый ход

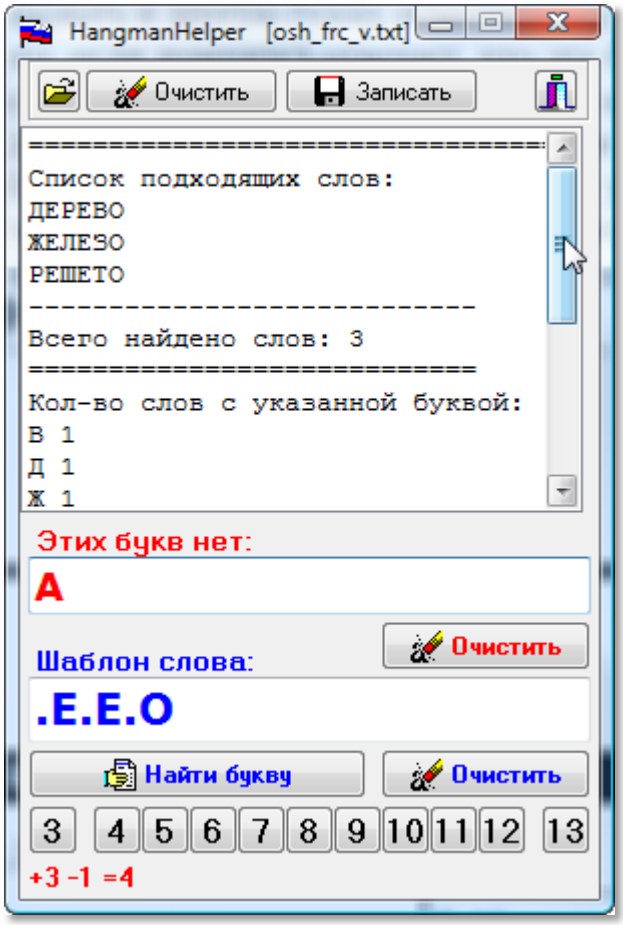


Рис. С10.9. Ряды редеют!

Если у вас есть какие-то догадки насчёт загаданного слова, вы можете сами выбрать его из списка и назвать букву из него. В данном же случае только буква **Р** имеется в двух словах, поэтому она и была выбрана для хода.

Ходим ещё раз (Рис. С10.10).

Слово отгадано - это **ЖЕЛЕЗО**. Последовательно называете буквы **Ж**, **Л, З** в любом порядке, и вы - победитель. Как видите, мы сделали всего один неверный ход.

Игра закончена. Для следующей игры вы стираете поле отсутствующих букв и так далее.

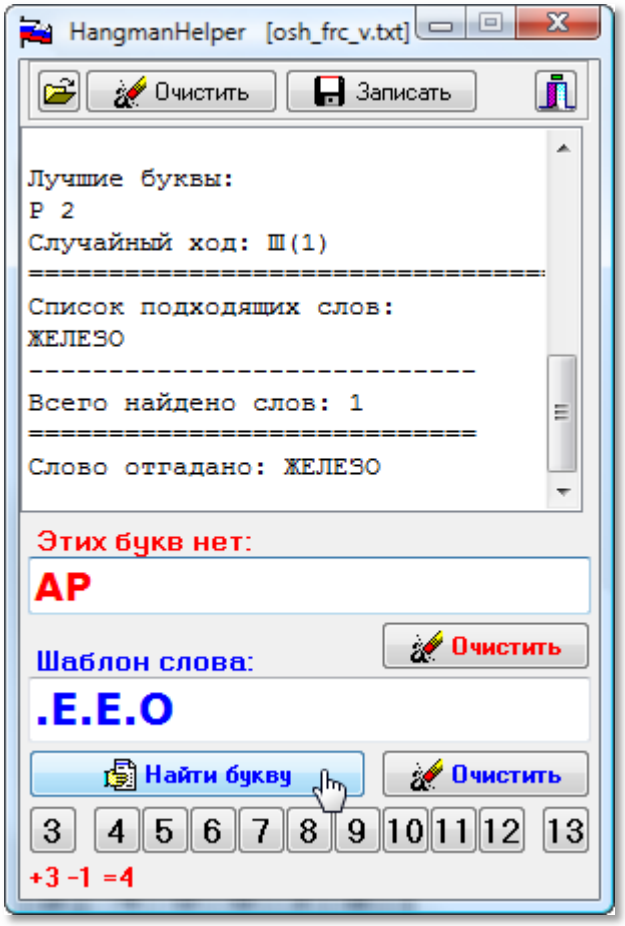


Рис. С10.10. Отгадали слово!



Замечание 13. В папке с программой вы найдёте ещё 2 дополнительных словаря. Попробуйте воспользоваться ими, если вам придётся разгадывать трудные, редкие слова. Не забывайте также пополнять свой словарный запас, он пригодится вам и в других играх! Все слова в словарях должны быть рассортированы по длине. На следующем семинаре мы напишем утилиту **Fraction.exe**, с помощью которой вы сможете переписать любой словарь (исходный файл не будет изменён!) так, чтобы его можно было использовать с программой **HangmanHelper**. Регистр букв значения не имеет, но лучше использовать заглавные буквы, так как их легче различать на экране.

В словаре **makarov_frc.txt** буква Ё везде заменена буквой Е, имейте это в виду.

Замечание 14. Эту программу легко приспособить и для отгадывания слов в кроссворде. Например, нужно вписать 6-буквенное сло-

во по горизонтали, в котором две буквы **РР** (..РР..) в середине слова. Вы, наверное, всё ещё роетесь в памяти, а я уже нашёл – это слово **ПЕРРОН**! Конечно, в кроссвордах загадывают гораздо больше слов, чем их имеется в словаре Ожегова, но эти претензии следует адресовать именно словарю, а не нашей программе.



Исходный код программы находится в папке **Hangman**.

Самостоятельная работа



Измените программу **HangmanHelper** так, чтобы она играла с пользователем в *Поле чудес*.

Программа должна загружать список слов (словарь), что уже нами сделано; выбирать из его произвольное слово и выводить в текстовое поле шаблона (его имя следует изменить в соответствии с новым назначением) столько точек, сколько букв в загаданном слове. Далее игрок вводит по одной букве, программа сообщает ему голосом Леонида Якубовича, есть ли такая буква в слове или нет, а очаровательная помощница открывает в слове отгаданные буквы. Желательно заготовить для ведущего несколько удачных комментариев к вашим ходам, чтобы компьютерный вариант игры ничем не уступал телевизионному оригиналу.

Единственный недостаток такой игры – необходимость подносить подарки ведущему, передавать приветы родственникам и петь народные песни дурным голосом.

Дополнительный материал

Всегда лучше учиться на чужих ошибках, поэтому в папке с исходным кодом программы *HangmanHelper* вы найдёте несколько готовых игр, которые могут пригодиться вам при работе над собственной программой.

Программы для игры в "Виселицу"

1. Виселица, версия 1.2. Автор Дм. Мехов, 2001. Неплохая игра для тренировки. Позволяет делать множество ошибок, что очень полезно для

начинающих, иначе они всегда будут проигрывать, а это навсегда отобьёт у них охоту играть в словесные игры (Рис. С10.11).



Рис. С10.11. *Виселица* Дм. Мехова

2. Виселица. Автор неизвестен. Напоминает предыдущую программу, но сделана значительно хуже. Загадывает только 4-5-буквенные слова (Рис. С10.12).



Рис. С10.12. Натуралистическая виселица

3. Виселица, версия 1.0. Автор Георгий Макаров. Игра для начинающих. Имеет небольшой словарь самых распространённых слов. В отличие от классического варианта сразу открывает одну букву, которая может находиться в любом месте слова. Позволяет выбрать один из трёх уровней сложности, которые определяются числом букв в загаданном слове. Более интересна версия 1.4, но она зависает в самом начале игры (Рис. С10.13).

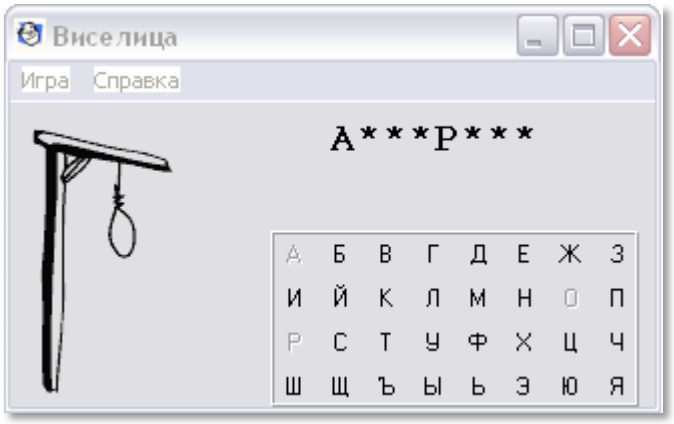


Рис. С10.13. *Виселица* Георгия Макарова

4. Виселица, версия 1.0. Автор ADV Soft, 2001. Программа имеет большой словарь, так что нередко загадывает трудные для отгадывания слова (советую загружать в *HangmanHelper* словарь **SSSRLfrc.txt** (или **makarov_frc.txt**), хотя и в нём есть не все слова, что вам может предложить эта программа. Например, вы знаете, что такое БУЛАНЖЕРИТ или ФОСФОРИЛИРОВАНИЕ? Вероятно, словарь взят со стороны, и вообще программа какая-то недоделанная). Правда, некоторые слова имеют "тематическую" подсказку, а также сразу открываются первая и последние буквы. Эту игру можно рекомендовать только ну очень крутым эрудитам... (Рис. С10.14).

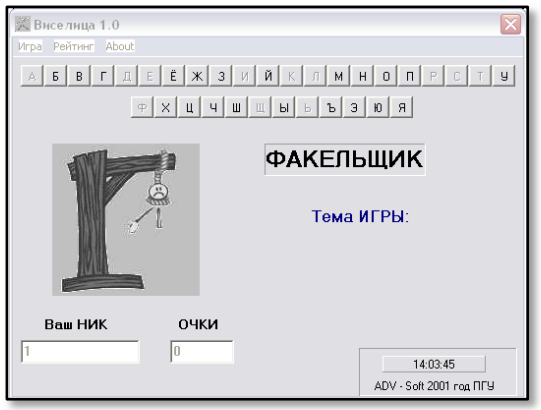


Рис. С10.14. *Виселица* от ADV Soft

5. "Hang Stan". Флэш-игра на английском языке. Отгадать нужно "фразу", тема которой всегда пишется над заданием. На отгадывание даётся 4 попытки и 8 минут времени. Отгадать фразу не очень трудно, если знать английский язык и культуру (Рис. С10.15).



Рис. С10.15. Английская виселица



Программы находятся в папке **Hangman/progs**.

Интернет-игра в *Виселицу*

Естественно, перед игрой вам нужно зарегистрироваться, взять кличку, выбрать пароль и проч.

1. Виселица - games.germany.ru. Здесь (Рис. С10.16) можно почитать правила (которые к самой игре отношения не имеют), посмотреть, кто сейчас играет, полюбоваться таблицей лучших результатов (наверх вы поднимитесь только через несколько лет, крепитесь).

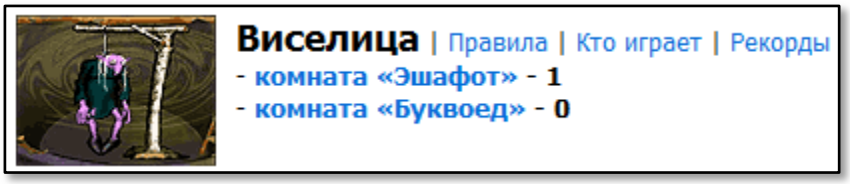


Рис. С10.16. Интернет-виселица

Получив эту не очень полезную информацию, щёлкните по надписи **комната "Эшафот"** (справа указано число игроков в сети), загрузится *Java*-апплет, и вы можете приступать к игре.

Для начала выберите невинный персонаж, которого будут вешать вместо вас. Я предпочитаю старушку, она страшная, её и повесить не жалко (Рис. С10.17).



Рис. С10.17. Выбираем персонаж

Для вас сразу же вроют столб и звёздочками покажут, сколько букв в загаданном слове. Как играть, мы уже подробно разобрали. С программой **HangmanHelper** проблем у вас не будет, так что набранные очки - только дело времени (правда, система начисления очков на сайте весьма причудлива и достоверно не известна никому; говорят, что количество очков зависит от сложности слова, но кем и чем она измеряется?). Зато игра имеет мини-чат (справа от игрового поля), в котором можно обсудить наболевшее, а также узнать, кто и сколько набрал очков.

Игра ведётся по классическим правилам, только число неверных попыток всего 7, что, конечно, маловато при отгадывании коротких слов.

Ваши набранные очки сохраняются, так что вы всегда можете продолжить своё восхождение к вершинам славы. Заодно вы получите за активность некие билеты и сможете участвовать в розыгрыше призов (или просто в розыгрыше).

HangmanHelper решает задачу (Рис. С10.18)

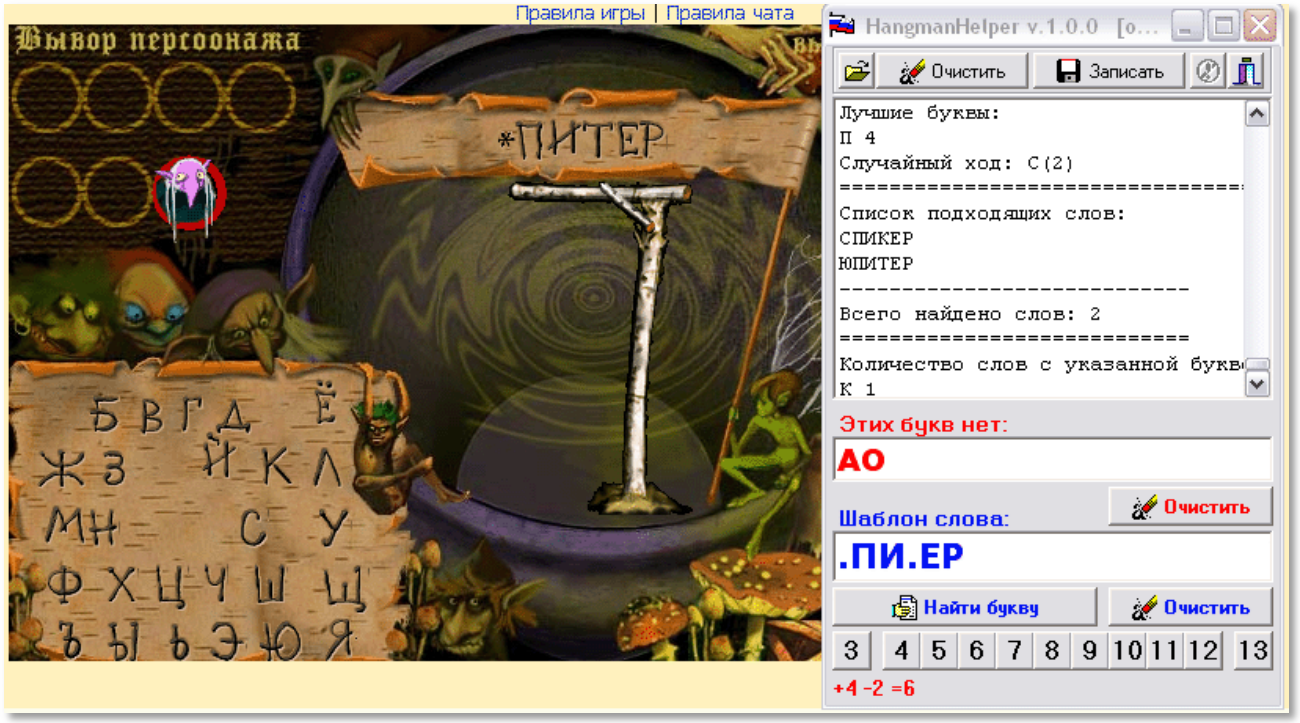


Рис. С10.18. С таким помощником нас не повесят!

Урок 11. Текстовые файлы, или FILEгранная работа со строками

А ты пиши мне письма мелким почерком,
поскольку места мало в рюкзаке.

Фильменная песня

Романтическое создание писем при дрожащем пламени свечи, увы, безвозвратно ушло в прошлое. И хотя некоторые креативные приверженцы эпистолярного стиля упорствуют в невежестве и доверяют свои невнятные мысли бумаге, последняя сейчас в основном используется по своему прямому назначению, то есть для складывания фигурок оригами и издания плакатов, призывающих беречь зелёного друга (коим для нас является лес, а не крокодил Гена и не пьяный товарищ).

Все наши винчестеры, флешки, диски CD- и DVD-ROM доверху забиты файлами. Каждый из них имеет имя и - почти всегда – расширение, которое следует за последней точкой в имени файла и обычно указывает на назначение файла и формат данных, хранящихся в нём.

Систематика файлов насчитывает многие сотни типов файлов: **исполняемых** *exe*, **видео** *avi*, *mpg*, **звуковых** *ogg*, *wav*, **музыкальных** *mp3*, *mid*, но, с точки зрения *Delphi*, разновидностей файлов всего 3: **текстовые** (они имеют тип **TextFile**), **типизированные** и **нетипизированные**.

На этом занятии мы рассмотрим **текстовые файлы**, которые состоят из строк, подобных тем, что мы изучили на прошлом уроке. Как в текстовых, так и в любых других файлах размещаются данные, которые можно сколь угодно долго хранить на диске даже после окончания работы с приложением, в котором эти данные были созданы, а потом снова загрузить в него.

Наберите в самом простом текстовом редакторе любой сумбурный текст, который только придёт вам в голову, - его глубокий смысл только отвлечёт нас от формы (Рис. У11.1).

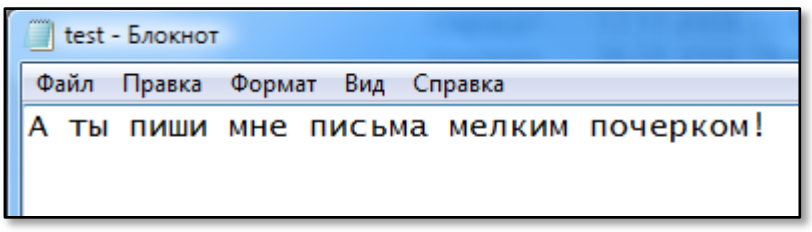


Рис. У11.1. Письмо в *Блокноте*

Очень хорош для разбора текстовых файлов стандартный *Блокнот*, который не добавляет в файл никакой «отсебятины», в отличие от «продвинутых» редакторов типа *MS Word*.

Не забудьте после набора текста нажать клавишу *Return*, чтобы завершить строку. Сохраните текст в файле под каким-нибудь именем – вот и готов настоящий текстовый файл! Обратите внимание, что к его названию автоматически добавлено расширение **.txt**. Теперь вы можете открыть файл в любом редакторе и увидеть вашу строку в первозданном виде (естественно, все ваши труды по форматированию текста пойдут прахом, потому что никакая информация о ваших шрифтовых предпочтениях не сохраняется!).

Когда мы изучали строки, то узнали, что разные типы строк по-разному хранятся в памяти компьютера. И вас, наверное, удивит тот факт, что на диске любые строки сохраняются одинаково, но не совсем так, как в ячейках памяти.

Если мы загрузим текст в редактирующую программу, то он будет разбит на строки так, как мы и его набирали. Но как редактор определяет конец строки? Может быть, вместе с каждой строкой записывается и её длина или после строки записывается нулевой символ? Второй вариант ближе к истине, но всё равно ошибочен.

В обычном текстовом редакторе мы никогда не увидим, что же там, в конце строки, поэтому воспользуемся таким текстовым редактором, который позволяет просматривать текст в 16-ричном виде, например, подойдёт **WinHex** (Рис. У11.2).

Offset	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	
00000000	C0	20	F2	FB	20	EF	E8	F8	E8	20	EC	ED	E5	20	EF	E8	А ты пиши мне пи
00000010	F1	FC	EC	E0	20	EC	E5	EB	EA	E8	EC	20	EF	EE	F7	E5	сьма мелким поче
00000020	F0	EA	EE	EC	21	0D	0A										рком!..

Рис. У11.2. 16-ричное письмо

И вот мы видим: все символы строки записаны без всяких изменений, но в конце строки находятся сразу 2 символа - **$0D** и **$0A** (или то же самое в более привычном десятичном виде - **#13** и **#10**). Они ещё в стародавние времена обозначали возврат каретки принтера и переход на новую строку. Этот анахронизм сохранился до сих пор в текстовых файлах, которые ничего и не знают о принтерах.

Нам осталось узнать, как же записываются строки таким причудливым образом, поэтому рассмотрим **основные операции с текстовыми файлами** – сохранение их на диске и загрузку в приложение. Они не доставят вам хлопот, если вы будете точно следовать «рецепту».

1. Независимо от того, сохраняем мы файл или загружаем его, первым делом нужно **объявить файловую переменную** типа **TextFile**:

```
var F: TextFile;
```

Её принято обозначать буквой **F** (от слова *File*, файл), но это необязательно. Именно эта переменная и будет псевдонимом реального файла.

2. Затем нужно **связать файловую переменную с именем файла**, воспользовавшись процедурой

```
procedure AssignFile(var F: file; FileName: string);
```

Её легче понять на конкретном *примере*.

Вот так можно связать нашу файловую переменную с файлом «osh_frc_v.txt» (так как путь к файлу не указан, то файл должен находиться в той же папке, что и само приложение!).

```
AssignFile(F, 'osh_frc_v.txt');
```

или

```
var s: string;

s:= 'osh_frc_v.txt'; // строка s содержит имя файла
AssignFile(F, s);
```

3. Теперь можно **открыть файл** для чтения или записи.

Для **считывания** данных указанный файл должен существовать на диске. Чтобы открыть его, пользуемся процедурой

```
procedure Reset(var F: file);
```
(в нашем примере - **Reset(F);**) ,

и можно загружать данные. Они будут считываться, начиная с первой строки.



Если файл уже был ранее открыт, то он сначала автоматически закрывается, чтобы не потерять данные, а потом вновь открывается.

Чтобы **сохранить** данные в файле, воспользуемся процедурой

`procedure Rewrite(var F: file);` (в нашем примере - **Rewrite(F);**)

Она создаст **новый файл** с заданным именем и откроет его для записи. Будьте осторожны: если файл с таким именем уже имеется на диске, он будет уничтожен! Поэтому безопаснее записывать данные в существующий файл функцией

`function Append(var F: Text): Integer;` (в нашем примере - **Append(F);**)

В этом случае новые строки будут добавлены в файл за последней строкой, так что все прежние данные не пострадают.

4. Для **записи данных** в файл служат процедуры

```
procedure Write(var F: file [; P1; ...; Pn]);
procedure Writeln(var F: file [; P1; ...; Pn]);
```

В квадратных скобках записаны необязательные параметры, которые являются выражениями следующих типов: *char, string, PChar, integer, real, boolean*. Таким образом, в текстовый файл можно записывать не только символы, слова, строки, но и числа (при записи выражений логического типа в файл выводятся слова *TRUE* и *FALSE*). Количество выражений в списке вывода может быть любым.

Если в одной строке файла процедурой **Write** сохраняются данные разных типов, то они будут записаны друг за другом без пробелов и других разделителей. Процедура **Writeln** делает то же самое, но затем закрывает строку в файле (добавляет в конец строки символы **$0D$0A**).

Например, процедура

```
Writeln(F, 'А ты пиши мне письма мелким почерком!');
```

Запишет строку в кавычках (но без кавычек!). Или то же самое:

```
var s: string;
s:= 'А ты пиши мне письма мелким почерком!';
Writeln(F, s);
```

Или

```
var s1,s2,s3,s4,s5,s6,s7: string;
s1:= 'А ';
s2:= 'ты ';
s3:= 'пиши ';
```

```
s4:= 'мне ';
s5:= 'письма ';
s6:= 'мелким ';
s7:= 'почерком!';

Writeln(F, s1,s2,s3,s4,s5,s6,s7);
```

Или

```
Writeln(F, 'А ', 'ты ', 'пиши ', 'мне ', 'письма ', 'мелким ',
'почерком!');
```

Или

```
Writeln(F, 'А ' + 'ты ' + 'пиши ' + 'мне ' + 'письма ' + 'мелким '
+ 'почерком!');
```

Вы также можете смешивать константы и переменные:

```
Writeln(F, s1 + 'ты ' + s3 + 'мне ' + s5 + 'мелким ' + s7);
```

Если у вас есть к тому большая нужда, вы можете записать в строку числа и логические константы:

```
Writeln(F, s1 + 'ты ' + s3 + 'мне ' + s5 + 'мелким ' + s7, ' ',
123, ' ', 321, ' ', 123 < 321);
```

Тогда на диске появится строка:

А ты пиши мне письма мелким почерком! 123 321 TRUE

Все числа при записи в файл должны отделяться друг от друга, по крайней мере, одним пробелом, знаком табуляции или кодами конца строки.



Процедура только с одним параметром - **Writeln(F)** - просто завершает строку, ничего в неё не записывая.

Процедуры **Writeln** и **Write** начинают записывать данные с первой строки файла, затем **Writeln** (!) последовательно переходит ко второй строке, третьей – и так далее до конца файла. С их помощью нельзя сохранить данные в произвольной строке файла, но можно записать нужное число пустых строк процедурой **Writeln(F)**, а уже затем записать нужную строку.

Для **чтения данных** из текстового файла имеются процедуры:

```
procedure Read(var F: file [; V1; ...; Vn]);
```

```
procedure Readln(var F: file[; V1; ...; Vn]);
```

В квадратных скобках записаны необязательные параметры, которые являются переменными следующих типов: **char**, **string**, **integer**, **real**, **PChar**. Количество переменных в списке ввода может быть любым.

Мы чаще будем использовать вторую процедуру (**Readln**) – чтобы считать целиком всю строку в указанную строковую переменную:

```
var
  s: String;

Readln(F, S);
```

Если в одной строке записаны данные других типов через пробел, например, «**123 321 TRUE**», то их можно считать так:

```
var
  s : String;
  n,m:integer;

Readln(F, n, m, s);
```

В результате выполнения этой операции значением целой переменной **n** будет число 123, целой переменной **m** – число 321, строковой переменной **s** – строка 'TRUE' (нельзя считать значение в логическую переменную!). Не забывайте следить за соответствием типов переменных и считываемых данных, иначе возникнет ошибка!



Если процедура **Readln** считывает только часть данных из строки (в списке ввода меньше переменных, чем в строке), то оставшаяся часть строки будет пропущена.

Если переменные при вызове процедуры отсутствуют - **Readln(F);**, - то текущая строка в файле пропускается и далее считываются данные из следующей строки.

Если строка считывается в переменную типа короткой строки или **PChar** и имеет больше символов, чем указанная максимальная длина строки, то вводимая строка усекается.

При вводе символа из файла в переменную типа **char** читается только 1 символ, в том числе и коды конца строки.

Процедуры **Read** и **Readln** начинают считывать данные с первой строки файла, затем **Readln** (!) последовательно переходит ко второй строке, третьей – и так далее до конца файла. С их помощью нельзя считать данные из произвольной строки файла, но можно пропустить нужное число строк процедурой **Readln(F)**, а уже затем считать нужную строку.

5. После завершения операций с файлом он должен быть **закрыт** процедурой:

`procedure CloseFile(var F: file);` (в нашем примере - **CloseFile(F);**)

Никогда не забывайте одну из заповедей программирования: *Уходя, гасите свет!*

Пифагоровы штаны

Файл не бумага – не всё стерпит!

Программист Козьма Прутков

Жизнь – сложная штука!

Остап Бендер

Однажды Пифагор на упрёк портного, что вот он великий учёный, а не может себе сшить банальных штанов, воскликнул: «Теоретически-то я могу, но практически – я без штанов!» Да, в реальных программах не всё так гладко, как описано в **пунктах 1-5** нашего «рецепта», и при файловых операциях могут возникать ошибки. Например, отсутствует указанный файл, произошла ошибка при его открытии, сбой при считывании или записи данных. Это вполне может быть и это бывает. Так вот, при работе с файлами программа автоматически отслеживает подобные ошибки и в случае возникновения оной так же автоматически «вырубается».

Такое «порочное» поведение программы обусловлено директивой компилятора **{$I+}**, которая включена по умолчанию. Если же вы хотите единолично управлять программой при файловых операциях, то в критических местах должны отключить автоматическую проверку директивой **{$I-}** и самостоятельно анализировать результат выполнения операций ввода-вывода с помощью функции **IOResult**. Она возвращает **0**, если операция завершилась успешно, или любое другое число в противном случае. Не за-

будьте потом снова включить автоматическую проверку директивой **{$I+}**!

Функции для работы с файлами

При считывании данных из файла необходимо следить, не закончился ли файл (строка), иначе при попытке загрузить данные после конца файла возникнет ошибка. В слежке нам помогут функции:

`function Eof(var F: file): Boolean;` (в нашем примере - **eof(F);**)
`function SeekEof(var F: Text): Boolean;` (в нашем примере - **SeekEof(F);**)



Eof – сокращение от «end of file».

Если функция **Eof (**или **SeekEof**) возвращает *TRUE*, то достигнут конец файла и данные считываются дальше. Функция **SeekEof** отличается от **Eof** только тем, что пропускает стоящие в конце файла пробелы и знаки табуляции.

Пример использования функций:

```
If eof(f) then … //Если файл закончен, то …
Else read(f,s);  //Если не закончен, то считать данные в перемен-
ную s
```

Впрочем, чаще используют условие, что файл ещё **не закончен**:

```
if not eof(f) then read(f,s);
```

Или

```
while not eof(f) do begin
  Readln(F, S);
…
end;
CloseFile(F);
```

Или

```
Repeat
  Readln(F, S);
  …
Until SeekEof(f);
CloseFile(F);
```

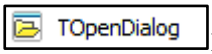
Для определения **конца строки** имеются очень похожие функции:

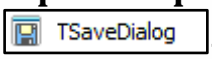
```
function Eoln(var F: Text):
function SeekEoln(var F: Text): Boolean;
```

Функция **Eoln** возвращает *TRUE*, если достигнут конец строки или конец файла (**Eof =** *TRUE*). Функция **SeekEoln** отличается от **Eoln** только тем, что пропускает стоящие в конце строки пробелы и знаки табуляции.

Компоненты для работы с файлами

Рассмотренный нами в пункте втором пример использования процедуры **AssignFile(var F: file; FileName: string)** имеет очевидный недостаток: в строке *FileName* должно быть заранее записано имя файла. А что делать, если пользователь захочет открыть или создать свой собственный файл, имени которого мы не знаем? Конечно, можно разместить на форме компонент **текстовое поле** 

, в которое пользователь и введёт с клавиатуры имя нужного ему файла, но зачем же так обременять человека? В *Delphi* имеется и более гуманный способ: достаточно заменить текстовое поле **диалоговым окном открытия файлов** 

или **диалоговое окно сохранения файлов** 

- и работа с файлами станет простой и удобной, поскольку пользователь сможет выбирать файлы привычным для него способом.

На *Семинаре* мы воспользуемся именно этим способом выбора файлов.

Семинар 11. Фракционирование, или Слова мелкого и крупного помола

Во многих случаях поиск нужных слов ведётся среди слов одной и той же длины. Например, в игре *Виселица*, при составлении списка анаграмм, метаграмм, палиндромов, при разгадывании кроссвордов число букв в словах известно, поэтому бывает очень полезно иметь список, в котором слова расположены не только по алфавиту, но и по длине слов.

Если у нас есть под рукой обычный словарь, то нажатием всего одной кнопки мы его превратим во «фракционированный», то есть «просеем» все слова через сито с разными по величине ячейками. В итоге получится вот такой полезный списочек:

АД
АЗ
АР
АС
ГО
ЁЖ
ИЛ
ОМ
ОР
ПА
РЭ
СУ
УЖ
УМ
УС
ЩИ
ЮГ
ЮЗ
ЮР
ЮС
ЯД
ЯК
ЯЛ
ЯМ
ЯР
АЗУ
АКР
АКТ
АУЛ
. . .

СВЕТОЧУВСТВИТЕЛЬНОСТЬ
ЕСТЕСТВОИСПЫТАТЕЛЬНИЦА
НЕУДОВЛЕТВОРИТЕЛЬНОСТЬ
ПЕРЕОСВИДЕТЕЛЬСТВОВАНИЕ
ЧЕЛОВЕКОНЕНАВИСТНИЧЕСТВО

Интерфейс программы ещё более скудный и лаконичный, чем у **HangmanHelper** (Рис. С11.1).

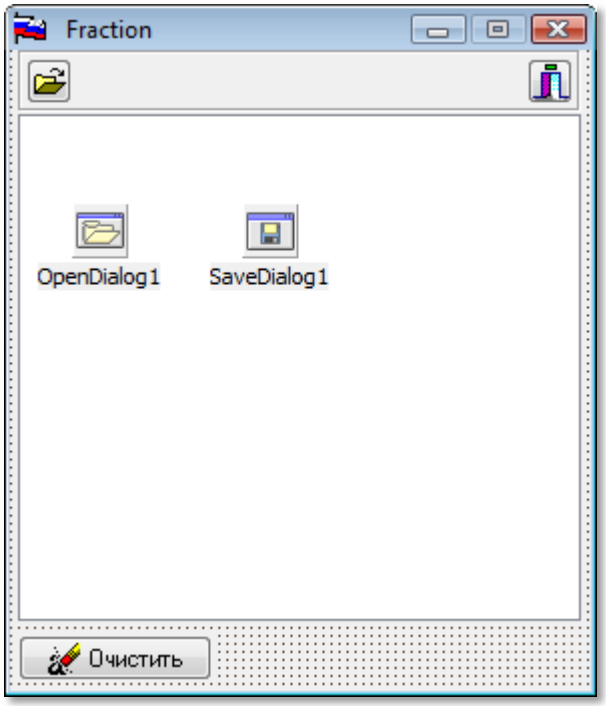


Рис. С11.1. Фракционный интерфейс

Обратите внимание на значки компонентов **OpenDialog** и **SaveDialog**! Первый даёт пользователю возможность выбрать файл словаря на диске для его загрузки, второй – для записи нового, фракционированного словаря.

На экран выводим только скупую информацию о загруженном и о записанном словарях. Выбрать же нужный файл со словарём на диске легко с помощью кнопки **sbtOpenFile**, которая приветливо встречает пользователя в верхнем левом углу формы.

```
// Загрузить словарь по выбору
procedure TfrmMain.sbtOpenFileClick(Sender: TObject);
begin
  opendialog1.DefaultExt:='txt';
  opendialog1.Filter:='Text files (*.txt)|*.TXT';
  opendialog1.InitialDir:= extractfilepath(application.exename);
  opendialog1.Title:='Загрузите новый словарь';
  if opendialog1.Execute then
  begin
    filename:= opendialog1.filename;
    loadfile;
  end
  else
```

```
    application.MessageBox ('Вы не загрузили словарь!',NAME_PROG,
MB_OK);
end;
```

В процедуре **loadfile** мы совместим приятное с полезным: и готовый словарь с диска загрузим, и новый создадим:

```
////////////////////////////////////////////
//                                        //
//          ЗАГРУЗИТЬ СЛОВАРЬ             //
//                                        //
////////////////////////////////////////////
procedure TfrmMain.LoadFile;
//имя файла хранится в переменной FileName
var
  s : String;
  F: TextFile;
  i,j: integer;

begin
  {обнуляем массив количества слов равной длины}
  For i := 0 To MAX_LEN_WORD do NumWords [i] := 0;

  //Открывем файл для чтения:
  {$i-}
    AssignFile(F,FileName);
    Reset(F);
  {$i+}
  if IOResult<>0 then    {ошибка при открытии файла}
  begin
    filename:='';     {словарь не загружен!}
    application.MessageBox   ('Словарь   не   загружен!',NAME_PROG,
MB_OK);
  end
  else                   {всё нормально}
  begin
    WordsAll:= 0;    //всего слов загружено
    BukvAll:= 0;     //всего букв в словах
    MaxLenWord:= 0;
    //считываем слова из файла:
    while not eof(f) do begin
      inc(wordsall);
      Readln(F, S);
      Spisok[WordsAll] := s;
      BukvAll:= BukvAll + length(s);
      NumWords[Length(s)]:= NumWords[Length(s)] + 1;
      If Length(s) > MaxLenWord Then MaxLenWord:= Length(s)
    end;
    CloseFile(F);

    frmMain.caption:= NAME_PROG + '   [' + filename + ']';
    lstProtokol.Items.Add ('Загружен словарь: ');
```

```
    lstProtokol.Items.Add (ExtractFileName(FileName));
    lstProtokol.Items.Add ('');

    lstProtokol.Items.Add ('Количество слов равной длины:');
    lstProtokol.Items.Add ('');
    For i:= 1 To MaxLenWord do
      lstProtokol.Items.Add (inttostr (i) + ' = ' + inttostr( Num-
Words[i] ));
    lstProtokol.Items.Add ('');
    lstProtokol.Items.Add ('Всего слов: ' + inttostr(wordsall));
    lstProtokol.Items.Add ('Всего букв: ' + inttostr(BukvAll));
    lstProtokol.Items.Add   ('Средняя   длина   слов:   '   +   co-
py(floattostr(BukvAll / WordsAll),1,4));

    //записать на диск фракционный словарь:
    s:= ExtractFileName(FileName);
    s:= copy(s,1,length(s)-4) + '_frc.txt';
    savedialog1.FileName:= s;
    if not savedialog1.Execute then exit;

    assignfile(f,savedialog1.filename);
    rewrite(f);

    for j:=1 to MaxLenWord do
      for i:=1 to WordsAll do
        if length(spisok[i])= j then begin
           writeln (f, spisok[i]);
        end;

    Closefile(f);
    Messagebeep(0);
    lstProtokol.Items.Add ('');
    lstProtokol.Items.Add ('======================');
    lstProtokol.Items.Add ('Записан словарь: ');
    lstProtokol.Items.Add (ExtractFileName(savedialog1.filename));
    lstProtokol.Items.Add ('');
  end;
end;
```

Все слова из файла мы считываем в массив строк **Spisok**, при этом находим число букв в самом длинном слове – оно хранится в переменной **MaxLenWord**. И остаётся только в цикле `for j:=1 to MaxLenWord do` просматривать все слова в массиве **Spisok** и последовательно записывать в файл слова длиной от 1 до максимальной, то есть весь алгоритм уместился в несколько строк кода (Рис. С11.2):

```
    for j:=1 to MaxLenWord do
      for i:=1 to WordsAll do
        if length(spisok[i])= j then begin
           writeln (f, spisok[i]);
        end;
```

А не подумавши, можно было бы сначала отсортировать массив **Spisok** по длине слов, и только потом записывать новый словарь в файл. Конечно, компьютеру это сделать несложно, но нам-то зачем делать лишнюю работу!

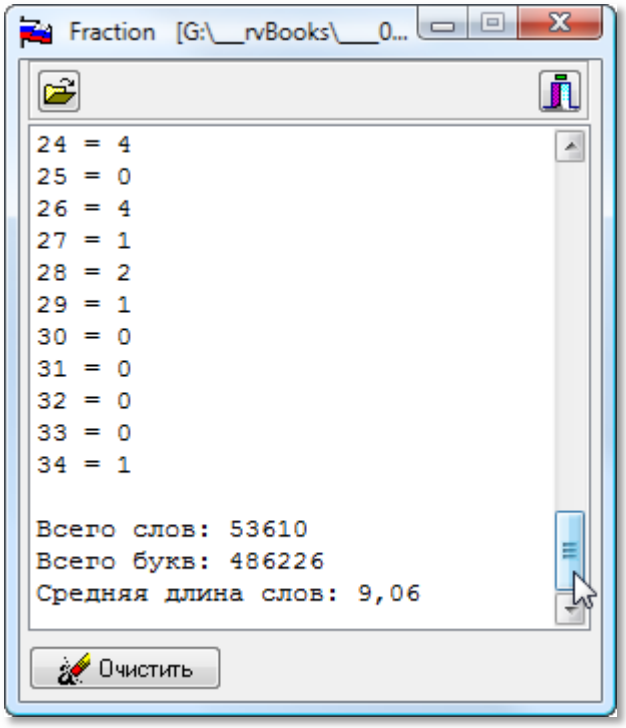


Рис. С11.2. Слова просеяны!



Исходный код программы находится в папке **Fraction**.

Урок 12. Процедуры и функции для работы со строками

Строки очень часто используются в программах - хотя бы для вывода различных сообщений пользователю, чтобы он не заблудился в интерфейсе. Ну, для таких простых случаев вполне достаточно тех сведений, которые вы получили на уроке, посвящённом строкам. Если же у вас более серьёзные намерения и вы хотите манипулировать строками по своему желанию (например, вы пишете какую-нибудь словесную игру вроде *Балды* или *Логогрифов*), то в *Delphi* найдётся всё нужное и для таких любителей словесности и жонглирования приставками и суффиксами.



Для определённости в примерах будем считать, что s1= 'СМОРОДИНА', а s2='ПАРАШЮТ'.

Когда мы изучали простые операции со строками, то рассматривали **конкатенацию строк** с помощью оператора +. Но объединить две строки можно и функцией

```
function Concat(S1: string; S2: string): string;
```

которая возвращает «сумму» двух строк:

```
concat(s1,s2) → СМОРОДИНАПАРАШЮТ
```

Для **сравнения** строк вместо операторов отношения можно использовать следующие функции.

```
function CompareStr(const S1: string; const S2: string): Integer;
function CompareText(const S1: string; const S2: string): Integer;
```

Обе функции сравнивают строки **s1** и **s2**. Первая – с учётом регистра букв (прописные/строчные), вторая – без учёта.

Эти функции возвращают значение

0, если строки равны;
< 0, если s1 < s2
> 0, если s1 > s2.

Их нельзя применять к русским словам, поэтому лучше пользоваться функциями:

```
function AnsiCompareStr(const S1: string; const S2: string): Integer;
function AnsiCompareText(const S1: string; const S2: string): Integer;
```

которые одинаково хорошо работают и с латинскими, и с русскими буквами.

Теперь перейдём к процедурам и функциям, которые выполняют со словами более сложные действия.

Довольно часто нужно преобразовать все буквы слова либо к верхнему, либо к нижнему регистру. Если слова состоят только из латинских букв, то можно использовать функции:

```
function LowerCase(const S: string): string;
function UpperCase(const S: string): string;
```

Первая переводит слова в нижний регистр (строчные буквы), вторая – в верхний (прописные буквы). Для русских (и для латинских) букв годятся функции:

```
function AnsiLowerCase(const S: string): string;
function AnsiUpperCase(const S: string): string;
```

Например:

Функция `AnsiLowerCase(S1)` вернёт строку «смородина».
Функция `AnsiUpperCase('смородина')` вернёт строку «СМОРОДИНА».

Процедуры вывода сообщений на экран (обычно в метку или список) требуют в качестве параметра строку, а что делать, если нужно предъявить пользователю число? – **Преобразовать число в строку**! Это можно сделать и самостоятельно, но гораздо удобнее воспользоваться встроенными функциями *Delphi*.

Чаще других в программах применяют функции для преобразования **целых чисел** в строку:

```
function IntToStr(Value: Integer): string;
function IntToStr(Value: Int64): string;
```

Например, `IntToStr(1234)` вернёт строку «1234».

Для действительных чисел тоже имеется подобная функция

```
function FloatToStr(Value: Extended): string;
```

Если вы хотите узнать, как десятичное число записывается в 16-ричной системе счисления, примените к нему функции:

```
function IntToHex(Value: Integer; Digits: Integer): string;
function IntToHex(Value: Int64; Digits: Integer): string;
```

Например, `IntToHex(1234, 8)` вернёт строку «000004D2», а `IntToHex(2009, 4)` – строку «07D9».



Второй параметр – **Digits** – задаёт число цифр в 16-ричном числе.

При вводе данных из файла или некоторых компонентов вместо числа мы получаем строку символов. И хотя она состоит из цифр, это не число. В этом случае поможет функция

```
function StrToInt(const S: string): Integer;
```

Причём строка **S** может содержать не только десятичное, но и 16-ричное число.

Например, функции `StrToInt('$07D9')` и `StrToInt('2009')` вернут одно и то же число 2009.

Однако эта функция «опасна»: если в строке обнаружится хотя бы один неверный символ, возникнет ошибка. Если у вас есть обоснованные сомнения в правильности строки, лучше обратиться к надёжной функции

```
function TryStrToInt(const S: string; out Value: Integer):
Boolean;
```

```
var b: boolean;
  i: integer;
 . . .
  b:=TryStrToInt('$07D9s', i);
  if b then lstProtokol.Items.Add (inttostr(i));
```

Она возвращает *TRUE*, если строка преобразована в число без ошибок, и *FALSE*, если с ошибками. Переменная **Value** (в нашем примере **i**) хранит число.

В примере строка '$07D9s' содержит неверный символ «s», поэтому функция вернёт *FALSE* и строку печатать не нужно (на самом деле значение пе-

ременной **i** будет равно 2009, так как функция успеет конвертировать первые символы строки – до первой «нецифры»).

Так же можно использовать и процедуру:

```
procedure Val(S: string; var V: Type; var Code: Integer);
```

S – это строка,
V – переменная целого или действительного типа.
Code – переменная целого типа, в которой возвращается номер ошибочного символа в строке. Если таких символов нет, то значение этой переменной будет равно нулю.

```
var i,v: integer;
begin
  val('2009',v,i);
  lstProtokol.Items.Add (inttostr(v));
end;
```

В результате выполнения этого примера будет напечатано число 2009.

```
begin
  val('$07D9s',v,i);
  lstProtokol.Items.Add (inttostr(v));
end;
```

Этот код также напечатает число 2009, но значение переменной i будет равно не нулю, как в первом случае, а 6, так как шестой символ строки не является 16-ричной цифрой.

Аналогичные функции для *вещественных* чисел:

```
function StrToFloat(const S: string): Extended;
function TryStrToFloat(const S: string; out Value: Extended):
Boolean;
function TryStrToFloat(const S: string; out Value: Double):
Boolean;
function TryStrToFloat(const S: string; out Value: Single):
Boolean;
```

В начале и/или конце строки могут присутствовать **ненужные пробелы**. От них легко избавиться, применив функции

```
function TrimLeft(const S: string): string;
function TrimRight(const S: string): string;
function Trim(const S: string): string;
```

Первая удаляет начальные пробелы, вторая – конечные, третья – и те, и другие. Эти функции возвращают **новую строку** – уже без пробелов.

Узнать длину строки (число символов в ней) можно с помощью функции

```
function Length(S: string): Integer;
```

Например, `Length(S1)` вернёт 9, а Length(S2) – 7.

Ну, а теперь – самые интересные процедуры и функции!

Удаляем часть строки **S** от символа номер **Index**:

```
procedure Delete(var S: string; Index: Integer; Count: Integer);
```

Здесь **Count** – число удаляемых символов.

Например, после выполнения процедуры:

```
delete(s1,1,3);
```

строка s1 будет иметь значение «РОДИНА».

А после процедуры `delete(s2,5,3);` строка s2 будет урезана до «ПАРА».



Если значение переменной **Index** меньше единицы или больше длины строки, то строка останется без изменений. Ничего не случится со строкой и если **Count** меньше единицы.

Если же **Count** больше, чем число символов в строке, начиная с **Index**, то будут удалены все символы до конца строки.

Процедура **Insert** действует противоположно – она вставляет подстроку **Substr** в строку **Dest** после символа номер **Index**:

```
procedure Insert(Substr: string; var Dest: string; Index: Integer);
```

Например, после выполнения процедуры **`insert('3',s1,6);`** строка **s1** превратится в смешное слово «СМОРОЗДИНА».

А процедура **`insert('КА',s2,8);`** позволит нам насладиться неологизмом «ПАРАШЮТКА».

Если же мы не хотим портить исходную строку, как это делает процедура *Delete*, то можем создать новую строку, вырезав из любой строки нужный нам кусок:

```
function Copy(S: string; Index: Integer; Count: Integer): string;
```

Например, строка *s* примет значение «ПАРА» после выполнения функции `s:=copy(s2,1,4);`



Обратите внимание: чтобы получить то же самое слово, что и в примере с **delete(s2,5,3);**, мы указали другие значение параметров в вызове функции!

Функция `copy(s1,4,9)` вернёт в переменную s слово «РОДИНА».

И последняя функция находит позицию подстроки *substr* в строке *str*:

```
function Pos(const substr: string; const str: string): Integer;
```



Если в строке искомая подстрока встречается несколько раз, то первый вызов функции **Pos** вернёт позицию первого вхождения, после чего поиск нужно продолжить со следующей позиции. Например, в слове Ф**ЕЛЬ**ДФЕБЕ**ЛЬ** так можно найти обе «ели».

Для наших слов:

`i:= pos('МОР',s1);` вернёт 2, а
`i:= pos('АШ',s2);` - 4.

Если же подстроки в строке нет, то вернётся безнадёжный ноль!

Семинар 12. Палиндромы, или Арабский взгляд на русские слова

Хоббит, или Туда и обратно

Толкиен

Стремление читать слова задом наперёд присуще даже самым отвратительным личностям, к коим мы, без тени сомнения, можем причислить Полиграфа Полиграфыча Шарикова из романа Михаила Булгакова *Собачье сердце.* Вспомните, как он начал своё восхождение к высотам низменных мыслей с причудливого слова *Абырвалг*, которое поначалу поставило в тупик доктора Борменталя и профессора Преображенского. Впрочем, они быстро догадались, что в оригинальном написании это была лишь вывеска магазина *Главрыба*, прочитанное Шариковым с конца.

Чему нас учит классика на данном примере? – Не всякое слово следует читать «по-арабски», дабы не смущать учёные умы!

Впрочем, истории известен и другой, весьма поучительный пример ретроградного чтения, опровергающий предыдущее утверждение. В порядком уже давние времена на Амуре село на мель судно под названием *Сунь Ятсен* (китайский политик), его пытались стянуть за корму, но безуспешно. Это мероприятие продолжалось очень долго – пока один из матросов не повторил лингвистический подвиг Шарикова и не прочитал название судна с конца. Получилось *Не стянусь*! (только не придирайтесь к мелочам). Тогда попробовали зацепить трос за нос судна – и оно легко снялось с мели.

Кто мешает тебе выдумать порох непромокаемый?

Козьма Прутков

Второй пример лучше первого потому, что в результате прочтения названия судна наоборот получилась вполне осмысленная фраза, обернувшаяся практической пользой. Конечно, это всего лишь случайность, но кто запретит нам придумывать такие «двусмысленные» фразы? Однако больше известны другие фразы – их можно без ущерба для смысла читать и слева направо, и справа налево. Их называют **палиндромами**. Пожалуй, самый известный палиндром *А роза упала на лапу Азора* придумал Афанасий Фет, но мы знаем его только потому, что именно эту «волшебную» фразу диктовала Мальвина своему дубовому ученику Буратино. Легко проверить, что она читается точно так же и в обратную сторону. Почему и волшебная. В литературе вы найдёте множество примеров фраз-палиндромов (или

более патриотично - перевёртышей), порой очень забавных и даже ненормативных. Но – придумывание таких афоризмов – настоящее искусство, которое совершенно не поддаётся алгоритмизации, поэтому мы поставим перед собой чисто техническую задачу – отыскать слова, которые не изменяются при чтении наоборот. Их тоже называют палиндромами, и вы наверняка знаете немало таких слов. Например, *РОТОР*, *ШАЛАШ*, *КАБАК*. Чтобы найти **все** такие слова, достаточно внимательно просмотреть словарь русского языка, но мы делегируем это пагубное занятие компьютеру, немного дополнив программу из предыдущего семинара.

Палиндромная программа

Итак, если вы ещё наигрались словами, давайте составим программу для поиска слов-палиндромов.

Объявим строковые переменные и присвоим им значения:

```
var
  string1: string;
  string2: string;

  string1:= 'РОТОР';  //читаем в прямом направлении
  string2:= 'РОТОР';  //читаем в обратном направлении
```

Очевидно, что если слово не изменяется при чтении задом наперёд, то *string1 = string2*. Велико искушение присвоить переменной *string2* значение перевёрнутой строки *string1*. Например, так:

```
var
  i: integer;
  len: integer;

len := length(string1);
for i:= 1 to len do
  string2[len-i+1] := string1[i];
```

Затем сравнить строки и при их равенстве сделать вывод, что слово-палиндром найдено. Но можно поступить проще и ограничиться одной строковой переменной. Для этого достаточно заметить, что в слове-палиндроме одинаковые буквы расположены симметрично относительно середины слова, то есть достаточно сравнить первую половину букв со второй (если в слове нечётное количество букв, то букву в середине слова ни с какой другой сравнивать не надо).

```
for i:= 1 to len div 2 do
```

```
  if string1[len-i+1]  <> string1[i] then . . .; // не палиндром
```

Остальное, как говорится, дело техники. После запуска программы загружаем нужный нам словарь:

```
// Загрузить словарь по выбору
procedure TfrmMain.sbtOpenFileClick(Sender: TObject);
begin
  opendialog1.DefaultExt:='txt';
  opendialog1.Filter:='Text files (*.txt)|*.TXT';
  opendialog1.InitialDir:= extractfilepath(application.exename);
  opendialog1.Title:='Загрузите новый словарь';
  if opendialog1.Execute then
  begin
    filename:= opendialog1.filename;
    loadfile;
  end
  else
    application.MessageBox  ('Вы  не  загрузили  словарь!',NAME_PROG,
MB_OK);
end;
```

И сразу же создаём и сохраняем на диске словарь палиндромов:

```
////////////////////////////////////////////
//                                        //
//          ЗАГРУЗИТЬ СЛОВАРЬ             //
//                                        //
////////////////////////////////////////////
procedure TfrmMain.LoadFile;
//имя файла хранится в переменной FileName
var
  s : String;
  F: TextFile;
  i,j: integer;
  len: integer;
  flag: boolean;
begin
  //Открываем файл для чтения:
  {$i-}
    AssignFile(F,FileName);
    Reset(F);
  {$i+}
  if IOResult<>0 then    {ошибка при открытии файла}
  begin
    filename:='';     {словарь не загружен!}
    application.MessageBox   ('Словарь   не   загружен!',NAME_PROG,
MB_OK);
  end
  else                   {всё нормально}
  begin
```

```
    WordsAll:= 0;    //всего слов загружено
    //считываем слова из файла:
    while not eof(f) do begin
      inc(wordsall);
      Readln(F, S);
      Spisok[WordsAll] := s;
    end;
    CloseFile(F);

    //записываем на диск словарь палиндромов:
    s:= ExtractFileName(FileName);
    s:= copy(s,1,length(s)-4) + '_pal.txt';
    savedialog1.FileName:= s;
    if not savedialog1.Execute then exit;

    assignfile(f,savedialog1.filename);
    rewrite(f);

    lstProtokol.Items.Add ('');
    for i:=1 to WordsAll do begin
      s:= spisok[i];
      len:= length(s);
      flag:= true;
      for j:= 1 to len div 2 do
         if s[j] <> s[len-j+1] then begin
           flag:= false;
           break;
         end;
      if flag= True then begin
         lstProtokol.Items.Add (s);
         lstProtokol.TopIndex:=lstProtokol.Items.Count-1;
         writeln (f, spisok[i]);
      end;
    end;

    Closefile(f);
    Messagebeep(0);
    lstProtokol.Items.Add ('');
    lstProtokol.Items.Add ('======================');
    lstProtokol.Items.Add ('Записан словарь: ');
    lstProtokol.Items.Add (ExtractFileName(savedialog1.filename));
    lstProtokol.Items.Add ('');
  end;
end;
```

Вы легко найдёте в этом коде наш алгоритм. Важно отметить, что мы добавили флаг, который первоначально устанавливаем в *TRUE*, но как только мы обнаружим несовпадение очередной пары букв, флаг сбрасываем и цикл *for* принудительно завершаем. После окончания проверки букв достаточно проверить состояние флага: если он установлен, то найден па-

линдром, который выводим в список на экране и записываем в файл. Вот и вся премудрость, зато каких «оборотней» мы отловили (Рис. С12.1)!

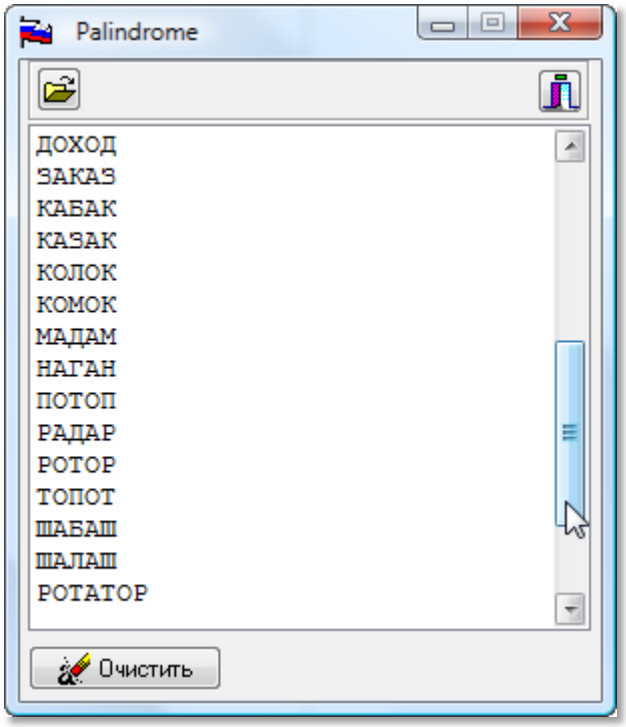


Рис. С12.1. Вот они - перевёртыши!



Исходный код программы находится в папке **palindr**.

Самостоятельная работа



Вертеть слова – занятие увлекательное, поэтому вы можете продолжить это достойное мероприятие.

1. Берите не одно, а пару **разных** слов. Среди них также можно найти палиндромы, хотя ни одно из слов, входящих в пару само палиндромом и не является. Два слова как бы дополняют друг друга до палиндромного состояния. Например, слова **КРАН** и **АРК** именно такие. Читать их поодиночке в обратном направлении бесполезно, но вместе они образуют перевертень: **КРАНАРК**. Конечно, смысла от этого слияния не добавилось, но зато это упражнение вполне может послужить подспорьем при составлении литературных палиндромов типа *ФРАУ И ЛЕДИ СИДЕЛИ У АРФ*.

2. Некоторые слова, не будучи палиндромами, при реверсивном чтении дают другое, вполне осмысленное слово, например **ТОПОР – РОПОТ**. Назовём такие слова **палиндромоидами**. Поскольку они имеют одинаковую длину, то палиндромоиды - это частный случай пар слов из предыдущего задания. Чтобы их найти, придётся каждое слово из списка сравнивать со всеми остальными той же длины, причём, в отличие от палиндромов, проверять придётся все буквы слов, а не половину.

3. В русском языке есть группа причудливых слов, состоящих из двух одинаковых половинок. Кстати говоря, первое человеческое слово как раз такое: *МАМА*. Затем из наших уст появятся *ПАПА*, ДЯДЯ, *БАБА*. Но можно найти примеры и посложнее: *КАНКАН*, *ВАРВАР*. Остальные – найдите сами. Для этого нужно слегка переделать программу для поиска палиндромов, а поиск ограничить словами длиной в 4, 6 и 8 букв. Как видите, задача совсем простая.

Урок 13. Множества

Первая любовь, школьные года,
В лужах голубых стекляшки льда.
Не повторяется, не повторяется,
Не повторяется такое никогда!

«Самоцветная» песня

Множество – это набор элементов (объектов) одного и того же типа. Оно может состоять из произвольного числа элементов (в *Delphi* не более чем из 256), в том числе не иметь ни одного элемента (**пустое множество**). Все элементы множества должны быть различны (если вы и попытаетесь добавить элемент, который уже имеется в множестве, то просто ничего не произойдёт).

Объявляются множества с помощью зарезервированных слов **set** и **of**, после которых указывается тип элементов (базовый тип):

```
Type
  Имя = set of тип_элементов;
```

Базовым типом могут быть: логический тип, символьный тип **char**, перечисляемый и интервальный типы, содержащие не более 256 элементов. Типы целых и вещественных чисел не могут быть базовым типом, поскольку они заведомо содержат более 256 элементов.

Например:

```
Type
  A= set of 1..100;
  B= set of boolean;
  C= set of char;

  TColor = (Red, Orange, Yellow, Green, Cyan, Blue, Violet);
  Color= set of TColor;
```

Или так

```
color= set of (Red, Orange, Yellow, Green, Cyan, Blue, Violet);
```

Множество **A** может содержать 100 элементов со значениями от 1 до 100.
Множество **B** может содержать 2 элемента со значениями *FALSE* и *TRUE*.
Множество **C** может содержать 255 символов.
Множество **color** может содержать до 7 цветов радуги.

Интервальный тип должен укладываться в диапазон *0..255*, поэтому объявление

```
A1= set of 0..255;
```

правильное, а A2= set of 1..256; - неправильное.

После задания новых типов, можно объявлять переменные и типизированные константы:

```
Var clr1, clr2, clr2: color;
  Ch: C;
  Numbers: A;
  Bool: B;
```



При объявлении переменные инициализируются пустым множеством **[]**, то есть не содержат ни одного элемента.

Объявление типизированной константы можно совмещать с инициализацией множества. Для этого в квадратных скобках нужно перечислить через запятую значения элементов множества (допускаются интервалы значений):

```
const
  prime: set of 2..10= [2,3,5,7];
  alphabet : set of char= ['a'..'я'];
```

Такой способ определения называется **конструктором множества**. С его помощью можно присваивать значения переменным:

```
clr1:= [];                      //пустое, «бесцветное» множество
clr2:= [Red, Orange, Yellow];//множество тёплых цветов
clr3:= [Green..Violet];         //множество холодных цветов
```

Поскольку элементы в множестве не упорядочены, то последовательность их добавления в множество значения не имеет. Например, три операции присваивания дадут то же самое множество, что `clr2:= [Red, Orange, Yellow];`:

```
clr2:= [Orange];
clr2:= clr2+[Red];
clr2:= clr2+[Yellow];
```

Для добавления элементов в множество можно использовать процедуру **include(S,I)**, которая выполняет те же действия, что и оператор S := S + [I], но даёт более быстрый код:

```
Include(clr2,Orange);
Include(clr2,Red);
Include(clr2,Yellow);
```

Для обратной операции – исключения элемента из множества – служит процедура **exclude(S,I)**, которая выполняет те же действия, что и оператор `S := S - [I]`, но даёт более быстрый код:

```
clr2:= [Red, Orange, Yellow];
exclude(clr2,Red); // исключить из множества красный цвет.
```

Останутся только оранжевый и жёлтый.

Операции над множествами

Пусть первое множество **M**:= [Red, Orange], второе **N**:= [Orange, Yellow], а результат выполнения операции – множество **S**.

+ оператор **объединения множеств**. В результате операции получается новое множество, которое содержит по одному разу все элементы исходных множеств:

`S := M + N;` → S = [Red, Orange, Yellow].

- оператор **разности множеств** (относительное дополнение). В результате операции получается новое множество, которое содержит элементы первого множества, не принадлежащие второму множеству:

`S := M - N;` → S = [Red].

***** оператор **пересечения множеств**. В результате операции получается новое множество, которое содержит элементы, принадлежащие обоим множествам:

`S := M * N;` → S = [Orange].

Операторы отношения

Операции отношения применяются к двум множествам, результате операции - логическое значение *FALSE* или *TRUE*.

= оператор **эквивалентности**. Результат сравнения равен *TRUE*, если множества содержат одни и те же элементы, и *FALSE* в противном случае.

<> оператор **неэквивалентности**. Результат сравнения равен *TRUE*, если множества отличаются хотя бы одним элементом, и *FALSE* – если множества содержат одни и те же элементы (то есть равны).

<= оператор **подмножества**. Результат операции равен *TRUE*, если все элементы первого множества принадлежат и второму множеству.

>= оператор **включающего множества**. Результат операции равен *TRUE*, если все элементы второго множества принадлежат и первому множеству.

in - оператор, с помощью которого можно проверить, входит ли какой-либо элемент в множество. Если входит, то результат равен *TRUE*, если не входит – *FALSE*.

Конструкция с оператором **in** записывается так: слева указывается элемент, затем оператор **in** и, наконец, множество. Например,

```
Orange in M
```

Поскольку элемент **Orange** принадлежит множеству **N**, то результат операции равен *TRUE*.

А эта проверка закончится неудачно:

```
Yellow in M,
```

так как элемент **Yellow** принадлежит множеству **N**, но не **M**.

Эта операция очень часто используется для сокращения записи длинных условий в логических условиях. Пусть нам нужно проверить, что некоторое число от 1 до 10 является простым. Мы можем записать так:

```
If (i=2) or (i=3) or (i=5) or (i=7) then . . . // число простое
```

Но с оператором **in** эта проверка гораздо понятнее:

```
If i in [2, 3, 5, 7] then . . . // число простое
```

Сколько элементов в множестве?

Иногда требуется узнать, сколько элементов содержит то или иное множество. Конечно, можно после каждой операции с множеством корректировать это число, но значительно проще написать пару строк кода и подсчитывать число элементов в нужном месте программы.

```
Type
  TColor = (Red, Orange, Yellow, Green, Cyan, Blue, Violet);
  color= set of TColor;

var
  i: integer;
  clr2: color;
  cs: Tcolor;

clr2:= []; //не обязательно
clr2:= [Red, Orange, Yellow];

  i:=0;
  for cs:= Red to Violet do
    if cs in clr2 then inc(i);
```

Как и следовало ожидать, *i* примет значение три, то есть в множестве содержится 3 элемента.

Приоритет выполнения операций

*
+ -
In = <> <= >=

Сначала выполняется операция пересечения множеств, затем – объединения и разности, и так далее.

Порядок выполнения операций можно изменить с помощью круглых скобок.

Диаграммы Венна

Для наглядного представления логических связей (отношений) между множествами часто используют *диаграммы Венна* (*Venn diagrams*), в которых множества изображают кругами, эллипсами или замкнутыми кривыми, а элементы множества – точками.



Графические методы использовались в науке с давних времен. Так, еще в 13-ом веке Раймонд Луллий употреблял в своих работах подобные круги. Найдем мы их и у немецкого математика Г.В. Лейбница. Чешский математик Бернард Больцано вместо кругов рисовал прямоугольники. В книге *Алгебра логики* круги применял немецкий математик Эрнест Шрёдер.

Распространению метода особенно способствовал Леонард Эйлер, который в *Письмах к немецкой принцессе* (1768) с их помощью пояснял решение силлогизмов (логических рассуждений) Аристотеля.

Английский логик *Джон Венн* (*John Venn*, 1843—1923) обобщил этот графический метод в книге *Символическая логика* (1881).

При внешнем сходстве кругов Эйлера и диаграмм Венна они отличаются областями применения. Первые использовались в традиционной силлогистике, вторые – в математической логике. В некоторых случаях их объединяют под общим названием *круги* (диаграммы) *Эйлера-Венна*. В нашем случае был бы более уместен именно этот термин, но мы всё-таки остановимся на более коротком варианте.

Объявим несколько множеств, элементами которых могут быть однозначные натуральные числа:

```
var
  M,N,K,L,P,S: set of 1..9;
```

Пока все множества пустые, но мы заполним их числами:

```
M:=[1..5];
N:=[3,5..7];
K:= M;
L:=[2,3];
P:=[1..7];
```

Теперь все множества можно представить в виде кругов, внутри которых находятся элементы множеств (Рис. 13.1).

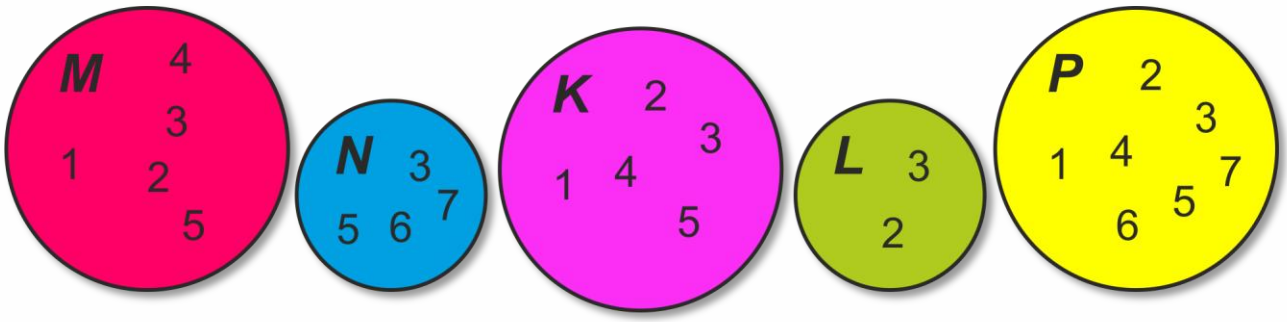


Рис. У13.1. Наши множества

А вот так с помощью диаграмм Венна можно изобразить операции с множествами.

Объединение множеств

S:= M + N;

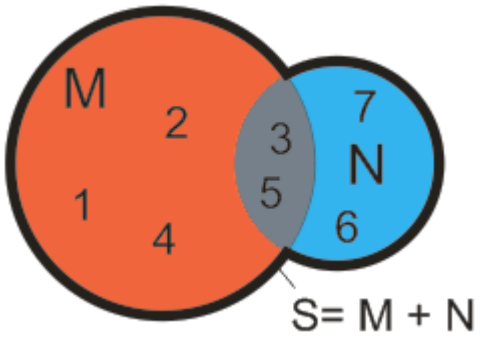


Рис. У13.2. Объединение множеств

Сразу видно (Рис. У13.2), что множество **S= [1,2,3,4,5,6,7]**, то есть содержит все элементы множеств **M** и **N** по одному разу.

Разность множеств (Рис. У13.3)

S:= M - N;

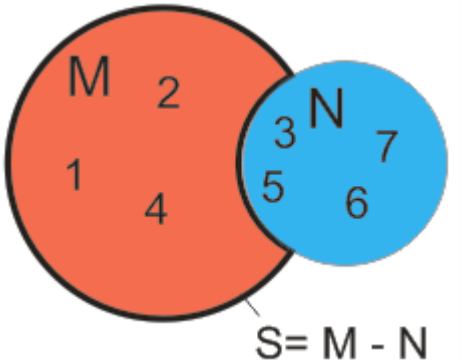


Рис. У13.3. Разность множеств **S**= [1,2,4]

Пересечение множеств (Рис. У13.4)

S:= M * N;

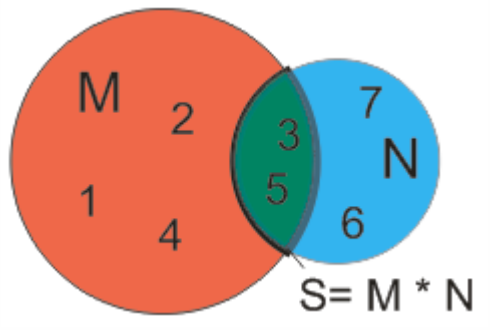
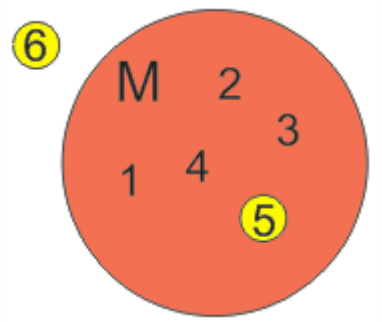


Рис. У13.4. Пересечение множеств **S**= [3,5]

Оператор in (Рис. У13.5)



6 in **M** → FALSE
5 in **M** → TRUE

Рис. У13.5. Оператор *in*

Эквивалентность множеств (Рис. У13.6)

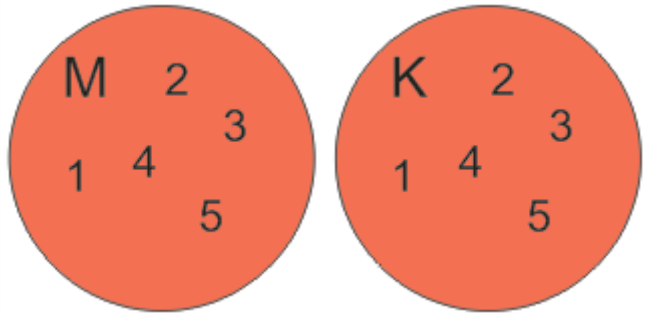


Рис. У13.6. Эквивалентность множеств *M* = *K*

Неэквивалентность множеств (Рис. У13.7)

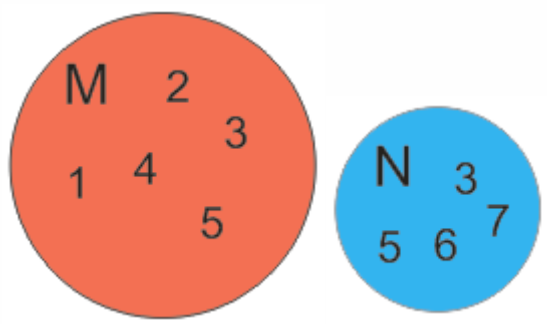


Рис. У13.7. Неэквивалентность множеств *M* <> *N*

Подмножество (Рис. У13.8)

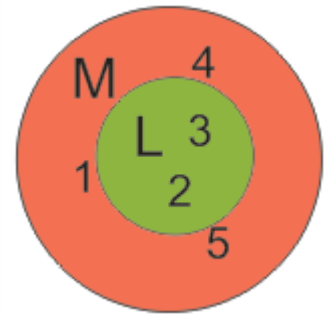


Рис. У13.8. Подмножество *L* <= *M*

Включающее множество (Рис. У13.9)

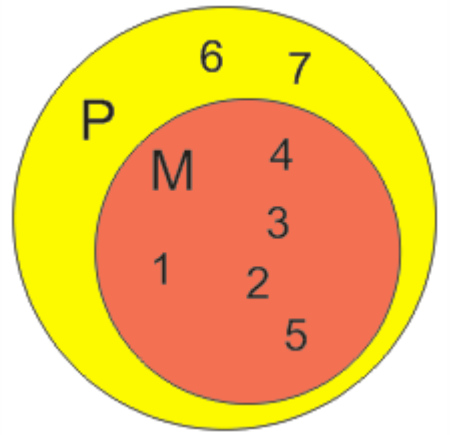


Рис. У13.9. Включающее множество *P* >= *M*

Семинар 13. Разнобуквица, или SETка для слов

Предметом наших изысканий станут слова, которые состоят из разных букв, то есть ни одна буква в них не повторяется. Например, *ШКОЛА*, *УЧЕНИК*, *УЧЕНИЦА* – это именно такие слова, а вот слово *КЛАСС* не годится – в нём не одна, а две буквы *С*.

Из этих примеров видно, что найти несколько разнобуквенных слов совсем не трудно, но наша задача – монументальная: найти все такие слова (точнее, все существительные) и наградить своим особым вниманием самое длинное из них.

Можно внимательно просмотреть словарь и выписать нужные слова, но это занятие очень долгое и утомительное. Оно больше подходит компьютеру - вот мы и составим для него небольшую программу, а уж он пусть просматривает словарь!

Благо словарь на диске у нас имеется, и мы легко загрузим его в массив строк **Spisok**:

```
const
   MAX_LEN_WORD=40;                 //!!! - макс. длина слов в слова-
ре !!!
   MAX_DICTIONARY=300000;           //!!! - макс. число слов в слова-
ре !!!

var
  frmMain: TfrmMain;
  NAME_PROG : PChar =' RasnBukvy';                    // - название
программы
  WordsAll : integer;                                // - всего счита-
но слов из файла словаря
  Spisok : array [1 .. MAX_DICTIONARY] of string;// - массив всех
слов словаря
  FileName : string='';                              // - имя файла
словаря (= '', если ни один словарь не загружен)
```

Теперь мы можем последовательно брать из него по одному слову и проверять, подходит ли оно нам или нет. **Самый простой способ** такой. Проверяем каждую букву слова, начиная с первой и кончая предпоследней (последняя буква уж точно не совпадёт ни с одной другой буквой слова), со всеми последующими. Если совпадёт, слово не подходит, не совпадёт – переходим к следующей букве. Если в результате **всех** проверок ни одна буква не совпала, значит, мы нашли разнобуквицу, которую сохраняем в файле для дальнейшего использования.

В коде наш алгоритм может выглядеть так:

```
     for i:=1 to WordsAll do  begin
      flag:= FALSE;
      len:= length(spisok[i]);
      for j:=1 to len-1 do
        for n:= j+1 to len do begin
          if spisok[i,j] = spisok[i,n] then begin //- такая буква
уже была!
            flag:= TRUE;
            break;
          end
      end;
      if not flag then
        writeln (f, spisok[i]);
    end;
```

Второй способ элегантнее. Для каждого слова мы заводим пустое **множество**, в которое будем добавлять буквы слова. Но перед этим проверим, нет ли уже в множестве такой буквы. Ну, а дальше действуем, как в первом случае:

```
    for i:=1 to WordsAll do  begin
      //в множестве нет букв:
      ch:=[];
      flag:= FALSE;

      for j:=1 to length(spisok[i]) do  begin
        if  (spisok[i,j]  in  ch)  then  begin  //-  такая  буква  уже
была!
         flag:= TRUE;
         break;
        end
        else
          //добавить букву в множество:
          include(ch, spisok[i,j]);
      end;
      if not flag then
        writeln (f, spisok[i]);
    end;
```



Строку **include(ch, spisok[i,j]);** вы можете заменить так: **ch:= ch + [spisok[i,j]];** Результат будет тот же самый.

Мы не только избавились от одного цикла **for**, но и придумали более простой алгоритм.

Да, а какое же всё-таки самое длинное слово в русском языке, в котором буквы не повторяются? Заглянем в самый конец полученного нами файла и прочитаем его:

ЧЕТЫРЁХУГОЛЬНИК

В нём 15 букв и все разные! Слово простое и всем известное, но легко ли было бы найти его без помощи компьютера? А ведь мы нашли не только это, но и вообще все разнобуквенные слова русского языка, и всего за несколько минут.

В интерфейсе программы для нас нет ничего нового и потому интересного (Рис. С13.1).

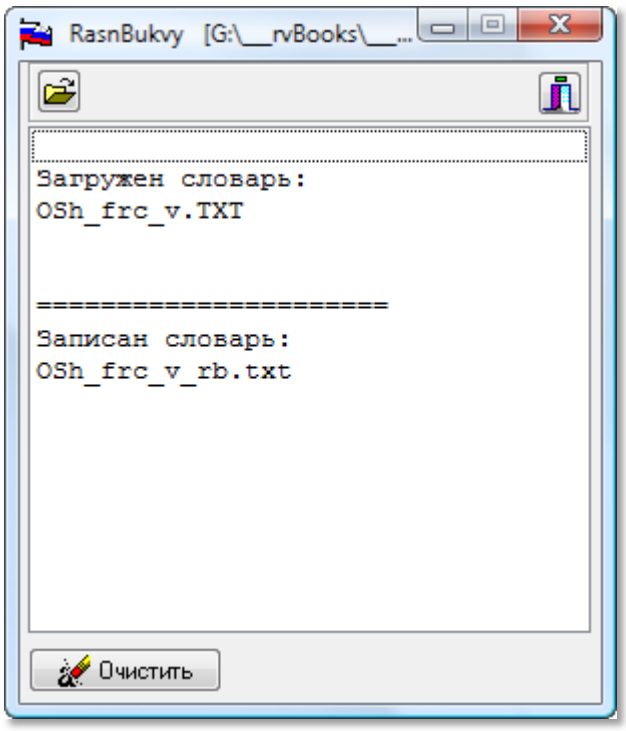


Рис. С13.1. Разнобуквенный интерфейс

А **программа работает так**. После её запуска нажмите кнопку с папкой и загрузите словарь (на диске 3 словаря, так что выбирайте сами, или можете использовать свой собственный словарь):

```
// Загрузить словарь по выбору
procedure TfrmMain.sbtOpenFileClick(Sender: TObject);
begin
  opendialog1.DefaultExt:='txt';
  opendialog1.Filter:='Text files (*.txt)|*.TXT';
  opendialog1.InitialDir:= extractfilepath(application.exename);
  opendialog1.Title:='Загрузите новый словарь';
```

```
  if opendialog1.Execute then
  begin
    filename:= opendialog1.filename;
    loadfile;
  end
  else
    application.MessageBox ('Вы не загрузили словарь!',NAME_PROG,
MB_OK);
end;
```

В процедуре **LoadFile** не только загружается словарь, но и отыскиваются нужные слова. Вы можете выбрать любой из двух разработанных нами алгоритмов. Они работают очень быстро, так что оба хороши.

```
////////////////////////////////////////////
//                                        //
//          ЗАГРУЗИТЬ СЛОВАРЬ             //
//                                        //
////////////////////////////////////////////
procedure TfrmMain.LoadFile;
//имя файла хранится в переменной FileName
var
  s : String;
  F: TextFile;
  i,j,n: integer;
  ch: set of Char;
  flag: boolean;
  len: integer;
begin
  //Открывем файл для чтения:
  {$i-}
    AssignFile(F,FileName);
    Reset(F);
  {$i+}
  if IOResult<>0 then    {ошибка при открытии файла}
  begin
    filename:='';     {словарь не загружен!}
    application.MessageBox   ('Словарь   не   загружен!',NAME_PROG,
MB_OK);
  end
  else                   {всё нормально}
  begin
    WordsAll:= 0;    //всего слов загружено

    //считываем слова из файла:
    while not eof(f) do begin
      inc(WordsAll);
      Readln(F, S);
      Spisok[WordsAll] := s;
    end;
    CloseFile(F);
```

```
    frmMain.caption:= NAME_PROG + '   [' + filename + ']';
    lstProtokol.Items.Add ('Загружен словарь: ');
    lstProtokol.Items.Add (ExtractFileName(FileName));
    lstProtokol.Items.Add ('');

    //записать на диск фракционный словарь:
    s:= ExtractFileName(FileName);
    s:= copy(s,1,length(s)-4) + '_rb.txt';
    savedialog1.FileName:= s;
    if not savedialog1.Execute then exit;

    assignfile(f,savedialog1.filename);
    rewrite(f);

    for i:=1 to WordsAll do  begin
      //в множестве нет букв:
      ch:=[];
      flag:= FALSE;

      for j:=1 to length(spisok[i]) do  begin
        if  (spisok[i,j]  in  ch)  then  begin  //-  такая  буква  уже
была!
         flag:= TRUE;
         break;
        end
        else
          //добавить букву в множество:include(ch, spisok[i,j]);
          ch:= ch + [spisok[i,j]];
      end;
      if not flag then
        writeln (f, spisok[i]);
    end;

//     for i:=1 to WordsAll do  begin
//      flag:= FALSE;
//      len:= length(spisok[i]);
//      for j:=1 to len-1 do
//        for n:= j+1 to len do begin
//             if spisok[i,j] = spisok[i,n] then begin //- такая
буква уже была!
//            flag:= TRUE;
//            break;
//          end
//      end;
//      if not flag then
//        writeln (f, spisok[i]);
//    end;

    Closefile(f);
    Messagebeep(0);
    lstProtokol.Items.Add ('');
    lstProtokol.Items.Add ('======================');
    lstProtokol.Items.Add ('Записан словарь: ');
```

```
    lstProtokol.Items.Add (ExtractFileName(savedialog1.filename));
    lstProtokol.Items.Add ('');
  end;
end;
```



Исходный код программы находится в папке **rasnbukvy**.

Супернаборщик

Наверное, всем известна школьная игра *Наборщик*. В ней требуется из букв какого-либо заданного длинного слова составлять другие, более короткие слова, с условием, что они будут состоять только из тех букв, которые имеются в заданном слове. При этом, если в длинном слове, например, две буквы А, то и в каждом составленном слове их должно быть не больше двух. Побеждает тот игрок, который составит больше слов. Очень хорошая игра, в которую интересно играть, особенно на уроках математики.

Мы сделаем себе небольшое послабление в правилах игры – не будем требовать, чтобы число одинаковых букв в искомых словах не превышало числа этих букв в заданном слове, иначе говоря, любую букву этого слова можно использовать сколько угодно раз. Все остальные правила трогать не будем.

Все компоненты программы нам уже хорошо известны, поэтому обратим своё внимание на процедуру составления списка слов, в которую мы попадём после нажатия на кнопку **sbtFind** *НАЙТИ СЛОВА*:

```
//НАЙТИ НУЖНЫЕ СЛОВА
procedure TfrmMain.sbtFindClick(Sender: TObject);
var
  word : AnSiString;
  i,j: integer;
  ch: set of Char;
  flag: boolean;
begin
  //конвертируем буквы в верхний регистр - как в словаре:
  word:= AnsiUpperCase(txtWord.Text);
  if length(word) < 2 then
    Application.MessageBox('Слишком        короткое        слово!',
'Супернаборщик', MB_OK);

  //в множестве пока нет букв:
  ch:=[];
  //формируем множество из букв заданного слова:
```

```
  for i:=1 to length(word) do  begin
    //добавить букву в множество:include(ch, word[i]]);
    ch:= ch + [word[i]];
  end;

  lstProtokol.Items.clear;
  for i:=1 to WordsAll do  begin
    flag:= TRUE;
    for j:=1 to length(spisok[i]) do  begin
      if not (spisok[i,j] in ch) then begin //- такой буквы нет!
         flag:= FALSE;
         break;
      end;
    end;
    if flag then
      lstProtokol.Items.Add (spisok[i]);
  end;
end;
```

Сначала мы должны конвертировать все буквы в верхний регистр, потому что в словаре все слова записаны «большими» буквами. Иначе ни одного слова программа не найдёт!

Ясно, что бесполезно искать слова, состоящие только из одной буквы, хотя бы и взятой много раз (но такие слова могут появиться в будущем!), поэтому необходимо **проверить введённое слово**.

Далее мы заполняем пустое множество буквами заданного слова и просматриваем весь словарь. Все буквы каждого слова проверяем, имеются ли они в множестве **ch** (а значит, и в заданном слове). Как только нам встретилась «лишняя» буква, проверку текущего слова прекращаем и переходим к следующему в массиве **spisok**. Слово, которое с честью выдержало все испытания, печатаем в окне протокола. Все найденные слова можно сохранить в файле, нажав кнопку **sbtSave** с зазывным предложением *Записать* (Рис. С13.2).

Всего за несколько минут мы написали очень интересную и полезную программу для игры в слова! Теперь вы смело можете играть в неё с кем угодно, не боясь ударить в грязь лицом, и каждый поверженный вами противник скажет о вас: «Да, этот парень в карман за словом не полезет!». Вот так, с помощью ловкости рук и без всякого мошенничества вы прослывёте настоящим эрудитом даже за пределами своей родной школы.

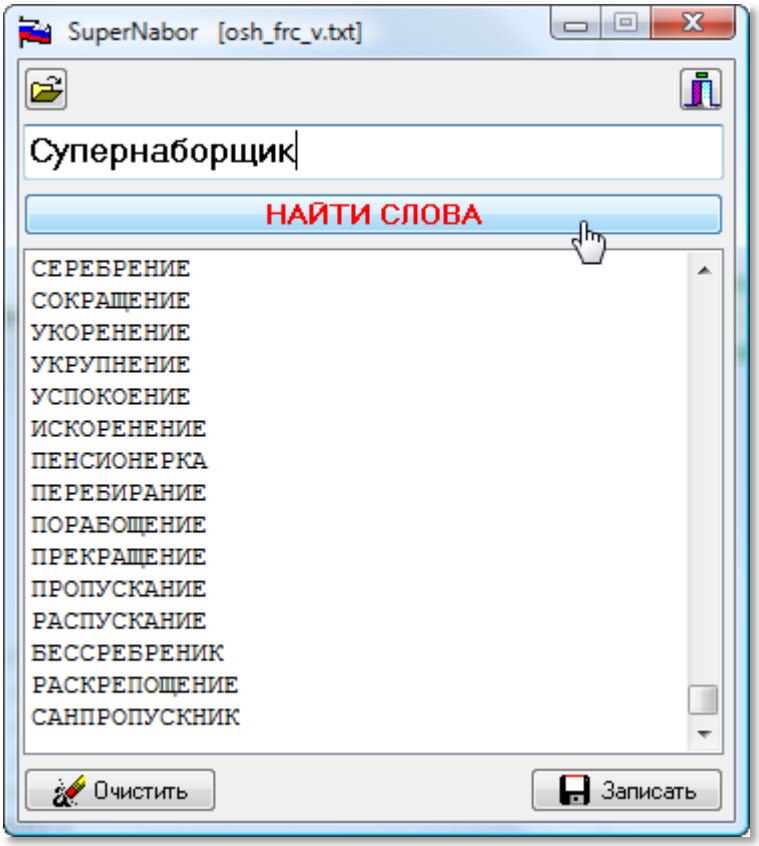


Рис. С13.2. Супернаборщик в действии



Исходный код программы находится в папке **supernabor**.

Нашу программу можно использовать и вот таким неожиданным образом.

Предположим, что на вашем компьютере имеются только латинские буквы. Сколько русских слов вы сможете напечатать?

Для определённости будем считать, что только русские буквы **АВСЕНКМОРТХ** имеют латинские аналоги (Рис. С13.3).



Злые шутки с идентификаторами

Так как *Delphi* позволяет записывать идентификаторы и русскими буквами, то легко разыграть начинающих программистов. Составьте идентификатор так, чтобы он мог быть записан русскими буквами, но при этом выглядел бы так, как будто записан латинскими.

Например, раса сера архар серсо. При попытке воспользоваться этими идентификаторами будет возникать ошибка, так как все программисты уверены, что идентификаторы составлены из латинских букв.

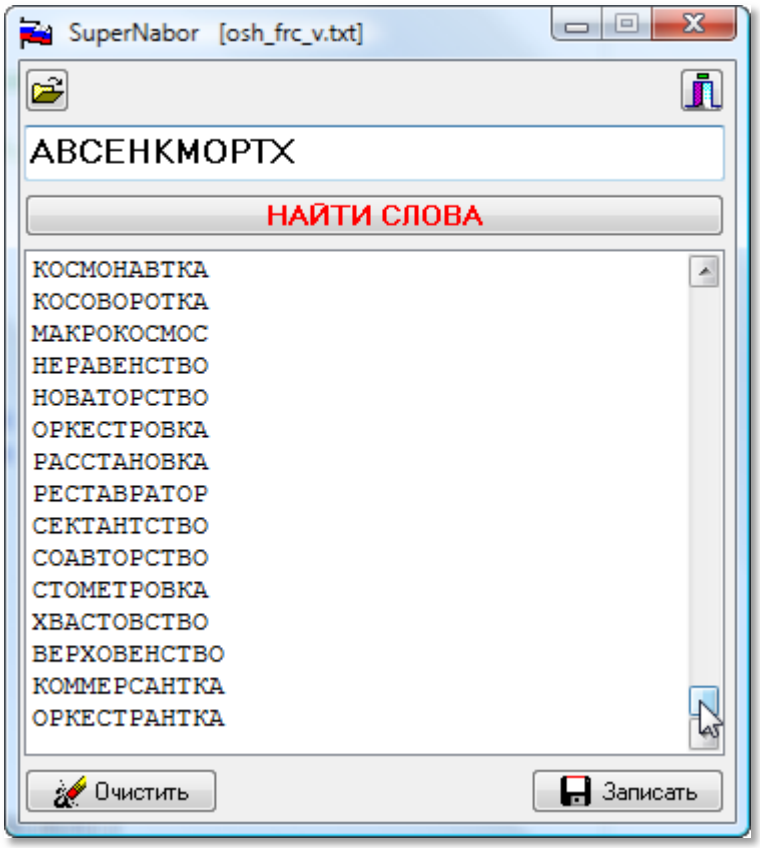


Рис. С13.3. Латинско-русский словарик составлен!



Самостоятельная работа

Не почивайте, однако, на лаврах, а лучше напишите программу для игры в классического *Наборщика*! За основу можно взять нашу программу, но процедуру поиска слов придётся изменить: буквы заданного слова придётся считать, так что нужно использовать массивы (например, `var word, curWord: array[char] of integer; . . .` заполнить массив **word** буквами заданного слова, скопировать массив `curWord:= word;` а затем вычитать из него найденные в очередном слове буквы), множество здесь бессильно.

Ответ на следующей странице.

Наборщик

Ирония судьбы: запрограммировать игру *Супернаборщик* оказалось легче, чем простого *Наборщика*! Однако, начнём.

Как мы и договаривались, заведём пару массивов для подсчёта букв в словах:

```
Var word, curWord: array['А'..'Я'] of integer;
```

Вместо массива **array[char]** нам вполне хватит массива **array['А'..'Я']** – других букв в наших словах нет.

В массиве **word** мы будем хранить число одинаковых букв в заданном слове. Сначала мы этот массив обнуляем, а потом подсчитываем все буквы заданного слова. Если каких-то букв в слове не окажется, их число будет равно нулю.

Как обычно, загружаем словарь в массив строк **spisok** и «вытаскиваем на свет божий» по одному слову. Для каждого из них копируем массив **word** в **curWord**:

```
curWord:= word;
```

Теперь мы получили в своё распоряжение все буквы (и в эквивалентном количестве) заданного слова. В цикле *for* уменьшаем на 1 число тех букв, которые в «подопытном» слове стоят на 1, 2 и последующих местах. Если вдруг букв не хватило, значит, слово из списка невозможно составить из букв заданного слова. Если всё нормально, выводим найденное слово в протокол:

```
//НАЙТИ НУЖНЫЕ СЛОВА
procedure TfrmMain.sbtFindClick(Sender: TObject);
var
  s : String;
  i,j: integer;
  flag: boolean;
  word, curWord: array['А'..'Я'] of integer;
  ch: char;
begin
  //конвертируем буквы в верхний регистр - как в словаре:
  s:= AnsiUpperCase(txtWord.Text);
  if length(s) < 2 then
    Application.MessageBox('Слишком короткое слово!', 'Наборщик',
MB_OK);
```

```
  //обнуляем массив - ни одной буквы пока в нём нет:
  for ch:='А' to 'Я' do
    word[ch]:=0;

  //считаем буквы в слове и формируем массив:
  for i:=1 to length(s) do  begin
    inc(word[s[i]]);
  end;

  lstProtokol.Items.clear;
  //отыскиваем подходящие слова и заносим их в протокол:
  for i:=1 to WordsAll do  begin
    curWord:= word;
    flag:= TRUE;
    for j:=1 to length(spisok[i]) do  begin
      dec(curWord[spisok[i,j]]);
      if curWord[spisok[i,j]] < 0  then begin //- такой буквы нет!
         flag:= FALSE;
         break;
      end;
    end;
    if flag then
      lstProtokol.Items.Add (spisok[i]);
  end;
end;
```

А не так уж и много пришлось переделать кода, зато вон сколько слов удалось составить из названия программы (Рис. С13.4).

С помощью этой программы легко отыскать и все разнобуквицы. Достаточно в качестве исходного слова использовать *все* буквы русского алфавита (Рис. С13.5).



Исходный код программы находится в папке **nabor**.

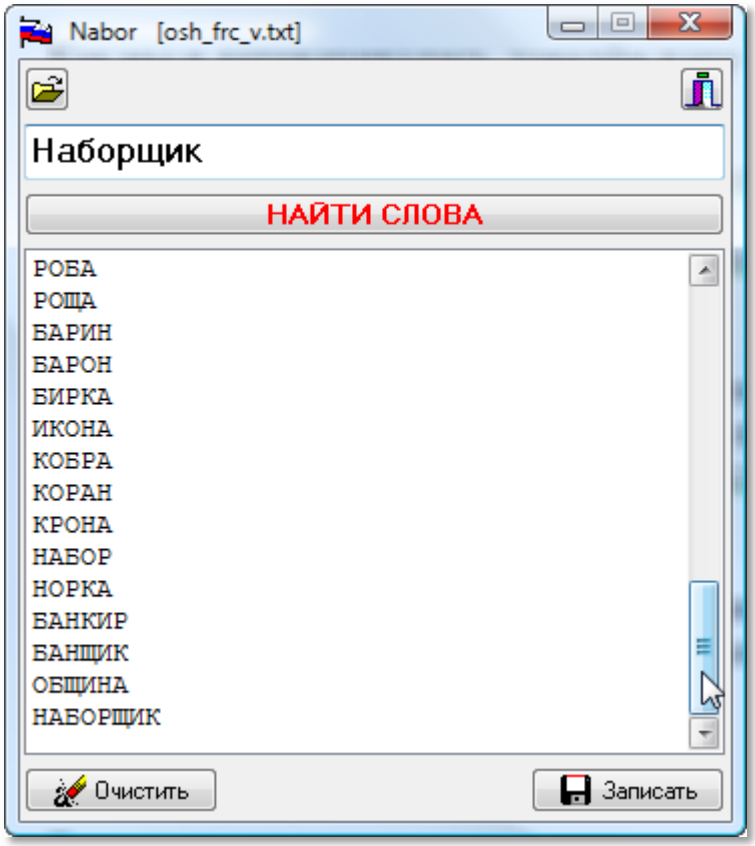


Рис. С13.4. «Наборщик» нашёл «наборщика»

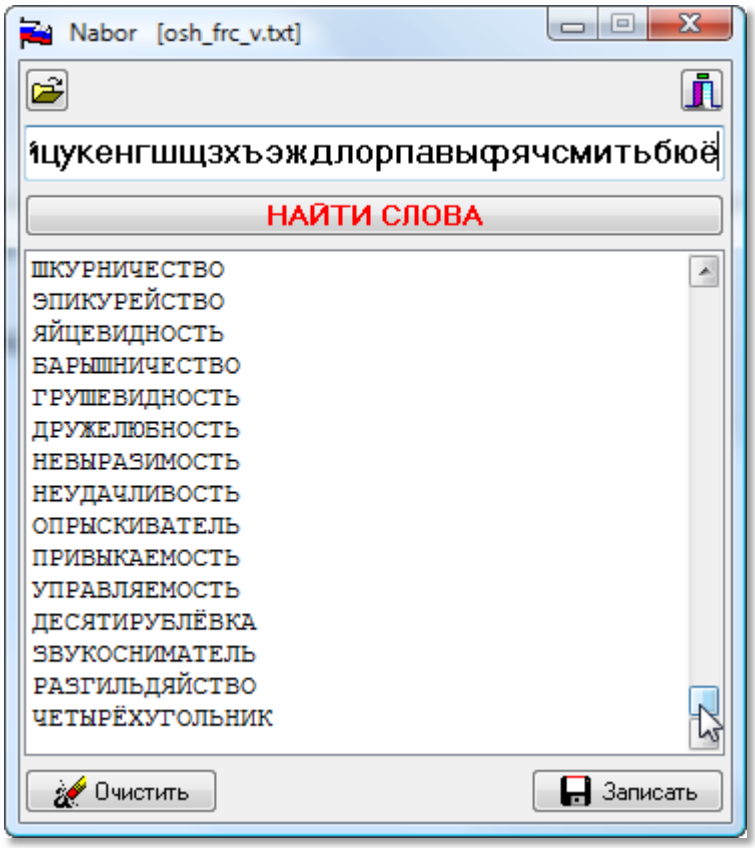


Рис. С13.5. Ничего себе побочный эффект!

Самостоятельная работа

Вы, должно быть, заметили, что программа *Наборщик* находит и анаграммы тех слов, которые их имеют (Рис. С13.6).

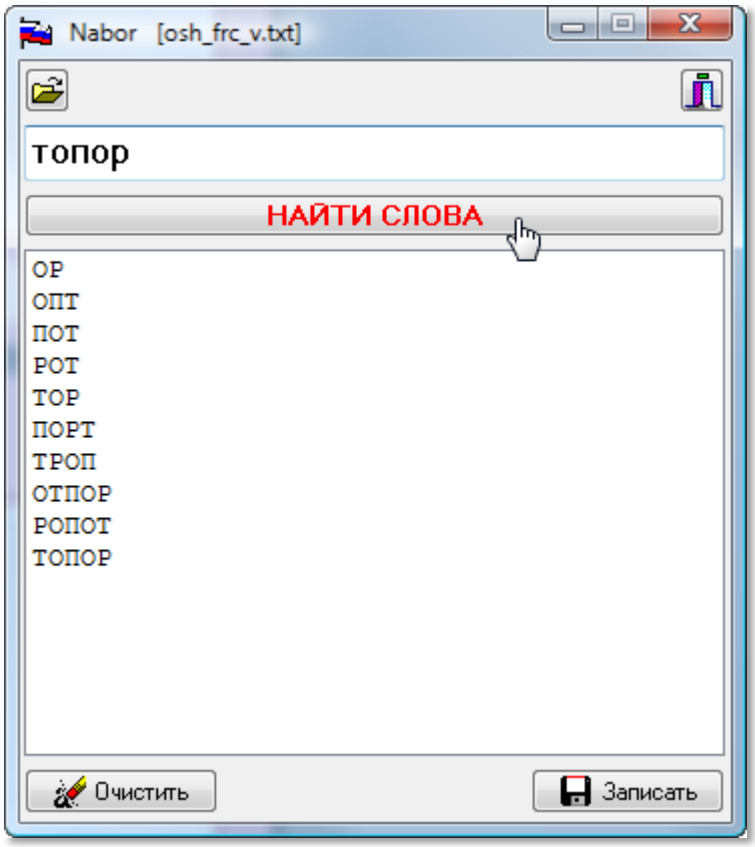


Рис. С13.6. Ищем анаграммы

Анаграммы – это слова, которые состоят из одних и тех же букв в равных количествах. Например, **ТОПОР – РОПОТ – ОТПОР**.

Естественно, все анаграммы состоят из равного количества букв, поэтому при поиске анаграмм нужно проверять только слова той же длины, что и заданное слово. Это значительно сокращает перебор слов, но – нужно заранее знать, что заданное слово имеет анаграмму, иначе придётся перебирать все слова.

Но мы уже выяснили, что полный перебор слов лучше поручить компьютеру. Напишите программу, которая найдёт все анаграммы русского языка!

Ответ на следующей странице.

Анаграммы

Мы начнём с того, что загрузим словарь и сразу же приступим к поиску анаграмм (Рис. С13.7):

```
////////////////////////////////////////////
//                                        //
//              СОЗДАТЬ ФОРМУ             //
//                                        //
////////////////////////////////////////////
procedure TfrmMain.FormCreate(Sender: TObject);
begin
   {загрузить словарь по умолчанию и запомнить его имя}
   filename:= DefaultFileName;
   loadfile;
end;

procedure TfrmMain.FormActivate(Sender: TObject);
begin
  Find;
end;
```

Казалось бы, задача непростая, но достаточно сделать несколько дополнений к *Наборщику*, и можно приниматься за дело:

```
//НАЙТИ НУЖНЫЕ СЛОВА
procedure TfrmMain.Find;
var
  s,str : String;
  i,j,n,len: integer;
  flag: boolean;
  word, curWord: array['A'..'Я'] of integer;
  ch: char;
begin
  sbtSave.Enabled:= false;
  sbtOpenFile.Enabled:= false;

  lstProtokol.Items.clear;
  for n:=1 to WordsAll do  begin
    //обнуляем массив - ни одной буквы пока в нём нет:
    for ch:='A' to 'Я' do
      word[ch]:=0;

    s:= spisok[n];
    len:= length(s);
    //считаем буквы в слове и формируем массив:
    for i:=1 to len do  begin
      inc(word[s[i]]);
    end;
```

```
    str:='';
    //отыскиваем анаграммы и заносим их в протокол:
    for  i:=BeginWord[len]  to  BeginWord[len]+NumWords[len]-1  do
begin
      curWord:= word;
      flag:= TRUE;

      for j:=1 to len do  begin
        dec(curWord[spisok[i,j]]);
        if  curWord[spisok[i,j]] < 0   then begin //- такой буквы
нет!
           flag:= FALSE;
           break;
        end;
      end;
      if  flag  and  (s<>  spisok[i])  then  str:=  str  +  '  '  +  spi-
sok[i];

    end;
    application.ProcessMessages;
    if str<>'' then lstProtokol.Items.Add(s + '' + str);
  end;
  sbtSave.Enabled:= true;
  sbtOpenFile.Enabled:= true;
end;
```

На что нужно обратить внимание в этой процедуре?

- Желательно отключить «лишние» кнопки. Всё-таки поиск анаграмм дело небыстрое, немного придётся и подождать, но ведь от безделья пользователь начнёт жать на все кнопки, а это чревато. Нужно оградить его от вредной забавы (по-научному, это называется «защита от дурака», не в обиду будь сказано).

- Роль заданного слова будут последовательно исполнять все слова из списка.

- Изящнее и правильнее анаграммы выписывать не в колонку, а в одну строку, поэтому придётся постараться и сделать красиво.

- Так как каждое слово является анаграммой самого себя, то мы для **всех** слов найдём «анаграммы», что нехорошо. Это надо учесть!

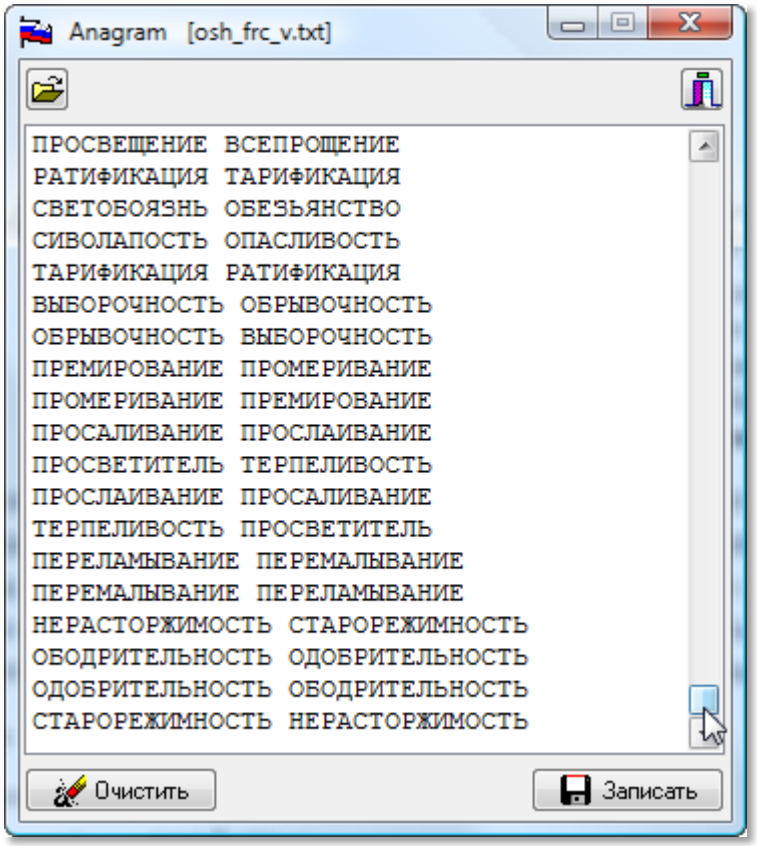


Рис. С13.7. А вы смогли бы найти такие анаграммы?!



Исходный код программы находится в папке **anagram**.

Самостоятельная работа



Напугались, небось, что самостоятельная работа никогда не кончится? А зря – на сегодня это всё!

Урок 14. Условный оператор *IF*

Если взять один кирпич, мало толку в нём,
Потому что из него не построишь дом.
Если пару кирпичей рядом положить,
Будет только две стены – неудобно жить.

Песенка Тыквы из мультфильма *Чиполлино*

С условными операторами будущий программист знакомится уже в раннем детстве благодаря педагогическому усердию родителей. Папа воспитывает кнутом, а мама – пряником: «**Если** ты будешь себя хорошо вести, **то** получишь большую сладкую конфету, а **иначе** не пойдёшь гулять во двор». В переводе на скупой язык *Delphi* мамины посулы выглядели бы так:

```
IF условие THEN результат1 ELSE результат2;
```

Здесь **условие** – «хорошее поведение»;
результат1 – «конфета»;
результат2 – «временное ограничение свободы».

Действует эта воспитательная конструкция – как в жизни: если условие соблюдено, то подопечный получает конфету, не соблюдено – принудительная домашняя отсидка.

Более лапидарный родитель, коим является отец семейства, изложил бы свои требования в более категоричной форме: «**Если** получишь двойку, **то** выпорю». Тут уж никакой надежды на конфеты и другие сладостные изделия, то есть альтернатива жёсткая: принёс из школы двойку – получил ремня, не принёс – избежал ремня:

```
IF условие THEN результат;
```

В школе порка запрещена, поэтому там так мало настоящих педагогов-мужчин.

А теперь о том же, но – серьёзно!

Условный оператор IF служит для того, чтобы изменять порядок выполнения операторов в программе в зависимости от некоторого логического условия. Он имеет две формы – *сокращённую* и *полную*:

```
IF условие THEN оператор;
IF условие THEN оператор1 ELSE оператор2;
```



Обратите внимание: весь оператор **IF** это единое предложение, поэтому ни перед словом **THEN**, ни перед словом **ELSE** точка с запятой не ставится!

Условие в этих записях – обычное логическое выражение, в котором используются знаки операций сравнения =, <, > и другие. Результатом логического выражения может быть константа *TRUE* (истина) или *FALSE* (ложь).

Действует условный оператор так. Если условие удовлетворено (значение выражения равно *TRUE*), то выполняется оператор после слова *THEN*. Если не удовлетворено, то для сокращённой формы **оператор** пропускается, а управление передаётся следующему за ним оператору. Для полной формы выполняется **оператор2**.

Например, если мы хотим найти *большее* из двух чисел, то легко сделаем это с помощью условного оператора:

```
var max, n1, n2: integer;

if n1 > n2 then max:= n1 else max:= n2;
```



Если после **THEN** и **ELSE** нужно выполнить несколько операторов, то следует использовать составной оператор **BEGIN – END**.

Если после зарезервированных слов **THEN** и **ELSE** следует условный оператор **IF**, то он называется **вложенным**. В этом случае каждый элемент **ELSE** относится к ближайшему оператору **IF**, который не имеет **ELSE**.

В этой записи **ELSE** принадлежит *второму*, вложенному, оператору **IF**:

```
IF усл1 THEN
   IF усл2 THEN оп1
   ELSE оп2;
```



Если **ELSE** должен входить в состав первого оператора **IF**, то используйте составной оператор:

```
IF усл1
THEN begin
  IF усл2 THEN оп1
end
ELSE оп2;
```

А в этой – *первому* оператору **IF**:

```
IF усл1 THEN оп1
ELSE
  IF усл2 THEN оп2;
```



Чтобы не запутаться, всегда выравнивайте отступом соответствующие друг другу элементы **IF THEN ELSE**.

Так

```
IF усл
THEN оп1
ELSE оп2;
```

Или так:

```
IF усл THEN оп1
ELSE оп2;
```

Семинар 14. Игра *Угадай число*, или Бинарный поиск

Ищите и обрящете.

Библейское

Попытка не пытка.

Фольклорное

Правила игры

Компьютер загадывает число от 1 до 100, а игрок должен угадать его за минимальное число попыток. Если число угадано, то компьютер поздравляет его и загадывает следующее. И так до тех пор, пока игроку не надоест. Если названное игроком число меньше задуманного, то компьютер сообщает ему о недоборе, если больше – о переборе.

Стратегия игры

Игра очень простая – и стратегия тоже не сложнее: нужно каждый раз называть число, находящееся посередине интервала оставшихся чисел. Например, на первом ходу игрок должен назвать число 100 : 2 = 50. Если угадал – сразу молодец! Если перебор, то диапазон уменьшается вдвое – 1..49. При недоборе – аналогично: 51..100. Далее действуем по заранее утверждённому плану, без особых затей. Легко подсчитать, что потребуется не более 8 ходов, чтобы угадать число от 1 до 100. Если вам этого мало, загадывайте число побольше.

Несмотря на кажущуюся простоту, наша стратегия является хорошим примером двоичного (бинарного) поиска. Действительно, с каждым ходом число претендентов уменьшается вдвое: вначале их было 100, после первого хода 50, затем 25 – 13 – 7 – 4 – 2 – 1. Обычно этот метод используют для поиска нужного элемента в упорядоченном массиве. Наше применение чуток нетрадиционное, но очень эффективное!

Программа для игры в *100*

Иногда задние мысли опережают передние.

Фольклорное

Если отвлечься от кнопок и других элементов интерфейса (Рис. С14.1), то в сухом остатке мы получим условный оператор, затаившийся в процедуре:

```
procedure TForm1.sbtTestClick(Sender: TObject);
var code: integer;
begin
  inc(popytka);
  lblPopytka.Caption:= 'Попытка '+inttostr(popytka);
  //otvet:= strtoint(txtOtvet.text);
  val(txtOtvet.text, otvet, code);
  if (code <> 0) or (otvet > MAX_NUMBER) or (otvet < 1) then begin
    application.MessageBox('Ваще мимо!!!', NAME_PROG, MB_OK);
    exit
  end;
  if otvet = zadum then
    lblMessage.Caption:='Поздравляю, угадал!'
  else
    if otvet > zadum then
      lblMessage.Caption:='Перебор!'
    else
      lblMessage.Caption:='Недобор!';
end;
```

С условными операторами вы легко разберётесь сами, а вот на что следует обратить пристальное внимание. Для ввода догадок пользователя мы установили **текстовое поле** 

txtOtvet. Игрок набирает число с клавиатуры и нажимает кнопку *Проверить* (**sbtTest**), после чего и выполняется процедура *TForm1.sbtTestClick*. Поскольку в текстовом поле находится строка, а нам нужно число, то мы должны позаботиться о её преобразовании. Первое, что приходит в голову, воспользоваться специально для этого предназначенной функцией преобразования строки в число *StrToInt*. И всё бы хорошо, но если игрок попадётся невнимательный или коварный, то сможет удумать недоброе и нажать кнопку *Проверить* вообще без всякого числа или с буквами вместо цифр. Функция *StrToInt*, увы, такого удара судьбы не переживёт, и вся программа закончит своё бренное существование полным коллапсом. Этого не надо бояться, но к этому надо готовиться, поэтому для преобразования строки в число безопаснее использовать процедуру *Val*, которая делает это более хитроумно - её не проведёшь! О результате конвертирования можно судить по значению переменной *code*: если оно равно **0**, значит, ошибок при вводе числа не найдено. Следует ехидно надсмеяться над криворуким игроком, который путается в клавиа-

туре, но такое дело по силам каждому, поупражняйтесь в язвительности сами.

Кнопка *Новая* **игра** её и начинает:

```
procedure TForm1.newGame;
begin
  randomize;
  lblMessage.Caption:='';
  txtOtvet.text:='';
  zadum:= GetRandom(1, MAX_NUMBER);
  popytka:=0;
  lblPopytka.Caption:= 'Попытка '+inttostr(popytka);
end;
```

А на сегодня это всё!

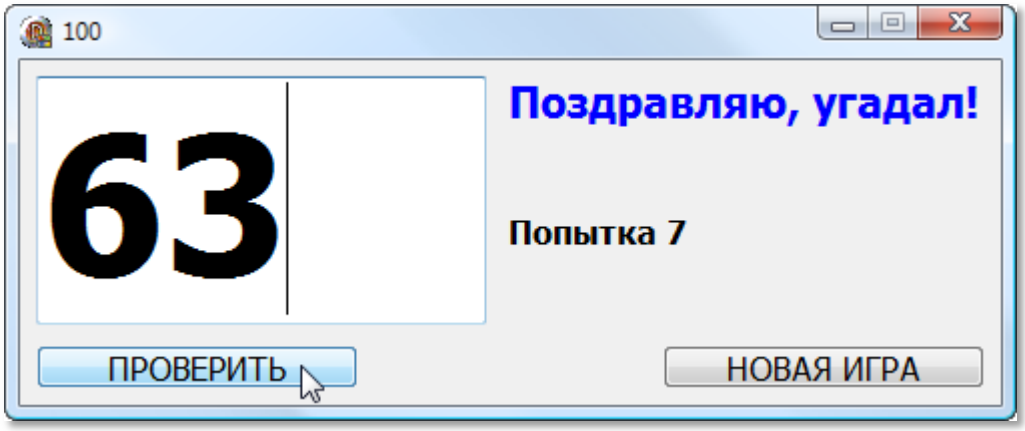


Рис. С14.1 Почему «угадал»? Я действовал методом научного тыка. Транспарантик нужно поправить!



Исходный код программы находится в папке **100**.

Урок 15. Оператор выбора CASE, или *Выбирай, но осторожно. Но выбирай!*

Направо пойдёшь – коня потеряешь;
налево пойдёшь – себя потеряешь;
прямо пойдёшь – и себя, и коня потеряешь!

Сказочный выбор

Условный оператор **IF** обычно применяют, когда вариантов выбора не очень много (два-три), в противном случае гораздо удобнее использовать оператор **CASE**, который делает запись более компактной и понятной.

В общем случае оператор **case** имеет вид:

```
Case выражение of
  Константа1: оператор1;
  Константа2: оператор2;
  . . .
  КонстантаN: операторN;
Else оператор;
End;
```



Точку с запятой перед **Else** можно ставить (а можно и не ставить). Оператор **Else** может вообще отсутствовать в записи оператора **case**.

Константы 1-N называются **константами выбора** или **метками случаев**. В пределах одного оператора **case** они не должны повторяться!

Выражение должно быть порядкового типа – целого, символьного, логического, перечисление или диапазон. Оно записывается между ключевыми словами **case** и **of** и называется также **селектором** или **индексом выбора**.

Иногда для нескольких констант следует выполнить одни и те же действия (оператор(ы)), тогда константы перечисляют через запятую:

```
2,3,5,7: print «Простые числа».
```

Точно так же можно указать и диапазон значений:

```
1..10: print «Первый десяток чисел».
```

Оператор должен быть **составным**, если для какой-либо константы выбора необходимо выполнить несколько операторов.

Закончим **пример** с простыми числами первого десятка.

```
Var number: 0..10;

Case number of
  2,3,5,7: Application.MessageBox('Простое число!', '', MB_OK)
else Application.MessageBox('Составное число!', '', MB_OK);
End;
```

Или без оператора **Else**:

```
Case number of
  2,3,5,7: Application.MessageBox('Простое число!', '', MB_OK);
  1,4,6,8,10:   Application.MessageBox('Составное   число!',   '',
MB_OK);
End;
```

Эта же смысловая конструкция с оператором *if* выглядела бы так:

```
if (number = 2) or (number = 3) or (number = 5) or (number = 7)
then Application.MessageBox('Простое число!', '', MB_OK)
else Application.MessageBox('Составное число!', '', MB_OK);
```

Без оператора **else** она была бы ещё ужасней!

Оператор **case действует** так.

- Значение **выражения** последовательно сравнивается со всеми **метками случаев**.

- При **равенстве** выполняются операторы, соответствующие совпавшей константе, а затем управление передаётся оператору, следующему за словом **End;** оператора **case**.

- Если значение выражения **не совпало** ни с одной константой выбора, то выполняется оператор после слова **else**, а затем управление передаётся оператору, следующему за словом **End;** оператора **case**.

Если ключевое слово **else отсутствует**, то ни один оператор в **case** вообще не выполняется.

Для нашего примера.

Если значение *number* равно **3**, то мы получим сообщение «Простое число!».
Если значение *number* равно **6**, то мы получим сообщение «Составное число!».

Если значение *number* равно **0**, то мы получим сообщение «Составное число!» в первой записи (с оператором **Else**) и не получим никакого сообщения во второй записи (без оператора **Else**).

Семинар 15. Лототрон, или Как помочь старушкам

А ты азартен, Парамоша!

Возглас генерала Чарноты

Кто людям помогает - тот тратит время зря…

Песенка Шапокляк, тоже старушки

Недавно в Москве наряд милиции штурмом взял врасплох группу старушек, игравших во дворе в русское бочковое лото, чем они (именно старушки, а не милиционеры!) завершили растление несовершеннолетней московской молодёжи, и без того отягощённой шопингом, допингом, дансингом и пирсингом. Протокол, подпись – и предводительница старушек была нещадно бита рублём (точнее, 500-ми рублями). Вот так, благодаря неусыпному бдению милиции, была пресечена преступная игорная деятельность полудюжины пенсионерок. Как говорил в подобных случаях Козьма Прутков, *иногда усердие превозмогает разум*.

И вообще, классическая литература, русский фольклор и выпуски киножурнала *Ералаш* определённо говорят нам о том, что именно наши, отечественные старушки самые вредные в мире. Всё самое-самое нужно беречь, поэтому мы всем скопом должны, как один, вступить в *Общество защиты старушек* и создать для них мобильное подпольное казино.

Как известно, в игре принимают участие, кроме собственно старушек, ещё и 90 бочонков, что мы и отразим константой:

```
const
  NUM_BOCH=90; // количество бочонков в игре
```

Для игры нам потребуется **мешок** для бочонков:

```
var
  Form1: TForm1;
  //мешок с бочонками:
  meshok: array[1..NUM_BOCH] of integer;
  //счётчик вытащенных бочонков 0..NUM_BOCH
  curBochonok: integer;
```

При запуске игры, чтобы не томить старушек долгим ожиданием, сразу начинаем новую игру:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  newGame();
end;
```

Для этого хорошенько перемешиваем бочонки в мешке (играем по-честному, без подвоха!):

```
procedure newGame();
begin
  shuffleMeshok();
  curBochonok:=0;
  Form1.lblNum.Caption:= '';
  Form1.lstProtocol.Items.Clear;
end;
```

Тут нам послужат 3 процедуры:

```
procedure shuffleMeshok();
var i: integer;
begin
    //восстановить номера бочонков в мешке:
    for i:= 1 to NUM_BOCH do
      meshok[i]:= i;
    //перемешать мешок:
    shuffle(meshok,NUM_BOCH);
end;

//ПЕРЕТАСОВАТЬ МАССИВ a[]
procedure shuffle(var a: array of integer; num: integer);
var
  iRnd: integer;
  i: integer;
begin
  Randomize;
  for i := 0 to num-1 do begin
    iRnd:= GetRandom(0, num-1);
    swap (a[i],a[iRnd]);
  end;
end;

//ПОМЕНЯТЬ ДВА ЧИСЛА
procedure swap(var v1, v2: integer);
var vT: integer;
begin
  vT := v1;
  v1 := v2;
  v2 := vT;
End;
```



Последнюю процедуру можно убрать, а числа переставлять непосредственно в процедуре **shuffle**, которую тоже можно сильно сократить, если перемешивать только конкретный мешок с бочонками.

Далее, по правилам игры, следует по одному (а не горстями!) доставать из мешка очередной бочонок и торжественно объявлять его. Не у всех старушек на это хватает чувства юмора, поэтому некоторые бочонки традиционно называют смешными словечками, что мы и используем в нашей программе, и именно для этого нам и пригодится оператор **case**!

```
//ВЫТАЩИТЬ НОВЫЙ БОЧОНОК
procedure TForm1.sbtGetClick(Sender: TObject);
var
  n: integer;
  s: string;
begin
  inc(curBochonok);
  //игра закончена:
  if curBochonok> NUM_BOCH then begin
     Application.MessageBox('Мешок пустой!', 'Лото', MB_OK);
  end
  else  begin
    n:= meshok[curBochonok];
    s:= inttostr(n);
    case n of
      1: s:= s+' - кол';
      3: s:= s+' - трое';
      7: s:= s+' - кочерга';
     10: s:= s+' - бычий глаз';
     11: s:= s+' - барабанные палочки';
     12: s:= s+' - дюжина';
     13: s:= s+' - чёртова дюжина';
     22: s:= s+' - уточки';
     44: s:= s+' - стульчики';
     55: s:= s+' - перчатки';
     66: s:= s+' - валенки';
     69: s:= s+' - туда-сюда';
     77: s:= s+' - вятские топоры';
     80: s:= s+' - бабка';
     90: s:= s+' - дед';
    end;
    lblNum.Caption:= s;
    if n < 10 then s:= ' '+s;
    if curBochonok <10 then s:= '  '+ inttostr(curBochonok)+'. '+s
    else                        s:= ' ' + inttostr(curBochonok)+'.
'+s;
    lstProtocol.Items.add(s);
    lstProtocol.TopIndex:=lstProtocol.Items.Count-1;
  end;
```

```
end;
```

Здесь, кроме шуток, нужно проверять, не пуст ли мешок. В жизни с мешком такое происходит редко, разве что все старушки разом уснут, ну а в программах всякое бывает.

Если в мешке удалось нащупать бочонок, вынимаем его и, если он именной, произносим его шутливое прозвище.

Для автоматического доставания бочонка на свет божий пред ясные старушечьи очи нам потребуется кнопка **sbtGet** с неопровержимой надписью *Достать бочонок*. А также ещё одна кнопка - **sbtNew**, - если вам приспичит начать новую игру, не закончив старую (игру, а не старушку):

```
procedure TForm1.sbtNewMouseDown(Sender: TObject; Button: TMouse-
Button;
  Shift: TShiftState; X, Y: Integer);
begin
  newGame();
end;
```

Через метку **lblNum** мы будем общаться со старушками, а в списке **lstProtocol** будем записывать вытащенные бочонки. Этим нужно озаботиться в первую очередь – был случай, когда одна вредная склеротическая старушка постоянно выигрывала уже на третьем ходу и никто не мог её официально убедить, что закрытые ей на карточке числа вообще из мешка не доставались. Она утверждала, что бочонки-таки вынимались, но потом скрытно смахивались обратно в мешок.

Ну, вот и всё: лототрон готов к эксплуатации (Рис. С15.1), можете зазывать старушек. Пароль: банка варенья. Портрет Штирлица в окне – казино провалено.

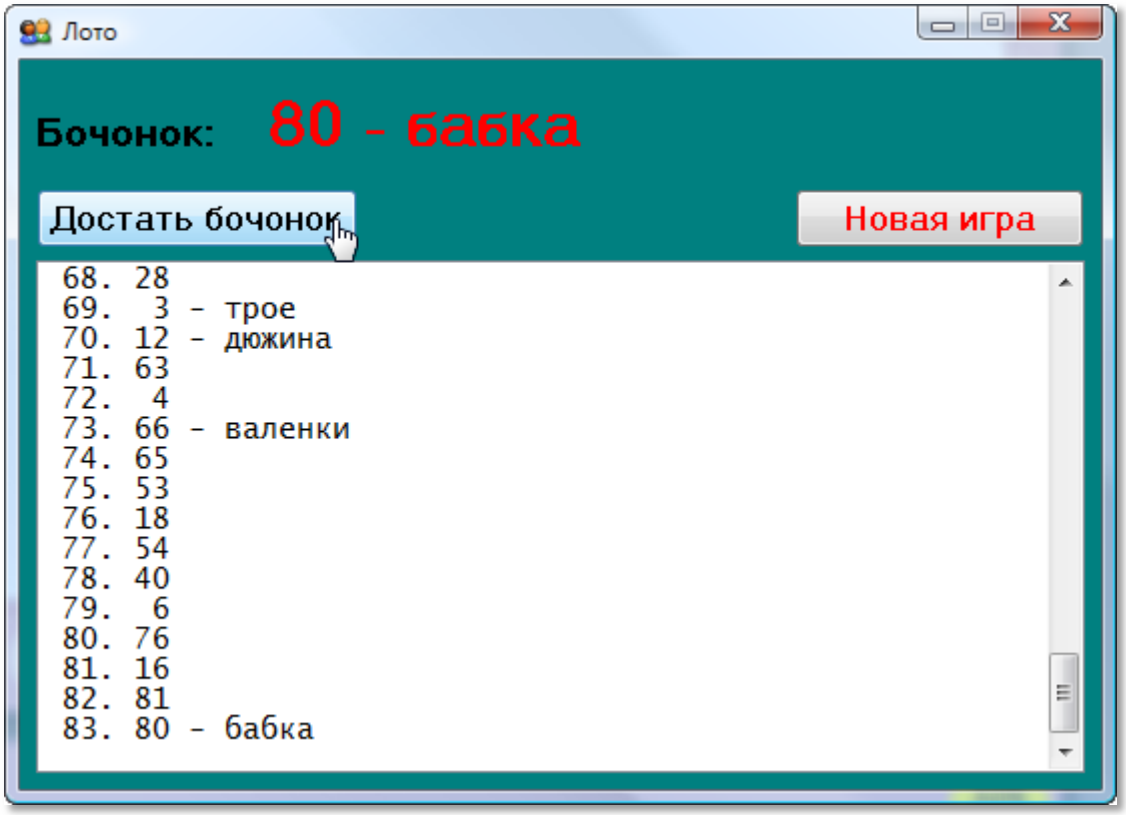


Рис. С15.1. А вот и бабка!



Исходный код программы находится в папке **Loto**.



Самостоятельная работа

Без труда не вынешь и рыбку из пруда.

Русская пословица

1. Вы можете уменьшить значение константы **NUM_BOCH** до 10 и тешить братьев своих меньших изучением названий чисел в пределах первого десятка, а потом – и далее.

2. Вместо чисел можно вытаскивать из мешка буквы и компьютеризировать изучение алфавита.

3. Если вы сумеете вытащить из мешка кролика, то бросайте программирование и ступайте в копперфильды!

Урок 16. Записи, или Рекорды в программировании

- Позвольте, товарищи,
у меня все ходы записаны!
- Контора пишет!

Из диалога Остапа Бендера
с одноглазым любителем шахмат

Записи, как и массивы, используются для хранения некоторого набора данных, но эти данные могут иметь разный тип (как вы помните, массив всегда содержит данные одного и того же типа). Эта особенность записей делает их очень удобными для объединения, группировки данных, относящихся к одному и тому же объекту. Например, карты в колоде имеют масть и вес (достоинство, значение); пуля в игре - координаты, скорость, направление движения, размеры; ученики в классе – имя, фамилию, возраст, место в классе, успеваемость.

Например, для каждой пули, которую выпускает из своего бластера герой, спасающий Галактику от монстров, мы должны ввести переменные (в реальной компьютерной игре переменных значительно больше):

```
var
  coordX, coordY: single;
  speed: single;
  direction: single;
  exist: boolean;
  width, height: integer;
```

Назначение и тип каждой переменной ясно следуют из названия, так что обсуждать мы их не будем.

Часто галактический герой настолько крут, что стреляет с сумасшедшей скоростью. Для нас это означает, что в полёте одновременно может находиться множество пуль и для каждой характеристики пули нужно заводить свой массив.

```
const
  MAX_BULLET=40; //максимальное число одновременно летящих пуль
var
  aCoordX: array [1..MAX_BULLET] of single;
  aCoordY: array [1..MAX_BULLET] of single;
  aSpeed: array [1..MAX_BULLET] of single;
  aDirection: array [1..MAX_BULLET] of single;
  aExist: array [1..MAX_BULLET] of boolean;
  aWidth: array [1..MAX_BULLET] of integer;
```

```
  aHeight: array [1..MAX_BULLET] of integer;
```

Согласитесь, в таком количестве массивов легко заплутать, а ведь они описывают в игре только пули! Вот если бы удалось объединить все данные о пуле в одной структуре, то нам потребовался бы только 1 массив:

```
var aBullet: array [1..MAX_BULLET] of TBullet;
```

Вот для таких программных казусов и служат записи.



В языке *C* записям соответствует тип данных **struct**.

Для **объявления** записей служит зарезервированное слово **Record**, которое и означает *запись*. После него перечисляются все данные с указанием их типа, а завершается объявление словом **end**. Объявим, например, запись с данными о пуле:

```
type
  TBullet= record
    coordX, coordY: single;
    speed: single;
    direction: single;
    exist: boolean;
    width, height: integer;
  end;
```

Все данные в записи называются её **полями**. Допускается любое количество полей. Они могут иметь следующие типы: числовой, символьный, строковый, логический, массив, множество, ранее определённый пользователем тип, **запись**.

Каждая запись имеет имя (идентификатор), в нашем примере это **TBullet** (идентификаторы всех типов в *Delphi* принято начинать с заглавной буквы **T**, хотя это и необязательно).

Поля записи различаются по типу и занимают в памяти компьютера разное число ячеек (соответственно типу поля), значит к ним нельзя обратиться по индексу, как к элементам массива, поэтому каждое поле должно иметь собственное имя, которое и позволяет обращаться к нему. Имя поля не должно повторяться внутри объявления записи. Идентификаторы для полей выбирают по тем же принципам, что и для переменных.



Обратите внимание, что имя поля отделяется от его типа двоеточием, а все поля отделяются друг от друга точкой с запятой.

Для полноты картины введём новую запись, содержащую координаты пули:

```
TCoord= record
  x,y: single;
end;
```



Если несколько полей записи имеют один и тот же тип, то их имена можно перечислять через запятую, а потом указать их общий тип:

```
x,y: single;
```

Теперь мы можем переписать наш тип **TBullet**:

```
TBullet= record
  //coordX, coordY: single;
  coord: TCoord;
  speed: single;
  direction: single;
  exist: boolean;
  width, height: integer;
end;
```



Впрочем, то же самое можно сделать и без объявления нового типа:

```
TBullet= record
  //coordX, coordY: single;
  coord: record
    x,y: single;
  end;
  speed: single;
  direction: single;
  exist: boolean;
  width, height: integer;
end;
```

Вернёмся к нашим полям. Запись **TBullet** содержит: поле **coord** типа *TCoord*, поля **speed**, **direction** типа single, поле **exist** логического типа и поля **width**, **height** целого типа. Чтобы **обратиться** к какому-либо полю записи, нужно указать имя записи, поставить точку, а затем написать имя нужного поля (такое «длинное» имя поля называется **составным**). Например, объявим переменную типа **TBullet**, содержащую сведения об одной пуле:

```
var
  bullet: TBullet;
```

Для **инициализации** переменной типа записи необходимо присвоить значения всем её полям:

```
  bullet.coord.x:= 10.0;
  bullet.coord.y:= 20.1;
  bullet.speed:= 100.0;
  bullet.direction:=270.0;
  bullet.exist:= true;
  bullet.width:= 32;
  bullet.height:= 32;
```

Для массива пуль эти обращения к полям будут выглядеть ещё хлеще:

```
  aBullet[1].coord.x:= 10.0;
  aBullet[1].coord.y:= 20.1;
  aBullet[1].speed:= 100.0;
  aBullet[1].direction:=270.0;
  aBullet[1].exist:= true;
  aBullet[1].width:= 32;
  aBullet[1].height:= 32;
```

Существенно облегчает набор составных имён «умный» текстовый редактор *Delphi*, который после того как вы поставите точку выводит список всех полей записи вместе с их типом (Рис. У16.1).



Рис. У16.1. Вести с полей

Достаточно кликнуть по нужному полю – и его имя появится после точки.

Так как каждое поле имеет определённый тип, то с ним можно обращаться точно так же, как и с любой переменной того же типа:

```
bullet.coord.x:= bullet.coord.y + 0.5;
bullet.speed:= bullet.coord.x * 10.0;
```

В последних версиях *Delphi* записи могут иметь не только поля, но и **свойства** и **методы** (в том числе и конструктор). Таким образом, по функциональности записи приблизились к классам.



Структуры в языке C# уже давно обзавелись этими атрибутами классов. А вот в языке *Java* ни структур, ни записей нет – только классы.

Однако при программировании следует учитывать и *отличия* между записями и классами:

- *Записи* не поддерживают наследование.
- *Записи* могут иметь вариативные секции, а *классы* не могут.
- *Запись* – значимый тип, а *класс* – ссылочный.
- *Записи* поддерживают перегрузку операторов, а *классы* не поддерживают.
- *Записи* создаются автоматически и потому не могут иметь конструктор по умолчанию (без параметров), а экземпляры *классов* необходимо создавать с помощью конструктора – по умолчанию или с параметрами.
- *Записи* не имеют деструктора.

Оператор над записями *With*

Вероятно, вы сильно раздосадованы необходимостью выписывать длинные имена полей и украшать их точками. Для облегчения столь жалкой участи программиста введено зарезервированное слово **with**, которое используется для сокращения длины имён при работе с полями одной записи. Если после слова **with** указать имя записи и затем слово **do**, то дальше можно записывать только имена самих полей:

```
with bullet do begin
  coord.x:= 10.0;
  coord.y:= 20.1;
  speed:= 100.0;
  direction:=270.0;
  exist:= true;
  width:= 32;
  height:= 32;
end;
```

Или для массивов:

```
with aBullet[1] do begin
  coord.x:= 10.0;
  coord.y:= 20.1;
  speed:= 100.0;
  direction:=270.0;
  exist:= true;
  width:= 32;
  height:= 32;
end;
```

Поскольку запись **TBullet** имеет в качестве поля запись **TCoord**, то для этого случая можно использовать конструкцию

```
with имя_записи do with имя_поля_записи do оператор,
```

которую часто записывают короче:

```
with имя_записи, имя_поля_записи do оператор.
```

Для нашего примера:

```
with bullet, coord do begin
  x:= 10.0;
  y:= 20.1;
  speed:= 100.0;
  direction:=270.0;
  exist:= true;
  width:= 32;
  height:= 32;
end;
```

или только для координат:

```
with bullet, coord do begin
  x:= 10.0;
  y:= 20.1;
end;

bullet := aBullet[1];
```

При этом всем полям первой переменной (**bullet**) будут присвоены значения полей второй переменной (**aBullet[1]**). Например, если

```
bullet.height:= 32;
aBullet[1].height:= 64;,
```

то после операции присвоения `bullet := aBullet[1];` поле **bullet.height** примет значение **64**.

Семинар 16. 21, или Игра в карты по-научному

Козыряй!

Козьма Прутков

Вы правы, мы будем играть в *Двадцать одно*, но в собственной интерпретации. Дело в том, что эта карточная игра настолько широко распространилась в народных массах, что и свиной грипп обзавидуется, поэтому в книгах и в Интернете можно найти невероятное количество вариантов этой игры. Например, в одной солидной книге предлагается играть двумя колодами карт, да ещё по 52 листа каждая! Но обычно используется универсальная российская колода в 36 карт, которая годится и для всеми любимого дурака и для множества других, не менее «дурацких» забав. К сожалению, в такой колоде отсутствуют карты от двойки до пятёрки, что неизбежно ведёт к переоценке ценностей. Так, валет приносит унизительные два очка, дама - три, король – четыре, а пять очков не даёт ни одна карта, что довольно неприятный изъян 36-листовой колоды! Поэтому правильнее вести игру одной колодой, но полной – в 52 карты. А вот классическую оценку веса карт придётся подправить: 11 очков за валета, 12 за даму и 13 за короля – слишком много для такой игры, как *21*. Даже две любые картинки могут привести к перебору и проигрышу партии, а на это мы пойти не можем, так что придётся разжаловать валета, даму и короля в двойку, тройку и четвёрку. Надеюсь, они это переживут с достоинством, тем более что в истории средних веков с ними обходились и того круче.

В реальной жизни в *21* обычно играют вчетвером, но в компьютерном варианте интереснее играть вдвоём, так как компьютер всё равно будет исполнять роли всех игроков, исключая вашу собственную. Карты всегда сдаёт компьютер (компьютер не человек – не обманет!). Сначала, как водится, по две карты, а затем, пользуясь своим скудным *IQ*, берёт ещё столько карт, сколько захочет. Вы можете видеть свои карты и карты компьютера (только с изнанки!) (Рис. С16.1).

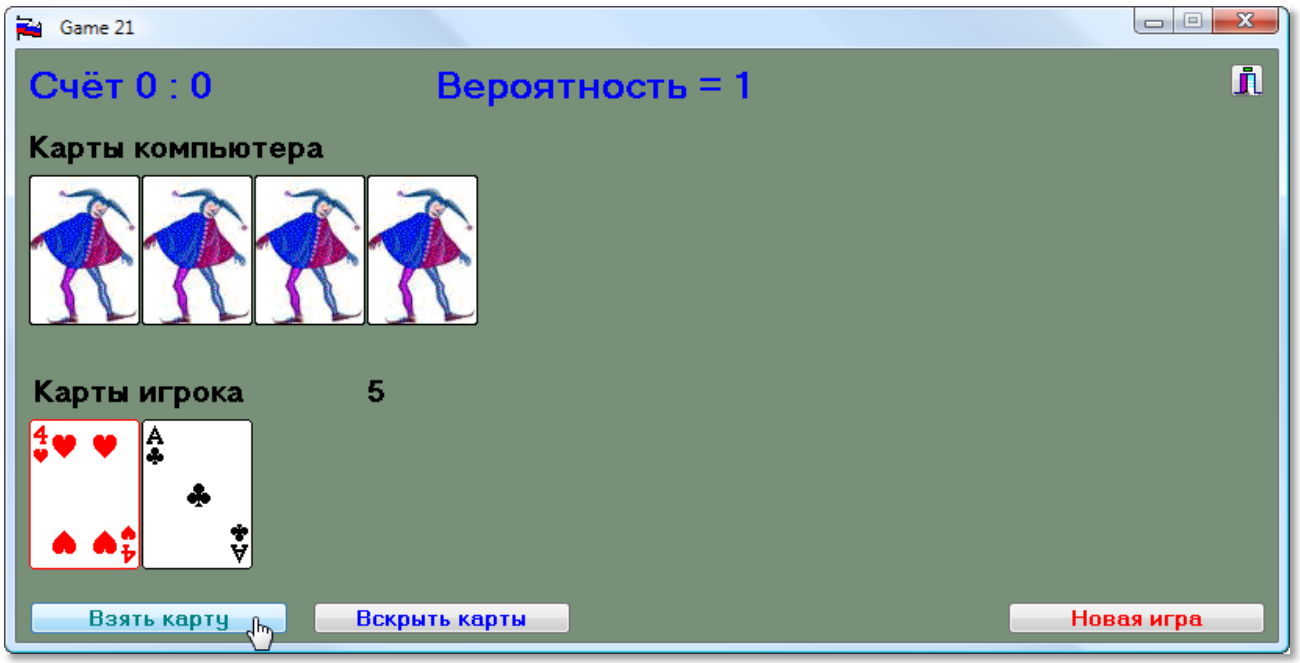


Рис. С16.1. К столу!

Для каждой партии берётся новая полная колода, которая тщательно тасуется.

Совершенно очевидно, что успех в игре зависит больше от слепой удачи, чем от трезвого расчёта, но мы попробуем увеличить «расчётливость» игры тем, что компьютер будет давать нам подсказки, насколько велика вероятность «взятия» ещё одной «хорошей» карты (чтобы по русской привычке не перебрать лишнего). Если бы мы знали карты противника, то и сами могли бы подсчитать свои шансы на шампанское. Но, во-первых, мы их не знаем (всё по-честному!), а, во-вторых, арифметические выкладки отвратят даже от самой увлекательной игры. А вот компьютер (не тот компьютер, который с нами играет!) знает все карты и считает с удовольствием. Примерно так.

Пусть банкомёт (у нас это компьютерный игрок) получил шестёрку и десятку. То есть у него 16 очков. Понтёр (человеческий игрок) – восьмёрку и девятку, и набрал 17 очков. В колоде осталось 52 – 4 = 48 карт. Нужно ли понтёру брать ещё одну карту? Конечно, если он понтёр вообще по жизни, а не только в картах, то непременно, но давайте подсчитаем, чему равна вероятность такого понта. Из 48 карт годятся 4 туза, 4 двойки, 4 тройки, 4 четвёрки и все 12 картинок. Остальные карты ведут к перебору. Таким образом, с вероятностью 28/48 (почти 60%) понтёр возьмёт карту удачно.



Случай с банкомётом рассмотрите самостоятельно.

Ближе к коду!

Для игры нам потребуются такие неизменные элементы *интерфейса* (Рис. С16.2):

- **Форма frmMain** с гордым названием программы **Caption**= Game 21. Флажок родной страны и прочие атрибуты патриотизма добавьте по вкусу.

- Для показа карт потребуются **компоненты** 

- от **Image1** до **Image22**. Попутно решим вопрос, а сколько карт может одновременно выйти из колоды? Поскольку после того как набрано 21 очко, дальше карты брать бессмысленно, то ограничимся этим числом. Максимум карт для него – 4 туза + 4 двойки + 3 тройки, что и составит как раз 21 очко. Таким образом, каждый игрок может взять не более 11 карт. Вот поэтому мы и должны разместить на форме две шеренги компонентов *TImage* по 11 в каждой.

- Счёт игры, вероятность удачной карты и количество очков у игроков мы будем выводить в **метки**, соответственно, **lblTablo**, **lblChance**, **lblCompScore**, **lblPlayerScore**.

- Ну, и чтобы было на что жать, 4 **кнопки**: **sbtClose**, **sbtCard**, **sbtOpen**, **sbtNewGame**.

- Наиболее важным из всех компонентов для нас является коллекция картинок 

ImlCards, доверху набитый весёлыми картинками (вы, конечно, догадались, что это карты). Под номером **0** здесь находится джокер, оправдывая тем своё место в общественной жизни, а затем по 13 треф, бубен, червей и пик – от туза до короля. Да с таким компонентом можно сотворить необъятное количество карточных игр!

Теперь перейдём к тыльной стороне приложения, то есть к коду.

Кода много, поэтому подробности пропускаем. Нас должны интересовать только *записи*!

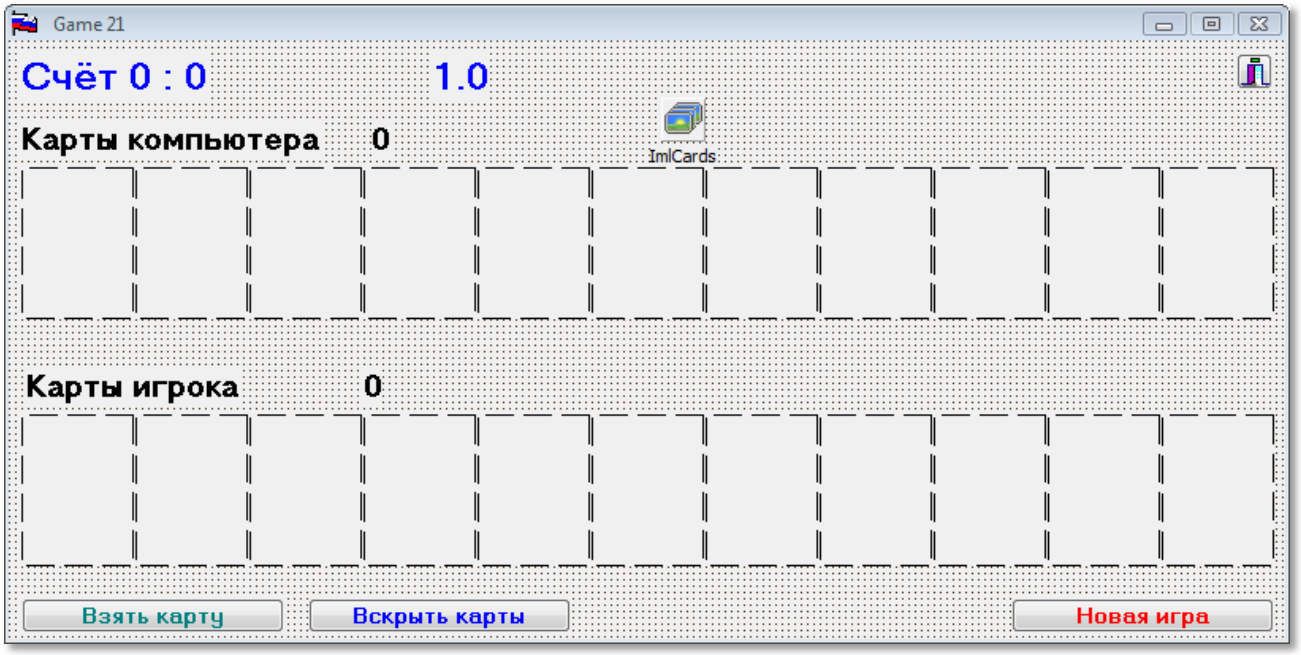


Рис. С16.2. Карточный интерфейс

Самый простой объект в карточной игре – это сама **карта**. Она имеет общеизвестные свойства - масть и вес, к которым мы должны присовокупить ещё 2. Свойство **img** хранит номер картинки в компоненте **ImlCards**. Действительно, на бумажной карте изображение уже напечатано, а для компьютерной карты мы должны хранить картинку в отдельном компоненте. И последнее свойство **exist** указывает, находится ли карта в колоде (*TRUE*) или у одного из игроков (*FALSE*).

```
Type
  //Масти карт:
  TSuit = (clubs, diamonds, hearts, spades);       // трефы (кре-
сти), бубны, червы, пики
  //Веса карт:
  TWeight= 1..10; //A,2,3,4,5,6,7,8,9,10,J,D,K

  //описание карты:
  TCard= record
    suit: TSuit;           //- масть
    weight: TWeight;       //- достоинство
    exist: boolean;        //= TRUE, если карта находится в колоде
    img: integer;          //- номер картинки с картой в ImlCards
  end;
```

Колода карт, конечно, тоже объект, но нам достаточно будет просто объявить массив из 52 карт:

```
const
```

```
  NUM_CARD=52; //- число карт в колоде

var
  pack: array[0..NUM_CARD] of TCard; // - колода карт
```

Ещё один объект (и в то же время субъект) игры – собственно **игрок**. Мы, правда, поступим неуважительно, и не станем перечислять фамилию, имя, отчество игрока, его место жительства и образование, которые больше важны для суда, чем для компьютерной игры. Оставим только те свойства, которые необходимы ему как игроку.

Так как игрок связан с внешним миром через компоненты, то внесём их в список: **frm** – это главная форма программы, **lbl** – метка для вывода очков, **img** – компоненты *TImage*, в которых мы будем рисовать карты игрока. Вообще говоря, без этих свойств можно и обойтись, но тогда нам придётся использовать «внешние» по отношению к объекту компоненты, хотя, по сути, они принадлежат ему, а не форме.

```
  TPlayer= record
    frm: TfrmMain;
    lbl: TLabel;
    //компоненты TImage для вывода карт:
    img: array[1..11] of TImage;

    nCard: integer; // число карт у игрока
    //номера карт из колоды, которые находятся у игрока
    idCards: array[1..11] of integer;
    score: integer; //число очков
    win: integer;   //число побед

    procedure setScore(sc: integer);
  end;
```

Ещё каждый игрок должен знать, сколько у него карт (**nCard**) и какие именно карты (массив **idCards**), а также набранные очки в партии (**score**) и число побед во всех играх (**win**).

Обратите внимание на процедуру **setScore** – запись может иметь не только поля, но также процедуры и функции для их обработки. Для данной программы такая возможность не очень нужна, но зато мы написали почти класс игрока, а не банальную запись, то есть вплотную приблизились к объектно-ориентированному программированию, которое станет предметом нашего пристального внимания на следующем уроке.

Осталось объявить пару переменных **comp** и **player** типа **TPlayer**:

```
var
```

```
  comp, player: TPlayer;
  nCard: integer; //число карт, вышедших из колоды
```

и можно начинать карточные баталии!

Здесь **nCard** – это общее число карт у игроков. Эта переменная введена только для удобства, поскольку её значение равно сумме значений переменных *comp*. *nCard* и *player*. *nCard*.

При запуске программы мы окрашиваем форму и все компоненты в какой-нибудь цвет. В азартных играх это обычно тёмно-зелёный, но я предпочёл более мерзкий оттенок.

```
//##############################################
//#              СОЗДАТЬ ФОРМУ                 #
//##############################################
procedure TfrmMain.FormCreate(Sender: TObject);
var i: integer;
  tempC: TComponent;
begin
  clrForm:= rgb(120,144,120);
  //окрасить форму:
  frmMain.Color:= clrForm;

  //для всех компонентов Image установить свойства:
  for i:= 1 to 22 do begin
    tempC:= FindComponent('Image'+ inttostr(i));
    with TImage(tempC) do begin
      //закрасить фон, чтобы не было видно уголков карт:
      Canvas.Brush.Color:=  clrForm;
      Canvas.FillRect(ClientRect);
      //прозрачность:
      transparent:= true;
      //компоненты  TImage:
      if i < 12 then comp.img[i]:= TImage(tempC)
      else           player.img[i-11]:= TImage(tempC);
    end;
  end;
  comp.frm:= frmMain;
  player.frm:= frmMain;
  comp.lbl:= lblCompScore;
  player.lbl:= lblPlayerScore;
  comp.win:=0;
  player.win:=0;
end;
```

Тут же мы передаём игрокам компоненты *TImage*, *TLabel* (для вывода очков) и форму - теперь игроки могут сами рисовать свои карты и печатать очки. И, наконец, обнуляем число побед: начальный счёт 0 : 0.

После создания формы мы начинаем *первую* игру:

```
//#################################################
//#      АКТИВАЦИЯ ФОРМЫ - ПЕРВАЯ РАСКЛАДКА       #
//#################################################
procedure TfrmMain.FormActivate(Sender: TObject);
begin
  delay(100);
  //начинаем новую игру:
  newGame();
end;

//#######################################################
//#                НАЧАТЬ НОВУЮ ИГРУ                    #
//#######################################################
procedure newGame();
begin
  //не показываем очки компьютера:
  comp.lbl.Visible:= false;
  comp.setScore(0);
  comp.nCard:= 0;
  player.setScore(0);
  player.nCard:= 0;
  nCard:=0;
  raskladka();
end;
```

Пока у игроков нет ни карт, ни очков, но они скоро появятся!

```
//#############################################
//#             РАЗЛОЖИТЬ КАРТЫ                #
//#############################################
procedure raskladka();
var i,j,n : integer;
  ch: single;
  ich: integer;
begin
  sound('shuffle');
  n:= 1;
  //восстановить колоду:
  for j:= 1 to 4 do             // 4 масти
     for i:= 1 to 13 do begin  // по 13 карт в масти
        pack[n].suit:= TSuit(j);
        //вес карт 1,2,3,4,5,6,7,8,9,10,2,3,4
        if i < 11 then pack[n].weight:= TWeight(i)
        else           pack[n].weight:= TWeight(i-9);
        pack[n].img:= (j-1)*13+i;
        pack[n].exist:=true;
        inc(n);
     end;

  //перемешать колоду:
```

```
  Shuffle(pack);//

  for i:= 1 to 11 do begin
    //для компьютера:
    //восстановить свойства компонентов:
    with comp do begin
      img[i].Visible:= false;
      img[i].Enabled:= false;
    end;

    //для игрока:
    with player do begin
      img[i].Visible:= false;
      img[i].Enabled:= false;
    end;
  end;

  //нарисовать карты раскладки на компонентах Image:
  for i:= 1 to 2 do begin
    //для компьютера:
    //восстановить свойства компонентов:
    with comp do begin
      img[i].Visible:= true;
      img[i].Enabled:= true;
      img[i].Cursor:= crNo;
      //стереть предыдущую картинку:
      img[i].Canvas.FillRect(frm.ClientRect);
      //frm.ImlCards.Draw(img[i].Canvas, 0, 0, pack[i].img, true);
      //не показываем карту:
      frm.ImlCards.Draw(img[i].Canvas, 0, 0, pack[0].img, true);
      setScore(score + pack[i].weight);
      idCards[i]:= i;
    end;
    pack[i].exist:= false;

    //для игрока:
    with player do begin
      img[i].Visible:= true;
      img[i].Enabled:= true;
      img[i].Cursor:= crNo;
      img[i].Canvas.FillRect(frm.ClientRect);
      frm.ImlCards.Draw(img[i].Canvas, 0, 0, pack[i+2].img, true);
      setScore(score + pack[i+2].weight);
      idCards[i+2]:= i+2;
    end;
    pack[i+2].exist:= false;
    delay(10);
  end;

  nCard:=4;
  comp.nCard:=2;
  player.nCard:=2;
```

```
  n:= test;
  if n<> 0 then newGame();

  //компьютер добирает карты:
  repeat
    ch:= calcChance(comp.score, false);
    ich:= round(ch*1000);
    if getRandom(0,1000) <= ich*1000 then begin//берём еще карту
      inc(nCard);
      inc(comp.nCard);
      pack[nCard].exist:= false;
      comp.idCards[comp.nCard]:= nCard;
      comp.setScore(comp.score + pack[nCard].weight);
      comp.img[comp.nCard].Visible:= true;
      comp.img[comp.nCard].Enabled:= true;
      comp.img[comp.nCard].Canvas.FillRect(comp.frm.ClientRect);
      //comp.frm.ImlCards.Draw(comp.img[comp.nCard].Canvas,  0,  0,
pack[nCard].img, true);
      //не показываем карту:
      comp.frm.ImlCards.Draw(comp.img[comp.nCard].Canvas,   0,   0,
pack[0].img, true);
      //если комп. набрал 21 очко, то дальше карты не брать!
      if comp.score=21 then break;
    end
  until getRandom(0,50)+calcChance(comp.score, false)*100 < 70;
  //until calcChance(comp.score, false) < 0.35;

  if comp.score <>21 then begin
     n:= test;
     if n<> 0 then newGame();
  end;
end;
```

Сначала мы восстанавливаем колоду, то есть раскладываем все карты по мастям и весам (учитываем, что валет даёт 2 очка, дама – 3, король - 4), а затем хорошенько её тасуем.

Масти карт в этой игре не учитываются, но вдруг вы захотите сделать совсем другую игру.

А вот теперь, как и принято, раздаём по две карты игрокам – компьютеру и человеку. Тут уж появляются и набранные очки, и карты на форме (компьютер не видит карты игрока вообще, а игрок видит только рубашки карт).

Самый смак для нас в этой «наукоёмкой» процедуре - в наборе карт компьютером. Так как мозгов у компьютера нет, то нужно вложить ему хоть какой-нибудь *IQ*, чтобы он не выглядел уж полным дураком. Но – парадокс! – вбить разум в его железную голову должны тоже вы.

После первой раздачи карт у игроков оказалось по 2 карты, которые могут дать максимум 20 очков, поэтому эти 2 строчки для игры по нашим правилам лишние, но могут пригодиться, если вы вернётесь к оригинальным правилам.

```
  n:= test;
  if n<> 0 then newGame();
```

Перед компьютером стоит непростая задача: брать иль не брать – вот в чём вопрос! Он оценивает шансы на благополучный исход

```
ch:= calcChance(comp.score, false);
```

и, исходя из него, принимает решение:

```
    ich:= round(ch*1000);
    if getRandom(0,1000) <= ich*1000 then begin//берём еще карту

…
```

А затем – опять проблема: брать ещё карты или удовлетвориться набранными очками:

```
  until getRandom(0,50)+calcChance(comp.score, false)*100 < 70;
  //until calcChance(comp.score, false) < 0.35;
```

Именно в этих местах кода и спрятан весь интеллект компьютера как игрока. Немудрёный, конечно, интеллект, но у меня, например, почти такой же: игра обычно заканчивалась ничьей. Ухудшить алгоритм игры компьютера смог бы и сам компьютер, а вот улучшить сможете только вы – если захотите проиграть железному другу.

Технологию подсчёта шансов мы уже подробно рассмотрели:

```
//ВЫЧИСЛИТЬ ВЕРОЯТНОСТЬ ПРИХОДА ХОРОШЕЙ КАРТЫ
function calcChance(sc:integer; print: boolean): single;
var i:integer;
  n:integer; //осталось карт в колоде
  p:integer; //подходящих карт в колоде
  ch: single;
begin
  n:=0;
  p:=0;
  for i:= 1 to 52 do begin
    if pack[i].exist=true then begin
      if sc+pack[i].weight <= 21 then inc(p);
      inc(n);
```

```
    end;
  end;
  ch:= p / n;
  if print then frmMain.lblChance.Caption:= 'Вероятность = '+
leftstr(floattostr(ch),5);
  result:= ch;
end;
```

Кстати говоря, эта же функция будет подсказывать и вам, когда придёт ваш черёд брать карты. Вот вам прекрасный случай показать, что ваш *IQ* выше. Вы можете учитывать подсказку, можете не учитывать, - дело ваше, но ежели захотите взять карту, то нажмите кнопку с надписью, визуализирующей ваше желание. Удовлетворившись достигнутым, нажмите кнопку *Вскрыть карты* (если вам повезло, и вы ещё не проиграли!). Всё – участь ваша решена: либо вы победите, либо проиграете, либо будет ничья.

Продолжайте игру до тех пор, пока не сломите сопротивление компьютера и не подавите его своим интеллектом. Если этого вдруг почему-то не случится (игра не заладилась или избыток пятен на Солнце), после каждого расклада нажимайте кнопку *Новая* игра. Время от времени компьютер перебирает очки, а вы со своими скудными двумя картами – никогда. Ну, а как поправите статистику, опять переходите на честную игру. Вот в этом и заключается главное преимущество игры с компьютером: он вас не только не отдубасит, но даже не обидится (Рис. С16.3).



Рис. С16.3. Счастливые мгновения игры: человек побеждает бездушную машину!

Подведём итоги

Применение в программе **записей** для группировки данных таких объектов, как карта и игрок, вкупе с оператором **with** позволили нам написать вполне осмысленный и легко читаемый код. Более того, мы добавили в запись процедуру для манипулирования свойствами объекта: каждый игрок может самостоятельно выводить на экран набранные им очки:

```
procedure TPlayer.setScore(sc: integer);
begin
  score:= sc;
  lbl.Caption:= inttostr(score);
end;
```

Это гораздо удобнее, чем писать ещё две процедуры - для каждого игрока свою. Таким образом, наш объект не только всё своё носит с собой, но и может нажитым добром пользоваться. На следующем уроке мы познакомимся с классами, которые доведут объекты до совершенства.



Исходный код программы находится в папке **21**.



Самостоятельная работа

Программа позволяет улучшить не только интеллект компьютера, но и свой собственный. Для этого вы можете:

- Улучшить функцию для оценки вероятности.
- Усилить игру компьютера.
- Изменить программу так, чтобы банкомётом мог быть не только компьютер, но и человек.
- Придумать другие варианты игры в *21* или запрограммировать покер.
- Решить парочку кроссвордов.

Урок 17. Трёхголовое ООП, или КЛАССный руководитель

Важнее ООП для человечества только ООН.

Из несостоявшейся беседы
Дмитрия Медведева с Крисом Касперски

Вот мы и добрались до сокровенных знаний современного программирования - классов, объектов и загадочного ООП! Хотя сами того не ведая, мы уже вовсю орудовали классами. Не верите, тогда создайте новый проект и посмотрите код:

```
type
  TForm1 = class(TForm)
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;
```

Оказывается тип **TForm1** главной формы, это и есть *класс*! А далее объявлена переменная **Form1** этого класса, которая и есть тот самый объект-форма, которую мы видим на экране. Действительно, запустите проект – и форма во всей красе предстанет перед вами. Если дальше посмотреть код модуля Unit1, то мы не найдём ни одной строчки. А кто и где создаёт форму? Нажмите пункт **View Source** в меню **Project**, и вы увидите файл проекта:

```
program Project1;

uses
  Forms,
  Unit1 in 'Unit1.pas' {Form1};

{$R *.res}

begin
  Application.Initialize;
  Application.CreateForm(TForm1, Form1);
  Application.Run;
end.
```

Форму создаёт приложение, которое само является объектом. В модуле **Forms** мы отыщем полное описание класса приложения

```
TApplication = class(TComponent) …
```

Оно очень длинное, так что полностью его цитировать здесь мы не будем. Далее объявлена и переменная типа **TApplication**, она и является нашим приложением:

```
var
  Application: TApplication;
```

Теперь обо всём подробнее.

Классы и объекты

Класс (или объектный тип) – это структура, сходная с записью, она содержит **поля**, **методы** и **свойства**.

Как **объявляется** класс, мы уже видели:

```
type
  TForm1 = class(TForm)
  private
    { Private declarations }
  public
    { Public declarations }
  end;
```

Имя класса, как и других типов, принято начинать с заглавной буквы **T** (от слова *type*). Затем пишут знак равенства и зарезервированное слово **class**. В теле объявления перечисляются поля, свойства и методы класса и завершается определение словом **end**.

Давайте создадим свой собственный класс, например, класс игрока. Пусть у него будет имя и процедура, которая может это имя напечатать:

```
type
  TPlayer = class
    name: string;
    procedure printName();
  end;

procedure TPlayer.printName();
begin
  lbl.Caption:= name;
```

```
end;
```

Вы, конечно, обратили внимание на то, что в объявлении класса формы после слова **class** в скобках указан тип **TForm**, который также является классом. А от него образован класс нашей формы. Такой способ создания новых классов от уже имеющихся называется наследованием, и о нём мы поговорим позже. Главное, что мы должны знать уже сейчас, - каждый новый класс создаётся на базе уже существующего! Так почему же в нашем объявлении скобки и исходный класс отсутствуют? Только потому, что наш класс образован от базового класса **TObject**, родоначальника всех других классов в *Delphi*, который можно не указывать, – только для краткости, но с некоторой потерей смысла. Если вы хотите быть дотошными, то объявите наш класс так:

```
TPlayer = class(TObject)
  name: string;
  procedure printName();
end;
```

Получилось то же самое, но уже и комар носа не подточит.

Классы обычно **описывают** в разделе *interface* модуля, там же, где и все остальные типы.

Внутри объявления класса, в каждом разделе последовательность объявления членов класса должна быть такой:

- - объявление полей,
- - объявление методов,
- - объявление свойств.

Поля, свойства и методы

Поля в классах ничем не отличаются от таковых в записях. Это разнообразные данные, которые характеризуют объект с лучшей стороны. В нашем случае поле **name** хранит имя игрока и поэтому имеет строковый тип. Ясно, что тип данных может быть каким угодно. В отличие от полей записи, мы можем упрятать поля в раздел *private* и тем оградить их от посягательств легкомысленного программиста. Тогда доступ к ним будет ограничен только тем модулем, в котором находится описание класса. Из другого модуля получить доступ к полям класса можно будет только через методы класса.

Методы – это обычные процедуры и функции, которые описаны внутри класса и служат для операций с его полями. Обычно для каждого поля необходимы два метода: один записывает данные в поле (процедура), другой считывает их оттуда и передаёт заинтересованным лицам (функция). Естественно, методы могут выполнять и другие обязанности, например, в нашем классе метод *printName* печатает имя игрока.

Методы могут быть статическими, динамическими, виртуальными и абстрактными. По умолчанию все методы являются **статическими**, чем мы и удовлетворимся.

Также классы могут иметь особые методы – **конструктор** и **деструктор**.

Класс – это только описание свойств, параметров, характеристик и поведения некоторого объекта. А чтобы создать сам объект по этому описанию, необходимо объявить переменную объектного типа, которая и является **экземпляром класса**, или **объектом**. Наглядно класс можно представить себе как чертёж, а объект – как деталь, изготовленную по этому чертежу.

```
var
  player1, player2: TPlayer;
```

У всех объектов одного класса поля одинаковые, но они могут иметь разные значения. Например, мы можем задать имя игроков:

```
player1.name:='Ivan';
player2.name:='Peter';
```

Каждый игрок, как и в жизни, теперь имеет своё собственное имя.

Эта операция присваивания значения полям объекта легко осуществима в пределах модуля с описанием класса, то есть доступ к ним ничем не ограничивается. Если вам не нравится такая доступность полей, то используйте свойства.

Методы объектов одного класса также одинаковы, но результат их действия зависит от значения *их* полей. Например, описанная нами процедура печати имени игрока, опубликует как раз его имя:

```
player1.printName; → Ivan
player2. printName; → Peter
```

Конструктор и деструктор

У каждого класса могут быть (а могут и отсутствовать) два специальных метода.

Конструктор создаёт и инициализирует объект. Выглядит он, как обычная процедура, но его объявление начинается со слова **constructor**. В *Delphi* его принято называть **Create**.



В классе может быть и более одного конструктора, но обычно хватает и одного.

Деструктор уничтожает ранее созданный объект и освобождает занятую им память. Его объявление начинается со слова **destructor**. В *Delphi* его принято называть **Destroy**.

Добавим эти методы в наш класс:

```
TPlayer = class(TObject)
  name: string;
  constructor create;
  destructor destroy;
  procedure printName();
end;
```

Для создания объекта, необходимо вызвать метод **create** класса (не объекта, ведь его ещё нет!).

```
player1:= TPlayer.create;
player2:= TPlayer.create;
```

При этом для нового объекта выделяется память, а все его поля автоматически обнуляются. Затем выполняются операторы конструкторов, которые служат для инициализации объекта (задания начальных значений его полям).



Если объект создан, то его метод **create** не создаст нового объекта, а только выполнит операторы, находящиеся в конструкторе.

В конструкторе первым делом нужно инициализировать поля класса-предка, вызвав **его** метод **create**:

```
Constructor TPlayer.create;
Begin
```

```
  Inherited create;
  . . .
end;
```

А вот для уничтожения созданного объекта используется *его* метод **destroy**:

```
player1:= TPlayer.destroy;
player2:= TPlayer.destroy;
```

На практике для уничтожения объекта вызывается метод **free** (доставшийся ему от предка – класса **TObject**):

```
player1:= TPlayer.free;
player2:= TPlayer.free;
```



Этот способ более безопасен, так как метод **free** сначала убеждается, что объект существует, и только потом вызывает метод **destroy** для уничтожения объекта и освобождения памяти.

В деструкторе последним оператором должен быть метод **destroy** класса-предка:

```
Constructor TPlayer.create;
Begin
  . . .
  Inherited destroy;
end;
```

Даже если в вашем классе отсутствуют объявления конструктора и деструктора, всё равно объекты создаются и уничтожаются теми же самыми способами.

Принципы ООП

Не вешать нос, гардемарины!

Лихая песня

Как Земля покоится на трёх слонах, так и ООП прочно стоит на трёх принципах. Они не очень просты для понимания, особенно при отсутствии опыта в программировании, но не делайте из ООП культа: на первых порах вы можете конструировать свои объекты с помощью классов, написанных достаточно просто, – и всё будет прекрасно работать!

Инкапсуляция. Свойства

Главное преимущество классов перед другими структурами в том, что они включают в себя не только данные (поля), но и функции и процедуры, оперирующие этими данными (методы). Как мы убедились, данные, хранящиеся в полях, легко могут быть изменены, потому что к ним имеется прямой доступ (все поля должны быть объявлены в разделе *private*, но это не убережёт их, если несколько классов описаны в одном модуле). Значит, случайно или намеренно пользователь класса, злоупотребив своей бесконтрольностью и безнаказанностью, может придать полям опасные для работы значения (например, укажет, что игроку 16 лет). Более того, считается, что пользователь класса вообще не должен знать, какие данные и какого типа в нём имеются. Это не только повышает надежность программирования, но и скрывает от посторонних глаз детали реализации класса. Для пользователя класс должен быть неким подобием чёрного ящика, состояние которого (данные) можно изменять только через его методы. Таким образом, *принцип инкапсуляции* заключается в том, что класс объединяет данные и методы, изменяющие эти данные, причём пользователь не имеет возможности напрямую изменять эти данные, а должен вызывать соответствующие методы класса.

Для упорядочения доступа к полям служат **свойства**, которые включают методы для чтения и записи данных в поля.

Свойство класса **объявляется** с помощью ключевого слова **property**, за которым следуют: имя свойства, через двоеточие его тип, слово **read**, после которого указывается имя поля или метод чтения, слово **write** с последующим именем поля или методом записи:

```
property name: string read … write …;
```

Если после слов **read** и **write** стоит имя поля, это значит, что к нему разрешён прямой доступ. Если указаны имена методов, то чтение и запись в поле будут производиться только через них.



Свойства должны быть объявлены в разделе **public**, иначе они будут недоступны.

Имя поля обычно совпадает с именем свойства, но начинается с заглавной буквы **F** (от слова *Field – поле*).

```
TPlayer = class(TObject)
private
```

```
    FName: string;
    function getName: integer;
    procedure setName(s: string);

  public
    property name: string read getName write setName;
    constructor create;
    destructor destroy;
    procedure printName();
  end;
```

Процедура записи имеет один параметр того же типа, что и свойство. Принято начинать имя процедуры записи с префикса **set**. Раздел **write** в объявлении свойства может и отсутствовать, тогда пользователь не сможет его изменить.

Функция чтения не имеет параметров и возвращает значение того же типа, что и свойство. Принято начинать имя функции чтения с префикса **get**.

Сами методы могут быть такими.

```
procedure TPlayer.setName(s: string);
begin
  if s = '' then FName:= 'Аноним';
  else if s = 'Дурак' then FName:= 'Сам дурак';
  else FName:= s;
end;

function TPlayer.getName: string;
begin
  result:= FName;
end;
```

Как вы видите из этого незамысловатого примера, можно без опаски позволять считывать имя игрока, но следует пресекать попытки опорочить его доброе имя. Тогда объявление класса можно сделать более либеральным:

```
  TPlayer = class(TObject)
  private
    FName: string;
    procedure setName(s: string);

  public
    property name: string read FName write setName;
    constructor create;
    destructor destroy;
    procedure printName();
  end;
```

Теперь вы можете изменять значение поля **FName**:

```
Player1.setName('Ivan');
Player2.setName('Peter');
```

Или считывать его:

```
Var s1, s2: string;

S1:= Player1.getName;
S2:= Player2.getName;
```

Для второго объявления класса:

```
S1:= Player1.FName;
S2:= Player2.FName;
```

Ксати говоря, подобная сокращённая запись допустима и для первого объявления. Разрешена даже такая вольность:

```
Player1.FName:= S1;
Player2.FName:= S2;
```

Казалось бы, мы так и не запретили несанкционированный доступ к полю имени игрока, но на самом деле при таких операциях всегда будут вызываться методы свойства и поле не пострадает от вандализма пользователя.

Если вы предполагаете использовать классы многократно или передавать их другим программистам, то всегда применяйте свойства вместо полей. Именно так спрятаны поля у всех компонентов *Delphi*.

И обязательно весь код класса размещайте в отдельном модуле, иначе толку от свойств не будет никакого!

Наследование

Наиболее полезная возможность ООП, которой вам придётся пользоваться в любом случае. В этом вы уже могли убедиться: любой ваш класс наследует классу **TObject**. Что нам это даёт? – Очень многое! Как только вы создали новый класс от родительского класса (класса-предка), вы тут же «безвозмездно, то есть даром» получили в своё полное распоряжение наследство в виде его полей, свойств и методов. Вспомните, как мы вызвали для уничтожения игрока (к счастью, только объекта) метод **free**, которого и в помине нет у нашего класса **TPlayer**. Вы легко можете полюбоваться своим богатством, если после имени объекта поставите точку. Услужливый

редактор текста выведет подсказку с перечнем всех имеющихся у вашего объекта методов и свойств (Рис. У17.1).

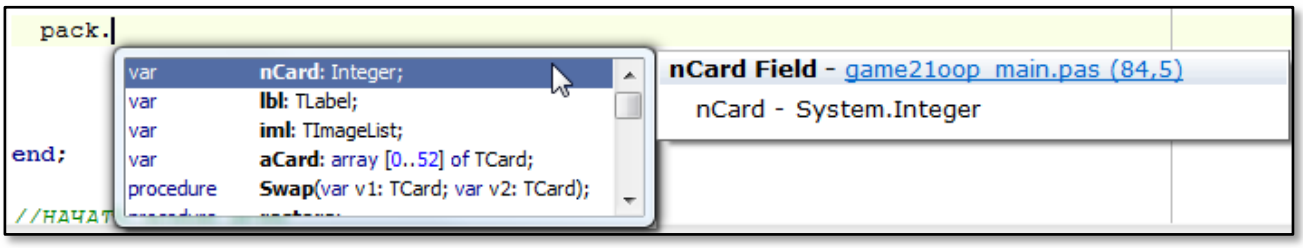


Рис. У17.1. Объектная подсказка

Тут же вы можете убедиться, что и все ваши усилия не пропали даром: ваш объект имеет и новые свойства и методы, которых не было у базового класса! То есть, используя своего родителя (вполне жизненная ситуация) в корыстных целях, вы можете создавать нужный вам объект, добавляя новые поля, свойства и методы. Например, у вас есть класс игрока **TPlayer**, но вам потребовался точно такой же, но более крутой. Вам не нужно заново писать новый класс, достаточно просто создать его из уже имеющегося:

```
TCoolPlayer= class(TPlayer)
 private
   cool: integer;
 public
   constructor create;
   destructor destroy;
   function getCool: integer;
   procedure setCool(c: integer);
 end;
```

Новый игрок, точно так же будет иметь имя и сумеет его написать, как и его предшественник, но он будет обладать и крутизной, которой вы сможете манипулировать с помощью новых методов.

Создаём нового игрока:

```
Var
  coolPlayer: TCoolPlayer;
. . .
coolPlayer:= TCoolPlayer.create;
```

- и вы можете крушить и громить всё вокруг – или спасать Галактику!

И ещё одна приятная возможность, которую даёт нам в руки ООП, - вы можете насолить родительскому классу и изменить (переопределить или перегрузить) его методы и свойства.

Собачья жизнь полиморфизма, или Воспоминания о будущем

Полиморфизм - наиболее туманный принцип объектно-ориентированного программирования. Суть его состоит в том, что классы-потомки могут иметь методы с тем же самым названием, что и в родительском классе, но при этом выполнять действия, специфические для каждого конкретного класса. Будет правильно, если мы рассмотрим полиморфизм на примере.

Как известно. Все атеисты произошли от обезьяны (точнее, от двух обезьян). И только один богохульник – Полиграф Полиграфыч Шариков - произошёл от собаки. Попробуем изложить это художественное преувеличение на языке науки, то есть на *Delphi*.

Создадим класс той самой собаки Шарика, которая и попала под острый нож профессора Преображенского:

```
type TSharik = class
  function Say(): String; virtual;
end;
```

В данном случае Михаил Булгаков не слишком согрешил против истины – Шарик мог только гавкать:

```
  function TSharik.Say: String;
  begin
    Say:= 'Гав-гав!';
  end;
```

Обратите внимание: метод **Say** объявлен как виртуальный – чтобы его можно было переопределить в классе-наследнике.

Из повести Михаила Булгакова нам достоверно известно. Что Шарик превратился в Шарикова. Объявим новый класс:

```
type TSharikov = class (TSharik)
  function Say(): String; override;
end;
```

Как вы помните, Шариков недалеко ушёл от оригинала, вот только лексикон его стал более ехидным и злобным:

```
  function TSharikov.Say: String;
  begin
    Say:= 'Что-то вы меня больно притесняете, папаша!';
```

```
  end;
```

В программе мы можем одновременно создать и Шарика, и Шарикова:

```
var
  sharik: TSharik;
  sharikov: TSharikov;
. . .
  sharik:= TSharik.create;
  sharikov:= TSharikov.create;
```

Мы легко можем убедиться в этом по тому, что они оба подают голос (Рис. У17.2):

```
procedure TForm1.btnSayClick(Sender: TObject);
begin
  label1.Caption:= sharik.Say;
  label2.Caption:= sharikov.Say;
end;
```

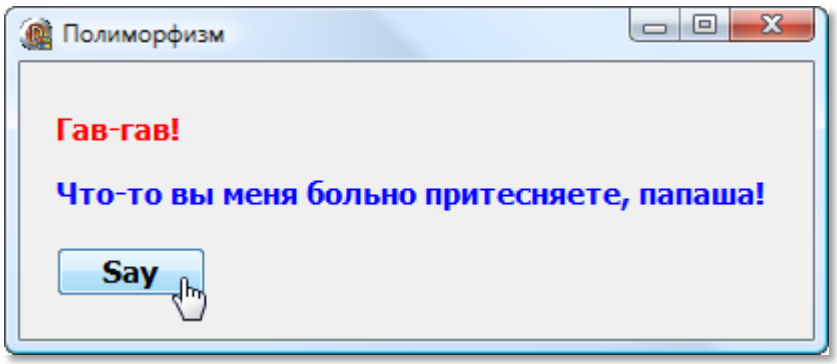


Рис. У17.2. Кто здесь голос подаёт?

А теперь – чудеса полиморфизма, которые и не снились профессору Преображенскому: Шарик мгновенно превращается в Шарикова:

```
sharik:= sharikov;
```

И вот уже Шарик, оставаясь. По сути, собакой, вспоминает, что в будущей жизни он был Шариковым, и лает совершенно по-человечески (Рис. У17.3).

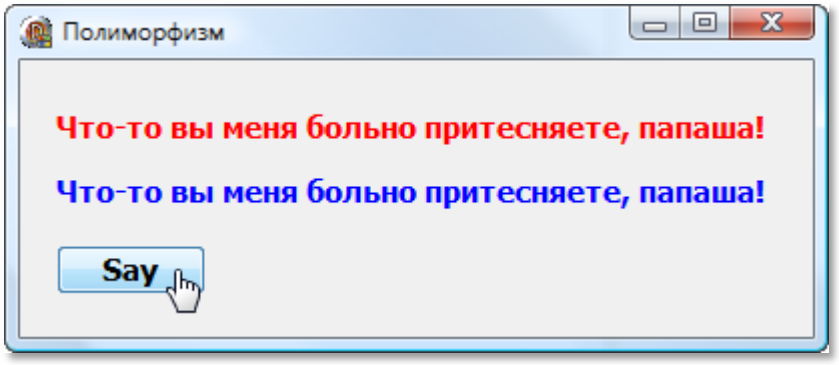


Рис. У17.3. Спелись!

В литературе описан и обратный процесс – превращения Шарикова снова в собаку:

```
sharikov:= sharik;
```

Увы, наука этого не допускает. Попытка повернуть эволюцию вспять пресекается компилятором (Рис. У17.4).

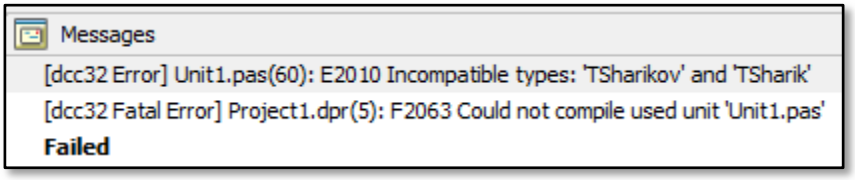


Рис. У17.4. Ошибка эволюции!

Тут мы вновь разошлись в суждениях – по богословским вопросам, как говаривал Арамис, - с Михаилом Булгаковым, но в своё оправдание мы можем создать в программе только Шарикова из собаки, а самого Шарика не создавать вообще:

```
  //sharik:= TSharik.create;
  sharikov:= TSharikov.create;
```

Насколько я помню, в повести Булгакова морали нет, а у нас она есть: даже став собакой, нужно помнить, что когда-то ты был человеком. Вот такой полиморфизм получается…



Исходный код программы находится в папке **Polymorphism**.

Области видимости и «доступность» членов класса

Знакомые нам уже разделы **public**, **private**, которые встретились уже при знакомстве с классом формы приложения, можно дополнить ещё двумя – **protected** и **published**. Они и определяют область видимости членов класса, то есть область кода программы, из которой к ним можно обратиться.

К полям, свойствам и методам, описанным в разделе **private**, можно обращаться только внутри процедур и функций того класса (точнее, модуля), в котором они объявлены. Здесь должны находиться все поля класса и методы, которые используются только для «внутриклассовых» операций и больше нигде не нужны.

Члены класса, имеющие область видимости **public**, доступны всем желающим. Если, конечно, модуль с данным классом занесён в список **uses** того модуля, из которого обращаются к членам класса..

Защищённые члены класса объявляются в разделе **protected**. Обычно там объявляются методы, которые будут доступны в потомках этого класса, а для всех остальных желающих они будут вести себя, как приватные члены класса.

Свойства и методы, описанные в разделе **published**, будут видны в окне *Инспектора объектов*, поэтому имеют смысл только для компонентов.



В классе-потомке можно переопределять область видимости члена класса, например, делать из **private public**. Например, в каком-либо классе есть нужное поле, однако оно объявлено в секции **private**. Вы просто создаёте потомка этого класса и объявляете поле как **public**, получая к нему доступ.

Выводы

Мораль сей басни такова…

И.А.Крылов

Какой же отсюда следует вывод…

Рассуждения ослика Иа-Иа

1. Если вы пишете простую программу или она нужна вам только на один раз, не конструируйте объекты! Программа наверняка будет работать медленнее, а вы потратите в 3 раза больше времени на описание класса.

2. Если программа большая, или вы планируете её долго использовать, делая новые версии, или хотите поделиться с другими программистами, то старайтесь каждый объект оформлять в виде записи или класса. Такой подход облегчит написание программы, её отладку, модификацию и сопровождение. Описание класса делайте по возможности простым, не поддавайтесь искушению использовать без нужды динамические, виртуальные и абстрактные методы и свойства.

3. Вы можете обойтись без свойств. Просто объявляйте поля в разделе *private*, а методы для записи и чтения – в разделе *public*. Именно так это делается в языке C++. Результат будет тот же самый, а код немного сократится. Но желательно не оставлять поля без присмотра, это проявление бесшабашности, которое в программировании не приветствуется.

Семинар 17. Игра в КЛАССики, или Опять бОЧКОвое ЛОТО

Обогатившись знаниями о классах, попробуем навести порядок в игре **21**.

Константы перенесём в начало программы, сразу после раздела *uses*:

```
const
  NAME_PROG : PChar =' Game 21 OOP';      //- название программы
  NUM_CARD=52;                            //- число карт в колоде
```



Константы заменяют в тексте программы малопонятные числа и строки более осмысленными, а также в случае необходимости позволяют быстро и удобно их корректировать. Например, достаточно вписать название новой программы в константу **NAME_PROG**, и оно будет использоваться во всём коде.

Название приложения также выводится в заголовке формы – свойство **Caption**. Мы его задали в *Инспекторе объектов*. Если изменить только значение константы **NAME_PROG**, то в заголовке формы и в *Панели задач* останется прежнее название. Поступаем так: свойство **Caption** стираем, а при создании формы задаём нужное значение:

```
procedure TfrmMain.FormCreate(Sender: TObject);
begin
  caption:= NAME_PROG;
  Application.Title:= NAME_PROG;
```

Вот теперь вы можете спать спокойно: название программы везде будет отображаться верно.

Идём дальше. Если в программе должны быть только объекты, то все переменные должны быть экземплярами классов, а все функции и процедуры – их методами.

Переменная **clrForm**, которая хранит цвет формы и компонентов на ней, должна быть полем формы. Переносим её в раздел *public*, чтобы до неё можно было легко добраться.

Рассмотрим процедуру **Sound**, которая воспроизводит звук из файла на диске. Методом какого объекта она должна быть? – А никакого! Это процедура общего назначения и может использоваться любым объектом. Зна-

чит, её можно оставить на прежнем месте, но, раз уж мы взялись наводить порядок, то правильнее будет создать для неё **новый модуль** и перенести процедуру туда:

```
unit utils;

interface

  uses mmsystem;

  procedure Sound(name: PChar);

implementation

//########################################
//# ВОСПРОИЗВЕСТИ ЗВУК С ЗАДАННЫМ ИМЕНЕМ #
//########################################
procedure Sound(name: PChar);
var s: PChar;
begin
  //сформировать имя файла:
  s:= PChar('wav\'+ name + '.wav');
  sndPlaySound(s, SND_ASYNC or SND_FILENAME);
end;

end.
```

Добавьте к программе новый модуль, назовите его, например, **utils** и под этим названием запишите на диск в папку с программой. В главном модуле программы подключите его:

```
uses
  …, utils;
```

Запустите программу – она будет работать, как и раньше, но теперь у вас появился универсальный модуль, который вы легко можете добавить в любую программу, и он будет радовать вас звуками и музыкой.

Бросим свой пылающий взор на процедуру **Delay**. Она приостанавливает выполнение программы на заданное время. Как и в истории со звуком, такая процедура может пригодиться в очень многих случаях, поэтому её тоже препровождаем в модуль **utils**:

```
  uses Windows, Forms, mmsystem;

  procedure Sound(name: PChar);
  procedure Delay(MSecs: Longint);

//################################
```

```
//#  ЗАДЕРЖКА В МИЛЛИСЕКУНДАХ   #
//################################
procedure Delay(MSecs: Longint);
var
  FirstTickCount, Now: Longint;
begin
  FirstTickCount := GetTickCount;
  repeat
    Application.ProcessMessages;
    Now := GetTickCount;
  until (Now - FirstTickCount >= MSecs) or (Now < FirstTickCount);
end;
```

Ещё одно недоумение вызывает функция **GetRandom**. Она тоже должна служить людям, а не отдельным объектам!

```
  procedure Delay(MSecs: Longint);
  function GetRandom(iLo, iHi: Integer): Integer;

//#####################################################################
//# ПОЛУЧИТЬ СЛУЧАЙНОЕ ЧИСЛО В ДИАПАЗОНЕ iLo <= GetRandom <= iHi #
//#####################################################################
Function GetRandom(iLo, iHi: Integer): Integer;
begin
  GetRandom := iLo + (Random (iHi - iLo + 1))
end;
```

На этом заполнение нового модуля можно считать законченным. Мы избавились от трёх подпрограмм, и они больше не заслоняют суть программы.

Не забудьте стереть эти подпрограммы в главном модуле.

Что ещё полезного можно сделать?

Поскольку игральная **карта** сама по себе ничего не делает, то мы не станем её преобразовывать в класс, а оставим записью, только подправим название одного поля. Так было:

```
  //описание карты:
  TCard= record
    suit: TSuit;           //- масть
    weight: TWeight;       //- достоинство
    exist: boolean;        //= TRUE, если карта находится в колоде
    img: integer;          //- номер картинки с картой в ImlCards
  end;
```

Так стало:

```
  //Карта
```

```
  TCard= record
    suit: TSuit;          //- масть
    weight: TWeight;      //- достоинство
    inPack: boolean;       //= TRUE, если карта находится в колоде
    img: integer;         //- номер картинки с картой в ImlCards
  end;
```

В первом варианте игры **колода карт** у нас была простым массивом, а число карт, вышедших из колоды, мы хранили в переменной **nCard**:

```
var
  pack: array[0..NUM_CARD] of TCard; // - колода карт
  nCard: integer; //число карт, вышедших из колоды
```

Уже нехорошо – данные о колоде хранятся в разных «местах». Правильнее было бы объединить их в запись. Однако колода уже может выполнять некоторые действия с картами – восстанавливать порядок, менять местами, тасовать и вычислять вероятность выхода хорошей карты. Естественно, этими возможностями следует наделить саму колоду, а для этого нужно создать новый класс, в котором указанные процедуры и функции будут его методами. При этом мы должны учитывать, что процедуры **Swap** и **restore** нужны только внутри самого класса и не должны использоваться извне, поэтому их нужно поместить в секцию *private*, а остальные две подпрограммы – процедуру **Shuffle** и функцию **calcChance** – в секцию *public*, так как они будут вызываться в разных местах кода.

В итоге получится такой класс для колоды:

```
  //Колода
  TPack= class
  private
    //число карт, вышедших из колоды:
    nCard: integer;
    lbl: TLabel;
    iml: TImageList;
    //все карты:
    aCard: array[0..NUM_CARD] of TCard;
    procedure Swap(var v1, v2: TCard);
    procedure restore();

  public
    constructor create(imlst: TImageList; labl: TLabel);
    procedure Shuffle();
    function calcChance(sc:integer; print: boolean): single;
    procedure draw(image: TImage; n: integer);
    function  getNCard: integer;
    procedure setNCard(n: integer);
  end;
```

Как видите, карты всё так же собраны в массив, но теперь класс сам распоряжается своими ими! Если класс колоды перенести в отдельный модуль, то изменять значение поля **nCard** можно через *свойства*:

```
function TPack.getNCard: integer;
begin
  result:= nCard;
end;

procedure TPack.setNCard(n: integer);
begin
  nCard:= n;
end;
```

Чтобы класс стал полностью «вещью в себе», добавим в него поля: **метку**, в которую будет выводиться вероятность, и **список картинок** с картами. На самом деле это не компоненты (они как были, так и останутся на форме), а только ссылки на них, но зато мы можем переименовывать их как угодно на форме, а в самом классе их названия останутся без изменения, что гораздо удобнее, если класс использовать в разных приложениях. Передавать компоненты можно при создании экземпляра класса, то есть объекта-колоды для игры:

```
//СОЗДАТЬ КОЛОДУ КАРТ
constructor TPack.create(imlst: TImageList; labl: TLabel);
begin
  iml:= imlst;
  lbl:=labl;
end;
```

Остальные методы класса благополучно импортируем из первой версии программы:

```
//ВОССТАНОВИТЬ КОЛОДУ
procedure TPack.restore();
var i,j,n: integer;
begin
  n:= 1;
  //восстановить колоду:
  for j:= 1 to 4 do             // 4 масти
     for i:= 1 to 13 do begin  // по 13 карт в масти
        aCard[n].suit:= TSuit(j);
        //вес карт 1,2,3,4,5,6,7,8,9,10,2,3,4
        if i < 11 then aCard[n].weight:= TWeight(i)
        else           aCard[n].weight:= TWeight(i-9);
        aCard[n].img:= (j-1)*13+i;
        aCard[n].inPack:=true;
        inc(n);
     end;
```

```
end;


//ПОМЕНЯТЬ ДВЕ КАРТЫ
procedure TPack.Swap(var v1, v2: TCard);
var vT: TCard;
begin
  vT := v1;
  v1 := v2;
  v2 := vT;
end;

//ПЕРЕТАСОВАТЬ КОЛОДУ
procedure TPack.Shuffle();
var
  iRnd: integer;
  i: integer;
begin
  sound('shuffle');
  Randomize;
  ///listbox1.Items.Add(inttostr(iFirst)+' '+inttostr(iLast));
  ///listbox1.Items.Add('')
  restore;
  For i := 52 DownTo 1 do begin
    frmMain.lstProtokol.Items.Add('i='+' '+inttostr(i));
    iRnd:= GetRandom(1, 52);
    frmMain.lstProtokol.Items.Add('iRnd='+' '+inttostr(iRnd));
    Swap (aCard[i],aCard[iRnd]);
    frmMain.lstProtokol.Items.Add('a[iRnd]='+'
'+inttostr(aCard[iRnd].weight)+'                          '+'a[i]='+'
'+inttostr(aCard[i].weight));
  end;
  frmMain.lstProtokol.Items.Add ('');
end;

//ВЫЧИСЛИТЬ ВЕРОЯТНОСТЬ ПРИХОДА ХОРОШЕЙ КАРТЫ
function TPack.calcChance(sc:integer; print: boolean): single;
var i:integer;
  n:integer; //осталось карт в колоде
  p:integer; //подходящих карт в колоде
  ch: single;
begin
  n:=0;
  p:=0;
  for i:= 1 to 52 do begin
    if aCard[i].inPack=true then begin
      if sc+aCard[i].weight <= 21 then inc(p);
      inc(n);
    end;
  end;
  ch:= p / n;
```

```
  if    print    then    lbl.Caption:=    'Вероятность    =    '+
leftstr(floattostr(ch),5);
  result:= ch;
end;
```

Некоторое неудобство возникает при рисовании карт игрока, поскольку это действие разворачивается на компоненте *TImage*, принадлежащем не колоде, а игроку:

```
//НАРИСОВАТЬ ЗАДАННУЮ КАРТУ НА УКАЗАННОМ КОМПОНЕНТЕ ИГРОКА
procedure TPack.draw(image: TImage; n: integer);
//image - компонент, на котором рисуем
//n - номер карты игрока
begin
  //стереть предыдущую картинку:
  image.Canvas.FillRect(game.frm.ClientRect);
  //показываем карту:
  iml.Draw(image.Canvas, 0, 0, aCard[n].img, true);
end;
```

Игрок в первой версии игры также был представлен **записью**:

```
  //Игрок
  TPlayer= record
    frm: TfrmMain;
    lbl: TLabel;
    //компоненты TImage для вывода карт:
    img: array[1..11] of TImage;
    //номера карт из колоды, которые находятся у игрока
    idCards: array[1..11] of integer;
    nCard: integer; // число карт у игрока
    score: integer; //число очков
    win: integer;   //число побед
    procedure setScore(sc: integer);
  end;
```

Но уже в ней мы видим процедуру **setScore**, которая изменяет значение поля **score**. Совершенно очевидно, что игрок – главное действующее лицо этой азартной игры - должен быть **объектом**, поэтому преобразуем запись в **класс**:

```
  //Игрок
  TPlayer= class
    constructor create(labl: TLabel);
    procedure setImage(n: integer; image: TImage);
    procedure setScore(sc: integer);
    procedure showCards(open: boolean; showScore: boolean);
    procedure restoreImage(act: boolean); overload;
    procedure restoreImage(id: integer; act: boolean); overload;
    //взять дополнительные карты:
```

```
    procedure addCards();
    procedure addCard();

  private
    lbl: TLabel;
    //компоненты TImage для вывода карт:
    img: array[1..11] of TImage;
     //номера карт из колоды, которые находятся у игрока
    idCards: array[1..11] of integer;
    nCard: integer; // число карт у игрока
    score: integer; //число очков
    win: integer;   //число побед
  end;
```

В секцию *private* поместим все переменные этого класса, что всегда нужно делать в первую очередь. Все методы (в данном случае - процедуры), которые вызываются из главной программы, следует сделать доступными из любого места кода, поэтому для них место в секции *public*.

При создании нового игрока мы отдаём ему **метку** для вывода набранных в партии очков и обнуляем число побед:

```
//#####################################################
//#                  КЛАСС  TPLAYER                   #
//#####################################################

//СОЗДАТЬ ИГРОКА
constructor TPlayer.create(labl: TLabel);
begin
  lbl:= labl;
  //число побед:
  win:=0;
end;
```

Метод **setImage** заполняет массив компонентов для демонстрации карт игрока:

```
//КОМПОНЕНТЫ ИГРОКА ДЛЯ ВЫВОДА КАРТ
procedure TPlayer.setImage(n: integer; image: TImage);
begin
  img[n]:= image;
end;
```

При выводе карт нам придётся убирать с глаз долой те компоненты, в которых карт нет, и, наоборот, показывать на экране компоненты с картами:

```
//ВОССТАНОВИТЬ КОМПОНЕНТ TImage ИГРОКА
//id - номер компонента
//act - скрыть/показать компонент
```

```
procedure TPlayer.restoreImage(id: integer; act: boolean);
begin
    img[id].Cursor:= crNo;
    img[id].Visible:= act;
    img[id].Enabled:= act;
end;

//ВОССТАНОВИТЬ ВСЕ КОМПОНЕНТЫ TImage ИГРОКА
procedure TPlayer.restoreImage(act: boolean);
var i: integer;
begin
  for i:= 1 to 11 do begin
    img[i].Cursor:= crNo;
    img[i].Visible:= act;
    img[i].Enabled:= act;
  end;
end;
```

Чтобы показать карты игрока, нужно просмотреть все его карты и, если какая-либо карта открыта (open=true), то рисуем её на экране лицевой стороной вверх, в противном случае рисуем рубашку. Обратите внимание на то, что для рисования карт мы пользуемся методом **draw** колоды карт:

```
//ПОКАЗАТЬ ВСЕ КАРТЫ ИГРОКА
procedure TPlayer.showCards(open: boolean; showScore: boolean);
var i: integer;
begin
  for i:= 1 to nCard do begin
    if open=true then
      //показываем карту:
      pack.draw(img[i], idCards[i])
      //показываем рубашку:
    else
      pack.draw(img[i], 0);
  end;
  //показываем очки:
  lbl.Visible:= showScore;
end;
```

Не забываем при этом показывать и набранные очки:

```
//ПОКАЗАТЬ НАБРАННЫЕ ОЧКИ
procedure TPlayer.setScore(sc: integer);
begin
  score:= sc;
  lbl.Caption:= inttostr(score);
end;
```

Одному из игроков (у нас это всегда человек) придётся добирать по одной карте. Как вы помните, для этого нужно нажать кнопку *Взять карту*:

```
//ИГРОК БЕРЁТ КАРТУ
procedure TfrmMain.sbtCardClick(Sender: TObject);
begin
  player.addCard;
end;


//ВЗЯТЬ ОДНУ ДОПОЛНИТЕЛЬНУЮ КАРТУ
procedure TPlayer.addCard();
var n: integer;
begin
  pack.nCard:= pack.nCard+1;
  n:= pack.nCard;
  with player do begin
    nCard:= nCard +1;
    img[nCard].Visible:= true;
    img[nCard].Enabled:= true;
    img[nCard].Cursor:= crNo;
    setScore(player.score + pack.aCard[n].weight);
    pack.aCard[n].inPack:= false;
    idCards[nCard]:= n;
  end;
  //показать карты и очки:
  showCards(true, true);

  n:= game.test;
  if n<> 0 then game.newGame();
end;
```

Второй игрок – компьютер – должен сразу взять столько карт, сколько ему позволяет разум:

```
//ВЗЯТЬ ДОПОЛНИТЕЛЬНЫЕ КАРТЫ
procedure TPlayer.addCards();
var
  ch: single;
  ich: integer;
begin
  repeat
    ch:= pack.calcChance(comp.score, false);
    ich:= round(ch*1000);
    if getRandom(0,1000) <= ich*1000 then begin//берём еще карту
      //карта вышла из колоды
      inc(pack.nCard);
      //и пришла к игроку:
      inc(comp.nCard);
      pack.aCard[pack.nCard].inPack:= false;
      idCards[comp.nCard]:= pack.nCard;
      setScore(score + pack.aCard[pack.nCard].weight);
      img[nCard].Visible:= true;
      img[nCard].Enabled:= true;
      img[nCard].Canvas.FillRect(game.getForm.ClientRect);
```

```
      //если комп. набрал 21 очко, то дальше карты не брать!
      if comp.score=21 then break;
    end;
  until  getRandom(0,50)+pack.calcChance(comp.score,  false)*100  <
70;
end;
```

У него, безусловно, задача гораздо сложнее, но мы её уже успешно решили в первой версии игры.

Если вы внимательно читали код, то обнаружили ещё один, доселе не упомянутый объект, – **game**, который делится с игроком главной формой. Действительно, любая игра – это главный объект программы, который и должен выполнять обязанности по ведению честной игры, то есть следить за соблюдением правил. Этот объект владеет главной формой программы и может передавать её нуждающимся методом **getForm**.

```
  //Игра
  TGame= class
    constructor create(form: TfrmMain; labl: TLabel);
    procedure newGame();
    procedure setTablo();
    function getForm(): TfrmMain;
    function test(): integer;
    procedure raskladka();
    procedure openCards();

  private
    frm: TfrmMain;
    lbl: TLabel;
  end;

//ВЕРНУТЬ ФОРМУ ПРИЛОЖЕНИЯ
function TGame.getForm(): TfrmMain;
begin
  result:= frm;
end;
```

У игры также имеется метка **lbl** для ведения счёта. Естественно, ни форма, ни эта метка не может принадлежать игрокам. Когда потребуется, объект-игра покажет счёт на экране:

```
//ПОКАЗАТЬ СЧЁТ
procedure TGame.setTablo();
  var s:string;
begin
  s:= 'Счёт '+ inttostr(comp.win) + ' : ' + inttostr(player.win);
  lbl.Caption:= s;
```

```
end;
```

Именно игра создаёт обоих игроков и колоду с картами:

```
//СОЗДАТЬ ИГРУ
constructor TGame.create(form: TfrmMain; labl: TLabel);
begin
  frm:= form;
  lbl:= labl;
  //создать игроков и колоду:
  comp:= TPlayer.Create(form.lblCompScore);
  player:= TPlayer.Create(form.lblPlayerScore);
  pack:=TPack.Create(form.ImlCards, form.lblChance);
end;
```

Когда начинается новая партия, «высший разум игры» возвращает карты в колоду и руководит колодой и игроками:

```
//НАЧАТЬ НОВУЮ ИГРУ
procedure TGame.newGame();
begin
  //у игроков нет карт и очков:
  comp.setScore(0);
  comp.nCard:= 0;
  player.setScore(0);
  player.nCard:= 0;
  //Из колоды не вышла ни одна карта:
  pack.nCard:=0;
  //разложить карты:
  raskladka();
end;
```

При раздаче карт объект-игра заставляет колоду перетасоваться, раздаёт по две карты игрокам, а затем передаёт ход игроку-компьютеру, который набирает себе карты:

```
//РАЗЛОЖИТЬ КАРТЫ
procedure TGame.raskladka();
var
  i,n : integer;
begin
  //перемешать колоду:
  pack.Shuffle;

  //скрыть компоненты для карт:
  player.restoreImage(false);
  comp.restoreImage(false);

  //сдать по две карты игрокам:
  for i:= 1 to 2 do begin
    //для компьютера:
```

```
    with comp do begin
      //компоненты видимы:
      restoreImage(i,true);
      //добавить очки:
      setScore(score + pack.aCard[i].weight);
      //запомнить полученную карту:
      idCards[i]:= i;
    end;
    //карта из колоды вышла:
    pack.aCard[i].inPack:= false;

    //для игрока:
    with player do begin
      restoreImage(i,true);
      setScore(score + pack.aCard[i+2].weight);
      idCards[i]:= i+2;
    end;
    pack.aCard[i+2].inPack:= false;

    delay(10);
  end;

  //число карт
  //вышло из колоды:
  pack.nCard:=4;
  //у игроков:
  comp.nCard:=2;
  player.nCard:=2;
  //нарисовать карты раскладки на компонентах Image:
  player.showCards(true,true);
  comp.showCards(false,false);

  n:= game.test;
  if n <> 0 then game.newGame();

  //компьютер добирает карты:
  comp.addCards;
  comp.showCards(false,false);

  if comp.score <>21 then begin
     n:= game.test;
     if n<> 0 then game.newGame();
  end;
end;
```

После каждого хода объект-игра проверяет, согласно правилам, не закончилась ли партия, а если закончилась, определяет победителя. Игроки никак не могут повлиять на «главного судью» - здесь всё по-честному!

```
//ПРОВЕРИТЬ, КТО ВЫИГРАЛ
function TGame.test(): integer;
//Возвращает 0, если никто не выиграл и не проиграл
```

```
//            1, если выиграл компьютер
//            2, если выиграл игрок
//           -1, если проиграл компьютер
//           -2, если проиграл игрок
//           11, ничья
var n: integer;
begin
  n:= 0;
  if (comp.score=21) and (player.score>21) then n:= 1;
  if (player.score=21) then n:= 2;
  if comp.score>21 then n:= -1;
  if player.score>21 then n:= -2;
  if (comp.score=21) and (player.score=21) then n:=11;

  if n<>0 then comp.showCards(true,true);

  if n=1 then begin
    inc(comp.win);
    application.MessageBox     ('Компьютер     выиграл!',NAME_PROG,
MB_OK);
  end;
  if n=2 then begin
    inc(player.win);
    application.MessageBox ('Игрок выиграл!',NAME_PROG, MB_OK);
  end;
  if n=-1 then begin
    inc(player.win);
    application.MessageBox     ('Компьютер     проиграл!',NAME_PROG,
MB_OK);
  end;
  if n=-2 then begin
    inc(comp.win);
    application.MessageBox ('Игрок проиграл!',NAME_PROG, MB_OK);
  end;
  if n=11 then begin
    application.MessageBox ('Ничья!',NAME_PROG, MB_OK);
  end;

  setTablo();
  pack.calcChance(player.score, true);
  result:= n;
end;
```

Игрок-человек в любой момент может нажать кнопку *Вскрыть карты* и тем самым завершить игру:

```
//ВСКРЫТЬ КАРТЫ
procedure TfrmMain.sbtOpenClick(Sender: TObject);
begin
  game.openCards;
end;
```

Однако результат партии определяет объект-игра:

```
//ВСКРЫТЬ КАРТЫ ИГРОКОВ
procedure TGame.openCards();
begin
  comp.showCards(true,true);
  if player.score = comp.score then begin
     application.MessageBox ('Ничья!',NAME_PROG, MB_OK);
  end
  else if game.test()=0 then begin
    if player.score > comp.score then begin
      inc(player.win);
      application.MessageBox ('Игрок выиграл!',NAME_PROG, MB_OK);
    end;
    if player.score < comp.score then begin
      inc(comp.win);
      application.MessageBox    ('Компьютер    выиграл!',NAME_PROG,
MB_OK);
    end;
  end;
  newGame();
end;
```

Теперь вернёмся к началу программы и убедимся, что остались только переменные-объекты:

```
var
  frmMain: TfrmMain;
  game: TGame;
  comp, player: TPlayer;
  pack: TPack;
```

Ничего лишнего - все остальные переменные и процедуры нашли свои места в соответствующих классах. В программе воцарил порядок! И теперь вам не нужно отыскивать переменные и подпрограммы, разбросанные по всему коду, - всё разложено по полочкам и аккуратно подписано. В этом и заключается одно из главных преимуществ объектно-ориентированного программирования перед процедурным.

Но – мы не рассмотрели пока самый первый класс – главной формы приложения, а ведь кто-то же должен создать собственно игру и поделиться своей формой и компонентами с игрой и игроками:

```
//##############################################
//#              КЛАСС TfrmMain                #
//##############################################

//СОЗДАТЬ ФОРМУ
procedure TfrmMain.FormCreate(Sender: TObject);
var i: integer;
  tempC: TComponent;
```

```
begin
  caption:= NAME_PROG;
  Application.Title:= NAME_PROG;

  clrForm:= rgb(120,144,120);
  //окрасить форму:
  frmMain.Color:= clrForm;

  //создать игру:
  game:= TGame.Create(frmMain, lblTablo);

  //для всех компонентов Image установить свойства:
  for i:= 1 to 22 do begin
    tempC:= FindComponent('Image'+ inttostr(i));
    with TImage(tempC) do begin
      //закрасить фон, чтобы не было видно уголков карт:
      Canvas.Brush.Color:=  clrForm;
      Canvas.FillRect(ClientRect);
      //прозрачность:
      transparent:= true;
      //компоненты  TImage для игроков:
      if i < 12 then comp.setImage(i,TImage(tempC))
      else           player.setImage(i-11,TImage(tempC));
    end;
  end;
end;


//АКТИВАЦИЯ ФОРМЫ - ПЕРВАЯ РАСКЛАДКА
procedure TfrmMain.FormActivate(Sender: TObject);
begin
  //delay(100);
  //начинаем новую игру:
  game.newGame();
end;
```

Дальше уже всем верховодит созданный в процедуре **FormCreate** объект-игра, но игрок-человек может вмешаться в ход игры, нажимая кнопки:

```
//ЗАКОНЧИТЬ РАБОТУ С ПРОГРАММОЙ
procedure TfrmMain.sbtCloseClick(Sender: TObject);
begin
  close
end;

//НАЧАТЬ НОВУЮ ИГРУ
procedure TfrmMain.sbtNewGameClick(Sender: TObject);
begin
  game.newGame;
end;

//ВСКРЫТЬ КАРТЫ
```

```
procedure TfrmMain.sbtOpenClick(Sender: TObject);
begin
  game.openCards;
end;

//ИГРОК БЕРЁТ КАРТУ
procedure TfrmMain.sbtCardClick(Sender: TObject);
begin
  player.addCard;
end;
```

Так как интерфейс программы не изменился по сравнению с первой версией, то повторять его здесь нет смысла. А вот код программы, хоть он и не изменился по существу (программа работает точно так же, как и её первая версия), стал гораздо более организованным, понятным и удобным для работы.



Исходный код программы находится в папке **21-OOP**.

Самостоятельная работа



Новая версия программы хорошо иллюстрирует первый принцип ООП - **инкапсуляцию**. Действительно, нам удалось объединить данные каждого объекта с процедурами, которые этими данными манипулируют. Как говорится, всё в одном флаконе. Но данные, которые мы спрятали от любопытных глаз в секцию **private**, остались легкодоступными, чем мы и сами пользовались в программе. Это плохо и вы должны довести дело ООП до конца.

Добавьте к проекту новые модули для каждого класса и перенесите в них код. Лучше это делать последовательно, чтобы не запутаться в исправлениях. Для образца я проделал все необходимые операции с классом колоды. Внимательно посмотрите на изменения в программе, а затем проделайте то же самое и с остальными классами.



Исходный код программы находится в папке **21-OOP2**.

ООПять лото!

Давайте охватим вниманием и второй принцип ООП – **наследование**. Игра в **21** уже порядком надоела, поэтому вернёмся к нашим бочонкам.

Сразу присоединим к проекту наш славный модуль **utils**, а функцию **getRandom** из главного модуля уберём. В отличие от карты, проявим к **бочонку** больше уважения и сделаем его классом:

```
TBochonok = class
  constructor create();

  protected
    leave: bool;
    number: integer;
end;
```

Конечно, много чести, но для учебных целей можно и смириться с некоторым бочковым культом личности. При создании бочонка просто отмечаем, что он ещё не вышел из мешка.

```
constructor TBochonok.create();
begin
  leave:= false;
end;
```

А вот и **наследник** бочонка – **класс мешка**, в котором эти бочонки до поры до времени хранятся, ожидая своей участи:

```
TMeshok = class(TBochonok)
    constructor create(num: integer);
    procedure shuffle();
    function getBoch(n: integer): integer;
    function getNumBoch(): integer;

  private
    nBoch: integer;
    boch: array[1..NUM_BOCHONOK] of TBochonok;
end;
```

Как и в случае с колодой, все бочонки собраны в массив - **boch**, а в переменной **nBoch** мы запоминаем общее число бочонков в игре. Создавая мешок, мы формируем и массив бочонков:

```
constructor TMeshok.create(num: integer);
var i: integer;
begin
  nBoch:= num;
```

```
  for i := 1 to num do
    boch[i]:= TBochonok.create();
end;
```

Обратите внимание, что при создании бочонок сразу помечается - `leave:= false`.

Затем мешок тщательно встряхиваем, чтобы ни у кого не осталось и тени сомнения в честности игры:

```
procedure TMeshok.shuffle;
var
  iRnd,n: integer;
  i,j: integer;
begin
  Randomize;

  //восстановить бочонки:
  for i:= 1 to nBoch do
    boch[i].number:= i;

  //перемешать бочонки:
  for i := 1 to nBoch do begin
    iRnd:= GetRandom(1, nBoch);
    n:= boch[i].number;
    boch[i].number:=boch[iRnd].number;
    boch[iRnd].number:=n;
  end;
end;


function TMeshok.getBoch(n: integer): integer;
begin
  result:= boch[n].number;
end;

function TMeshok.getNumBoch(): integer;
begin
  result:= nBoch;
end;
```

Для игры нам понадобится игрок, который вытаскивает бочонки из мешка. Хоть он и наследует мешку, мы назовём его **боссом**:

```
TBoss = class(TMeshok)
    constructor create();
    procedure shuffleMeshok();
    //вытащить следующий бочонок:
    function getBoch(): integer;
    function getCurBoch(): integer;
```

```
  private
    curBoch: integer;
end;
```

Когда игра (класс игры в этой программе отсутствует, поэтому допишите его сами!) создаёт босса, он вызывает метод **create** своего предка (как все настоящие игроки произошли от обезьяны, так и лоточный босс произошёл от мешка), чтобы создать мешок с бочонками:

```
//СОЗДАТЬ ГЛАВНОГО ИГРОКА
constructor TBoss.create();
begin
  //создать мешок, в котором NUM_BOCHONOK бочонков:
  inherited create(NUM_BOCHONOK);
end;
```

Дальше босс перемешивает бочонки в мешке:

```
procedure TBoss.shuffleMeshok();
begin
  shuffle;
  curBoch:=0;
end;
```

Заметьте: как всякий начальник, он делает это опять же не сам, а пользуется процедурой **shuffle**, полученной им в наследство от мешка! Число вышедших бочонков пока равно нулю.

Тут уже начинается игра как таковая, и босс вынимает бочонки по одному:

```
function TBoss.getBoch(): integer;
begin
   inc(curBoch);
  //игра закончена
  if curBoch> getNumBoch() then result:=0
  else result:= inherited getBoch(curBoch);
end;
```

Причём снова делегирует свои обязанности безотказному мешку!

Осталось создать само приложение, босса с мешком и бочонками и начать игру:

```
//СОЗДАТЬ ФОРМУ ПРИЛОЖЕНИЯ
procedure TForm1.FormCreate(Sender: TObject);
begin
  //создать главного игрока:
  boss:= TBoss.create();
  //и начать новую игру:
```

```
  newGame();
end;

//НАЧАТЬ НОВУЮ ИГРУ
procedure TForm1.newGame();
begin
  boss.shuffleMeshok;
  Form1.lblNum.Caption:= '';
  Form1.lstProtocol.Items.Clear;
end;
```

Интерфейс программы нам тоже достался в наследство от предыдущей версии, а модуль **utils** – от игры в *21*, так что мы можем и должны оценить все преимущества ООП, который позволяет нам беспардонно пользоваться плодами чужого труда.

Когда пользователь программы нажимает кнопку *Достать бочонок*, босс послушно выполняет команду:

```
procedure TForm1.sbtGetClick(Sender: TObject);
var
  n,m: integer;
  s: string;
begin
  n:= boss.getBoch();
  //игра закончена
  if n = 0 then begin
     Application.MessageBox('Мешок пустой!', 'Лото', MB_OK);
  end
  else  begin
    s:= inttostr(n);
    case n of
      1: s:= s+' - кол';
      3: s:= s+' - трое';
      7: s:= s+' - кочерга';
     10: s:= s+' - бычий глаз';
     11: s:= s+' - барабанные палочки';
     12: s:= s+' - дюжина';
     13: s:= s+' - чёртова дюжина';
     22: s:= s+' - уточки';
     44: s:= s+' - стульчики';
     55: s:= s+' - перчатки';
     66: s:= s+' - валенки';
     69: s:= s+' - туда-сюда';
     77: s:= s+' - топорики';
     80: s:= s+' - бабка';
     90: s:= s+' - дед';
    end;
    lblNum.Caption:= s;
    if n < 10 then s:= ' '+s;
    m:= boss.getCurBoch();
```

```
    if m <10 then s:= '  '+ inttostr(m)+'. '+s
    else                  s:= ' ' + inttostr(m)+'. '+s;

    lstProtocol.Items.Add(s);
    lstProtocol.TopIndex:=lstProtocol.Items.Count-1;
  end;
end;
```

И последняя кнопка – *Новая игра* - начинает новый замес бочонков:

```
procedure TForm1.sbtNewMouseDown(Sender: TObject; Button: TMouse-
Button;
  Shift: TShiftState; X, Y: Integer);
begin
  newGame();
  boss.shuffleMeshok;
end;
```

Чтобы оценить все достоинства ООП, вам нужно использовать его в своих программах. Как видите, сделать это совсем нетрудно. Вы можете начать с перекройки своих старых программ на новый лад, а уж новые следует с самого начала писать «классно»!



Исходный код программы находится в папке **Loto-OOP**.

Ну, а если вы не пришли в восторг от программирования классов, значит, вам пора на *процедурное* программирование!

Часть II. Практикум

Практика – критерий истины.

Философизм

Во второй части книги вы обобщите и углубите свои знания основных конструкций языка *Delphi* и укрепите навыки по работе в среде разработки программ. Здесь вас ждёт много нового, интересного материала.

Вы научитесь создавать фигурные и «живописные» окна для своих приложений на примере таких замечательных игр, как **Mind Game**, **Тетрис**, **Викторина**, **Тест космонавта**. Вы напишете и другие интересные игры: **XorGame**, **ShorGame**, **Бокалы**, **Кладоискатель**.

Вы изучите новые приёмы программирования, а также компоненты и классы *Delphi*: **TDrawGrid**, **TCanvas**, **TGraphic**, **TPen**, **TBrush**, узнаете, как написать простой компонент и установить его в *Палитру компонентов*.

Большое внимание на *Практикуме* будет уделено разработке графического дизайна и эффективных алгоритмов для всех программ.

Занятие 1. Неформальные формы

Карлсон прервал размышления Малыша.
- Я бы не прочь сейчас слегка поразвлечься, -
сказал он и с любопытством огляделся вокруг.

Астрид Линдгрен

Наша главная задача – научиться создавать приложения, полезные для ума и приятные для глаз. О дизайне программ мы будем говорить ещё много в следующих главах, а пока займёмся «офигуриванием» форм и раскраской их во все цвета радуги.

Кроме примеров, демонстрирующих технику работы с окнами, мы напишем и полноценные программы, которыми вполне можно пользоваться для укрепления серых клеточек мозга, которыми так гордился частный детектив Пуаро по имени Эркюль (по-французски – *Геркулес*; то ли шутка Агаты Кристи (Пуаро был невысок ростом), то ли намёк на его недюжинные аналитические способности (вспомните и другого «гиганта мысли» - Кису Воробьянинова)).

Начнём, пожалуй …

Шейпинг-класс, или Форменное безобразие

90-60-90

Фигурка мирового класса

Джентльмены, как известно, предпочитают блондинок, а системные программисты – прямоугольные формы. «Бессистемные» же программисты, то есть любители, нередко склонны к упадничеству и украшательству своих изделий через фривольные окна. Тут вспоминается анекдот. Возвращается муж с работы, жена жарит котлеты: все разного размера и формы. Муж с удивлением спрашивает, для чего она понаделала такие странные экземпляры. На что мудрая жена отвечает: «Так ты же сам просил разнообразить наше питание!» Что ж, каждый ищет своё, а находит чужое.

Области в *Windows*

Есть несколько способов придать форме замысловатый вид. В этой главе мы рассмотрим только два - наиболее простых. Начнём с функций *Windows*

API, которые позволяют создать из геометрических примитивов область любой конфигурации, а затем назначить полученную область любому оконному компоненту, включая, естественно, и сам форму.



Назначить область можно только 1 раз, после чего она станет вам недоступной и вам придётся формировать её снова!

Областью (или *регионом*) в *Microsoft Windows* называется прямоугольник, многоугольник, эллипс или комбинация любого количества этих фигур. Не очевидно, но из этого следует, что область может иметь любую конфигурацию. Например, функция

```
function CreateEllipticRgn (nLeftRect, nTopRect,
                            nRightRect,
                            nBottomRect: integer): HRGN;
```

где

nLeftRect, **nTopRect** - координаты x и y верхнего левого угла описанного (вокруг эллипса) прямоугольника,

nRightRect, **nBottomRect** - координаты x и y нижнего правого угла описанного прямоугольника

позволяет создать *эллиптическую* (круглую, овальную) область. Прямоугольник определяет размеры, форму и расположение этой области. Длинная сторона прямоугольника задаёт размер большой оси эллипса, короткая сторона – меньшую ось, а центр прямоугольника совпадает с точкой пересечения осей эллипса (Рис. П1.1).

Функция возвращает описатель области типа **HRGN** (абракадабра, образованная от *handle region*), если она завершилась удачно, или *NULL* в противном случае.

Пример использования этой функции вы можете найти в разделе, посвящённом игре *Mind Game*.

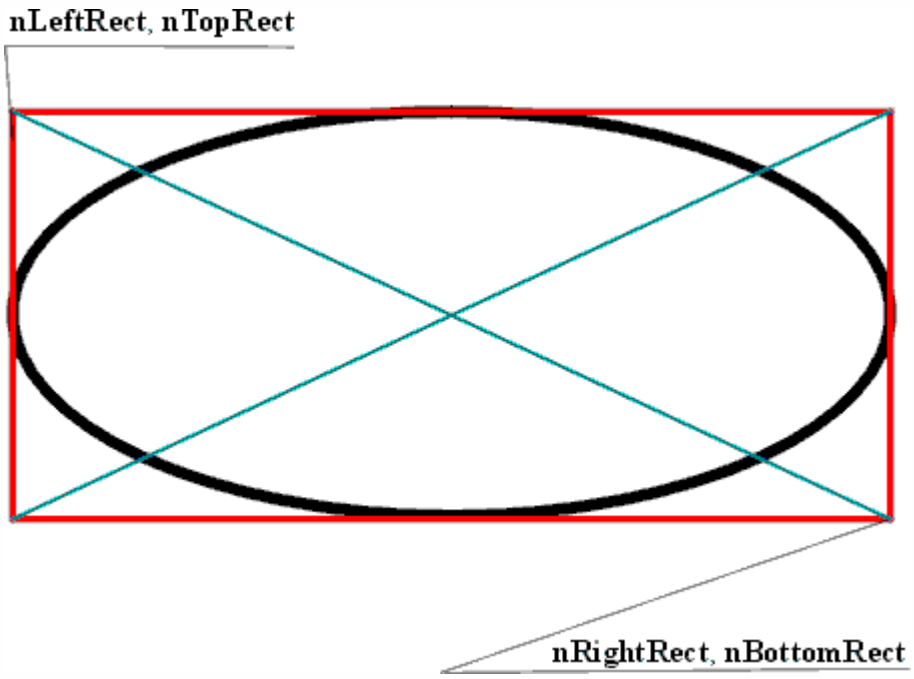


Рис. П1.1. Эллипс

Точно так же действует и другая функция:

`function CreateEllipticRgnIndirect(CONST RECT: TRect) : HRGN;`,

только вместо координат углов описанного прямоугольника нужно задать переменную **RECT** типа **TRect**, содержащую координаты левого верхнего и правого нижнего углов этого прямоугольника.

Чтобы поближе познакомиться с тонкостями и прелестями округлых форм, откроем новый проект, выкрасим форму в синий цвет (**Color** = clBlue**;** люди, подверженные цветовым предрассудкам и ассоциациям, могут выбрать любой другой), придадим форме привлекательный вид, растянув её по вертикали (**Width** = 409, **Height** = 576). Дважды щёлкнем форму по носу и в раскрывшейся заготовке процедуры напишем:

```
//СОЗДАТЬ ФОРМУ
procedure TForm1.FormCreate(Sender: TObject);
var area: TRect;
begin
  //описанный прямоугольник
  area:=Rect(0,0,Width,Height);

  //сформировать область
  rgn:= CreateEllipticRgnIndirect (area);
```

```
  //задать форму окна по области rgn
  SetWindowRgn(Handle, rgn, True);
end;
```



Функция **SetWindowRgn** назначает область заданному окну. Та часть окна, которая находится вне области, на экран не выводится.

```
int SetWindowRgn(
  HWND hWnd,      // описатель окна
  HRGN hRgn,      // описатель области. Если hRgn=NULL,
то область уничтожается. Координаты области отсчитываются
от верхнего левого угла всего окна, а не его клиентской
области.
  BOOL bRedraw    // условие перерисовки окна. Если равно
TRUE, то окно перерисовывается. Это значение типично для
видимых окон.
);
```

После назначения области заданному окну, она передаётся системе, которая сама уничтожает область, когда она становится ненужной. Это значит, в частности, что вы не можете использовать одну и ту же область дважды – при каждом использовании области вы должны создавать её заново.

Возвращаемое значение. Если функция завершилась успешно, то возвращается ненулевое значение.

Переменную **rgn** мы объявим в разделе **private** типа формы: **`rgn: HRGN;`**

Правила хорошего тона требуют очистить память от мусора, поэтому созданную область безжалостно уничтожаем при выходе из программы:

```
//УНИЧТОЖИТЬ ФОРМУ
procedure TForm1.FormDestroy(Sender: TObject);
begin
  DeleteObject(rgn)
end;
```

Осталось в центре формы поставить кнопку **bbtClose** типа **BitBtn** (**Caption** = Закрыть, **Kind** = bkClose) как раз для выхода из программы и добавить код для неё:

```
//ЗАКРЫТЬ ПРОГРАММУ
procedure TForm1.bbtCloseClick(Sender: TObject);
begin
  close
end;
```

Запускайте программу и убедитесь – форма славно закруглилась и потеряла «голову» (Рис. П1.2).

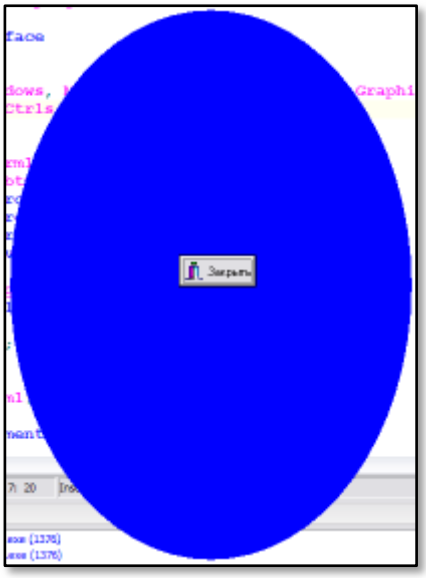


Рис. П1.2. Эллиптическое окно приложения

Есть в *Windows API* и две более прозаические функции - для создания *прямоугольных* областей (Рис. П1.3). Они не на шутку аналогичны первым двум:

```
function CreateRectRgn (nLeftRect,
                        nTopRect,
                        nRightRect,
                        nBottomRect: integer): HRGN;
```

и

```
function CreateRectRgnIndirect (CONST RECT: TRect) : HRGN;.
```

Вряд ли они могут привлечь внимание ценителя изысканных окон, поэтому мы их обойдём стороной, заметив, однако, что прямоугольные области (как и все прочие) могут пригодиться не только при обрамлении окон, но и во многих других случаях.

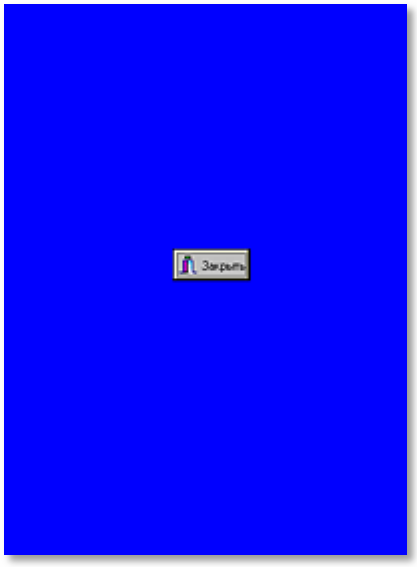


Рис. П1.3. Прямоугольное окно приложения

Возможно, вас больше заинтересует форма, объединяющая достоинства и недостатки прямоугольной и эллиптической форм. Вы правы, в результате этого мезальянса получится прямоугольник с закруглёнными углами, реализуемый с помощью функции:

```
function CreateRoundRectRgn (nLeftRect,
                             nTopRect,
                             nRightRect,
                             nBottomRect,
                             nWidthEllipse,
                             nHeightEllipse: integer): HRGN;
```

где

nLeftRect, **nTopRect** - координаты x и y верхнего левого угла описанного прямоугольника,

nRightRect, **nBottomRec** - координаты x и y нижнего правого угла описанного прямоугольника,

nWidthEllipse, **nHeightEllipse** – ширина и высота эллипса для закруглённого угла.

Чтобы увидеть это чудо, заменим одну строчку в методе ***TForm1.FormCreate*** другой:

```
//закруглённый прямоугольник
rgn:= CreateRoundRectRgn (0, 0, Width, Height, 100,100);
```

Запустите программу – результатом вполне можно восхищаться: углов нет, а прямоугольник остался – конформистам и политикам это должно понравиться (Рис. П1.4).

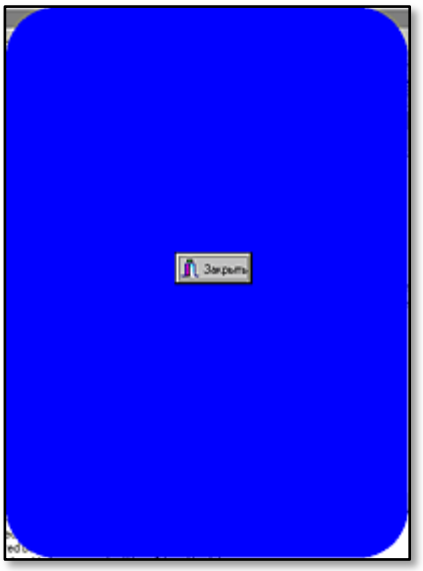


Рис. П1.4. Закругляем острые края!

Среди функций *Windows API* имеются и более сильнодействующие средства для создания хитроумных поверхностей. Так, вы легко можете «вырезать» любой многоугольник из своей формы, воспользовавшись вот такой функцией:

```
function CreatePolygonRgn (CONST POINT: array of TPoint;
                           cPoints,
                           fnPolyFillMode: integer) : HRGN;,
```

где

POINT – массив точек, определяющих вершины многоугольника (он всегда будет замкнутым, так как последняя точка соединяется с первой),

cPoints – количество точек в массиве,

fnPolyFillMode – режим закраски области:

ALTERNATE – заполняются области между чётными и нечётными сторонами,
WINDING – заполняется вся область.



Если области имеют простую форму, то разницы между этими двумя режимами вы не увидите.

В качестве упражнения создадим несколько эффектных многоугольных окон, после чего вы сможете самостоятельно вырубать окна не хуже самого Петра Великого.

Для начала в процедуре **FormCreate** после строчки *area: TRect;* объявим несколько нужных *массивов*:

```
Triangle: array [0..2] of TPoint;        //- треугольник
Trapezoid: array [0..3] of TPoint;     //- трапеция
Arrow: array [0..6] of TPoint;           //- стрелка
Diamond: array [0..3] of TPoint;      //- ромб
Star: array [0..9] of TPoint;             //- звезда

//вспомогательные переменные для вычисления вершин звезды
Side, BottomOff, BaseTri, SideTri, HeightTri: Integer;
```

Кроме того, потребуется ещё модуль **Math**, который нужно дописать в раздел *uses*.

Начнём с самой простой фигуры – *треугольника*. В том же методе **FormCreate** заполним массив **Triangle** координатами его вершин (предыдущий код нужно закомментировать, за исключением последней строки *SetWindowRgn(Handle, rgn, True);*, которая необходима для всех фигур):

```
//треугольник
Triangle[0]:= Point(Width div 2, 0);
Triangle[1]:= Point(Width, Height);
Triangle[2]:= Point(0, Height);

rgn:= CreatePolygonRgn (Triangle, 3, WINDING);
```

Запустите программу – всё, как мы и обещали: окно приобрело прелестный, именно треугольный вид (Рис. П1.5).

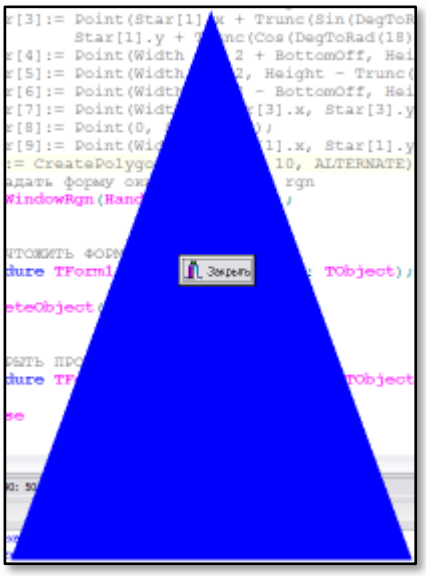


Рис. П1.5. Треугольное окно

Совершенно по той же схеме нужно действовать и при построении других фигур (Рис. П1.6-1.9). Выбирайте на свой вкус!

```
//трапецоид
Trapezoid[0]:= Point(Width div 2 div 2, 0);
Trapezoid[1]:= Point(Width - Width div 2 div 2, 0);
Trapezoid[2]:= Point(Width, Height);
Trapezoid[3]:= Point(0, Height);

rgn:= CreatePolygonRgn (Trapezoid, 4, WINDING);
```

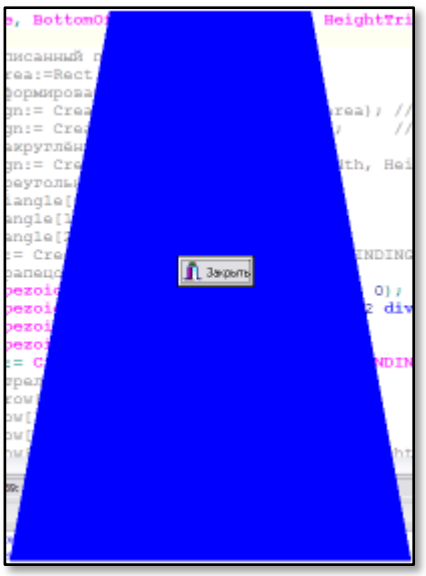


Рис. П1.6. Трапециевидное окно

```
//стрелка
Arrow[0]:= Point(0, Height div 3);
Arrow[1]:= Point(Width div 2, 0);
Arrow[2]:= Point(Width, Height div 3);
Arrow[3]:= Point(Width - Width div 4, Height div 3);
Arrow[4]:= Point(Width - Width div 4, Height);
Arrow[5]:= Point(Width div 4, Height);
Arrow[6]:= Point(Width div 4, Height div 3);

rgn:= CreatePolygonRgn (Arrow, 7, WINDING);
```

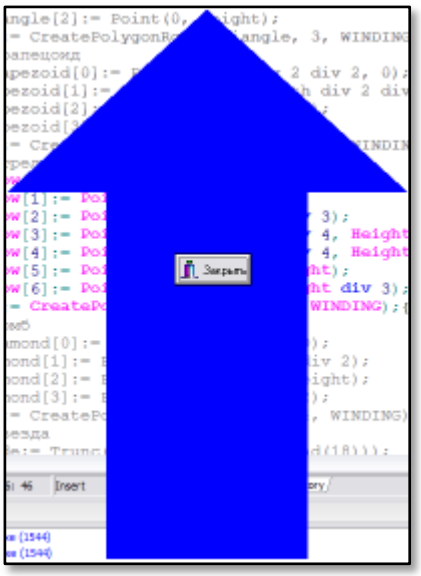


Рис. П1.7. Окно-стрелка

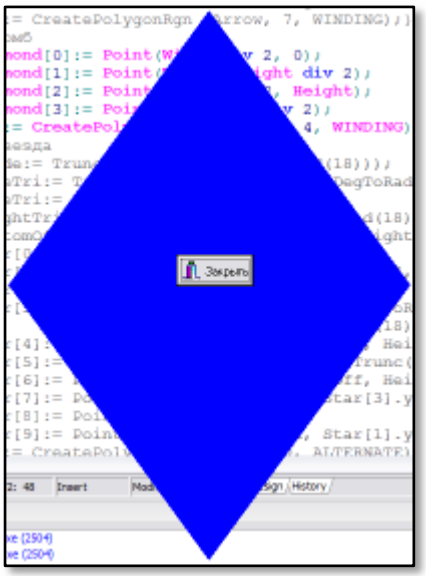


Рис. П1.8. Окно-ромб

```
//ромб
Diamond[0]:= Point(Width div 2, 0);
Diamond[1]:= Point(Width, Height div 2);
Diamond[2]:= Point(Width div 2, Height);
Diamond[3]:= Point(0, Height div 2);

rgn:= CreatePolygonRgn (Diamond, 4, WINDING);
```

И, наконец, апофеоз совершенства форм – наша *звезда*:

```
//звезда
Side:= Trunc(Height / Cos(DegToRad(18)));
SideTri:= Trunc(Side / (2 + 2 * Sin(DegToRad(18))));
BaseTri:= Side - 2 * SideTri;
HeightTri:= Trunc(SideTri * Cos(DegToRad(18)));
BottomOff:= Trunc(Tan(DegToRad(18)) * Height);
Star[0]:= Point(Width div 2, 0);
Star[1]:= Point(Width div 2 + BaseTri div 2, HeightTri);
Star[2]:= Point(Width, Star[1].y);
Star[3]:= Point(Star[1].x + Trunc(Sin(DegToRad(18)) * BaseTri),
Star[1].y + Trunc(Cos(DegToRad(18)) * BaseTri));
Star[4]:= Point(Width div 2 + BottomOff, Height);
```

```
Star[5]:= Point(Width div 2, Height - Trunc(Sin(DegToRad(36)) *
SideTri));
Star[6]:= Point(Width div 2 - BottomOff, Height);
Star[7]:= Point(Width - Star[3].x, Star[3].y);
Star[8]:= Point(0, Star[2].y);
Star[9]:= Point(Width - Star[1].x, Star[1].y);

rgn:= CreatePolygonRgn (Star, 10, ALTERNATE);
```

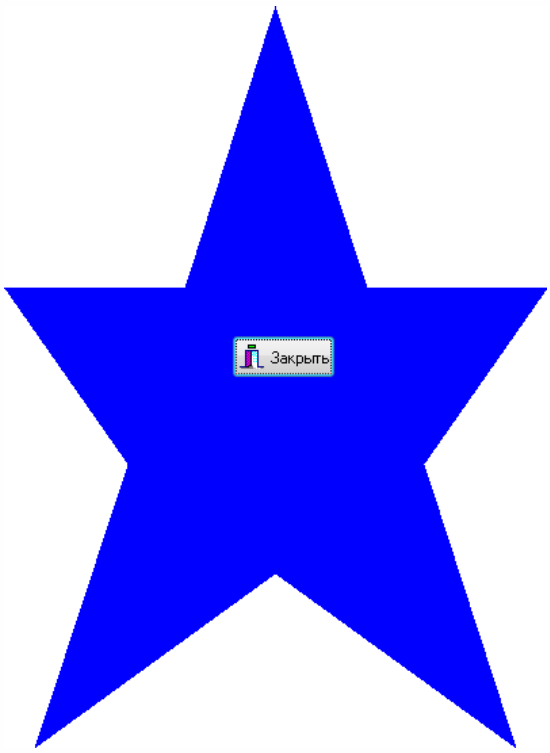


Рис. П1.9. Окно-звезда

Ещё более мудрёные, изысканные и даже «ажурные» окна можно получить, объединяя фигурные области с помощью ещё одной функции *Windows API*:

```
function CombineRgn (hrgnDest,
                     hrgnSrc1,
                     hrgnSrc2: HRGN;
                     fnCombineMode: integer): integer;,
```

которая разными способами комбинирует две области и сохраняет результат в третьей (эти области не обязательно должны быть различными, например, **hrgnSrc1** одновременно может быть и **hrgnDest**).

Здесь:

hrgnDest – новая область, в которую помещается результат комбинирования двух исходных; она должна существовать до вызова функции;

hrgnSrc1 – первая из двух областей;

hrgnSrc2 – вторая область;

fnCombineMode – режим объединения областей:

RGN_AND - пересечение областей, в **hrgnDest** помещается общая часть **hrgnSrc1**и **hrgnSrc2**.

RGN_COPY - копирование **hrgnSrc1** в **hrgnDest**.

RGN_DIFF - обрезка, область **hrgnSrc2** вырезает из **hrgnSrc1** общую часть, остаток **hrgnSrc2** помещается в **hrgnDest**.

RGN_OR – объединение двух областей.

RGN_XOR - в **hrgnDest** помещаются **hrgnSrc1** и **hrgnSrc2**, за исключением области пересечения.

Возвращаемые значения:

NULLREGION – пустая область.
SIMPLEREGION - прямоугольная область.
COMPLEXREGION – область имеет более сложную форму, чем прямоугольник.
ERROR – область не создана.

Воспользуемся этой функцией, чтобы получить прелестное прорезное окно, всё – в звёздах (Рис. П1.10).

Как всегда, задайте **BorderStyle** = bsNone, чтобы заголовок формы не портил вид из окна.

Пусть окно располагается в центре стола, как праздничный торт:

Position = poDesktopCenter.

Цвет тоже сгодится праздничный:

Color = clRed.

Размеры:

Width = 600, **Height** = 400.

Для закрытия окна потребуется одна небольшая кнопочка **SpeedButton1**: *TSpeedButton* (**Width** = 21, **Height** = 21, **Left** = 285, **Top** = 286 – это важно, чтобы кнопка не «провалилась» в прорези). Осталось добавить немного кода:

```
unit shaping2;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
Dialogs, StdCtrls, Buttons;

type
  TForm1 = class(TForm)
    SpeedButton1: TSpeedButton;
    procedure FormCreate(Sender: TObject);
    procedure SpeedButton1Click(Sender: TObject);

  private
    { Private declarations }

  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation

{$R *.DFM}

//СОЗДАТЬ ФОРМУ
procedure TForm1.FormCreate(Sender: TObject);
const
  fi = pi/5;

var
  Rgn,Rgn1,Rgn2: HRgn;
  X,Y,i:integer;

  function S(alpha:integer;R:integer):integer;
  begin
    Result:=round(R*sin(fi*alpha));
  end;
```

```
  function C(a:integer;R:integer):integer;
  begin
    Result:=round(R*cos(fi*a));
  end;

  function CreateStarReg(X,Y,R:integer):HRGN;
  //R - радиус описанной окружности
  var  P : array [0..4] of TPoint;
  begin
    P[0] := Point(X, Y-R);
    P[1] := Point(X-S(4,R), Y-C(4,R));
    P[2] := Point(X-S(8,R), Y-C(8,R));
    P[3] := Point(X-S(2,R), Y-C(2,R));
    P[4] := Point(X-S(6,R), Y-C(6,R));
    //Result := CreatePolygonRgn(P, 5, ALTERNATE);
    Result := CreatePolygonRgn(P, 5, WINDING);
  end;

  begin
    //координаты центра формы:
    X:=Width div 2;
    Y:=Height div 2;

    //создать звёздчатую область
    Rgn:=CreateStarReg(X,Y,100);

    //добавить 5 маленьких звёзд вокруг большой звезды
    i:=1;
    repeat
      Rgn1:=CreateStarReg(X-S(i,120),Y-C(i,110),40);
      CombineRgn(Rgn,Rgn,Rgn1,RGN_OR);
      inc(i,2);
    until i>9;

    //вырезать в центральной звезде звёздчатое отверстие
    Rgn1:=CreateStarReg(X,Y,30);
    CombineRgn(Rgn,Rgn,Rgn1,RGN_DIFF);

    //создать 2 эллиптические области
    Rgn1:=CreateEllipticRgn(0,0,Width,Height);
    Rgn2:=CreateEllipticRgn(20,10,Width-20,Height-10);

    //и в большом эллипсе прорезать эллиптическое отверстие
    CombineRgn(Rgn1,Rgn1,Rgn2,RGN_DIFF);
    CombineRgn(Rgn,Rgn,Rgn1,RGN_OR);

    //вывести на экран окно по результирующей области
    SetWindowRgn(Handle, Rgn, True);
end;

procedure TForm1.SpeedButton1Click(Sender: TObject);
begin
```

```
  close
end;

end.
```



Рис. П1.10. Окно со звёздами

Кстати, вы можете почувствовать разницу между режимами закраски области **fnPolyFillMode**:

в функции **CreateStarReg** в строчке `Result:= CreatePolygonRgn` **WINDING** замените **ALTERNATE** и запустите программу. Теперь звёздочки получились с пятиугольными отверстиями внутри (Рис. П1.11).



Рис. П1.11. Другое окно со звёздами

Это происходит потому, что в данном примере звёзды задаются только пятью вершинами, которые соединяются так, как мы это привыкли делать с детства (Рис. П1.12).

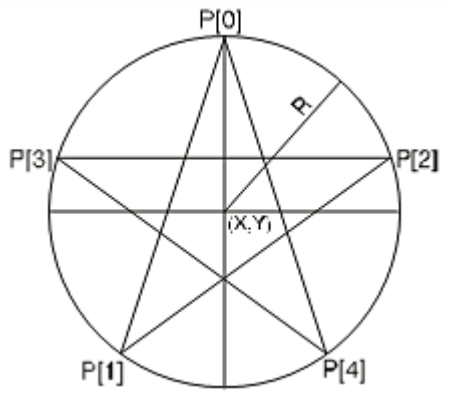


Рис. П1.12. Как начертить звезду

При закраске смежных сторон в режиме **ALTERNATE** центральная часть звезды останется не закрашенной.

Следует отметить, что можно не только задавать форму окна при его создании, но и динамически изменять в работающей программе, так что нетрудно создать приложение с окном-трансформером, изменяющим свою форму в зависимости от действия пользователя. Для примера объединим наши разрозненные действия по созданию разнообразных форм в одну программу.

Откроем новый проект (**shaping3**) и придадим форме подходящий вид:

```
BorderStyle = bsNone;
Color = clFuchsia;
Height = 576;
Position = poDesktopCenter;
Width = 409.
```

Разместим на ней **кнопки**:

```
bbtClose: TBitBtn;
Caption = 'Закрыть';
Kind = bkClose;
Left = 164;
Top = 180;
Width = 77;
Height = 24;
```

Установим **Height** = 25, **Width** = 93, **Left** = 156 у всех остальных кнопок.

```
sbtTriangle: TSpeedButton;
Caption = 'Треугольник'; Top = 212;
```

```
sbtRect: TSpeedButton;
Caption = 'Прямоугольник'; Top = 240;

sbtRoundRect: TSpeedButton;
Caption = 'Закругл. пр-к'; Top = 268;

sbtEllipse: TSpeedButton;
Caption = 'Эллипс'; Top = 296;

sbtTrapezoid: TSpeedButton;
Caption = 'Трапеция'; Top = 324;

sbtArrow: TSpeedButton;
Caption = 'Стрелка'; Top = 352;

sbtDiamond: TSpeedButton;
Caption = 'Ромб'; Top = 376;

sbtStar: TSpeedButton;
Caption = 'Звезда'; Top = 400;
```

Зададим **Cursor = crHandPoint** – для всех кнопок.

Теперь остаётся только добавить код для обработки нажатия этих кнопок, после чего вы сможете выбирать для своего окна любую форму.

```
unit shaping3;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
Dialogs, StdCtrls, Buttons, Math;

type
  TForm1 = class(TForm)
    bbtClose: TBitBtn;
    sbtTriangle: TSpeedButton;
    sbtRect: TSpeedButton;
    sbtRoundRect: TSpeedButton;
    sbtEllipse: TSpeedButton;
    sbtTrapezoid: TSpeedButton;
    sbtArrow: TSpeedButton;
    sbtDiamond: TSpeedButton;
    sbtStar: TSpeedButton;
    procedure bbtCloseClick(Sender: TObject);
    procedure FormDestroy(Sender: TObject);
    procedure sbtTriangleClick(Sender: TObject);
    procedure sbtRectClick(Sender: TObject);
    procedure sbtRoundRectClick(Sender: TObject);
    procedure sbtEllipseClick(Sender: TObject);
```

```
    procedure sbtTrapezoidClick(Sender: TObject);
    procedure sbtArrowClick(Sender: TObject);
    procedure sbtDiamondClick(Sender: TObject);
    procedure sbtStarClick(Sender: TObject);

  private
    { Private declarations }
    rgn: HRGN;

  public
    { Public declarations }
end;

var
  Form1: TForm1;

implementation

{$R *.DFM}

//УНИЧТОЖИТЬ ФОРМУ
procedure TForm1.FormDestroy(Sender: TObject);
begin
  DeleteObject(rgn)
end;

//ЗАКРЫТЬ ПРОГРАММУ
procedure TForm1.bbtCloseClick(Sender: TObject);
begin
  close
end;

//треугольник
procedure TForm1.sbtTriangleClick(Sender: TObject);
var
  Triangle: array [0..2] of TPoint;
begin
  Triangle[0]:= Point(Width div 2, 0);
  Triangle[1]:= Point(Width, Height);
  Triangle[2]:= Point(0, Height);

  rgn:= CreatePolygonRgn (Triangle, 3, WINDING);
  //задать форму окна по области rgn
  SetWindowRgn(Handle, rgn, True);
end;

//прямоугольник
procedure TForm1.sbtRectClick(Sender: TObject);
var
  area: TRect;
begin
  area:=Rect(0,0,Width,Height);
```

```
  rgn:= CreateRectRgnIndirect (area);
  SetWindowRgn(Handle, rgn, True);
end;

//закруглённый прямоугольник
procedure TForm1.sbtRoundRectClick(Sender: TObject);
begin
  rgn:= CreateRoundRectRgn (0, 0, Width, Height, 100,100);
  SetWindowRgn(Handle, rgn, True);
end;

//эллиптическая область
procedure TForm1.sbtEllipseClick(Sender: TObject);
var
  area: TRect;
begin
  //описанный прямоугольник
  area:=Rect(0,0,Width,Height);

  //сформировать область
  rgn:= CreateEllipticRgnIndirect (area);
  SetWindowRgn(Handle, rgn, True);
end;

//трапеция
procedure TForm1.sbtTrapezoidClick(Sender: TObject);
var
  Trapezoid: array [0..3] of TPoint;
begin
  Trapezoid[0]:= Point(Width div 2 div 2, 0);
  Trapezoid[1]:= Point(Width - Width div 2 div 2, 0);
  Trapezoid[2]:= Point(Width, Height);
  Trapezoid[3]:= Point(0, Height);

  rgn:= CreatePolygonRgn (Trapezoid, 4, WINDING);
  SetWindowRgn(Handle, rgn, True);
end;

//стрелка
procedure TForm1.sbtArrowClick(Sender: TObject);
var
  Arrow: array [0..6] of TPoint;
begin
  Arrow[0]:= Point(0, Height div 3);
  Arrow[1]:= Point(Width div 2, 0);
  Arrow[2]:= Point(Width, Height div 3);
  Arrow[3]:= Point(Width - Width div 4, Height div 3);
  Arrow[4]:= Point(Width - Width div 4, Height);
  Arrow[5]:= Point(Width div 4, Height);
  Arrow[6]:= Point(Width div 4, Height div 3);
  rgn:= CreatePolygonRgn (Arrow, 7, WINDING);
```

```
  SetWindowRgn(Handle, rgn, True);
end;

//ромб
procedure TForm1.sbtDiamondClick(Sender: TObject);
var
  Diamond: array [0..3] of TPoint;
begin
  Diamond[0]:= Point(Width div 2, 0);
  Diamond[1]:= Point(Width, Height div 2);
  Diamond[2]:= Point(Width div 2, Height);
  Diamond[3]:= Point(0, Height div 2);
  rgn:= CreatePolygonRgn (Diamond, 4, WINDING);
  SetWindowRgn(Handle, rgn, True);
end;

//звезда
procedure TForm1.sbtStarClick(Sender: TObject);
var
  Star: array [0..9] of TPoint;
  Side, BottomOff, BaseTri, SideTri, HeightTri: Integer;
begin
  Side:= Trunc(Height / Cos(DegToRad(18)));
  SideTri:= Trunc(Side / (2 + 2 * Sin(DegToRad(18))));
  BaseTri:= Side - 2 * SideTri;
  HeightTri:= Trunc(SideTri * Cos(DegToRad(18)));
  BottomOff:= Trunc(Tan(DegToRad(18)) * Height);
  Star[0]:= Point(Width div 2, 0);
  Star[1]:= Point(Width div 2 + BaseTri div 2, HeightTri);
  Star[2]:= Point(Width, Star[1].y);
  Star[3]:= Point(Star[1].x + Trunc(Sin(DegToRad(18)) * BaseTri),
Star[1].y + Trunc(Cos(DegToRad(18)) * BaseTri));
  Star[4]:= Point(Width div 2 + BottomOff, Height);
  Star[5]:= Point(Width div 2, Height - Trunc(Sin(DegToRad(36)) *
SideTri));
  Star[6]:= Point(Width div 2 - BottomOff, Height);
  Star[7]:= Point(Width - Star[3].x, Star[3].y);
  Star[8]:= Point(0, Star[2].y);
  Star[9]:= Point(Width - Star[1].x, Star[1].y);

  rgn:= CreatePolygonRgn (Star, 10, ALTERNATE);
  SetWindowRgn(Handle, rgn, True);
end;

end.
```

Поскольку все фигурные окна не имеют заголовка (при **BorderStyle**=bsNone, иначе корявая часть заголовка останется), то их невозможно передвигать, взявшись за него мышкой. Но есть несколько способов преодолеть это препятствие. Один из них вы найдёте в игре *Mind Game*. Специально для ленивых – ещё один вариант:

```
//перетаскивание формы за её поле
procedure TForm1.FormMouseDown(Sender: TObject; Button: TMouseBut-
ton; Shift: TShiftState; X, Y: Integer);
const
  SC_DragMove = $F012;
begin
  ReleaseCapture;
  Perform(WM_SysCommand, SC_DragMove, 0)
end;
```

Добавьте эту процедуру к своему проекту – и таскать вам не перетаскать!

Вот бледный образ одного из возможных окон (Рис. П1.13).



Рис. П1.13. Фигурные окна

Часто бывает необходимо узнать, находится ли точка в заданной области. Например, в игре *Mind Game* при нажатии кнопки мышки в области формы последнюю можно передвинуть на другое место. В этом случае удобно использовать специальную функцию для работы с областями

```
function  PtInRegion (hrgn: HRGN;
                      X, Y: integer): BOOL;,
```

где

hrgn – описатель области, для которой проводится проверка,
X, Y – координаты точки.

Возвращаемые значения:
Если заданная точка находится в области, то функция возвращает **TRUE** (ненулевую величину). В противном случае – **FALSE** (нуль).

Если вас угнетает отсутствие заголовка в фигурном окне, то можно сымитировать его, подставив сверху подходящую область. Правда, придётся потрудиться, да что поделаешь – новизна ощущений того стоит (Рис. П1.14)!

```
unit shaping5;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
Dialogs, Buttons, StdCtrls;

type
  TForm1 = class(TForm)
    procedure FormCreate(Sender: TObject);
    procedure Button1Click(Sender: TObject);
    procedure FormPaint(Sender: TObject);

  private
    { Private declarations }
    rTitleBar : THandle;
    Center    : TPoint;
    CapY      : Integer;
    SB1       : TSpeedButton;
    RL, RR    : Double;
    procedure TitleBar(Act : Boolean);
    procedure WMNCHITTEST(var Msg: TWMNCHitTest); message
WM_NCHITTEST;
    procedure WMNCACTIVATE(var Msg: TWMNCACTIVATE);  message
WM_NCACTIVATE;
    procedure WMSetText(var Msg: TWMSetText);  message WM_SETTEXT;

  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation

{$R *.DFM}

CONST
  //TitlColors : ARRAY[Boolean] OF TColor = clInactiveCaption,
clActiveCaption);
  TitlColors : ARRAY[Boolean] OF TColor = (clGray, clRed);
  TxtColors : ARRAY[Boolean] OF TColor = (clInactiveCaptionText,
clCaptionText);

//создать форму-бублик
procedure TForm1.FormCreate(Sender: TObject);
```

```
VAR
  rTemp, rTemp2: THandle;
  Vertices : ARRAY[0..2] OF TPoint;
  X, Y     : INteger;

begin
  Caption := '                    БУБЛИК - 2012';
  BorderStyle := bsNone;       //- убрать полосу заголовка
  Width:= 500; Height:= 500; //- размеры формы
  Color:= clBlue; //цвет формы

  //сделать форму квадратной:
  //IF Width > Height THEN Width := Height
  //ELSE Height := Width;

  //вычислить центр формы:
  Center  := Point(Width DIV 2, Height DIV 2);
  CapY := GetSystemMetrics(SM_CYCAPTION)+8;

  //создать круглую область по размерам формы:
  rTemp := CreateEllipticRgn(0, 0, Width, Height);

  //и круг меньшего размера:
  rTemp2 := CreateEllipticRgn((Width DIV 4), (Height DIV 4),
3*(Width DIV 4), 3*(Height DIV 4));

  //вырезать в большем круге круглое отверстие:
  CombineRgn(rTemp, rTemp, rTemp2, RGN_DIFF);

  //установить получившуюся область в качестве окна:
  SetWindowRgn(Handle, rTemp, True);

  DeleteObject(rTemp2); //- уничтожить переменную

  //сформировать собственную полоску заголовка:
  rTitleBar  := CreateEllipticRgn(4, 4, Width-4, Height-4);
  rTemp := CreateEllipticRgn(CapY, CapY, Width-CapY, Height-CapY);
  CombineRgn(rTitleBar, rTitleBar, rTemp, RGN_DIFF);
  Vertices[0] := Point(0,0);
  Vertices[1] := Point(Width, 0);
  Vertices[2] := Point(Width DIV 2, Height DIV 2);
  rTemp := CreatePolygonRgn(Vertices, 3, ALTERNATE);
  CombineRgn(rTitleBar, rTitleBar, rTemp, RGN_AND);
  DeleteObject(rTemp);

  //установить на полоске заголовка кнопку для закрытия окна:
  RL:= ArcTan(Width / Height);
  RR:= -RL + (22 / Center.X);
  X:= Center.X-Round((Center.X-1-(CapY DIV 2))*Sin(RR));
  Y:= Center.Y-Round((Center.Y-1-(CapY DIV 2))*Cos(RR));
  SB1:= TSpeedButton.Create(Self);
  WITH SB1 DO
```

```
    BEGIN
      Parent := Self;
      Left     := X;
      Top     := Y;
      Width  := 14;
      Height := 14;
      OnClick := Button1Click;
      Caption  := 'X';
      Font.Style := [fsBold];
    END;
end;

//процедура для кнопки на полосе заголовка
procedure TForm1.Button1Click(Sender: TObject);
begin
  Close;
End;

//перетаскивание окна за его заголовок
procedure TForm1.WMNCHITTEST(var Msg: TWMNCHitTest);
begin
  Inherited;
  WITH Msg DO
    WITH ScreenToClient(Point(XPos,YPos)) DO
      IF PtInRegion(rTitleBar, X, Y) AND (NOT
PtInRect(SB1.BoundsRect, Point(X,Y))) THEN
        Result := htCaption;
end;

procedure TForm1.WMNCActivate(var Msg: TWMncActivate);
begin
  Inherited;
  TitleBar(Msg.Active);
end;

procedure TForm1.WMSetText(var Msg: TWMSetText);
begin
  Inherited;
  TitleBar(Active);
end;

//прорисовать заголовок соответствующим цветом
procedure TForm1.TitleBar(Act: Boolean);
VAR
  TF : TLogFont;
  R  : Double;
  N, X, Y : Integer;
begin
  IF Center.X = 0 THEN Exit;

  WITH Canvas DO
    begin
```

```
      Brush.Style := bsSolid;
      Brush.Color := TitlColors[Act];
      PaintRgn(Handle, rTitleBar);
      R  := RL;
      Brush.Color := TitlColors[Act];
      Font.Name := 'Arial';
      Font.Size := 12;
      Font.Color := TxtColors[Act];
      Font.Style := [fsBold];
      GetObject(Font.Handle, SizeOf(TLogFont), @TF);

      FOR N := 1 TO Length(Caption) DO
        BEGIN
          X := Center.X-Round((Center.X-6)*Sin(R));
          Y := Center.Y-Round((Center.Y-6)*Cos(R));
          TF.lfEscapement := Round(R * 1800 / pi);
          Font.Handle := CreateFontIndirect(TF);
          TextOut(X, Y, Caption[N]);
          R := R - (((TextWidth(Caption[N]))+2) / Center.X);
          IF R < RR THEN Break;
        END;

      Font.Name := 'MS Sans Serif';
      Font.Size := 8;
      Font.Color := clWindowText;
      Font.Style := [];

    end;
end;

//прорисовать заголовок
procedure TForm1.FormPaint(Sender: TObject);
begin
  WITH Canvas DO
  BEGIN
      Pen.Color := clBlack;
      Brush.Style := bsClear;
      Pen.Width := 1;
      Pen.Color := clWhite;
      Arc(1, 1, Width-1, Height-1, Width, 0, 0, Height);
      Arc((Width DIV 4)-1, (Height DIV 4)-1,  3*(Width DIV 4)+1,
3*(Height DIV 4)+1, 0, Height, Width, 0);
      Pen.Color := clBlack;
      Arc(1, 1, Width-1, Height-1, 0, Height, Width, 0);
      Arc((Width DIV 4)-1, (Height DIV 4)-1,  3*(Width DIV 4)+1,
3*(Height DIV 4)+1, Width, 0, 0, Height);
      TitleBar(Active);
    END;
end;

end.
```

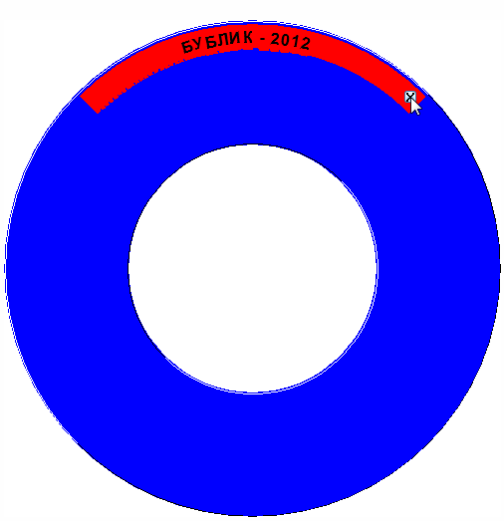


Рис. П1.14. Окно-бублик

Хорошенько разобравшись в этой программе, вы найдёте немало полезного для своих программ. Только не забудьте связать события **OnCreate** и **OnPaint** с **FormCreate** и **FormPaint**. А также разместите на форме кнопку **Button1** типа *TButton*, щёлкните на ней дважды, напишите в открывшейся заготовке процедуры **close**, а затем удалите кнопку с формы – она не нужна, её роль будет выполнять динамически создаваемая кнопка в заголовке формы.



Исходный код программы находится в папке **Shaping**.

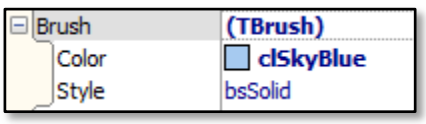
Художественное «выпиливание»

Почти тех же результатов можно добиться и средствами самого *Delphi*, используя свойства формы **TransparentColor** и **TransparentColorValue**. Если значение **TransparentColor** = TRUE, то цвет, заданный в свойстве **TransparentColor** будет прозрачным.

Откроем новый проект.

Разместим на форме компонент 

с такими свойствами (Рис. П1.15).



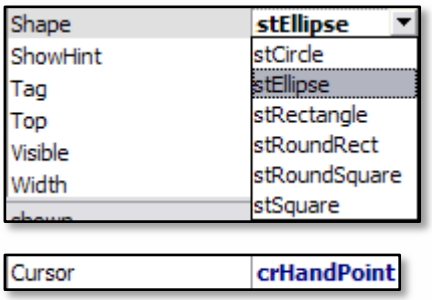


Рис. П1.15. Фигурные свойства

Цвет, стиль закраски, форму фигуры и её размеры вы можете выбрать любыми.

Уберём у формы *заголовок* (Рис. П1.16).

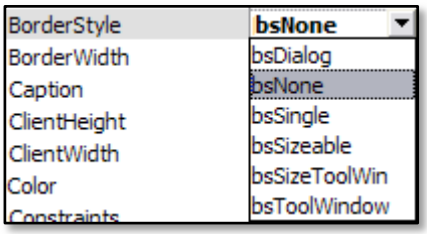


Рис. П1.16. Обезглавливаем окно

Окрасим форму в такой цвет, который заведомо не будет встречаться у других компонентов (Рис. П1.17).



Рис. П1.17. Окрашиваем форму фуксином

И самое главное – сделаем этот цвет прозрачным (рис. П1.18).

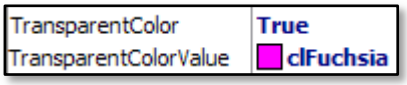


Рис. П1.18. Устанавливаем прозрачный цвет

Также нам потребуется кнопка, чтобы закрывать приложение. В итоге в *Конструкторе формы* должна получиться примерно такая картина (Рис. П1.19).

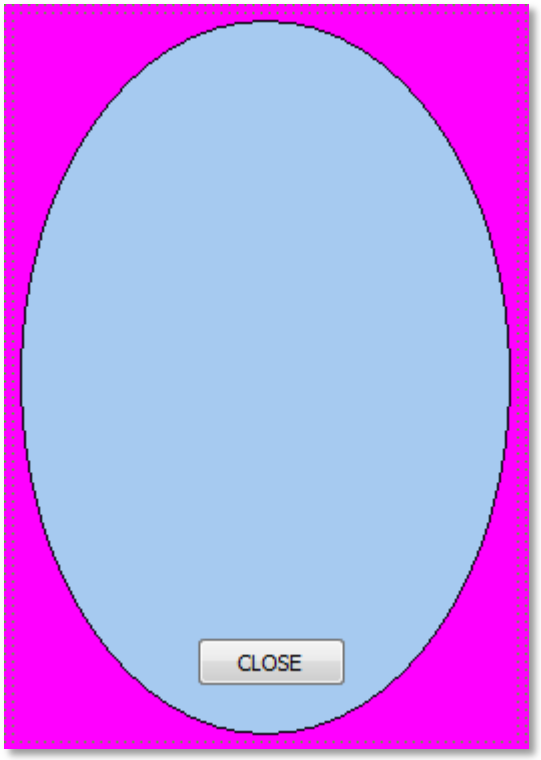


Рис. П1.19. Конструируем форму

Перейдём в *Редактор кода*.

Так как у нашего окна нет заголовка, то перетаскивать его придётся за фигуру:

```
//ПЕРЕТАСКИВАНИЕ ФОРМЫ
procedure TForm1.Shape1MouseDown(Sender: TObject; Button: TMouse-
Button;
  Shift: TShiftState; X, Y: Integer);
const
  SC_DragMove = $F012;
begin
  ReleaseCapture;
  Perform(WM_SysCommand, SC_DragMove, 0)
end; //Shape1MouseDown
```

Запускаем приложение (Рис. П1.20).

Как видите, результат точно такой же, что и при использовании функции *Windows API* **CreateEllipticRgn**, но нам не пришлось заниматься вычислениями, всё приложение мы создали совершенно «визуально»!

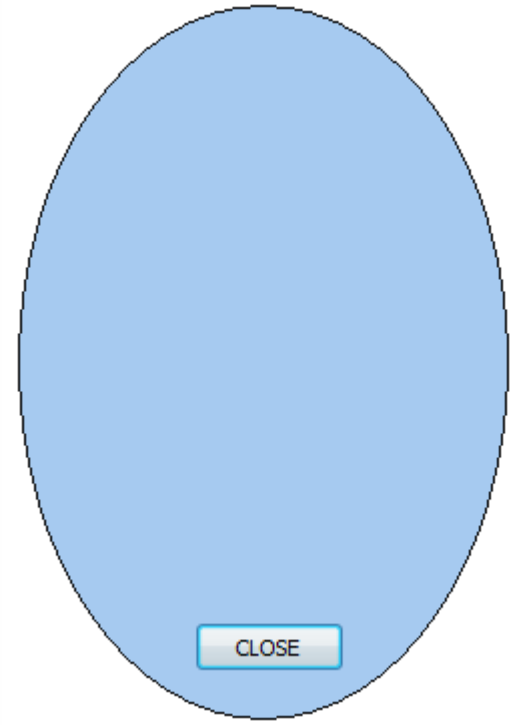


Рис. П1.20. Эллиптическое окно – вторая версия

Точно так же можно вместо эллиптической фигуры выбрать прямоугольную или объединить несколько фигур в одну.

Поместим на форму ещё одну фигуру, для этого скопируем первую в буфер обмена и вставим на форму. В результате этого действия сохранятся все свойства первой фигуры. Но для разнообразия пусть вторая фигура будет прямоугольной (Рис. П1.21).

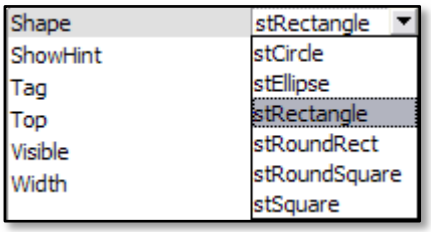


Рис. П1.21. Изменяем форму окна

Всё бы хорошо, но линии по границам фигур мешают объединению фигур (Рис. П1.22).

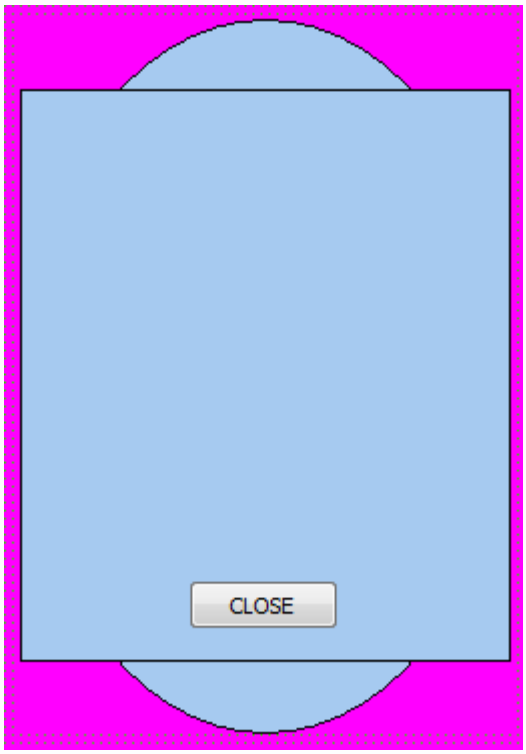


Рис. П1.22. Лишние линии

Поэтому удалим границы, установив их свойство **Pen > Style** в *psClear* (Рис. П1.23).

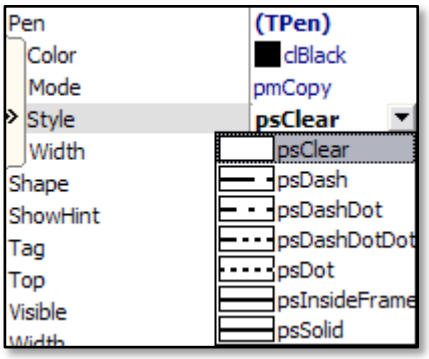


Рис. П1.23. Удаляем границы

Теперь всё нормально (Рис. П1.24).

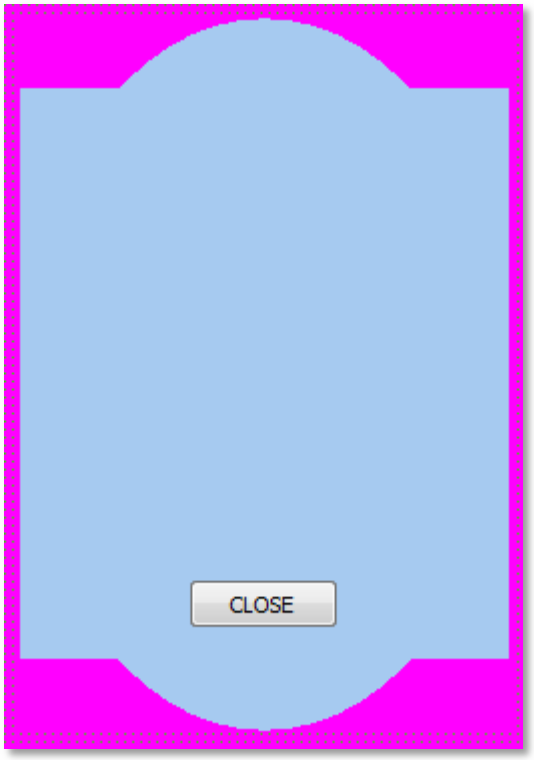


Рис. П1.24. Единое объединение

Запускаем программу (Рис. П1.25).

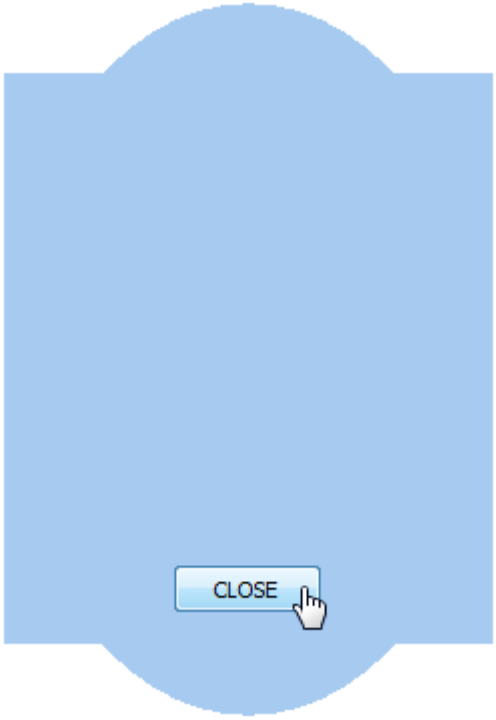


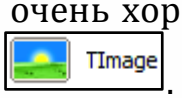
Рис. П1.25. Окно получилось весьма фигуристым!

При желании можно в окне приложения вырезать *отверстия*. Сделать это очень просто – достаточно поместить на форму ещё одну фигуру и закрасить её цветом формы (Рис. Р1.26).



Рис. П1.26. Форма в *Редакторе кода* Окно приложения

Тут нужно признать, что функции *Windows API* имеют больше возможностей по созданию областей сложной формы, чем средства *Delphi*, но во многих случаях и их вполне достаточно.

Ну, а если хочется большего? – Всё тоже достаточно просто – мы можем нарисовать любую фигуру в каком-нибудь графическом редакторе (например, очень хорош для этого *CorelDraw*) и загрузить изображение в компонент 

.

Сделаем это в новом проекте. Уберём все фигурки и вместо них положим на форму компонент **TImage.** Нарисуем любую фигурку и сохраним, например, в векторном формате *EMF*. Загрузим в свойство **Picture** этого компонента нашу картинку (Рис. П1.27).

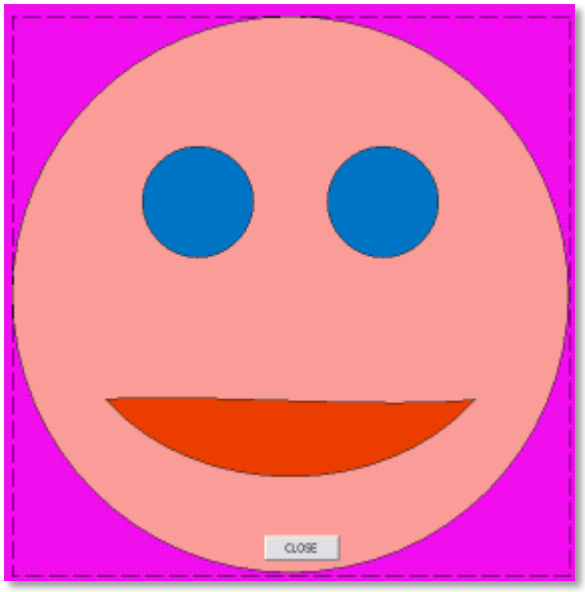


Рис. П1.27. Картинная форма

Немного подправим код:

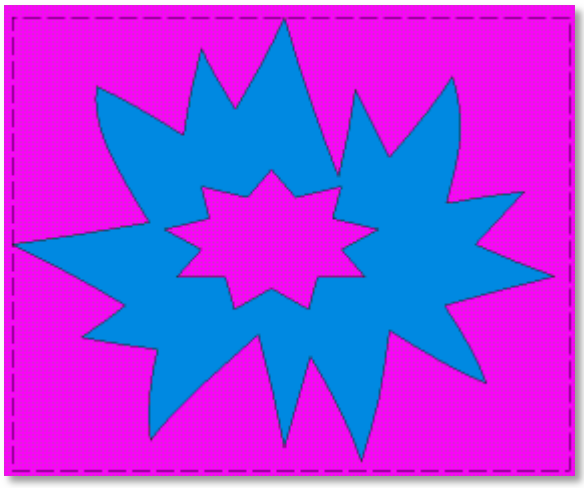
```
//ПЕРЕТАСКИВАНИЕ ФОРМЫ
procedure TForm1.Image1MouseDown(Sender: TObject; Button: TMouse-
Button;  Shift: TShiftState; X, Y: Integer);
const
  SC_DragMove = $F012;
begin
  ReleaseCapture;
  Perform(WM_SysCommand, SC_DragMove, 0)
end;
```

Можно запускать программу (Рис. П1.28).



Рис. П1.28. Окно-колобок

Конечно, круглый колобок, хоть и с глазами и ртом, не производит должного впечатления, но кто мешает нам нарисовать более сложную фигуру (Рис. П.1.29)?



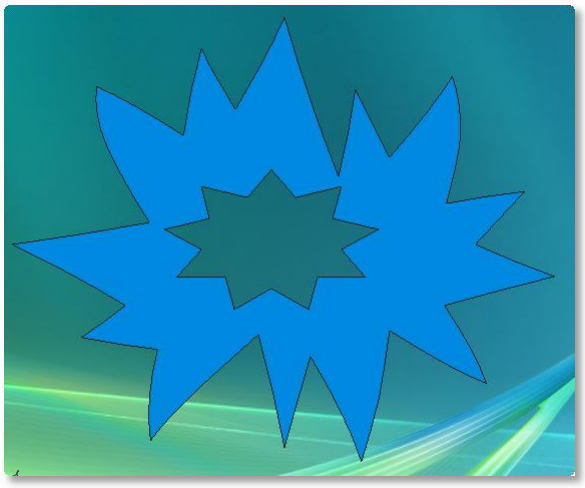


Рис. П1.29. Форма в *Редакторе кода* Окно приложения

Уберём с формы кнопку, чтобы она не портила вид художественного произведения, а её функции передадим компоненту *TImage*. В обработчике нажатия кнопки мыши на картинке определяем, какая кнопка нажата – левая или правая – и соответственно этому либо перетаскиваем форму, либо закрываем её:

```
//ПЕРЕТАСКИВАНИЕ ФОРМЫ
procedure TForm1.Image1MouseDown(Sender: TObject; Button: TMouse-
Button; Shift: TShiftState; X, Y: Integer);
const
  SC_DragMove = $F012;
begin
  if Button=mbLeft then begin
     ReleaseCapture;
     Perform(WM_SysCommand, SC_DragMove, 0)
  end
  else close;
end;
```

Конечно, вы можете использовать не только векторные картинки, но и растровые. Вид окон будет практически один и тот же, однако сохранять картинки лучше в формате *BMP* и без сглаживания краёв (опция *Anti-aliasing* должна быть выключена), чтобы по границам окна не появился светлый ореол из посторонних пикселей (Рис. П1.30-31). В крайнем случае, вы можете вручную почистить края картинки…

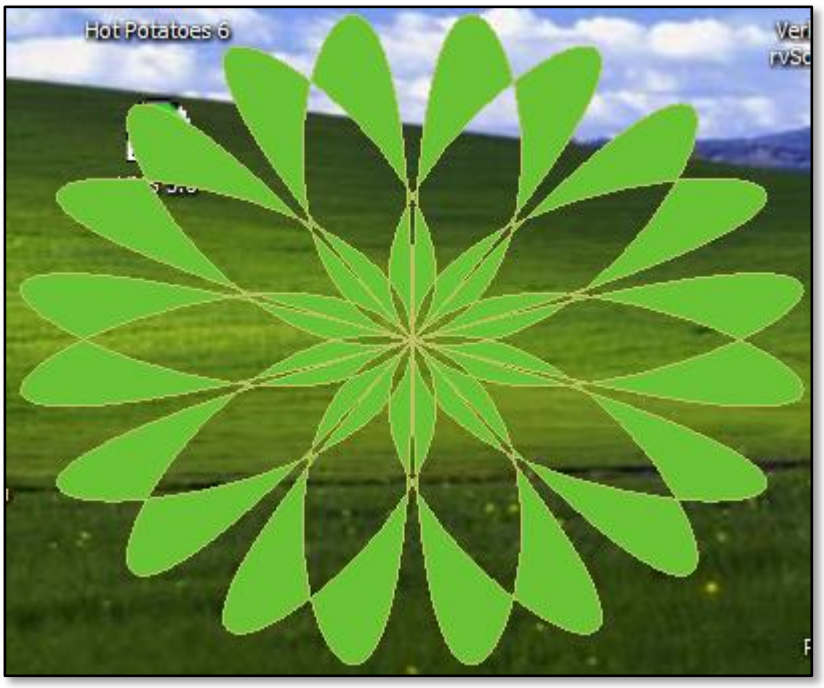


Рис. П1.30. Картинка в векторном формате *EMF*



Рис. П1.31. Картинка в растровом формате *BMP*

Ещё пара примеров фигурных окон, словно выпиленных лобзиком (Рис. П1.32-33).



Рис. П1.32. Турбо-окно

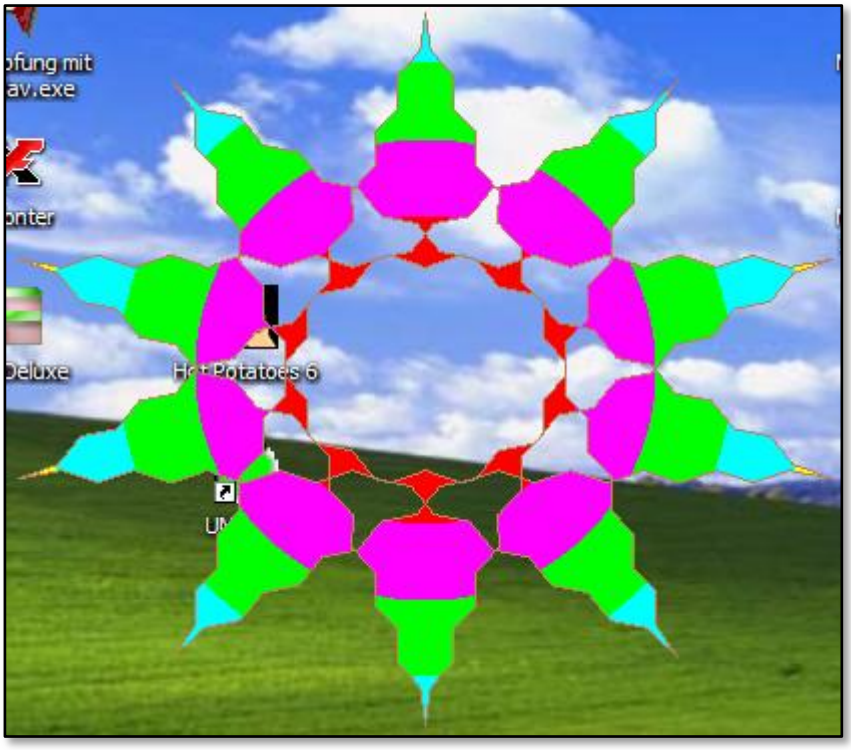


Рис. П1.33. Лучистое окно

Я надеюсь, что эти примеры убедили вас, что красивую форму можно сделать средствами самого *Delphi* и не обладая высокими художественными способностями. Однако хотел бы вас предостеречь от бездумного применения в своих проектах окон вычурной формы – такие визуальные экспе-

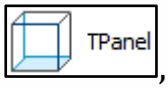
рименты редко бывают удачными. Лучше всего использовать такие окна в формах-заставках.



Исходный код программы находится в папке **FormForm**.

Форточки и панелевозы

Как мы уже упоминали, готовую область можно назначить любому оконному компоненту, а не только форме.

Для примера возьмём очень полезный компонент **TPanel** 

, хотя для создания окон на главном окне (то есть форточек) сгодятся и другие оконные компоненты: **TButton, TFlowPanel, TGridPanel, TPageControl, TPageScroller**.

За основу нового проекта возьмём программу **Shaping3** для формы. И – лёгким движением руки панель превращается в элегантные фигуры, которые можно запросто двигать по форме (Рис. П1.34).

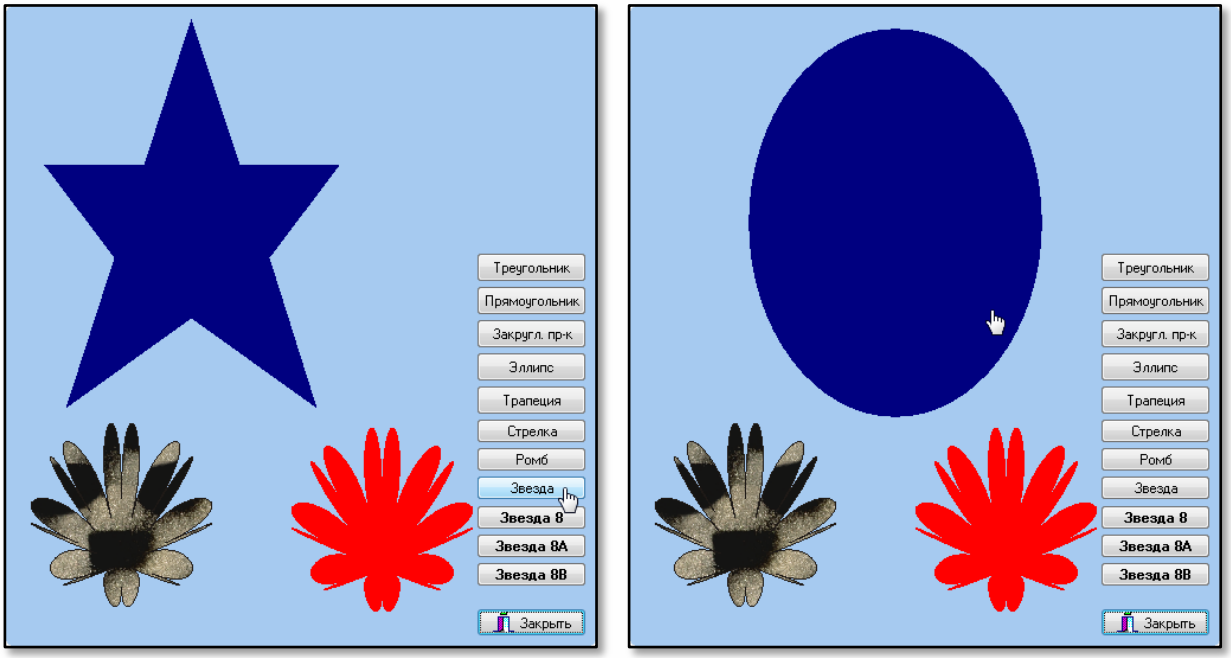


Рис. П1.34. Панели можно придать любую форму и свободно перемещать её в пределах окна приложения

Правда, у панели нет таких приятных свойств, как **TransparentColor** и **TransparentColorValue**, поэтому новые формы панелям можно придавать только с помощью областей, но в следующей главе мы рассмотрим уни-

версальный способ, который поможет нам без особых усилий обрабатывать панель в соответствии с нашими высокими художественными запросами. Зато сколько возможностей нам открывают фигурные движущиеся панели, а ведь на них ещё можно наложить и разные картинки!

Ну, и наконец, изменённый код программы:

```
//треугольник
procedure TForm1.sbtTriangleClick(Sender: TObject);
var
  Triangle: array [0..2] of TPoint;
begin
  with Panel1 do begin
    Triangle[0]:= Point(Width div 2, 0);
    Triangle[1]:= Point(Width, Height);
    Triangle[2]:= Point(0, Height);
    rgn:= CreatePolygonRgn (Triangle, 3, WINDING);
    //BringToFront;
    //задать форму окна по области rgn:
    SetWindowRgn(Handle, rgn, True);
  end;
end;
//прямоугольник
procedure TForm1.sbtRectClick(Sender: TObject);
var
  area: TRect;
begin
  with Panel1 do begin
    area:=Rect(0,0,Width,Height);
    rgn:= CreateRectRgnIndirect (area);
    SetWindowRgn(Handle, rgn, True);
  end;
end;
//закруглённый прямоугольник
procedure TForm1.sbtRoundRectClick(Sender: TObject);
begin
  with Panel1 do begin
    rgn:= CreateRoundRectRgn (0, 0, Width, Height, 100,100);
    SetWindowRgn(Handle, rgn, True);
  end;
end;
//эллиптическая область
procedure TForm1.sbtEllipseClick(Sender: TObject);
var
  area: TRect;
begin
  with Panel1 do begin
    //описанный прямоугольник
    area:=Rect(0,0,Width,Height);
    //сформировать область
    rgn:= CreateEllipticRgnIndirect (area);
```

```
    SetWindowRgn(Handle, rgn, True);
  end;
end;
//трапеция
procedure TForm1.sbtTrapezoidClick(Sender: TObject);
var
  Trapezoid: array [0..3] of TPoint;
begin
  with Panel1 do begin
    Trapezoid[0]:= Point(Width div 2 div 2, 0);
    Trapezoid[1]:= Point(Width - Width div 2 div 2, 0);
    Trapezoid[2]:= Point(Width, Height);
    Trapezoid[3]:= Point(0, Height);
    rgn:= CreatePolygonRgn (Trapezoid, 4, WINDING);
    SetWindowRgn(Handle, rgn, True);
  end;
end;
//стрелка
procedure TForm1.sbtArrowClick(Sender: TObject);
var
  Arrow: array [0..6] of TPoint;
begin
  with Panel1 do begin
    Arrow[0]:= Point(0, Height div 3);
    Arrow[1]:= Point(Width div 2, 0);
    Arrow[2]:= Point(Width, Height div 3);
    Arrow[3]:= Point(Width - Width div 4, Height div 3);
    Arrow[4]:= Point(Width - Width div 4, Height);
    Arrow[5]:= Point(Width div 4, Height);
    Arrow[6]:= Point(Width div 4, Height div 3);
    rgn:= CreatePolygonRgn (Arrow, 7, WINDING);
    SetWindowRgn(Handle, rgn, True);
  end;
end;
//ромб
procedure TForm1.sbtDiamondClick(Sender: TObject);
var
  Diamond: array [0..3] of TPoint;
begin
with panel1 do begin
  Diamond[0]:= Point(Width div 2, 0);
  Diamond[1]:= Point(Width, Height div 2);
  Diamond[2]:= Point(Width div 2, Height);
  Diamond[3]:= Point(0, Height div 2);
  rgn:= CreatePolygonRgn (Diamond, 4, WINDING);
  SetWindowRgn(Handle, rgn, True);
end;
end;
//звезда
procedure TForm1.sbtStarClick(Sender: TObject);
var
  Star: array [0..9] of TPoint;
  Side, BottomOff, BaseTri, SideTri, HeightTri: Integer;
```

```
begin
with panel1 do begin
  Side:= Trunc(Height / Cos(DegToRad(18)));
  SideTri:= Trunc(Side / (2 + 2 * Sin(DegToRad(18))));
  BaseTri:= Side - 2 * SideTri;
  HeightTri:= Trunc(SideTri * Cos(DegToRad(18)));
  BottomOff:= Trunc(Tan(DegToRad(18)) * Height);
  Star[0]:= Point(Width div 2, 0);
  Star[1]:= Point(Width div 2 + BaseTri div 2, HeightTri);
  Star[2]:= Point(Width, Star[1].y);
  Star[3]:= Point(Star[1].x + Trunc(Sin(DegToRad(18)) * BaseTri),
            Star[1].y + Trunc(Cos(DegToRad(18)) * BaseTri));
  Star[4]:= Point(Width div 2 + BottomOff, Height);
  Star[5]:= Point(Width div 2, Height - Trunc(Sin(DegToRad(36)) *
SideTri));
  Star[6]:= Point(Width div 2 - BottomOff, Height);
  Star[7]:= Point(Width - Star[3].x, Star[3].y);
  Star[8]:= Point(0, Star[2].y);
  Star[9]:= Point(Width - Star[1].x, Star[1].y);
  rgn:= CreatePolygonRgn (Star, 10, ALTERNATE);
  SetWindowRgn(Handle, rgn, True);
end;
end;

//ПЕРЕТАСКИВАНИЕ ПАНЕЛИ
procedure TForm1.Panel1MouseDown(Sender: TObject; Button: TMouse-
Button; Shift: TShiftState; X, Y: Integer);
const
  SC_DragMove = $F012;
begin
  ReleaseCapture;
  (sender as Tpanel).Perform(WM_SysCommand, SC_DragMove, 0)
end;
```



Мы продолжим работу над панелями на Занятии 2.



Исходный код программы находится в папке **PanelForm**.

Для тех, кто не любит математику с её кропотливыми вычислениями координат точек и параметров кривых, мы припасли особый способ создания фигурных окон - из обычных картинок формата *Bitmap*. Он позволит из любой нарисованной вами в графическом редакторе формы создать пре-

восходное окно для вашего приложения. И тут уж всё будет зависеть только от вашей фантазии и художественного вкуса. Жалко было бы ограничиться только простой демонстрацией возможностей этого способа, поэтому мы напишем полноценную программу *Tetris*, где и разберём подробно, как он работает.

Формы-невидимки

Людям, умеющим находить прекрасное где угодно, можно предложить поэкспериментировать с формой-невидимкой (осуществлённая мечта Герберта Уэллса и американских «стэлсов»).

Сделать придётся не много, но и результат будет отвратительный: что может быть ужасней кнопок, висящих в бездне *Windows*!

А всего-то нужно задать форме стиль **WS_EX_TRANSPARENT** – и она перейдёт в царство духов. В типе формы опишите процедуру:

```
protected
   procedure CreateParams (var Params: TCreateParams); override;,
```

а в разделе **Implementation** – саму процедуру:

```
procedure TForm1.CreateParams (var Params: TCreateParams);
begin
   inherited CreateParams (Params);
   Params.ExStyle:= Params.ExStyle or WS_EX_TRANSPARENT
end;
```

Разместите на форме кнопку и запустите программу: форма исчезнет, а кнопка останется.

Ещё проще можно получить прозрачную форму, изменив кисть, которой закрашивается форма, так, что она сотрёт изображение:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  brush.Style:= bsClear;
  BorderStyle:= bsNone
end;
```

Впрочем, эта форма тоже не имеет больших художественных достоинств.

При желании можно сделать так, что форма вообще не появится на экране. В файле проекта после `Application.CreateForm(TForm1, Form1);` добавьте строчку `Application.ShowMainForm := false;` Наверное,

это может заинтересовать только шпионов и параноиков – зачем нормальным людям эдак скрывать себя?

Используя функцию **CombineRgn**, можно пойти ещё дальше и сделать по-настоящему полноценную форму, которая при этом останется совершенно прозрачной (если не нравится заголовок, просто уберите его, как это было описано раньше, но тогда уже форму не передвинуть). Кода написать придётся поболее ранешнего, но зато и полюбоваться будет на что!

```
unit shaping4;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
Dialogs, StdCtrls;

type
  TForm1 = class(TForm)
    Button1: TButton;
    procedure Button1Click(Sender: TObject);

  private
    { Private declarations }

  protected
    procedure CreateWindowRgn;
    procedure Resize; override;

  public
   { Public declarations }
   constructor Create(AOwner: TComponent); override;
  end;

var
  Form1: TForm1;

implementation

{$R *.DFM}

//создать прозрачную форму
constructor TForm1.Create(AOwner: TComponent);
begin
  inherited;
  //убираем полосы прокрутки, чтобы не появлялись
  //при уменьшении размеров формы:
  HorzScrollBar.Visible:= False;
  VertScrollBar.Visible:= False;
```

```
  //строим новую область:
  CreateWindowRgn;
end;

//отреагировать на изменение размеров окна:
procedure TForm1.Resize;
begin
  inherited;
  //строим новую область
  CreateWindowRgn;
end;

//создать прозрачное окно
procedure TForm1.CreateWindowRgn;
var
  FullRgn, Rgn: THandle;
  ClientX, ClientY, I: Integer;
begin
  //определяем относительные координаты клиентской части:
  ClientX:= (Width - ClientWidth) div 2;
  ClientY:= Height - ClientHeight - ClientX;

  //создаем область для всей формы:
  FullRgn:= CreateRectRgn(0, 0, Width, Height);

  //создаем область для клиентской части формы
  //и вычитаем его из FullRgn:
  Rgn:= CreateRectRgn(ClientX, ClientY, ClientX + ClientWidth,
ClientY + ClientHeight);
  CombineRgn(FullRgn, FullRgn, Rgn, rgn_Diff);

  //теперь добавляем к FullRgn области каждого управляющего эле-
мента на форме:
  for I:= 0 to ControlCount -1 do
    with Controls[I] do begin
      Rgn:= CreateRectRgn(ClientX + Left, ClientY + Top, ClientX +
Left + Width, ClientY + Top + Height);
      CombineRgn(FullRgn, FullRgn, Rgn, rgn_Or);
    end;

  //устанавливаем новую область окна:
  SetWindowRgn(Handle, FullRgn, True);
end;

//закрыть окно
procedure TForm1.Button1Click(Sender: TObject);
begin
  close
end;

end.
```

Зрелище поистине фантастическое (Рис. П1.35).



Рис. П1.35. В окне – только кнопка!

Здесь вы, конечно, должны вспомнить наши эксперименты по «опрозрачиванию» формы-заставки, но в том случае не только форма становилась прозрачной, но и всё, что на ней находится, а сейчас все компоненты остаются в своём обычном состоянии (Рис. П1.36).

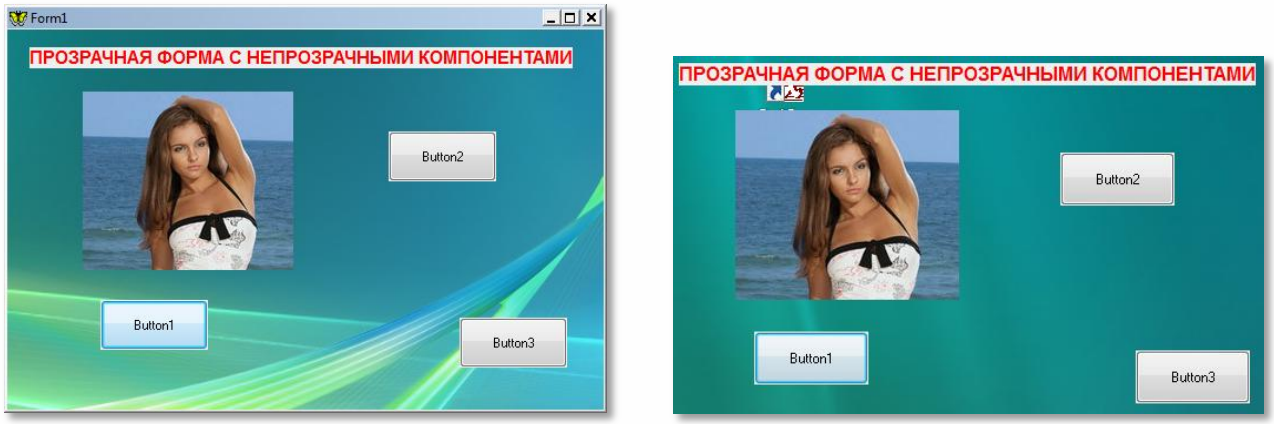


Рис. П1.36. Форма с заголовком Форма без заголовка (*BorderStyle = bsNone*)

Это значит, что таким способом вы сможете придавать окнам своих приложений совершенно уникальные качества!



Исходный код программы находится в папке **Shaping**.

Колоритная фигура, или Долой серость!

Не место красит человек,
а человек место.

Трюизм

Научившись придавать окну приложения практически любую форму, вставим в него цветные стёкла, а то и витражи.

Окрасить окно в любой цвет очень просто, достаточно для свойства **Color** формы выбрать из выпадающего списка один из стандартных цветов *Delphi*. К несчастью, эта палитра не слишком колоритная, так что и выбирать почти не из чего. Но ведь никто не запрещает вам использовать любой из многих миллионов доступных цветов! Вот только какой – за числами трудно разглядеть соответствующий им цвет. Попробуем облегчить себе проблему выбора, написав небольшую программу, позволяющую задать самые тонкие оттенки цвета. А, запомнив нужные значения для красной, зелёной и синей составляющей, вы сможете выкрасить форму (да и не только её) по своему разумению.

Откроем новый проект (**_RGB**), размеры формы увеличим так, чтобы она занимала почти весь экран, и разместим на ней, внизу компонент **ScrollBar** (*Палитра компонентов*, страница *Standard*). Назовём его **sbR**, вытянем так, чтобы слева и справа осталось место, и зададим диапазон изменения значений **Min** = 0, **Max** = 255. Дважды кликнем на полосе прокрутки и в заготовке процедуры напишем:

```
//Изменить цвет формы:
procedure TForm1.sbRChange(Sender: TObject);
begin
  txtR.Text:= inttostr(sbR.position);
  txtG.Text:= inttostr(sbG.position);
  txtB.Text:= inttostr(sbB.position);

  Color:= rgb(sbR.position,sbG.position,sbB.position);
end;
```

Слева поставим компонент **Label** (**Caption** = Red, **Color** = clBlack, **Transparent** = False), справа – компонент **Edit** (**Name** = txtR). Теперь нужно создать такие же элементы и для двух других составляющих цвета. Сделаем так: выделим полосу прокрутки **sbR**, щёлкнув на ней, скопируем в буфер обмена (*Ctrl + C*) и дважды нажмём *Ctrl +V* – появятся ещё две полоски, назовём их **sbG** и **sbB**. Аналогично «клонируем» метки (**Caption** = Green и **Caption** = 'Blue') и компоненты **Edit** (**Name** = txtG и txtB).

Всё – двигайте ползунки, следите за изменениями цвета формы – и записывайте нужные значения (Рис. П1.37).

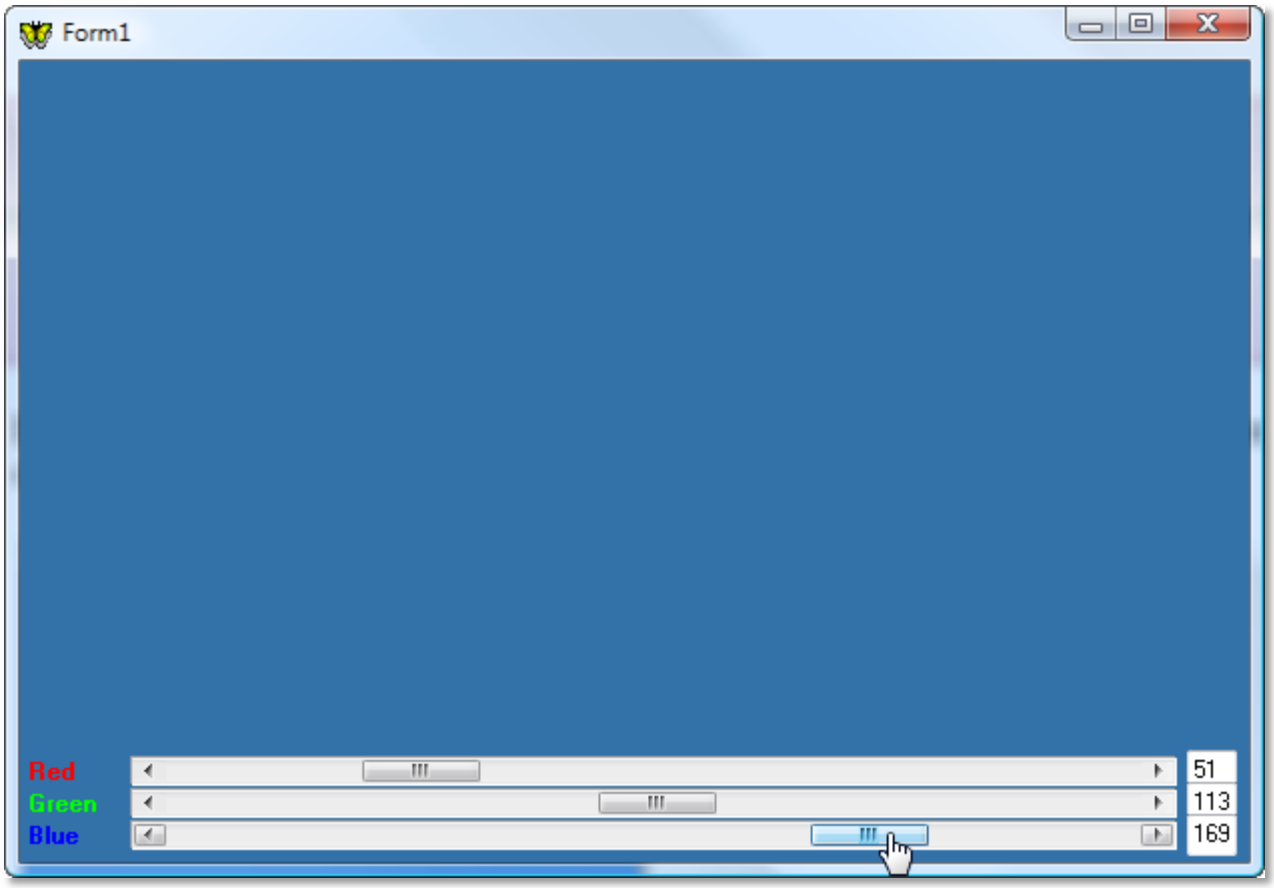


Рис. П1.37. Самоцветная форма

Если же вам однотонная форма кажется скучной, испытайте модную сейчас градиентную заливку, когда один цвет плавно переходит в другой. Этот приём часто применяется в заставках, которые появляются на экране при установке программ.

Интерфейс такой программы (**_grad**) немудрёный, как вечер (Рис. П1.37).

Начнём с **кнопок**. Поставьте одну кнопку типа *SpeedButton* со страницы *Additional* (**Name** = sbtYR, **Hint** = 'Переход от жёлтого к красному', **Caption** = 'Ж --> К'). Щёлкните на ней дважды и напишите код для процедуры **TForm1.sbtYRClick**:

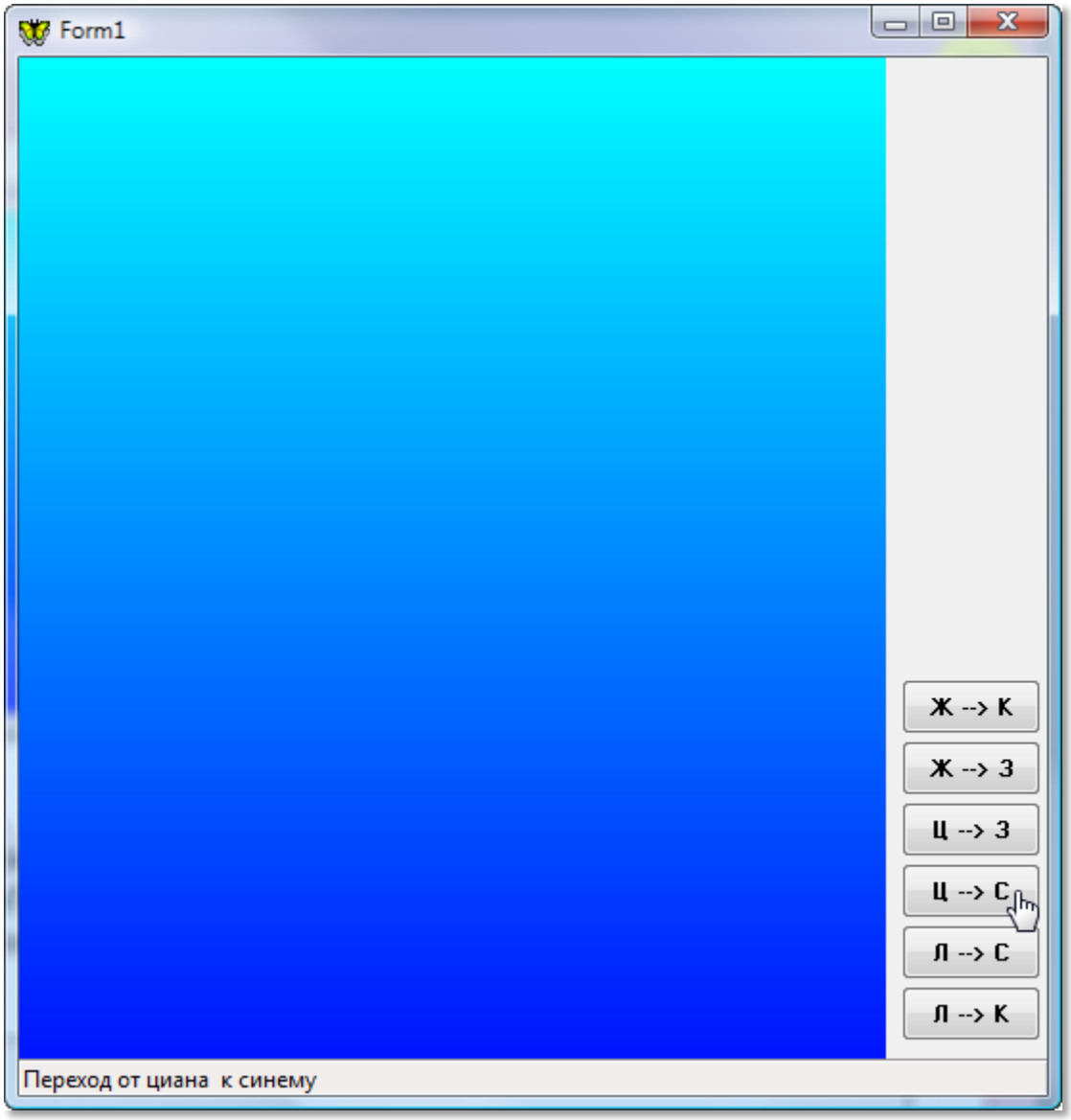


Рис. П1.38. Градиентный интерфейс

```
unit grad;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
Dialogs, Buttons, ComCtrls;

type
  TForm1 = class(TForm)
    sbtYR: TSpeedButton;
    SpeedButton1: TSpeedButton;
    SpeedButton2: TSpeedButton;
    SpeedButton3: TSpeedButton;
    SpeedButton4: TSpeedButton;
    SpeedButton5: TSpeedButton;
    StatusBar1: TStatusBar;
    procedure FormPaint(Sender: TObject);
```

```
    procedure sbtYRClick(Sender: TObject);
    procedure FormCreate(Sender: TObject);

  private
    { Private declarations }

  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation

{$R *.DFM}

 //НАЖАТЬ КНОПКУ
procedure TForm1.sbtYRClick(Sender: TObject);
var
  i : integer;
  NewBrush, OldBrush : hBrush;
  r, g, b : integer;           //- составляющие цвета
  DC: hDC;

begin
  DC:= Form1.Canvas.Handle;
  case (Sender as TSpeedButton).tag of
    0: //- переход от жёлтого к красному - Yellow To Red
      begin
        r := 255;  b := 0;
        for i := 0 to Height do begin
          //вычислить интенсивность зелёной составляющей цвета:
          g:= 255 * i div Height;
          //создать новую кисть:
          NewBrush := CreateSolidBrush(Rgb(r, g, b));
          //выбрать её и запомнить старую кисть:
          OldBrush := SelectObject(Dc, NewBrush);
          //начертить линию новым цветом:
          PatBlt(Dc, 0, Height-i, Width- 90, 1, PatCopy);
          //восстановить старую кисть:
          SelectObject(Dc, OldBrush);
          //удалить новую кисть:
          DeleteObject(NewBrush);
        end;
      end;

    1: //- переход от жёлтого к зелёному - Yellow To Green
      begin
        g:=255; b:=0;
        for i := 0 to Height do begin
          r := 255 * i div Height;
```

```
          NewBrush := CreateSolidBrush(Rgb(r, g, b));
          OldBrush := SelectObject(Dc, NewBrush);
          PatBlt(Dc, 0,Height - i, Width-90, 1, {DSTINVERT}
PatCopy {PATINVERT});
          SelectObject(Dc, OldBrush);
          DeleteObject(NewBrush);
        end;
      end;

    2: //- переход от циана  к зелёному - Cyan To Green
      begin
        r:=0; g:=255;
        for i:= 0 to Height do begin
          b := 255 * i div Height;
          NewBrush := CreateSolidBrush(Rgb(r, g, b));
          OldBrush := SelectObject(Dc, NewBrush);
          PatBlt(Dc, 0,Height - i, Width-90, 1, PatCopy);
          SelectObject(Dc, OldBrush);
          DeleteObject(NewBrush);
        end;
      end;

    3://- переход от циана  к синему - Cyan To Blue
      begin
        r := 0; b := 255;
        for i:= 0 to Height do begin
          g := 255 * i div Height;
          NewBrush := CreateSolidBrush(Rgb(r, g, b));
          OldBrush := SelectObject(Dc, NewBrush);
          PatBlt(Dc, 0,Height - i, Width-90, 1, PatCopy);
          SelectObject(Dc, OldBrush);
          DeleteObject(NewBrush);
        end;
      end;

    4://- переход от лилового к синему - Magenta To Blue
      begin
        g := 0; b := 255;
        for i:= 0 to Height do begin
          r := 255 * i div Height;
          NewBrush := CreateSolidBrush(Rgb(r, g, b));
          OldBrush := SelectObject(Dc, NewBrush);
          PatBlt(Dc, 0,Height - i, Width-90, 1, PatCopy);
          SelectObject(Dc, OldBrush);
          DeleteObject(NewBrush);
        end;
      end;

    5: //- переход от лилового к красному - Magenta To Red
      begin
        r := 255; g := 0;
        for i := 0 to Height do begin
```

```
          b := 255 * i div Height;
          NewBrush := CreateSolidBrush(Rgb(r, g, b));
          OldBrush := SelectObject(Dc, NewBrush);
          PatBlt(Dc, 0,Height - i, Width-90, 1, PatCopy);
          SelectObject(Dc, OldBrush);
          DeleteObject(NewBrush);
        end;
      end;
  end;
end.
```

Скопируйте кнопку в буфер обмена и выведите на форму 5 раз. Сделайте на них необходимые надписи, заполните свойство **Hint** и присвойте свойству *Tag* значения, соответственно, 1, 2, 3, 4, 5, чтобы в процедуре **sbtYRClick**, обрабатывающей нажатие кнопок, можно было узнать, какая из них была нажата.

Для вывода подсказок лучше использовать компонент **StatusBar** со страницы *Win32* (кликните на нём и в раскрывшемся *Редакторе панели* нажмите кнопку **Add New**, чтобы появилось поле для вывода подсказок, а также присвойте свойству **AutoHint** значение *True* – теперь подсказки будут появляться не у кнопок, а в панели, что бывает очень удобно во многих случаях).

Теперь никто не помешает вам сыграть на кнопках и полюбоваться своеобразной цветомузыкой форм в стиле известного французского композитора Жана-Мишеля Жарра, а также беспощадно заливать свои окна фонтанами цвета.

В этой программе главную роль исполнила шустрая функция **PatBlt** *Windows API*, которой вы наверняка найдёте работу в своих будущих проектах. Следует отметить, что **DC**, описатель контекста устройства, в данном случае – это просто описатель канвы формы. Хотя функция **PatBlt** закрашивает целые прямоугольники, но в этой программе её задача скромнее: нарисовать только одну линию текущего цвета (по сути это тоже прямоугольник, но высотой в 1 пиксель), чтобы получить незаметный переход оттенков по всей области. Код операции закрашивания здесь – **PATCOPY**, но вы можете попробовать и **PATINVERT**, заменив где-нибудь в процедуре **TForm1.sbtYRClick** **PATCOPY** на **PATINVERT**.

Функция PatBlt

Функция *Windows API* **PatBlt** рисует на контексте устройства прямоугольник заданных размеров, используя текущую кисть. Цвет кисти

и поверхности рисования комбинируются в соответствии с выбранной растровой операцией **rwRop**.

```
BOOL PatBlt(
  HDC hdc,        // описатель контекста устройства
  int nXLeft,     // x-координата верхнего левого уг-
ла прямоугольника
  int nYLeft,     // y-координата верхнего левого уг-
ла прямоугольника
  int nWidth,     // ширина прямоугольника
  int nHeight,    // высота прямоугольника
  DWORD dwRop     // код растровой операции
);
```

Коды растровой операции rwRop:

- **PATCOPY** – копировать цвет (или узор) кисти на контекст устройства.
- **PATINVERT** – комбинировать цвет (или узор) кисти с цветами контекста устройства с помощью логической операции *XOR*. Если применить функцию дважды, то целевой прямоугольник восстановится.
- **DSTINVERT** – инвертировать цвета целевого прямоугольника.
- **BLACKNESS**– закрасить целевой прямоугольник цветом с индексом **0** физической палитры. Если палитра не изменялась, то это чёрный цвет.
- **WHITENESS**– закрасить целевой прямоугольник цветом с индексом **1** физической палитры. Если палитра не изменялась, то это белый цвет.

Возвращаемое значение. Если функция завершилась успешно, то возвращается ненулевое значение.

Впрочем, для настоящего компьютерного художника – это грубые поделки, кустарщина.

Но отнюдь не трудно расписать форму каким угодно заковыристым рисунком с узорами и разводами в стиле Остапа Ибрагимовича Бендера. Для примера возьмём две небольшие картинки, которые послужат фоном для других компонентов на форме. Результаты вы видите на снимках с экрана (Рис. П1.39).

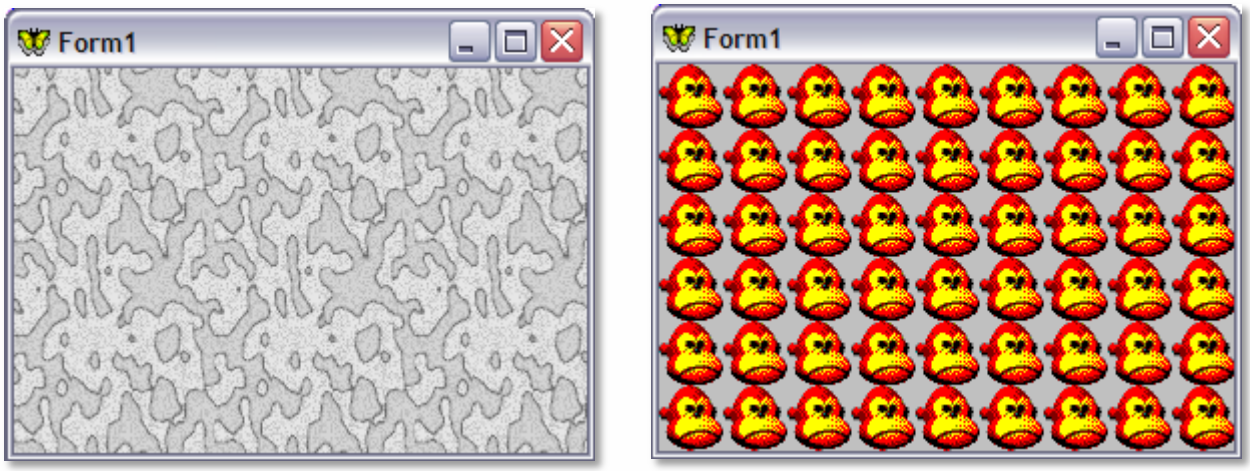


Рис. П1.39. Расписные формы

Вы легко можете заменить картинки любыми другими, чтобы получить надлежащий эффект. А вот как это делается:

```
unit bmpfrm;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
Dialogs;

type
  TForm1 = class(TForm)
    procedure FormCreate(Sender: TObject);
    procedure FormDestroy(Sender: TObject);
    procedure FormPaint(Sender: TObject);

  private
    { Private declarations }
    Bitmap: TBitmap;

  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation

{$R *.DFM}

//СОЗДАТЬ ФОРМУ
procedure TForm1.FormCreate(Sender: TObject);
begin
```

```
  Bitmap := TBitmap.Create;
  //Bitmap.LoadFromFile('back.bmp');
  Bitmap.LoadFromFile('обезьяна.bmp');
end;

//УНИЧТОЖИТЬ ФОРМУ
procedure TForm1.FormDestroy(Sender: TObject);
begin
 Bitmap.Free;
end;

//ОТРИСОВАТЬ ФОРМУ
procedure TForm1.FormPaint(Sender: TObject);
var
  x, y: Integer;

begin
  y := 0;
  while y < Height do
  begin
    x := 0;
    while x < Width do
    begin
      Canvas.Draw(x, y, Bitmap);
      x := x + Bitmap.Width;
    end;

    y := y + Bitmap.Height;
  end;
end;

end.
```

Сначала мы создаём в памяти растр (битовую карту) **Bitmap**, в который загружаем из файла нужную нам картинку. Если её размер меньше размера формы, то она будет повторяться, как рисунок на обоях. Процедура **FormPaint** как раз и выполняет копирование нашей картинки на форму, используя метод **Canvas.Draw**. Рисунок на форме благодаря **FormPaint** будет перерисовываться каждый раз при изменении её размеров, сворачивании-разворачивании и так далее. При выходе из программы остаётся только удалить объект **Bitmap** из памяти.

Вместо небольшой картинки можно взять примерно равную окну приложения и подогнать её другой процедурой - **Canvas.StretchDraw.** В результате получится полноценный фон, который вполне может украсить любую игру. Именно поэтому мы используем возможность похудожничать, чтобы написать небольшую программу *Тест космонавта*, где более подробно исследуем процедуру **StretchDraw**.

Надо заметить, что канва (*canvas*) имеется не только у формы, но и у многих других визуальных компонентов, - значит, есть смысл более подробно изучить её свойства и методы, что мы и сделаем в скором будущем, ведь рисовать в современных программах приходится изрядно. Сейчас же напишем несколько обещанных программ, что тоже очень интересно!



Исходный код программы находится в папке **ColorFrm**.

Проба пера, или *Mind Game* - гимнастика для ума

Берегите мозги – если они у вас есть!

Рекламное

Вот и пришло время написать первую настоящую игру. И хотя наша первостепенная задача – придать окну приложения эллиптическую форму, но мы используем эту возможность на все 100%, то есть заодно сделаем полноценную игрушку.

Как и у большинства других популярных игр, у **Mind Game** имеется несколько клонированных двойников, которые, впрочем, не в пример овечке Долли, чуточку отличаются друг от друга. Мы всерьёз возьмёмся только за один вариант – кому этого покажется мало, тот сам сможет продлить себе удовольствие, подправив программу по своему хотению и разумению.

Прежде чем заняться кодированием уточним «техзадание». В первую очередь нужно выполнить правила игры, а затем – создать удобное управление программой (интерфейс пользователя).

Правила игры

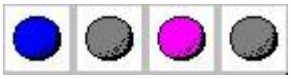
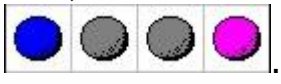
Играют двое. Один загадывает ряд из четырёх цветных фишек, а второй старается побыстрее отгадать его, предлагая свою последовательность фишек, на что первый игрок называет ему количество полностью угаданных фишек (они в загаданной и названной цепочке совпадают по цвету и по месту) и частично угаданных (у них совпадает только цвет, места же в цепочках они занимают разные). Исходя из полученных данных, отгадчик

составляет новую комбинацию фишек. На всё про всё даётся 10 ходов (попыток), которых вполне достаточно для смекалистого игрока.

Полностью угаданная фишка обычно называется **бык** или **пика**, а частично угаданная - **корова** или **фама**. Поэтому русские варианты этой славной игры называются, соответственно, **Быки и Коровы** и **Пика-фама**.

Составлять цепочки можно из фишек 6 разных цветов: синего, красного, зелёного, пурпурного (лилового), жёлтого, серого. Каждую фишку при построении цепочки можно использовать сколько угодно раз. Полностью угаданную фишку будем отмечать галочкой, частично угаданную – кружком.

Поскольку любая интеллектуальная игра – не кролик с его «легкоусваиваемым» мясом, то для прояснения правил и набора опыта сыграем одну пробную партию - между собой.

Пусть первый игрок загадал такой ряд фишек:  (в реальной игре противник этого, конечно, не знает). Всего существует 6*6*6*6 = 1296 различных цепочек, и на первом ходу можно назвать любую из них (по крайней мере, при игре с компьютером - люди обычно выбирают определённые сочетания фишек). Например, мы можем узнать, есть ли в цепочке красные фишки: . В ответ – тишина: таких фишек нет. А синего - ? Вот и первая ласточка, то есть галочка! Ещё не ура, но лёд тронулся. Делаем третий ход - . Опять галочка! Ну, всё понятно: синяя фишка занимает первое место, а зелёных и вовсе нет. Продолжаем наступать: . Две птички – значит, пурпурная фишка есть. Испытываем цепочку на желтизну: . Ответ получаем такой: галочка и кружок. Галочку нам дала синяя фишка, а кружок – пурпурная. Вывод: пурпурную фишку нужно переселить, а жёлтые фишки нас не должны интересовать из-за их полного отсутствия в цепочке. Пурпурную фишку мы можем поставить на третью или четвёртую позицию, а «пустоты» придётся заполнять фишками – «Никогда не догадаетесь какими!», - воскликнула бы сейчас Регина Дубовицкая, – серого цвета. Тут уж, конечно, как повезёт, поэтому выберем худшую половину вариантов, пойдём . Две галочки, два кружка – и судьба игры

предрешена, следующим ходом мы добиваем упрямую цепочку и поздравляем себя с красивой игрой и заслуженной победой.

Вот это баталия! Нравится? «По глазам вижу, что нравится!» - резюмировал бы Остап Ибрагимович, который знал толк в тонких комбинациях. Надеемся, что среди читателей все - сплошь последователи великого маэстро, поэтому вернёмся к нашей программе, в которой роль первого игрока всегда будет играть компьютер, а мы с вами и все наши друзья по несчастью обречены на вечное угадывание пёстрых фишек. Правда, мы-то успеем получить удовольствие от программирования, а всё остальное предоставим ближним своим

Соблюдаем ФОРМАльности

Итак, запускаем *Delphi* и конструируем **форму**. При желании ей можно придать сколь угодно замысловатый вид. Если вы склонны к изыскам и всяким там рококо, то поупражняйтесь в Шейпинг-классе, мы же остановим свой выбор на простой, но изящной овальной форме (согласитесь, в овальности есть что-то особенное!), тем более что сделать её очень **легко**, а выглядеть она будет так, словно вы трудились над ней целую жизнь.
Сейчас мы это всё быстро закодируем, но сначала обработайте «заготовку», то есть установите в *Инспекторе объектов* для **Form1** такие значения:

```
Left = 276
Top = 0
Position = poDesktopCenter
BorderStyle = bsNone
Ctl3D = False
Height = 576
ClientWidth = 409
Color = clBlue.
```

Если вы ненароком запустите программу, то увидите синий **прямоугольник**. Теперь, как советовал нам Роден, достигнем совершенства форм, убирая лишнее. В раздел **private** класса формы поместим объявление переменной **rgn: HRGN;** Чтобы форма приобрела законченный облик, напишем процедуру

```
//СОЗДАТЬ ФОРМУ
procedure TForm1.FormCreate(Sender: TObject);
begin
  //сформировать овальную форму:
  rgn:= CreateEllipticRgn(0,0,Width,Height);  //- функция Windows
API для создания эллиптических областей

  //функция Windows API для создания окна, форма которого опреде-
```

```
ляется областью rgn:
  SetWindowRgn(Handle, rgn, True);
End;
```

Запускаем программу – совсем другое дело: одним движением руки мы превратили прямоугольник в овал (честно говоря, это всё-таки эллипс). Что характерно: форма в *Конструкторе форм* всегда прямоугольная, а окно приложения может иметь любую форму. Это интересно, но создаёт проблемы: нет кнопок на полоске заголовка, нам нечем закрыть окно (приложение) и даже нельзя его передвинуть, потому что мы не можем «подцепить» окно за заголовок. Если завершить программу при запуске из ИСР легко - нужно просто выполнить команду *Главного меню* **Run > Program Reset** (*Ctrl + F2*), то при работе готовой программы придётся жать клавиши *Ctrl-Alt-Del*, что уж совсем некрасиво. Вывод один: нужно поставить кнопку для выхода из программы и изловчиться двигать форму за разные места.
Так как форма и окно отличаются по конфигурации, то трудно найти надлежащее место для кнопки выхода, да и всех остальных управляющих элементов. Поступим так: положим на форму компонент **Shape1** (он хранится в *Палитре компонентов*, страница *Additional*) и подгоним его по окну программы:

```
Left = 4
Top = 4
Width = 393
Height = 541
Brush.Style = bsClear
Pen.Color = clYellow
Pen.Width = 2
Shape = stEllipse.
```

Теперь окно программы приобрело законченный вид, и нам стало удобнее устанавливать на форму компоненты. В качестве «выходной» кнопки можно взять **Button** со страницы *Standard* или **BitBtn** со страницы *Additional*. А вот кнопка **SpeedButton** с той же страницы работать не будет. Кнопку назовём **bbtClose** и наградим её такими характеристиками:

```
LeftLeft = 304
Top = 116
Width = 32
Height = 32
```

Точный «адрес» кнопки пока, конечно, не известен, но потом мы можем задвинуть её куда угодно. Что касается размеров, то 32 х 32 пикселя как раз достаточно, чтобы на кнопке удобно разместились стандартные картинки. С курсором тоже мудрить не следует: раз кнопку придётся нажимать, то лучше прозрачно на это намекнуть: **Cursor** = crHandPoint, а для тех, кто продолжает сомневаться в назначении кнопки, соорудим подсказ-

ку: **Hint** = 'Закрыть программу', **ParentShowHint** = False, **ShowHint** = True. Картинку (**Glyph**) для кнопки выхода из программы лучше всего взять стандартную , чтобы во всех ваших программах одинаковые по действию кнопки имели бы одно и то же изображение.

Дважды щёлкните на кнопке и в окне редактирования кода напишите «close» в заготовке процедуры **bbtCloseClick**:

```
//ЗАКРЫТЬ ПРОГРАММУ
procedure TForm1.bbtCloseClick(Sender: TObject);
begin
  close
end;
```

Займёмся второй проблемой. После строки `rgn: HRGN;` добавим:

```
procedure WMNCHitTest(var Message: TWMNCHitTest); message
WM_NCHITTEST;
```

А после `TForm1.FormCreate` поместим ещё одну процедуру, которая «обманывает» *Windows*: она подменяет нажатие кнопки мыши на заголовке нажатием кнопки в области окна.

```
//ПЕРЕМЕЩЕНИЕ ОКНА
procedure TForm1.WMNCHitTest(var Message: TWMNCHitTest);
begin
  if PtInRegion(rgn,Message.XPos, Message.YPos) then
    Message.Result:= HTCAPTION
  else
    Message.Result:= HTNOWHERE
end;
```

Функция PtInRegion

Функция *Windows API* определяет, находится ли заданная точка в заданной области.

```
BOOL PtInRegion(
  HRGN hrgn,          // описатель области
  int X,              // x-координата точки
  int Y               // y- координата точки
);
```

Возвращаемое значение. Если точки находится в заданной обла-

сти, то возвращается ненулевое значение.

Этот трюк можно заменить другим (процедуру *WMNCHitTest* и её объявление уберите):

```
//ПЕРЕТАСКИВАНИЕ ФОРМЫ
procedure TForm1.Shape1MouseDown(Sender: TObject; Button: TMouse-
Button;
Shift: TShiftState; X, Y: Integer);
const
SC_DragMove = $F012;
begin
ReleaseCapture;
Perform(WM_SysCommand, SC_DragMove, 0)
end; //Shape1MouseDown
```

Осталось при завершении программы уничтожить область окна, созданную в **FormCreate**:

```
//УНИЧТОЖИТЬ ФОРМУ
procedure TForm1.FormDestroy(Sender: TObject);
begin
  DeleteObject(rgn)
end;
```

Запустите программу – теперь окно можно передвигать, сворачивать (щёлкните на значке программы в панели задач *Windows*) и снова раскрывать (щёлкните на значке ещё раз).

На этом заканчивается подготовительный этап – время браться за нашу игрушку обеими руками.

Как это будет по-дельфийски?

Моя твоя не понимает!

Кризис интерфейса

Парле ву франсе?

Обращение инопланетянина к корове

Для анализа «стрельбы» по фишкам неплохо было бы иметь перед собой подробный «протокол» всех «выстрелов» и их результативность. Приспособим для этого две сетки – **dgPole** и **dgGuess** типа *TDrawGrid*. Чтобы вы могли лучше ориентироваться в пространстве, приведём для образца го-

товую форму – так она должна выглядеть в окне *Конструктора формы* после того, как вы разместите на ней все управляющие элементы (Рис. П1.40).

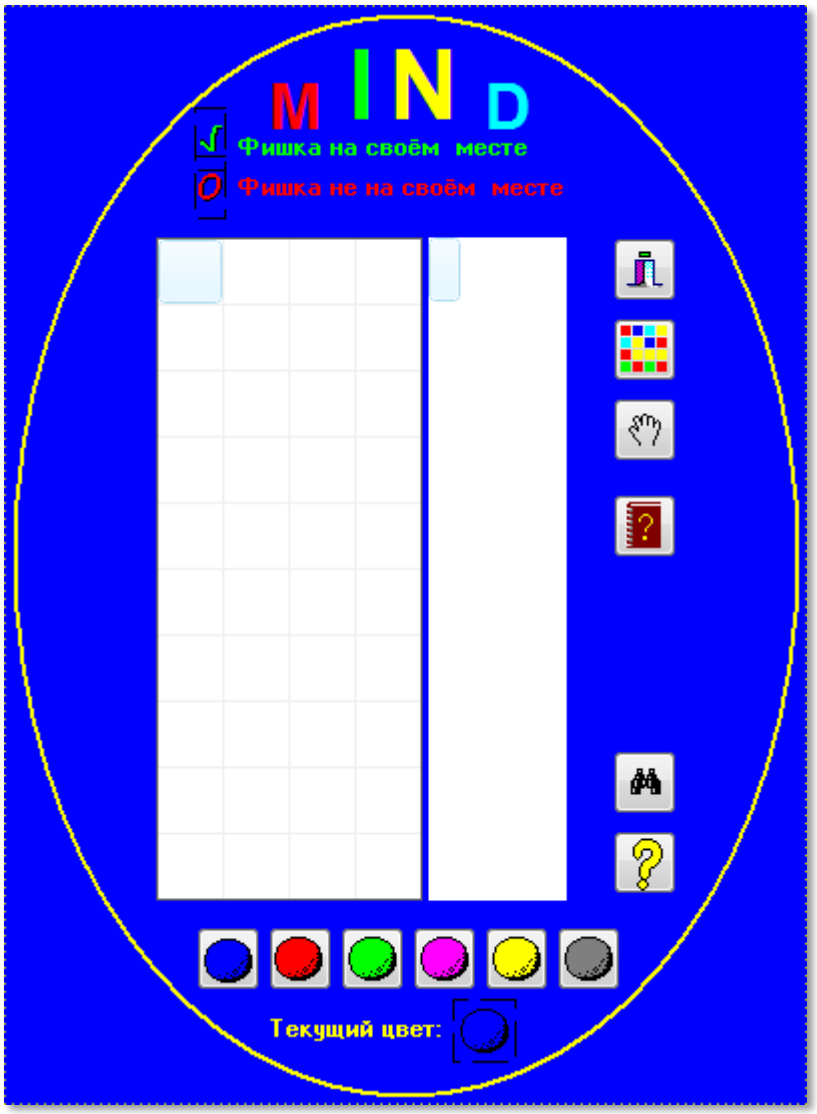


Рис. П1.40. Умный интерфейс

Сначала просто установите сетки на форме, а затем обязательно запишите проект в новую папку **Mind**, выбрав для модуля формы и проекта подходящие имена.



Числа, которые будут встречаться в программе, следует назвать «своими именами», иначе будет трудно понять, что означает, например, число 10. А если вместо него написать **MAX_HOD**, то легко догадаться, что речь идёт о допускаемом правилами игры числе ходов. Поступайте так всегда и везде – это избавит вас от многих досадных и трудно отыскиваемых ошибок в тексте программы. У этого правила имеется и ещё один плюс: если вы захотите добавить игроку пару ходов, то просто измените соответствующую константу

– и всё (иногда придётся поработать и над компонентами на форме, но от этого не уйдёшь).

Итак, в модуле формы заведём раздел констант и в первую очередь объявим те, которые нам известны из правил игры:

```
unit MindUnit;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
Dialogs, Buttons, ExtCtrls, StdCtrls, Grids;

const
  NAME_PROG = 'MIND GAME'; //- название программы
  MAX_HOD = 10;            //- макс. число ходов
  MAX_FISH = 6;            //- количество фишек разных цветов
  LEN_ROW = 4;             //- число загаданных фишек
  POLE_WIDTH = LEN_ROW;    //- ширина поля в клетках
  POLE_HEIGHT = MAX_HOD;   //- высота поля в клетках
```

Придадим сетке игрового поля **dgPole** нужные свойства:

```
  Left = 76
  Top = 116
  Width = 133
  Height = 331 (положение и размеры)
  Cursor = crHandPoint (курсор)
  Color = clNavy            (цвет клеток)
  ColCount = 4              (число колонок = POLE_WIDTH)
  RowCount = 10             (число строк = POLE_ HEIGHT)
  Ctl3D = False             (плоская сетка)
  DefaultColWidth = 32
  DefaultRowHeight = 32     (размеры клеток)
  DefaultDrawing = False    (содержимое клеток будем выводить сами)
  FixedCols = 0   FixedRows = 0  (фиксированные клетки нам не нуж-
ны)
  Options = [goVertLine, goHorzLine]  (оставляем только линии, об-
разующие сетку)
  ScrollBars = ssNone       (скроллеры не нужны)

  OnDrawCell = dgPoleDrawCell  - процедуры рассматриваются дальше
  OnMouseDown = dgPoleMouseDown
```

Аналогично поступаем и со второй сеткой – **dgGuess**, в которой будем отмечать полностью и частично угаданные фишки:

```
  Left = 212
  Top = 116
```

```
  Width = 69
  Height = 331
  Cursor = crHandPoint
  BorderStyle = bsNone
  Color = clBlue
  ColCount = 4
  RowCount = 10
  Ctl3D = False
  DefaultColWidth = 16
  DefaultRowHeight = 32
  DefaultDrawing = False
  FixedCols = 0
  FixedRows = 0
  Options = []          (линии не нужны)
  ScrollBars = ssNone

  OnClick = dgGuessClick
  OnDrawCell = dgGuessDrawCell
```

Для хранения возможных комбинаций фишек (подсказок) введём новый тип данных:

```
Type THint = array[1..MAX_FISH*MAX_FISH*MAX_FISH*MAX_FISH,
0..POLE_HEIGHT-1] of byte;
```

Первый индекс - комбинация из 4-х фишек (общее число таких комбинаций указано выше). Второй указывает на текущую строку в сетке поля (ход).

После раздела описания типов начнём новый раздел – для описания глобальных переменных программы, который начнём с объявления двух массивов указанного типа:

```
var
   mHint,mHint2: THint;                    //- массивы подсказок
```

Сначала оба эти массива заполняем всеми допустимыми комбинациями из четырёх фишек, так как может быть задумана любая комбинация фишек. Подсказки мы будем использовать для облегчения выбора очередного хода – достаточно указать на одну комбинацию из списка, чтобы сделать его. Ясно, что после выполнения хода нужно заново переписывать массив подсказок так, чтобы они удовлетворяли сделанным ходам. Число подсказок в массиве будем хранить в *переменной*:

```
nHint: integer;                                //- число подсказок
```

В начале игры заносим в массивы все комбинации фишек:

```
//СФОРМИРОВАТЬ МАССИВ ПОДСКАЗОК
procedure CreateHint;
var i0,i1,i2,i3: integer;
begin
  nHint:=0;

  //сначала заносим в массив ВСЕ возможные комбинации фишек:
  for i0:= 0 to MAX_FISH-1 do
    for i1:= 0 to MAX_FISH-1 do
      for i2:= 0 to MAX_FISH-1 do
        for i3:= 0 to MAX_FISH-1 do begin
          inc(nHint);
          mHint[nHint,0]:=i0; mHint[nHint,1]:=i1;
          mHint[nHint,2]:=i2; mHint[nHint,3]:=i3;
        end;
  mHint2:= mHint;
end; //CreateHint
```

Глобальную переменную **hod** мы отведём под хранение номера очередного хода (попытки):

```
hod: integer;                                    //- номер хода,
```

а в переменную **CurrentColor** поместим цвет текущей фишки:

```
CurrentColor: integer;                          //- цвет текущей фишки
```

Текущая фишка (цвет) находится в самом низу формы. Чтобы её заменить, следует щёлкнуть на одной из шести фишек над ней. Текущая фишка используется для набора очередного хода: если щёлкнуть на какой-либо клетке поля, то в ней появится точно такая же фишка.

В разделе описания переменных объявим два числовых массива:

```
  //массивы игрового поля и поля результатов:
  masPole: array[0..POLE_WIDTH-1,0..POLE_HEIGHT-1] of integer;
  masGuess: array[0..POLE_WIDTH-1,0..POLE_HEIGHT-1] of integer;
```

В них будем помещать числовые значения цвета фишек, клеток сетки и результаты – для каждого хода. Для удобства введём новые **константы**:

```
  NUM_BLUE = 0;              //синяя фишка
  NUM_RED = 1;               //красная фишка
  NUM_GREEN = 2;             //зелёная фишка
  NUM_PURPLE = 3;            //пурпурная фишка
  NUM_YELLOW = 4;            //жёлтая фишка
  NUM_GREY = 5;              //серая фишка
  NUM_CURRENTROW = -1;       //цвет клеток текущего ряда
  NUM_GRID = -2;             //цвет клеток сетки
```

```
  NUM_CORRECT = 1;          //верный цвет
  NUM_INCORRECT = 2;        //неверный цвет
```

Также нам пригодятся константы, хранящие *цвет* клеток полей:

```
  COLOR_CURRENTROW: TColor= clWhite; //цвет текущей строки - белый
  COLOR_POLE: TColor= clNavy;        //цвет клеток игрового поля
  COLOR_GUESS: TColor= clBlue;       //цвет клеток поля результа-
тов
```

Загаданная комбинация фишек будет храниться в *массиве:*

```
answer: array[0..POLE_WIDTH-1] of integer; //- загаданные фишки
```

И последний массив – **bmFish**:

```
bmFish: array[0..MAX_FISH-1] of TBitmap;   //- картинки с фишками
```

В него мы загрузим растровые изображения всех фишек. Удобнее всего сделать это при создании формы приложения. Допишите в процедуре **FormCreate** такие строки:

```
  //загрузить изображения фишек:
  for i:=0 to MAX_FISH-1 do begin
    bmFish[i]:= TBitmap.Create;
    bmFish[i].LoadFromResourceName(HInstance, FISH_NAME[i]);
  end;
```

Все картинки будем хранить в файле ресурсов, который нужно подключить к проекту:

```
{$R *.DFM}
//картинки с фишками
{$R extrares.res}
```

Файл ресурсов можно создать в *Редакторе ресурсов*. Самое «сложное» здесь – нарисовать фишки.

Для названий фишек в файле ресурсов заведём *массив констант*, чтобы не переписывать процедуру **LoadFromResourceName** шесть раз:

```
  //названия фишек (цвета):
  FISH_NAME: array [0..MAX_FISH-1] of
string=  ('BLUE','RED','GREEN','PURPLE','YELLOW','GREY');
```

Теперь нужно позаботиться о выводе сеток на экран после каждого хода. В *Инспекторе объектов*, на вкладке *Events* щёлкните дважды в окне ввода названия обработчика процедуры **OnDrawCell**. В окне *Редактора кода* по-

явится заготовка соответствующей процедуры, в которой напишите следующий код:

```
//ПЕРЕРИСОВАТЬ ЯЧЕЙКУ ИГРОВОГО ПОЛЯ
procedure TForm1.dgPoleDrawCell(Sender: TObject; ACol, ARow: Inte-
ger; Rect: TRect; State: TGridDrawState);
var n: integer;

begin
  n:= masPole[ACol, ARow];
  case n of
    NUM_CURRENTROW: //пустая клетка текущего ряда
      begin
        dgPole.Canvas.Brush.Color:= COLOR_CURRENTROW;
        dgPole.Canvas.FillRect(Rect);
      end;
    NUM_BLUE..NUM_GREY:  //цветная фишка
      dgPole.canvas.BrushCopy(Rect,bmFish[n],Bounds(0, 0, 32,
32),$777777);
    else begin           //пустая клетка поля
      dgPole.Canvas.Brush.Color:= COLOR_POLE;
      dgPole.Canvas.FillRect(Rect);
    end;
  end;
end; //dgPoleDrawCell
```

Работает он так. По числу в массиве **masPole** определяем, что находится в клетке с координатами **ACol**, **ARow**. Если это клетка текущей строки поля, то закрашиваем квадратик белым цветом (*COLOR_CURRENTROW*). Пустую клетку окрашиваем в синий цвет (*COLOR_POLE*). А если в клетке находится фишка, то копируем в неё соответствующую картинку из массива **bmFish**.

Подобным же образом прорисовываем клетки поля результатов, только картинки с галочкой и кружком поместим на форме - в компонентах **imgCorrect** и **imgInCorrect**, которые заодно послужат и подсказками, если вы забудете назначение значков (рядом установите метки **Label1** с **Caption** = ‘Фишка на своём месте’ и **Label2** с **Caption** = ‘Фишка не на своём месте’ – смотрите картинку с формой).

```
//ПЕРЕРИСОВАТЬ ЯЧЕЙКУ ПОЛЯ РЕЗУЛЬТАТОВ
procedure TForm1.dgGuessDrawCell(Sender: TObject; ACol, ARow: In-
teger; Rect: TRect; State: TGridDrawState);
begin
  case masGuess[ACol, ARow] of
    NUM_CORRECT:   //фишка на своём месте
      dgGuess.canvas.BrushCopy(Rect,imgCorrect.picture.Bitmap,
imgCorrect.ClientRect,clWhite);
    NUM_INCORRECT: //фишка не на своём месте
      dgGuess.canvas.BrushCopy(Rect,imgInCorrect.picture.Bitmap,
```

```
imgInCorrect.ClientRect,clWhite);
    else
    begin          //пустая клетка
      dgGuess.Canvas.Brush.Color:= COLOR_GUESS;
      dgGuess.Canvas.FillRect(Rect)
    end
  end;
end; //dgGuessDrawCell
```

Чтобы вывести на экран поля целиком, последовательно перебираем все клетки и для каждой вызываем процедуру рисования **dgPoleDrawCell** или **dgGuessDrawCell**:

```
//ВЫВЕСТИ ПОЛЯ
procedure TForm1.OutPole;
var i,j: integer;
  area: TRect;
begin
  //игровое поле:
  for j:= 0 to POLE_HEIGHT-1 do
    for i:= 0 to POLE_WIDTH-1 do begin
      area:= dgPole.CellRect(i, j);
      dgPoleDrawCell(Self, i,j, area, []);
    end;
  //поле результатов:
  for j:= 0 to POLE_HEIGHT-1 do
    for i:= 0 to POLE_WIDTH-1 do begin
      area:= dgGuess.CellRect(i, j);
      dgGuessDrawCell(Self, i,j, area, []);
    end;
end; //OutPole
```

Перед началом каждой игры нужно убрать из обоих полей ненужную информацию, то есть занести в массивы коды соответствующих цветов:

```
//ОЧИСТИТЬ ПОЛЯ
procedure TForm1.ClearPole;
var i,j: integer;
begin
  for j:= 0 to POLE_HEIGHT-1 do
    for i:= 0 to POLE_WIDTH-1 do begin
      //занести в массив игрового поля: для клеток первого ряда -
      //код белого цвета (NUM_CURRENTROW),
      //для остальных - тёмно-синего (NUM_GRID):
      if j>0 then masPole[i,j]:= NUM_GRID
      else masPole[i,j]:= NUM_CURRENTROW;

        //в массив результатов хода заносим
        //код голубого цвета (NUM_GRID):
        masGuess[i,j]:= NUM_GRID
```

```
      end;
end;  //ClearPole;
```

Не удивляйтесь, что один и тот же код **NUM_GRID** одновременно означает и голубой, и тёмно-синий цвет. Дело в том, что этот код по-разному интерпретируется в процедурах рисования ячеек игрового поля **dgPoleDrawCell** и **dgGuessDrawCell**. В первой из них для этого кода устанавливается цвет **COLOR_POLE**, во второй - **COLOR_GUESS**.

И, наконец, мы можем написать окончательный вариант процедуры-обработчика, вызываемой при создании формы:

```
//СОЗДАТЬ ФОРМУ
procedure TForm1.FormCreate(Sender: TObject);
var i: integer;
begin
  //сформировать овальную форму:
  rgn:= CreateEllipticRgn(0,0,Width,Height); SetWindowRgn(Handle,
rgn, True);
  //загрузить изображения фишек:
  for i:=0 to MAX_FISH-1 do begin
    bmFish[i]:= TBitmap.Create;
    bmFish[i].LoadFromResourceName(HInstance, FISH_NAME[i]);
  end;
  //подготовиться к новой игре:
  Prepare;
end;  //FormCreate
```

Таким образом, при запуске программы мы формируем эллиптическую область, которая определяет форму окна приложения. Затем загружаем в массив картинки фишек и готовим всё необходимое для начала игры в процедуре **Prepare**:

```
//ПОДГОТОВИТЬСЯ К НОВОЙ ИГРЕ
procedure TForm1.Prepare;
var i: integer;
begin
  //первый ход (отсчёт начинается с нуля):
  hod:= 0;
  //создать массив подсказок:
  CreateHint;
  //текущая фишка - синяя:
  imgColorHand.Picture.Bitmap:= bbtColor0.Glyph;
  //imgColorHand.Picture.Bitmap.Assign(bbtColor0.Glyph);
  CurrentColor:= NUM_BLUE;
  //очистить игровые поля:
  ClearPole;
  //вывести игровые поля:
  OutPole;
```

```
  //загадать фишки:
  randomize;
  for i:= 0 to POLE_WIDTH-1 do
    answer[i]:= random (MAX_FISH);
end; //Prepare
```

Здесь мы инициализируем переменную **hod**, в массив подсказок заносим все возможные комбинации фишек, текущей объявляем синюю фишку, очищаем поля, в результате чего верхний ряд игрового поля станет белым, все остальные – тёмно-синими, а поле результатов просто сольётся с фоном формы и станет невидимым. И последнее действие – загадываем 4 фишки, которые бережно сохраним в массиве **answer**.

Хотя мы и знаем, что текущая фишка синяя, но в ходе игры её цвет будет меняться – не запоминать же его каждый раз! Поэтому установите на форме компонент **imgColorHand** типа *TImage* (**Left** = 224, **Top** = 496, **Width** = 32, **Height** = 32, **Transparent** = True) и загрузите в него картинку с синей фишкой. Не лишним будет объяснить её назначение – в метке **Label7** напишем **Caption** = 'Текущий цвет:'

Установите в верхней части формы 4 метки - **Label2**..**Label5** – и напишите в них буквы, образующие название игры. Постарайтесь сделать это красиво. Если хотите, можете вместо букв сделать в графическом редакторе подходящую картинку и загрузить её в ещё один компонент *Image*.

Вы вполне можете запустить программу и убедиться воочию, что мы хорошо подготовились к игре. Попробуйте сделать первый ход. Так как программа загадывает ряд фишек совершенно случайно, то вы вправе выбрать любые четыре фишки. Если вам очень повезёт, то вы сразу попадёте на ответ. Впрочем, особенно на это не уповайте, из чего следует мораль: желательно чтобы и первый ход был полезным. Например, вы можете выбрать фишки одного цвета и узнать, имеются ли они среди загаданных. Другой неплохой ход – выставить фишки разного цвета, чтобы определить число «разноцветных» фишек в загаданной комбинации и т.д. Вариантов «хороших» ходов немало – но как сделать ход? Выставить в клетку игрового ряда можно только фишку текущего цвета. Для этого достаточно просто кликнуть мышкой на нужной клетке (организовать буксировку фишки посложнее, да и менее удобно). Как вы, должно быть, помните, мы заготовили на этот случай процедуру **dgPoleMouseDown** (щёлкните на окне ввода имени процедуры для события **OnMouseDown**, чтобы открыть заготовку):

```
//ПОСТАВИТЬ ФИШКИ НА КЛЕТКУ ИГРОВОГО ПОЛЯ
procedure TForm1.dgPoleMouseDown(Sender: TObject; Button: TMouse-
Button; Shift: TShiftState; X, Y: Integer);
```

```
var
  ACol,ARow: integer;
begin
  //ACol,ARow - координаты ячейки игрового поля:
  dgPole.MouseToCell(x,y,ACol,ARow);
  if ARow=hod then //- текущая строка
    //в клетке - фишка текущего цвета:
    masPole[ACol, ARow]:=CurrentColor
  else if ARow<hod then //- строка сверху
    //цвет верхней фишки:
    masPole[ACol, hod]:=masPole[ACol, ARow];
  //обновить поле:
  OutPole;
end;  //dgPoleMouseDown
```

Если вы достаточно наблюдательны, то наверняка заметили, что мы используем не событие **OnClick** (щелчок мышкой), а **OnMouseDown** (нажатие кнопки мыши). Дело в том, что нам необходимо знать, в какую клетку игрок желает поставить фишку, а координаты мыши передаются в обработчик второго события, а не первого. И нам остаётся только по координатам курсора в пикселях определить клетку игрового поля. Это легко сделать с помощью метода сетки **MouseToCell**. Если «щелчок» был в клетке текущего ряда (**ARow**=hod), то заносим в массив **masPole** цвет текущей фишки (**CurrentColor**). Когда места некоторых фишек в цепочке уже известны, было бы очень удобно переносить их в соответствующие позиции текущего ряда из какого-либо верхнего. Нет ничего проще – щёлкаем не в клетке текущего ряда, а в нужной клетке над ней (**ARow<hod**) и копируем в массив её цвет (`masPole[ACol, hod]:=masPole[ACol, ARow];`). Чтобы фишка появилась не только в массиве, но и на экране, выполняем процедуру **OutPole**. Щёлкайте сколько угодно – текущая фишка тут же появится на экране. Но пока у нас только одна текущая фишка – синяя, значит, нужно научить программу заменять одну текущую фишку на другую.

Мы опять остановимся на самом простом решении этой проблемы – разместим на форме 6 кнопок типа *TBitBtn*: **bbtColor0**..**bbtColor5** со свойствами:

Left = 96, 132, 168, 204, 240, 276 **Top** = 460
Width = 32 **Height** = 32
Cursor = crHandPoint
OnClick = bbtColor0Click
Glyph – загрузите картинки с фишками в той последовательности, как показано на снимке с экрана.

Нажатие **всех** кнопок будем обрабатывать в **одной** процедуре **bbtColor0Click**, поэтому щёлкните на самой левой кнопке, чтобы получить заготовку процедуры, а для всех остальных кнопок в *Инспекторе объектов*

свяжите событие **OnClick** с этой процедурой. Чтобы определить, какая именно кнопка вызвала обработчик, установите свойство **Tag** всех кнопок соответственно их порядковому номеру: 0..5.

Нажав одну из кнопок с фишками, мы попадаем в процедуру **bbtColor0Click**, где по свойству **Tag** определяем номер нажатой кнопки, а затем с помощью метода формы **findComponent** переходим на нужный компонент и копируем картинку. И, наконец, устанавливаем новый текущий цвет:

```
//ВЫБРАТЬ ТЕКУЩИЙ ЦВЕТ
procedure TForm1.bbtColor0Click(Sender: TObject);
var i: integer;
begin
  //номер кнопки:
  i:= (Sender as TBitBtn).tag;
  //скопировать фишку из нажатой кнопки в картинку
  //с фишкой текущего цвета:
  imgColorHand.Picture.Bitmap :=  (findComponent('bbtColor'+
inttostr(i)) as TBitBtn).Glyph;
  //установить текущий цвет:
  CurrentColor:= i;
end; //bbtColor0Click
```

Эти кнопки позволяют нам выбирать любой текущий цвет и, следовательно, устанавливать в текущем ряду игрового поля нужную комбинацию фишек (кстати, если вы поставили в клетку фишку одного цвета, а затем решили её заменить, то, выбрав новый текущий цвет, просто щёлкните на этой клетке, и в ней появится «правильная» фишка).

В ответ на наш ход компьютер должен вывести его результаты, то есть число угаданных фишек. Поскольку программа не может самостоятельно определить, закончили вы набирать комбинацию фишек или нет (вы ведь можете выставить все 4 фишки, а потом изменить своё решение), то добавим на форму кнопку, которой вы будете сигнализировать об окончании своего выбора. Пусть все кнопки будут одного типа, дабы не вносить сумбур в интерфейс. Поэтому разместите на форме ещё одну кнопку типа *TBitBtn*. Уместно назвать её **bbtTest** и подобрать ей такое место на форме, чтобы она всегда была под рукой. На нашем скриншоте это кнопка с вопросительным знаком - , - но вы можете подобрать и другую картинку. Свойства кнопки такие:

Left = 304 **Top** = 412 **Width** = 32 **Height** = 32
Cursor = crHandPoint
Hint = 'Проверить ход' **ShowHint** = True
Glyph – загрузите картинку
OnClick = bbtTestClick

Несмотря на кажущуюся простоту проверки хода, в этом процессе есть и своя «изюминка», связанная с фишками, которые присутствуют и в загадке, и в отгадке, но занимают в них разные позиции. Важно отмечать те фишки, которые уже использованы при сравнении, иначе мы несколько раз посчитаем одну и ту же фишку и как следствие получим неверный результат хода. Поясним это на примере.

Пусть загадана комбинация фишек (Рис. П1.41).



Рис. П1.41. Загаданные фишки

Сделав два хода, мы должны получить следующие результаты (Рис. П1.42).

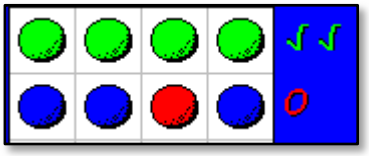


Рис. П1.42. Два хода сделаны

В первом случае, найдя две зелёные фишки, стоящие на своих местах, мы при дальнейших сравнениях пропускаем их в загаданной комбинации, так как в противном случае получим в ответ, что загадано 4 зелёные фишки, из которых две стоят «правильно» и ещё две - «неправильно». Во втором случае **одна** синяя фишка в нашей комбинации занимает неверное место, а остальных двух нет вообще.

Итак, любую фишку в загаданной комбинации мы считаем *единожды*. Отмечать использованные фишки будем в массиве **checked**:

```
//ПРОВЕРИТЬ ХОД
procedure TForm1.bbtTestClick(Sender: TObject);
var
  i,j: integer;
  right: array[0..POLE_WIDTH-1] of boolean;       //=TRUE, если
фишка угадана
  checked: array[0..POLE_WIDTH-1] of boolean; //=TRUE, если в этой
позиции
// загаданной комбинации фишка уже использована при
нии  nRight: integer; //- число угаданных фишек

begin
  //проверить, все ли фишки поставлены в текущий ряд:
  for i:=0 to POLE_WIDTH-1 do
    if masPole[i,hod]= NUM_CURRENTROW then //- не все!
```

```
    begin
      application.MessageBox('Вы должны поставить 4
фишки!',NAME_PROG,0);
      exit
    end;
  //выставлены все фишки - проверяем их на полное совпадение:
  for i:=0 to POLE_WIDTH-1 do
    begin right[i]:= false; checked[i]:= false end;
    nRight:=0;

 for i:=0 to POLE_WIDTH-1 do
    if masPole[i,hod]= answer[i] then
    begin masGuess[nRight,hod]:= NUM_CORRECT; inc(nRight);
      right[i]:= true; checked[i]:= true;
    end;
  //если все фишки стоят на своих местах, игрок выиграл!
  if nRight= POLE_WIDTH then
  begin
    //вывести правильный ответ:
    OutPole;
    application.MessageBox('Вы победили!',NAME_PROG,0);

    //начать новую игру:
    Prepare;
    exit
  end;
  //проверить ответ на частичное совпадение:
  for i:=0 to POLE_WIDTH-1 do
    if right[i]= false then //- на этом месте в ответе не стоит
"правильная" фишка
    //проверить, есть ли такая фишка в загаданной комбинации:
      for j:=0 to POLE_WIDTH-1 do
        if (not checked[j]) and (masPole[i,hod]=answer[j]) then
//- есть
        begin
          masGuess[nRight,hod]:= NUM_INCORRECT; checked[j]:= true;
          inc(nRight); break
        end;
  //номер следующего хода:
  inc(hod);
  if hod= MAX_HOD then //- все ходы уже сделаны!
  begin
    OutPole;
    frmGameOver.Memo1.Lines.Text:='Вы не справились с
заданием.'+#10+  'Попробуйте ещё раз.';
    //вывести правильный ответ:
    Solution;
    //начать новую игру:
    Prepare;
    exit
  end;
  //выделить цветом текущую строку:
  PaintCurrentRow;
```

```
  //обновить поля:
  OutPole;
end; //bbtTestClick
```

Однако, прежде всего, следует убедиться, что игрок выставил в текущий ряд все **четыре** фишки (в запале борьбы всякое бывает!) и в случае «недостачи» напомнить о его обязанностях. Если всё нормально, то проверяем, сколько фишек игрок расставил по своим местам («тютелька в тютельку»). Сделать это очень просто, достаточно сравнить, совпадают ли фишки в одинаковых позициях на игровом поле (`masPole[i,hod]`) и в загаданной комбинации (`answer[i]`). Для каждой правильной фишки увеличиваем их число (**nRight**), ставим в массиве результата хода галочку `masGuess[nRight,hod]:= NUM_CORRECT` и отмечаем, что эта фишка использована (`right[i]:= true; checked[i]:= true;`). При полном совпадении хода и загадки остаётся только поздравить игрока с победой. В противном случае подсчитываем число частичных совпадений. Здесь мы пропускаем фишки, которые отмечены нами ранее как использованные, и сравниваем только оставшиеся. Найдя фишку «не в своей тарелке», помечаем её в массиве кружочком - `masGuess[nRight,hod]:= NUM_INCORRECT`.

Дальше в этой процедуре нет ничего хитрого, отметим только две новые процедуры – **PaintCurrentRow**, которая закрашивает белым цветом следующий текущий ряд

```
//ВЫДЕЛИТЬ ТЕКУЩУЮ СТРОКУ
procedure TForm1.PaintCurrentRow;
var i: integer;
begin
  for i:=0 to POLE_WIDTH-1 do begin
    masPole[i, hod]:= NUM_CURRENTROW;
  end;
end;  //PaintCurrentRow
```

и **Solution** (не забудьте объявить её в разделе **private** описания формы), которая показывает игроку загаданную комбинацию, чтобы у него не возникло сомнений насчёт честной игры:

```
//ВЫВЕСТИ ПРАВИЛЬНОЕ РЕШЕНИЕ
procedure TForm1.Solution;
begin
  frmGameOver.Image1.Picture.Bitmap := (findComponent('bbtColor'+
inttostr(answer[0])) as TBitBtn).Glyph;
  frmGameOver.Image2.Picture.Bitmap := (findComponent('bbtColor'+
inttostr(answer[1])) as TBitBtn).Glyph;
  frmGameOver.Image3.Picture.Bitmap := (findComponent('bbtColor'+
inttostr(answer[2])) as TBitBtn).Glyph;
```

```
  frmGameOver.Image4.Picture.Bitmap := (findComponent('bbtColor'+
inttostr(answer[3])) as TBitBtn).Glyph;
  frmGameOver.showmodal;
end; //Solution
```

Так как места на форме для этой информации нет, то добавим к проекту новую форму – **frmGameOver** (Рис. П1.43).

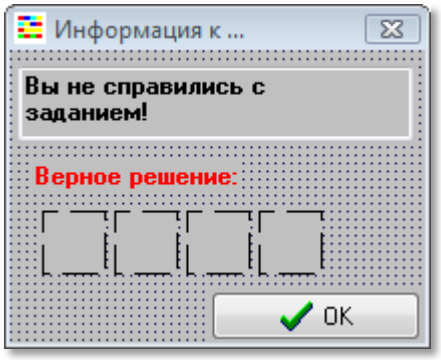


Рис. П1.43. ИнФОРМАтивная форма

Вы можете оформить её как угодно, важно лишь разместить на ней 4 компонента **Image1..Image4** для вывода картинок с загаданными фишками. Обратите внимание, что мы копируем их с кнопок, а не из массива, как в главной форме. Следовательно, массив картинок можно и не заводить – в этой программе, но нам ведь нужно изучить разные способы решения сходных задач!

Свойства **формы** и **компонентов** на ней:

```
frmGameOver:
  BorderIcons = [biSystemMenu]  BorderStyle = bsSingle
  Caption = 'Информация к ...'
  Icon – загрузите значок

Label1:
  Caption = 'Верное решение:'
  Font.Color = clRed    Font.Style = [fsBold]

Image1.. Image4:
  Width = 32    Height = 32
Memo1: TMemo
  Font.Style = [fsBold]
  Lines.Strings = (
      'Вы не справились с заданием! '
      'Попробуйте ещё раз.')
  ReadOnly = True
BitBtn1: TbitBtn
  Kind = bkOK
  OnClick = BitBtn1Click
```

Новую форму нужно объявить в модуле главной формы:

```
implementation
uses GameOverUnit
Допишите также код в модуле дочерней формы:
unit GameOverUnit;
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
Dialogs, StdCtrls, Buttons, ExtCtrls;
type
  TfrmGameOver = class(TForm)
    Memo1: TMemo;
    Label1: TLabel;
    BitBtn1: TBitBtn;
    Image1: TImage;
    Image2: TImage;
    Image3: TImage;
    Image4: TImage;
    procedure BitBtn1Click(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;
var
  frmGameOver: TfrmGameOver;
implementation
uses MindUnit;
{$R *.DFM}
procedure TfrmGameOver.BitBtn1Click(Sender: TObject);
begin
  close
end;
end.
```

Вам всего-то потребуется объявить модуль главной формы и написать обработчик для кнопки. Зато незадачливый игрок сможет получить вот такой «подарок» от программы (Рис. П1.44).

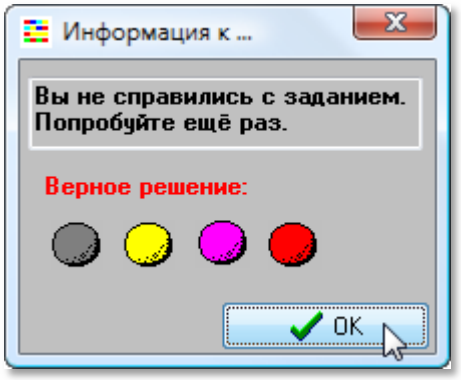


Рис. П1.44. Увы!

И вот настало время вспомнить о **подсказках**. Что это такое, долго рассказывать не надо – они всем известны по школе. Когда не знаешь, какой сделать ход, нажми на кнопку – и программа выведет список всех возможных ходов. Игроку останется только выбрать один из них. Конечно, программа - не человек: сама подсказывать не станет, сколько её ни проси об этом. О подсказках придётся позаботиться самим. Правда, кое-что мы уже сделали: при запуске приложения создали массив **mHint**, в который поместили все возможные комбинации фишек. Естественно, на первом ходу пользоваться им бессмысленно, потому что с тем же успехом можно выставить произвольные четыре фишки в текущий ряд. А вот после первого хода «круг подозреваемых» существенно сузится - число возможных (загаданных) комбинаций будет уже не столь велико (кроме того, в список не будут включены бессмысленные ходы). Нам останется выбрать один ход из списка. А вот какой – думайте сами (хотя опыт игры показывает, что даже при случайном выборе ходов из списка игра всё равно заканчивается довольно быстро).

На нашей форме кнопка-подсказка **bbtHint** имеет рисунок с биноклем:  Установите для неё такие свойства:

```
Left = 304     Top = 372     Width = 32     Height = 32
Cursor = crHandPoint
Hint = 'Подсказка'     ShowHint = True
Glyph – загрузите картинку
OnClick = bbtHintClick
```

Это всё просто, но что происходит в программе после нажатия на эту кнопку? – А вот что. Ежели горе-игрок обратился к подсказке, не сделав ни единого хода, мы должны пристыдить его обидной надписью.

После первого сделанного хода мы уже можем поощрить игрока за энтузиазм и волю к победе, предоставив ему медвежью услугу. Для вывода списка ходов придётся присоединить к проекту ещё одну форму – **frmHint**, - о которой речь пойдёт дальше. А сейчас нам, дай Бог, разобраться с подсказками!

Зная результаты нескольких (по крайней мере, одного) ходов и имея список подсказок для всех ходов за исключением последнего (для первого хода – это весь «спектр» комбинаций), мы можем отобрать из них только те, которые полностью удовлетворяют результатам ходов. Результат хода проверяем по той же схеме, по которой раньше подсчитывали совпадения фишек. И, если текущая комбинация фишек из предыдущего списка подсказок удовлетворяет результатам всех ходов, то заносим её в новый список подсказок, увеличиваем их число **nh** на единицу, и формируем строку с

комбинацией фишек для списка на форме **frmHint**. Замечание: в списке находятся только те комбинации, которые уже проверены для предыдущих ходов, поэтому можно ограничиться последним ходом.

```
//ВЫВЕСТИ СПИСОК ПОДХОДЯЩИХ КОМБИНАЦИЙ
procedure TForm1.bbtHintClick(Sender: TObject);
var i,n,m: integer;
  sovp: array[0..POLE_WIDTH-1] of integer;
  flgNo: boolean;
  s: string;
  right: array[0..POLE_WIDTH-1] of boolean;
  checked: array[0..POLE_WIDTH-1] of boolean;
  nRight: integer;
  nh: integer; //- число подходящих комбинаций
begin
  if hod=0 then begin
    application.MessageBox('Вы хотя бы один ход сдела-
ли!',NAME_PROG,0);
    exit;
  end;
  //очистить список комбинаций:
  frmHint.listbox1.clear;
  nh:=0
  //выбрать подходящие комбинации:
  for i:= 1 to nHint do begin
    //сравнить с введённым рядом фишек -
    //проверяем на полное совпадение:
    for n:=0 to POLE_WIDTH-1 do begin //- по всему ряду фишек
      right[n]:= false; checked[n]:= false; sovp[n]:=NUM_GRID end;
    nRight:=0;
    for n:=0 to POLE_WIDTH-1 do
      if masPole[n,hod-1]=mHint[i,n] then
      begin sovp[nRight]:= NUM_CORRECT; inc(nRight);
        right[n]:= true; checked[n]:= true;
      end;
    //проверить ход на частичное совпадение:
    for n:=0 to POLE_WIDTH-1 do
      if right[n]=false then
        for m:=0 to POLE_WIDTH-1 do
          if (not checked[m]) and (masPole[n,hod-1]=mHint[i,m])
then
          begin
            sovp[nRight]:= NUM_INCORRECT; inc(nRight);
            checked[m]:= true;  break
          end;
     //совпало ли с результатом предыдущего хода в masGuess?
     flgNo:= false;
     for n:=0 to POLE_WIDTH-1 do
       if sovp[n]<> masGuess[n,hod-1] then begin
         flgNo:= true; break;
        end;
```

```
    if flgNo=false then begin //- подходящая комбинация
      //увеличиваем число найденных комбинаций:
      inc(nh);
      s:='';
      //формируем подсказку:
      for n:=0 to POLE_WIDTH-1 do begin s:=s+ intto-
str(mhint[i,n]);
        mhint2[nh,n]:=  mhint[i,n]   end;
      //заносим в список:
      frmHint.listbox1.items.add(s);
    end;
  end;
  //новый массив подсказок:
  mhint:=mhint2;
  //число подсказок в массиве:
  nHint:= nh;
  //показываем форму с подсказками:
  frmHint.Show
end; //bbtHintClick
```

Форма для вывода подсказок может выглядеть так (Рис. П1.45).

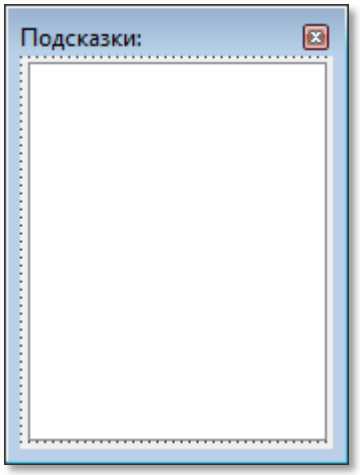


Рис. П1.45. Форма для подсказок

Разумеется, можно набрать одну из комбинаций фишек в списке «вручную», но лучше скопировать «кликнутую» строку из списка в массив **masPole**, а затем обновить игровое поле. Именно так и сделано в программе. Согласитесь, если уж пользоваться подсказками, так с комфортом (Рис. П1.46)!

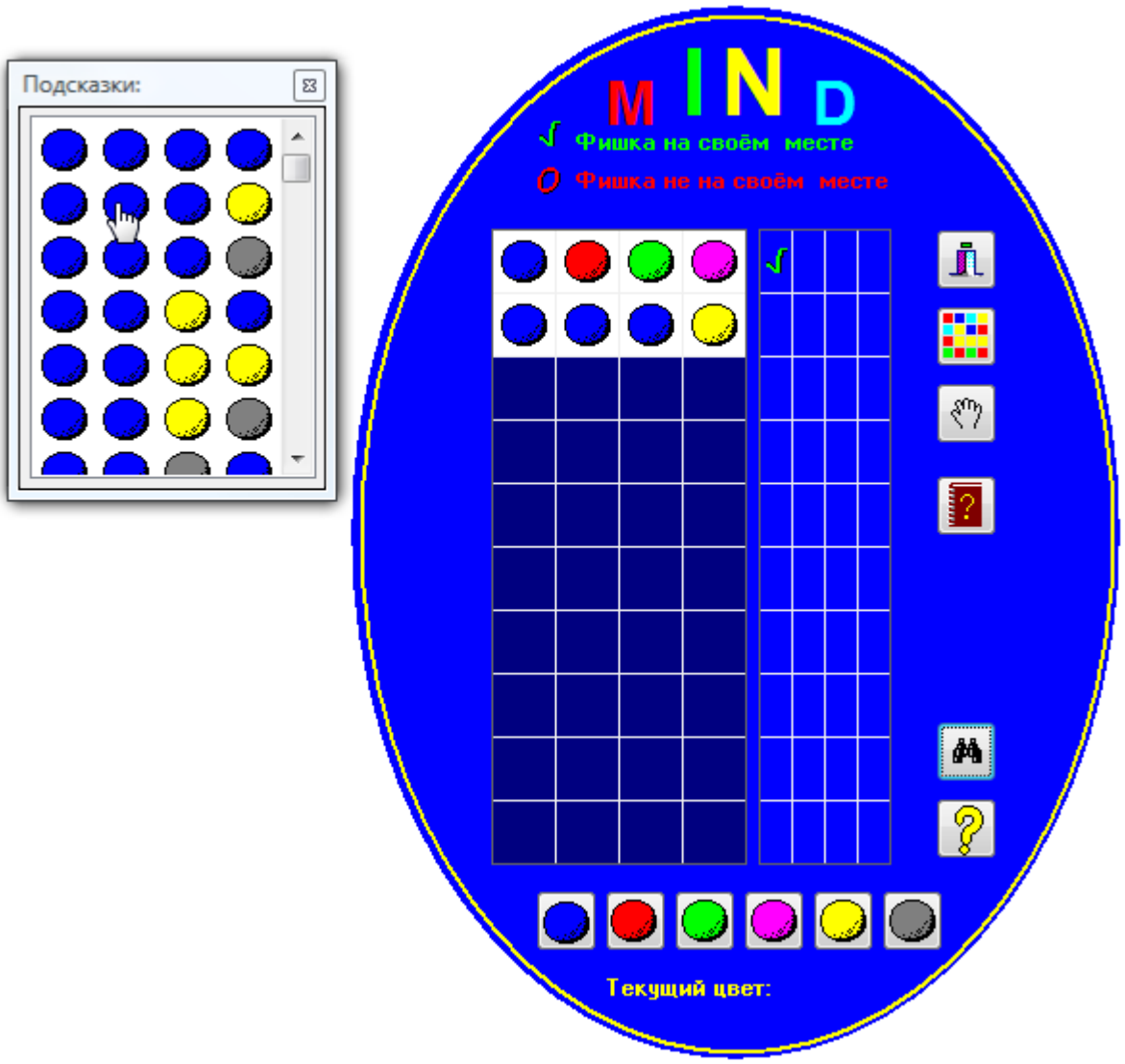


Рис. П1.46. Игра с подсказками!



Очевидно, что, имея список подсказок, можно играть «не думая», то есть выбирать случайным образом один из ходов в списке. Этому легко научить компьютер, поэтому попробуйте написать программу, в которой **вы** задумываете комбинацию фишек, а компьютер её угадывает. Правда, пользы от такой игры будет немного, так что лучше модифицируйте нашу игру так, чтобы вы и компьютер загадывали и угадывали фишки по очереди. Кто найдёт ответ за меньшее число ходов, тот и побеждает в партии из двух «таймов» (конечно, может «выйти» и ничья). Несмотря на то, что компьютер ходит «как попало», а вы обдумываете свои ходы, обыграть его будет совсем непросто!

Ещё более интересно организовать состязание двух алгоритмов. Один нам уже известен – это случайный выбор одного из ходов в

списке. А вот второй придумайте сами! Если он будет стабильно обыгрывать первый – честь вам и хвала. Тем более что ваши упражнения не пройдут даром – в интеллектуальных играх вам часто придётся заниматься созданием искусственного разума.

О другом применении программы *Mind* мы поговорим ниже.

Наверное, вы знаете, что «хитрые» программисты предусматривают для себя возможность «плутовать» в игре, например, делая себя бессмертным. Ну что ж, давайте и мы сделаем для себя поблажку (по правде сказать, программистам эти уловки нужны скорее для отладки игры, чем для победы в ней). Дважды щёлкнув на поле результатов хода, мы увидим загаданные фишки. Особо пронырливые пользователи методом ненаучного тыка тоже могут воспользоваться этим приёмчиком, поэтому их следует пристыдить обидной надписью:

```
 //ПОСМОТРЕТЬ ЗАГАДАННУЮ КОМБИНАЦИЮ
procedure TForm1.dgGuessDblClick(Sender: TObject);
begin
  frmGameOver.Memo1.Lines.Text:='А вы большой хитрец!'+#10+  'Ну
что ж, играйте!';
  Solution;
  OutPole;
end;  //dgGuessDblClick
```

Обеспечив себе возможность заглядывать в ответ, мы можем без опаски сыграть с компьютером несколько партий. «В натуре» это выглядит примерно так (Рис. П1.47).



Рис. П1.47. Игра в разгаре!

Если вдруг игра не заладилась, а пользоваться подсказками гордость не позволяет, вы можете начать новую игру, кликнув по кнопке **bbtNew** - :

```
    Left = 304    Top = 156    Width = 32    Height = 32    Cursor
= crHandPoint
    Hint = 'Начать новую игру'    ShowHint = True
    Glyph – загрузите картинку
    OnClick = bbtNewClick
```

```
//НАЧАТЬ НОВУЮ ИГРУ
procedure TForm1.bbtNewClick(Sender: TObject);
begin
  Prepare;
end;  //bbtNewClick
```

А если вы решили поднять руки вверх, но любопытство распирает вас? Тогда воспользуйтесь кнопкой **bbtPeek** - , чтобы приободриться и узнать-таки, что за каверзную комбинацию фишек учудила умная машина. Свойства кнопки:

Left = 304 **Top** = 196 **Width** = 32 **Height** = 32 **Cursor** = crHandPoint
Hint = 'Сдаюсь!' **ShowHint** = True
Glyph – загрузите картинку
OnClick = bbtPeekClick

```
//СДАЮСЬ!
procedure TForm1.bbtPeekClick(Sender: TObject);
begin
  frmGameOver.Memo1.Lines.Text:='Не огорчайтесь!'+#10+ 'Попробуйте
ещё раз!';
  Solution;
  OutPole;
  //начать новую игру:
  Prepare;
end;  //bbtPeekClick
```

И последнее, что нам осталось сделать в этой игре, – это подготовить помощь для будущих благодарных пользователей. Разместим на форме кнопку **bbtHelp**:

Left = 304 **Top** = 244 **Width** = 32 **Height** = 32 **Cursor** = crHandPoint
Hint = 'Правила игры' **ShowHint** = True
Glyph – картинка 
OnClick = bbtHelpClick

Из опыта программирования этой игры нам уже известно, что для вывода обширных сообщений места на форме нет (издержки «изящных» форм!), поэтому добавим к проекту специальную форму для вывода помощи – **frmHelp**:

BorderIcons = [biSystemMenu, biMinimize] **BorderStyle** = bsSingle
Caption = ' MIND GAME'
Icon – загрузите подходящий значок

Для размещения строк справки приспособим компонент **Memo1**. У вас ещё будет возможность убедиться, что это не лучшее решение для вывода справочной информации, но пока мы не будем столь далеко отклоняться от темы главы, чтобы заниматься созданием полноценной справочной системы. Установите свойства компонента:

```
    Left = 4    Top = 4    Width = 233    Height = 329
    Lines.Strings = (
                    Правила игры:
                   -------------------------
   Компьютер загадывает комбинацию
из четырёх фишек шести цветов, причём
фишки одного цвета могут повторяться.
   Ваша задача - найти эту комбинацию
не более чем за 10 ходов.

   Каждый ход заключается в том, что вы
собираете свой ряд из четырёх фишек,
нажимаете кнопку "Проверить ход" - и
получаете результат сравнения вашей
последовательности с фишек с искомой.

   Если фишка в вашем ряду совпадает
по цвету и по месту в цепочке с
загаданной, то это событие отмечается
галочкой справа. Если же цвет фишки
угадан, а её место - нет, то выводится
кружок.

   На основании полученных данных вы
можете сделать предположение о
загаданной комбинации фишек и ввести
следующий ряд фишек.

   Всего существует 6*6*6*6 = 1296
различных цепочек, так что просто
угадать за 10 ходов вам вряд ли
удастся, поэтому нужно думать над
каждым ходом. Для чего, собственно,
 эта игра и придумана.
-----------------------------------------------------------------

   Если вы кликнете на клетке в игровом
ряду поля (он выделен белым цветом),
то в ней появится фишка текущего цвета
(его можно видеть в самом низу поля).
Для изменения текущего цвета
пользуйтесь клавишами с
изображением соответствующей фишки.

   Если вы кликнете на клетке
предыдущего ряда, то в игровом ряду
под этой клеткой будет выставлена
точно такая же фишка.

  Для ускорения набора комбинаций вы
можете воспользоваться кнопкой
"Подсказка" - тогда появится список
```

```
всех подходящих комбинаций. Щёлкните
на одной из них - и она появится в
игровом ряду.)

    ReadOnly = True
    ScrollBars = ssVertical
```

Весь текст нужно набрать в *Редакторе строк*, который открывается, когда вы дважды щёлкнете на окошке ввода свойства **Lines** (там вы увидите строку *TStrings*) компонента **Memo1** в *Инспекторе объектов* (Рис. П1.48, 49).

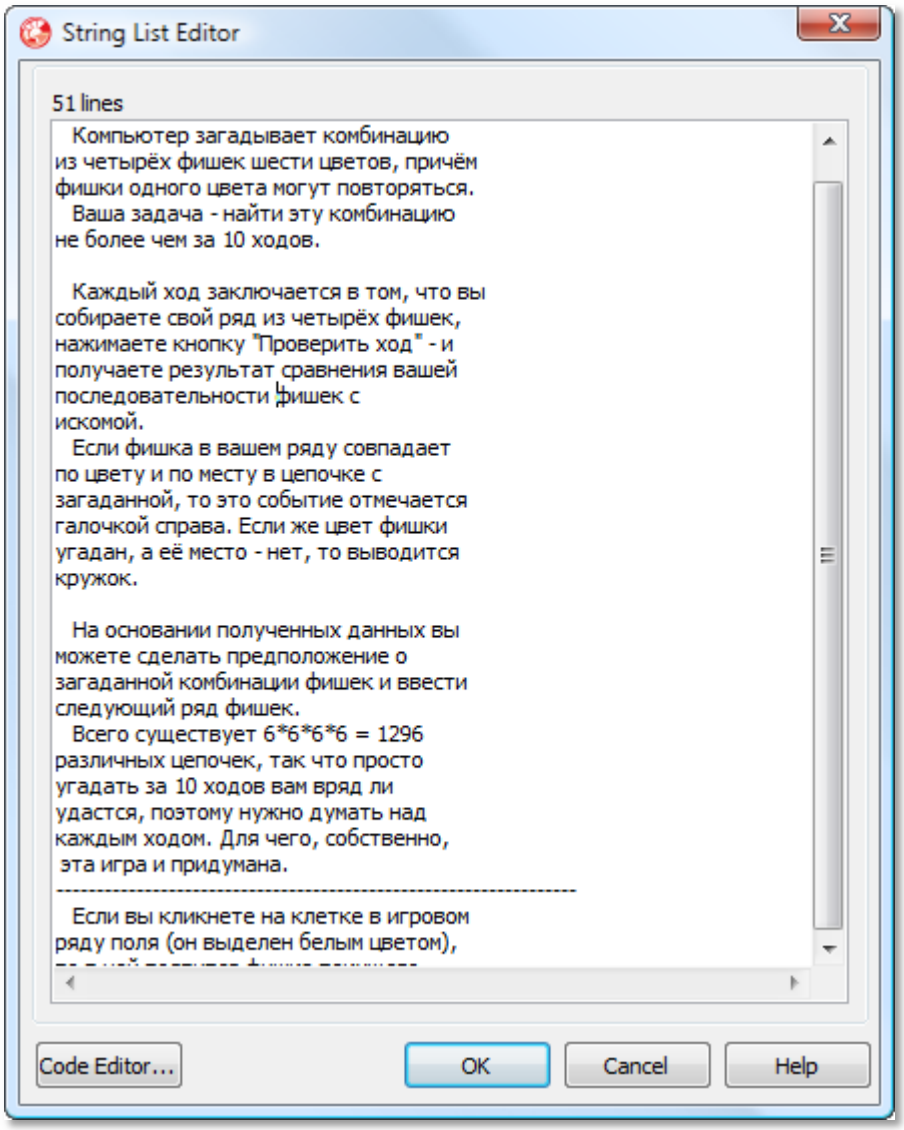


Рис. П1.48. Правила игры в *Редакторе строк*

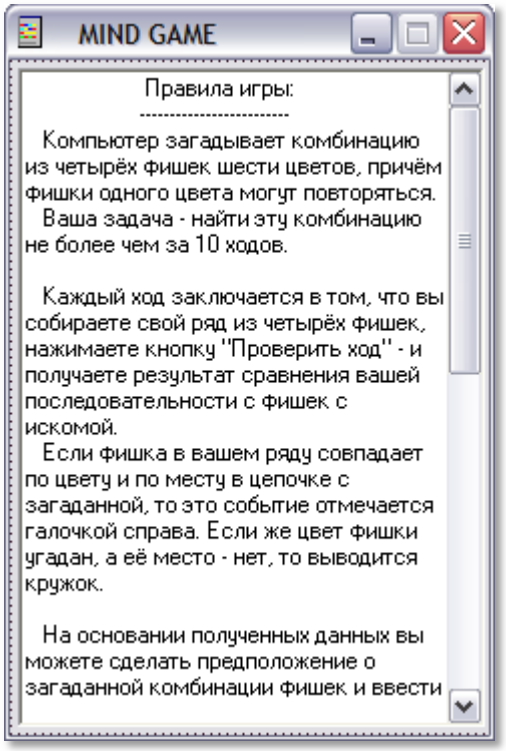


Рис. П1.49. Правила игры в компоненте *Memo1*

Конечно, текст справки вы можете редактировать до полного прояснения.

Как превратить игру в головоломку

Многие логические игры – и **Mind Game** не исключение – можно использовать для составления головоломок. В нашей игре информация к игроку поступает порциями – после каждого хода, поэтому он не может однозначно определить загаданную комбинацию фишек, не сделав предварительно несколько ходов (конечно, игрок может просто угадать ответ, но это дело случая, а не ума). В головоломке же, напротив, отгадчику должны быть сразу известны **все** необходимые данные. Ему остаётся только с помощью логических размышлений вывести из них единственно правильный результат.

Например, рассмотрим решение такой головоломки (Рис. П1.50).

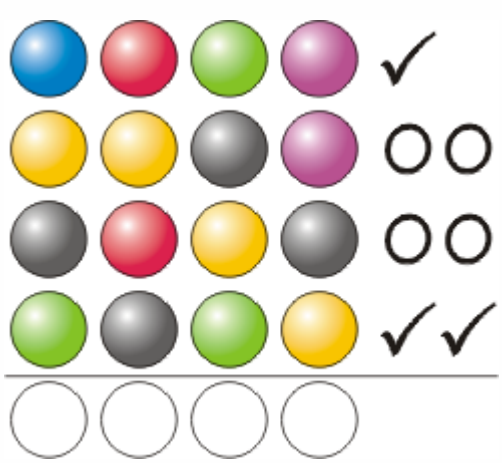


Рис. П1.50. Головоломка!

1. В первых двух строках (рядах) лиловая (она же пурпурная) фишка находится в одной и той же позиции. Если она присутствует в загаданной комбинации, то она может занимать только последнее место в цепочке (в первой строке стоит галочка). Но тогда и во второй строке должна быть хотя бы одна галочка, а её нет. Отсюда следует вывод: **лиловой фишки вообще нет** среди задуманных.

2. В строках 1 и 3 **красная** фишка также занимает одно и то же место в цепочке. И по той же причине, что и лиловая **фишка**, **отсутствует** в задуманной комбинации.

3. В строках 2 и 3 находятся, соответственно, две жёлтых и одна серая и 2 серых и одна жёлтая фишки. Из них в загаданной цепочке точно имеются две – либо две жёлтых, либо две серых, либо одна жёлтая и одна серая фишки.
а. Если загаданы две жёлтые фишки, то они могут занимать в цепочке места 3 и 4 (иначе в строке 2 была бы, по крайней мере, одна галочка), но тогда в строке 3 жёлтая фишка дала бы галочку, а не кружок. Следовательно, в загаданной комбинации не может быть одновременно двух жёлтых фишек.
б. Рассуждая аналогично, приходим к выводу, что в исходной цепочке не может быть и двух серых фишек.

Вывод: **в загаданной цепочке имеется одна жёлтая и одна серая фишки**.
4. В строке 4 как раз имеются эти фишки, значит, они и дают две галочки – места этих фишек в цепочке найдены, а **зелёных фишек в цепочке нет**.

5. Тогда из строки 1 следует, что **синяя фишка возглавляет шеренгу**.

6. Осталось определить, какая фишка находится в третьей позиции цепочки. Это не может быть зелёная, серая, жёлтая, лиловая, красная фишка. Остаётся только **синяя** **фишка** – именно она и занимает это место.

После этих логических изысканий мы можем со всей ответственностью утверждать, что была загадана такая цепочка (Рис. П1.51).

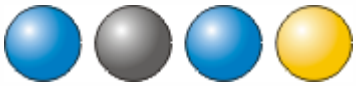


Рис. П1.51. Разгадка!

Ничего не скажешь – отличная логическая задача! А теперь о том, как с помощью нашей программы можно очень легко составить сколько угодно таких заданий.

Запустите игру **Mind**, выставите в первом ряду любую комбинацию из четырёх фишек и проверьте её. Если вы, «к несчастью», сразу угадали цепочку, то начните решать следующую. Получив результат первого хода, нажмите кнопку , чтобы открыть список подсказок. Выберите по своему вкусу любую комбинацию фишек и щёлкните на ней. Таким образом, вы сделаете второй ход. Узнайте, сколько фишек вы угадали, и снова разверните список подсказок. Если в нём осталась **единственная** комбинация фишек, то она и была задумана (если же вы «по неосторожности» решили задачу, то просто начните всё сначала). Вам нужно только переписать все ходы вместе с результатами – и задача готова! Как видите, простыми средствами вы можете составлять прекрасные логические задачи. И что важно – вам даже не придётся проверять корректность их решения, никаких дуалей и побочных вариантов в решении задач не будет.



Вы без особого труда можете научить нашу программу решать такие задачи. Самое «трудное» здесь – добавить пару кнопок для галочки и кружочка и процедуру для размещения их в сетке результатов хода. После этого выставляйте нужные фишки, галочки и кружочки, заполняя массивы **masPole** и **masGuess** соответствующими значениями. И вам останется только выбрать из массива подсказок **mHint** те, которые удовлетворяют всем условиям задачи. В списке должна остаться, естественно, всего одна строка – с загаданной комбинацией фишек. Если это не так, значит, задача составлена неверно.

Разумеется, решать свои же задачи не очень интересно, а других задач вы не найдёте (к сожалению!) ни в одной газете. Однако почти так же можно решать и другие логические задачки, поэтому ваши упражнения не будут совсем бесполезны!

Хорошо составленную задачу вполне можно послать и в какую-нибудь газету. Давайте попробуем подготовить такое задание!



Исходный код программы находится в папке **MIND**.

Золотой ключик, или Воспоминания Буратино

...дцатого ...бря ...го года (точной даты не помню). Ночью я был подвешен вниз головой неизвестными разбойниками, а поутру меня избавила от петли Мальвина, девочка с голубыми волосами (несколько лет спустя она вышла замуж за Карабасяна и нынче работает на телевидении).

Поставив меня на ноги с помощью пса Артемона и касторки, Мальвина взялась за моё воспитание и обучение. Очень быстро выяснилось, что у меня совершенно отсутствуют способности как к первому, так и ко второму. Но, будучи ярым последователем Монтессори и Никитина, Мальвина на этом не успокоилась и попыталась заинтриговать меня игровым моментом.

Она сняла со своей прекрасной фарфоровой шейки разноцветные бусы и сказала:

– Буратино, вот бусинки шести разных цветов – красного, синего, лилового, серого, жёлтого и зелёного. Сейчас я возьму четыре бусинки (только, чур, не подглядывать!), а вы постарайтесь угадать, какого они цвета и какую бусинку я взяла первой, какую – второй и так

далее. Вы тоже возьмите четыре бусинки и разложите их в нужном порядке, а я вам подскажу, какие из них есть среди загаданных: если бусинки совпадут по цвету и по "номеру", то я скажу "галочка", а если только по цвету - "кружочек". Например, если я возьму бусинки КРАСНУЮ-КРАСНУЮ-ЖЁЛТУЮ-ЗЕЛЁНУЮ, а вы - ЗЕЛЁНУЮ-КРАСНУЮ-КРАСНУЮ-КРАСНУЮ, то получите в ответ такую подсказку: "галочка и два кружочка". Вы всё поняли, Буратино?

Сказать по правде, я и по сей день так до конца и не уразумел всех премудростей этой игры, но тогда - не мог же я признаться в этом какой-то девчонке! - я бодро крикнул: понял, чего тут не понять!!! Шесть раз Мальвина прятала бусинки, и каждый раз я разгадывал её хитрые штучки! И тогда она, с досады, упрятала меня в тёмный чулан (а вовсе не за кляксу в тетради, как об этом сочинил Толстой).

Я записал все наши баталии на бумаге, но в том самом чулане крыса Шушара отгрызла часть листка - как раз ту, на которой были мои последние, решающие ходы. Говорят, что их можно восстановить, но, увы, я уже не тот... Хотите - попробуйте сами (Рис. П1.52)!

А чтобы у вас не сложилось превратное впечатление о моих отношениях с Мальвиной, должен добавить, что благодаря ей я окончил Экономический университет при Дворце пионеров и стал Министром финансов в Стране дураков. О, сколько бюджетных денег я закопал в землю на Поле чудес, пока не отошёл от дел! Но это уже совсем другая история...

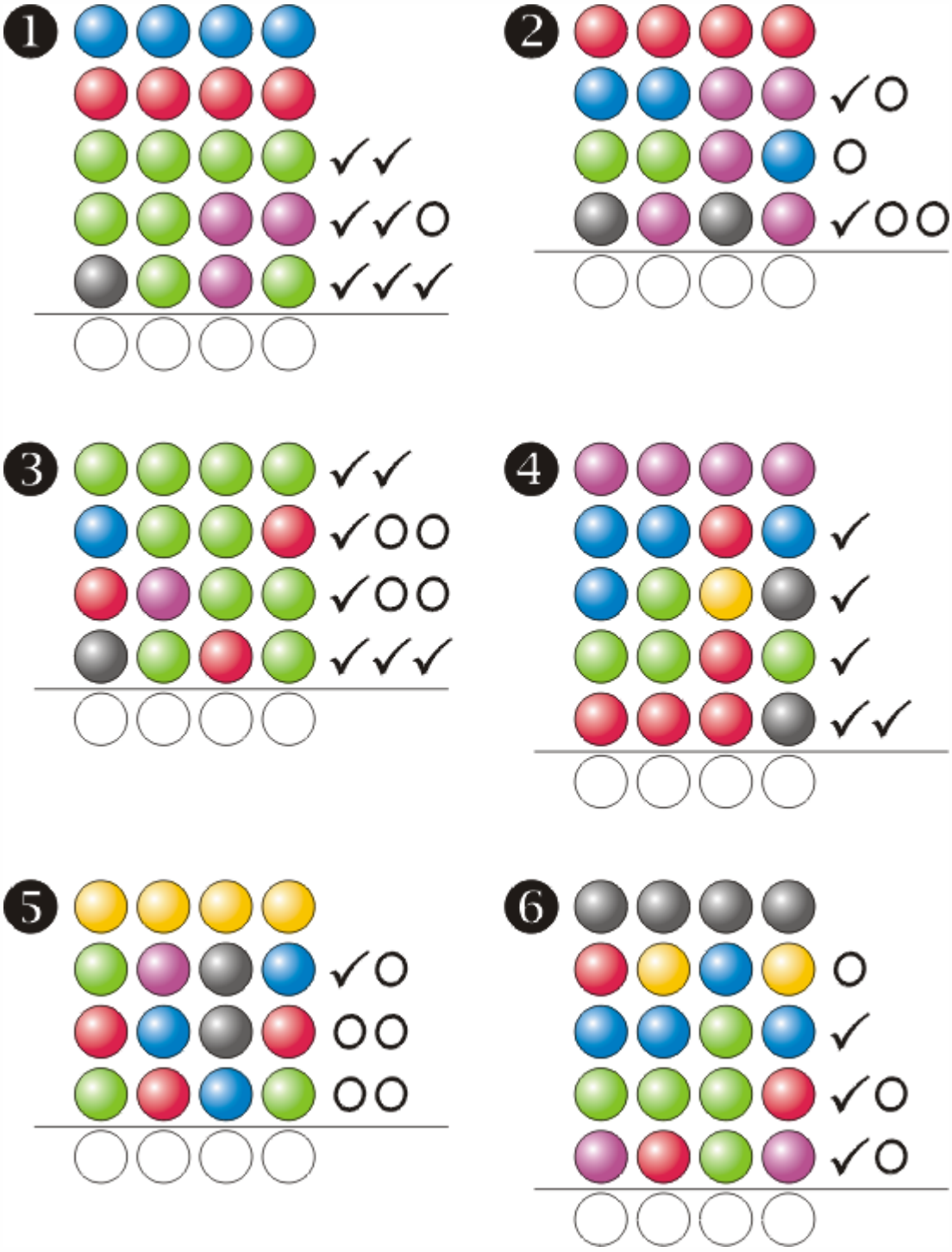
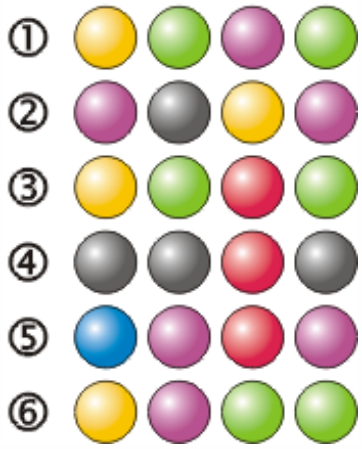


Рис. П1.52. Головоломки Буратины

Ответ – на следующей странице.

Ответ:



Полезный совет. Можно и, наоборот, головоломку переделать в игру. Примером этому может служить знаменитый *тетрис*, которым мы сейчас и займёмся.

Занятие 2. *Тетрис*, или Великолепная семёрка

Что нам стоит дом построить –
нарисуем, будем жить.

Архитектурная утопия

А всё-таки она вертится!

Галилео Галилей

Буйная фантазия древних греков породила кентавра, удивительное существо: наполовину человек, наполовину – конь. Нам предстоит повторить мифологический подвиг эллинов – мы напишем программу, в которой научимся создавать окна самой причудливой формы. А чтобы наша программа не стала пустой демонстрацией абстрактных изысков, мы сделаем из неё настоящий *тетрис*!

Тетрис, - пожалуй, самая известная компьютерная игра – был придуман московским программистом Алексеем Пажитновым несколько десятилетий тому назад и с тех пор был реализован на всех компьютерах, калькуляторах и швейных машинках. Совершенно невозможно рассказать обо всех вариациях *тетриса*, поэтому для примера приведём только несколько картинок (Рис. П2.1).



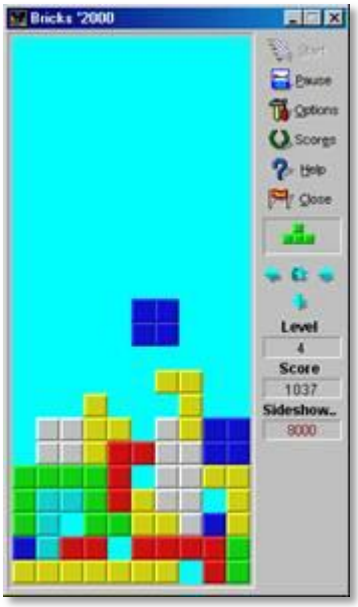






Рис. П2.1. *Тетрисы* на любой вкус!

Есть ещё и трёхмерные, объёмные, *метрисы*, созданные специально для инопланетян, пентиксы, бриксы и прочая неразбериха, которых мы и вовсе касаться не будем, дабы не сгинуть навсегда в их безумном разнообразии. Наш выбор - классический *метрис*, игра насколько простая, настолько и увлекательная: сверху падают цветные четырёхклеточные фигурки, которые игрок должен быстро, но аккуратно укладывать ровными рядами. Вот и вся недолга! И несмотря на это интерес к *метрису* не угас и по сей день.

А куда интереснее сделать **собственного** *метриса*, наделить его неповторимыми чертами и получить хотя и виртуального, но настоящего друга для игр и развлечений, называемых жизнью.

Новый Пигмалион, или Картинная форма

Начнём же мы с создания оригинальной **формы** для нашей игры. Технологии, использованные при этом, можно применить и в других программах, так как они достаточно просты и универсальны, а получающиеся в результате окна украсят любую программу.

Прежде всего, конечно, нужно нарисовать форму в каком-нибудь растровом графическом редакторе. При этом учитывайте, что цвет точки (пиксела), находящейся в нижнем левом углу рисунка, будет прозрачным, то есть точки этого цвета не являются «формообразующими». Если вам эта точка дорога как память, то задайте в процедуре **FormCreate** другую точку или укажите другой цвет при вызове функции **fcCreateRegionFromBitmap** (основа алгоритма позаимствована из библиотеки компонентов *First Class*), которая по картинке формирует область, состоящую из всех **непрозрачных** точек растра. При создании формы удобно выполнить и некото-

рые подготовительные операции: очистить массивы и загрузить таблицу лучших результатов:

```
//СОЗДАТЬ ФОРМУ
procedure TForm1.FormCreate(Sender: TObject);
var
  TransColor: TColor;
  i, j: integer;
  appPath, s: string;
  spisok: TIniFile;
begin
  //за прозрачный цвет принимаем цвет точки в левом нижнем углу
  //картинки:
  TransColor:= Im-
age1.Picture.Bitmap.Canvas.Pixels[0,Image1.Picture.Bitmap.height-
1];

  //создать область из непрозрачных точек картинки:
  rgn:= fcCreateRegionFromBitmap(Image1.Picture.Bitmap, TransCol-
or);

  //задать форму окна по области rgn:
  SetWindowRgn(Handle, rgn, True);

  Randomize;
  //очистить массивы:
  for j:= 0 to 3  do
    for i:= 0 to 3  do
      masNext[i,j]:= -1;

  for j:= -4 to POLE_HEIGHT-1  do
    for i:= 0 to POLE_WIDTH-1  do
      masPole[i,j]:= -1;

  //загрузить список лучших игроков:
  appPath:= ExtractfilePath(Application.ExeName);
  spisok:= TIniFile.Create(appPath+'Records.ini');
  for i:= 1 to 10 do begin
    s:= 'name'+inttostr(i);
    top10[i].name:= spisok.ReadString('NAME', s, 'Tetris');
    s:= 'res'+ inttostr(i);
    top10[i].score:= spisok.ReadInteger('SCORE', s, 0);
  end;

  spisok.Free;

end; //FormCreate
```

О рекордах мы поговорим дальше, а пока разберёмся с конструированием фигурной области. Всё таинство совершается в функции **fcCreateRegion-**

FromBitmap, которая возвращает область, созданную из непрозрачных точек битовой картинки **ABitmap**. Действует она так. Сверху вниз сканируется исходный растр, и найденные непрозрачные точки добавляются к результирующей области, которая при входе в функцию обнуляется:

```
function fcCreateRegionFromBitmap(ABitmap: TBitmap; TransColor:
TColor): HRgn;
//ABitmap - исходный растр
//TransColor - прозрачный цвет
var
  TempBitmap: TBitmap;
  Rgn1, Rgn2: HRgn;
  Col, StartCol, Row: integer;
  Line: PByteArray;

  //формат растра?
  function fcIsTrueColorBitmap(Bitmap: TBitmap): boolean;
  begin
    result:= Bitmap.PixelFormat = Graphics.pf24bit;
  end;

  function fcThisThat(const Clause: Boolean; TrueVal, FalseVal:
Integer): Integer;
  begin
    if Clause then result := TrueVal else Result := FalseVal;
  end;

  //вычислить цвет точки:
  function ColToColor(Col: integer): TColor;
  begin
    if fcIsTrueColorBitmap(TempBitmap) then
      result:= Line[Col * 3] * 256 * 256 + Line[Col * 3 + 1] * 256
+ Line[Col * 3 + 2]
    else result := TColor(fcThisThat((Line[Col div 8] and Bit-
Mask[Col mod 8])<> 0, clBlack, clWhite));
    end;

 begin
  //результирующая область пока пустая:
  result := 0;
  if (ABitmap <> nil) and (ABitmap.Width = 0) or (ABitmap.Height =
0) then
    Exit; //- картинка отсутствует!
  //результирующая область:
  Rgn1 := 0; //- пока пустая
  //создать временный растр:
  TempBitmap := TBitmap.Create;
  //и скопировать в него исходный растр:
  TempBitmap.Assign(ABitmap);
  //если формат растра не TrueColor, то -
  if not fcIsTrueColorBitmap(TempBitmap) then
  begin
```

```
    TempBitmap.Mask(TransColor);
    TransColor := clBlack;
  end;
  //сканируем растр и создаём область:
  with TempBitmap do
  begin
    //по всем строкам растра:
    for Row := 0 to TempBitmap.height-1 do
    begin
      Line:= scanLine[row];
      Col := 0;
      //по всей строке:
      while Col < TempBitmap.Width do
      begin
        //пропускаем прозрачные точки:
        while (Col < TempBitmap.Width) and (ColToColor(Col) =
TransColor)do inc(Col);
        if Col >= TempBitmap.Width then Continue;
        //начало непрозрачных точек:
        StartCol := Col;
        //подсчитать их:
        while (Col < TempBitmap.Width) and (ColToColor(Col) <>
TransColor)do inc(Col);
        if Col >= TempBitmap.Width then Col := TempBitmap.Width;
        //если область ещё не создана, то создаём её из текущей
строки пикселей:
        if Rgn1 = 0 then Rgn1 := CreateRectRgn(StartCol, Row, Col,
Row + 1)
        else begin
          //иначе добавляем к области пиксели текущей строки:
          Rgn2 := CreateRectRgn(StartCol, Row, Col, Row + 1);
          if (Rgn2 <> 0) then CombineRgn(Rgn1,Rgn1,Rgn2,RGN_OR);
            Deleteobject(Rgn2);
        end;
      end;
    end;
  end;

  //результирующая область:
  result := Rgn1;
  TempBitmap.Free;
end; //fcCreateRegionFromBitmap
```

Сам исходный растр загружается в компонент **Image1**, размещённый на форме. В нашем примере форма **Form1** имеет такие свойства:

AutoSize = True
BorderStyle = bsNone
KeyPreview = True – для обработки нажатия клавиш на клавиатуре (с их помощью мы будем управлять движением фигурок).

Размеры формы устанавливаем точно по размерам картинки (**AutoSize** = True, **BorderStyle** = bsNone).

Дважды щёлкните по компоненту **Image** в *Палитре компонентов* – и он появится на форме (Name=**Image1**). Передвиньте его в левый верхний угол формы (**Left** = 0, **Top** = 0) и установите **AutoSize** = True, чтобы подогнать под размеры картинки. Загрузить её очень просто. Опять щёлкните дважды – на этот раз на **Image1** и в открывшемся диалоге загрузки графических файлов выберите заранее подготовленную растровую картинку с расширением *BMP*. Забегая в скорое будущее, мы увидим форму с загруженной картинкой: она стилизована под дерево и имеет весьма необычную конфигурацию. Цвет фона, он же прозрачный, здесь красный (естественно, на «дереве» точек такого цвета быть не должно, иначе образуются «дупла») (Рис. П2.2).

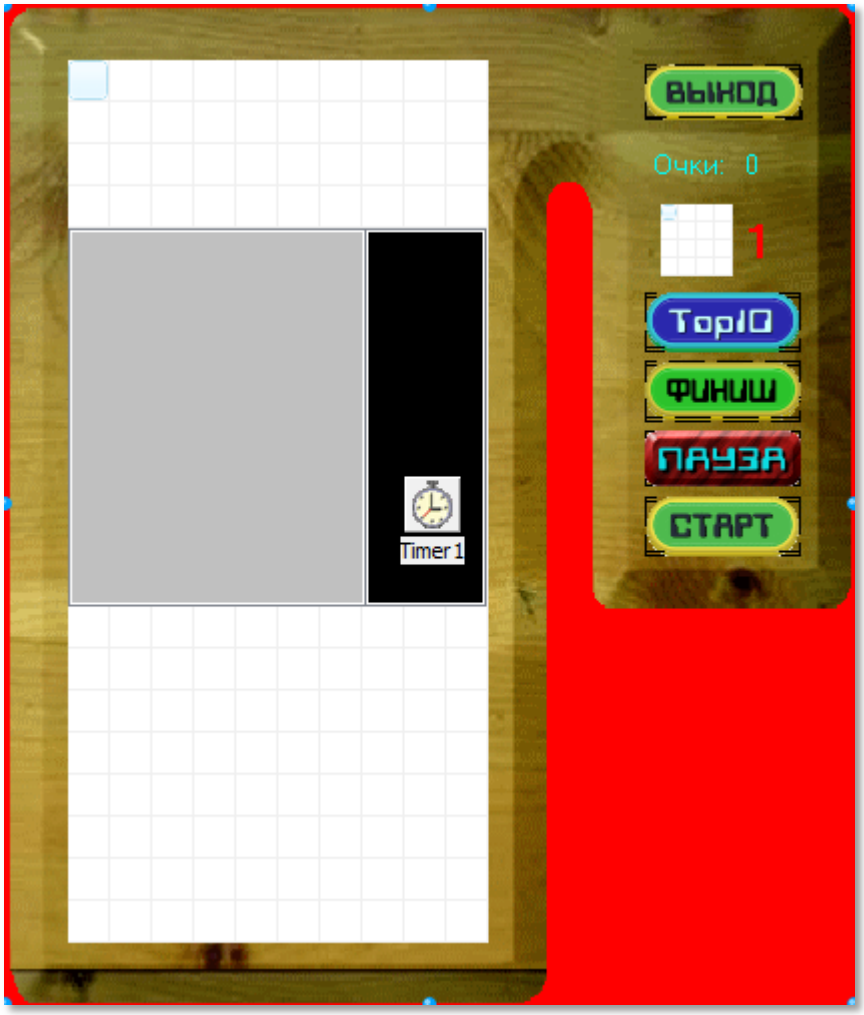
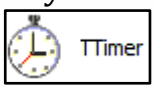


Рис. П2.2. Тетрисовый интерфейс

Перед выходом из программы нужно прибрать за собой:

```
//УНИЧТОЖИТЬ ФОРМУ
procedure TForm1.FormDestroy(Sender: TObject);
begin
 DeleteObject(rgn);
 Timer1.Enabled:= False;
end; //FormDestroy
```

Коли мы упомянули таймер, то давайте двойным щелчком на компоненте **Timer** 

(страница *System Палитры компонентов*) загоним его на форму (**Name** = Timer1) и выключим (**Enabled** = False), чтобы он не тикал без нужды.

Поскольку форму мы "обезглавили", то передвигать её, ухватив за заголовок (**Caption**), не удастся. Более того, за форму вообще невозможно взяться, так как она целиком закрыта «дровами». Но есть **очень полезный** способ таскать форму за что угодно, хотя бы и за картинку **Image1**:

```
//ПЕРЕТАСКИВАНИЕ ФОРМЫ ЗА ЕЁ ПОЛЕ
procedure TForm1.Image1MouseDown(Sender: TObject; Button: TMouse-
Button;
Shift: TShiftState; X, Y: Integer);
const
  SC_DragMove = $F012;
begin
  ReleaseCapture;
  Perform(WM_SysCommand, SC_DragMove, 0)
end; //Image1MouseDown
```

Никакая военно-полевая игра немыслима без игровой доски, в качестве которой у нас пожизненно служит компонент **TDrawGrid**:

```
    Name = dgPole
    Left = 32
    Top = 28
    Width = 210
    Height = 441
    Cursor = crHandPoint
    BorderStyle = bsNone
    Color = clBlack
    ColCount = 10
    RowCount = 21
    DefaultColWidth = 20
    DefaultRowHeight = 20
    DefaultDrawing = False
    FixedCols = 0
    FixedRows = 0
    Options = [goVertLine, goHorzLine]
    ScrollBars = ssNone
```

Размеры поля в игре не изменяются и для удобства хранятся в *константах*:

```
POLE_WIDTH= 10;    //ширина игрового поля
POLE_HEIGHT= 21;   //высота игрового поля
```

Меньшого братца этого поля используем для дефиле следующей фигурки (если вы способны думать на ход вперёд себя, то укладывайте падающую фигурку с учётом следующей):

```
Name = dgNext
Left = 328
Top = 100
Width = 36
Height = 36
BorderStyle = bsNone
Color = clWhite
ColCount = 4
RowCount = 4
DefaultColWidth = 8
DefaultRowHeight = 8
DefaultDrawing = False
FixedCols = 0
FixedRows = 0
Options = [goVertLine, goHorzLine]
ScrollBars = ssNone
```

Умелые руки, или Кнопки – своими руками

Займёмся **кнопками**. Обычные, стандартные кнопки *Delphi* на шикарной форме выглядели бы несуразно. Можно воспользоваться готовыми компонентами частных производителей, но мы-то хотим всё сделать собственными руками! Разрабатывать кнопку в виде компонента мы сейчас не будем, это длинная история, а попробуем сымитировать кнопки из подручного материала. Вполне сгодятся для этого всё те же компоненты **Image**. На каждую кнопку их потребуется пара (можно обойтись и одним, но тогда кнопка не сможет отобразить охватившей её радости от поглаживания мышкой).

В нужном месте формы (конечно, сразу вы его не найдёте, придётся подвигать) размещаем друг над другом со смещением 2 пикселя пару компонентов **TImage**, в которые загружаем картинки с видами кнопки – не нажатой и нажатой. Для примера рассмотрим дизайн кнопки **СТАРТ**, дальше действуйте аналогично.

```
Name = imgStart
Left = 320
Top = 246
Width = 78
Height = 28
Cursor = crHandPoint
AutoSize = True
Transparent = True

Name = imgStart2
Left = 320
Top = 248
Width = 78
Height = 28
Cursor = crHandPoint
AutoSize = True
Transparent = True
Visible = False
```

Обратите внимание, что вторая картинка невидима (свойство **Visible** = False), поэтому при запуске программы мы сможем видеть только **imgStart** (левая картинка, чёрная надпись). Нажав на кнопку **СТАРТ**, мы должны увидеть **imgStart2** (правая картинка, красная надпись):



Для этого нужно скрыть верхнюю картинку, присвоив свойству **Visible** компонента **imgStart** значение False, и, наоборот, сделать видимой нижнюю картинку (**imgStart2: Visible**= True). Одновременно изображение кнопки сдвинется вниз, симулируя смещение при вдавливании.

Если вы не сможете найти готовых кнопок для своей программы, то довольно легко их можно «изготовить» в какой-нибудь программе для оформления *Web*-страниц или в обычном растровом графическом редакторе типа *Adobe PhotoShop*. Записать их лучше в формате *BMP* (чтобы фон кнопок был прозрачным, нужно установить в компонентах **TImage: Transparent** = True). Тогда кнопке можно придать любую форму, не обязательно прямоугольную. Конечно, работать с двумя компонентами **Image** более хлопотно, чем с одним **SpeedButton**, но зато вы можете обработать кнопку до нужной формы.

Бегло познакомимся с другими кнопками:

```
Name = imgExit
Left = 320
Top = 30
AutoSize = True
Transparent = True
```

```
Name = imgExit2
Left = 320
Top = 32
AutoSize = True
Transparent = True
Visible = False
```




```
Name = imgRecords
Left = 320
Top = 144
AutoSize = True
Transparent = True

Name = imgRecords2
Left = 320
Top = 146
AutoSize = True
Transparent = True
Visible = False
```




```
Name = imgFinish
Left = 320
Top = 178
Cursor = crHandPoint
AutoSize = True
Transparent = True

Name = imgFinish2
Left = 320
Top = 180
AutoSize = True
Transparent = True
Visible = False
```




```
Name = ImgPause
Left = 320
Top = 213
AutoSize = True
Transparent = True

Name = ImgPause2
Left = 320
Top = 213
AutoSize = True
```

```
Transparent = True
Visible = False
```



Статус программы опишем в типе **TgameStatus**, а хранить будем в переменной **GameStatus**:

```
//состояние игры:
type TgameStatus=(Stop, Pause, Play);
//текущее состояние игры:
GameStatus: TgameStatus= Stop;
```

Пока игра не запущена (не нажата кнопка **СТАРТ**), **GameStatus** = Stop. Если игра приостановлена (нажата кнопка **ПАУЗА**), **GameStatus** = Pause. Иначе **GameStatus** = Play. При нажатии той или иной кнопки программа проверяет статус игры, а уже затем выполняет соответствующие действия.

Нажмём кнопку **СТАРТ** - если игра ещё не запущена, запускаем её. Показываем нажатую кнопку и убираем с экрана таблицу рекордов (о ней речь пойдёт ниже).

```
//НАЖАТЬ КНОПКУ СТАРТ
procedure Tform1.imgStartMouseDown(Sender: Tobject; Button:
TmouseButton; Shift: TshiftState; X, Y: Integer);
begin
  if GameStatus<> Stop then exit;

  imgStart.Visible:= False;
  imgStart2.Visible:= True;
  HideRecords;
end; //imgStartMouseDown
```

Остальные предстартовые действия переносим в процедуру, которая срабатывает при *отпускании* кнопки **СТАРТ**. Возвращаем на место верхнюю кнопку, очищаем игровое поле от обломков предыдущей игры, выбираем две фигурки – одна начнёт свое падение, вторая будет ждать своего часа (её можно увидеть в маленькой сетке сверху). Обнуляем очки (набранные в текущей игре очки хранятся в переменной **Score**) и выводим их на экран. Для этого используем два компонента **Label**:

```
Name = lblScore
Left = 370
Top = 72
Width = 7
Height = 16
Caption = '0'
Font.Color = clAqua
```

```
    Font.Height = -13
    Font.Name = 'MS Sans Serif'
    Font.Style = []
    Transparent = True

    Name = Label1
    Left = 324
    Top = 72
    Width = 36
    Height = 16
    Caption = 'Очки:'
    Font.Color = clAqua
    Font.Height = -13
    Font.Name = 'MS Sans Serif'
    Font.Style = []
    Transparent = True
```

И последнее – устанавливаем скорость игры и запускаем *таймер*:

```
//СТАРТ НОВОЙ ИГРЫ
procedure Tform1.imgStartMouseUp(Sender: Tobject; Button: Tmouse-
Button;
  Shift: TshiftState; X, Y: Integer);
var
  I, j: integer;
begin
  imgStart.Visible:= True;
  imgStart2.Visible:= False;
  if GameStatus<> Stop then exit;

  //очистить игровое поле:
  for j:= -4 to POLE_HEIGHT-1 do
    for I:= 0 to POLE_WIDTH-1 do
      masPole[I,j]:= -1;

  //выбрать текущую фигуру со случайным поворотом:
  NewFigure(CurrFigure);

  //выбрать следующую фигуру со случайным поворотом:
  NewFigure(NextFigure);

  //показать следующую фигуру:
  ShowNextFig;

  //обнулить очки:
  Score:= 0;
  lblScore.Caption:= '0';

  GameStatus:= Play;

  //установить скорость падения фигурок:
  Timer1.Interval:= 1000 div Speed;
```

```
  //запустить таймер:
  Timer1.Enabled:= true;
end; //imgStartMouseUp
```

Скорость игры хранится в переменной **Speed** и сначала равна 1. При такой скорости интервал срабатывания таймера – 1000 миллисекунд (1 секунда), то есть фигурка будет смещаться вниз на 1 клетку каждую секунду. Это довольно медленно, поэтому вы можете увеличить скорость игры и, соответственно, частоту срабатывания таймера. Заданная скорость отображается в компоненте **Label**:

```
Name = lblSpeed
Left = 370
Top = 104
Width = 15
Height = 29
Cursor = crHandPoint
Hint = 'Скорость игры'
Alignment = taCenter
Caption = '1'
Font.Color = clRed
Font.Height = -24
Font.Name = 'MS Sans Serif'
Font.Style = [fsBold]
ShowHint = True
Transparent = True
```

Его же разумно использовать и для установки новых значений. Нажмёте левую кнопку – скорость возрастёт, правую – уменьшится:

```
//УСТАНОВИТЬ СКОРОСТЬ ИГРЫ
procedure Tform1.lblSpeedMouseDown(Sender: Tobject; Button:
TmouseButton;
  Shift: TshiftState; X, Y: Integer);
begin
  case Button of
    mbLeft:  //увеличить скорость
      if Speed< MAX_SPEED then inc(Speed);
    mbRight: //уменьшить скорость
      if Speed>1 then dec(Speed);
  end;

  Timer1.Interval:= 1000 div Speed;
  lblSpeed.Caption:= inttostr(Speed);
end; //lblSpeedMouseDown
```

Значение максимальной скорости игры задаётся *константой*:

```
  MAX_SPEED= 10;     //максимальная скорость игры
```

Её можно ещё увеличить, но и при такой скорости вам будет не до шуток!

Тетрис *a la Delphi*

Самое главное в любой программе это, безусловно, не оригинальная форма и красивые кнопки, а хороший алгоритм. Начнём его разработку с описания фигурок.

Их всего семь, состоят они из четырёх «квадратов» каждая и называются - **тетрамино**. Кстати говоря, в некоторых тетрисах используются фигурки и с другим числом клеток – 1, 2, 3, 5. Иногда дозволительно и самим создавать различные наборы фигурок в редакторе. Мы этого делать не станем, так как исследуем только классический *тетрис*. Общее количество фигурок в наборе задаётся *константой*:

```
  NUM_ELEMENTS= 7;  //число элементов в наборе
```

Если вы захотите добавить фигурок, то измените эту константу (или замените её переменной).

Введём тип **TTetra**, в котором перечислим все фигурки. Их названия условны и даны по аналогии с обозначением элементов пентамино. Вы можете назвать их мурка, васька и так далее.

```
//список фигурок:
type TTetra= (oTetra, iTetra, sTetra, zTetra, yTetra, fTetra,
lTetra);
```

Каждую фигурку можно повернуть вокруг оси на 90, 180, 270 градусов, то есть все фигурки имеют свой «фас» и «профиль». Всего фигурки могут занимать по 4 различных положения на плоскости (конечно, некоторые из них могут и совпадать), для описания которых введём тип **TRotate**:

```
//повороты элемента:
type TRotate=(R0, R90, R180, R270);
```

Любое тетрамино можно "втиснуть" в квадрат 4 х 4 клетки, поэтому для описания фигурок используем матрицу именно такого размера. Единицей обозначим в ней клетку, принадлежащую фигурке, нулём – клетку «фона». Итого получим последовательность из 16-ти единиц и нулей, которые логично представить в виде битов 16-разрядного слова.

Теперь в квадратиках 4 х 4 клетки нарисуем все фигурки во всех возможных положениях (можно обойтись только «неповёрнутой» фигуркой, но тогда придётся вычислять остальные три положения, что значительно сложнее, чем их прорисовка). Каждые 4 бита образуют полубайт, который легко вычисляется по рисунку. Всего фигурка описывается четырьмя полубайтами. Например, фигурка **oTetra** задаётся полубайтами: **0** (верхняя строка равна 0), **6** (вторая сверху строка равна 0+4+2+0), **6**, **0**. В итоге получаем 16-ричное число $0660. Точно так же описываем и все остальные фигурки. Для описания фигурок введём тип **TFigure**:

```
//описание фигурок:
type TFigure= array[TRotate] of word;
```

Все данные о фигурках набора будем хранить в *массиве* (Рис. П2.3):

```
 //описание набора фигур тетрамино:
 Element: array[TTetra] of TFigure= (($0660, $0660, $0660, $0660),
                                    ($0f00, $2222, $0f00, $2222),
                                    ($06c0, $4620, $06c0, $4620),
                                    ($0c60, $2640, $0c60, $2640),
                                    ($0e40, $2620, $04e0, $4640),
                                    ($08e0, $6440, $0e20, $2260),
                                    ($02e0, $4460, $0e80, $6220));
```

Для хранения информации о падающей фигурке введём новый *тип*:

```
//информация о падающей фигуре:
type TMoveFig= Record
  Fig: TTetra; //- вид фигурки
  Rot: TRotate;//- поворот
  //коорд. верхнего левого угла матрицы:
  CoordX: integer;
  CoordY: integer;
end;
```

В *переменные*:

```
 //текущая (падающая) и следующая фигуры:
CurrFigure, NextFigure: TMoveFig;
```

мы будем записывать вид фигурки (**oTetra..lTetra**), её поворот (**R0..R270**) и координаты верхнего левого угла матрицы. Этого вполне достаточно, чтобы однозначно определить любую фигурку на игровом поле.

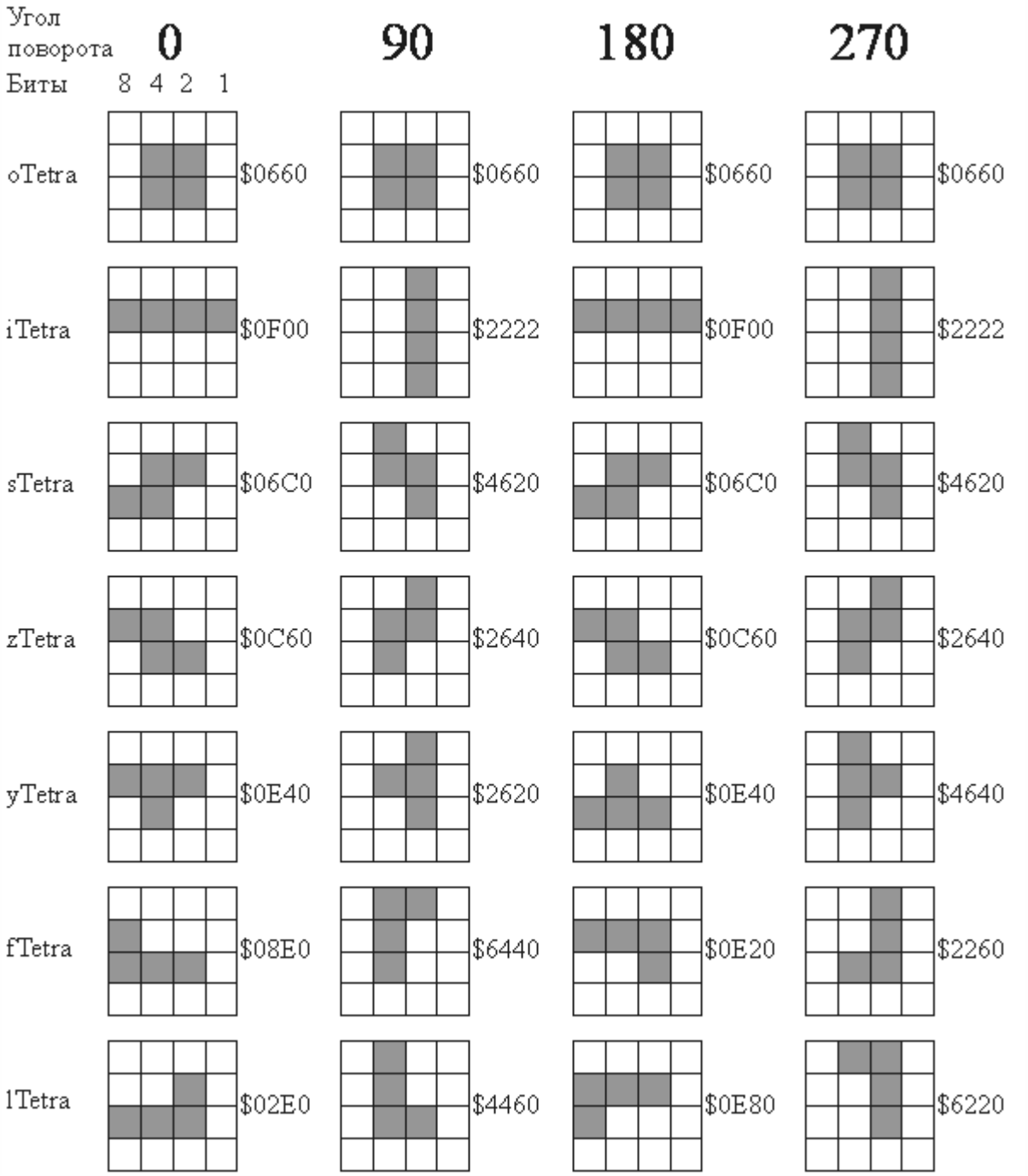


Рис. П2.3. Фигуры в разных положениях

Когда нам потребуется новая фигурка, мы передадим в процедуру **NewFigure** переменную типа *TMoveFig*, которая случайным образом выберет фигурку и её поворот:

```
//ВЫБРАТЬ НОВУЮ ФИГУРУ
procedure NewFigure(var mf: TMoveFig);
begin
  Randomize;
  //случайным образом выбираем фигуру:
```

```
  mf.Fig:= TTetra(Random(NUM_ELEMENTS));
  //и её поворот:
  mf.Rot:=  TRotate(Random(NUM_ROTATE));
  //задаём начальные координаты фигуры:
  mf.CoordX:= Random(POLE_WIDTH-3);
  mf.CoordY:= -3;
end; //NewFigure
```

Новая фигурка в других тетрисах обычно появляется в середине верхнего ряда, а мы, чтобы запутать противника, будем запускать её в произвольной позиции.

В маленькой сетке можно показать, какая фигурка «выпадет в осадок» после «приземления» падающей. По заданному виду фигурки и повороту находим слово, описывающее её в матрице. Определяем, установлен ли старший бит слова. Если установлен, то это клетка фигуры - заносим в массив **masNext** номер фигурки в наборе, иначе в **masNext** записываем –1, что означает пустую клетку. Затем сдвигаем все биты влево и снова проверяем старший бит. Так продолжаем до тех пор, пока не обработаем все 16 битов слова.

```
//ПОКАЗАТЬ СЛЕДУЮЩУЮ ФИГУРУ
procedure TForm1.ShowNextFig;
var
  n: word;
  i, j: integer;
begin
  //слово, описывающее фигурку:
  n:= Element[NextFigure.Fig, NextFigure.Rot];
  for j:= 0 to 3  do
    for i:= 0 to 3  do begin
      if n and $8000 <> 0 then //- клетка фигуры
        masNext[i,j]:= ord(NextFigure.Fig)
      else
        masNext[i,j]:= -1;     //- клетка "фона"
      n:=  n shl 1;
    end;
  dgNext.Invalidate;
end; //ShowNextFig
```

По этим данным выводим в сетку **dgNextDraw** фигурку в нужном положении. В нашей программе она всегда красная, но вы можете закрашивать каждый вид фигурок своим цветом:

```
//НАРИСОВАТЬ СЛЕДУЮЩУЮ ФИГУРКУ
procedure TForm1.dgNextDrawCell(Sender: TObject; ACol, ARow: Inte-
ger;
  Rect: TRect; State: TGridDrawState);
begin
```

```
  case masNext[ACol, ARow] of
    -1: dgNext.Canvas.Brush.Color:= clWhite;
    else dgNext.Canvas.Brush.Color:= clRed;
  end;
    {0: dgNext.Canvas.Brush.Color:= clRed;
     1: dgNext.Canvas.Brush.Color:= clGreen;
     2: dgNext.Canvas.Brush.Color:= clYellow;
     3: dgNext.Canvas.Brush.Color:= clBlack;
     4: dgNext.Canvas.Brush.Color:= clOlive;
     5: dgNext.Canvas.Brush.Color:= clNavy;
     6: dgNext.Canvas.Brush.Color:= clTeal;
    end;}
    dgNext.Canvas.FillRect(Rect);
end; //dgNextDrawCell
```

И вот процесс пошёл – фигурка выбрана, таймер включен. На сцене появляется падающая фигурка. Через заданный промежуток времени срабатывает таймер и сдвигает фигурку вниз на 1 клетку. Но всегда ли это возможно – ведь она могла уже упасть «на дно колодца» или «упереться» в другую фигурку, закончившую свой полёт в бездну раньше. Проверять фигурку на возможность сдвига будем в функции **TestNextPos**. Она возвращает *True*, если необходимое место для фигурки в новом положении есть, и *False*, если нет.

Прежде всего, нам нужно убрать все воспоминания о падающей фигурке из массива **masPole**, чтобы она не стала препятствием для самой себя. Затем попытаемся каждую клетку фигуры вывести на поле. Если она не выходит за границы поля (такая ситуация может возникнуть, если вы неудачно руководите полётом фигурки) и клетка поля пустая (**`masPole[]<> -1`**), то с этой клеткой фигуры всё нормально. Так проверяем все клетки фигурки. При благоприятном исходе проверки фигурке можно выписывать ордер на новое место жительства. В противном случае восстанавливаем фигурку на прежнем месте и возвращаем *False*. Особо надо рассмотреть случай, когда новая фигурка не может появиться на поле, так как это печальное событие предзнаменует окончание игры – всё поле забито бедными фигурками до отказа! Впрочем, любая игра рано или поздно заканчивается, но вы могли показать себя настоящим тетрисоводом и заработать хорошие очки. Тогда вас ждёт почётная обязанность увековечить своё имя среди лучших игроков.

```
//ПРОВЕРИТЬ, МОЖНО ЛИ ВЫВЕСТИ ПАДАЮЩУЮ ФИГУРУ В НОВОМ ПОЛОЖЕНИИ
function TestNextPos(dX, dY: integer; dRot: TRotate; var flg:
Boolean): Boolean;
//dX, dY - смещение фигурки от текущего положения,
//dRot - её новый поворот,
//flg = True, если новую фигурку нельзя вывести на поле
var
```

```
  n: word;
  i, j: integer;
  i1, j1: integer;
  x, y: integer;
begin
  Result:= True;
  flg:= False;
  //координаты падающей фигурки:
  x:= CurrFigure.CoordX;
  y:= CurrFigure.CoordY;
  //биты фигурки:
  n:= Element[CurrFigure.Fig, CurrFigure.Rot];
  //стереть фигуру в текущем положении:
  for j:= y to y+3  do
    for i:= x to x+3  do begin
      if n and $8000 <> 0 then
        masPole[i,j]:= -1;
      n:=  n shl 1;
    end;
  //проверить новое положение:
  n:= Element[CurrFigure.Fig, dRot];
  x:= x+ dX;
  y:= y+ dY;
  for j:= y to y+3  do
    for i:= x to x+3  do begin
      if n and $8000 <> 0 then //- клетка фигуры
        if (j> POLE_HEIGHT-1) or (i<0) or (i>POLE_WIDTH-1) or
           (masPole[i,j]<> -1) then begin //- не ложится!
          //восстановить фигуру в текущем положении:
          x:= CurrFigure.CoordX;
          y:= CurrFigure.CoordY;
          n:= Element[CurrFigure.Fig, CurrFigure.Rot];
          for j1:= y to y+3  do
            for i1:= x to x+3  do begin
              if n and $8000 <> 0 then
                masPole[i1,j1]:= ord(CurrFigure.Fig);
              n:=  n shl 1;
            end;
          //фигурку невозможно вывести в начальной позиции:
          if j= 0 then flg:= True;
          //фигурку невозможно вывести в следующей позиции:
          Result:= False;
          Exit
        end;
      n:=  n shl 1;
    end;
end; //TestNextPos
```

Пристроив - удобно или не очень - падающую фигурку, следующую фигурку переводим в разряд текущих и выбираем новую - для замены выбывшей. Тут же необходимо посмотреть, не образовались ли на поле горизон-

тальные ряды, целиком состоящие из клеток фигурок. По правилам игры такие ряды уничтожаются, остальные сдвигаются вниз, а вы получаете призовые очки!

```
//ОБРАБОТКА ПОЛНЫХ РЯДОВ
procedure TForm1.TestFullLine;
var
  i, j: integer;
  flg: boolean;
  //сдвинуть поле вниз
  procedure MovePole(n: word);
  var
    i, j: integer;
  begin
    for j:= n downto -3 do begin
      //жалобно пикнуть:
      MessageBeep($ffffff);
      for i:= 0 to POLE_WIDTH-1 do begin
        masPole[i,j]:= masPole[i,j-1];
      end;
    end;
  end;
begin
  for j:= 0 to POLE_HEIGHT-1 do begin
    flg:= true;
    for i:= 0 to POLE_WIDTH-1 do begin
      if masPole[i,j]= -1 then begin //- не полный ряд
        flg:= false;
        break
      end;
    end;
    if flg then begin//- полный ряд!
      MovePole(j);
      //добавить очки:
      score:= score+ (1 + Speed div 3)*5;
      lblScore.Caption:= inttostr(Score);
    end;
  end;
end; //TestFullLine
```

Фигурку, прошедшую испытание на живучесть, рисуем в её новом положении:

```
//ВЫВЕСТИ ФИГУРУ НА ПОЛЕ В ТЕКУЩЕЙ ПОЗИЦИИ
procedure TForm1.SetFigure;
var
  n: word;
  i, j: integer;
  x, y: integer;
begin
```

```
  //данные фигуры:
  n:= Element[CurrFigure.Fig, CurrFigure.Rot];
  x:= CurrFigure.CoordX;
  y:= CurrFigure.CoordY;
  //записать в массив:
  for j:= y to y+3  do
    for i:= x to x+3  do begin
      if n and $8000 <> 0 then
        masPole[i,j]:= ord(CurrFigure.Fig);
      n:=  n shl 1;
    end;
  //обновить поле:
  dgPole.Invalidate;
end; //SetFigure
//ПЕРЕМЕСТИТЬ ПАДАЮЩУЮ ФИГУРУ ВНИЗ НА 1 КЛЕТКУ
procedure TForm1.Timer1Timer(Sender: TObject);
var
  flgGameOver: Boolean;
begin
  if GameStatus<> Play then exit;
  application.ProcessMessages;
  //проверить, можно ли сдвинуть фигуру вниз:
  if TestNextPos(0,1, CurrFigure.Rot, flgGameOver)= False then
begin
    //новая фигура не может быть выведена на поле -
    if flgGameOver then begin //- игра закончена
      //остановить таймер:
      Timer1.Enabled:= false;
      //остановить игру:
      GameStatus:= Stop;
      //если показан хороший результат,
      //занести его в таблицу:
      if Score> top10[10].score then SetRecord
      else begin //иначе вывести сообщение:
        frmGameOver.Left:= form1.Left+ dgPole.Left +
          (dgPole.Width- frmGameOver.Width) div 2;
        frmGameOver.Top:= form1.Top+ dgPole.Top +
          (dgPole.Height- frmGameOver.Height) div 2;
        frmGameOver.ShowModal;
      end;
    end
    else begin //- текущая фигура имеет снизу препятствие
      //теперь текущей становится следующая фигура:
      CurrFigure:= NextFigure;
      //выбрать следующую фигуру со случайным поворотом:
      NewFigure(NextFigure);
      //показать следующую фигуру:
      ShowNextFig;
      //проверить, нет ли полных рядов:
      TestFullLine;
    end
  end
  else begin
```

```
    //нарисовать фигуру ниже:
    inc(CurrFigure.CoordY);
    SetFigure;
    //добавить очки:
    score:= score+ 1 + Speed div 3;
    lblScore.Caption:= inttostr(Score);
  end;
end; //Timer1Timer
```

Сообщение об окончании игры можно выдать разными способами, но особенно огорчительно она будет выглядеть на художественно оформленной табличке (Рис. П2.4).



Рис. П2.4. Тут и тетрису конец!

Заведём для неё новую форму - **frmGameOver**:

```
  AutoSize = True
  BorderStyle = bsNone
```

Разместим на ней компонент **Image1**:

```
    Left = 0
    Top = 0
    Cursor = crHandPoint
    AutoSize = True
```

И загрузим в него нужную картинку.

Кроме огорчений, эта форма ничего делать не будет:

```
unit GameOverUnit;
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
Dialogs, ExtCtrls;
type
  TfrmGameOver = class(TForm)
    Image1: TImage;
    procedure Image1DblClick(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;
var
  frmGameOver: TfrmGameOver;
```

```
implementation
{$R *.DFM}
procedure TfrmGameOver.Image1DblClick(Sender: TObject);
begin
  Close
end;
end.
```

Щёлкните на ней дважды – она и закроется.

Более приятные хлопоты вас ждут, если вы насобираете много очков. Чтобы слава нашла именно вас, нужно отметиться в ещё одной форме – **frmSetName** (Рис. П2.5).



Рис. П2.5. Поздравляем!

```
AutoSize = True
BorderStyle = bsNone
```

Далее двигаемся по проторённой дорожке: в **Image1** загружаем заранее подготовленную картинку:

```
    Align = alClient
    AutoSize = True
```

загружаем заранее подготовленную картинку, делаем поясняющие надписи в **Label1:**

```
    Left = 44
    Top = 8
    Width = 161
    Height = 16
    Caption = 'Отличный результат!'
    Font.Color = clRed
    Font.Height = -13
    Font.Name = 'MS Sans Serif'
    Font.Style = [fsBold]
    Transparent = True
```

и **Label2:**

```
Left = 8
Top = 36
Width = 140
Height = 16
Caption = 'Введите ваше имя:'
Font.Color = clWindowText
Font.Height = -13
Font.Name = 'MS Sans Serif'
Font.Style = [fsBold]
Transparent = True
```

Свои анкетные данные можно ввести в поле компонента TEdit **txtName**:

```
Left = 8
Top = 64
Width = 169
Height = 24
Color = clTeal
Font.Color = clWhite
Font.Height = -13
Font.Name = 'MS Sans Serif'
Font.Style = [fsBold]
MaxLength = 14
TabOrder = 0
```

Для разумного ограничения графомании длина строки лимитирована 14-ю символами.

Свернуть форму можно кнопкой **BitBtn1**:

```
Left = 184
Top = 64
Width = 55
Height = 25
Cursor = crHandPoint
Kind = bkOK
TabOrder = 1
```

```
unit SetNameUnit;
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
Dialogs,  StdCtrls, Buttons, ExtCtrls, jpeg;
type
  TfrmSetName = class(TForm)
    txtName: TEdit;
    Image1: TImage;
    BitBtn1: TBitBtn;
    Label1: TLabel;
    Label2: TLabel;
    procedure BitBtn1Click(Sender: TObject);
```

```
    procedure FormClose(Sender: TObject; var Action: TCloseAc-
tion);
  private
    { Private declarations }
  public
    { Public declarations }
  end;
var
  frmSetName: TfrmSetName;
implementation
uses tetris;
{$R *.DFM}
procedure TfrmSetName.BitBtn1Click(Sender: TObject);
begin
  close
end;
procedure TfrmSetName.FormClose(Sender: TObject; var Action:
TCloseAction);
begin
  PlayerName:= txtName.text;
  if PlayerName='' then PlayerName:= 'Tetris';
  PlayerName:=  PlayerName + '..........';
  PlayerName:= copy(PlayerName, 1,14)
end;
end.
```

И вот вы ввели свою фамилию, теперь нужно найти ей достойное место среди других отличившихся по части складирования фигурок:

```
//ЗАНЕСТИ РЕЗУЛЬТАТ В СПИСОК
procedure TForm1.SetRecord;
var
  i, n: integer;
begin
  n:= 0;
  //найти своё место в списке:
  for i:= 1 to 10 do
    if Score> top10[i].score then begin
      n:= i; break
    end;
  //записать результат на место n:
  if n>0 then begin
    frmSetName.ShowModal;
    //переместить список вниз:
    for i:= 10 downto n do
      top10[i]:= top10[i-1];
    //вставить новые данные:
    top10[n].score:= Score;
    top10[n].name:= PlayerName;
    ShowRecords;
  end;
end; //SetRecord
```

Все данные о лучших игроках хранятся в *списке*:

```
  //десятка лучших игроков:
  top10: array[0..10] of TTop10;
```

Он содержит имя игрока и количество набранных очков. Вы можете добавить и другие данные. Что касается очков, то в программе они начисляются щедро, но никто не мешает вам повредничать.

```
//информация об игроке:
type TTop10= record
  name: string;
  score: integer;
end;
//имя последнего рекордсмена:
PlayerName: string;
```

В процедуре **SetFigure** в массив **masPole** заносится номер фигурки, в зависимости от которого клетки фигурки на поле закрашиваются своим цветом. Пустые клетки - чёрные, все остальные вы можете выбрать по своему вкусу. Функция **DrawEdge** придаёт клеткам объёмный вид. Если её убрать, то они станут плоскими. Для ещё большего украшательства можно копировать в клетки поля рисунки, как, например, это сделано в программе *Mind Game*.

```
//НАРИСОВАТЬ КЛЕТКУ ПОЛЯ
procedure TForm1.dgPoleDrawCell(Sender: TObject; ACol, ARow: Inte-
ger;
  Rect: TRect; State: TGridDrawState);
begin
  case masPole[ACol, ARow] of
    -1: dgPole.Canvas.Brush.Color:= clBlack;
     0: dgPole.Canvas.Brush.Color:= rgb(255,0,0);//clRed;
     1: dgPole.Canvas.Brush.Color:= rgb(0,255,0);//clGreen;
     2: dgPole.Canvas.Brush.Color:= rgb(0,0,255);//clYellow;
     3: dgPole.Canvas.Brush.Color:= rgb(255,255,0);//clWhite;
     4: dgPole.Canvas.Brush.Color:= rgb(255,0,255);//clOlive;
     5: dgPole.Canvas.Brush.Color:= rgb(0,255,255);//clNavy;
     6: dgPole.Canvas.Brush.Color:= rgb(255,179,54);//clTeal;
    end;
    dgPole.Canvas.FillRect(Rect);
    if masPole[ACol, ARow]<>-1 then
      DrawEdge(dgPole.Canvas.Handle, Rect, EDGE_RAISED, BF_RECT)
end; //dgPoleDrawCell
```

Чтобы навсегда покончить с тетрисом, просто нажмите кнопку **ВЫХОД**:

```
//ЗАКРЫТЬ ПРОГРАММУ
procedure TForm1.imgExitMouseDown(Sender: TObject; Button: TMouse-
```

```
Button;
  Shift: TShiftState; X, Y: Integer);
begin
  imgExit.Visible:= False;
  imgExit2.Visible:= True;
  application.ProcessMessages;
  sleep(444);
  imgExit.Visible:= True;
  imgExit2.Visible:= False;
  close
end; //imgExitMouseDown
```

Очень полезной может оказаться кнопка **ПАУЗА**. Смело нажимайте её, когда голова пойдёт кругом от коловорота падающих фигур.

```
//ПАУЗА В ИГРЕ
procedure TForm1.ImgPauseMouseDown(Sender: TObject; Button:
TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  if GameStatus<> Play then exit;
  GameStatus:= Pause;
  imgPause2.Visible:= True;
  imgPause.Visible:= False
end; //ImgPauseMouseDown
```

Нажмите эту кнопку ещё раз - и снова вы окунётесь в круговерть событий.

```
//ВЕРНУТЬСЯ К ИГРЕ
procedure TForm1.ImgPause2MouseDown(Sender: TObject; Button:
TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  if GameStatus<> Pause then exit;
  GameStatus:= Play;
  imgPause.Visible:= True;
  imgPause2.Visible:= False
end; //ImgPause2MouseDown
```

Чтобы закончить неудачную игру, нажмите кнопку **ФИНИШ**:

```
//ОСТАНОВИТЬ ИГРУ
procedure TForm1.imgFinishMouseDown(Sender: TObject; Button:
TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  if GameStatus<> Play then exit;
  imgFinish.Visible:= False;
  imgFinish2.Visible:= True;
  application.ProcessMessages;
  sleep(444);
```

```
  imgFinish.Visible:= True;
  imgFinish2.Visible:= False;
  GameStatus:= Stop;
  //выключить таймер:
  Timer1.Enabled:= False;
end; //imgFinishMouseDown
```

Осталась последняя кнопка - **Top10**. Нажав её, вы увидите десятку лучших игроков:

```
//ПОКАЗАТЬ ТАБЛИЦУ РЕКОРДОВ
procedure TForm1.imgRecordsMouseDown(Sender: TObject; Button:
TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  if GameStatus= Play then exit;
  imgRecords.Visible:= False;
  imgRecords2.Visible:= True;
  ShowRecords;
end; //imgRecordsMouseDown
procedure TForm1.ShowRecords;
var
  i: integer;
  s: string;
begin
  lstName.Clear; lstScore.Clear;
  lstName.Items.add(''); lstScore.Items.add('');
  for i:= 1 to 10 do begin
    s:= inttostr(i)+'.'+top10[i].name;
    if i<10 then s:= ' '+ s;
    lstName.Items.add(s);
    s:= '     '+ inttostr(top10[i].score);
    s:= Copy(s, Length(s)-5,6);
    lstScore.Items.add(s);
  end;
  lstName.Visible:= True;
  lstScore.Visible:= True;
end; //ShowRecords
```

Список *исчезнет*, как только вы отпустите кнопку:

```
procedure TForm1.imgRecordsMouseUp(Sender: TObject; Button:
TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  imgRecords.Visible:= True;
  imgRecords2.Visible:= False;
  HideRecords;
end;//imgRecordsMouseUp
procedure TForm1.HideRecords;
begin
```

```
  lstName.Visible:= False;
  lstScore.Visible:= False;
end; //HideRecords
```

Для «показухи» лучше бы использовать отдельную форму, но мы поступим скромнее – просто разместим на форме два компонента **ListBox** – один для фамилий (**lstName**):

```
Left = 32
Top = 112
Width = 149
Height = 189
Color = clSilver
Font.Charset = RUSSIAN_CHARSET
Font.Color = clWindowText
Font.Height = -13
Font.Name = 'Courier New'
Font.Style = [fsBold]
ItemHeight = 16
Visible = False
```

Другой (**lstScore**) – для результатов:

```
Left = 180
Top = 112
Width = 61
Height = 189
Color = clBlack
Font.Charset = RUSSIAN_CHARSET
Font.Color = clWhite
Font.Height = -13
Font.Name = 'Courier New'
Font.Style = [fsBold]
ItemHeight = 16
Visible = False
```

Их можно и объединить, но с двумя проще работать.

После внесения вашего имени в список рекордсменов он появится на экране. Оценив себя по достоинству, щёлкните на любом из списков (установите для **lstScore OnClick**=lstNameClick), чтобы они не заслоняли поле.

```
//УБРАТЬ ТАБЛИЦУ РЕКОРДОВ
procedure TForm1.lstNameClick(Sender: TObject);
begin
  HideRecords;
end; //lstNameClick
```

Хитрая механика, или Алгоритм – душа программы

Вторая составная часть основного алгоритма программы это, безусловно, **управление падающими фигурками**. В большинстве тетрисов перемещение фигурок осуществляется с клавиатуры (чаще всего используются клавиши со стрелками: стрелка влево – перемещение фигурки влево, стрелка вправо – вправо, стрелка вниз – сброс фигурки вниз, стрелка вверх – поворот фигурки по часовой стрелке) и дублируется кнопками на форме, которые нужно нажимать мышкой. От второго способа мы сразу откажемся из-за его медлительности - при высокой скорости падения фигурок вы просто не успеете передвигать курсор на нужную кнопку. Контролировать полёты клавишами значительно удобнее и привычнее для многих игроков, поэтому мы с них и начнём, а потом подумаем, как привлечь мышь к делу наставления фигурок на путь истинный.

Установив значение свойства формы **KeyPreview** в True, мы получили возможность обрабатывать нажатие клавиш на клавиатуре в процедуре **FormKeyDown**. Если нажата одна из клавиш со стрелками, то реагируем должным образом, остальные клавиши напрочь игнорируем:

```
//УПРАВЛЕНИЕ ПАДАЮЩЕЙ ФИГУРКОЙ С КЛАВИАТУРЫ
procedure TForm1.FormKeyDown(Sender: TObject; var Key: Word;
Shift: TShiftState);
begin
  if GameStatus<> Play then exit;

  case Key of
    //клавиша "влево":
    VK_LEFT: FigLeft;
    //клавиша "вправо":
    VK_RIGHT: FigRight;
    //клавиша "вверх" - поворот:
    VK_UP: FigUp;
    //клавиша "вниз":
    VK_DOWN: FigDown;
    //неверная клавиша:
    else Beep;
  end;
end; //FormKeyDown
```

Пусть нажата клавиша «стрелка влево». Проверяем, осуществим ли такой манёвр, и, получив от функции **TestNextPos** «добро», сдвигаем фигурку влево на 1 клетку и выводим её на поле в новом положении:

```
//СДВИНУТЬ ФИГУРКУ ВЛЕВО
procedure TForm1.FigLeft;
var flg: boolean;
```

```
begin
  if TestNextPos(-1, 0, CurrFigure.Rot, flg) then begin
    dec(CurrFigure.CoordX);
    SetFigure;
  end;
end; //FigLeft
```

Те же манипуляции проводим и при давлении на клавишу *стрелка вправо*:

```
//СДВИНУТЬ ФИГУРКУ ВПРАВО
procedure TForm1.FigRight;
var flg: boolean;
begin
  if TestNextPos(1, 0, CurrFigure.Rot, flg) then begin
    inc(CurrFigure.CoordX);
    SetFigure;
  end;
end; //FigRight
```

Клавиша *стрелка вниз* сбрасывает фигурку на самое дно колодца. Пике получается весьма стремительное, так что позаботьтесь о точной наводке на цель, иначе фигурка может «прилипнуть» в самом неподходящем месте. Зато за риск полагается приз. Не шампанское, конечно, а дополнительные очки (размер надбавки скорректируйте сами).

```
//СБРОСИТЬ ФИГУРКУ ВНИЗ
procedure TForm1.FigDown;
var flg: boolean;
begin
  if TestNextPos(0, 1, CurrFigure.Rot, flg) then begin
    inc(CurrFigure.CoordY);
    SetFigure;
    //добавить очки:
    score:= score+ (1 + Speed div 3)*2;
    lblScore.Caption:= inttostr(Score);
  end;
end; //FigDown
```

Самый большой ералаш вызовет клавиша *стрелка вверх*. Во-первых, нужно перейти к следующему повороту и проверить новое положение фигурки. Если всё нормально, просто выводим фигурку на поле в новом ракурсе. Некоторые неудобства вращению создают боковые стенки (границы) поля. Если фигурка находится у стенки в «вертикальном» положении, то при переходе в «горизонтальное» она будет стремиться «проломить» стенку. Мы, естественно, обязаны пресечь эти безобразные поползновения какой-то там фигурки на целостность нашего поля. Сему есть два противодействия: либо запретить всякое верчение у стенок (в недемократических тетрисах так это всё и устроено), либо сместить фигурку так, чтобы она

смогла-таки провернуться. Как гуманисты мы не можем пройти мимо второй возможности, но тогда нам придётся добавить код в «конституцию» и смириться с тем, что фигурки в приграничных областях иногда будут себя вести не совсем адекватно:

```
//ПОВЕРНУТЬ ФИГУРКУ ПО ЧАСОВОЙ СТРЕЛКЕ
procedure TForm1.FigUp;
var
  flg: boolean;
  r: TRotate;
  n: integer;
begin
  //текущий поворот фигурки:
  r:= CurrFigure.Rot;
  //новый поворот фигурки:
  if r= R270 then r:= R0
  else  r:=  succ(CurrFigure.Rot);
  //повернуть:
  if TestNextPos(0, 0, r, flg) then begin
    CurrFigure.Rot:= r;
    SetFigure;
    sleep(1);
    exit;
  end;

  //если повороту мешает стенка поля, сдвинуть фигурку:
  if CurrFigure.CoordX< 0 then //- у левой стенки
    n:= -CurrFigure.CoordX
  else if CurrFigure.CoordX=POLE_WIDTH-3 then //- у правой стенки
    n:= -1
  else
    n:= 0;

  if TestNextPos(n, 0, r, flg) then begin
    CurrFigure.CoordX:= CurrFigure.CoordX+n;
    CurrFigure.Rot:= r;
    SetFigure;
    sleep(1);
  end;
end; //FigUp
```

С клавиатурой покончено, вернёмся к нашей мышке. Если бы у неё было 4 кнопки, то их можно было бы задействовать аналогично клавишам, но их только две (или три, что тоже мало). Поэтому **правой** кнопкой будем фигурку вертеть:

```
//ПРАВАЯ КЛАВИША МЫШИ - ПОВОРОТ ПАДАЮЩЕЙ ФИГУРЫ
procedure TForm1.dgPoleMouseDown(Sender: TObject; Button: TMouse-
Button;
  Shift: TShiftState; X, Y: Integer);
```

```
begin
  if GameStatus<> Play then exit;

  //запомнить координаты мыши:
  dgPole.MouseToCell(x,y,cellmouse.x, cellmouse.y);
  //повернуть фигурку:
  if ssRight in shift then begin
    FigUp;
    timer1.Enabled:= False
  end;
end; //dgPoleMouseDown
```

Не забудем запомнить координаты клетки поля с курсором в *переменной*:

```
  //координаты мышки:
  cellmouse: TPoint;
```

Если нажата **левая кнопка** мыши, то это всё. Если правая, то поворачиваем фигурку и **останавливаем время**. Это значит, что, пока мы держим мышку за правую кнопку, фигурка падать не будет. Альтернативный вариант – нажать кнопку **ПАУЗА** – менее динамичен, к тому же он вообще тормозит игру, а мышкой мы можем двигать фигурку по полю. Безусловно, такая возможность тянуть резину сильно облегчает игру в цейтноте, поэтому избавьтесь от строки `timer1.Enabled:= False`, чтобы не искушать себя поблажками и гандикапами.

Подхватив фигурку мышкой, вы можете передвигать её влево-вправо вслед за движением мышки, сбрасывать вниз, если мышка перемещается вниз (будьте внимательны: курсор постоянно должен находиться в одном и том же вертикальном ряду сетки, иначе фигурка сдвинется в сторону), и вращать, нажимая в любое время правую кнопку.

```
//ДВИГАТЬ ФИГУРКУ МЫШКОЙ
procedure TForm1.dgPoleMouseMove(Sender: TObject; Shift:
TShiftState; X,Y: Integer);
var
  ACol,ARow: integer;
begin
  if GameStatus<> Play then exit;

  //координаты мыши:
  dgPole.MouseToCell(x,y,ACol,ARow);

  //если нажата кнопка мыши, то переместить фигурку
  //в сторону движения мышки:
  if (ssLeft in shift) or (ssRight in shift) then begin
    if cellmouse.x>ACol then FigLeft;
    if cellmouse.x<ACol then FigRight;
    //if cellmouse.y>ARow then FigUp;
```

```
    if cellmouse.y< ARow then FigDown;
  end;
  //if ssRight in shift then FigUp;

  //запомнить новые координаты мыши:
  cellmouse.x:= ACol; cellmouse.y:= ARow;
end; //dgPoleMouseMove
```

Если при этом нажата **левая кнопка**, то фигурка будет продолжать падение с заданной скоростью, а вот **правая кнопка** лишает её свободы перемещения – она будет слепо следовать за мышкой (но только не вверх!). «Падучесть» вернётся к фигурке, как только вы отпустите кнопку:

```
//ОТПУСТИТЬ КНОПКУ МЫШИ
procedure TForm1.dgPoleMouseUp(Sender: TObject; Button: TMouseBut-
ton;
  Shift: TShiftState; X, Y: Integer);
begin
 timer1.Enabled:= True;
end; //dgPoleMouseUp
```

Этот необычный способ управления фигурками в тетрисе никем, кажется, не был реализован. К нему нужно привыкнуть, но он значительно удобнее виртуальных кнопок. Что нужно помнить, поступая таким образом, - нажимать кнопку мыши нужно только на игровом поле (при движении мышка не должна покидать границы поля, иначе вы утратите контроль над падающей фигуркой). Бессмысленно гнать мышку, если вам нужно быстро переместить фигурку, всё равно программа не успеет обработать все промежуточные положения курсора на поле. Мораль: нужно быстро думать, а не шустро мышку катать.

Дважды кликнув на форме, можно убрать её в дальний уголок стола:

```
//СВЕРНУТЬ ФОРМУ
procedure TForm1.Image1DblClick(Sender: TObject);
begin
  Form1.WindowState:=wsMinimized;
end; //Image1DblClick
```

Кому это надо, никто не знает, да мало ли…

«Киса и Ося здесь были», или «Забьём Мике баки»

Мы хотим всем рекордам
Наши гордые дать имена!

Олимпийская песня

Список лучших результатов можно хранить в обычном текстовом файле. Работать с ним достаточно просто, поэтому во многих наших программах используются именно такие файлы.

Но *Delphi* предоставляет и другую возможность настраивать параметры при запуске программы и сохранять настройки пользователя при выходе из неё – это **инициализационные файлы** (они имеют расширение **INI** и называются также **ini-файлами**). Собственно, это те же текстовые файлы (из этого следует, что их можно создавать и править в любом текстовом редакторе), но специальным образом структурированные.

Все данные в **ini-файле** разбиты на **разделы** (секции), названия которых заключены в квадратные скобки. За названием следуют строки, содержащие необходимую информацию. В начале каждой строки записывается имя (идентификатор) параметра, затем знак равенства «=» и, наконец, его значение строкового, целого или логического типов.

Например, таблица рекордов нашей программы в файле **records.ini** выглядит в редакторе как обычный текст:

```
[NAME]
name1= Tetris1
name2= Tetris2
name3= Tetris3
name4=Tetris4
name5=Tetris5
name6= Tetris6
name7= Tetris7
name8= Tetris8
name9= Tetris9
name10=Tetris10

[SCORE]
res1=4777
res2=4145
res3=3596
res4=2838
res5=2454
res6=2250
res7=1664
res8=1468
```

```
res9=1466
res10=1385
```

Вся информация разбита на два раздела: **[NAME]**, в котором хранятся фамилии игроков (идентификаторы **name1.. name10;** настоящие имена заменены здесь псевдонимами **Tetris1.. Tetris10**), и **[SCORE]** – с их результатами.

Для работы с файлами инициализации в разделе **uses** того модуля, в котором выполняются операции с этими файлами, нужно подключить модуль **IniFiles**:

```
unit tetris;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
Dialogs,  ExtCtrls, Grids, Buttons, StdCtrls, IniFiles;
```

Загружать список рекордов удобно при создании формы. Соответствующую процедуру мы уже рассматривали раньше, поэтому дальше она приведена в сокращённом виде:

```
//СОЗДАТЬ ФОРМУ
procedure TForm1.FormCreate(Sender: TObject);
var
  i, j: integer;
  appPath, s: string;
  spisok: TIniFile;
begin
      . . . . .
  //загрузить список лучших игроков:
  appPath:= ExtractfilePath(Application.ExeName);
  spisok:= TIniFile.Create(appPath+'Records.ini');

  for i:= 1 to 10 do begin
    s:= 'name'+inttostr(i);
    top10[i].name:= spisok.ReadString('NAME', s, 'Tetris');
    s:= 'res'+ inttostr(i);
    top10[i].score:= spisok.ReadInteger('SCORE', s, 0);
  end;

  spisok.Free;
end; //FormCreate
```

Все действия по считыванию информации из **ini-файла** практически одинаковы, так что с небольшими изменениями вы можете использовать этот код и в других программах.

Прежде всего, нужно объявить переменную типа *TIniFile* (у нас она называется **spisok** – только для пояснения её назначения). Далее создаём объект типа *TIniFile*, используя метод *Create (const FileName: string).* В результате получаем экземпляр класса *TIniFile*, связанный с инициализационным файлом, имя которого задаётся параметром **FileName**. Если не указать полный путь к файлу, то он ищется в каталоге *Windows*. Удобнее хранить **ini-файл** вместе с выполняемым файлом, поэтому мы передаём полный путь к файлу **Records.ini**.

Чтение данных из **ini-файла** выполняется с помощью *методов*:

```
function ReadString(const Section, Ident, Default: string): string;

function ReadBool (const Section, Ident: string; Default: Boolean): Boolean;


function ReadInteger(const Section, Ident: string; Default: Longint): Longint;

function ReadDate (const Section, Ident: string; Default: TDateTime): TDateTime;

function ReadDateTime (const Section, Ident: string; Default: TDateTime): TDateTime;

function ReadTime (const Section, Ident: string; Default: TDateTime): TDateTime;

function ReadFloat (const Section, Ident: string; Default: Double): Double;
```

- в зависимости от типа значения считываемого параметра. Например, фамилии игроков считываем в цикле методом `ReadString('NAME', s, 'Tetris');`

Здесь:

- **'NAME'** – название раздела,
- **s** – идентификатор параметра,
- 'Tetris' – возвращаемое значение по умолчанию, если указанный идентификатор не найден.

Подобным же образом загружаем из файла очки игроков из раздела **'SCORE'** - **`ReadInteger('SCORE', s, 0);`** В данном случае возвращаемое значение будет равно нулю, если идентификатор **s** отсутствует.

Последнее действие с инициализационным файлом – освобождение переменной: **spisok.Free;**

Таблицу рекордов можно в любое время увидеть на экране, нажав кнопку **Top10** (Рис. П2.6).



Рис. П2.6. Таблица рекордов

При выходе из программы нужно сохранить в файле все изменения. Делается это почти так же, как и при его загрузке, только используются методы:

```
procedure WriteString(const Section, Ident, Value: string);

procedure WriteBool(const Section, Ident: string; Value: Boolean);

procedure WriteInteger(const Section, Ident: string; Value:
Longint);

procedure WriteFloat(const Section, Ident: string; Value: Double);

procedure WriteDate(const Section, Ident: string; Value:
TDateTime);

procedure WriteDateTime(const Section, Ident: string; Value:
TDateTime);

procedure WriteTime(const Section, Ident: string; Value:
TDateTime);
```

```
//ЗАКРЫТЬ ФОРМУ
procedure TForm1.FormClose(Sender: TObject; var Action: TCloseAc-
tion);
var
  appPath, s: string;
```

```
  spisok: TIniFile;
  i: integer;
begin
  //записать список лучших игроков:
  appPath:= ExtractfilePath(Application.ExeName);
  spisok:= TIniFile.Create(appPath+'Records.ini');
  for i:= 1 to 10 do begin
    s:= 'name'+inttostr(i);
    spisok.WriteString('NAME', s, top10[i].name);
    s:= 'res'+ inttostr(i);
    spisok.WriteInteger('SCORE', s, top10[i].score);
  end;

  spisok.Free;
end; //FormClose
```

Следует отметить очень полезную особенность этих методов: если указанный раздел или идентификатор отсутствуют, то они автоматически создаются. Таким образом, вам не нужно специально добавлять в инициализационный файл новые разделы и идентификаторы. А для их **удаления** служат, соответственно, *методы*:

```
procedure EraseSection(const Section: string);

procedure DeleteKey(const Section, Ident: string);
```

Тут, как говорится, и сказочке конец. Над чем вам можно поработать? Вынесите список рекордов на отдельную форму, добавьте к программе помощь, поэкспериментируйте с наборами фигурок (например, можно добавить фигурки из двух клеток – димино, из трёх клеток – тримино - и из пяти клеток – пентамино (конечно, исключая **I-пентамино**, которое невозможно вписать в матрицу 4 х 4 клетки). В игре *Тетрипок* (смотри последний скриншот в начале главы) имеется такая разновидность тетриса: фигурки одного цвета беспрепятственно проходят друг через друга. Ценность этой новации не очень велика, но можете ей озадачиться.

Родословная тетриса

Нет никаких сомнений в том, что тетрис – прямой потомок тетрамино. **Тетрамино** – это головоломка, в которой требуется из набора всех (или только некоторых) элементов, состоящих из 4-х квадратиков, сложить заданную фигуру.

Чтобы из этой головоломки получилась игра, нужно добавить **неопределённость**. И автор тетриса Пажитнов сделал это удивительно просто: элементов тетрамино здесь бесконечное множество, но последовательность

их появления совершенно случайна. Игрок не может заранее распределить их на поле нужным образом, ему остаётся только располагать падающие фигурки как можно более компактно. К тому же процесс сборки полных линий проходит в условиях недостатка времени и динамически изменяющейся ситуации на игровом поле.

Таким образом, тетрис возник из тетрамино, но не может быть однозначно сведён к нему, так как в игре имеет значение и **последовательность** выпадения фигурок, тогда как в тетрамино мы можем выбирать и укладывать элементы в произвольном порядке. Это означает, что из тетриса можно "сгенерировать" **новую** головоломку, в которой важен и порядок укладки элементов в фигуру.

Как же следует в него играть, чтобы продержаться как можно дольше (выиграть, естественно, в такой игре нельзя)? Очевидно, что верхний срез фигурок желательно выстраивать как можно более ровным, чтобы иметь больше возможностей пристроить следующие фигурки, а, значит, и составить полные ряды. Если вы сможете сформулировать ещё несколько правил ведения игры, то почему бы вам не научить программу самостоятельно складывать фигурки в «стопку»? В конце концов, неважно, кто реализует ваши правила, человек или машина, лишь бы правила были удачными. Например, в головоломке *Японский кроссворд* единственное правило позволяет решать *любые* задачи, но там отсутствует ситуация неопределённости, как в тетрисе.

Научив машину уму-разуму, посоревнуйтесь с ней сами – часто ученик превосходит учителя, тем паче это относится к такому «вундеркинду», как компьютер. Установите на форме два стакана (колодца, игровых поля) и сбрасывайте в каждый из них одни и те же фигурки (синхронно и синфазно), но пусть их движением в одном стакане управляет компьютер (вы можете заменить его и другим гуманоидом), а во втором – вы сами. Сможете ли вы победить своё творение или вас ожидает полный франкенштейн? Но, проиграв, всё же не огорчайтесь – ещё много-много раз ваши программы обставят вас в два счёта, а вам как хорошему родителю останется только гордиться своим детищем! Воспитайте в себе виртуального родителя, глядишь – это и пригодится вам в реальной жизни…

Итак, что мы имеем? Из игры мы сподобились смастерить головоломку, а из головоломки – игру. С этим мы разобрались, но можно продвинуться и дальше – из одной игры сделать другую (этот абзац – чистый анахронизм, но он рисует перспективы!). Например, известная игра *Цветные линии* (*Color Lines*) явно берёт своё начало от ещё более известной игры гомоку

(крестики-нолики, пять в ряд), в которой нужно выстроить в непрерывный ряд пять своих фишек. Но если гомоку – игра для двоих, то в *Цветные линии* играют в одиночку. (А если всё-таки вдвоём? Например, один игрок собирает шарики, а второй – кубики. Тут есть над чем поразмыслить!) Кстати говоря, аналогичным образом можно преобразовать и другие игры в некое подобие *Цветных линий*. Если мы когда-нибудь доберёмся до *Цветных линий*, то обязательно сделаем шахматно-шашечные версии этой игры. А сейчас, как говаривал Огурцов, благодарю за внимание!



Исходный код программы находится в папке **Tetris**.

Художественные п(р)омыслы

Вернёмся к нашим баранам.

Рабле. *Гаргантюа и Пантагрюэль*

И крашу, крашу я заборы,
чтоб тунеядцем не прослыть.

Мультфильм
Вовка в Тридевятом царстве

Применим наши знания по обработке форм к панелям (и другим оконным компонентам). Чтобы не писать весь код заново, воспользуемся проектом из предыдущей главы.

Добавим ещё три кнопки: **sbtStar8, sbtStar8A, sbtStar8B** и обработчики для них:

```
//звезда 8
procedure TForm1.sbtStar8Click(Sender: TObject);
begin
  //за прозрачный цвет принимаем цвет точки в левом нижнем углу
  //картинки:
  TransColor:=
Image1.Picture.Bitmap.Canvas.Pixels[0,Image1.Picture.Bitmap.height
-1];

  //создать область из непрозрачных точек картинки:
  rgn:= fcCreateRegionFromBitmap(Image1.Picture.Bitmap,
TransColor);
```

```
  //задать форму окна по области rgn:
  SetWindowRgn(Panel1.Handle, rgn, True);
end;

//звезда 8A
procedure TForm1.sbtStar8AClick(Sender: TObject);
begin
  TransColor:=
Image2.Picture.Bitmap.Canvas.Pixels[0,Image2.Picture.Bitmap.height
-1];

  //создать область из непрозрачных точек картинки:
  rgn:= fcCreateRegionFromBitmap(Image2.Picture.Bitmap, TransCo-
lor);

  //задать форму окна по области rgn:
  SetWindowRgn(Panel2.Handle, rgn, True);
end;

//звезда 8B
procedure TForm1.sbtStar8BClick(Sender: TObject);
begin
  TransColor:=
Image3.Picture.Bitmap.Canvas.Pixels[0,Image3.Picture.Bitmap.height
-1];

  //создать область из непрозрачных точек картинки:
  rgn:= fcCreateRegionFromBitmap(Image3.Picture.Bitmap, TransCo-
lor);

  //задать форму окна по области rgn:
  SetWindowRgn(Panel3.Handle, rgn, True);
end;
```

Объявление переменной **TransColor** перенесём в раздел **implementation**, чтобы не переписывать 3 раза. Также нам потребуется константа **BitMask** для функции **fcCreateRegionFromBitmap**, которую мы просто позаимствуем из тетриса.

```
implementation

const
  BitMask: array[0..7] of byte = (128, 64, 32, 16, 8, 4, 2, 1);
var
  TransColor: TColor;

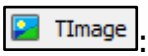
function fcCreateRegionFromBitmap(ABitmap: TBitmap; TransColor:
TColor): HRgn;

…
```

Для разнообразия добавим на форму ещё две панели: **Panel2** красного цвета и **Panel3** – чёрного. Для **Panel3** зададим свойства (Рис. П2.7).



Рис. П2.7. Панельные свойства

Осталось положить на каждую панель по компоненту **TImage** 

: **Image1**, **Image2** и **Image3**. Загрузите в них какие-нибудь картинки на одноцветном фоне – и интерфейс программы готов (Рис. П2.8).

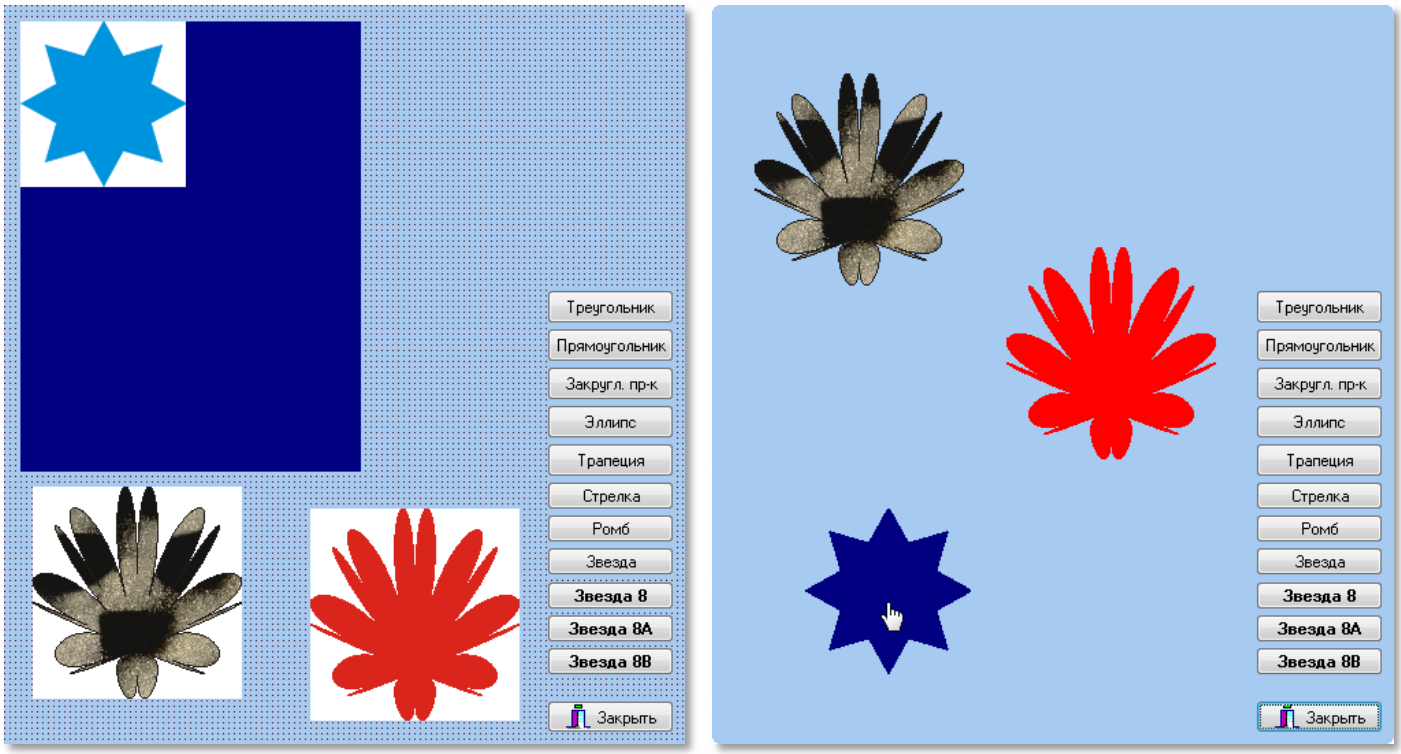


Рис. П2.8. Форма приложения в *Конструкторе формы* *Окно* приложения с фигурными плавающими панелями

Первые две кнопки просто вырезают из панелей окна по форме, совпадающие с изображением на картинке, поэтому панели сохраняют свой цвет, а изображение с картинки на них не переносится (сделать этого и нельзя, так как у компонента **TPanel** отсутствует свойство *Canvas*). Из этого следует, что картинку можно разместить не только на панели, но и вообще в любом месте формы, а также сделать её невидимой (Рис. П2.9).

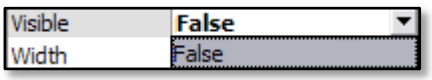


Рис. П2.9. Невидимая картинка

Если же вы хотите не только придать панели определённую форму, но и иметь на ней полноценную картинку, то тогда картинку обязательно нужно положить на панель, сделать видимой, а также установить свойства панели так, как было указано выше.

Поскольку теперь **Panel3** становится недоступной для мышки (на ней сверху лежит картинка), то нужно слегка подправить обработчик события нажатия мышки на панели:

```
//ПЕРЕТАСКИВАНИЕ ПАНЕЛЕЙ
procedure TForm1.Panel1MouseDown(Sender: TObject; Button:
TMouseButton;  Shift: TShiftState; X, Y: Integer);
const
  SC_DragMove = $F012;
begin
  ReleaseCapture;
  if Sender is TImage then panel3.Perform(WM_SysCommand,
SC_DragMove, 0)
  else(sender as Tpanel).Perform(WM_SysCommand, SC_DragMove, 0)
end;
```

Курсор для **Image3** нужно установить в *crHandPoint*, а также связать её обработчик с обработчиками для панелей: OnMouseDown | Panel1MouseDown.

Теперь нажатие на картинке **Image3** будет трактоваться в пользу панели **Panel3**, и её можно передвигать, как и все остальные панели.

На всякий случай добавлю: для первой панели действуют и все прежние кнопки, а для остальных – только их собственные. Причём свою форму панели изменяют только после нажатия на кнопку, а двигать их по форме можно в любое время.

Таким же способом можно разместить на форме любое количество панелей с соответствующими картинками и затеять какой-нибудь визуальный конструктор типа пентамино или танграма…



Исходный код программы находится в папке **PanelForm**.

Занятие 3. Викторина, или *Все флаги в гости будут к нам*

Я спросил у ясеня …

В.Киршон

Тут давай его пытать я…

В.Высоцкий.
Инструкция перед поездкой за рубеж

"Когда я был маленьким, я тоже отдыхал в пионерском лагере..." – говаривал товарищ Дынин, а я в его годы играл в викторину. Правда, тогда викторины были хотя и «электрифицированные», но не компьютерные, а посему запас вариантов у них был очень скудный и все премудрости сочетания вопросов и ответов я разгадал за пару дней, и игрушка навсегда исчезла из круга моих интересов.

Но, если верить философам, всё развивается по спирали (к счастью, восходящей), поэтому вполне разумно сделать самим такую викторину, которая никогда не надоест. Однако сейчас мы сделаем только самую простую версию этой игрушки – только для того, чтобы продемонстрировать, что с помощью передвигаемых резных панелей можно сделать действительно полезные вещи, а не только бездумно таскать их по форме.

Суть викторины состоит в том, что имеется некоторое число вопросов и столько же ответов, и игрок должен подобрать нужный ответ к каждому вопросу. Подобный принцип заложен и в некоторых психологических и других тестах (например, на мышление), поэтому нашу программу можно будет применить в самых разных областях умственной деятельности.

Пусть максимальное число вопросов в викторине равняется 12:

```
Const
  //Название программы:
  NAME_PROG='Victorina';
  //Максимальное число вопросов в викторине:
  MAX_FRAGE=12;
```

Вы, конечно, имеете полное право задать столько вопросов, сколько сумеете расположить на форме.

В конкретном задании может быть и меньше вопросов, но у нас их будет ровно 12.

```
Var
  Form1: TForm1;
  //Число вопросов в текущей викторине (1..MAX_FRAGE):
  VsegoVoprosov: integer;
```

Расположим на форме 12 компонентов **TImage** - **Image1…Image12**, в которые загрузим какие-нибудь картинки (в данной программе они все одного размера - это не обязательно, но так все вопросы получатся «аккуратными»). У нас это будут флаги весьма известных государств (Рис. П3.1).



Рис. П3.1. Викторинный интерфейс

Справа от них разместим ещё столько же компонентов **TImage** - **Image101…Image102**. Они нужны для того, чтобы показать игроку, куда он должен поместить карточки с ответами. А дальше идут наши любимые панели **Panel1…Panel12**, которые в этот раз будут играть роль карточек с

ответами. Так как *Delphi* не позволяет в *Конструкторе формы* создавать массивы компонентов, а работать с ними значительно удобнее, чем с большим количеством отдельных компонентов, то заведём три массива для рассмотренных нами компонентов:

```
  //Массив компонентов TImage с картинками-вопросами:
  masFrage: array[0..MAX_FRAGE] of TImage;
  //Массив компонентов TImage, обозначающих места для карточек с
ответами:
  masAntwort: array[0..MAX_FRAGE] of TImage;
  //Массив компонентов TPanel с карточками-ответами:
  masPanel: array[0..MAX_FRAGE] of TPanel;
```

При запуске программы заполняем эти массивы:

```
//СОЗДАТЬ ФОРМУ
procedure TForm1.FormCreate(Sender: TObject);
var i: integer;
begin
  Randomize;

  //создать массив карточек с вопросами из компонентов
  //TImage на форме:
  for i:= 1 to MAX_FRAGE do begin
    masFrage[i]:= FindComponent('Image'+inttostr(i)) as TImage;
    //расположить карточку с вопросом на форме
    //(это нужно сделать только при запуске программы,
    //так как карточки с вопросами имеют одинаковый размер
    //и в игре только меняются местами в каждой новой игре):
    masFrage[i].Top:= 4+(i-1)*(masFrage[i].Height-4);
    masFrage[i].Left:= 8;
    //номер вопроса хранится в свойстве Tag:
    masFrage[i].Tag:= i;

    //создать массив карточек с ответами из компонентов
    //TPanel на форме:
    masPanel[i]:= FindComponent('Panel'+inttostr(i)) as TPanel;
    with masPanel[i] do begin
      //запомнить начальные координаты карточек,
      //так как в ходе игры карточки перемещаются по форме:
      masCoordPanel[i].X:=masFrage[i].Left        +        masFra-
ge[i].Width+masPanel[i].Width+30;
      masPanel[i].Height:= 32;
      masPanel[i].Color:= clrNetOtveta;
      masPanel[i].Font.Color:= clGreen;
      //номер ответа хранится в свойстве Tag
      //и совпадает с номером вопроса:
      masPanel[i].Tag:= i;
      //придать панели-карточки нужную форму:
      CreateFigure(masPanel[i]);
    end;
```

```
    //создать массив "посадочных мест" из компонентов TImage
    //на форме для карточек с ответами :
    masAntwort[i]:=    FindComponent('Image'+inttostr(i+100))    as
TImage;
    masAntwort[i].Top:= masFrage[i].Top+4;
    masAntwort[i].Left:= masFrage[i].Left+masFrage[i].Width+ 4;
    masAntwort[i].Height:= masFrage[i].Height-8;
    masAntwort[i].Width:= masPanel[i].Width+11;
    masAntwort[i].Canvas.Brush.Color:= clTeal;
    masAntwort[i].Canvas.FillRect(masAntwort[i].clientrect);

    //запомнить начальные координаты карточек,
    //так как в ходе игры карточки перемещаются по форме:
    masPanel[i].top:=   masAntwort[i].Top+   (masAntwort[i].Height-
masPanel[i].Height) div 2;
    masCoordPanel[i].Y:=masPanel[i].Top;
  end;

  //количество вопросов в задании:
  VsegoVoprosov:=12;
  //начать новую игру:
  NewGame;
end;
```

Теперь мы можем работать с компонентами, как с массивами! Обратите внимание, что на форме вопросы и соответствующие им ответы находятся в компонентах с одинаковыми номерами: **Image1 – Panel1, Image2 – Panel2**, и так далее. Безусловно, так значительно проще составлять задания и следить за решением задачи. А при создании массивов свойству **Tag** этих компонентов мы присваиваем такие же значения.

Кроме того, мы должны запомнить начальные координаты карточек с ответами, так как в процессе игры они будут перемещаться:

```
  //Массив начальных координат компонентов TPanel с карточками-
ответами:  masCoordPanel: array[0..MAX_FRAGE] of TPoint;
```

Когда потребуется, мы легко сможем вернуть карточки на их законное место.

Здесь же мы придаём панелям привлекательный вид:

```
//ПРИДАТЬ ПАНЕЛЯМ-КАРТОЧКАМ НУЖНУЮ ФОРМУ
procedure CreateFigure(panel: TPanel);
var
  fig: array [0..5] of TPoint;
  rgn: HRGN;
begin
```

```
  with panel do begin
    fig[0]:= Point(0,Height div 2);
    fig[1]:= Point(10, 0);
    fig[2]:= Point(Width, 0);
    fig[3]:= Point(Width-10,Height div 2);
    fig[4]:= Point(Width, Height);
    fig[5]:= Point(10, Height);
    rgn:= CreatePolygonRgn (fig, 6, WINDING);  //ALTERNATE
    SetWindowRgn(Handle, rgn, True);
  end;
end;
```

Я выбрал для карточек с ответами многоугольную форму в виде стрелок, указывающих направление их перемещения (Рис. П3.2).



Рис. П3.2. Панели-стрелки

Как мы уже выяснили раньше, они могут быть любой, самой замысловатой формы.

И, наконец, мы указываем программе число вопросов в задании и начинаем новую игру:

```
//НАЧАТЬ НОВУЮ ИГРУ
procedure NewGame;
var i,n1,n2: integer;
  imgTemp: TImage;
begin

  for I := 1 to VsegoVoprosov do begin
    //на этот вопрос ещё не отвечали:
    masMoiOtvety[i]:= clrNetOtveta;
    //все панели с ответами находятся в исходном положении:
    masRowPanel[i]:= 0;
    masPanel[i].Left:=masCoordPanel[i].X;
    masPanel[i].Top:=masCoordPanel[i].Y;
    masPanel[i].Color:= clrNetOtveta;
  end;

  //перемешать вопросы:
  imgTemp:= TImage.Create(nil);
  for i:=1 to 61 do begin
    //выбрать случайным образом две карточки с вопросами:
    n1:= Random(VsegoVoprosov)+1;
    n2:= Random(VsegoVoprosov)+1;
    //и, если они не совпадают, поменять их местами
    //(вместе с номерами вопросов!):
    if n1<> n2 then begin
       imgTemp.Picture.Bitmap := masFrage[n1].Picture.Bitmap;
       imgTemp.tag := masFrage[n1].tag;
       masFrage[n1].Picture.Bitmap:= masFrage[n2].Picture.Bitmap;
       masFrage[n1].tag:= masFrage[n2].tag;
       masFrage[n2].Picture.Bitmap:= imgTemp.Picture.Bitmap;
       masFrage[n2].tag:= imgTemp.tag;
    end;
  end;
  imgTemp.Free;

  //показать, что не дано ни одного ответа:
  PrintDanoOtvetov;
  //обнулить счётчик времени игры:
  iSec:=0;
  //все карточки лежат на месте:
  nDrag:=0;
end;
```

В этой процедуре мы расставляем панели с ответами по местам и перемешиваем вопросы, чтобы условия игры не повторялись (помните о моей

детской викторине, которая не обладала этим похвальным свойством непредсказуемости).

И вот - игра началась! Мы берём карточку с нужным ответом и перемещаем её по форме – точно так же, как мы это делали раньше. Но – теперь мы должны запомнить, какую именно карточку взяли. Как вы помните, её номер хранится в свойстве *Tag*, но нам удобнее запомнить номер карточки в переменной:

```
  //номер перемещаемой карточки:
  nDrag: integer;
```

```
//БЕРЁМ КАРТОЧКУ С ОТВЕТОМ
procedure    TForm1.Panel1MouseDown(Sender:    TObject;    Button:
TMouseButton;
  Shift: TShiftState; X, Y: Integer);
const
  SC_DragMove = $F012;
var i: integer;
begin
  //игра началась - режим подсказок изменять нельзя!
  sbtPodskazki.Enabled:=FALSE;
  //берём карточку:
  sound('взять');
  //номер карточки с ответом 1..:
  nDrag:= TPanel(sender).tag;
  //listbox1.Items.Add('Берём '+inttostr(nDrag));

  //если эта карточка уже лежит в строке i
  //как ответ на вопрос, то убрать её из строки ответов:
  for I := 1 to VsegoVoprosov do
     if masRowPanel[i]= nDrag then masRowPanel[i]:=0;

  //располагаем её над всеми другими карточками:
  (sender as Tpanel).BringToFront;
  //перетаскиваем:
  ReleaseCapture;
  (sender as Tpanel).Perform(WM_SysCommand, SC_DragMove, 0);
end;
```

Как только мы берём первую карточку, кнопка переключения режимов подсказок становится недоступной:

```
  //игра началась - режим подсказок изменять нельзя!
  sbtPodskazki.Enabled:=FALSE;
```

Это же событие запускает отсчёт времени игры в *таймере*:

```
//ВЫВОДИМ ВРЕМЯ ИГРЫ
```

```
procedure TForm1.Timer1Timer(Sender: TObject);
var
  s: string;
  min, sec: integer;
begin
  //время игры:
  if sbtPodskazki.Enabled=FALSE then inc(iSec);//игра идёт

  min:= Round(iSec) div 60;
  if min> 59 then begin //- ваш час пробил!
    Timer1.Enabled:= False;
    s:= 'Ваше время закончилось!';
    Application.MessageBox(pChar(s), NAME_PROG, IDOK);
    exit
  end;
  //вывести время:
  sec:= Round(iSec) mod 60;
  s:= inttostr(sec);
  if sec< 10 then s:= '0' + s;
  s:= inttostr(min)+ ' : ' + s;

  if min>= 5 then lblTime.Font.Color:= clrNevernyOtvet
  else lblTime.Font.Color:= clrVernyOtvet;

  lblTime.Caption:= s;
end;
```

В этой программе время никак не учитывается и служит только для информирования игрока. Впрочем, если игрок в течение 5 минут не справился с заданием, время выводится красным цветом!

Всё самое интересное происходит в процедуре, в которой мы бросаем карточку на одно из предписанных мест для ответов (или мимо, что тоже исключать нельзя!).

```
//БРОСАЕМ КАРТОЧКУ
procedure    TForm1.Panel1MouseMove(Sender:    TObject;    Shift:
TShiftState; X,
  Y: Integer);
//X,Y - координаты мыши на панели
var nCard,k,px,py,pw,t: integer;
i: integer;
begin
  //карточка не перетаскивается:
  if nDrag=0 then exit;

   //номер карточки с ответом:
   nCard:= TPanel(sender).tag;

  //карточку бросили:
  sound('бросиликарту');
```

```
  nDrag:=0;

  px:= TPanel(sender).left;
  py:= TPanel(sender).top;

  //определяем номер "посадочного места":
  t:=1000; k:=0;
  for i:=1 to MAX_FRAGE do
      if (py-masFrage[i].Top< t) and (py-masFrage[i].Top>=0) then
begin
         t:= (py-masFrage[i].Top);
         k:=i;
      end;

  //если на этом месте уже есть карточка, то новую вернуть на её
место:
  if (masRowPanel[k]> 0) then begin
     sound('мимо');
     with masPanel[nCard] do begin
       Color:= clrNetOtveta;
       SetBounds(masCoordPanel[nCard].X,    masCoordPanel[nCard].Y,
Width, Height);
       //masPanel[nCard].Left:=masCoordPanel[nCard].X;
       //masPanel[nCard].Top:=masCoordPanel[nCard].Y;
     end;
     masMoiOtvety[nCard]:= clrNetOtveta;
     PrintDanoOtvetov;
     exit;
  end;

  //карточку бросили на посадочное место:
  if (px <= masAntwort[1].Left+masAntwort[1].Width) then begin
    //скорректировать положение панели:
     with TPanel(sender) do begin
       SetBounds(masAntwort[1].left+  (masAntwort[1].Width-  Width)
div  2,  masAntwort[k].Top+  (masAntwort[k].Height-  Height)  div  2,
Width, Height);
       //left:=  masAntwort[1].left+  (masAntwort[1].Width-  Width)
div 2;
       //top:=  masAntwort[k].Top+  (masAntwort[k].Height-  Height)
div 2;
     end;
      //запомнить ответ:
     masRowPanel[k]:= nCard;

    if nCard = masFrage[k].tag then begin
       if    sbtPodskazki.Down    then    masPanel[nCard].Color:=
clrVernyOtvet;
       masMoiOtvety[nCard]:= clrVernyOtvet
    end
    else begin
      if  sbtPodskazki.Down  then  masPanel[nCard].Color:=  clrNe-
vernyOtvet;
```

```
      masMoiOtvety[nCard]:= clrNevernyOtvet
    end
  end

  else begin //мимо!
   sound('мимо');
   with masPanel[nCard] do begin
     Color:= clrNetOtveta;
       SetBounds(masCoordPanel[nCard].X,    masCoordPanel[nCard].Y,
Width, Height);
       //masPanel[nCard].Left:=masCoordPanel[nCard].X;
       //masPanel[nCard].Top:=masCoordPanel[nCard].Y;
   end;
   masRowPanel[k]:=0;
   masMoiOtvety[nCard]:= clrNetOtveta;
  end;

  PrintDanoOtvetov;
end;
```

Казалось бы, обработка бросания карточки должна происходить в процедуре **Panel1MouseUp**, однако при отпускании кнопки мыши событие **OnMouseUp** не возникает, что объясняется нашим экстравагантным способом перемещения компонентов.

Сначала мы определяем, перетаскивается ли вообще карточка или мышка пустая:

```
  //карточка не перетаскивается:
  if nDrag=0 then exit;
```

Затем определяем номер карточки:

```
   //номер карточки с ответом:
   nCard:= TPanel(sender).tag;
```

Бросаем карточку с надлежащим звуком и обнуляем переменную **nDrag** как признак того, что перетаскивание карточки закончено:

```
  //карточку бросили:
  sound('бросиликарту');
  nDrag:=0;
```

Для интерпретации ответа нам нужно знать, в какую «строку» с ответами попала карточка:

```
  px:= TPanel(sender).left;
  py:= TPanel(sender).top;
```

```
  //определяем номер "посадочного места":
  t:=1000; k:=0;
  for i:=1 to MAX_FRAGE do
      if (py-masFrage[i].Top< t) and (py-masFrage[i].Top>=0) then
begin
         t:= (py-masFrage[i].Top);
         k:=i;
      end;
```

Для этого мы находим минимальное расстояние между верхней стороной карточки и верхней стороной картинок с вопросами.

Так как здравый смысл нам подсказывает, что на каждый вопрос можно дать **только один** правильный ответ, то и на каждом «посадочном месте» может находиться только одна карточка с ответом (но вы можете убрать эту проверку, и тогда карточки можно будет складывать друг на друга пачками, что усложнит и увеселит игру). Если же на данный вопрос ответ уже получен (карточка лежит в «строке» с вопросом), то брошенную карточку следует вернуть на её место:

```
  //если на этом месте уже есть карточка, то новую вернуть на её
место:
  if (masRowPanel[k]> 0) then begin
     sound('мимо');
     with masPanel[nCard] do begin
       Color:= clrNetOtveta;
       SetBounds(masCoordPanel[nCard].X, masCoordPanel[nCard].Y,
Width, Height);
       //masPanel[nCard].Left:=masCoordPanel[nCard].X;
       //masPanel[nCard].Top:=masCoordPanel[nCard].Y;
     end;
     masMoiOtvety[nCard]:= clrNetOtveta;
     PrintDanoOtvetov;
     exit;
  end;
```

Если всё нормально, то выравниваем карточку по центру посадочного места, запоминаем и обрабатываем полученный ответ.

Чтобы игроку было удобно следить за своими успехами, показываем на экране число данных ответов:

```
//ПОКАЗАТЬ НА ЭКРАНЕ ЧИСЛО ОТВЕТОВ
procedure PrintDanoOtvetov;
var n: integer;
begin
  Test(n);
  if n < VsegoVoprosov then
form1.lblDanoOtvetov.Font.Color:=clrNevernyOtvet
```

```
  else form1.lblDanoOtvetov.Font.Color:=clrVernyOtvet;

  form1.lblDanoOtvetov.Caption:= inttostr(n)
+'/'+inttostr(VsegoVoprosov);
end;
```

Когда игрок ответит на все вопросы викторины, он должен проверить своё решение, нажав соответствующую кнопку (Рис. П3.3).

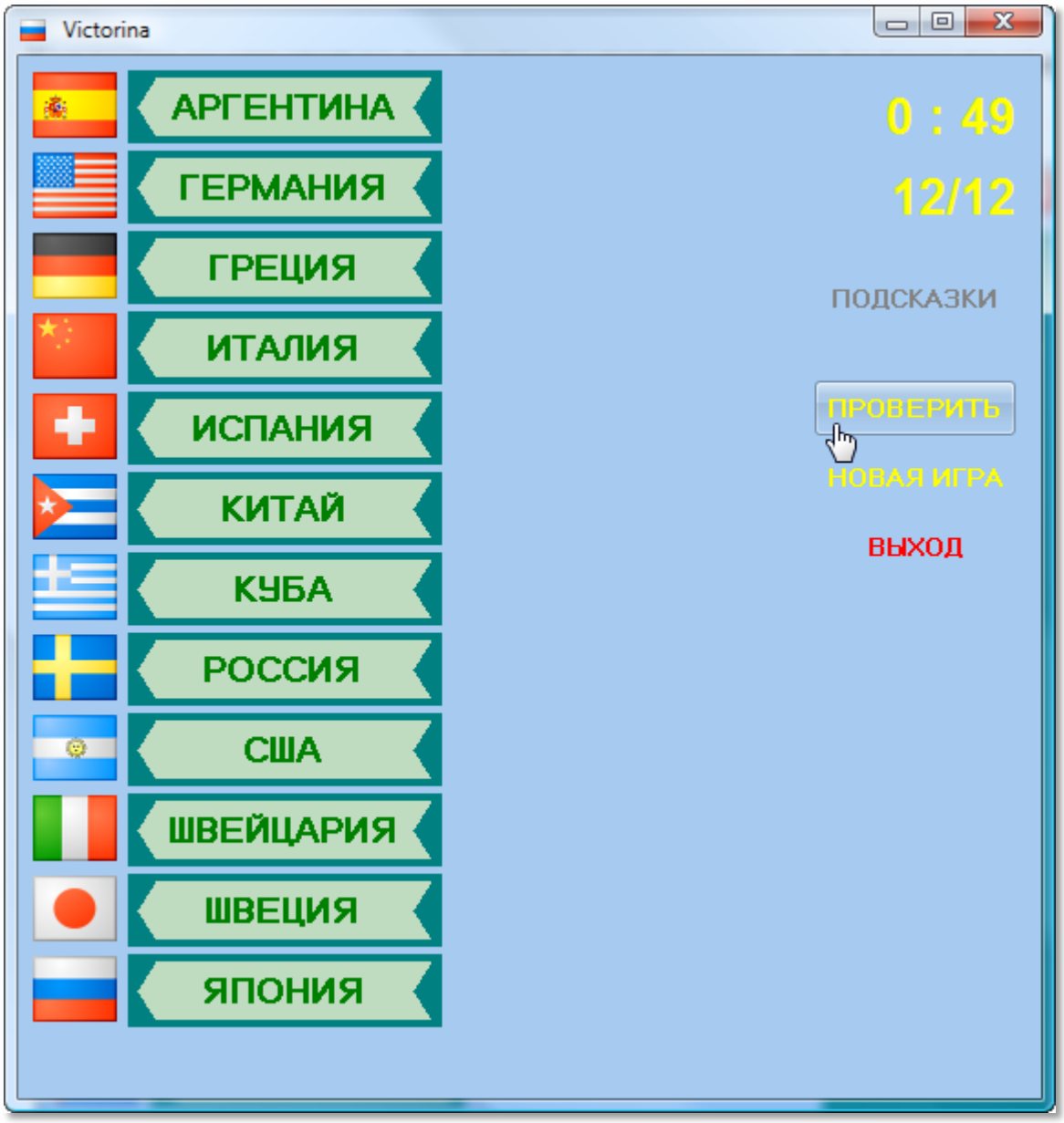


Рис. П3.3. Викторина страшнее ЕГЭ!!!

```
//ПРОВЕРИТЬ РЕШЕНИЕ
function Test (var nDanoOtvetov:integer): integer;
//Возвращает число правильных ответов
//В nDanoOtvetov - общее число ответов
var i: integer;
```

```
begin
  Result:= 0;
  //число вопросов, на которые даны ответы:
  nDanoOtvetov:=0;
  for i := 1 to VsegoVoprosov do begin
    if masMoiOtvety[i]= clrVernyOtvet then inc(Result);
    if masMoiOtvety[i]<> clrNetOtveta then inc(nDanoOtvetov);
  end;
end;
```

```
//НАЖАТЬ КНОПКУ "ПРОВЕРИТЬ" - ПРОВЕРИТЬ РЕШЕНИЕ
procedure TForm1.sbtTestClick(Sender: TObject);
var n,k,i: integer;
    s: string;
begin
  n:= Test(k);
  if k<VsegoVoprosov then begin
     s:='Вы ответили не на все вопросы! (' + inttostr(k)
+'/'+inttostr(VsegoVoprosov)+')';
     messagebox(handle,PCHar(s),'Victorina',MB_OK+MB_ICONWARNING);
     //showmessage(s);
     exit
  end;

  if n = VsegoVoprosov then begin
     s:= 'Вы правильно ответили на все вопросы! (' + inttostr(n)
+')';
     message-
box(handle,PCHar(s),'Victorina',MB_OK+MB_ICONINFORMATION);
     //showmessage(s)
  end
  else begin
     s:= 'Вы правильно ответили не на все вопросы! (' + int-
tostr(n) +'/'+inttostr(VsegoVoprosov)+')';
     message-
box(handle,PCHar(s),'Victorina',MB_OK+MB_ICONASTERISK);
     //showmessage(s);
  end;

  //отметить цветом правильные и неправильные ответы:
  for i := 1 to VsegoVoprosov do masPanel[i].Color:=
masMoiOtvety[i];

  sbtPodskazki.Enabled:= TRUE;
end;
```

Если игрок ответил не на все вопросы, он продолжает игру, иначе получает заключение о своих успехах в виде таблички с надписью и соответствующим окрашиванием карточек с ответами. Правильные ответы – **жёлтого** цвета, неправильные – **красного** (Рис. П3.4).

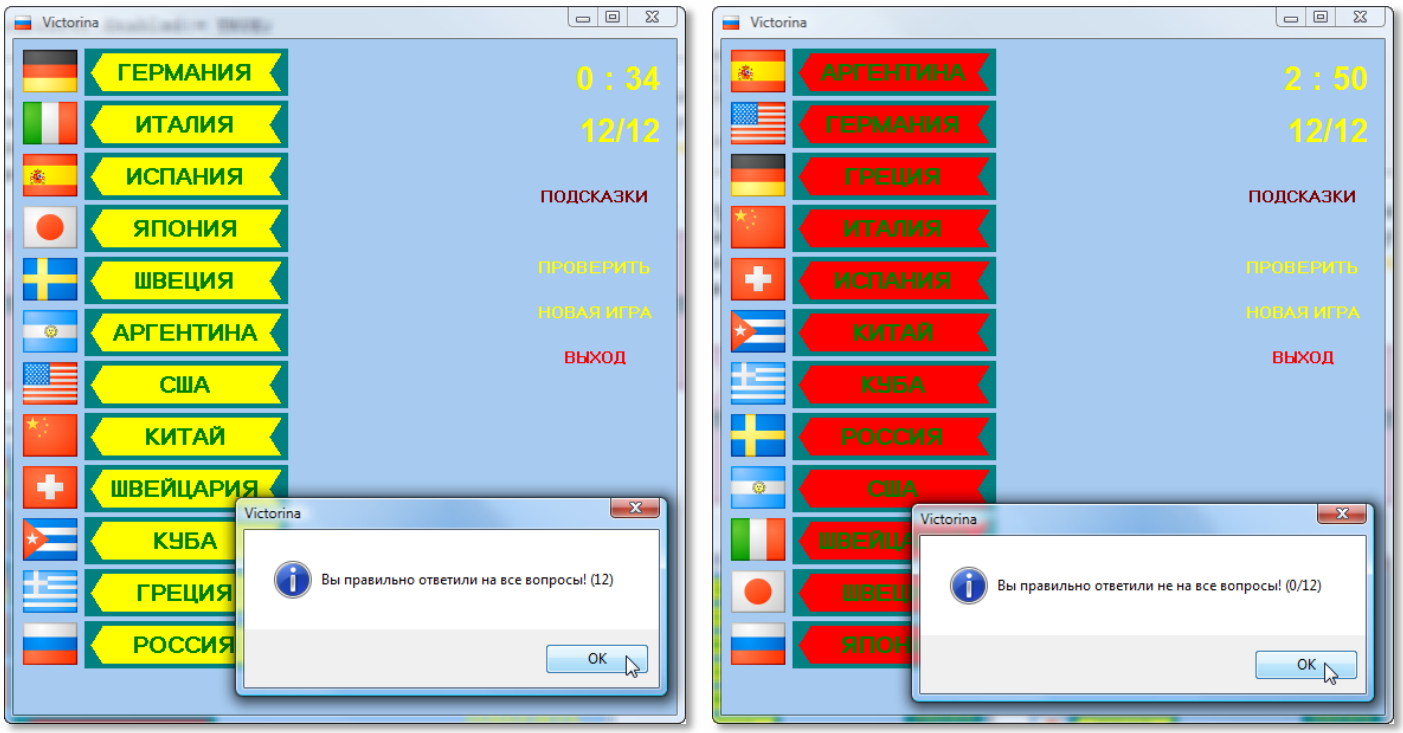


Рис. П3.4. Слева отвечал Путин, справа - Барак

Цвет карточек определяется *константами*:

```
const
  //Название программы:
  NAME_PROG='Victorina';
  //Максимальное число вопросов в викторине:
  MAX_FRAGE=12;
  //Цвет карточек:
  clrNetOtveta= clMoneyGreen; //- карточка не использована
  clrVernyOtvet= clYellow;    //- карточка на правильном месте
  clrNevernyOtvet= clRed;     //- карточка на неправильном месте
```

так что вы легко можете подобрать собственную цветовую гамму для карточек.

Впрочем, по умолчанию карточки **сразу** окрашиваются в тот цвет, который характеризует её правильность. За это отвечает кнопка **sbtPodskazki** *ПОДСКАЗКИ*: если она нажата, то карточки окрашиваются сразу, если отжата – только по требованию игрока. Режим по умолчанию разумно использовать при «тренировочных заездах», а затем проверять «усвоение материала» уже без подсказок. Например, выучив флаги разных стран, вы можете взяться на изучение иностранных слов или дорожных знаков.
Как вы видите, по цвету карточки сразу можно определить, насколько правильным был ответ, беда в том, что в режиме без подсказок мы не можем контролировать правильность ответов, так как карточки свой цвет не

меняют. Поэтому введём дополнительный массив, в котором и будем сохранять нужные цвета:

```
  //Массив ответов, данных игроком - нет ответа=clrNetOtveta,
  //верный ответ=clrVernyOtvet, неверный ответ=clrNevernyOtvet:
  masMoiOtvety: array[0..MAX_FRAGE] of TColor;
```

Осталось напомнить, что в любое время вы можете начать новую игру, нажав кнопку

```
//НАЖАТЬ КНОПКУ "НОВАЯ ИГРА" - НАЧАТЬ НОВУЮ ИГРУ
procedure TForm1.sbtNewGameClick(Sender: TObject);
begin
  //перед началом игры можно изменить режим подсказок:
  sbtPodskazki.Enabled:=TRUE;
  NewGame;
end;
```

Используя этот же метод перемещения панелей, можно написать и более изощрённые программы, например, карточные пасьянсы, ведь в них тоже главное – перемещать карты и складывать их в кучки согласно правилам игры (Рис. П3.5).



Рис. П3.5. Панельный пасьянс

Исходный код программы находится в папке **PanelForm/Victorina**.

Занятие 4. Тест космонавта, или Форма как ракетоноситель

Открылась бездна, звезд полна.

М.В.Ломоносов

Много-много лет тому назад Юрий Гагарин сказал своё знаменитое «Поехали!» и всех нас позвал в космос… Да только полёт в межпланетное пространство – это вам не пикник на обочине жизни! Тут нужно готовиться основательно!

Вы, конечно, знаете (или хотя бы догадываетесь), что готовят будущих космонавтов по Суворовскому принципу «Тяжело в учении – легко в бою»: их крутят, жмут, мнут, бросают, ловят и снова бросают. А в редкие минуты «отдыха» от экстремальной физкультуры «подопытным» наддают психологического жару – заставляют решать задачи и тесты и при этом всячески им мешают: включают мигающий свет, сирену, бубнят текст не по теме, щекочут за пятки и травят жареным гусем. Во как! Мы такого наслаждения вам не обещаем, но немного помучить можем.

Мы собрались здесь затем, чтобы украсить форму эффектной картинкой и воспользоваться этим случаем для создания себе невыносимых условий с помощью нашей очередной программы. Она явит собой некое подобие одного из космических тестов. Правда, в щадящем, гуманистическом виде. Итак…

Стресс как порча жизни

Тест представляет собой квадратную таблицу, в клетках которой случайным образом расставлены последовательные натуральные числа. Испытуемый должен находить эти числа: сначала 1, затем 2, 3 и так далее - до самого последнего.

В более простом варианте теста все числа в таблице – чёрные, в усложнённом – **чёрные** и **красные** (в этом случае после каждого чёрного числа нужно отыскать такое же красное).

Нам осталось перевести тест на компьютерный язык, который у нас к этому времени уже должен быть хорошо подвешен.

Так как картинку-фон каждый может подобрать самостоятельно, то и размеры формы задайте под картинку. Особая точность здесь не нужна –

функция **StretchDraw**, подобно мифическому Прокрусту растянет или сожмёт картинку так, чтобы она целиком заняла всю форму (клиентскую часть). В этом примере ширина формы равна 677 пикселям, высота – 510. Лучше подобрать картинку примерно такой величины, иначе космическое пространство не произведёт должного эффекта.

Для отображения таблицы правильнее всего взять наш любимый компонент **DrawGrid**:

```
DrawGrid: Name=dgPole

Colcountt=5
RowCount=5
Cursor=crHandPoint
DefaultColWidth=25
DefaultRowHeight=25
DefaultDrawing=False
FixedCols=0
FixedRows=0
Height=133
Width=133
Left=96
Top=96 (если вы взяли для фона другую картинку, то расположите на
ней сетку так, чтобы она не портила красивый вид)
Options=[goVertLine,goHorzLine]
ScrollBars=ssNone
```

Чтобы вы не заблудились в программе, приведу её начало сразу в готовом виде:

```
unit TstcomUnit;
////////////////////////////////////////////////////////////////////////
//                              ТЕСТ КОСМОНАВТА                       //
////////////////////////////////////////////////////////////////////////

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
Dialogs, Grids, Menus, StdCtrls, ExtCtrls, ComCtrls, Buttons;

type
  TForm1 = class(TForm)
    dgPole: TDrawGrid;
    PopupMenu1: TPopupMenu;
    N5x51: TMenuItem;
    N6x61: TMenuItem;
    N7x71: TMenuItem;
    N8x81: TMenuItem;
    N9x91: TMenuItem;
    N1: TMenuItem;
```

```
    N2: TMenuItem;
    Timer1: TTimer;
    miStart: TMenuItem;
    N4: TMenuItem;
    sbtPlay: TSpeedButton;
    N3: TMenuItem;
    N5: TMenuItem;
    N6: TMenuItem;
    miGreen: TMenuItem;
    miHelp: TMenuItem;
    procedure FormCreate(Sender: TObject);
    procedure FormDestroy(Sender: TObject);
    procedure FormPaint(Sender: TObject);
    procedure N5x51Click(Sender: TObject);
    procedure dgPoleDrawCell(Sender: TObject; ACol, ARow: Integer;
Rect: TRect; State: TGridDrawState);
    procedure N2Click(Sender: TObject);
    procedure dgPoleMouseDown(Sender: TObject; Button: TMouseBut-
ton; Shift: TShiftState; X, Y: Integer);
    procedure Timer1Timer(Sender: TObject);
    procedure miStartClick(Sender: TObject);
    procedure N4Click(Sender: TObject);
    procedure miGreenClick(Sender: TObject);
    procedure miHelpClick(Sender: TObject);

  private
    { Private declarations }
    Bitmap: TBitmap;
    procedure Prepare(n: integer);
    procedure SetNumbers;

  public
    { Public declarations }
  end;

const
  NAME_PROG= '  Тест космонавта    ';
  MAX_N= 9;  //макс. размеры поля

//разновидность игры:
type TGameMode=(gmBlackOnly, gmBlackRed);

//статус клетки - не выбрана/выбрана:
type TStatusCell=(scWhite, scGreen);

//статус игры - остановлена/запущена:
type TGameStatus= (STOP, PLAY);

//информация о клетке поля:
type TCell= Record
  num: integer; //- число в клетке
```

```
  StatusCell: TStatusCell
end;

var
  Form1: TForm1;
  //всего чисел на поле:
  AllNumbers: integer;
  //выбрано чисел на поле:
  ReadyNumbers: integer;
  //массив, в котором хранится информация о поле:
  masPole: array[0..MAX_N, 0..MAX_N] of TCell;
  //вид игры:
  GameMode: TGameMode;
  time,time0: single;
  //переключатель цвета чисел:
  switch: integer= 1;
  //статус игры:
  GameStatus: TGameStatus= STOP;

implementation

uses HelpUnit;

{$R *.DFM}
```

При запуске программы сетка имеет размеры 5 x 5 клеток. Для усложнения задачи предусмотрена возможность увеличения поля до 9 x 9 клеток. Максимально допустимое значение размеров поля будем хранить в константе **MAX_N**. Вы можете сделать поле ещё больше, но тогда придётся либо увеличивать размеры ячеек сетки, либо уменьшать высоту шрифта. И то, и другое одинаково «вредно».

```
const
  NAME_PROG= '  Тест космонавта     ';
  MAX_N= 9;  //макс. размеры поля
```

Так как игра имеет два варианта, то нам необходимо их различать. Чтобы избежать безликой цифири, введём тип **TGameMode** с удобными обозначениями вида игры:

```
//разновидность игры:
type TGameMode=(gmBlackOnly, gmBlackRed);
```

и объявим соответствующую *переменную*:

```
//вид игры:
GameMode: TGameMode;
```

Клетки с невыбранными ещё числами пусть будут белыми, а с выбранными – зелёными. Соответственно, статус белых клеток назовём **scWhite**, а зелёных – **scGreen**:

```
//статус клетки - не выбрана/выбрана:
type TStatusCell=(scWhite, scGreen);
```

Дальше мы сделаем так, чтобы цвет клеток на игровом поле всегда оставался белым, но их статус при этом так и останется «белым» и «зелёным». Для каждой клетки поля мы должны знать, какое число в ней стоит и выбрано оно или нет. Для хранения этих данных введём новый *тип*:

```
//информация о клетке поля:
type TCell= Record
  num: integer; //- число в клетке
  StatusCell: TStatusCell
end;
```

Нам потребуется также *массив*, в который мы будем записывать данные обо всех клетках поля:

```
//массив, в котором хранится информация о поле:
masPole: array[0..MAX_N, 0..MAX_N] of TCell;
```

Количество чисел (клеток) на поле будем хранить в *переменной*:

```
//всего чисел на поле:
AllNumbers: integer;
```

А уже найденных - в *переменной*:

```
//выбрано чисел на поле:
ReadyNumbers: integer;
```

Очевидно, что, когда **ReadyNumbers** сравняется с **AllNumbers**, все числа будут найдены, а задача решена.

Все переменные объявлены, сетка готова – нужно заполнить её числами. Перебираем последовательно все клетки поля. Выбираем случайное число и, если оно ещё не использовано (**iMas** для него равно *False*), то ставим его в очередную клетку. В противном случае берём другое случайное число – пока не найдём нового.

```
 //РАССТАВИТЬ ЧИСЛА
procedure TForm1.SetNumbers;
var
  i, j, n, m: integer;
```

```
  iMas: array[1..MAX_N * MAX_N] of boolean;
  flg: boolean;
begin
  //"перемешать" числа:
  Randomize;

  //размер поля:
  m:= dgPole.ColCount;

  //ни одно число не поставлено в сетку:
  for i:= 1 to MAX_N * MAX_N do iMas[i]:= false;

  //расставляем числа:
  for j:= 0 to m-1 do
    for i:= 0 to m-1 do begin
      flg:= false; //- число не выбрано
      repeat
        //случайное число:
        n:= Random(AllNumbers)+ 1;
        if iMas[n]= false then begin //- такого числа не было
          iMas[n]:= True;         //- число вышло
          masPole[i, j].num:= n; //- занести его в массив
          flg:= True;             //- число выбрано
          //клетка не выбрана:
          masPole[i, j].StatusCell:= scWhite;
        end;
      until flg= True; //повторять, пока не будет выбрано число
    end;

  if GameMode= gmBlackRed then begin
    //если "чёрно-красный" вариант игры, то вторую половину чисел
    //заменяем на красные (отрицательные):
    for j:= 0 to m-1 do
      for i:= 0 to m-1 do begin
        if masPole[i, j].num> ((m*m +1) div 2) then
          masPole[i, j].num:= -(masPole[i, j].num - ((m*m +1) div
2))
      end;
  end;
end; //SetNumbers
```

Если с чёрными числами всё просто, то с красными придётся повозиться. Их можно расставлять разными способами. Мы остановимся на самом простом. Вторую половину чисел (меньшую - при нечётном количестве клеток на поле) заменим отрицательными, равными по абсолютной величине чёрным. Тогда нам легко будет отличать их друг от друга: чёрные числа положительные, красные – отрицательные.

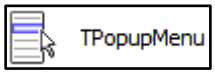
Применённый в программе алгоритм заполнения сетки числами прост, но не лишён недостатков. Так, несколько последовательных чисел могут ока-

заться в соседних клетках, поэтому их легко будет найти. При больших размерах поля выигрыш во времени будет не очень велик, но при малых вы можете стать рекордсменом незаслуженно. Подумайте над более качественным алгоритмом, чтобы каждое следующее число сравнивалось бы с теми, что уже стоят в соседних клетках.

Так как размеры поля мы хотим изменять, то в процедуре **Prepare** вычисляем размеры компонента **dgPole**, а заодно подготавливаем нужные переменные к новой игре:

```
//ПОДГОТОВКА К ИГРЕ
procedure TForm1.Prepare(n: integer);
var
  w, h, lw: integer;
begin
  //размер клетки в пикселях:
  w:= dgPole.DefaultColWidth;
  h:= dgPole.DefaultRowHeight;
  //толщина линий:
  lw:= dgPole.GridLineWidth;
  //размеры игрового поля в пикселях:
  dgPole.Width:= 3 + (w + lw)* n;
  dgPole.Height:= 3 + (h + lw)* n;
  //размеры игрового поля в клетках:
  dgPole.ColCount:= n;
  dgPole.RowCount:= dgPole.ColCount;

  //всего чисел на поле:
  AllNumbers:= n * n;
  //ни одно число не выбрано:
  ReadyNumbers:= 0;
  //расставить числа в сетке:
  SetNumbers;
  //обнулить время решения:
  caption:= NAME_PROG + '0 : 00';
  switch:= 1;
  dgPole.Invalidate;
end;
```

Обычно для управления программой мы расставляем на форме кнопки. И это правильно. Но – воспользуемся тем, что у каждого правила есть свои приятные исключения и не будем портить красивую картинку посторонними элементами. Среди компонентов *Delphi* есть и такой симпатичный, как **PopupMenu**. Он ненавязчив: его не кликнешь, он и не появится. Вы его легко найдёте на странице *Standard Палитры компонентов* 

. Дважды щёлкните на нём – и он появится на форме. Забежим немного впе-

рёд себя и поставим аналогичным щелчком **таймер** 

, он возглавляет группу товарищей на странице *System*. В итоге небывалых усилий должна получиться такая конструкция (Рис. П4.1).

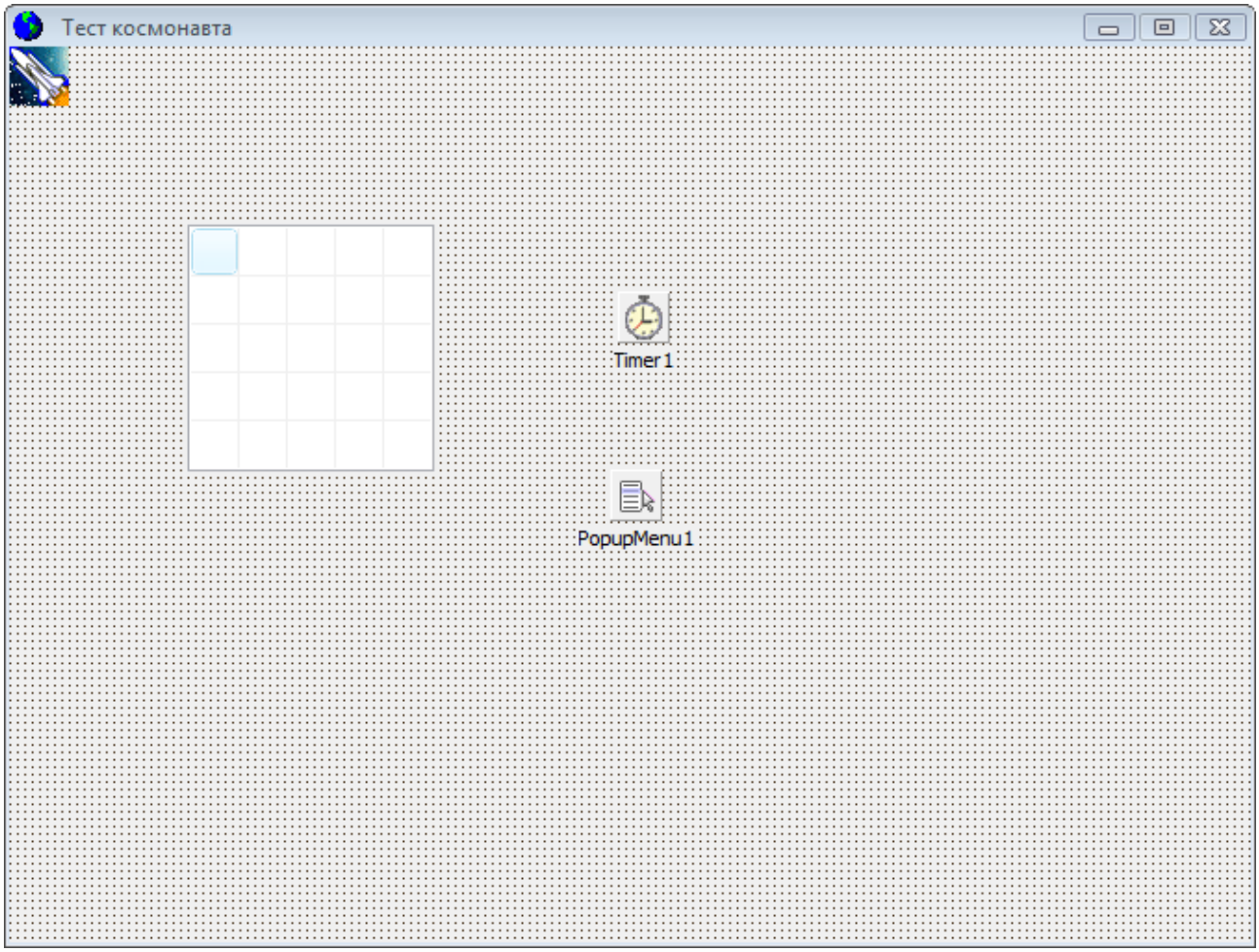


Рис. П4.1. Космический интерфейс

Форма, как видите, уныло-серая. Не грустите – соловей вон тоже невзрачный, а как чирикает! Картинку нарисуем при запуске программы. Хотя небольшой её кусочек вы уже сейчас можете наблюдать в левом верхнем углу. Никому ни слова – это секретная кнопка! На фоне космоса она будет неприметна, но каждый знакомый с трюками может ей пользоваться.

Для хранения картинки создадим переменную **Bitmap**, в которую и загрузим её при запуске программы:

```
//СОЗДАТЬ ФОРМУ
procedure TForm1.FormCreate(Sender: TObject);
begin
  Bitmap := TBitmap.Create;
  Bitmap.LoadFromFile('cosmos.bmp');
  Randomize;
```

```
  //игра по умолчанию:
  GameMode:= gmBlackOnly;
  //сетка 5 х 5 клеток:
  Prepare(5);
end;
```

Каждый раз, когда с формой будут происходить метаморфозы, мы будем переносить картинку-фон на канву формы:

```
//ОТРИСОВАТЬ ФОРМУ
procedure TForm1.FormPaint(Sender: TObject);
begin
  Canvas.StretchDraw(Rect(0,0,ClientWidth,ClientHeight), Bitmap);
end;
```

Закрывая программу, мы должны убраться культурно, то есть без шума и пыли:

```
//УНИЧТОЖИТЬ ФОРМУ
procedure TForm1.FormDestroy(Sender: TObject);
begin
 Bitmap.Free;
 Timer1.Enabled:= False;
end;
```

И вот наша форма - во всей красе (Рис. П4.2).

На ней уже порезвился незадачливый кандидат в космонавты, поэтому часть клеток позеленела. Готовьтесь к этому и вы (к тестированию, а не к позеленению), а пока продолжим подготовку к полёту.

Дважды щёлкните на компоненте **PopupMenu1** – раскроется *Редактор меню*. Нажмите также кнопку **F11**, чтобы появился и *Инспектор объектов*. Последовательно вводите пункты меню:

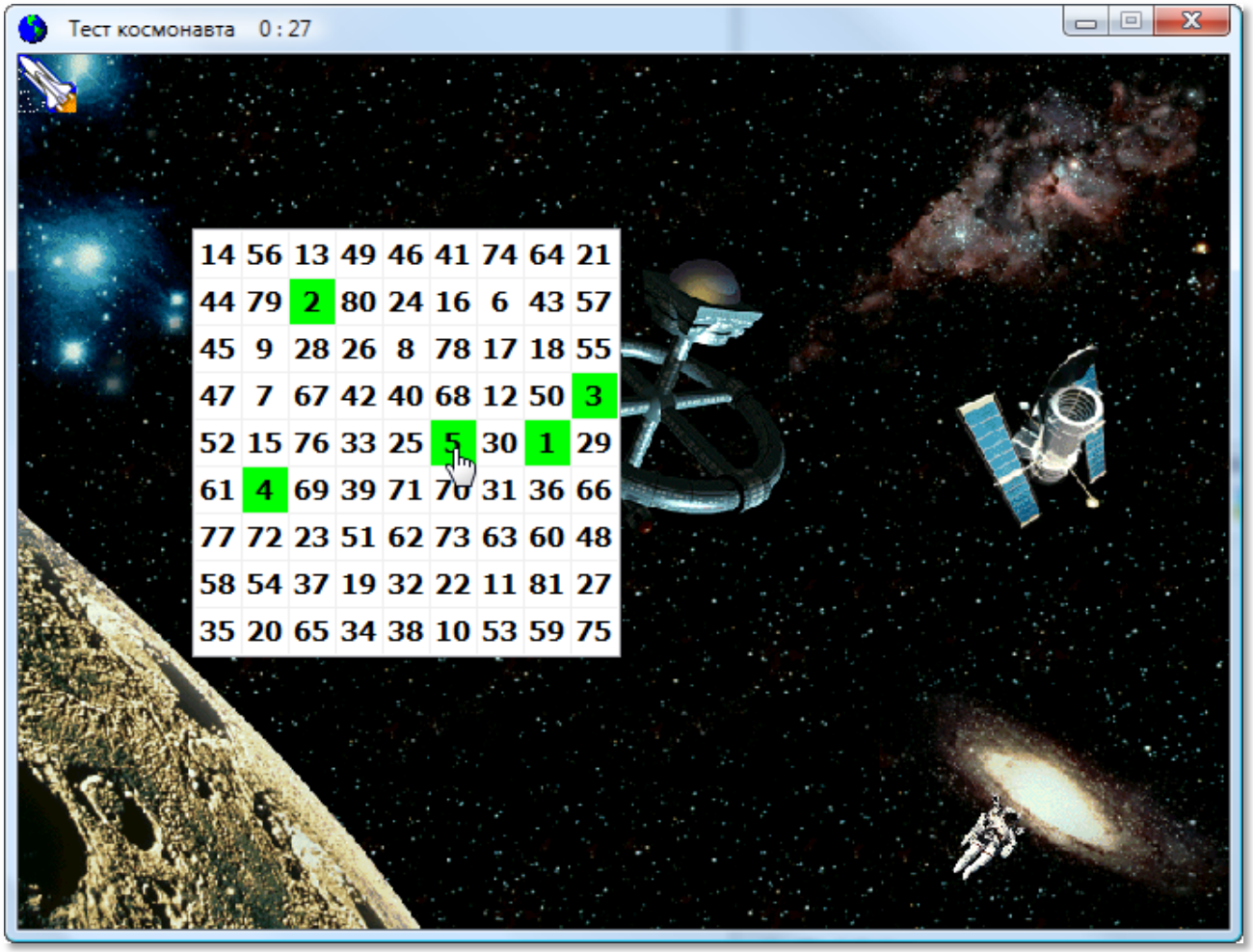


Рис. П4.2. Таких не берут в космонавты!

```
Caption=' СТАРТ'
Name=miStart
Shortcut=Ctrl+S
Caption=' СТОП'
Name=miStop
Shortcut=F2
Caption='-'
Caption=' 5 x 5'
Name=mi5x5
Shortcut=F5
Checked=True (размер поля по умолчанию)
GroupIndex=1
Radioitem=True
Tag=5
Caption=' 6 x 6'
Name=mi6x6
Shortcut=F6
GroupIndex=1
Radioitem=True
Tag=6
Caption=' 7 x 7'
```

```
Name=mi7x7
Shortcut=F7
GroupIndex=1
Radioitem=True
Tag=7
Caption=’ 8 x 8’
Name=mi8x8
Shortcut=F8
GroupIndex=1
Radioitem=True
Tag=8
Caption=’ 9 x 9’
Name=mi9x9
Shortcut=F9
GroupIndex=1
Radioitem=True
Tag=9
Caption=’-’
Caption=’ Только чёрные’
Name=miBlackOnly
Shortcut=Ctrl+B
Checked=True (игра по умолчанию)
GroupIndex=2
Radioitem=True
Caption=’ Чёрные и красные’
Name= miBlackRed
Shortcut=Ctrl+R
GroupIndex=2
Radioitem=True
Tag=1
Caption=’-’
Caption=’ Отмечать клетки’
Name= miGreen
Shortcut=Ctrl+O
GroupIndex=3
Radioitem=False
Caption=’ Правила’
Name= miHelp
Shortcut=Ctrl+P
GroupIndex=4
Radioitem= False
```

Несомненно, меню украсят картинки (их размер должен быть 16 x 16 пикселей) для каждого пункта. Загрузите их, дважды кликнув на свойстве **Bitmap**. Правда, никаких изменений вы не увидите - вот так выглядит всплывающее меню в *Редакторе* (Рис. П4.3).

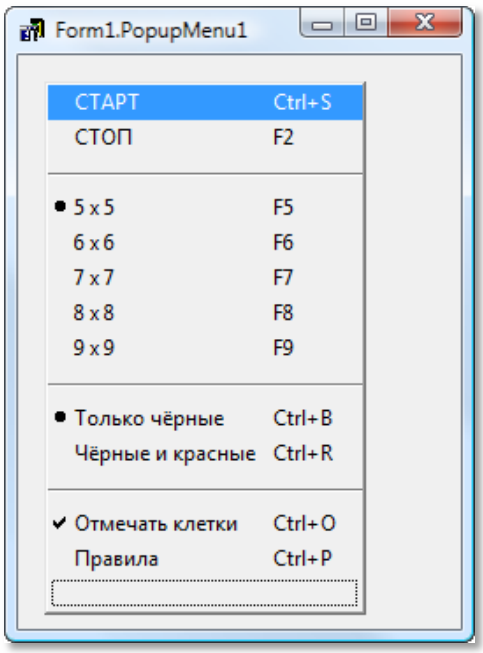


Рис. П4.3. Редактируем меню

А вот так – в работающей программе (Рис. П4.4).

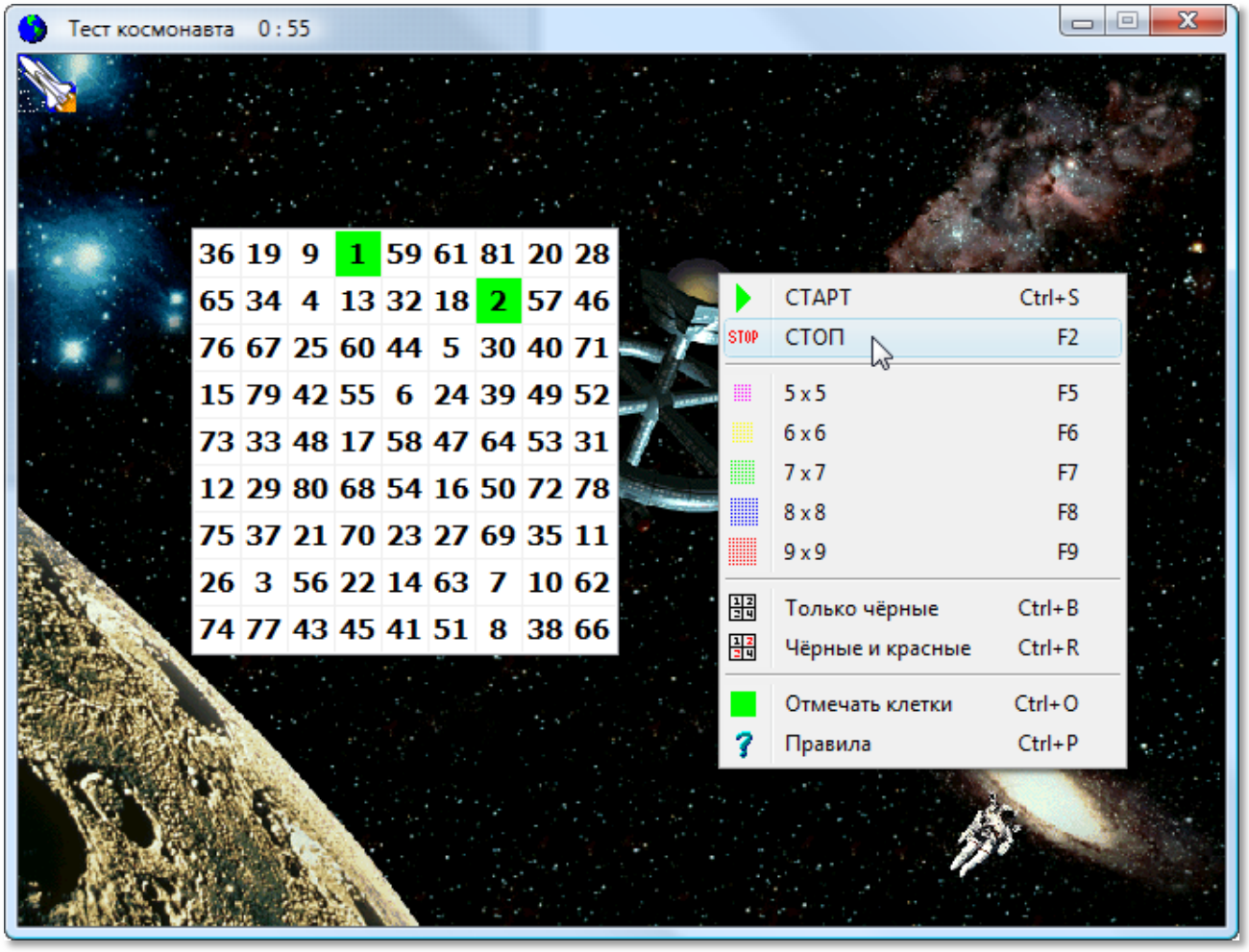


Рис. П4.4. Как хотите, но так – лучше!

Чтобы это меню появилось, достаточно нажать **правую** кнопку мыши в любом месте формы. После выбора нужного пункта меню снова исчезнет в пучине космоса.

Запускайте программу, щёлкайте на здоровье пункты как орешки, но помните: ядрышек в них нет, пустые они. А надо чтобы были полные!

Возвращайтесь ни с чем в режим проектирования и теперь уже щёлкайте по-настоящему! Пойдём сверху вниз, от пункта **СТАРТ**. После того как вы на нём кликнете, откроется заготовка процедуры, которую следует до краёв наполнить кодом. В первую очередь заявим всем остальным пунктам меню, что игра началась, и они могут покурить (конечно, космонавты курят до старта, но пунктам в космос не лететь). Затем выводим новое задание, включаем таймер и засекаем время: вам не до перекуров, жмите на газ обеими ногами!

```
//СТАРТ!
procedure TForm1.miStartClick(Sender: TObject);
begin
  if GameStatus= PLAY then exit;
  GameStatus:= PLAY;

  //вывести новое задание:
  Prepare(dgPole.ColCount);

  //включить таймер:
  Timer1.Enabled:= True;

  //засечь время начала тестирования:
  time0:=GetTickCount();
end;
```

Если дела в космосе не заладились, включайте торможение и надейтесь на мягкую посадку.

```
//СТОП!
procedure TForm1.miStopClick(Sender: TObject);
begin
  //выключить таймер:
  Timer1.Enabled:= False;
  GameStatus:= STOP;
end;
```

Для асов космического пилотажа заготовим неожиданный подвох в виде глобального изменения комфорта. Выбрав соответствующий пункт в меню, мы получим по носу – поле вырастет быстрее, чем нос у обманщика-Буратино:

```
//ЗАДАТЬ РАЗМЕРЫ ИГРОВОГО ПОЛЯ
procedure TForm1.mi5x5Click(Sender: TObject);
begin
  if GameStatus= PLAY then exit;
  (Sender as TMenuItem).Checked:= True;
  Prepare((Sender as TMenuItem).Tag);
end;
```

Чтобы всегда попадать в этот обработчик, все события **OnClick** пунктов меню, изменяющих размеры поля, свяжите с процедурой **mi5x5Click**. Так как свойству **Tag** этих пунктов мы уже присвоили значения размеров поля, то достаточно просто передать его в процедуру **Prepare**, которую мы научили работать.

Можно ещё больше отравить себе жизнь, если выбрать игру с чёрными и красными числами – вот уж где у вас зарябит в глазах до слёз!

В остальном действуйте так же, как с предыдущими пунктами, то есть укажите **miBlackRed** верный путь:

```
//ВЫБРАТЬ ВИД ИГРЫ: ТОЛЬКО ЧЁРНЫЕ ЧИСЛА - ЧЁРНЫЕ И КРАСНЫЕ ЧИСЛА
procedure TForm1.miBlackOnlyClick(Sender: TObject);
begin
  if GameStatus= PLAY then exit;
  (Sender as TMenuItem).Checked:= True;
  GameMode:= TGameMode((Sender as TMenuItem).Tag);
  Prepare(dgPole.ColCount);
end;
```

Люди, предрасположенные к неожиданному мазохизму, могут в ходе поисков полезных ископаемых из космоса отключать (но и включать тоже!) закрашивание уже «готовых» клеток и плутать в лабиринтах чисел без всякой надежды:

```
//ОТМЕЧАТЬ КЛЕТКИ ЗЕЛЁНЫМ ЦВЕТОМ
procedure TForm1.miGreenClick(Sender: TObject);
begin
  miGreen.Checked:= not miGreen.Checked;
  dgPole.Repaint;
  dgPole.Invalidate;
end;
```

Дайте себе также возможность козырнуть гуманизмом и напишите **правила игры**. Для этого достаточно прицепить к проекту форму помощи из игры **Mind Game**, заменив свойство **Lines** компонента **Memo** на более подобающее торжественности момента:

Правила игры:

Эта программа является упрощённой версией известного Теста космонавта и позволяет в игровой форме проверить ваши возможности к концентрации и переключению внимания.

Суть теста состоит в выборе последовательных чисел на квадратном игровом поле, начиная с 1. Сделать это вы должны как можно быстрее.

Игра имеет две разновидности. В первой - все числа чёрные, во второй - чёрные и красные. Если на поле числа разного цвета, то сначала выбирается чёрное число, а затем такое же красное.

Клетки с правильно выбранными числами окрашиваются в зелёный цвет, что облегчает нахождение следующих чисел. Но вы можете отключить эту подсказку. Для этого нажмите правую кнопку мыши и в появившемся меню нажмите пункт "Отмечать клетки". Повторным нажатием вы вернёте выделение клеток.

В меню вы можете также изменить размеры поля и выбрать вид игры.

Тестирование начинается после нажатия на пункт меню "СТАРТ", остановить его можно, кликнув на пункте "СТОП". Время игры выводится в заголовке формы.

Дерзайте - космос ждёт вас!

```
//ПОКАЗАТЬ ПРАВИЛА ИГРЫ
procedure TForm1.miHelpClick(Sender: TObject);
begin
  frmHelp.Show{Modal}
end;
```

Осталось помянуть добрым словом нашу «секретную» кнопку **sbtPlay**, затаившуюся где-то на краю Галактики (**Cursor**=crHandPoint, **Flat**=True, о свойстве **Glyph** позаботьтесь сами, подобрав картинку и впечатав её в фон). А заботы у неё самые простые – она дублирует действие пункта меню **miStart** (эксклюзив для мягкотелых сибаритов и диванных обломовых), поэтому её событие **OnClick** нужно связать с обработчиком **miStartClick**. Для потехи можете использовать эту кнопку и не по назначению.

Счастливые часов не наблюдают, но наше счастье – впереди, поэтому добросовестно считаем все секунды, затраченные на поиски виртуальных удовольствий:

```
//ВЫВОДИМ ВРЕМЯ ИГРЫ И ПРОВЕРЯЕМ, НЕ ЗАКОНЧЕНА ЛИ ОНА
procedure TForm1.Timer1Timer(Sender: TObject);
var
  s: string;
  min, sec: integer;
  i, j: integer;
  flg: boolean;
begin
  //время игры:
  time:=(GetTickCount()-time0)/1000;
  min:= Round(time) div 60;
  if min> 59 then begin //- ваш час пробил!
    Timer1.Enabled:= False;
    s:= 'Ваше время закончилось!'#13#10'Вам ещё рано в
космонавты!';
    Application.MessageBox(pChar(s), NAME_PROG, IDOK);
    GameStatus:= STOP;
    exit
  end;

  //вывести время в заголовок формы:
  sec:= Round(time) mod 60;
  s:= inttostr(sec);
  if sec< 10 then s:= '0' + s;
  s:= inttostr(min)+ ' : ' + s;
  caption:= NAME_PROG + s;

  //не решена ли задача?
  flg:= true;
  for j:= 0 to dgPole.ColCount-1 do
    for i:= 0 to dgPole.ColCount-1 do
      if masPole[i,j].StatusCell<> scGreen then begin
        flg:= false;
        break
      end;
  if flg= true then begin
    Timer1.Enabled:= False;
    s:= 'Поздравляем!'#13#10'Вы справились с заданием!';
    Windows.MessageBox(Application.Handle, pChar(s), NAME_PROG,
MB_OK);
    GameStatus:= STOP;
  end;
end; //Timer1Timer
```

Тут же проверяем все клетки поля и, если найдём хотя бы одну не зелёную (по статусу, а не по цвету!), то ваше время ещё не пришло, иначе – вас поздравят, всё культурно.

Плавно переходим к самотестированию. Щёлкаем в меню на строке **СТАРТ** (или нажимаем секретную кнопку) – и время пошло. Найдите в таблице чёрную единицу и кликните на ней мышкой. Если вы это не отменили, то клетка окрасится в зелёный цвет. Далее продолжайте эту процедуру до полной виктории!

```
//ВЫБРАТЬ ЧИСЛО
procedure TForm1.dgPoleMouseDown(Sender: TObject; Button: TMouse-
Button; Shift: TShiftState; X, Y: Integer);
var
  ACol,ARow: integer;
  n: integer;
begin
  if GameStatus<> PLAY then exit; //- уже играем!

  //координаты мыши:
  dgPole.MouseToCell(x,y,ACol,ARow);
  //число в клетке:
  n:= masPole[ACol,ARow].num;
  case GameMode of
    //только чёрные числа:
    gmBlackOnly: begin
      if (n <> succ(ReadyNumbers)) then //- не следующее число!
        MessageBeep(MB_ICONASTERISK)
      else begin
        masPole[ACol,ARow].StatusCell:= scGreen;
        //нашли ещё одно число:
        inc(ReadyNumbers);
        //пикнуть:
        MessageBeep($FFFFFFFF)
      end;
    end;
    //чёрные и красные числа:
    gmBlackRed: begin
      n:= n*switch;
      if (switch= 1) and (n = succ(ReadyNumbers)) then begin
        masPole[ACol,ARow].StatusCell:= scGreen;
        inc(ReadyNumbers);
        //переключиться на другой цвет:
        switch:= - switch;
        //щелчок:
        MessageBeep($FFFFFFFF);
        exit;
      end
      else if (switch= -1) and (n = ReadyNumbers) then begin
        masPole[ACol,ARow].StatusCell:= scGreen;
        switch:= - switch;
        //щелчок:
        MessageBeep($FFFFFFFF);
      end
      else //- неверное число!
```

```
      //изрядно пикнуть:
      MessageBeep(MB_ICONASTERISK);
    end;
  end;
end; //dgPoleMouseDown
```

Даже при выключенной закраске клеток вы сможете узнать о правильности своих действий по звуковым сигналам.

Так как выбранные клетки отмечаются в массиве **masPole**, то необходимо в процедуре **dgPoleDrawCell** закрасить клетки соответственно их цвету, а также впечатать число – чёрным цветом, если оно положительное, или красным, если отрицательное.

```
//ОТРИСОВАТЬ КЛЕТКУ ИГРОВОГО ПОЛЯ
procedure TForm1.dgPoleDrawCell(Sender: TObject; ACol, ARow: Inte-
ger;  Rect: TRect; State: TGridDrawState);
var
  sNum: string;
  n: integer;
begin
  //цвет клетки: белый, если число ещё не выбрано,
  //зелёный - если выбрано и разрешено отмечать клетки
  if masPole[ACol, ARow].StatusCell= scWhite then
    dgPole.Canvas.Brush.Color:= clWhite
  else
    if miGreen.Checked then dgPole.Canvas.Brush.Color:=
RGB(0,255,0);

  dgPole.Canvas.FillRect(Rect);
  //вывести число в клетке (ACol, ARow) соответствующим цветом:
  n:= masPole[ACol, ARow].num;
  dgPole.Canvas.Font.Size:= 11;
  dgPole.Canvas.Font.Style:= dgPole.Font.Style+ [fsBold];
  dgPole.Canvas.Font.Color:= clBlack;
  if (GameMode= gmBlackRed) and(n < 0) then begin
    dgPole.Canvas.Font.Color:= clRed;
    n:= abs(n);
  end;

  sNum:= inttostr(n);
  with Rect, dgPole.Canvas do
    textrect(Rect, left+(right-left-textwidth(sNum)) div 2,
top+(bottom-top-textheight(sNum)) div 2, sNum);

end; //dgPoleDrawCell
```



Если вам станет скучно играть в одиночку, вызовите на соревнование родных и близких. Для увековечивания своих подвигов заве-

дите таблицу рекордов, где им всем сможете дать своё имя! Как это сделать, смотрите в программе *Тетрис*. И ещё один **полезный совет**: отыскивая числа, запоминайте, где находятся следующие, а то придётся вам клевать по зёрнышку – а время-то идёт!

Нужно сильно исхитриться, чтобы из этой немудрёной программы извлечь головоломку, а вот интересное задание для детей сделать можно. Просто перенесите на бумагу один из тестов, выдаваемых программой. Решать его нужно практически так же, как и на компьютере: быстро-быстро вычёркивать числа в заданной последовательности.



Исходный код программы находится в папке **PanelForm/testcosm**.

Занятие 5. ДОСКОнально о сетке, или Занимательное полеводство

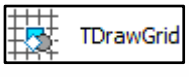
В трёх последних программах роль поля сыграл компонент **TDrawGrid** (сетка, таблица). И в наших следующих игрушках он ещё не раз нам потребуется. Вообще, удивительно, сколько игр и головоломок (в том числе и самых знаменитых, например кроссворды или шахматы) используют поле (или доску), разделённую на клетки!

Конечно, можно вместо **TDrawGrid** установить на форме компонент **TImage** и нарисовать клетки на нём. Но в этом случае программисту придётся значительно больше заниматься вычислениями координат точек на плоскости и вычерчиванием линий. Поэтому, если вы не предполагаете применять в программе суперспецэффекты, то предпочтительно использовать именно сетку **TDrawGrid**, которая имеет практически всё необходимое для создания полноценного игрового поля. С некоторыми из достоинств сетки мы уже познакомились, программируя *Mind Game, Tetris и Тест космонавта*. А в этой главе мы систематизируем наши знания об этом компоненте и познакомимся с другими возможностями сетки. Это пригодится нам при создании новых игр и головоломок. В одной книге, разумеется, невозможно охватить все игры на клеточном поле. Мы подробно рассмотрим только несколько, а остальным – придётся подождать.

Поле в клеточку, или Компонент *TDrawGrid*

Компоненты **TForm** и **TDrawGrid** - дальние родственники, ибо они имеют общего предка – **TWinControl**, базовый класс для всех оконных управляющих элементов (компонентов), от которого они унаследовали много общих черт (свойств, событий и методов). Мы не будем описывать их второй раз, так как они достаточно освещены в главе, посвящённой формам. А из специфических свойств самой сетки мы остановимся только на тех, которые могут быть полезны головоломному программисту.

Свойства сетки

Значок компонента **TDrawGrid** - 

- находится в *Палитре компонентов*, на странице *Additional*. Если щёлкнуть на нём два раза, то сетка появится в центре формы со свойствами, установленными по умолчанию: (Рис. П5.1).

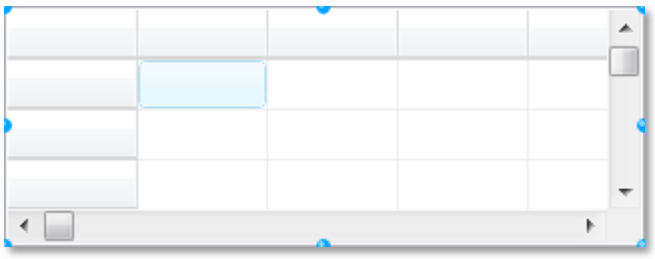


Рис. П5.1. Компонент-сетка

Положение сетки на форме, как и любого другого компонента, определяется свойствами **Left** и **Top**, которые задают координаты верхнего левого угла сетки в пикселях. Размеры компонента (видимая часть сетки) хранят свойства **Height** (высота в пикселях) и **Width** (ширина). Их можно установить в *Инспекторе объектов* или во время выполнения программы.

Слева и сверху находятся **фиксированные клетки** (на рисунке они имеют серый цвет), которые не перемещаются при прокручивании (скроллинге) сетки в соответствующем направлении. Обычно их используют при оцифровке сетки. Например, в календаре на них располагают названия дней недели, а при программировании шахмат на них можно нанести буквы и цифры, обозначающие горизонтальные и вертикальные ряды. Образец такого применения фиксированных клеток вы найдёте в этой главе.

Число колонок с фиксированными клетками зависит от значения свойства **FixedCols**. По умолчанию сетка имеет только одну такую колонку. Если вам не нужны фиксированные колонки, уберите их, установив свойство **FixedCols** в ноль. С другой стороны, вы можете и увеличить их число, но при этом помните, что в сетке должна остаться хотя бы одна колонка обычных клеток (то есть **FixedCols < ColCount**).

Свойство **FixedRows** отвечает за количество фиксированных строк в сетки. К нему относятся все замечания по предыдущему свойству.

Аналогично, общее число колонок и строк в сетке (обычных и фиксированных) задаётся свойствами **ColCount** и **RowCount**. Их может быть сколько угодно, но не менее одной колонки и одной строки.

Вид фиксированных клеток и обрамления сетки зависит от свойства **Ctl3D**. Если оно равно True, то они выглядят объёмными (имеют скосы), если **False** – плоскими. Здесь же можно упомянуть добрым словом свойство **BorderStyle**, которое придаёт сетке законченный вид, оконтуривая её тонкой линией. В зависимости от ситуации вы можете выбирать для сетки любой вид.

Свойства **Color** и **FixedColor** типа **TColor** назначают цвет сетке и её фиксированным элементам. По умолчанию свойство **Color** имеет значение **clWindow** (цвет фона *Windows*), а свойство **FixedColor** – значение **clBtnFace** (цвет кнопок). Их легко изменить так, чтобы они больше соответствовали замыслу программы. Вспомните, что в *Тесте космонавта* мы устанавливали цвет сетки чёрным, а в *Mind Game* – синим. К сожалению, цвет линий сетки изменить нельзя – они всегда серые (**clSilver**), кроме случая, когда такой цвет вы установите и для сетки. Тогда, чтобы линии не сливались с фоном, их цвет меняется на тёмно-серый (**clGray**).

Правда, вы легко можете вообще убрать линии из сетки, присвоив свойству **GridLineWidth** значение **0** (есть и другой способ, о котором рассказано ниже). По умолчанию толщина линий между ячейками сетки равняется 1 пикселю. Но вы имеете право увеличивать её сколько захотите, однако при значениях, превышающих 3 пикселя, сетка будет выглядеть не лучшим образом.

Ширина **каждого** столбца сетки в пикселях зависит от значения свойства **DefaultColWidth**. По умолчанию оно равняется **64**, что довольно много для большинства программ. Переустановить это свойство можно в *Инспекторе объектов* или во время выполнения программы. Но ширина всех столбцов вовсе не обязана всегда быть одинаковой – вы можете изменить размеры любого столбца во время выполнения программы, если потянете мышкой границу столбца в нужном направлении (для этого нужно установить параметр **goColSizing** в True). В играх этого, кажется, делать не нужно, но вдруг круг ваших интересов простирается значительно дальше!
Высота всех столбцов на этапе конструирования формы определяется значением свойства **DefaultRowHeight**, к которому вы можете отнести, с необходимыми поправками, всё сказанное о предыдущем свойстве.

Настройка параметров сетки возможна через свойство **Options** типа *TGridOptions*. Это свойство представляет собой множество, состоящее из следующих элементов:

goFixedVertLine	- между фиксированными столбцами выводятся разделительные вертикальные линии.
goFixedHorzLine	- между фиксированными строками выводятся разделительные горизонтальные линии
goVertLine	- между столбцами выводятся разделительные вертикальные линии.
goHorzLine	- между строками выводятся разделительные горизонтальные линии.
goRangeSelect	– пользователь может выделить прямоугольную область ячеек сетки. Если свойство Options включает **goEditing**, то значение **goRangeSelect** игнорируется.

goDrawFocusSelected	– клетка, имеющая фокус ввода, выделяется цветом (**clHighlight**), если **goDrawFocusSelected** имеет значение **True**. В противном случае клетка выделяется только точечным прямоугольником ввода, без изменения цвета фона.
goRowSizing	- пользователь может с помощью мыши изменять высоту строк сетки.
goColSizing	- пользователь может с помощью мыши изменять ширину колонок сетки.
goRowMoving	- пользователь может с помощью мыши перемещать (изменять их порядок) строки сетки.
goColMoving	- пользователь может с помощью мыши перемещать колонки.
goEditing	– пользователю разрешается редактировать содержимое ячеек сетки. Если **goEditing = True**, то **goRangeSelect** не действует.
goTabs	– для перемещения по клеткам сетки можно использовать клавиши **Tab** и **Shift+Tab**.
goRowSelect	- выделяется вся строка, а не отдельная клетка. При установке g**oRowSelect** в **True goAlwaysShowEditor** игнорируется.
goAlwaysShowEditor	– сетка не блокирует режим редактирования. Если **Options** не включает **goEditing**, то **goAlwaysShowEditor** не действует. Если **Options** включает **goRowSelect**, то **goAlwaysShowEditor** также не действует.
goThumbTracking	- если этот элемент включён в Options, то сетка постоянно обновляется при её прокрутке скроллерами, иначе сетка будет отрисована только по окончании прокрутки.

Если свойство **DefaultDrawing** равно True (значение по умолчанию), то сетка автоматически отображает ячейки, закрашивает их цветом фона, придаёт фиксированным клеткам объемный вид (если установлено свойство **Ctl3D**) и рисует прямоугольник выделения вокруг выбранной ячейки. Но программист может создать и собственную процедуру отображения сетки. Она может понадобиться, когда ячейки выделяются цветом, в неё выводится текстовая (буквы, цифры) или графическая (небольшие картинки) информация. Именно это нам и приходилось делать в программах *Тест космонавта*, *Тетрис* и *Mind*. В этом случае необходимо свойство **DefaultDrawing** установить в False, тогда при рисовании сетки будут изображаться **только** линии (если не отменено их отображение), а содержимое ячеек выводиться не будет. Но при отрисовке любой ячейки сетки возникает событие **OnDrawCell**, в обработчике которого и следует поместить все операции по выводу содержимого ячеек. Так как это весьма сильнодействующее средство для создания эффектного графического интерфейса, то мы разберём его более подробно чуть позже, хотя немало примеров использования события **OnDrawCell** в «корыстных целях» вы можете найти во многих программах, рассматриваемых в этой книге.

Если размеры компонента **DrawGrid** (значения свойств **ClientHeight** и **ClientWidth**) не позволяют видеть целиком все ячейки сетки, то справа и/или снизу добавляются полосы прокрутки. По умолчанию разрешены оба скроллера – свойство **ScrollBars** = ssBoth. Вы их можете видеть на картинке в начале главы. Вы можете запретить один или оба скроллера, выбрав для этого свойства соответствующее значение, но тогда вам самим придётся позаботиться о прокрутке сетки, если не все её ячейки видны на экране. Это довольно хлопотно, но иногда бывает полезно. Один из способов реализации этой затеи вы можете найти в главе, посвящённой японским кроссвордам.

Мы разобрали все значимые свойства сетки, которые можно устанавливать в *Инспекторе объектов*. Но у компонента **DrawGrid** имеются и такие свойства, которые доступны только во время выполнения программы. Среди них мы можем найти и очень полезные.

- Свойство **Col** позволяет определить номер текущего столбца (того, в котором находится выделенная ячейка), имейте в виду, что нумерация начинается с **0**. Узнать номер текущей строки вы можете, считав значение свойства **Row**.

- Свойство **ColWidths**[Index: Longint]: Integer содержит ширину в пикселях столбца с заданным номером **Index**. Если ширина столбцов не изменялась с помощью мыши или установки новых значений свойству **ColWidths**, то все они имеют одинаковую ширину, равную **DefaultColWidth**. Общее число элементов в массиве равно **ColCount** (нумерация начинается с **0**).

- Аналогично вы можете использовать для чтения и установки высоты любой строки сетки свойство **RowHeights**[Index: Longint]: Integer.

- Свойство **EditorMode**: Boolean управляет редактированием содержимого ячеек.

- Свойство **GridHeight**: Integer определяет высоту сетки в пикселях. Если значение этого свойства меньше высоты клиентской области **ClientHeight**, то все строки сетки будут видны на экране, а под ними будет пустое пространство.

- Свойство **GridWidth** позволяет узнать высоту сетки.

- В свойстве **LeftCol**: Longint хранится номер первой (левой) видимой колонки сетки (не считая фиксированных). Это свойство доступно

как для чтения, так и для записи, поэтому вы можете использовать его для горизонтальной прокрутки сетки, как это сделано, к примеру, в программе *Nippon* (*Японский кроссворд*).

- Свойство **TopRow**: Longint содержит номер первой (верхней) видимой строки сетки.

- Свойство **VisibleColCount**: Integer показывает число **целиком** видимых колонок сетки, исключая фиксированные, число которых равно **FixedCols**.

- Свойство **VisibleRowCount**: Integer показывает число **целиком** видимых строк сетки, исключая фиксированные, число которых равно **FixedRows**.

- С помощью свойства **Selection**: TGridRect вы можете выделить прямоугольную область ячеек в сетке (если **goRangeSelect**= True, иначе – только одну ячейку). Вы можете задать эту область двумя способами:

```
type TGridRect = record
                  case Integer of
                  0: (Left, Top, Right, Bottom: Longint);
                  1: (TopLeft, BottomRight: TGridCoord);
             end;

type TGridCoord = record
                 X: Longint;
                 Y: Longint;
             end;
```

То есть, устанавливая нужные значения для левого столбца, верхней строки, правого столбца, нижней строки, либо указывая координаты верхнего левого и нижнего правого углов прямоугольника.

Замечательное свойство **Canvas**, благодаря которому можно закрашивать ячейку любым цветом (см. *Тест космонавта*), выводить в неё текст (см. там же) и рисовать картинки (см. *Mind Game* и другие программы) будет разобрано по косточкам в следующей главе. Как говорится, не пропустите!

Без картинок жизнь скучна, поэтому представим некоторые свойства сетки в наглядной форме (Рис. П5.2).

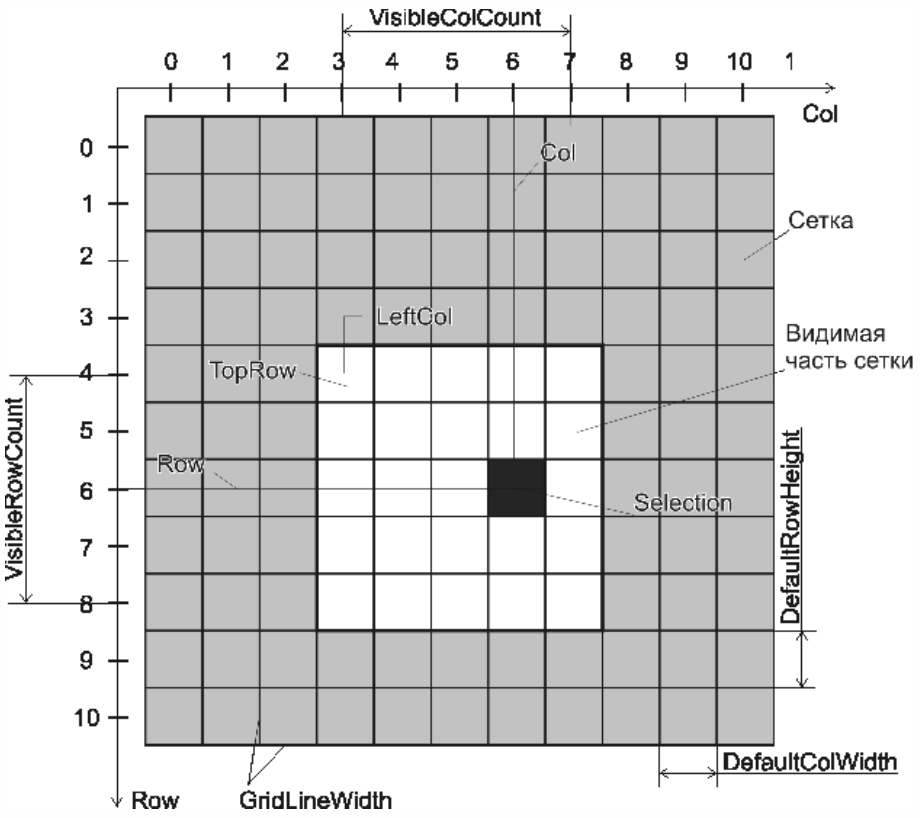


Рис. П5.2. Анатомия сетки



Если сетка имеет *фиксированные* ряды, то они всегда, – может быть, частично – видимы. На рисунке они отсутствуют, чтобы не загромождать его несущественными подробностями. То же самое относится и к полосам прокрутки, которые автоматически добавляются к сетке (если они не отменены), когда размеры клиентской части компонента меньше размеров всей сетки.

События сетки

Если свойство **Options** содержит **goColMoving**, то пользователь может перемещать столбцы сетки с помощью мыши (или программно). Сразу после этого генерируется событие:

```
OnColumnMoved: TMovedEvent.
type TMovedEvent = procedure (Sender: TObject; FromIndex,
ToIndex: Longint) of object;
```

Где **FromIndex** – прежний номер колонки,
ToIndex – новый номер колонки.

- Подобным же образом вы можете использовать и событие **OnRowMoved**: **TMovedEvent**.

- Событие **OnTopLeftChanged**: TNotifyEvent возникает сразу после изменения свойств **TopRow** или **LeftCol**.

- Событие **OnSelectCell**: TSelectCellEvent возникает перед выделением ячеек сетки.

- Событие **OnEnter**: TNotifyEvent возникает, когда сетка получает фокус ввода, а **OnExit**: TNotifyEvent – когда он переходит к другому элементу управления.

- Событие **OnDrawCell**: TDrawCellEvent – одно из самых полезных при программировании игр. Оно даёт возможность выводить в ячейки сетки текстовую и графическую информацию. Оно было разобрано выше, поэтому – не будем повторяться.

- Очень часто в играх используется событие **OnMouseDown**: TMouseEvent, которое возникает при нажатии кнопки мыши. При этом в процедуру-обработчик передаются координаты (на экране) мыши в пикселях:

```
type TMouseEvent = procedure (Sender: TObject; Button: TMouseButton; Shift: TShiftState; X, Y: Integer) of object.
```

По ним с помощью метода

```
MouseToCell(X, Y: Integer; var ACol, ARow: Longint);
```

вы можете легко определить колонку **ACol** и строку **ARow** той ячейки сетки, в которой находился курсор мыши при нажатии кнопки. Посмотреть, как это делается, можно в игре *Mind Game* и во многих других программах.

Чуть реже применяются события мыши **OnMouseMove** и **OnMouseUp**. Так как они совершенно аналогичны событию **OnMouseDown**, то не будем на них тратить время попусту. А найти примеры вы можете в программе *Tempuc* и *Logos*.

Если вы хотите самостоятельно организовать ввод текстовой информации в ячейки сетки, то вам будут необходимы обработчики событий **OnKeyDown** и **OnKeyUp** типа **TKeyEvent** и **OnKeyPress** типа **TKeyPressEv-**

ent (имейте в виду, что клавиши со стрелками используются компонентом **TDrawGrid** для прокрутки сетки и выделения ячеек). Пример такого ввода вы можете увидеть в программе *Nippon*.

Коды виртуальных клавиш, используемые в процедурах обработки нажатия и отпускания клавиш:

VK_CANCEL	CTRL+BREAK
VK_BACK	BACKSPACE
VK_TAB	TAB
VK_RETURN	ENTER
VK_SHIFT	SHIFT
VK_CONTROL	CTRL
VK_MENU	ALT
VK_PAUSE	PAUSE
VK_CAPITAL	CAPS LOCK
VK_ESCAPE	ESK
VK_SPACE	SPACE
VK_PRIOR	PAGE UP
VK_NEXT	PAGE DOWN
VK_END	END
VK_HOME	HOME
VK_LEFT	LEFT ARROW
VK_UP	UP ARROW
VK_RIGHT	RIGHT ARROW
VK_DOWN	DOWN ARROW
VK_PRINT	PRINT
VK_SNAPSHOT	PRINT SCREEN
VK_INSERT	INSERT
VK_DELETE	DELETE
VK_NUMPAD0	0 (числовая панель)
VK_NUMPAD1	1
VK_NUMPAD2	2
VK_NUMPAD3	3
VK_NUMPAD4	4
VK_NUMPAD5	5
VK_NUMPAD6	6
VK_NUMPAD7	7
VK_NUMPAD8	8
VK_NUMPAD9	9

VK_MULTIPLY	*
VK_ADD	+
VK_SUBTRACT	-
VK_DECIMAL	.
VK_DIVIDE	/
VK_F1.. VK_F24	F1.. F24
VK_NUMLOCK	NUM LOCK

Методы сетки

Метод

```
MouseToCell(X, Y: Integer; var ACol, ARow: Longint);
```

обычно используется в обработчиках событий **OnMouseDown**, **OnMouseMove**, **OnMouseUp** для нахождения по координатам мыши *(X, Y)* колонки и строки, в которых находится ячейка сетки с курсором. Технология этого процесса описана выше.

Метод

```
CellRect(ACol, ARow: Longint): TRect;
```

возвращает координаты прямоугольника, описывающего ячейку сетки (если указанная ячейка не видна на экране, то возвращается пустой прямоугольник). Необходимый инструмент по отрисовке ячеек сетки в обработчике события **OnDrawCell**. Используется практически в каждой программе этой книги.

Для перерисовки сетки после вывода сообщений или новой информации используются методы **Invalidate**, **Repaint** и **Update**. Они уже были нами рассмотрены в главе, посвящённой формам. Примеры их употребления вы сможете найти, например, в программах *Тетрис* и *Тест космонавта*.

И последний из «полезных» методов – это **SetFocus**, который устанавливает фокус ввода на сетке. Это необходимо сделать при вводе информации в ячейки сетки с клавиатуры (см. программу *Nippon*).

Игры-оборотни

В этом разделе книги речь пойдёт, разумеется, не о тех играх, в которых фигурируют вурдалаки, докторы хайды, вовкулаки, вервольфы и прочая нечисть. Всё гораздо милее и интереснее – фишки для таких игр окрашены в *два* цвета (например, с одной стороны они белые, с другой – чёрные) и время от времени они *изменяют* свой цвет на противоположный (иначе говоря, *переворачиваются* на другую сторону).

Классическим образцом такой игры является *реверси* (*отелло*), но эта игра заслуживает, конечно, гораздо большего внимания, чем мы можем уделить ей в данной книге. Как ни странно, все другие игры-оборотни значительно менее интересны и известны, поэтому мы займёмся программированием *головоломок*-оборотней. Зато мы убьём ещё парочку зайцев: закрепим на практике знания о компоненте *TDrawGrid* и продолжим совершенствовать мастерство изготовления кнопок из подручного материала.

XorGame, или Как обнулить матрицу

Сказать по правде, так и головоломок-оборотней придумано не очень много. А между тем, как говаривал гроссмейстер товарищ Бендер, вертеть фишки – *плодотворная дебютная идея*! Все известные головоломки отличаются друг от друга в основном правилами переворачивания фишек. Значит, достаточно написать программу для решения одной головоломки, а затем модифицировать её для всех остальных.

А начнём мы, пожалуй, с самого известного «оборотня», который называется *Флип-Флоп* (прояснить сущность этой игры нам поможет англо-русский словарь, в котором вы без труда найдёте и *flip*, и *flop*). Придумали её на фирме *Геймос* (*Gamos*), которая породила множество весьма полезных для ума игр. Эта же головоломка использована и в игре *Братья Пилоты. По следам полосатого слона*, где Шеф и Коллега должны открыть сейф, похожий на холодильник, поворачивая ручки, которые могут занимать вертикальное и горизонтальное положение.



В коллекции игр от *Home Games Center*, версия 10.19 от 2000 г. также имеется эта игра, но в более скромном обличии (Рис. П5.3).



Рис. П5.3. Не все простое гениально!

Здесь предлагается все крестики заменить галочками. Больше ничего нового нет.

Суть игры довольно проста: в некоем доме имеется 16 окон – по четыре на каждом из четырёх этажей. Некоторые окна *открыты*, остальные – *закрыты*. *Ваша цель* – открыть все окна настежь. Для этого вы можете поочерёдно открывать любые закрытые окна (или закрывать открытые). Но – по странной прихоти городского архитектора - все другие окна на том же этаже и в том же подъезде изменят своё состояние на противоположное (Рис. П5.4). Несмотря на «аскетизм» формулировки, задача не очень простая, в чём вы сами можете убедиться, похлопав створками окон.

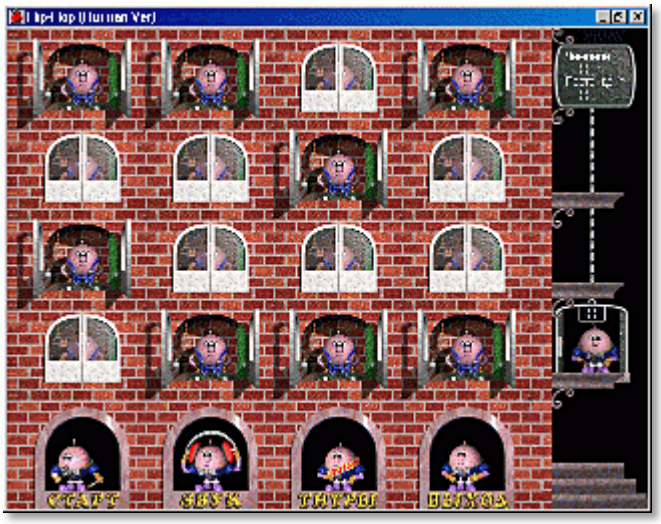


Рис. П5.4. Эх, флип-флопнем и окнами хлопнем!

В этой игре-головоломке нет никаких переворачивающихся фишек, но это ровным счётом ничего не меняет – для нас важно только то, что какие-то объекты могут находиться в двух «противоположных» состояниях. Например, окно может быть либо открыто, либо закрыто, фишка может быть либо чёрной, либо белой, и так далее. Совершенно естественно возникает желание обозначить одно состояние объекта *единицей*, а другое *нулём*, не различая при этом особенностей реализации этих состояний в конкретной программе. А игровое поле удобно описать в виде *двумерного массива*.

Обозначив закрытые окна *единицей*, а открытые – *нулём*, мы можем условие задачи переписать так (Рис. П5.5).

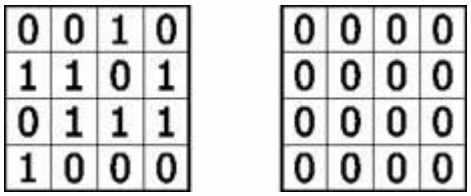


Рис. П5.5. *Начальная* (исходная) позиция - *Конечная* позиция

Сформулируем правила игры в «матричном» виде.

На квадратном игровом поле размером 4 на 4 клетки расположены числа 1 и 0. Задача игрока заключается в том, чтобы за *наименьшее* число ходов перевести *начальную* позицию на игровом поле в *конечную*, состоящую из одних нулей. Чтобы сделать *ход*, нужно любое из чисел на поле заменить противоположным, то есть единицу – нулём, а нуль – единицей. При этом все остальные числа в том же столбце и в той же строке также изменяют своё значение на *противоположное*.

Предварительные изыскания

Все эти размышления мои
до сих пор предварительные были
не более как одною глупостью моею…

Н.Лесков, *Соборяне*

Можно предположить, что и эта и другие задачи в игре *Флип-Флоп* имеют решения. Очевиден ли для вас тот факт, что *любую* начальную позицию можно перевести в конечную? Если это так, то вы смело можете пропустить несколько следующих страниц, на которых мы попробуем убедиться в этом, написав небольшую программу.

Кстати говоря, приём, с помощью которого мы будем перечислять позиции, очень даже пригодится вам при работе над такими играми, как лабиринты, сокобан или *Цветные линии*.

Размеры игрового поля мы обозначим своими именами и сохраним в *константах*:

```
const
  //размеры поля:
  POLE_WIDTH=4;
  POLE_HEIGHT=4;
```

Опишем массив поля в глобальной переменной типа *Byte*, чтобы зря не расходовать память компьютера:

```
var
  //массив поля:
  masPole: array[0..POLE_WIDTH-1, 0..POLE_HEIGHT-1] of byte;
```

Всего нулей и единиц на поле 16 штук, что в точности равняется числу битов в одном двухбайтовом слове, которое имеет тип *Word*. Нам потребуется при расчётах и число, на единицу меньшее этого (попросту – 15). Заведём для него *константу*

```
  //число клеток на поле-1:
  AllCells= POLE_WIDTH*POLE_HEIGHT-1;
```

Легко также определить, что всего различных позиций на поле может быть 2^{16} = 65536. Нам остаётся только закодировать каждую позицию двухбайтовым словом. Для этого перепишем, например, заданную начальную позицию в одну строку: 0010110101111000. Так как каждое из

чисел теперь является битом, то все вместе они составляют двухбайтовое слово. Вы можете записать его и в 16-ричной форме - *$2D78*. Как это делается, мы уже рассматривали в игре *тетрис*. Аналогично кодируется и любая другая начальная позиция, а конечная позиция с одними нулями как раз и описывается нулём.

Мы договорились всякое клеточное поле представлять компонентом *TDrawGrid*, поэтому и поместим его на свежей форме, желательно в верхнем левом углу, чтобы он не портил мизансцену. С установкой свойств сетки (**Name** = dgPole) тоже никаких затруднений не предвидится:

```
ColCount = 4
RowCount = 4
DefaultDrawing = False
FixedCols = 0
FixedRows = 0
OnDrawCell = dgPoleDrawCell (процедуры будут описаны ниже)
OnMouseDown = dgPoleMouseDown
```

Остальные свойства вы можете установить по своему вкусу – они никак не влияют на работоспособность программы (только не забывайте о присущем вам здравом смысле).

Как же нам подсчитать число *разрешимых* позиций, если мы пока ещё не научились толком решать такие задачи? А нужно сделать ловкий ход – воспользоваться *ретроспективным* анализом, то есть рассматривать позиции, начиная с *заключительной* (конечной). К сожалению, объём книги не позволяет подробно остановиться на этом приёме, добавлю только, что его применение бывает очень полезно при решении шахматных и других подобных головоломок (задач).

Одно из самых важных понятий в ретроспективном анализе – *ранг позиции*. Это число ходов, которые нужно сделать из исходной (у нас это конечная) позиции в заданную. Легко сообразить, что нужная нам позиция с одними нулями имеет ранг *0*, так как она получается «сама собой», без всяких ходов. Присвоим также позициям, для которых ранг неизвестен, значение *–1*, чтобы отличать их от уже ранжированных.

Сделав ход из позиции с рангом 0, мы получим позицию с рангом 1. Так как этот ход может быть сделан в любую из 16 клеток, то всего существует 16 разных «предвыигрышных позиций» и столько же «первых» ходов. Давайте для начала научимся делать хотя бы один *ход* в нашей игре-головоломке.

При запуске программы мы инициализируем массив игрового поля, заполняя его нулями, то есть создаём конечную позицию с рангом 0:

```
//СОЗДАТЬ ФОРМУ
procedure TForm1.FormCreate(Sender: TObject);
var i, j: integer;
begin
  for j:= 0 to POLE_HEIGHT-1 do
    for i:= 0 to POLE_WIDTH-1 do
      //очистить клетку:
      masPole[i,j]:= 0;
end;  //FormCreate
```

Как обычно, все изменения в массиве поля нужно отобразить на экране, закрасив каждую клетку поля своим цветом: та, что в массиве обозначена *нулём*, – пусть будет *белой*, *единицей* – *красной* (вы можете выбрать другие цвета):

```
//ОТРИСОВАТЬ ЯЧЕЙКУ СЕТКИ
procedure TForm1.dgPoleDrawCell(Sender: TObject; ACol, ARow: Inte-
ger; Rect: TRect; State: TGridDrawState);
var
  i,j,n: integer;
begin
  //закрасить клетку своим цветом:
  case masPole[ACol, ARow] of
    0: dgPole.Canvas.Brush.Color:= clWhite;
    else dgPole.Canvas.Brush.Color:= clRed;
  end;
  dgPole.Canvas.FillRect(Rect);
end;  //dgPoleDrawCell
```

Чтобы сделать ход, нужно просто щёлкнуть (точнее – нажать кнопку мыши, иначе мы не сможем узнать её координаты) мышкой в любой клетке. С помощью метода сетки *MouseToCell* мы легко определим, в какую клетку поля был сделан ход. Обратите внимание, что столбец и строка для этой клетки возвращаются в переменных *ACol* и *ARow*, которые нужно объявить в процедуре, использующей этот метод. Чтобы иметь возможность вводить *любую* начальную позицию, а не только нулевую, мы нагрузим и *правую* кнопку мыши – нажимая её, мы просто изменяем цвет клетки на противоположный, не затрагивая других клеток поля. А настоящий ход выполняется при нажатии *левой* кнопки мыши. В этом случае мы меняем цвет всех клеток поля, которые лежат в том же столбце и строке, что и «щёлкнутая». Как это сделано, вы без труда сможете разобраться сами (в принципе, это можно сделать и иначе). Ну и наконец, мы обновляем поле на экране, чтобы отразить произошедшие на нём изменения:

```
//СДЕЛАТЬ ХОД
procedure TForm1.dgPoleMouseDown(Sender: TObject; Button: TMouse-
Button; Shift: TShiftState; X, Y: Integer);
var
  ACol,ARow: integer;
  i, j: integer;
begin
  //координаты мыши:
  dgPole.MouseToCell(x,y,ACol,ARow);
  if ssRight in shift then //инвертируем цвет клетки:
    if masPole[ACol,ARow]= 1 then masPole[ACol,ARow]:= 0
    else masPole[ACol,ARow]:= 1
  else if ssLeft in shift then begin //инвертировать ряды
    for j:= 0 to POLE_HEIGHT-1 do
      for i:= 0 to POLE_WIDTH-1 do
        if (i=ACol) or (j=ARow) then
        //инвертировать клетку:
        masPole[i,j]:= abs(masPole[i,j]-1)
  end;
  dgPole.Invalidate;
end;  //dgPoleMouseDown
```

Запустите программу и удостоверьтесь, что она работает правильно. Вы даже можете сыграть в тот же *Флип-Флоп*, предварительно расставив правой кнопкой мыши красные клетки (они соответствуют, если вы помните, *закрытым* окнам). Настоятельно рекомендую вам попробовать решить задачу, приведённую в начале раздела, чтобы потом у вас было с чем сравнивать свои успехи.

Научившись делать ходы, мы возвращаемся к нашей задаче – подсчёту разрешимых позиций в этой головоломке. Теперь мы можем находить все 16 позиций с рангом 1. Взяв любую из них, мы получим позиции с рангом 2, сделав ход в одну из 15 клеток (конечно, нет никакого смысла ходить в ту же клетку, что и перед этим, иначе мы просто вернёмся на 1 ход назад). И вот здесь нас ждут первые неприятности: позиции начнут *повторяться*. Это следствие того, что результирующая позиция не зависит от порядка ходов. Вы легко можете убедиться в этом, если сходите, например, сначала в верхний левый угол, а затем в правый нижний, после чего вернётесь к чистой доске и выполните ходы в обратном порядке. Если мы будем считать *все* получающиеся позиции, то некоторые посчитаем несколько раз, а в таком подсчёте нет никакого смысла. Конечно, можно записывать все ходы и позиции для сравнения, чтобы исключить повторы, но позиций-то многие тысячи, так что нас ждёт удовольствие не из приятных! Будет правильнее поручить ведение протокола компьютеру, он с этим делом легко справится. Но прежде мы должны научить его переходить от одной позиции к другой и записывать нужную нам информацию.

Для анализа головоломки нам нужно запомнить ранг каждой позиции, координаты клетки, в которую был сделан ход, приведший к ней, и предыдущую позицию, чтобы мы могли проследить всю цепочку ходов. Для того чтобы все данные хранились в *одном* месте, мы заведём глобальную переменную *Status,* представляющую собой массив данных о каждой возникающей на поле позиции:

```
Status: array[word] of TStatus; //информация о позиции на поле:
type TStatus= Record
  Range: integer; //ранг позиции - ход, на котором
                  //впервые могла возникнуть данная позиция
  Hod: TPoint;    //координаты клетки поля, на которую сделан ход
  Pred: word;     //предыдущая позиция
end;
```

Алгоритм подсчёта позиций можно представить себе так. Мы знаем, что ранг всех позиций в начале равен –1, так как ни одна из них ещё не возникла на поле (кроме последней, с одними нулями – она получается сразу же, на «нулевом» ходу, и ранг ее равен 0). Из нулевой позиции мы можем сделать 16 ходов и получить все позиции с рангом 1. Естественно, позиция с рангом 2 может возникнуть только в результате хода из позиции с рангом 1, поэтому мы перебираем все возможные позиции и, отыскав ещё не посчитанную (её ранг равен -1), делаем ходы во все клетки поля. Если в результате этого возникает новая позиция, то мы присваиваем ей ранг 2 и записываем всю информацию о ней.

Точно так же мы делаем и следующие ходы (переходим к позициям очередного ранга) – до тех пор, пока на очередном ходе будет найдена хотя бы одна новая позиция.

Хранить число найденных позиций мы будем в *переменной:*

```
  //всего найдено позиций:
  AllPos: cardinal;
```

Так как теперь все ходы за нас будет делать компьютер, то мы должны научить его этому. Результат хода зависит от расстановки «фишек» на поле и от координат клетки, в которую производится ход. Возникающую после хода позицию нам нужно вернуть в процедуру, подсчитывающую позиции, поэтому все премудрости выполнения хода следует поместить в *функцию*. «Ручной» ход мало чем отличается от машинного, единственное отличие – нужно «упаковать» единицы и нули на поле в двухбайтовое слово, которое однозначно описывает позицию.

Делается это так. Находим в двухбайтовом слове *w*, описывающем текущую позицию, те биты, которые соответствуют переворачиваемым столбцу и строке, и инвертируем их с помощью логической операции *xor* (это «магическое» слово и было использовано для именования головоломки). Чтобы подобраться к этой позиции в слове *w*, нужно сдвинуть единицу *влево* в нужную позицию логическим оператором *shl*. Например, если нужно инвертировать третью клетку поля, то следует передвинуть единицу в 13-ю позицию (нумерация битов в двухбайтовом слове такая: 15-14-13-12-11-10-9-8-7-6-5-4-3-2-1-0). В результате мы получим двоичное число 10000000000000, а операция *w xor 10000000000000* как раз и инвертирует нужный бит. Во всех остальных тонкостях этой функции разберитесь сами:

```
//ВЫПОЛНИТЬ ХОД
function XorBits(x, y: byte; w: word): word;
//x, y - координаты хода
//w - биты поля
var
  i, j: integer;
begin
  Result:= w;
  for j:= 0 to POLE_HEIGHT-1 do
    for i:= 0 to POLE_WIDTH-1 do
      if (i=x) or (j=y) then begin
        //инвертировать клетку:
        masPole[i,j]:= abs(masPole[i,j]-1);
        Result:= Result xor (1 shl (AllCells-POLE_WIDTH*j-i));
      end;
end;  //XorBits
```

Для прояснения действия этой функции, мы проведём её проверку на тех ходах, результаты которых легко проверить. Например, мы наизусть знаем все первые ходы из нулевой позиции. Давайте протестируем новую функцию на них. Установим на форме *кнопку sbtTesting*, нажав на которую мы сможем последовательно выполнить все 16 ходов.

Нелишним будет научиться по двухбайтовому слову *w*, описывающему поле, выводить «фишки» на экран . Для этого, начиная со *старшего* бита (соответственно, с самой *первой* клетки поля), мы определяем значение каждого бита слова и заносим его в массив поля. Закончив формирование массива, мы обновляем поле на экране. Обратите внимание на то, как выделяется старший бит слова *w*.

```
//ПОКАЗАТЬ ПОЛЕ
procedure TForm1.ShowPole(w: word);
//w - слово, описывающее позицию
var
```

```
  i, j: integer;
begin
  //занести в массив 1 или 0:
  for j:= 0 to POLE_HEIGHT-1 do
    for i:= 0 to POLE_WIDTH-1 do begin
      if w and $8000 <> 0 then //- клетка фигуры
        masPole[i,j]:= 1
      else
        masPole[i,j]:= 0;     //- клетка "фона"
      w:=  w shl 1;
    end;
  //отрисовать сетку:
  dgPole.Invalidate;
end; //ShowPole
```

Процедура, обрабатывающая нажатие кнопки *sbtTesting*:

```
//ПРОВЕРКА
procedure TForm1.sbtTestingClick(Sender: TObject);
var
  i, j: integer;
  w: word;
begin
  //делаем ходы:
  for j:= 0 to POLE_HEIGHT-1 do
    for i:= 0 to POLE_WIDTH-1 do begin
      w:= XorBits(i,j,0);
      ShowPole(w);
      showmessage (inttostr(w));
    end;  //for i
end; //sbtTestingClick
```

Запустите программу и удостоверьтесь, что, по крайней мере, из нулевой позиции ходы выполняются верно. Вы сможете продлить себе удовольствие, если вместо нулевой позиции возьмёте любую другую и проверите её. Всё это рано или поздно убедит вас, что функция работает правильно в *любой* позиции.

Теперь у нас есть всё необходимое для подсчёта позиций. Установите на форме *кнопку sbtCreateList* и напишите в обработчике её нажатия код:

```
//СОЗДАТЬ СПИСОК ПОЗИЦИЙ В МАССИВЕ
procedure TForm1.sbtCreateListClick(Sender: TObject);
var
  Hod: integer;
  flg: boolean;
  i,j,k,w: word;
begin
  Hod:= 0;
  //если ранг позиции равен -1, то она ещё не возникала на поле:
```

```
  for i:= 0 to 65535    do
    Status[i].range:= -1;

  //начальная позиция - её ранг равен 0:
  Status[0].range:= 0;

  //инициализируем счётчик найденных позиций:
  AllPos:= 1;

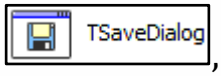
  //делаем следующие ходы:
  repeat
    flg:= False;
    inc(hod);
    listbox1.Items.add(inttostr(hod));
    for k:= 0 to 65535 do //- по всем возможным позициям
      if status[k].range= Hod- 1 then //- нашли позицию, возникшую
на
      begin                              //предыдущем ходу
      //делаем ходы из этой позиции:
      for j:= 0 to POLE_HEIGHT-1 do
        for i:= 0 to POLE_WIDTH-1 do begin
          w:= XorBits(i,j,k);
          if status[w].range= -1 then begin //- новая позиция
            //на этом ходу была найдена по крайней мере 1 позиция:
            flg:= True;
            //записать позицию в массив:
            Status[w].range:= hod;
            Status[w].hod.x:= i;
            Status[w].hod.y:= j;
            Status[w].Pred:= k;
            inc(AllPos);
          end;
        end;
      end;
  until flg= false; //продолжать, пока найдена хотя бы одна
                    //позиция на очередном ходу
```

```
  Showmessage('hodov= '+ inttostr(hod));
  Showmessage('AllPos= '+ inttostr(AllPos));
end;  //sbtCreateListClick
```

После всего содеянного у вас не должно возникнуть вопросов о том, *как* она работает. Поэтому запускайте программу и нажимайте кнопку. Очень скоро вы получите ответ на мучивший нас вопрос: все 65535 позиций (65536-я – это нулевая) «правильные» и для решения любой задачи потребуется не более 16 ходов (не удивляйтесь, что последний ход – 17-й, ведь он не дал ни одной позиции). Способ доказательства, конечно, «топорный», но зато и само доказательство железное!

Итак, мы «напрасно» потратили время – не найдено ни одной позиции, из которой нельзя было бы перейти к нулевой. Правда, мы теперь точно знаем, что на решение задач может потребоваться от 1 до 16 ходов. А это уже кое-что: если вы сделали 16 ходов, а задачу так и не решили, значит, вы сделали *лишние* ходы.

Вот ещё бы узнать, *какие* ходы нужно сделать, чтобы решить задачу за минимальное число ходов!

А ведь у нас уже есть вся необходимая информация для этого! Она хранится в массиве *Status*, вот только пользоваться ей не очень удобно. Поэтому запишем все данные на диск. Для этого установите на форме *компонент TSaveDialog* 

, ещё одну *кнопку – sbtSaveList* и наберите код для неё:

```
//ЗАПИСАТЬ ДАННЫЕ НА ДИСК
procedure TForm1.sbtSaveListClick(Sender: TObject);
var
  F: textfile;
  fn,s: string;
  i,j: integer;
  w: word;
begin
  //открыть файл для записи:
  savedialog1.DefaultExt:='txt';
  //расширение файла:
  savedialog1.Filter:='Данные (*.txt)|*.TXT';
  savedialog1.FilterIndex:=1;
  //записываем файл в папку res:
  s:=extractfilepath(application.exename)+'res\';
  savedialog1.InitialDir:= s;
  savedialog1.Title:='Запишите данные на диск';
  //имя файла по умолчанию:
  savedialog1.filename:='temp.txt';
  if not savedialog1.Execute then exit;
  //имя конечного файла:
  fn:= savedialog1.filename;
  assignfile(f,fn);
  rewrite(f);

  //записать данные:
  for i:=0 to 65535 do begin
    s:=''; w:= i;
    for j:= 0 to 15 do  begin
      //ставим пробел между четвёрками чисел:
      if j mod 4 = 0 then s:= s+' ';
      if w and $8000 <> 0 then //- клетка фигуры
        s:= s+'1'
      else
```

```
        s:= s+'0';              //- клетка поля
      w:=  w shl 1;
    end;
    s:=inttostr(i)+ '/'+s+'. Range= '+ inttostr(status[i].range)+'
Ход= '+        inttostr(status[i].hod.x)+'
'+inttostr(status[i].hod.y)+ ' Pred= '+inttostr(status[i].pred);
    if status[i].range<> -1 then writeln (f, s);
  end;

  //закрыть файл:
  closefile(f);
  messagebeep(0)
end; //sbtSaveListClick
```

Запустите программу, нажмите кнопку *Создать список*, а затем *Записать*. Если вы не изменяли значений по умолчанию, то сможете найти файл в *папке res* под именем *temp.txt*. Его начало выглядит так:

```
0/ 0000 0000 0000 0000. Range= 0 Ход= 0 0 Pred= 0
1/ 0000 0000 0000 0001. Range= 7 Ход= 3 3 Pred= 4382
2/ 0000 0000 0000 0010. Range= 7 Ход= 3 3 Pred= 4381
```

Загрузите файл в любой текстовый редактор и найдите помощью команды *Find* строку *0010 1101 0111 1000*, которая представляет собой *начальную* позицию игры *Флип-Флоп*. Вот что вы должны увидеть:

```
11640/ 0010 1101 0111 1000. Range= 8 Ход= 2 1 Pred= 602
```

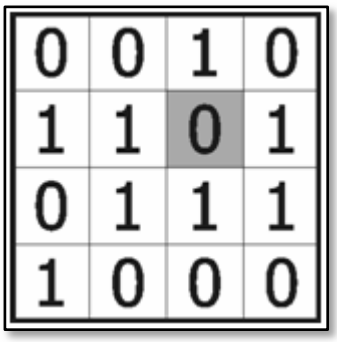
Первое число – это наша позиция, записанная в 10-тичном виде. Четыре 4-значных числа после него – запись позиции по строкам. Затем идёт ранг позиции, ход (строка-колонка, отсчёт начинает с *нуля*) и, наконец, предыдущая позиция, записанная в 10-тичном виде.

Следовательно, в исходной позиции следует выполнить ход в клетку, находящуюся на пересечении второй колонки (так как мы считаем с нуля, то физически она третья слева; если вы путаетесь в координатах, то измените вывод информации в процедуре *sbtSaveListClick* так, чтобы самая левая колонка имела номер 1) и первой строки (второй сверху). В результате хода получится позиция, имеющая номер *602* в списке. Найдите её аналогично первой строке:

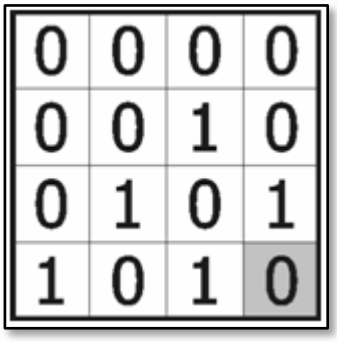
```
602/ 0000 0010 0101 1010. Range= 7 Ход= 3 3 Pred= 4933
```

Представим всю информацию из файла в удобном виде (Рис. П5.6).

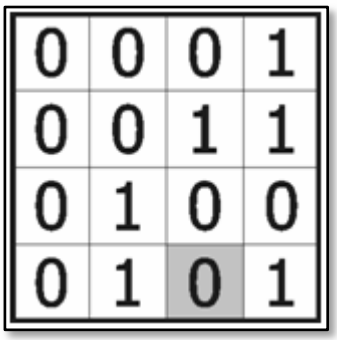
```
11640/ 0010 1101 0111 1000. Range= 8 Ход1= 2 1 → 602
```



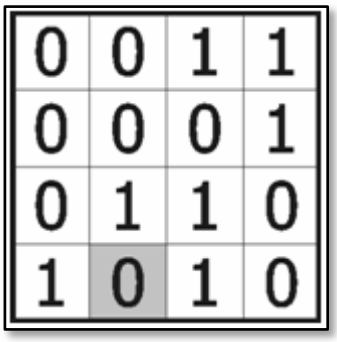
602/ 0000 0010 0101 1010. Range= 7 Ход2= 3 3 → 4933



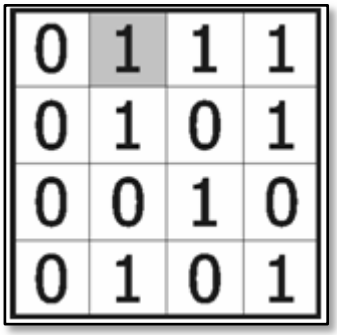
4933/ 0001 0011 0100 0101. Range= 6 Ход3= 2 3 → 12650



12650/ 0011 0001 0110 1010. Range= 5 Ход4= 1 3 → 29989

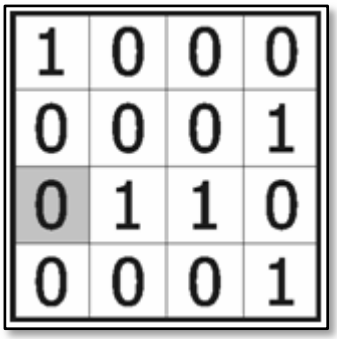


29989/ 0111 0101 0010 0101. Range= 4 Ход5= 1 0 → 33121



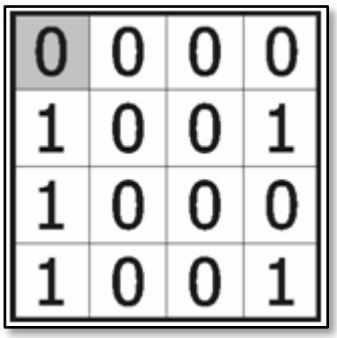
0	1	1	1
0	1	0	1
0	0	1	0
0	1	0	1

33121/ 1000 0001 0110 0001. Range= 3 Ход6= 0 2 → 2457



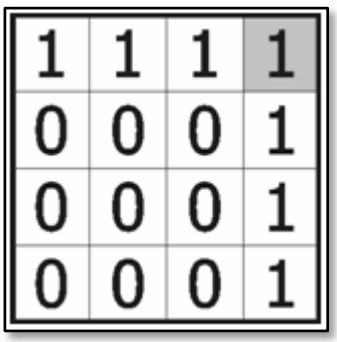
1	0	0	0
0	0	0	1
0	1	1	0
0	0	0	1

2457/ 0000 1001 1001 1001. Range= 2 Ход7= 0 0 → 61713



0	0	0	0
1	0	0	1
1	0	0	0
1	0	0	1

61713/ 1111 0001 0001 0001. Range= 1 Ход8= 3 0 → 0



1	1	1	1
0	0	0	1
0	0	0	1
0	0	0	1

0 Range= 0

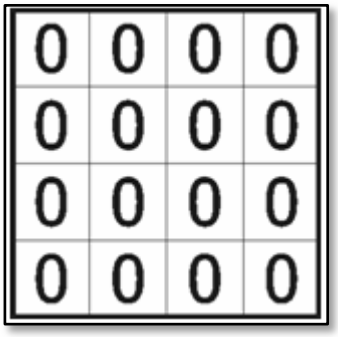


Рис. П5.6. Делай, как я!

Теперь мы с уверенностью можем утверждать, что для решения предложенной задачи достаточно *восьми* ходов. И мы даже знаем, *какие* ходы следует делать, но до сих пор не выяснили, *почему* нужно ходить так, а не иначе. Попытаемся справиться и с этой проблемой.

Если вы подобным же образом проследите решение ещё нескольких задач, то наверняка обратите внимание на то, что ни один ход *не повторяется*. Когда это знаешь, легко найти объяснение данному «явлению»: повторный ход в ту же клетку просто «нейтрализует» первый ход, поэтому, выбросив любую пару одинаковых ходов, мы нисколько не изменим позицию, возникающую в результате выполнения остальных ходов. Отсюда слкдует вывод, что *любая задача может быть решена не более чем за 16 ходов*, в противном случае ходы неизбежно начнут *повторяться*. Ранее мы пришли к такому же выводу, произведя полный перебор позиций. Также нами было установлено, что порядок выполнения ходов не имеет никакого значения, поэтому существует *множество* решений нашей задачи, отличающихся только порядком ходов.

Вот ещё бы научиться определять те клетки, в которые нужно делать ходы! Это, пожалуй, самое трудное в этой головоломке, здесь вам потребуется уже вся ваша наблюдательность и сообразительность!

Последуем нашему принципу: не знаешь, с чего начать, начинай с конца! Вернёмся к *заключительной* позиции, состоящей из одних нулей. Какую бы клетку мы ни взяли, сумма чисел в соответствующей колонке и строке равна 0. Сделаем ход в клетку 3,0 (или в любую другую, безразлично). Получится такая позиция (Рис. П5.7).

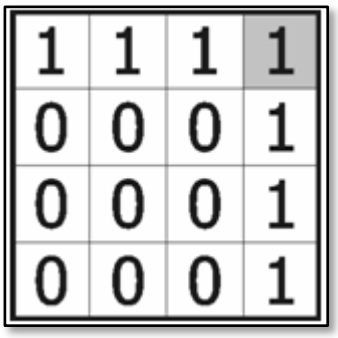


Рис. П5.7. Первый ход

Во всех клетках с нулями сумма чисел станет равной 2. Во всех остальных, кроме угловой, 4. И только в угловой клетке сумма чисел будет равна 7. Запомните, что именно в неё мы и сделали ход.

Предположим, что мы сделали *второй* ход в одну из клеток с *нулями*. Сумма чисел в них равна двум. Значит, в соответствующей колонке и столбце 7 – 2 = 5 нулей (не посчитайте клетку на их пересечении дважды!). В результате хода появятся, наоборот, 5 единиц и 2 нуля, то есть сумма увеличится до пяти.

Если вы сделаете второй ход в клетку с *единицей* (исключая угловую, так как в неё второй раз ходить нельзя), то вместо четырёх единиц и трёх нулей возникнут 4 нуля и 3 единицы. Итак, в какую бы клетку мы ни сходили, сумма чисел в колонке и в строке, проходящих через неё, будет *нечётной*. Убедитесь самостоятельно, что чётность всех остальных клеток поля не изменяется: нечётные клетки (с нечётной суммой) так и останутся нечётными, а чётные (с чётной или нулевой суммой) также останутся чётными.

Для клеток, которые не лежат в колонке и столбце с «сыгравшей» клеткой, всё объясняется просто: в результате хода изменяются 2 числа – либо 0 + 0 → 1 + 1, либо 1 + 1 → 0 + 0, либо 1 + 0 → 0 + 1, поэтому сумма в чётных клетках изменится на чётное число (или останется прежней) и их чётность не изменится. Для нечётных клеток ситуация аналогичная.

В клетках, лежащих в упомянутых колонке и строке, изменится значение 4-х чисел. Если в колонке или строке было *чётное* количество единиц (или вообще не было), то количество нулей также было *чётным*. После выполнения хода чётность ряда не изменится. Рассуждая аналогично, можно доказать, что нечётный ряд также сохранит свою нечётность. Из этого следует, что в результате хода меняется чётность только *единственной* клетки – той, в которую сделан ход.

Сделав *третий* ход, мы изменим чётность соответствующей клетки. А так как ходить следует исключительно в чётные клетки, то с каждым ходом на поле будет добавляться одна нечётная клетка.

Задача из игры *Флип-Флоп* решается за 8 ходов. Вы можете легко проверить, что в исходной позиции ровно 8 нечётных клеток. Если мы теперь пойдем не от конечной позиции к начальной, а наоборот, то и ходить нам нужно в *нечётные* клетки. Тогда с каждым ходом их количество будет уменьшаться, и мы неминуемо придём к конечной позиции с одними нулями. Вот, как говорится, и весь сказ!

Благодаря проведённому анализу головоломки мы можем для *любой* начальной позиции указать число ходов, необходимых для её решения, и клетки, в которые эти ходы следует выполнять. В награду за умственное напряжение мы можем сделать себе послабление в виде дополнительного кода (выделенные строки) в процедуре вывода клеток игрового поля на экран, которая сама будет отмечать *плюсиком* нечётные клетки, и нам останется только усердно нажимать на них.

```
//ОТРИСОВАТЬ ЯЧЕЙКУ СЕТКИ
procedure TForm1.dgPoleDrawCell(Sender: TObject; ACol, ARow: Inte-
ger; Rect: TRect; State: TGridDrawState);
var
  i,j,n: integer;
begin
  //закрасить клетку своим цветом:
  case masPole[ACol, ARow] of
    0: dgPole.Canvas.Brush.Color:= clWhite;
    else dgPole.Canvas.Brush.Color:= clRed;
  end;
  dgPole.Canvas.FillRect(Rect);

  //нечётный ряд?
  n:= 0;
  for j:= 0 to POLE_HEIGHT-1 do
    for i:= 0 to POLE_WIDTH-1 do
      if (i=ACol) or (j=ARow) then begin
        n:=n+masPole[i,j];
      end;
  //нечётный - вывести плюсик:
  if odd(n) then
  with Rect, dgPole.Canvas do
    textrect(Rect, left+(right-left-textwidth('+')) div 2,
top+(bottom-top-textheight('+')) div 2, '+');
end;  //dgPoleDrawCell
```

Запустите программу и правой кнопкой мыши введите исходную позицию. Картина, которую вы увидите, бесспорно, порадует настоящего ценителя головоломок (Рис. П5.8).



Рис. П5.8. Начальная позиция

Обидно только за игру *Флип-Флоп* - она пала жертвой нашего любопытства, в ней больше нет тайн для пытливого ума. А ведь хорошая игра, жаль с ней так быстро расставаться, поэтому давайте продлим очарованье и вдохнём в неё новую жизнь, усложнив последнюю себе тем, что чуточку подправим условия игры – в качестве конечной позиции возьмём не «завсегдашнюю» нулевую, а *произвольную*. Вы уже догадались, как нужно играть в этом случае?

Поскольку мы умеем любую позицию сводить к нулевой, то можно поступить так: от начальной позиции перейти к нулевой и от конечной также перейти к нулевой. Объединив все ходы, мы получим решение задачи в целом. Понятно, что решение некоторых задач при такой стратегии будет не самым коротким: если одна из позиций требует для «обнуления», например, 9 ходов, а вторая 10, то в сумме придётся затратить 19 ходов, что больше 16. Стало быть, мы сумеем из любой начальной позиции перейти в любую конечную, но при этом сделаем *лишние* ходы. Как же от них избавиться? Самые умные читатели могут отдохнуть, а мы пока составим процедуру для перехода от одной позиции к другой.

Установите на форме *кнопку sbtPerehod*, а в процедуре-обработчике напишите код, он очень простой, так как весь поиск кратчайшего перехода будет осуществляться в отдельной процедуре *Solution*. При её вызове нужно указать начальную и конечную позиции (в 10-тичном или 16-ричном виде), а также минимальное число ходов (если вы его не знаете, просто подбирайте, пока не найдёте самого короткого решения).

```
procedure TForm1.sbtPerehodClick(Sender: TObject);
begin
  //Solution(0, $e997, 2);
  //Solution(0, $1886, 3);
  //Solution(63897, 39409, 11);
  //Solution(0, 19, 5);
```

```
  //Solution(63897, 112, 5);
  //Solution(112, 39409, 6);
  //Solution($ffff, 0, 16);
  Solution(11640, 0, 16);
end;  //sbtPerehodClick
```

Процедуру поиска решения мы подробно рассматривать не будем – если она вас интересует, то постарайтесь разобраться в ней самостоятельно. Проследите решение нескольких задач и найдите причину, по которой предложенный нами ранее приём давал лишние ходы. Если вам не удастся справиться с этой проблемой, то читайте следующий раздел этой главы, в ней туман над *Флип-Флопом* рассеется окончательно.

```
//НАЙТИ ПЕРЕХОД ОТ ОДНОЙ ПОЗИЦИИ К ДРУГОЙ
procedure TForm1.Solution(w0, w1: word; nStep: integer);
//w0 - начальная позиция,
//w1 - конечная позиция,
//nStep2 - количество шагов (ходов)
label
  nextHod, nextCell;
var
  Hod: integer;
  nCells: integer;
  nCell: array[0..99] of integer;
  Pos: array[0..99] of word;
  Move: array[0..99] of integer;
  x, y: byte;
  w: word;

  procedure SavePos;
  var
    w: word;
    i, j: integer;
    x, y: integer;
    s: string;
  begin
    for i:= 1 to Hod do begin
      s:=''; w:=Pos[i];
      for j:= 0 to 15 do  begin
        if  w and $8000 <> 0 then //- клетка фигуры
          s:= s+'1'
        else
          s:= s+'0';              //- клетка "фона"
        w:= w shl 1;
      end;
     listbox1.items.add('Hod= '+ inttostr(i)+ ' Pos= '+ s);
     x:= Move[i] mod 4; y:= Move[i] div 4;
     listbox1.items.add('Move= '+ inttostr(x)+ ' ' +inttostr(y));
     listbox1.items.add('');
    end;
  end;
```

```
begin
  nCells:= 16;
  Hod:= 0; Pos[0]:= w0; nCell[0]:= -1; Move[0]:= -1;
nextHod:
  Inc(Hod); nCell[Hod]:= nCell[Hod-1];  Pos[Hod]:= Pos[Hod-1];
nextCell:
  inc(nCell[Hod]);
  if nCell[Hod]> nCells then begin //- прошли всё поле
    dec(Hod);
    if Hod< 0 then begin
      ShowMessage('OK');
      exit;
    end;
    Goto nextCell
  end;
  x:= nCell[Hod] mod 4; y:= nCell[Hod] div 4;
  w:= XorBits(x, y, Pos[Hod-1]);
  Pos[Hod]:= w;
  //запомнить ход:
  Move[Hod]:= nCell[Hod];

  if w = w1 then begin //- нашли!
    savepos;
    ShowPole(w);
    exit;
  end;

  if Hod< nStep then goto NextHod;

  goto nextCell
end;  //Solution
```

Напоследок – форма со всеми принадлежностями для успешной работы (Рис. П5.9).

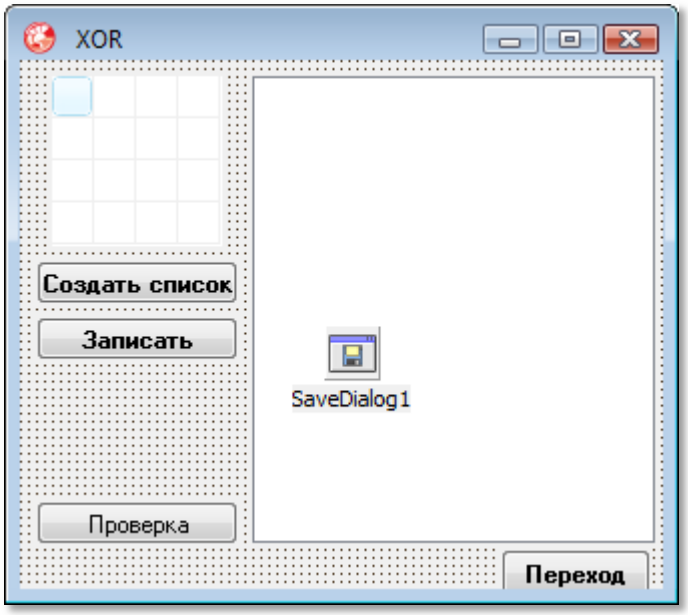


Рис. П5.9. Немудрёный интерфейс!



Некоторые читатели могут упрекнуть автора в бесполезной трате времени и бумаги, ведь можно было сразу же перейти к программированию головоломки, а не демонстрировать «муки творчества». Действительно, обычно в книгах приводятся готовые, отлаженные программы, которые ясно свидетельствуют о недюжинных умственных способностях их авторов и создают у неразумных «чайников» превратное впечатление, что есть этакие «профессионалы», которые пишут программы в один присест, без напряжения и ошибок, и «любители», мучающиеся над каждой строкой исходного кода.

На самом-то деле умные мысли посещают и умных людей отнюдь не сразу (иначе компьютеры придумали бы ещё во времена Архимеда) и невозможны без предварительных «глупостей» и ошибок. Так как для вас, вероятно, гораздо важнее научиться писать *собственные* программы, чем переписывать чужие, то я и рискнул показать, какими окольными путями приходится подбираться к хорошим решениям и алгоритмам. Считайте, что вся проделанная нами работа – это черновик, который дальше мы будем использовать для создания «настоящей программы». (В скобках следует заметить, что и «черновик» тоже изрядно «отредактирован»: из него удалены очевидные заблуждения, опечатки и тупиковые мысли, так что не удивляйтесь, если ваши предварительные изыскания отнимут у вас гораздо больше времени и сил.)

Естественно, невозможно все программы разбирать так подробно, для этой мы сделали исключение - для примера, поэтому всегда имейте в виду, что прежде чем браться за написание игры, нужно предварительно отладить наиболее важные места вашей будущей программы.

Игра как она есть

Если для игры *Флип-Флоп* достаточно *одного* игрового поля, так как конечная позиция не изменяется и очень проста, то в *XorGame* ситуация осложняется тем, что конечная позиция в каждой партии *другая*. Естественно, держать в голове конечную позицию, когда нужно думать о выборе ходов, весьма затруднительно, поэтому для нашей игры мы сделаем *два* поля – с *начальной* позицией (размером побольше) и с *конечной* позицией (размером поменьше; вы можете сделать оба поля и равной величины, но в этом случае возможна неразбериха).

Поскольку с игрой *Флип-Флоп* мы досконально разобрались, то будет довольно скучно писать программу, которой нам пользоваться будет не интересно. Поэтому мы предусмотрим в нашей игре возможность *изменять* размеры поля – или вы считаете, что справитесь с головоломкой на любом поле? Достаточно иметь поля таких размеров: 4 x 4 (для начинающих), 5 x 5, 6 x 6, 7 x 7 (для головастых). Вы можете сделать поля ещё масштабнее, но ничего нового в игре от этого не появится.

Итак, откроем новый проект *XorGame* и займёмся *формой*:

```
Width = 397
Heightt = 312
BorderIcons = [biSystemMenu, biMinimize]
Caption = '  XorGame'
Icon – нарисуйте подходящий значок
PopupMenu = PopupMenu1 – добавим дальше
Position = poDesktopCenter
OnClose = FormClose – процедуры-обработчики.
OnCreate = FormCreate
```

Затем, оставив сверху место для кнопок, расположим на форме две сетки – *dgPole* и *dgPole2*. В целом форма со всеми компонентами должна выглядеть так (Рис. 5.10).

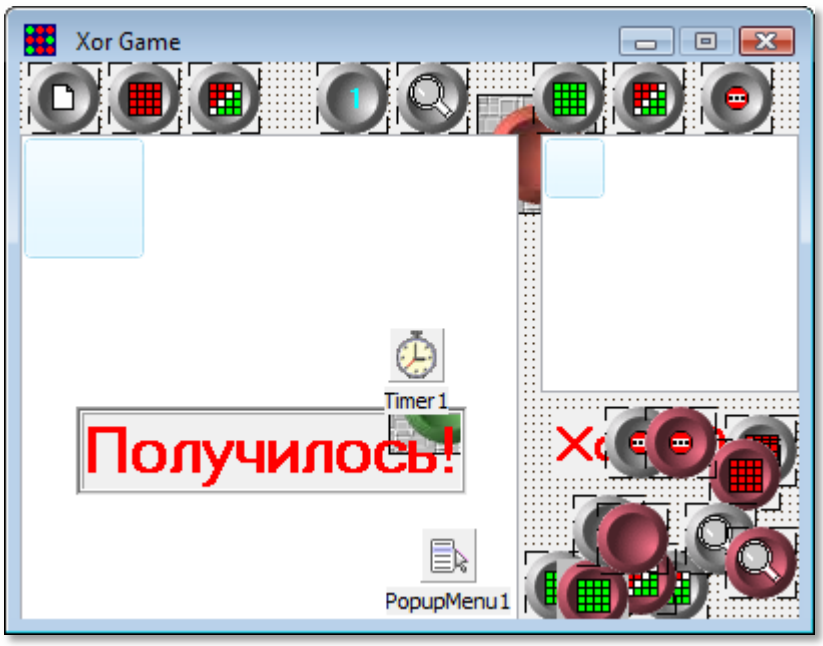


Рис. П5.10. Двупольная форма!

Свойства сетки *dgPole*:

```
Left = 0
Top = 36
```

```
    Width = 249
    Height = 249
    Cursor = crHandPoint  («ручка» - нужно нажимать фишки)
    ColCount = 4
    RowCount = 4  (запускаем игру в режиме «Флип-Флоп»)
    DefaultColWidth = 60
    DefaultRowHeight = 60  (размеры клеток поля достаточно боль-
шие, чтобы фишки выглядели «эффектно»)
    DefaultDrawing = False
    FixedCols = 0
    FixedRows = 0
    GridLineWidth = 0  (разметка сетки не нужна, её заменят линии
на рисунках с фишками)
    Options = []
    ScrollBars = ssNone
    OnDrawCell = dgPoleDrawCell
    OnMouseDown = dgPoleMouseDown
```

Свойства сетки *dgPole2*:

```
    Left = 260
    Top = 36
    Width = 129
    Height = 129
    Cursor = crArrow
    ColCount = 4
    RowCount = 4
    DefaultColWidth = 30
    DefaultRowHeight = 30
    DefaultDrawing = False
    FixedCols = 0
    FixedRows = 0
    GridLineWidth = 0
    Options = []
    ScrollBars = ssNone
    OnDrawCell = dgPole2DrawCell
    OnMouseDown = dgPole2MouseDown
```

Для удобства введём несколько глобальных *констант*:

```
const
  //макс. размеры поля:
  MAX_POLE_WIDTH= 7;
  MAX_POLE_HEIGHT= 7;
  //цвет фишек:
  WHITE=0;
  BLACK=1;
```

Хотя цвета фишек мы обозначили как «белый» и «чёрный», на самом деле они могут иметь любой цвет, важно только помнить, что *чёрные* фишки – это закрытые окна, а *белые* – открытые. Поэтому игра *Флип-Флоп* в

варианте с фишками должна иметь конечную позицию, состоящую только из *белых* фишек. Если у вас есть страстное желание украсить игру, то вместо фишек вы можете взять произвольные картинки (хотя бы и окна), на самой игре это никак не отразится. Вот только надо ли отвлекать игрока совершенно ненужными визуальными и звуковыми эффектами – это БОЛЬШОЙ вопрос!

Как обычно, для хранения информации о фишках на поле мы заведём два глобальных *массива*:

```
type
  TPole= array[0..MAX_POLE_WIDTH-1, 0..MAX_POLE_HEIGHT-1] of byte;
var
  //поля:
  masPole, masPole2: TPole;
```

Нам обязательно понадобится в игре *метка lblHod*, чтобы показывать число сделанных при решении головоломки ходов:

```
    Left = 268
    Top = 176
    Width = 86
    Height = 29
    Caption = 'Ход - 0'
    Font.Color = clRed
    Font.Height = -24
    Font.Style = [fsBold]
    ParentFont = False
    OnClick = lblHodClick
```

Иногда бывает полезно начать отсчёт ходов с какой-то определённой позиции, для этого достаточно *щёлкнуть* на метке:

```
//ОБНУЛИТЬ ХОДЫ
procedure TForm1.lblHodClick(Sender: TObject);
begin
  Hod:= 0;
  lblHod.Caption:=  'Ход - 0';
end;  //lblHodClick
```

При поиске решения также очень важно уметь сохранять и восстанавливать позицию на поле, а также переходить к предыдущему или последующему шагу в цепочке ходов. Для хранения текущей позиции мы введём переменную *Memory*:

```
type
  TPole= array[0..MAX_POLE_WIDTH-1, 0..MAX_POLE_HEIGHT-1] of byte;
  TMemory= Record
```

```
    //номер хода:
    hod: integer;
    //размеры полей:
    w, h: integer;
    //позиция на полях:
    PosL, PosR: TPole;
  end;

var
  //поля:
  masPole, masPole2: TPole;
  Memory: TMemory;
```

Чтобы не загромождать форму кнопками, добавим в проект всплывающее меню *PopupMenu1* (компонент 

), которое будет появляться при нажимании *правой* кнопки мыши. Так оно выглядит в *Редакторе меню* (Рис. П5.11, слева) и в игре (Рис. П5.11, справа).

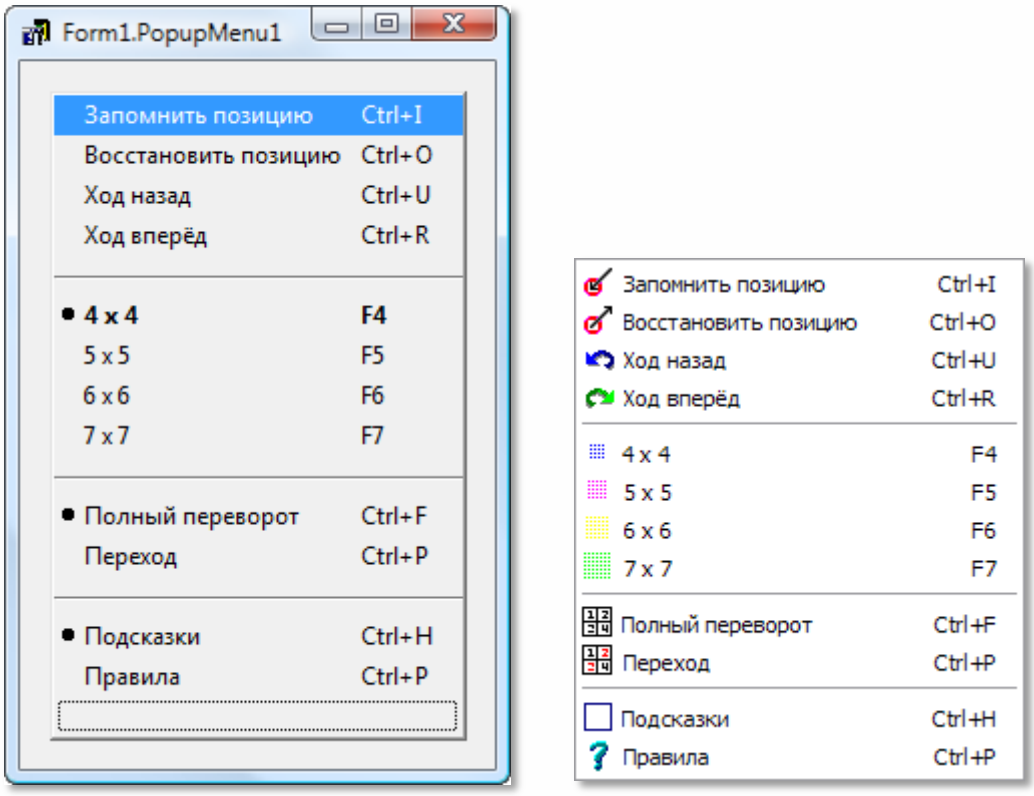


Рис. П5.11. Всплывающее меню

Лучше немного забежать вперёд и сразу создать *все* пункты меню, чем много раз возвращаться в *Редактор меню* и добавлять по одному пункту.

Чтобы каждый пункт меню предварял *значок*, загрузите в свойство *Bitmap* подходящую картинку. Часть из них вы можете взять из наших предыдущих программ, а другие нарисовать самостоятельно.

Первый пункт меню – *miMemoryIn* – отвечает за *сохранение* в памяти информации о текущей позиции на игровом поле:

```
Caption= 'Запомнить позицию'
ShortCut= Ctrl+I
OnClick = miMemoryInClick
```

```
//ЗАПОМНИТЬ ПОЗИЦИЮ
procedure TForm1.MemoryIn;
begin
  Memory.hod:= Hod;
  Memory.w:= dgPole.ColCount;
  Memory.h:= dgPole.RowCount;
  Memory.PosL:= masPole;
  Memory.PosR:= masPole2;
end; //MemoryIn

procedure TForm1.miMemoryInClick(Sender: TObject);
begin
  if GameState= gsSolution then exit;
  MemoryIn;
end; //miMemoryInClick
```

Раз уж появилось упоминание о *статусе* игры, то следует рассказать о нём. Введём новый *тип:*

```
TGameState= (gsWait, gsSolution);
```

и *переменную* этого типа:

```
//состояние программы:
GameState: TGameState= gsWait;
```

В *первом* состоянии программа находится, когда ждёт ваших действий (выполнения хода, нажатия на кнопку, выбора пункта меню). Во *втором* – когда она ищет решение головоломки. Ясно, что в этом случае нужно отвергнуть все другие действия в программе (кроме прекращения самого поиска, иначе может возникнуть ситуация, когда нам придётся завершать программу негуманным способом).

Второй пункт меню – *miMemoryOut* – выполняет противоположную операцию - *восстанавливае*т позиции на обоих полях:

```
Caption = 'Восстановить позицию'
ShortCut = Ctrl+O
OnClick = miMemoryOutClick
```

Так как размеры полей могли быть изменены, то нужно вывести *новые* поля, отметить в меню их действительные размеры и переместить форму в центр экрана (размеры формы будут автоматически подгоняться под размеры полей, поэтому её размеры зависят от размеров сеток):

```
//ВОССТАНОВИТЬ ЗАПОМНЕННУЮ ПОЗИЦИЮ
procedure TForm1.miMemoryOutClick(Sender: TObject);
begin
  if GameState= gsSolution then exit;

  with Memory do begin
    Prepare(dgPole, w, h);
    Prepare(dgPole2, w, h);
    masPole:= PosL;
    masPole2:= PosR;
    case w of
      4: mi4x4.Checked:= True;
      5: mi5x5.Checked:= True;
      6: mi6x6.Checked:= True;
      7: mi7x7.Checked:= True;
    end;
    form1.Position:= poScreenCenter;
  end;

  dgPole.Invalidate;
  dgPole2.Invalidate;
  Hod:= Memory.hod;
  lblHod.Caption:= 'Ход - ' + inttostr(hod);
end;  //miMemoryOutClick
```

Размеры полей задаются в *процедуре Prepare:*

```
//УСТАНОВИТЬ РАЗМЕР ЗАДАННОГО ПОЛЯ
procedure TForm1.Prepare(Pole: TDrawGrid; ColCount, RowCount: in-
teger);
var w,h, lw: integer;
begin
  //размер клетки в пикселях:
  w:= Pole.DefaultColWidth;
  h:= Pole.DefaultRowHeight;

  //толщина линий:
  lw:= Pole.GridLineWidth;

  //размеры игрового поля в клетках:
  Pole.ColCount:= ColCount;
  Pole.RowCount:= RowCount;

  //размеры в пикселях видимой части игрового поля:
  Pole.Width:= 3 + (w + lw)* ColCount+1;
  Pole.Height:= 3 + (h + lw)* RowCount+1;
```

```
  if pole= dgPole2 then
    pole.Left:= dgPole.Left+dgPole.Width+10;

  //установить метку, показывающую число ходов:
  lblHod.Left:= dgPole.Left+dgPole.Width+10;
  lblHod.Top:= dgPole2.Top+ dgPole2.Height+10;

  //обнулить ходы:
  Hod:=0;
  lblHod.Caption:= 'Ход - 0';
end;  //Prepare
```

Так как большинство действий одинаково для обоих полей, то для изменения их размеров мы используем *одну* процедуру. Нужно только учесть, что *горизонтальные* координаты полей, безусловно, отличаются.

Следующие два пункта меню мы пока пропустим, потому что их действие сейчас трудно объяснить. А далее в меню идут пункты, позволяющие выбрать один из доступных размеров полей. По умолчанию игровое поле имеет такие же размеры, как и в игре *Флип-Флоп*, то есть 4 на 4 клетки. Это естественно – неискушённый пользователь должен начинать с более лёгких заданий.

Пункт *mi4x4* устанавливает *минимальные* размеры полей:

```
      Tag = 4
      Caption = '4 x 4'
      Checked = True  (размер по умолчанию)
      GroupIndex = 1
      RadioItem = True  (одновременно можно выбрать только один
размер поля)
      ShortCut = F4
      OnClick = mi4x4Click
```

Аналогично действуют и другие пункты из этой группы (обратите внимание, что значение свойства *GroupIndex* у них равно 1, а *RadioItem* = *True;* это сделано для того, чтобы при выборе одного из пунктов все остальные выключались).

```
mi5x5:
      Tag = 5
      Caption = '5 x 5'
      GroupIndex = 1
      RadioItem = True
      ShortCut = F5
      OnClick = mi4x4Click

mi6x6:
      Tag = 6
```

```
      Caption = '6 x 6'
      GroupIndex = 1
      RadioItem = True
      ShortCut = F6
      OnClick = mi4x4Click

 mi7x7:
      Tag = 7
      Caption = '7 x 7'
      GroupIndex = 1
      RadioItem = True
      ShortCut = F7
      OnClick = mi4x4Click
```

Выбор всех пунктов меню мы будем обрабатывать в *одной* и той же процедуре *mi4x4Click*. А чтобы узнать, какие размеры полей нужно установить, присвоим свойству *Tag* соответствующее значение:

```
//ИЗМЕНИТЬ РАЗМЕРЫ ПОЛЕЙ
procedure TForm1.mi4x4Click(Sender: TObject);
var
 i, j, n: integer;
begin
  if GameState= gsSolution then exit;

  //показать пункт меню как выбранный:
  (Sender as TMenuItem).Checked:= True;

  //размеры поля:
  n:= (Sender as TMenuItem).Tag;

  //новое задание:
  dgPole.RowCount:= n; dgPole.ColCount:= n;
  dgPole2.RowCount:= n; dgPole2.ColCount:= n;
  NewPlay;

  //показать на экране:
  Prepare(dgPole, n, n);
  Prepare(dgPole2, n, n);
  dgPole.Invalidate;
  dgPole2.Invalidate;
  form1.Position:= poScreenCenter;
end;  //mi4x4Click
```

После изменения размера полей текущее задание теряет всякий смысл, поэтому оба поля просто заполняются фишками одного цвета. О том, как начать новую игру, будет рассказано далее.

В игре *Флип-Флоп* требуется все чёрные фишки (ну хорошо, окна!) перевернуть на белую сторону. Для нас эта задача оказалась слишком

простой, поэтому мы придумали усложнённый вариант игры, в котором нужно перейти от одной «ненулевой» позиции к другой. Сохраним *обе* разновидности игры в нашей программе. Опишем их в новом *типе TGame:*

```
  //вид игры:
  TGame=(FullSalto, Perehod);
```

и заведём новую *переменную* для хранения выбранного вида игры:

```
var
  …
  GameState: TGameState= gsWait;
  Game: TGame;
```

По умолчанию мы будем играть в более простой вариант:

```
miFullSalto:
      Tag = 0
      Caption = 'Полный переворот'
      Checked = True    (игра по умолчанию)
      GroupIndex = 2
      RadioItem = True
      ShortCut = Ctrl+F
      OnClick = miFullSaltoClick

miPerehod:
      Tag = 1
      Caption = 'Переход'
      GroupIndex = 2
      RadioItem = True
      ShortCut = Ctrl+P
      OnClick = miFullSaltoClick
```

Выбор пункта меню в этом случае осуществляется точно так же, как и при изменении размеров поля:

```
//ВЫБРАТЬ ВИД ИГРЫ
procedure TForm1.miFullSaltoClick(Sender: TObject);
begin
  if GameState= gsSolution then exit;

  (Sender as TMenuItem).Checked:= True;

  //вид игры:
  Game:= TGame((Sender as TMenuItem).Tag);

  //начать новую игру:
  NewPlay;
end; //miFullSaltoClick
```

Управившись с меню, мы возвращаемся к началу программы. При запуске приложения в процедуре создания формы мы устанавливаем её свойство *AutoSize* в *True*, чтобы форма *подгонялась* под размеры полей. Это позволит нам избежать «пустот» на форме, когда размеры полей 4 x 4 клетки, а форма рассчитана на поля максимального размера.

```
//СОЗДАТЬ ФОРМУ
procedure TForm1.FormCreate(Sender: TObject);
begin
  //подстраиваем величину формы под размеры сеток:
  AutoSize:= True;

  //установить размеры полей по умолчанию:
  Prepare(dgPole, 4, 4);
  Prepare(dgPole2, 4, 4);

  //начать новую игру:
  NewPlay;
end; //FormCreate
```

Вся подготовка к новой игре производится в процедуре *NewPlay*. Здесь мы должны обнулить число ходов, а затем выставить фишки на игровом поле случайным образом, но так, чтобы задача не получилась слишком простой:

```
   repeat
    Randomize;
   //выставить на поле фишки случайным образом:
    for j:= 0 to dgPole.RowCount-1 do
      for i:= 0 to dgPole.ColCount-1 do
        masPole[i,j]:= Random(2);

    //подсчитать число нечётных рядов:
    nrow:=0;
    for j1:= 0 to dgPole.RowCount-1 do
      for i1:= 0 to dgPole.ColCount-1 do begin
        n:= 0;
        for j:= 0 to dgPole.RowCount-1 do
          for i:= 0 to dgPole.ColCount-1 do
            if (i=i1) or (j=j1) then
              n:=n+masPole[i,j];
        if odd(n) then inc(nrow);
      end;
  until nrow>5;  //повторять, пока не будет больше 5
  dgPole.Invalidate;
```

На поле-образец мы выставляем фишки в зависимости от вида игры. Если это *Флип-Флоп*, то мы просто ставим «белые» фишки, если *Xor* – поступаем аналогично первому полю:

```
    //выставить на поле фишки случайным образом:
    for j:= 0 to dgPole2.RowCount-1 do
      for i:= 0 to dgPole2.ColCount-1 do
        masPole2[i,j]:= Random(2);
```

Казалось бы, всё верно – ан нет: до этого мы занимались только полем 4 на 4 клетки, на котором *все* позиции разрешимы. Мы распространили наши познания на поля *любых* размеров и ошиблись - например, на поле 5 х 5 клеток некоторые позиции нельзя «обнулить». Разговор об этом пойдёт дальше, но, чтобы не возвращаться к этой процедуре, мы изменим алгоритм выбора случайной позиции так, чтобы *любая* игровая позиция была решаемой. Сделать это достаточно просто – нужно выполнить несколько ходов из *нулевой* позиции. Ясно, что, поменяв порядок ходов на обратный, мы всегда сможем из заданной позиции вернуться к нулевой. Причём размер поля в данном случае не имеет никакого значения:

```
//ПОДГОТОВКА К НОВОЙ ИГРЕ
procedure TForm1.NewPlay;
var
 i, j: integer;
 x, y: integer;
begin
  //обнулить число ходов:
  Hod:= 0;
  lblHod.Caption:=  'Ход - 0';

  //сделать случайные ходы:
  randomize;
  //очистить левое поле:
  for j:= 0 to dgPole.RowCount-1 do
    for i:= 0 to dgPole.ColCount-1 do
      masPole[i,j]:= WHITE;

  //сделать случайные ходы:
  for i:= 1 to 7 + random(dgPole.ColCount*dgPole.RowCount+1-7) do
begin
    x:= Random(dgPole.ColCount); y:= Random(dgPole.RowCount);
    masPole:= DoMove(masPole, x, y);
  end;
  dgPole.Invalidate;

  //задать поле-образец:
  if Game= FullSalto then begin //- полный переворот
    //выставить на поле зелёные фишки:
    for j:= 0 to dgPole2.RowCount-1 do
      for i:= 0 to dgPole2.ColCount-1 do
        masPole2[i,j]:= WHITE
  end
  else begin //- переход к образцу
    //сделать случайные ходы:
```

```
    for i:= 1 to 7 + random(dgPole2.ColCount*dgPole2.RowCount+1-7)
do begin
      x:= Random(dgPole2.ColCount); y:= Random(dgPole2.RowCount);
      masPole2:= DoMove(masPole2, x, y);
    end;
  end;
  dgPole2.Invalidate;

  //запомнить начальную позицию:
  MemoryIn;
end;  //NewPlay
```

Чтобы выполнить ход, мы поступаем как и раньше, но теперь нам не удастся хранить информацию о поле в двухбайтовом слове, так как поля могут содержать более 16 клеток. Как вы помните, для этого мы ввели специальный тип *TPole*. Расход памяти под хранение информации о полях увеличился, но зато не нужно считать биты!

```
//ВЫПОЛНИТЬ ХОД
function TForm1.DoMove(arr: TPole; x, y: integer): TPole;
var i: integer;
begin
  //инвертировать заданную клетку:
  arr[x,y]:= abs(arr[x,y]-1);
  //инвертировать ряды:
  for i:= 0 to dgPole.RowCount-1 do //- вертикальный
    arr[x,i]:= 1- arr[x,i];
  for i:= 0 to dgPole.ColCount-1 do //- горизонтальный
    arr[i,y]:= 1- arr[i,y];
  result:= arr;
end; //DoMove
```

Остались последние два пункта меню – вывод *подсказок* и *помощи*. Подсказка заключается в том, что *нечётные* клетки (в них должен быть сделан очередной ход, если вы забыли) будут отмечены *синим квадратиком* (раньше мы использовали плюсик, но для настоящей игры он не годится по эстетическим соображениям).

```
miHint:
      Caption = 'Подсказки'
      Checked = True  (по умолчанию подсказки выводятся на экран)
      GroupIndex = 3
      RadioItem = True
      ShortCut = Ctrl+H
      OnClick = miHintClick
```

При переключении режима подсказок оба поля обновляются, чтобы показать или убрать синие квадраты:

```
//ПОДСКАЗКИ
procedure TForm1.miHintClick(Sender: TObject);
begin
  miHint.Checked:= not miHint.Checked;
  dgPole.Invalidate;
  dgPole2.Invalidate;
end;  //miHintClick
```

```
miHelp:
      Caption = 'Правила'
      GroupIndex = 4
      ShortCut = Ctrl+P
```

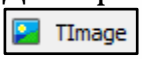


После того как вы закончите программировать игру, напишите сами «справочное пособие» для начинающих пользователей!

Как я уже отмечал выше, фишки могут быть *любого* цвета, а не только белыми и чёрными (они названы так, чтобы не было путаницы при изменении цвета настоящих фишек). В данном случае роль чёрных фишек играют *красные*, а белых – *зелёные*. Чтобы игровое поле выглядело ещё привлекательнее, можно добавить *фон*. Размер клеток поля 60 х 60 пикселей, такой же величины должны быть и растровые картинки. Кнопки можно нарисовать в любом графическом редакторе, но лучше использовать специализированные редакторы для *Web*-графики. И последнее: не забудьте по контуру рисунка выполнить чёрную рамку (Рис. П5.12).



Рис. П5.12. Цветные фишки

Для хранения этих картинок установите на форме два компонента *TImage* 

и загрузите в свойство *Picture* картинки с диска.

```
imgWhite:
    Width = 60
    Height = 60
    AutoSize = True
    Visible = False

imgBlack:
    Width = 60
    Height = 60
```

```
    AutoSize = True
    Visible = False
```

При выводе сетки на экран вызывается *процедура dgPoleDrawCell*, в которую и нужно поместить все операции по отрисовке каждой клетки поля в зависимости от того, какая фишка в ней находится. В нужную ячейку поля копируется соответствующая картинка, а затем, если необходимо, в нечётной клетке рисуется синий квадрат:

```
//ОТРИСОВАТЬ ЯЧЕЙКУ ИГРОВОГО ПОЛЯ
procedure TForm1.dgPoleDrawCell(Sender: TObject; ACol, ARow: Inte-
ger;
  Rect: TRect; State: TGridDrawState);
var
  i, j, n, m: integer;
  r, dr: TRECT;
begin
  //фишка в ячейке:
  n:= masPole[ACol, ARow];

  //размеры картинок:
  r:= Bounds(0, 0, 60, 60);
  dr:= Bounds(Rect.Left, Rect.Top, 60, 60);

  case n of
      WHITE: //- белая фишка
        dgPole.canvas.CopyRect(dR, imgWhite.Canvas, R);
      else   //- чёрная фишка
        dgPole.canvas.CopyRect(dR, imgBlack.Canvas, R);
  end;

  //выводить подсказки для нечётных позиций?
  m:= dgPole.ColCount-1;
  if miHint.Checked then begin // - выводить подсказки
    //нечётная клетка?
    n:= 0;
    for j:= 0 to m do
      for i:= 0 to m do
        if (i=ACol) or (j=ARow) then begin
          n:=n+masPole[i,j];
        end;
    if odd(n) then begin // - нечётная!
      //нарисовать контурный квадрат:
      InflateRect(dr,-3,-3);
      dgPole.Canvas.Brush.Style:= bsClear;
      //цвет линий:
      dgPole.Canvas.Pen.Color:= clBlue;
      dgPole.Canvas.Pen.Width:= 1;
      dgPole.Canvas.Rectangle(dr.left, dr.top, dr.right,
dr.bottom);
    end;
```

```
  end;
end;  //dgPoleDrawCell
```

Точно так же выводятся на экран и клетки поля-образца. Есть только одно различие: клетки этого поля в 2 раза меньше размера картинок, поэтому метод канвы *CopyRect* не подходит, ему на смену приходит более мощный метод *StretchDraw*, который «умеет» подгонять картинку под любые размеры:

```
//ОТРИСОВАТЬ ЯЧЕЙКУ ПОЛЯ-ОБРАЗЦА
procedure TForm1.dgPole2DrawCell(Sender: TObject; ACol, ARow: In-
teger;
  Rect: TRect; State: TGridDrawState);
var
  i, j, n, m: integer;
  dr: TRECT;
begin
  n:= masPole2[ACol, ARow];
  dr:= Bounds(Rect.Left, Rect.Top, Rect.Right-
Rect.Left,Rect.Bottom-Rect.Top);
  case n of
      WHITE: //- белая фишка
        dgPole2.canvas.StretchDraw(dR, imgWhite.Picture.Graphic);
      else   //- чёрная фишка
        dgPole2.canvas.StretchDraw(dR, imgBlack.Picture.Graphic);
  end;

  //выводить подсказки для нечётных позиций?
  m:= dgPole2.ColCount-1;
  if miHint.Checked then begin
    n:= 0;
    for j:= 0 to m do
      for i:= 0 to m do
        if (i=ACol) or (j=ARow) then begin
          n:=n+masPole2[i,j];
        end;
    if odd(n) then begin
     InflateRect(dr,-2,-2);
     dgPole2.Canvas.Brush.Style:= bsClear;
     dgPole2.Canvas.Pen.Color:= clBlue;
     dgPole2.Canvas.Pen.Width:= 1;
     dgPole2.Canvas.Rectangle(dr.left, dr.top, dr.right,
dr.bottom);
    end;
  end;
end;  //dgPole2DrawCell
```

Вполне уместно сразу рассмотреть и вторую процедуру для игрового поля. При нажатии в какой-либо клетке *левой* кнопки мыши выполняется ход, то есть «переворачиваются» фишки в соответствующей колонке и строке.

Как это делается, мы уже разобрали. Если вы захотите выставить на поле *свою* позицию, то при нажатии кнопки мыши удерживайте клавишу *Shift* (правая кнопка мыши уже занята под вызов меню). В этом случае изменит цвет только *одна* фишка.

```
//ПЕРЕВЕРНУТЬ ФИШКУ ИЛИ СДЕЛАТЬ ХОД
procedure TForm1.dgPoleMouseDown(Sender: TObject; Button: TMouse-
Button;
  Shift: TShiftState; X, Y: Integer);
var
  ACol,ARow: integer;
  i, j: integer;
begin
  //координаты мыши:
  dgPole.MouseToCell(x,y,ACol,ARow);
  //нажата левая кнопка мыши и клавиша Shift -
  //инвертируем цвет клетки:
  if (ssLeft in shift) and (ssShift in shift ) then begin
    if masPole[ACol,ARow]= 1 then masPole[ACol,ARow]:= 0
    else masPole[ACol,ARow]:= 1;
    dgPole.Invalidate;
  end
  //нажата левая кнопка мыши без клавиши Shift -
  //делаем ход - инвертируем ряды:
  else if ssLeft in shift then begin
    inc(Hod); lblHod.Caption:= 'Ход - ' + inttostr(Hod);
    for j:= 0 to dgPole.RowCount-1 do
      for i:= 0 to dgPole.ColCount-1 do
        if (i=ACol) or (j=ARow) then
        //инвертировать клетку:
        masPole[i,j]:= abs(masPole[i,j]-1);
    dgPole.Invalidate;
    //запомнить ход:
    Moves[Hod].x:= ACol;
    Moves[Hod].y:= ARow;
    //проверить, не решена ли задача:
    if IsReady then Ready;
  end;
end;  //dgPoleMouseDown
```

Чтобы иметь возможность брать ходы назад, следует вести «протокол» - заполнять массив *Moves* координатами каждого хода. Объявите его в разделе глобальных переменных. Там же разместите и *переменную* для хранения числа сделанных ходов:

```
var
  …
  Game: TGame;
  //запись ходов:
  Moves: array[0..999] of TPoint;
```

```
  //номер хода:
  Hod: integer= 0;
```

После каждого хода нужно проверять, не решена ли задача. Доверим это *функции IsReady*. Если найдётся *хотя бы одна* фишка на *игровом* поле, отличная от фишки в соответствующей клетке *поля-образца*, то радоваться, увы, рано, в противном случае мы дошли до конечной позиции и имеем полное право поздравить себя с победой:

```
//ПРОВЕРИТЬ, НЕ ПОЛУЧИЛАСЬ ЛИ КОНЕЧНАЯ ПОЗИЦИЯ
function TForm1.IsReady: Boolean;
var
  i, j: integer;
begin
  Result:= True;
  for j:= 0 to dgPole.RowCount-1 do
    for i:= 0 to dgPole.ColCount-1 do
      if masPole[i,j]<> masPole2[i,j] then
        begin Result:= False; exit end
end; //IsReady
```

Поздравление мы оформим без изысков (Рис. П5.13).



Рис. П5.13. Скромненько, но со вкусом!

За основу примем компонент 

Panel1:

```
Width = 196
Height = 45
AutoSize = True
BevelInner = bvLowered
BorderStyle = bsSingle
Color = clBlack
Visible = False,
```

на котором расположим *метку Label1*:

```
Left = 2
Top = 2
Width = 188
Height = 37
Caption = 'Получилось!'
Font.Color = clRed
Font.Height = -32
Font.Style = [fsBold]
```

В *процедуре Ready* мы показываем панель в середине формы, обнуляем счётчик (объявите глобальную переменную *rep: integer= 0;*) и включаем *таймер*:

```
//ЗАДАЧА РЕШЕНА!
procedure TForm1.Ready;
begin
  panel1.Left:= (Form1.Width- panel1.Width) div 2;
  panel1.Top:= (Form1.Height- panel1.Height) div 2;
  panel1.Visible:= true;
  rep:= 0;
  timer1.Enabled:= true;
end; // Ready
```

Настройте компонент *TTimer1* так, чтобы он срабатывал примерно 2 раза в секунду:

```
    Interval = 400
    OnTimer = Timer1Timer
```

Через установленный промежуток времени будет выполняться следующая процедура:

```
//МИГАЮЩАЯ НАДПИСЬ
procedure TForm1.Timer1Timer(Sender: TObject);
begin
  inc(rep);
  if odd(rep) then label1.Font.Color:= clRed
  else label1.Font.Color:= clGreen;
  if rep>15 then begin
    timer1.Enabled:= false;
    panel1.Visible:= false;
  end;
end;  // Timer1Timer
```

В ней цвет надписи изменяется с красного на зелёный и наоборот. Через 15 миганий таймер выключится, а панель исчезнет с экрана.

Так как на правом поле находится *конечная* позиция, то можно позволить себе устанавливать её по собственному желанию, а вот делать ходы на этом поле не нужно, поэтому процедура-обработчик нажатия кнопки мыши будет проще, чем для игрового поля:

```
//ПЕРЕВЕРНУТЬ ФИШКУ НА ПОЛЕ-ОБРАЗЦЕ
procedure TForm1.dgPole2MouseDown(Sender: TObject; Button: TMouse-
Button;
  Shift: TShiftState; X, Y: Integer);
var
  ACol,ARow: integer;
```

```
begin
  //координаты мыши:
  dgPole2.MouseToCell(x,y,ACol,ARow);
  //нажата левая кнопка мыши - инвертируем цвет клетки:
  if ssLeft in shift then begin
    if masPole2[ACol,ARow]= 1 then masPole2[ACol,ARow]:= 0
    else masPole2[ACol,ARow]:= 1;
    dgPole2.Invalidate;
  end
end;  // dgPole2MouseDown
```

Теперь у нас имеется всё необходимое для того, чтобы решать задачи наподобие тех, что мы рассматривали в программе *Флип-Флоп*. Как вы помните, нахождение *короткого* перехода от одной ненулевой позиции к другой было оставлено для самостоятельного анализа. Конечно, вы догадались, что лишние ходы в решении возникают из-за клеток с плюсиками, одинаково расположенными на *обоих* полях (обе клетки *нечётные*). Выполнив *два* хода в одну и ту же клетку, мы только продлили решение задачи, но не изменили его. Отсюда следует правило: *для перехода из одной позиции в другую следует ходить на игровом поле в те нечётные клетки, которым соответствуют чётные клетки на поле-образце, и, наоборот, в те чётные клетки, которым там соответствуют нечётные*.

Технически это выглядит так. Мы последовательно перебираем пары соответствующих клеток обоих полей. Если ровно одна из них нечётная, то мы делаем ход в эту клетку игрового поля. Если они обе чётные или обе нечётные, то такие клетки мы пропускаем и переходим к следующей паре. Закончив обход всего поля (или раньше, если повезёт), мы неминуемо решим задачу.

Для примера рассмотрим конкретную задачу (Рис. П5.14).

На рисунках *серым* цветом выделены *нечётные* клетки на *обоих* полях. Ходы нужно выполнять только в те клетки игрового поля, которые отмечены *крестиком*. Всего для решения задачи потребуется 9 ходов.

Легко проверить, что, действуя аналогично, вы справитесь с *любой* задачей на поле 6 x 6 клеток (а также на любом другом квадратном с *чётным* количеством клеток; именно поэтому в программе отсутствует режим 8 x 8 клеток – начинающим игрокам задачи на таком большом поле будут не по зубам, а для знатоков в них нет ничего нового).

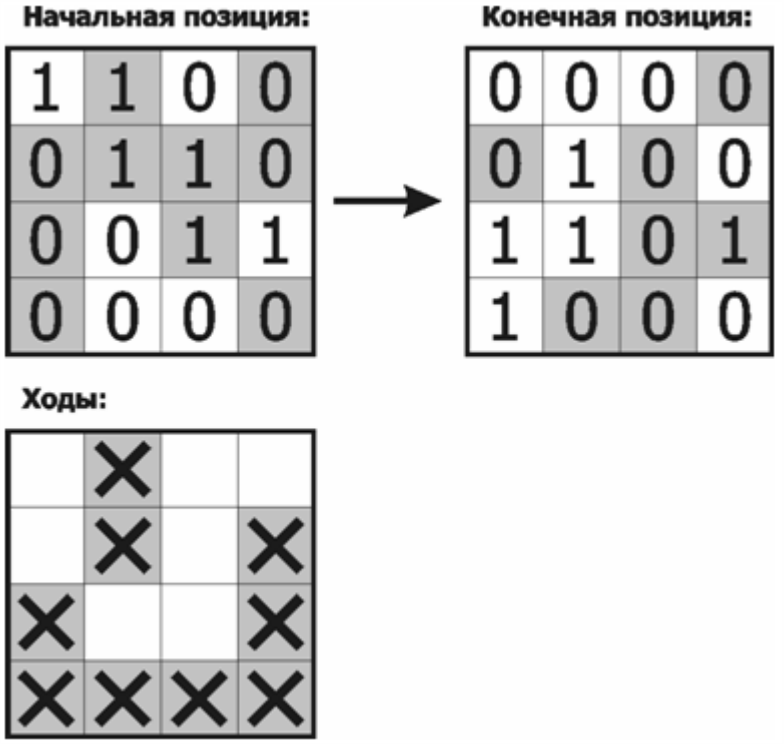


Рис. П5.14. Как решить задачу

Сюрпризы начинаются на поле 5 x 5 клеток. Куда бы вы ни сделали *первый* ход из нулевой позиции, вы получите не одну, а сразу *несколько* нечётных клеток (Рис. П5.15).

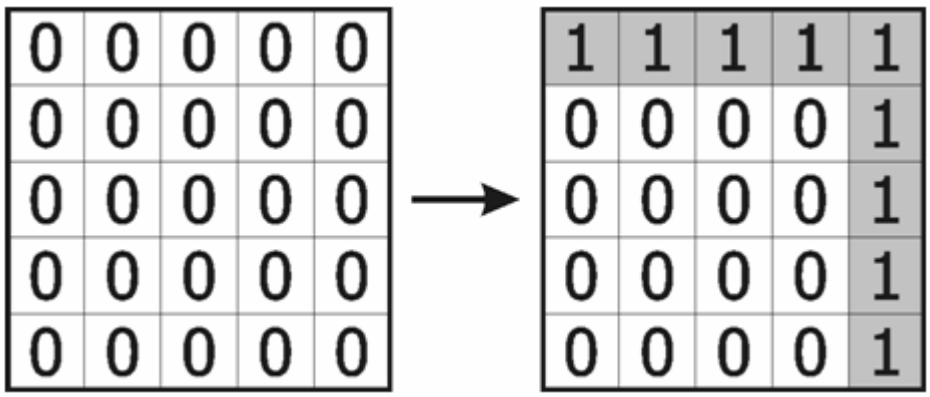


Рис. П5.15. Приключения начинаются!

Причём только одна из них «правильная», а все остальные не ведут к решению задачи. Ситуация становится ещё более напряжённой после *второго* хода (Рис. П5.16).

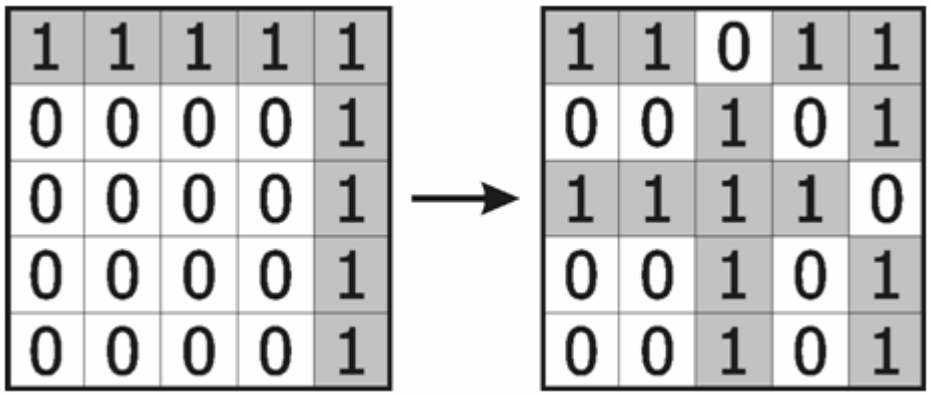


Рис. П5.16. Усугубляем!

А творится это «безобразие» потому, что теперь в каждой колонке и строке *нечётное* количество клеток, а это значит – все клетки с единицей будут нечётными, а с нулём – чётными. Это легко проверить для позиции, возникающей из нулевой после *первого* хода (впрочем, все дальнейшие рассуждения верны и для этой позиции). Если *второй* ход сделать в клетку с 0 (она чётная), то в ней появится 1, при этом изменятся числа в 9 клетках соответствующей колонки и строки. До второго хода в них было чётное количество единиц (ходим в чётную клетку!) и нечётное количество нулей (всего чисел 9 – нечётное число), а после него будет нечётное количество единиц, так что клетка превратится в *нечётную*. Можете проверить, что, сделав ход в нечётную клетку (с единицей), вы получите *чётную* клетку. Естественно, третий ход только подтвердит эту печальную традицию.

Так как чётность клеток *изменяется* при каждом ходе, то невозможно определить, в какие из них были сделаны предыдущие ходы. Зато мы теперь можем легко показать существование *неразрешимых* позиций. Выставим на нулевое поле одну единицу. Например, в левый верхний угол (Рис. П5.17).

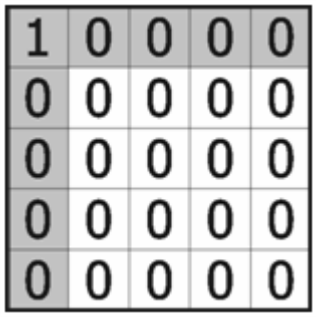


Рис. П5.17. Первый ход – и сразу в угол!

В этой позиции нечётной оказывается не только клетка с единицей, но и ещё 8 клеток с нулями! Но мы доказали, что во всех позициях, возникающих из нулевой, клетки с нулями могут быть исключительно *чётными*. Таким образом, указанная позиция не может получиться из нулевой и, соответственно, не может быть сведена к ней никакими ходами. Вы можете легко найти *множество* неразрешимых позиций, если запустите программу, очистите левое поле (все фишки станут *зелёными*), установите его размеры 5 на 5 клеток и будете выставлять *красные* фишки левой кнопкой мыши с шифтом. Если в какой-либо позиции хотя бы одна красная фишка станет *чётной* (не будет выделена синим квадратом) или хотя бы одна зелёная нечётной (будет выделена) – значит, вы получили *неразрешимую* позицию. Смело заключайте пари, что никто эту задачу не решит (то есть не сможет по правилам игры перевернуть все фишки зелёной стороной вверх). Так развлекался ещё сам Лойд со своими *пятнашками* – почему бы и вам не пошутить (разумеется, речь идёт о шуточном пари!)?

Вследствие того, что при случайном выставлении фишек на поле размером 5 на 5 клеток нередко будут возникать *неразрешимые* позиции, нам и пришлось изменить алгоритм задания новой позиции в процедуре *NewPlay*. Произвольное выставление фишек мы заменили случайными ходами, которые, безусловно, никак не могут привести к неразрешимым позициям. Это верно и для *второго* варианта игры, когда совершается переход от одной ненулевой позиции к другой ненулевой позиции, так как и в этом случае конечная позиция возникает как результат случайных ходов на поле-образце.

Итак, любая задача в *нашей* игре (имеется в виду поле 5 х 5) решается, вопрос только в том – *как* следует ходить, чтобы её решить. Нетрудно убедиться, что ходы в нечётные клетки (с красными фишками), в отличие от поля 4 х 4, к успеху не ведут – число нечётных клеток хаотически изменяется, но отнюдь не убывает! Всё, что мы *нажили непосильным трудом*, *пошло прахом* – нужно искать *новые* подходы к решению задач на поле 5 х 5 клеток. Или вы уже смекнули, как необходимо действовать в этом случае? Если это так, то можно только позавидовать вашей прозорливости. А всех остальных я приглашаю на следующий тур наших изысканий.

Попробуем воспользоваться нашим умением решать задачи на *чётных* полях. Выделим на поле 5 х 5 клеток квадрат размером 4 х 4 клетки. Сделать это можно четырьмя разными способами, но нам удобнее совместить оба поля верхними левыми углами. Для примера рассмотрим реальную задачу, предложенную нашей программой.

Мы будем выделять нечётные клетки только для *малого* квадрата, исключая остальные клетки *большого* квадрата при подсчёте единиц (красных фишек) в соответствующих колонках и строках (Рис. П5.18).

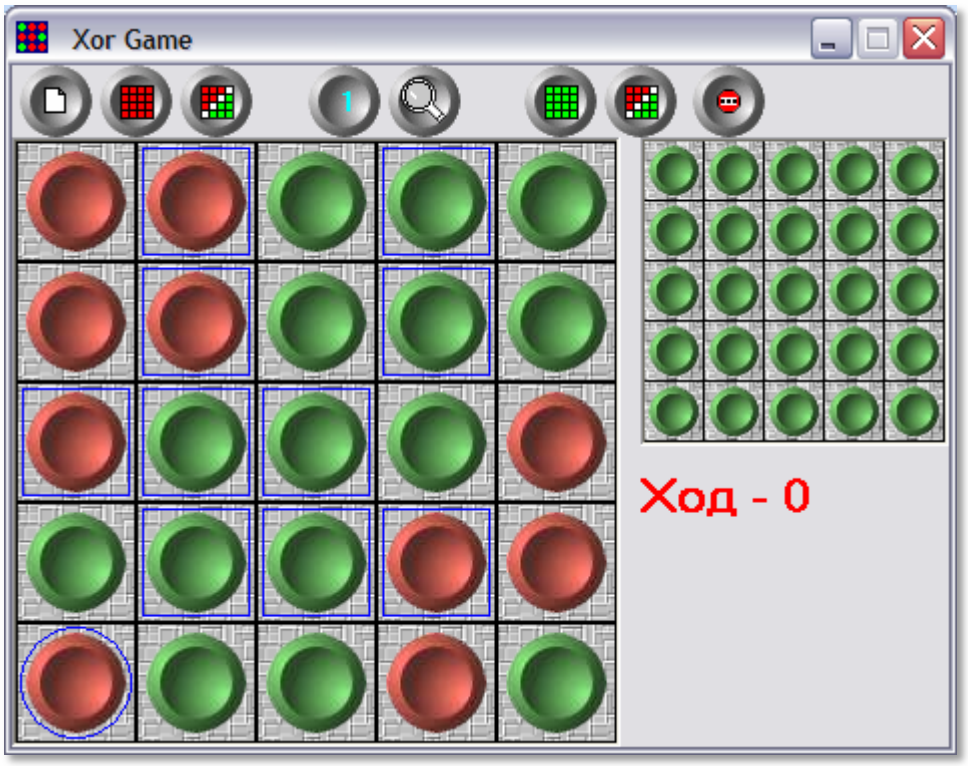


Рис. П5.18. Изыскательские работы продолжаются!

Если этого не сделать, то решение квадрата 4 х 4 будет затруднено (придётся нам самим отыскивать в нём нечётные клетки). Разумеется, чётность клеток при таком подсчёте изменится, что и отражено на рисунках.

Само решение задачи на поле 4 на 4 клетки не должно вызвать у вас никаких затруднений. Как и прежде, ходить нужно в *нечётные*, выделенные клетки. Ясно видно, что потребуется 10 ходов, так как выделено именно столько клеток. В результате получится такая позиция (Рис. П5.19).

0	0	0	0	?
0	0	0	0	?
0	0	0	0	?
0	0	0	0	?
?	?	?	?	0

Рис. П5.19. Начинаем с нулей!

Число в *правом нижнем* углу останется тем же самым, а все остальные числа по периметру поля могут измениться (решая задачу на доске 4 x 4 мы обязаны инвертировать полные ряды длиной 5 клеток, иначе мы не сможем распространить наше решение на полный квадрат 5 x 5), поэтому числа в этих клетках заменены *знаком вопроса*. Попытаемся определить, какие числа за ними скрываются.

Так как в левом верхнем углу находится 0 (*чётная* клетка), то последние числа в левой колонке и в верхней строке либо оба нули, либо оба единицы (в противном случае упомянутый нуль будет стоять в нечётной клетке, что невозможно). Идя от верхнего левого угла вниз по диагонали и рассуждая по пути аналогично, мы придём к выводу, что периферийные клетки (исключая угловую с нулём) содержат: либо все единицы, либо все нули.

Пусть это будут *единицы*, тогда каждая из периферийных клеток должна быть *чётной*, так как в последней колонке и нижней строке будет по 4 единицы (Рис. П5.20).

0	0	0	0	1
0	0	0	0	1
0	0	0	0	1
0	0	0	0	1
1	1	1	1	0

Рис. П5.20. Единицы?

Но такая позиция *невозможна*, поэтому во всех периферийных клетках должны находиться *нули*, и задача решена (Рис. П5.21).

0	0	0	0	0
0	0	0	0	0
0	0	0	0	0
0	0	0	0	0
0	0	0	0	0

Рис. П5.21. Нули!

Если бы в правом нижнем углу стояла *единица*, то, сделав ход в эту клетку, мы пришли бы к рассмотренному нами решению. Но можно поступить иначе: сначала решить квадрат 4 x 4, тогда во всех периферийных клетках останутся *единицы* (доказательство этого очевидного факта проводится аналогично предыдущему случаю) и потребуется ещё *один* ход для окончания решения.

Пользуясь этим алгоритмом, вы легко справитесь с любой задачей и на поле 5 x 5 клеток. Правда, процедуры рисования фишек *dgPoleDrawCell* и *dgPole2DrawCell* не умеют выделять на нём квадрат 4 x 4. Прежде чем мы улучшим эти процедуры, нужно ещё решить, что делать с полем 7 x 7 клеток. Горячие головы, наверное, уже успели подумать, что и на нём следует выделить квадрат 4 на 4 клетки. А что вы прикажете делать с оставшимися *тремя* рядами периферийных клеток? И не надейтесь, что они так же благополучно «обнулятся», как и на поле 5 x 5 клеток. И тем не менее всё просто – вместо поля 4 x 4 нужно взять поле 6 x 6, вот тогда останется *один* периферийный слой фишек – со всеми вытекающими отсюда последствиями (если вы захотите ввести в программу режим 9 x 9 клеток, то сами знаете, какое нужно взять для него меньшее поле).

Добавьте в упомянутые процедуры такой код (выделенные строки):

```
//ОТРИСОВАТЬ ЯЧЕЙКУ ИГРОВОГО ПОЛЯ
procedure TForm1.dgPoleDrawCell(Sender: TObject; ACol, ARow: Inte-
ger;
  Rect: TRect; State: TGridDrawState);
…

  //выводить подсказки для нечётных позиций?
  if (dgPole.ColCount= 5) and ((ACol>3) or (ARow>3)) then exit;
  if (dgPole.ColCount= 7) and ((ACol>5) or (ARow>5)) then exit;
  if dgPole.ColCount= 5 then m:= 3
  else if dgPole.ColCount= 7 then m:= 5
  else m:= dgPole.ColCount-1;
…
end;  //dgPoleDrawCell

//ОТРИСОВАТЬ ЯЧЕЙКУ ПОЛЯ-ОБРАЗЦА
procedure TForm1.dgPole2DrawCell(Sender: TObject; ACol, ARow: In-
teger;
  Rect: TRect; State: TGridDrawState);
…
  //выводить подсказки для нечётных позиций?
  if (dgPole2.ColCount= 5) and ((ACol>3) or (ARow>3)) then exit;
  if (dgPole2.ColCount= 7) and ((ACol>5) or (ARow>5)) then exit;
  if dgPole2.ColCount= 5 then m:= 3
  else if dgPole2.ColCount= 7 then m:= 5
  else m:= dgPole2.ColCount-1;
```

```
…
end;  //dgPole2DrawCell
```

Порадовавшись своим успехам, давайте всё же задумаемся, а *всегда* ли наш алгоритм решения задач на *нечётных* полях находит *самое короткое* из всех возможных решений, как в случае чётных полей. Можно вручную перебирать ходы во множестве задач, чтобы проверить наши сомнения. Но это процесс трудоёмкий и нетворческий. Поэтому мы напишем процедуру поиска кратчайшего перехода между двумя позициями, взяв за основу ту, что сослужила нам хорошую службу в программе *Флип-Флоп*. Но мы изменим её так, чтобы она находила лучшее решение *самостоятельно*, без нашего участия. Задав такое число ходов, которого заведомо хватит для решения задачи, мы всё остальное поручим процедуре поиска. Например, в разобранной задаче для решения потребуется не более 10 ходов, поэтому разумно ограничить глубину поиска именно этим числом. Найдя такое решение, процедура продолжит поиск более коротких. И если они существуют, то обязательно будут обнаружены. Так, в нашем случае довольно *шести* ходов (нумерация клеток в этот раз более привычная для непрограммистов – начинается с *единицы*):

```
1.          2 - 3.
2.          5 - 3.
3.          3 - 4.
4.          1 - 5.
5.          3 - 5.
6.          5 - 5.
```

Кроме указанного, имеется ещё 3 варианта решения задачи такой же длины. Таким образом, мы делали 4 *лишних* хода. Это свидетельствует о том, что наш алгоритм не позволяет находить лучшее решение (по крайней мере, для некоторых задач). Причину этого «дефекта» несложно обнаружить, если внимательно посмотреть на последовательность ходов, выданную процедурой. В ней присутствуют ходы в периферийные клетки, которых мы избегали.

Длина решения на поле 4 x 4 определяется числом *нечётных* клеток на нём и может быть уменьшена хорошим ходом в *периферийную* зону. Для поля 5 x 5 клеток нужно проверить 8 клеток (угловая не в счёт), сделав в каждую из них ход и подсчитав нечётные клетки на поле 4 x 4. Для нашей задачи следует выбрать ход в нижний левый угол, что уменьшит число нечётных клеток на поле 4 x 4 до четырёх. При этом в правом нижнем углу будет стоять уже не 0, а 1, так что для решения задачи понадобится 6 ходов:

```
1.        1 - 5.
2.        3 - 1.
3.        3 - 2.
```

```
4.        1 - 3.
5.        4 - 3.
6.        5 - 5.
```

Это один из вариантов (второй), выданный процедурой поиска, которая в программе выглядит так:

```
//НАЙТИ ПЕРЕХОД ОТ ОДНОЙ ПОЗИЦИИ К ДРУГОЙ
procedure TForm1.Solution(nStep: integer);
//nStep2 - количество шагов (ходов)
label
  nextHod, nextCell;
var
  Hod: integer;
  nCells: integer;
  nCell: array[0..99] of integer;
  Pos: array[0..99] of TPole;
  Move: array[0..99] of integer;
  x, y: byte;
  w,h: integer;
  variant: integer;
  arr1, arr2: TPole;
  s: string;

  //ПРОВЕРИТЬ, НЕ ПОЛУЧИЛАСЬ ЛИ КОНЕЧНАЯ ПОЗИЦИЯ
  function IsOK: Boolean;
  var
    i, j: integer;
  begin
    Result:= True;
    for j:= 0 to h-1 do
      for i:= 0 to w-1 do
        if Pos[Hod][i,j]<> arr2[i,j] then
          begin Result:= False; exit end
  end;

  //записать ходы в протокол:
  procedure SavePos;
  var
    n, i, j: integer;
    x, y: integer;
    s: string;
  begin
    frmProtokol.listbox1.items.add('');
    for n:= 1 to Hod do begin
      s:='';
      x:= Move[n] mod w; y:= Move[n] div w;
      frmProtokol.listbox1.items.add('Шаг= '+ inttostr(n) +
'  Ход= '+ inttostr(x+1)+ ' ' +inttostr(y+1));

      for j:= 0 to h-1 do begin
        for i:= 0 to w-1 do
```

```
          if Pos[n][i,j]= BLACK then //- клетка фигуры
            s:= s+'1'
          else
            s:= s+'0';     //- клетка "фона"
        frmProtokol.listbox1.items.add(s);
        s:= '';
      end;
    end;
  end;

begin
  //очистить протокол:
  frmProtokol.listbox1.Clear;
  //скопировать массивы полей:
  arr1:= masPole; arr2:= masPole2;
  //размеры полей:
  w:= dgPole.ColCount; h:= dgPole.RowCount;
  //число найденных решений:
  variant:= 0;
  //число клеток на поле:
  nCells:= w*h;
  //глубина поиска:
  Hod:= 0;
  //начальная позиция:
  Pos[0]:= arr1;
  nCell[0]:= -1; Move[0]:= -1;
//делаем следующий ход:
nextHod:
  application.ProcessMessages;
  if flgStop= True then begin flgStop:= False; exit end; //-
остановить поиск
  Inc(Hod); nCell[Hod]:= nCell[Hod-1];  Pos[Hod]:= Pos[Hod-1];
//переходим к следующей клетке поля:
nextCell:
  inc(nCell[Hod]);
  if nCell[Hod]>= nCells then begin//-прошли всё поле
    dec(Hod);
    if Hod< 1 then begin
      if variant= 0 then
        s:= 'Задача решений не имеет!'
      else begin
        frmProtokol.Show;
        s:= 'Найдены все варианты - ' + inttostr(variant);
      end;
      ShowMessage(s);
      exit;
    end;
    Goto nextCell
  end;

  x:= nCell[Hod] mod w; y:= nCell[Hod] div w;
  //выполнить ход и запомнить позицию:
  Pos[Hod]:=DoMove(Pos[Hod-1], x, y);
```

```
  //запомнить ход:
  Move[Hod]:= nCell[Hod];

  if IsOK then begin //- нашли!
    //не искать длинных вариантов:
    if hod< nStep then begin
      nStep:= hod;
      variant:= 0;
    end;
    inc(variant);
    frmProtokol.listbox1.items.add('Вариант - '+ intto-
str(variant)+ '  Длина - ' + inttostr(hod));
    frmProtokol.listbox1.items.add('----------------------');
    savepos;
  end;

  if Hod< nStep then goto NextHod;
  goto nextCell
end;  // Solution
```

Для вывода найденных вариантов ходов мы добавим к проекту новую *форму – frmProtokol* (Рис. П5.22).

```
  Width = 205
  Height = 321
  BorderIcons = [biSystemMenu]
  Caption = '  Варианты'
```

с компонентом *список* - 

ListBox1:

```
  Left = 0
  Top = 0
  Width = 197
  Height = 293
  Font.Charset = RUSSIAN_CHARSET
  Font.Name = 'Courier New'
```

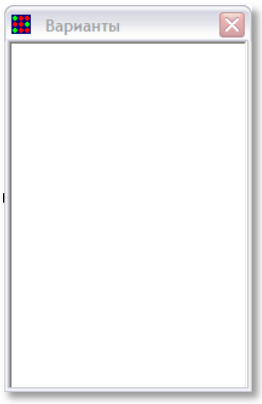


Рис. П5.22. Контора пишет!

В модуле формы добавьте строку:

```
implementation
uses ProtokolUnit;
```

Код модуля формы протокола (набирать вам его не придётся):

```
unit ProtokolUnit;
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
Dialogs, StdCtrls;

 type
  TfrmProtokol = class(TForm)
    ListBox1: TListBox;

  private
    { Private declarations }

  public
    { Public declarations }
  end;

var
  frmProtokol: TfrmProtokol;

implementation

{$R *.DFM}

end.
```

Если при поиске решения вы зададите *большую* глубину перебора, то этот процесс может и затянуться. Поэтому необходимо предусмотреть возможность выхода из процедуры поиска. В этих двух строках проверяется, не приняла ли переменная *flgStop* значение *True*. И если это так, поиск превращается:

```
  application.ProcessMessages;
  if flgStop= True then begin flgStop:= False; exit end; //-
остановить поиск
```

Объявите эту переменную в разделе глобальных переменных:

```
flgStop: boolean= false;
```

Чтобы остановить решение, нужно нажать кнопку со значком *СТОП*. Прежде чем написать процедуру для обработки нажатия на кнопку, мы поговорим немного о самих *кнопках*.

Выберем для нашей программы кнопки подходящей формы. Давайте сделаем их в том же стиле, что и фишки, чтобы они гармонировали друг с другом. Среди стандартных кнопок *Delphi* вы таких не найдёте, но ведь мы можем и сами сделать *круглые* кнопки! Воспользуемся тем же приёмом, что и в *метрисе*, но сделаем их более удобными для работы.

Роль кнопок будет играть компонент *TImage*, в который нужно загрузить изображение соответствующей кнопки. Как их подготовить, я уже рассказывал, хотел бы только напомнить, что изображения кнопок должны быть в формате *BMP* на *белом* фоне.

Начнём с *последней* кнопки на форме, которая и будет останавливать поиск решений (Рис. П5.23).



Рис. П5.23. Стоповая кнопка

Установите на форме компонент *imgStop* типа *TImage* и задайте ему свойства:

```
Left = 340
Top = 0     (это свойство для всех кнопок одинаково)
Width = 36
Height = 36    (размеры всех кнопок 36 x 36 пикселей)
Cursor = crHandPoint          (и курсор - тоже)
Hint = 'Остановить решение'
ParentShowHint = False
ShowHint = True
AutoSize = True
Picture - загрузите нужную картинку.
Transparent = True
OnMouseDown = imgStopMouseDown
OnMouseUp = imgStopMouseUp
```

Потребуется и *вторая* картинка для этой кнопки, она будет заменять первую, когда кнопка находится в *нажатом* положении (Рис. П5.24).



Рис. П5.24. Полный стоп!

За основу картинок для всех кнопок можно взять те же фишки, но меньшего размера и другого цвета (Рис. П5.25).



Рис. П5.25. Заготовки для кнопок

В каком-нибудь графическом редакторе на них нужно наложить значки-пиктограммы, как это принято в современных приложениях.

Поскольку нам придётся выводить на кнопку *два* разных изображения, то их где-то следует хранить. Например, можно приспособить для этого тот же компонент *TImage*. Тогда на каждую кнопку придётся ещё пара этих компонентов. Так, жизнедеятельность нашей кнопки обеспечивают *Image15* и *Image16*. Их свойства мало чем отличаются от компонента с кнопкой:

```
    Width = 36
    Height = 36
    AutoSize = True
    Picture – картинка с кнопкой в нормальном (Image15) и нажатом
(Image16) состоянии
    Transparent = True
    Visible = False
```

При нажатии на кнопку одна картинка на ней заменяется другой:

```
//ОСТАНОВИТЬ ПОИСК РЕШЕНИЙ
procedure TForm1.imgStopMouseDown(Sender: TObject; Button: TMouse-
Button;
  Shift: TShiftState; X, Y: Integer);
begin
  if GameState<> gsSolution then exit;
  imgStop.Picture.Assign(Image16.Picture);
end;  // imgStopMouseDown
```

А при отпускании возвращается первоначальная картинка и устанавливается флаг выхода из процедуры поиска:

```
procedure TForm1.imgStopMouseUp(Sender: TObject; Button: TMouse-
Button;
  Shift: TShiftState; X, Y: Integer);
begin
  if GameState<> gsSolution then exit;
  imgStop.Picture.Assign(Image15.Picture);
  flgStop:= True;
end;  // imgStopMouseUp
```

Если вы захотите сразу же и выйти из программы, то в *процедуре FormClose* также установите флаг выхода из процедуры поиска (на всякий случай – вдруг в это время программа углубилась в поиск перехода от одной позиции к другой):

```
//ЗАКРЫТЬ ФОРМУ
procedure TForm1.FormClose(Sender: TObject; var Action: TCloseAc-
tion);
begin
  flgStop:= True;
end; // FormClose
```

Глубина поиска решения задаётся *кнопкой imgStep*:

```
    Left = 148
    Hint = ' Глубина перебора'
    OnMouseDown = imgStepMouseDown
    OnMouseUp = imgStepMouseUp
```

Картинки для неё хранятся в компонентах *Image11* и *Image12*. Это просто кнопка, без картинок, так как на ней будет выводиться максимальное число ходов при поиске решения. Поэтому поместите на кнопку метку *lblStep*, которая и будет показывать это число:

```
    Left = 164
    Top = 10
    Width = 9
    Height = 16
    Cursor = crHandPoint (дублируем свойства самой кнопки, ведь
мышка во время нажатия кнопки может оказаться и на метке)
    Hint = 'Глубина перебора'
    Caption = '1'
    Font.Color = clAqua
    Font.Height = -13
    Font.Name = 'MS Sans Serif'
    Font.Style = [fsBold]
    ShowHint = True
    Transparent = True
    OnMouseDown = imgStepMouseDown
    OnMouseUp = imgStepMouseUp
```

При каждом нажатии на кнопку глубина перебора будет увеличиваться или уменьшаться на единицу в допустимых пределах:

```
//ЗАДАТЬ ГЛУБИНУ ПЕРЕБОРА
procedure TForm1.imgStepMouseDown(Sender: TObject; Button: TMouse-
Button;
  Shift: TShiftState; X, Y: Integer);
begin
  //заменить картинку:
```

```
  imgStep.Picture.Assign(Image12.Picture);
  //если кнопка мыши нажата с шифтом, то уменьшить глубину перебо-
ра:
  if (ssLeft in Shift) and (ssShift in Shift) then begin
    if Step> 1 then dec(Step);
    //показать новое значение:
    lblStep.Caption:= inttostr(Step);
    //отцентровать число:
    lblStep.Left:= imgStep.Left+ (imgStep.Width- lblStep.Width)
div 2+2;
    exit;
  end;

  //если кнопка мыши нажата без шифта, то увеличить глубину пере-
бора:
  if ssLeft in Shift then begin
    if Step< dgPole.ColCount*dgPole.RowCount then inc(Step);
    lblStep.Caption:= inttostr(Step);
    lblStep.Left:= imgStep.Left+ (imgStep.Width- lblStep.Width)
div 2+2;
  end;
end; // imgStepMouseDown
```

Отпуская ее, мы возвращаем на место *серую* кнопку:

```
procedure TForm1.imgStepMouseUp(Sender: TObject; Button: TMouse-
Button;
  Shift: TShiftState; X, Y: Integer);
begin
  imgStep.Picture.Assign(Image11.Picture);
end; // imgStepMouseUp
```

Глубина перебора хранится в глобальной *переменной*

```
Step: integer= 1;
```

Поиск решения активизируется *кнопкой imgSolution*:

```
    Left = 188
    Hint = ' Найти решение '
    OnMouseDown = imgSolutionMouseDown
    OnMouseUp = imgSolutionMouseUp
```

Картинки хранятся в компонентах *Image13* и *Image14* (Рис. П5.26).



Рис. П5.26. «Решительные» кнопки

При нажатии (точнее, при последующем отпускании) кнопки *Solution* программа переводится в режим *gsSolution* и не реагирует на любые команды, кроме остановки поиска:

```
//НАЙТИ РЕШЕНИЕ ЗАДАЧИ
procedure TForm1.imgSolutionMouseDown(Sender: TObject; Button:
TMouseButton; Shift: TShiftState; X, Y: Integer);
begin
  if GameState= gsSolution then exit;
  imgSolution.Picture.Assign(Image14.Picture);
end; // imgSolutionMouseDown

procedure TForm1.imgSolutionMouseUp(Sender: TObject; Button:
TMouseButton;  Shift: TShiftState; X, Y: Integer);
begin
  if GameState= gsSolution then exit;
  imgSolution.Picture.Assign(Image13.Picture);
  //перевести программу в режим поиска:
  GameState:= gsSolution;
  //искать решение на заданную глубину:
  solution(step);
  //перевести программу в режим ожидания ввода команд:
  GameState:= gsWait;
end; // imgSolutionMouseUp
```

Самая первая кнопка на форме – *imgNewGame* – запускает новую игру:

```
    Left = 4
    Hint = ' Новая игра '
    OnMouseDown = imgNewGameMouseDown
    OnMouseUp = imgNewGameMouseUp
```

Ее картинки хранятся в компонентах *Image1* и *Image2* (Рис. П5.27).



Рис. П5.27. «Запускательные» кнопки

```
//НОВАЯ ИГРА
procedure TForm1.imgNewGameMouseDown(Sender: TObject; Button:
TMouseButton;  Shift: TShiftState; X, Y: Integer);
begin
  if GameState= gsSolution then exit;
  imgNewGame.Picture.Assign(Image2.Picture);
end; // imgNewGameMouseDown

procedure TForm1.imgNewGameMouseUp(Sender: TObject; Button:
TMouseButton; Shift: TShiftState; X, Y: Integer);
begin
  if GameState= gsSolution then exit;
```

```
  NewPlay;
  imgNewGame.Picture.Assign(Image1.Picture);
end; // imgNewGameMouseUp
```

Действие этой кнопки заключается в вызове процедуры *NewPlay*, которую мы уже разобрали.

При *ручной* расстановке фишек на поле могут быть очень полезны следующие кнопки.

Вторая справа кнопка – *imgRedOnly* – переворачивает все фишки на игровом поле *красной* стороной вверх:

```
    Left = 44
    Hint = ' Все фишки красные '
    OnMouseDown = imgRedOnlyMouseDown
    OnMouseUp = imgRedOnlyMouseUp
```

Картинки хранятся в компонентах *Image3* и *Image4* (Рис. П5.28).

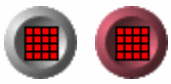


Рис. П5.28. Вгоняем в краску!

```
//ВСЕ ФИШКИ НА ЛЕВОМ ПОЛЕ - КРАСНЫЕ
procedure TForm1.imgRedOnlyMouseDown(Sender: TObject; Button:
TMouseButton; Shift: TShiftState; X, Y: Integer);
begin
  if GameState= gsSolution then exit;
  imgRedOnly.Picture.Assign(Image4.Picture);
end; // imgRedOnlyMouseDown

procedure TForm1.imgRedOnlyMouseUp(Sender: TObject; Button:
TMouseButton; Shift: TShiftState; X, Y: Integer);
var
  i, j: integer;
begin
    if GameState= gsSolution then exit;
    for j:= 0 to dgPole.RowCount-1 do
      for i:= 0 to dgPole.ColCount-1 do
        masPole[i,j]:= BLACK;

    dgPole.Invalidate;
    imgRedOnly.Picture.Assign(Image3.Picture);
end; // imgRedOnlyMouseUp
```

Следующая кнопка – *imgInverse* – изменяет цвет фишек на игровом поле на противоположный:

```
    Left = 84
    Hint = ' Инвертировать '
    OnMouseDown = imgInverseMouseDown
    OnMouseUp = imgInverseMouseUp
```

Картинки хранятся в компонентах *Image5* и *Image6* (Рис. П5.29).



Рис. П5.29. «Инверсные» кнопки

```
//ИЗМЕНИТЬ ЦВЕТ ФИШЕК НА ИГРОВОМ ПОЛЕ НА ПРОТИВОПОЛОЖНЫЙ
procedure TForm1.imgInverseMouseDown(Sender: TObject; Button:
TMouseButton; Shift: TShiftState; X, Y: Integer);
begin
  if GameState= gsSolution then exit;
  imgInverse.Picture.Assign(Image6.Picture);
end; // imgInverseMouseDown

procedure TForm1.imgInverseMouseUp(Sender: TObject; Button:
TMouseButton; Shift: TShiftState; X, Y: Integer);
var
  i, j: integer;
begin
  if GameState= gsSolution then exit;
  for j:= 0 to dgPole.RowCount-1 do
    for i:= 0 to dgPole.ColCount-1 do
      masPole[i,j]:= BLACK- masPole[i,j];

  dgPole.Invalidate;
  imgInverse.Picture.Assign(Image5.Picture);
end; // imgInverseMouseUp
```

Точно так же действует и кнопка *imgInverse2*, только она меняет цвет фишек на правом поле-образце:

```
    Left = 296
    Hint = ' Инвертировать '
    OnMouseDown = imgInverse2MouseDown
    OnMouseUp = imgInverseMouse2Up
```

Картинки хранятся в тех же компонентах.

```
//ИНВЕРТИРОВАТЬ ЦВЕТ ФИШЕК НА ПРАВОМ ПОЛЕ
procedure TForm1.imgInverse2MouseDown(Sender: TObject; Button:
TMouseButton; Shift: TShiftState; X, Y: Integer);
begin
  if GameState= gsSolution then exit;
  imgInverse2.Picture.Assign(Image6.Picture);
end; //imgInverse2MouseDown
```

```
procedure TForm1.imgInverse2MouseUp(Sender: TObject; Button:
TMouseButton; Shift: TShiftState; X, Y: Integer);
var
  i, j: integer;
begin
  if GameState= gsSolution then exit;
  for j:= 0 to dgPole2.RowCount-1 do
    for i:= 0 to dgPole2.ColCount-1 do
      masPole2[i,j]:= BLACK- masPole2[i,j];

  dgPole2.Invalidate;
  imgInverse2.Picture.Assign(Image5.Picture);
end; // imgInverse2MouseUp
```

И вот мы добрались до последней кнопки – *imgGreenOnly*. Она родственница кнопки *imgRedOnly*, но переворачивает фишки на правом поле и *зелёной* стороной вверх:

```
    Left = 256
    Hint = ' Все фишки зелёные  '
    OnMouseDown = imgGreenOnlyMouseDown
    OnMouseUp = imgGreenOnlyUp
```

Картинки хранятся в компонентах *Image7* и *Image8* (Рис. П5.30).

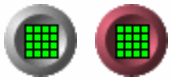


Рис. П5.30. Озеленяем!

```
//ПЕРЕВЕРНУТЬ ФИШКИ НА ПРАВОМ ПОЛЕ ЗЕЛЁНОЙ СТОРОНОЙ ВВЕРХ
procedure TForm1.imgGreenOnlyMouseDown(Sender: TObject; Button:
TMouseButton; Shift: TShiftState; X, Y: Integer);
begin
  if GameState= gsSolution then exit;
  imgGreenOnly.Picture.Assign(Image8.Picture);
end; // imgGreenOnlyMouseDown

procedure TForm1.imgGreenOnlyMouseUp(Sender: TObject; Button:
TMouseButton; Shift: TShiftState; X, Y: Integer);
var
  i, j: integer;
begin
  if GameState= gsSolution then exit;
  for j:= 0 to dgPole2.RowCount-1 do
    for i:= 0 to dgPole2.ColCount-1 do
      masPole2[i,j]:= WHITE;

  dgPole2.Invalidate;
  imgGreenOnly.Picture.Assign(7.Picture);
end; // imgGreenOnlyMouseDown
```

Согласитесь, такие кнопки удобнее размещать на форме, ведь каждая из них содержится только в *одном* компоненте *TImage* (ещё два компонента нужны для хранения картинок, но их можно убрать в любое место на форме и больше не трогать), а не в двух, как в *метрисе*.

Если вы помните, мы ещё не написали процедуры для двух пунктов меню. *Один* позволяет при решении задач возвращаться на один ход *назад* (вплоть до исходной позиции):

```
//ВЕРНУТЬ ХОД
procedure TForm1.miUndoClick(Sender: TObject);
begin
  if GameState= gsSolution then exit;
  if hod< 1 then exit;

  masPole:= DoMove(masPole, Moves[hod].x, Moves[hod].y);
  dgPole.Invalidate;
  dec(hod);
  lblHod.Caption:=  'Ход - ' + inttostr(hod);
end; // miUndoClick
```

Второй, наоборот, даёт возможность двигаться *вперёд* по уже сделанным, но отменённым ходам:

```
//ОТМЕНИТЬ ПОСЛЕДНИЙ ВОЗВРАТ ХОДА
procedure TForm1.miRedoClick(Sender: TObject);
begin
  if GameState= gsSolution then exit;

  inc(hod);
  masPole:= DoMove(masPole, Moves[hod].x, Moves[hod].y);
  dgPole.Invalidate;
  lblHod.Caption:= 'Ход - ' + inttostr(hod);
end;  // miRedoClick
```

Для большего удобства при решении задач на нечётных досках можно добавить подсказки и для периферийных клеток. Как вы знаете, удачный ход в периферийную клетку уменьшает число нечётных клеток на меньшем поле, а, значит, и общее число ходов, что позволяет найти лучшее решение задачи. Конечно, можно последовательно делать ходы в периферийные клетки и подсчитывать число нечётных клеток, а затем выбирать ту клетку, которая даёт оптимальный результат, но правильнее будет переложить эту рутину на плечи компьютера. Правда, нам придётся поработать головой, но зато потом пользоваться программой будет значительно удобнее.

В первую очередь, нам потребуется *функция*, умеющая подсчитывать количество нечётных клеток на *малом* поле (его ширина и высота на одну клетку меньше игрового поля). Для этого мы проверяем каждую клетку малого поля и, если она нечётная, увеличиваем на 1 возвращаемое значение:

```
//ПОДСЧИТАТЬ ЧИСЛО НЕЧЁТНЫХ КЛЕТОК НА ЗАДАННОМ ПОЛЕ
function TForm1.CalcOdd(Pos: TPole): Integer;
var
  i, j, i1, j1: integer;
  n: integer;
begin
    Result:= 0;
    //размеры малого поля:
    m:= dgPole.ColCount-2;
    //по всем клеткам малого поля:
    for j:= 0 to m do
      for i:= 0 to m do begin
        //число нечётных клеток в соотв. колонке и строке:
        n:= 0;
        for j1:= 0 to m do
          for i1:= 0 to m do
            if (i1= i) or (j1= j) then n:=n+ Pos[i1,j1];

        if odd(n) then inc (Result);
      end;
end; // CalcOdd
```

В процедуре выбора «хороших» периферийных клеток мы сначала определяем, сколько нечётных клеток содержится на малом поле в текущей позиции. Затем последовательно выполняем ход во все периферийные клетки (за исключением угловой) и вновь подсчитываем нечётные клетки. Если их число уменьшилось, то мы запоминаем новое значение. После обхода всех периферийных клеток мы проверяем, удалось ли улучшить результат. Если удалось, отмечаем соответствующие периферийные клетки *кружком* (в отличие от квадрата на малом поле):

```
//НАЙТИ ХОДЫ В ПЕРИФЕРИЙНЫЕ КЛЕТКИ
procedure TForm1.PerifHint(massiv:TPole; Pole: TDrawGrid);
var
  i, j, m, n: integer;
  min0, min: integer;
  Position: TPole;
  Rect: TRect;
begin
  if miHint.Checked= False then exit; //- не выводить подсказки
  //размеры полей:
  m:= Pole.ColCount-1;
```

```
  //подсчитать число нечётных клеток на малом поле в текущей позиции:
  min0:= CalcOdd(massiv);
  min:= min0;

  //проверить периферийные клетки:
  for j:= 0 to m do
    for i:= 0 to m do
      if ((i= m) or (j= m)) and (i<>j) then begin
        //выполнить ход:
        Position:=DoMove(massiv, i, j);
        //подсчитать число нечётных клеток на малом поле:
        n:= CalcOdd(Position);
        //запомнить минимальное число нечётных клеток
        //после хода в периферийную клетку:
        if n< min then min:= n;
       end;

  if min= min0 then exit; //- не удалось уменьшить число ходов

  //отметить периферийные клетки, уменьшающие число ходов:
  for j:= 0 to m do
    for i:= 0 to m do
      if ((i= m) or (j= m)) and (i<>j) then begin
        //выполнить ход:
        Position:=DoMove(massiv, i, j);
        //подсчитать число нечётных клеток на малом поле:
        n:= CalcOdd(Position);
        if n= min then begin
          //нарисовать окружность:
          Rect:= Pole.CellRect(i, j);
          //dgPole.Canvas.Brush.Style:= bsClear;
          //цвет линий:
          Pole.Canvas.Pen.Color:= clBlue;
          Pole.Canvas.Pen.Width:= 1;
          Pole.Canvas.Ellipse(Rect.left+2, Rect.top+2,Rect.right-
2, Rect.bottom-2);
        end;
      end;
end;  // PerifHint
```

Осталось решить, из какого места программы должна вызываться описанная процедура. Удобнее всего это делать в тех процедурах, которые отвечают за прорисовку ячеек игрового и «образцового» полей. Придётся добавить толику кода:

```
//ОТРИСОВАТЬ ЯЧЕЙКУ ИГРОВОГО ПОЛЯ
procedure TForm1.dgPoleDrawCell(Sender: TObject; ACol, ARow: Inte-
ger;
  Rect: TRect; State: TGridDrawState);
```

```
…
  //выводить подсказки для нечётных позиций?
  if (dgPole.ColCount= 5) and ((ACol>3) or (ARow>3)) then
    begin  PerifHint(masPole, dgPole); exit; end;
  if (dgPole.ColCount= 7) and ((ACol>5) or (ARow>5)) then
    begin  PerifHint(masPole, dgPole); exit; end;

…

//ОТРИСОВАТЬ ЯЧЕЙКУ ПОЛЯ-ОБРАЗЦА
procedure TForm1.dgPole2DrawCell(Sender: TObject; ACol, ARow: In-
teger;
  Rect: TRect; State: TGridDrawState);

…

  //выводить подсказки для нечётных позиций?
  if (dgPole2.ColCount= 5) and ((ACol>3) or (ARow>3)) then
    begin  PerifHint(masPole2, dgPole2); exit; end;
  if (dgPole2.ColCount= 7) and ((ACol>5) or (ARow>5)) then
    begin  PerifHint(masPole2, dgPole2); exit; end;
```

После этих усовершенствований вы легко сможете решить наилучшим образом любую задачу и на нечётных полях 5 х 5 и 7 х 7. Правда, речь до сих пор шла о режиме «полного переворота» на таких полях, но и режим «перехода» вам также доступен. Действуйте на малом поле совершенно по тому же «сценарию», что и на чётных полях. В итоге вы либо сразу совершите переход из одной позиции в другую, либо вам придётся сделать ещё один ход в правый нижний угол, если фишки в этой клетке на игровом и образцовом полях отличаются по цвету.



Мы выяснили, что «нерешаемая» позиция на нечётном поле не может быть сведена к нулевой или любой другой решаемой. Но это ничуть не мешает переводу одной нерешаемой позиции в другую нерешаемую. Наша программа не предлагает для решения такие задачи, но вы можете самостоятельно ввести такие позиции на обоих полях. Не кажется ли вам, что здесь есть над чем подумать?

Например, эти две прекрасные симметричные (но «неправильные») позиции разделяют всего 4 хода (Рис. П5.31):

1. 3 – 1 (3 –2).
2. 1 – 3 (2 –3).
3. 5 – 3 (4 – 3).
4. 3 – 5 (3 – 4).

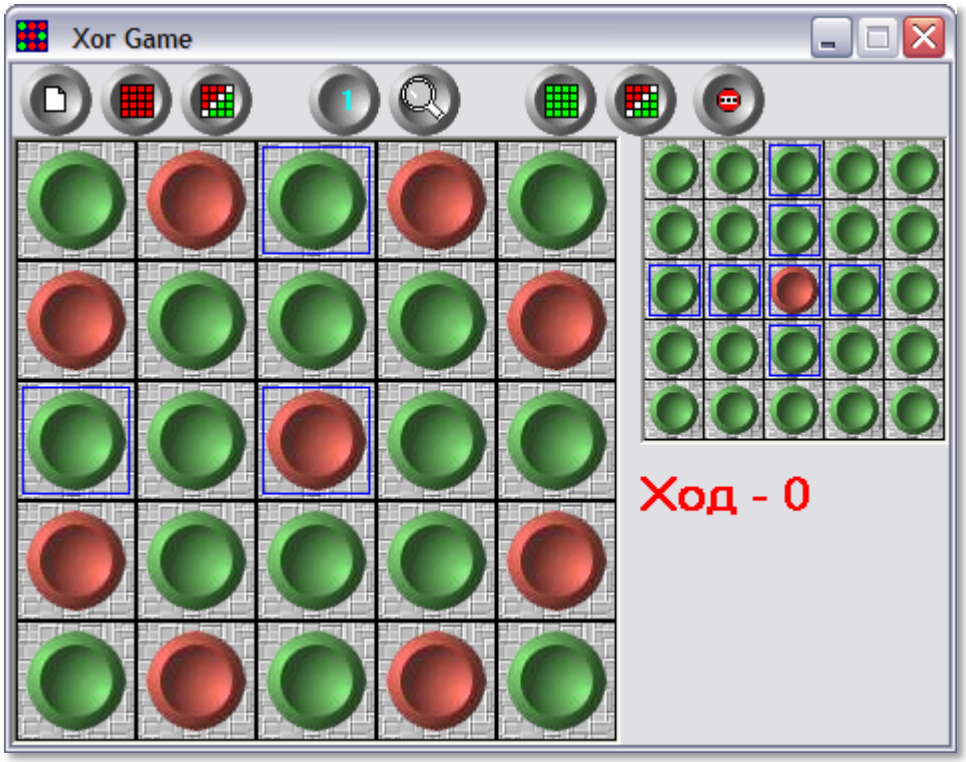


Рис. П5.31. Решаем нерешаемые задачи!

Левая позиция получена из правой методом «блуждания» по правому полю за 9 ходов, но для решения задачи хватит и 4-х, причём самый короткий переход можно найти, пользуясь всё тем же алгоритмом, что и раньше. Таким образом, если между двумя позициями существует переход, то мы сумеем его отыскать. Это значит, что позицию на игровом поле можно задавать произвольно, а позицию на правом поле следует образовывать из позиции на правом с помощью серии случайных ходов. Ясно, что между этими позициями переход всегда возможен, а вот полный переворот фишек на зелёную сторону – не всегда. Если у вас есть желание, вы можете изменить процедуру *NewPlay* так, чтобы в режиме перехода использовались бы и «нерешаемые» позиции. Кстати говоря, среди них вы найдёте множество красивых, симметричных позиций, которые можно использовать в качестве заданий, создав на диске библиотеку избранных задач. Как это сделать, читайте в следующем разделе.

Помоги себе сам, или Даём справочку

Спасение утопающих –
дело рук самих утопающих.

И.Ильф, Е.Петров

Попробуйте доказать, что существуют пары нерешаемых позиций, которые *невозможно* перевести друг в друга. Это трудно, зато я подсоблю вам в написании помощи для этой игры. Вызываться она будет при выборе пункта меню *Правила*. Но в этот раз мы «окажем помощь» более профессионально – с привлечением *Справочной системы Windows*. Такую «справку» вы можете увидеть практически во всех «серьёзных» программах, сделать её нетрудно и самим, особенно если начать с самого простого варианта.

Вывод справки на экран осуществляется в процедуре-обработчике щелчка на пункте меню *Помощь*. Для этого мы используем функцию *Windows API WinHelp*, в которой надлежит указать имя файла справки (у нас это будет *XorGame.hlp*), который должен находиться в той же папке, что и выполняемый файл программы.

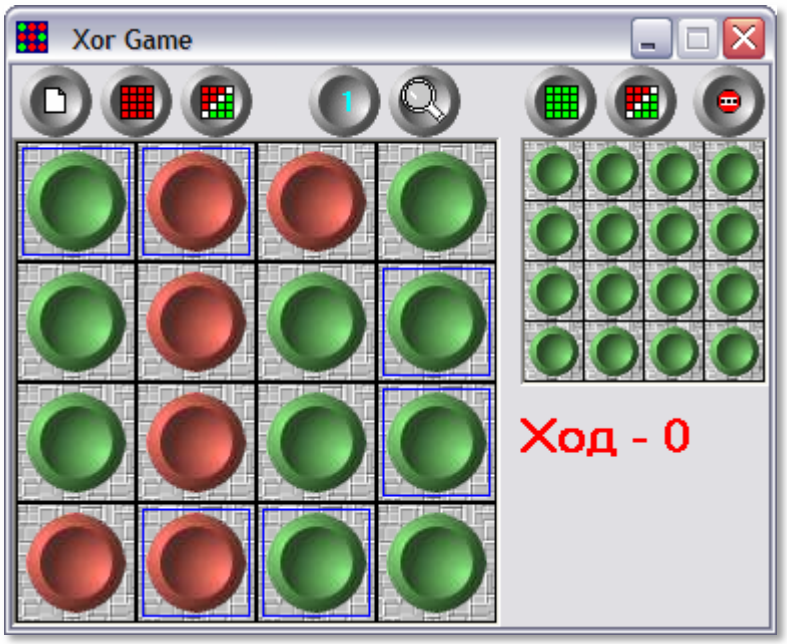
```
//ПОКАЗАТЬ ФАЙЛ СПРАВКИ
procedure TForm1.miHelpClick(Sender: TObject);
begin
  WinHelp(handle, PChar('XorGame.hlp'), HELP_CONTENTS,0)
end;  // miHelpClick
```

Вот только где взять файл с расширением *hlp*? Оказывается, его нетрудно получить из текстового файла в формате *rtf,* который можно создать во многих текстовых редакторах, хотя бы в *WordPad* - он поставляется с *ОС Windows* - или *MS Word*.

В тексте справки допустимо использовать любые шрифты, а также добавлять картинки. Например, короткая справка для нашей игры могла бы выглядеть так:

XorGame Help

Логическая игра-головоломка, в которой требуется перейти от начальной позиции к конечной за минимальное число ходов, переворачивая на квадратном поле двухцветные фишки.

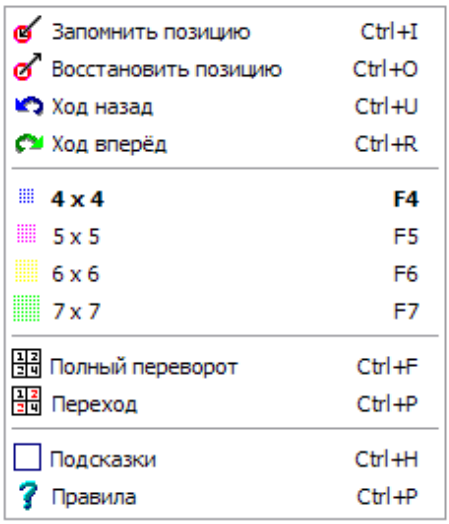


Ход состоит в переворачивании любой фишки на левом, игровом поле (зелёная фишка становится красной, красная – зелёной), но при этом переворачиваются и все другие фишки в той же колонке и строке. На правом поле вы можете видеть конечную позицию.

Управляющие кнопки:

- *начать новую игру,*
- *перевернуть все фишки игрового поля на красную сторону,*
- *изменить цвет всех фишек игрового поля на противоположный,*
- *установить глубину поиска решения задачи,*
- *найти решение задачи,*
- *перевернуть все фишки правого поля на зелёную сторону,*
- *изменить цвет всех фишек правого поля на противоположный,*
- *остановить поиск решения задачи.*

Вы также можете управлять программой, вызывая всплывающее меню (для этого нажмите правую кнопку мыши):



Первая группа команд поможет вам найти самое короткое решение задачи. Вы можете запомнить любую позицию на поле, а затем восстановить её, а также вернуться назад на любое число ходов (или вернуть уже сделанные ходы при откате).

Вторая группа команд устанавливает размеры полей. Разумеется, чем больше размер поля, тем труднее будет решить задачу.

У игры имеется две разновидности - полный переворот, когда все фишки на игровом поле нужно перевернуть зелёной стороной вверх, и переход, когда следует перевернуть фишки на игровом поле так, как они расположены на правом поле-образце.

Если включен режим подсказки, то программа будет отмечать те клетки игрового поля, на которые нужно ходить.

Закончив набор, сохраните файл в формате *RTF* (в этом случае имя файла справки на диске будет *XorGame.rtf*).

Затем запустите программу *Microsoft Help Workshop* под именем *hcw.exe* (в *Delphi XE3* она отсутствует, но в папке *Tools* вы найдёте её копию, взятую из *Delphi 5*). Выполните команду *File|New...* и в открывшемся диалоговом окне нажмите кнопку *OK* (по умолчанию будет создаваться *Help Project*). В следующем диалоговом окне перейдите в папку с программой и введите имя файла справки (без расширения). Снова нажмите кнопку *OK*, и вы окажетесь в окне с вашим проектом. *Help Workshop* самостоятельно запишет в него строки:

```
[OPTIONS]
LCID=0x419 0x0 0x0 ; Русский
REPORT=Yes
```

Сохраните файл на диске (к заданному вами имени файла добавится расширение *hpj*), а потом откройте в любом текстовом редакторе (лучше всего подойдёт *Блокнот* из *ОС Windows*) и допишите следующие строки:

```
TITLE=XorGame Help
COPYRIGHT=В.Рубанцев, 2012
HLP=.\XorGame.hlp
[FILES]
XorGame.RTF
```

TITLE – это название справки, которое появится на панели задач при запуске помощи.

COPYRIGHT – фамилия автора программы, она выводится при выборе пункта меню *Справка|Версия* в окне справочной системы *Windows* с вашей справкой.

HLP – размещение файла помощи и его имя.

В разделе *[FILES]* следует указать имя файла справки с расширением *RTF*.

То же самое можно сделать и в самой программе *Help Workshop*, нажав кнопку *Options* (Рис. П5.32).

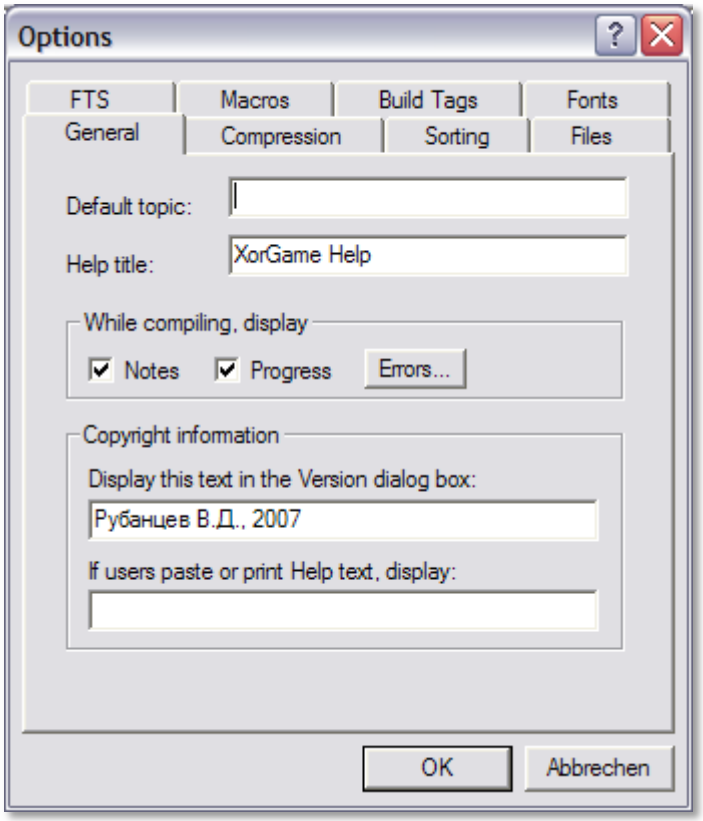


Рис. П5.32. Создаём справочный файл

Сохраните отредактированный файл проекта. Осталось только скомпилировать его в файл помощи. Запустите *Microsoft Help Workshop* (или дважды щёлкните на вашем файле с расширением *hpj* на диске). В открывшемся окне вы увидите название файла помощи и его текст. Нажмите кнопку *Save and Compile* (она находится в правом нижнем углу формы), и через мгновение на диске появится файл помощи с расширением *hlp*. Щёлкните на нём дважды и убедитесь, что справочный файл выглядит именно так, как вы и планировали. Если необходимо, внесите изменения и дополнения в файл справки и проекта, а после этого перекомпилируйте проект с помощью *Microsoft Help Workshop*.

Точно такое же окно с вашей справкой будет появляться, когда пользователь выберет пункт *Правила* игры *XorGame*. Вероятно, такой способ создания справки к программам вам покажется более привлекательным, чем рассмотренные нами ранее, - сил и времени набор текста справки отнимает немного, а результаты дает впечатляющие! Тем более что вы можете добавить к справке и другую полезную информацию. Сделать это уже сложнее, поэтому мы пока ограничимся только простейшими возможностями *Справочной системы Windows*.



Для операционной системы *Windows Vista* в папке **Tools** вы найдёте файл **WinHelp32.rar**. Распакуйте его и установите нужное обноовление.

А для операционной системы *Windows 7* поищите обновления в Интернете самостоятельно!



Исходный код программы находится в папке **xor**.

Есть варианты, или *ShortGame*

Кроме игры *Флип-Флоп*, есть ещё несколько подобных игр, из которых мы подробно остановимся только на одной, а короткому обзору остальных посвятим следующий раздел.

Основной признак всех игр-оборотней – это переворот двухцветных фишек на квадратном поле (в игре *Камешки* поле не квадратное, а прямоугольное, но вы можете сделать его и крестообразным и гексагональным…), поэтому близкие родственники игры *Флип-Флоп* отличаются от неё только правилом (порядком, шаблоном) переворачивания фишек.



Если нарушить это условие и взять фишки с тремя или четырьмя сторонами, окрашенными в два или более цветов, то можно создать очень любопытные разновидности игр-оборотней. Если пойти ещё дальше и *выставлять* на поле по одной фишке, то можно получить прекрасную игру *Logos*, о которой будет рассказано в четвертом факультативе.

Так, в игре *Blinkin'* (версия 1.0, 1999 г.) (Рис. П5.33), которая послужит нам основой для очередной игры, фишки переворачиваются точно так же, как и в *XorGame*, но только на расстояние в 1 фишку во все стороны от «центральной». Поэтому вполне естественно назвать нашу программу *коротким Xor'ом*, или *ShortGame*.



На рисунке ясно видно, что «нормальный» ход возможен только в *середине* поля, а у *границ* и в *углах* шаблон очень сильно искажается. Но вы легко сможете уберечь шаблон от «ампутации», если возьмёте тороидальное поле (в виде бублика). На экране оно так и останется квадратным (прямоугольным), но первый и последний ряды (столбцы и колонки) нужно считать *соседними*, то есть «лишние» фишки должны появляться на *другой* стороне поля. Запрограммировать эти особенности игры на торе не составит большого труда.

Вы также можете попытаться изменить длину «лучей» у стандартного шаблона или сделать их диагональными. А вдруг вам удастся придумать новые игры-оборотни!

Для преобразования достаточно исправить процедуры выполнения хода, а всё остальное можно оставить практически без изменений. Но этот путь

слишком лёгкий для нас, а, значит, и не очень интересный и не очень полезный, поэтому мы дополнительно изменим дизайн программы и добавим ещё несколько симпатичных новинок.

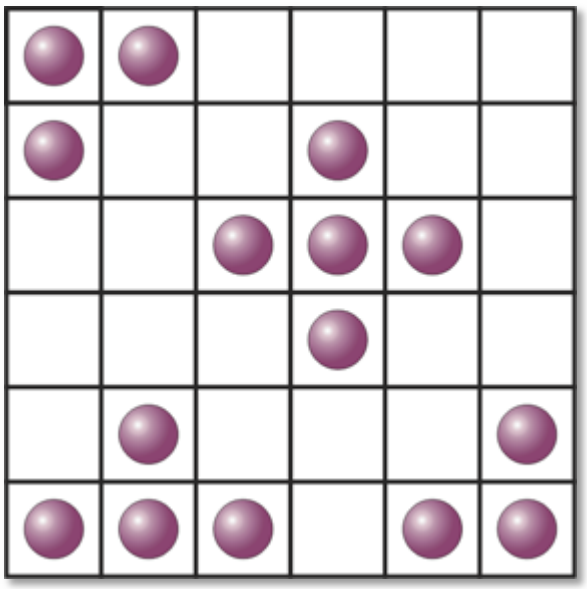


Рис. П5.33. Варианты ходов (шаблоны для переворачивания фишек) в игре *Blinkin'*

Зачем нужны фиксированные ряды?

При изучении свойств компонента *TDrawGrid*, я отмечал, что *фиксированные* ряды могут использоваться для *оцифровки* игровых полей. В этой игре я покажу вам два способа оцифровки. В *первом* все числа и буквы будут изображены на растровых картинках, во *втором* – просто напечатаны стандартным шрифтом. Вы можете оценить достоинства и недостатки обоих способов и выбрать подходящий. Хотя если вы можете нарисовать в графическом редакторе красивые картинки, то второй способ вам никогда и не понадобится – результаты его применения далеки от требований сегодняшнего дня.

Чтобы не переписывать весь код приложения заново, откройте в *Delphi* проект *XorGame* и сохраните все модули, а затем и сам проект под новыми именами (кроме формы протокола). Теперь вы можете исправлять и добавлять новый код в проект игры *ShortGame* (Рис. П5.34).

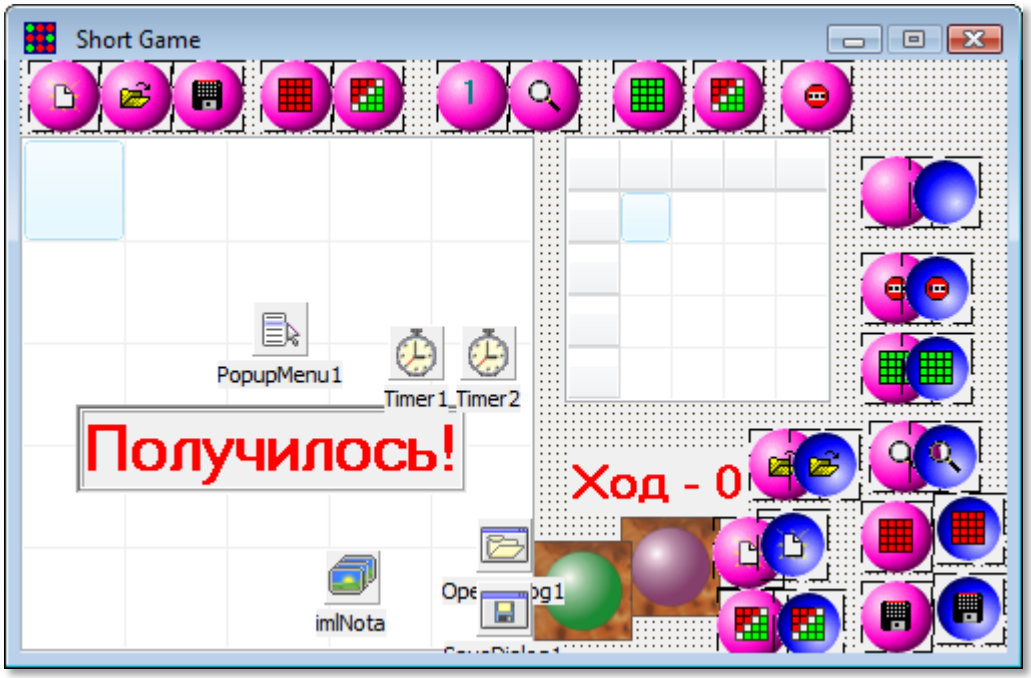


Рис. П5.34. Главная форма проекта в окне *Конструктора формы*

Изменения начнём с *сеток*. Свойства сетки *dgPole* (указаны только те свойства, которые нужно исправить):

```
    Top = 38
    Width = 257
    Height = 257  (немного опустили сетку, чтобы кнопки чувствовали себя более уютно, размеры сетки увеличились из-за того, что потребовались ещё ряды для оцифровки)
    ColCount = 5
    RowCount = 5
    DefaultColWidth = 50
    DefaultRowHeight = 50  (размеры клеток пришлось уменьшить, так как в игре будут использоваться поля до 9 x 9 клеток)
    GridLineWidth = 1  (очертим все клетки – фон клеток стал темнее, и чёрные линии будут плохо заметны)
   Options = [goVertLine, goHorzLine]
```

Свойства сетки *dgPole2*:

```
    Left = 272
    Top = 38
    Width = 133
    Height = 133
    Cursor = crHandPoint
    ColCount = 5
    RowCount = 5
    DefaultColWidth = 25
    DefaultRowHeight = 25
    FixedColor = clSilver
```

```
    FixedCols = 1
    FixedRows = 1
    GridLineWidth = 1
    Options = [goFixedVert-
Line,goFixedHorzLine,goVertLine,goHorzLine]
```

На *правой* сетке мы разместим два фиксированных ряда – горизонтальный и вертикальный – для вывода оцифровки. Это сделано только для примера – никаких преимуществ этот способ оцифровки не даёт.

Перейдём к процедурам рисования клеток игрового поля. Ради разнообразия мы заменим фишки шариками на «паркете» (Рис. П5.35).



Рис. П5.35. Шариковые фишки

Для оцифровки потребуется довольно много картинок (Рис. П5.36).



Рис. П5.36. Картинки для оцифровки полей

Как и раньше, их можно загрузить в компоненты *TImage*, расположенные на форме, но картинок так много, что разумнее использовать для их хранения специальный компонент *TImageList* 

(страница *Win32* в *Палитре компонентов*). Впрочем, это создаёт и определённые неудобства, в чём вы скоро сами сможете убедиться.

Итак, щёлкните на этом компоненте в *Палитре*, и он послушно возникнет на форме. Сразу же установите размеры картинок 50 x 50 пикселей, иначе при загрузке они будут искажены:

```
    Name = imlNota
    Height = 50
    Width = 50
```

А вот теперь откройте *Редактор*, дважды щёлкнув компонент *TImageList* (тот, который на форме, естественно) (Рис. П5.37).

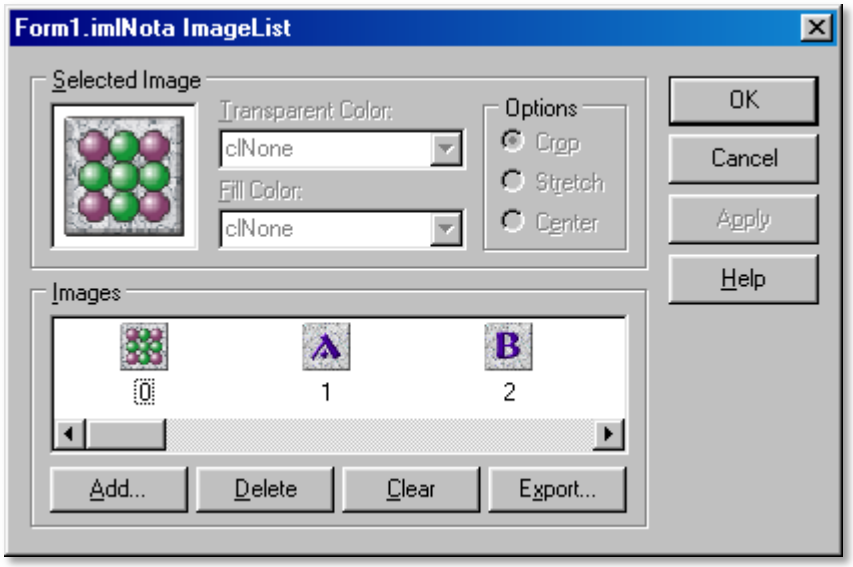


Рис. П5.37. Картинки в списке!

Нажимайте кнопку *Add...* и загружайте заранее подготовленные картинки с оцифровкой. Их примерный облик представлен выше. Порядок загрузки картинок – тоже. Под каждой картинкой вы увидите число – это *индекс* соответствующей картинки в списке, он используется для ее вывода на канву компонента. Как только список заполнится, закрывайте *Редактор* и переходите к написанию кода.

В процедуре *dgPoleDrawCell* вычисляется «номер» *m* клетки с координатами *ACol, ARow* на игровом поле, а по нему определяется дальнейшая судьба клетки. В *верхнюю* строку выводятся картинки с индексами 0-9 (*буквы*), а в *первую* колонку - картинки с индексами 10-18 (*цифры*), во все остальные – шарики соответствующего цвета.

Чтобы перенести картинку из компонента *TImageList* на канву сетки, приходится использовать его метод

```
procedure Draw(Canvas: TCanvas; X, Y, Index: Integer; Enabled:
Boolean=True);
```

Из чего следует, что вы не сможете непосредственно применить более мощные методы самой канвы.

```
//ОТРИСОВАТЬ ЯЧЕЙКУ ИГРОВОГО ПОЛЯ
procedure TForm1.dgPoleDrawCell(Sender: TObject; ACol, ARow: Inte-
ger;
  Rect: TRect; State: TGridDrawState);
var
  n, m: integer;
  r, dr: TRECT;
```

```
begin
  //фишка в ячейке:
  n:= masPole[ACol, ARow];
  //размеры картинок:
  r:= Bounds(0, 0, 50, 50);
  dr:= Bounds(Rect.Left, Rect.Top, 50, 50);
  m:= 100 * ARow + ACol;
  case m of
    0..9: //- верхняя строка
      imlNota.Draw(dgPole.Canvas, dr.left, dr.top, m);
    100, 200, 300, 400, 500, 600, 700, 800, 900: //- первая
колонка
      imlNota.Draw(dgPole.Canvas, dr.left, dr.top, m div 100 + 9);
    else begin
      case n of
        WHITE:    //- белая фишка
          dgPole.canvas.CopyRect(dR, imgWhite.Canvas, R);
        BLACK:    //- чёрная фишка
          dgPole.canvas.CopyRect(dR, imgBlack.Canvas, R);
      end
    end
  end;
end;  //dgPoleDrawCell
```

В остальном эта процедура не таит в себе никаких неприятных сюрпризов. Обратите внимание на то, что подсказки теперь не выводятся. Правила выбора ходов для игры *XorGame* здесь работать не будут, поэтому их нужно заменить другими. А вот какими – попробуйте найти сами!

В правой сетке имеются фиксированные ряды для печати оцифровки, но – на всякий случай – предусмотрена и возможность вывода тех же картинок с буквами и цифрами, что и на игровом поле, но в масштабе 1 : 2.

Для букв и цифр выбран *жёлтый* цвет на сером фоне, но вы легко можете заменить их другими. Вывод символов осуществляется методом канвы

```
procedure TextRect(Rect: TRect; X, Y: Integer; const Text:
string);
```

Сделать это чрезвычайно просто, и если вас удовлетворяет результат, то вы можете смело пропустить следующий абзац.

Вывод картинок на канву *правой* сетки осложняется тем, что размер картинок в компоненте *ImageList1* отличается от размеров клеток на правом поле. Конечно, можно заготовить ещё один комплект букв и цифр подходящего размера, но это не лучший вариант. Если бы картинки хранились в компоненте *TImage*, то всё бы решалось так же просто, как и для картинок с шариками. Однако для картинок с оцифровкой недоступен

метод канвы *StretchDraw*. Вместо него мы должны применять метод *Draw* компонента *TImageList*, который не позволяет масштабировать изображения. Придётся создать в памяти компьютера растр *bmp* по размеру картинок, скопировать в него изображение из компонента *TImageList* и только потом перенести в клетку в уменьшенном масштабе. Чтобы воспользоваться этим способом, раскомментируйте строчки, в которых присутствует *bmp* и закомментируйте вывод текста.

```
//ОТРИСОВАТЬ ЯЧЕЙКУ ПОЛЯ-ОБРАЗЦА
procedure TForm1.dgPole2DrawCell(Sender: TObject; ACol, ARow: In-
teger;
  Rect: TRect; State: TGridDrawState);
var
  n, m: integer;
  r, dr: TRECT;
  //bmp: TBitmap;
  ch: string;
begin
  //фишка в ячейке:
  n:= masPole2[ACol, ARow];
  //размеры картинок:
  r:= Bounds(0, 0, 25, 25);
  dr:= Bounds(Rect.Left, Rect.Top, Rect.Right-
Rect.Left,Rect.Bottom-Rect.Top);

  m:= 100 * ARow + ACol;

  //создать промежуточный растр:
  //bmp:= TBitmap.Create;
  //bmp.Width:= 50;
  //bmp.Height:= 50;


  //оцифровка:
  dgPole2.Canvas.Font.Color:= clYellow;
  dgPole2.Canvas.Font.Name:='MS Sans Serif';
  dgPole2.Canvas.Font.Style:= [fsBold];
  dgPole2.Canvas.Font.Size:= -14;
  dgPole2.Canvas.Brush.Color:= clGray;

  case m of
    0: //- угловая клетка
      begin
        imlNota.Draw(bmp.Canvas, 0, 0, 0);
        dgPole2.canvas.StretchDraw(dR, bmp);
      end;

    1..9: //- верхняя строка
      begin
        //imlNota.Draw(bmp.Canvas, 0, 0, m);
        //dgPole2.canvas.StretchDraw(dR, bmp);
```

```
        ch:= letter[m];
        with dR, dgPole2.Canvas do
          textrect(dr, left+(right-left-textwidth(ch)) div 2,
             top+(bottom-top-textheight(ch)) div 2, ch);
      end;

    100, 200, 300, 400, 500, 600, 700, 800, 900: //- первая
колонка
      begin
        //imlNota.Draw(bmp.Canvas, 0, 0, m div 100 + 9);
        //dgPole2.canvas.StretchDraw(dR, bmp);
        ch:= inttostr(m div 100);
        with  dR,dgPole2.Canvas do
          textrect(dr, left+(right-left-textwidth(ch)) div 2,
top+(bottom-top-textheight(ch)) div 2, ch);
      end;
    else begin
      case n of
        WHITE:    //- белая фишка
          dgPole2.canvas.StretchDraw(dR,
imgWhite.Picture.Graphic);
        BLACK:   //- чёрная фишка
          dgPole2.canvas.StretchDraw(dR,
imgBlack.Picture.Graphic);
      end
    end
  end;
  //bmp.Free;
end;  //dgPole2DrawCell
```

Если вы помните, у нас остались ещё две картинки в компоненте *ImageList1* – первая и последняя. Они предназначены для заполнения пустой клетки в левом верхнем углу сетки, в которой отсутствуют символы оцифровки. Можно ограничиться только первой картинкой, которая выводится в процедуре *dgPoleDrawCell*, но мы сделаем простейшую *анимацию*. Всякое оживление изображения заключается в последовательной смене картинок на экране. У нас таких картинок всего две, и мы будем поочерёдно выводить их в угловую клетку. Промежуток времени между сменой изображений проще всего отмерять *таймером*. Один таймер у нас имеется, но он уже занят, поэтому добавим к проекту второй таймер *Timer2*. Интервал установите в 500 мс (полсекунды). Чтобы следить за тем, какая в данный момент картинка на экране, мы заведём глобальную *переменную:*

```
flgFlash: boolean= false;
```

При каждом срабатывании таймера мы будем изменять её значение на противоположное, что позволит нам определить индекс картинки для вывода.

Дважды щёлкните на значке компонента *Timer2* и в заготовке процедуры напишите код:

```
//МИГАЮЩИЙ КВАДРАТИК
procedure TForm1.Timer2Timer(Sender: TObject);
var
  rect: TRect;
  n: integer;
begin
  rect:= dgPole.CellRect(0,0);
  flgFlash:= not flgFlash;
  if flgFlash= false then n:= 0 else n:= 19;
  imlNota.Draw(dgPole.Canvas, rect.left, rect.top, n);
end;  //Timer2Timer
```



Немного усложнив эту процедуру, вы сможете сделать и более качественную анимацию. Смотрите пример в главе, посвящённой игре *Logos*.

Кнопки

В этой игре мы сделаем кнопки по тому же принципу, что и раньше, но при нажатии они будут изменять не только цвет, но и *размеры*. Изготовить их можно в растровом графическом редакторе и записать в формате *BMP* с одноцветным фоном. Хранить их придётся на форме в компонентах *TImage* (но можно и «по-новому» - в компоненте *TListImage*).

Итак, загружаем картинки с кнопками в соответствующие компоненты (их свойства те же, что и в предыдущей игре) (Рис. П5.38).



Сразу установим свойства компонентов *Image* для «настоящих» кнопок, которые находятся над сетками:

```
Name = imgNewGame
Left = 4
Top = 0  (такое же значение и у всех других кнопок)
Width = 36
Height = 36
AutoSize = True
Cursor = crHandPoint
Hint = 'Новая игра'
ShowHint = True
```

```
Transparent = True
OnMouseDown = imgNewGameMouseDown
OnMouseUp = imgNewGameMouseUp
```

 - Image9  - Image10

```
Name = imgOpen
Left = 40
Hint = ' Загрузить задачу с диска '
OnMouseDown = imgOpenMouseDown
OnMouseUp = imgOpenMouseUp
```

 - Image18  - Image19

```
Name = imgSavePos
Left = 76
Hint = ' Записать текущую позицию '
OnMouseDown = imgSavePosMouseDown
OnMouseUp = imgSavePosMouseUp
```

 - Image3  - Image4

```
Name = imgRedOnly
Left = 120
```

 - Image5  - Image6

```
Name = imgInverse
Left = 156
```

 - Image11  - Image12

```
Name = imgStep
Left = 208
```

 - Image13  - Image14

```
Name = imgSolution
Left = 244
```

 - Image7  - Image8

Name = imgGreenOnly
Left = 296

 - Image15  - Image16

Name = imgStop
Left = 380

Рис. П5.38. Новые кнопки!

Теперь наполним глубоким смыслом и содержанием *процедуры*, которые будут вызываться при нажатии кнопок. Некоторые из них останутся в неизменном виде, поэтому их код здесь не приводится.

Минимальным изменениям подверглась процедура для кнопки *imgStep*. Они касаются только центрирования надписи с числом шагов:

```
//ЗАДАТЬ ГЛУБИНУ ПЕРЕБОРА
procedure TForm1.imgStepMouseDown(Sender: TObject; Button: TMouse-
Button;
  Shift: TShiftState; X, Y: Integer);
Begin
…
    //отцентровать число:
    lblStep.Left:= imgStep.Left+ (imgStep.Width- lblStep.Width)
div 2-1;
…
    lblStep.Left:= imgStep.Left+ (imgStep.Width- lblStep.Width)
div 2-1;
    lblStep.Top:= imgStep.Top+ (imgStep.Height- lblStep.Height)
div 2-1;
…
end; //imgStepMouseDown

procedure TForm1.imgStepMouseUp(Sender: TObject; Button: TMouse-
Button;
  Shift: TShiftState; X, Y: Integer);
begin
…
  lblStep.Left:= imgStep.Left+ (imgStep.Width- lblStep.Width) div
2;
  lblStep.Top:= imgStep.Top+ (imgStep.Height- lblStep.Height) div
2;
end; //imgStepMouseUp
```

При срабатывании кнопки *imgRedOnly* позиция на поле изменяется, поэтому её следует запомнить:

```
procedure TForm1.imgRedOnlyMouseUp(Sender: TObject; Button:
TMouseButton;
  Shift: TShiftState; X, Y: Integer);
…
    dgPole.Invalidate;
    MemoryIn;
  imgRedOnly.Picture.Assign(Image3.Picture);
end; //imgRedOnlyMouseUp
```

Эту же строку добавьте в процедуры *imgGreenOnlyMouseUp*, *imgInverseMouseUp* и *imgInverse2MouseUp*.

В этой игре добавились ещё две кнопки. *Одна* позволяет сохранить позицию на игровом поле:

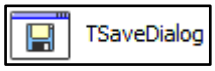
```
//ЗАПИСАТЬ ПОЗИЦИЮ НА ДИСК
procedure TForm1.imgSavePosMouseDown(Sender: TObject; Button:
TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  imgSavePos.Picture.Assign(Image19.Picture);
end; //imgSavePosMouseDown

procedure TForm1.imgSavePosMouseUp(Sender: TObject; Button:
TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  imgSavePos.Picture.Assign(Image18.Picture);
  SavePos;
end; //imgSavePosMouseUp
```

Вторая – загрузить позицию с диска:

```
//ЗАГРУЗИТЬ ЗАДАЧУ
procedure TForm1.imgOpenMouseUp(Sender: TObject; Button: TMouse-
Button;
  Shift: TShiftState; X, Y: Integer);
begin
  if GameState= gsSolution then exit;
  imgOpen.Picture.Assign(Image9.Picture);
  Open;
end; //imgOpenMouseUp

procedure TForm1.imgOpenMouseDown(Sender: TObject; Button: TMouse-
Button;
  Shift: TShiftState; X, Y: Integer);
begin
  if GameState= gsSolution then exit;
  imgOpen.Picture.Assign(Image10.Picture);
  //Open;
end; //imgOpenMouseUp
```

Для их работы потребуются компоненты *TSaveDialog* 

и *TOpenDialog* 

(страница **Dialogs** *Палитры компонентов*).

Чтобы записать позицию на игровом поле, достаточно скопировать массив поля *masPole* в файл на диске. Обратите внимание, что для хранения задач (позиций) нужно завести *папку Figures* в той папке, в которой находится выполняемый файл программы. Второе замечание: название задачи (имя файла), для удобства, выводится в заголовке формы. Так делается во многих программах, и мы не будем отступать от этой доброй традиции. Но придётся объявить *константу* с названием программы:

```
NAME_PROG =' ShortGame';
```

и *переменную* – для хранения имени файла:

```
NameFig: string= '';
```

И мне осталось напомнить вам, что процедура *SavePos* также должна быть объявлена – в описании типа формы:

```
  private
    { Private declarations }
    procedure Prepare(Pole: TDrawGrid; ColCount, RowCount: inte-
ger);
    function IsReady: Boolean;
    procedure NewPlay;
    procedure Solution(nStep: integer);
    procedure Ready;
    function DoMove(arr: TPole; x, y: integer): TPole;
    procedure MemoryIn;
    procedure SavePos;

//ЗАПИСАТЬ ПОЗИЦИЮ НА ИГРОВОМ ПОЛЕ
procedure TForm1.SavePos;
var
  F: textfile;
  fn,s: string;
  i,j: integer;
  ch: char;
begin
  savedialog1.DefaultExt:='txt';
  savedialog1.Filter:='Задачи (*.txt)|*.TXT';
  savedialog1.FilterIndex:=1;
  s:=extractfilepath(application.exename)+'Figures\';
  savedialog1.InitialDir:= s;
  savedialog1.Title:='Запишите позицию на диск';
  if NameFig<>'' then savedialog1.filename:= NameFig
```

```
  else savedialog1.filename:='temp.txt';
  if not savedialog1.Execute then exit;

  //имя конечного файла:
  fn:= savedialog1.filename;
  NameFig:=fn;
  assignfile(f,fn);
  rewrite(f);

  //записать позицию:
  for j:= 1 to dgPole.RowCount-1 do begin
    s:='';
    for i:= 1 to dgPole.ColCount-1 do begin
      if masPole[i][j]= 1 then ch:='1'
      else ch:='0';
      s:=s+ch;
    end;
   writeln (f,s);
  end;

  closefile(f);
  form1.caption:= NAME_PROG + '   [' + NameFig + ']';
  messagebeep(0)
end;  //SavePos;
```

Обратное действие, то есть загрузка задачи с диска, ненамного сложнее: дополнительно придётся *проверить* данные задачи (файл с условием задачи - обычный текстовый файл, поэтому его можно набрать в любом текстовом редакторе; это удобно, но может привести к ошибкам) и настроить размеры полей, ведь они могут измениться. Труднее всего выделить пункт в меню, соответствующий размерам нового поля, но и с этой проблемой можно справиться.

```
//ЗАГРУЗИТЬ ЗАДАЧУ
procedure TForm1.Open;
var
  s: string;
  F: TextFile;
  nLines,Len: integer;
  i,j:integer;
  ss: array[1..MAX_POLE_HEIGHT] of string;
  comp: TComponent;
begin
  form1.opendialog1.DefaultExt:='txt';
  form1.opendialog1.Filter:='Text files (*.txt)|*.TXT';
  s:=extractfilepath(application.exename)+'FIGURES\';
  form1.opendialog1.InitialDir:= s;
  form1.opendialog1.Title:='Загрузите новую задачу';
  if not form1.opendialog1.Execute then exit;
```

```
  s:= form1.opendialog1.filename;
  //название задачи:
  NameFig:=s;
  {$i-}
  AssignFile(F,NameFig);
  Reset(F);
  {$i+}

  if IOResult<>0 then begin  //- ошибка при загрузке файла
    application.MessageBox ('Такой сетки нет!',NAME_PROG, MB_OK);
    exit
  end;

  nLines:=0;
  Len:=0;
  while not eof(f) do begin
    inc(nLines);
    //считать строку из файла:
    Readln(F, S);
    ss[nLines]:=s;
    If (Length(s) <> Len) and (Len<>0) Then begin
      application.MessageBox ('Неверная длина строки!',NAME_PROG,
MB_OK);
      exit
    end
    else
      Len:= Length(s);
  end;

  //закрыть файл:
  CloseFile(F);

  //проверить данные:
  if (Len> MAX_POLE_WIDTH) or (nLines> MAX_POLE_HEIGHT) then begin
    application.MessageBox('Неверные данные!',NAME_PROG, MB_OK);
    exit
  end;
  form1.caption:= NAME_PROG + '   [' + NameFig + ']';

  //вывести новые поля:
  Prepare(dgPole, Len+1, nLines+1);
  Prepare(dgPole2, Len+1, nLines+1);

  //очистить массивы полей:
  for j:= 1 to dgPole.RowCount-1 do
    for i:= 1 to dgPole.ColCount-1 do begin
      masPole[i,j]:= WHITE;
      masPole2[i,j]:= WHITE
    end;

  //заполнить массив новыми данными:
  for j:=1 to nLines do begin
```

```
    for i:=1 to Len do begin
      if ss[j,i]= '0' then
        masPole[i][j]:= 0
      else
        masPole[i][j]:= 1;
    end
  end;

  //запомнить начальную позицию:
  MemoryIn;
  dgPole.Invalidate;
  dgPole2.Invalidate;

  //выделить пункт в меню, соотв. размеру поля:
  comp:=FindComponent('mi'+inttostr(len)+'x'+inttostr(len));
 (comp as TMenuItem).Checked:= True;
end; //Open;
```

Пользуясь этими процедурами, вы можете создать *библиотеку заданий* на диске и в будущем использовать свои лучшие находки. Дальнейшее развитие этой темы вы найдёте в описании игры *Logos*.

Будет разумно сохранять на диске и *решения* задач, что и сделано в модуле формы *frmProtokol* (Рис. П5.39).

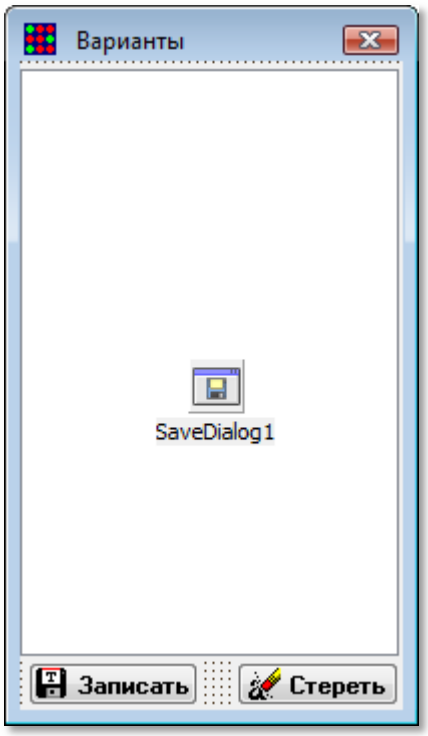


Рис. П5.39. Протокольная форма!

```
unit ProtokolUnit;
```

```
interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
Dialogs, StdCtrls, Buttons;

type
  TfrmProtokol = class(TForm)
    ListBox1: TListBox;
    sbtSaveOtv: TSpeedButton;
    SpeedButton1: TSpeedButton;
    SaveDialog1: TSaveDialog;
    procedure SpeedButton1Click(Sender: TObject);
    procedure sbtSaveOtvClick(Sender: TObject);

  privaterivate
    { Private declarations }

  public
    { Public declarations }
  end;

var
  frmProtokol: TfrmProtokol;

implementation

uses ShortGame;

{$R *.DFM}

//ОЧИСТИТЬ СПИСОК
procedure TfrmProtokol.SpeedButton1Click(Sender: TObject);
begin
  ListBox1.Items.Clear;
end;

//ЗАПИСАТЬ ОТВЕТ
procedure TfrmProtokol.sbtSaveOtvClick(Sender: TObject);
var
  s,ss,fn: string;
  i: integer;
  f: textfile;
begin
  savedialog1.DefaultExt:='txt';
  savedialog1.Filter:='Решения (*.txt)|*.TXT';
  savedialog1.FilterIndex:=1;
  s:=extractfilepath(application.exename)+'Figures\';
  savedialog1.InitialDir:= s;
  savedialog1.Title:='Запишите решение на диск';
```

```
  if NameFig<>'' then begin
    s:=extractFileName(NameFig);
    //listbox1.items.add (s);
    ss:='';
    i:=1;
    while (s[i]<>'.') and (i<=length(s)) do begin
      ss:=ss+s[i];
      inc(i)
    end;
    ss:=ss+'otv.txt';
    //listbox1.items.add (ss);
    savedialog1.filename:=ss
   end
   else
     savedialog1.filename:='otv_temp.txt';

   if not savedialog1.Execute then exit;

   //имя конечного файла:
   fn:= savedialog1.filename;
   assignfile(f,fn);
   rewrite(f);
   for i:=0 to (ListBox1.Items.Count - 1) do
     writeln (f,ListBox1.Items.Strings[i]);
   Closefile(f);
   Messagebeep(0)
 end;
end.
```

Теперь пройдёмся по остальным процедурам программы *ShortGame*.

При создании формы нужно увеличить размеры полей до 5 х 5, так как вокруг них добавилась оцифровка:

```
//СОЗДАТЬ ФОРМУ
procedure TForm1.FormCreate(Sender: TObject);
begin
  //подстраиваем величину формы под размеры сеток:
  AutoSize:= True;
  //установить размеры полей по умолчанию:
  Prepare(dgPole, 5, 5);
  Prepare(dgPole2, 5, 5);
  //начать новую игру:
  NewPlay;
end; //FormCreate
```

Основные изменения в процедуре *NewPlay* связаны с выводом названия задачи в заголовок формы и ограничением числа ходов в процедуре возврата ходов *Redo*. В глобальной *переменной*

```
  Hod_max: integer= 0;
```

будет храниться число ходов, сделанных при решении задачи, тогда процедура *Redo* не уйдёт «за границу».

В *ShortGame* даже на поле 4 х 4 клетки возможны *нерешаемые* задачи, поэтому я оставил прежний алгоритм для создания случайной позиции на полях. Впрочем, никакая другая позиция, кроме нулевой, на поле-образце в данной программе не используется и оно присутствует только для дальнейшего развития игры.

```
//ПОДГОТОВКА К НОВОЙ ИГРЕ
procedure TForm1.NewPlay;
var
 i, j: integer;
 x, y: integer;
begin
  //обнулить число ходов:
  Hod:= 0; Hod_max:= 0;
  lblHod.Caption:=  'Ход - 0';

  //задача не имеет названия:
  form1.caption:= NAME_PROG + '   []';

  //сделать случайные ходы:
  randomize;
  //очистить левое поле:
  for j:= 1 to dgPole.RowCount-1 do
    for i:= 1 to dgPole.ColCount-1 do
      masPole[i,j]:= WHITE;
  //сделать случайные ходы:
  for i:= 1 to 7 + random(dgPole.ColCount*dgPole.RowCount+1-7) do
begin
    x:= Random(dgPole.ColCount-1)+1; y:= Random(dgPole.RowCount-
1)+1;
    masPole:= DoMove(masPole, x, y);
  end;
  dgPole.Invalidate;

  //задать поле-образец:
  if Game= FullSalto then begin //- полный переворот
    //выставить на поле зелёные фишки:
    for j:= 1 to dgPole2.RowCount-1 do
      for i:= 1 to dgPole2.ColCount-1 do
        masPole2[i,j]:= WHITE
  end
  else begin //- переход к образцу
    //сделать случайные ходы:
    for i:= 1 to 7 + random(dgPole2.ColCount*dgPole2.RowCount+1-7)
do begin
```

```
      x:= Random(dgPole2.ColCount); y:= Random(dgPole2.RowCount);
      masPole2:= DoMove(masPole2, x, y);
    end;
  end;
  dgPole2.Invalidate;

  //запомнить начальную позицию:
  MemoryIn;
end; //NewPlay
```

В процедуре нажатия кнопки мыши нужно проверить, не нажата ли она на клетке с оцифровкой. В этом случае ход, разумеется, делать не следует.

```
//ПЕРЕВЕРНУТЬ ФИШКУ ИЛИ СДЕЛАТЬ ХОД
procedure TForm1.dgPoleMouseDown(Sender: TObject; Button: TMouse-
Button;
  Shift: TShiftState; X, Y: Integer);
var
  ACol,ARow: integer;
begin
  //координаты мыши:
  dgPole.MouseToCell(x,y,ACol,ARow);
  //нельзя делать ходы на клетки с оцифровкой:
  if ACol * ARow = 0 then exit;
  //нажата левая кнопка мыши и клавиша Shift -
  //инвертируем цвет клетки:
  if (ssLeft in shift) and (ssShift in shift ) then begin
    if masPole[ACol,ARow]= 1 then masPole[ACol,ARow]:= 0
    else masPole[ACol,ARow]:= 1;
    dgPole.Invalidate;
  end

  //нажата левая кнопка мыши без клавиши Shift -
  //делаем ход:
  else if ssLeft in shift then begin
    inc(Hod); lblHod.Caption:= 'Ход - ' + inttostr(Hod);
    masPole:=DoMove(masPole, ACol, ARow);
    dgPole.Invalidate;
    //запомнить ход:
    Moves[Hod].x:= ACol;
    Moves[Hod].y:= ARow;
    Hod_max:= Hod;
    //проверить, не решена ли задача:
    if IsReady then Ready;
  end;
end;  //dgPoleMouseDown
```

В процедуру *Prepare* добавьте только одну строку:

```
Hod_max:= 0;
```

Практически целиком придётся исправить процедуру выполнения хода, так как изменились правила игры:

```
//ВЫПОЛНИТЬ ХОД
function TForm1.DoMove(arr: TPole; x, y: integer): TPole;
var i: integer;
begin
  //инвертировать заданную клетку:
  arr[x,y]:= abs(arr[x,y]-1);
  //инвертировать ряды:
  for i:= Max(y-1,1) to Min(y+1,dgPole.RowCount-1) do //-
вертикальный
    arr[x,i]:= 1- arr[x,i];
  for i:= Max(x-1,1) to Min(x+1,dgPole.ColCount-1) do //-
горизонтальный
    arr[i,y]:= 1- arr[i,y];

  result:= arr;
end; //DoMove
```

В процедуре решения задачи мы должны учесть ряды с оцифровкой, весь остальной код изменять не нужно:

```
//НАЙТИ ПЕРЕХОД ОТ ОДНОЙ ПОЗИЦИИ К ДРУГОЙ
procedure TForm1.Solution(nStep: integer);

…

  //записать ходы в протокол:
  procedure SavePos;
  var
    n: integer;
    x, y: integer;
    s: string;
  begin
    frmProtokol.listbox1.items.add('');
    for n:= 1 to Hod do begin
      s:='';
      x:= Move[n] mod w; y:= Move[n] div w;
      frmProtokol.listbox1.items.add(inttostr(n) + '. ' + let-
ter[x]+ ' ' +inttostr(y));
    end;
    frmProtokol.listbox1.items.add('');
  end;

begin
…

//переходим к следующей клетке поля:
nextCell:
  inc(nCell[Hod]);
```

```
  while (nCell[Hod] mod w = 0) or (nCell[Hod] div w= 0) do
inc(nCell[Hod]);
…
 end;  //Solution
```

Для вывода оцифровки потребуется глобальная *константа*

```
letter= 'ABCDEFGHI';
```

Конечно, можно обойтись и без нее – в этой программе, а вот в других такой способ может и пригодиться.

Так как мы решили увеличить размеры полей до 9 х 9 клеток, то в меню нужно добавить два пункта - *mi8x8* (**Tag**=8, **ShortCut**=F8) и *mi9x9* (**Tag**=9, **ShortCut**=F9). Из-за оцифровки нам придётся подправить код в процедуре, обрабатывающей нажатие на пункты меню *mi4x4 - mi9x9*:

```
//ИЗМЕНИТЬ РАЗМЕРЫ ПОЛЕЙ
procedure TForm1.mi4x4Click(Sender: TObject);
…
begin
…
  //размеры поля:
  n:= (Sender as TMenuItem).Tag+1;
…
end;  //mi4x4Click
```

То же самое касается и следующей процедуры:

```
//ВОССТАНОВИТЬ ЗАПОМНЕННУЮ ПОЗИЦИЮ
procedure TForm1.miMemoryOutClick(Sender: TObject);
begin
…
    case w-1 of
      4: mi4x4.Checked:= True;
      5: mi5x5.Checked:= True;
      6: mi6x6.Checked:= True;
      7: mi7x7.Checked:= True;
      8: mi8x8.Checked:= True;
      9: mi9x9.Checked:= True;
    end;
…
end;  // miMemoryOutClick
```

В процедуру *miRedoClick* нужно добавить проверку на возможность возврата хода, о чём мы говорили ранее:

```
//ОТМЕНИТЬ ПОСЛЕДНИЙ ВОЗВРАТ ХОДА
procedure TForm1.miRedoClick(Sender: TObject);
```

```
begin
…
  if Hod< Hod_max then inc(hod) else exit;
…
end;  //miRedoClick
```

И наконец, я хочу напомнить, что пункт меню *miHint* оставлен для ваших собственных изысканий, поскольку в программе отсутствует процедура вывода подсказок.

Помогаем по-новому!

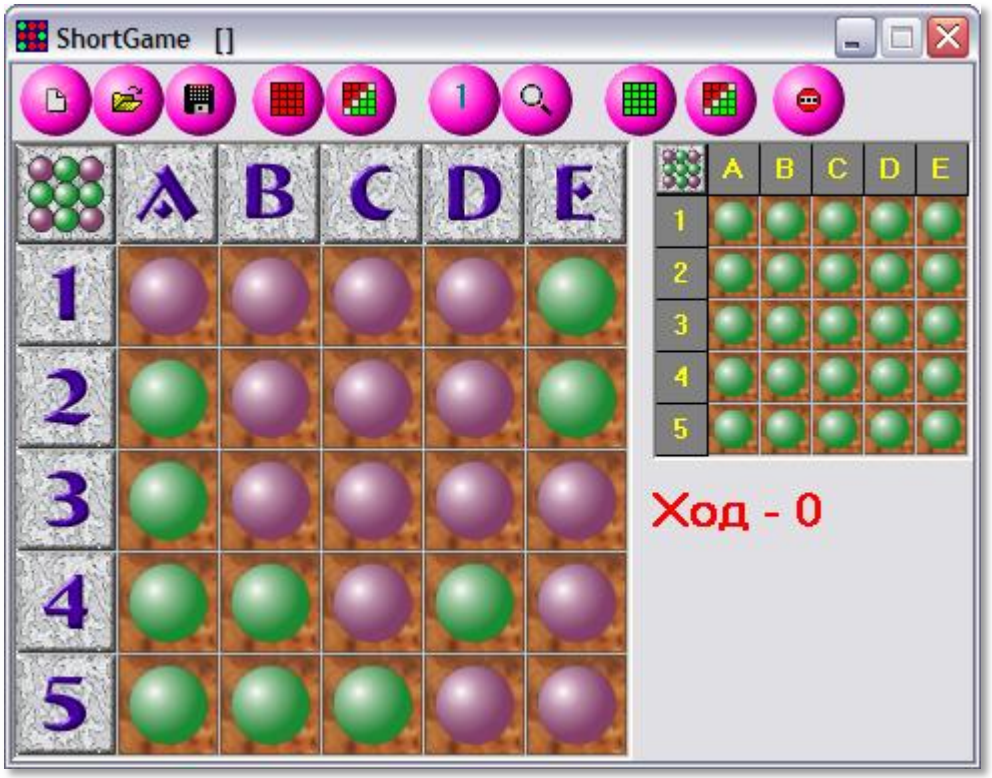
Давайте немного улучшим *помощь* для этой игры. Если файл справки достаточно длинный, то просматривать его будет не очень удобно. Поэтому лучше разбить его на отдельные разделы (темы), а на первой странице сделать небольшое *оглавление*, позволяющее быстро перейти в нужный раздел.

Прежде всего, подготовьте в текстовом редакторе файл справки (на этот раз мы воспользуемся редактором *Microsoft Word*, так как нам потребуются такие его возможности, которые могут отсутствовать в более простых программах). Примерный вид справки показан ниже:

ShortGame Help

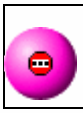
ShortGame - занимательная логическая игра-головоломка, в которой требуется все шарики на левом, игровом поле сделать зелёными (как это показано на правом, контрольном поле) за минимальное число ходов.

Ход состоит в изменении цвета любого шарика на игровом поле (зелёный шарик становится фиолетовым, фиолетовый – зелёным), но при этом изменяют цвет и все соседние по вертикали и горизонтали шарики. Чтобы выполнить ход, достаточно щёлкнуть мышкой по выбранному шарику.



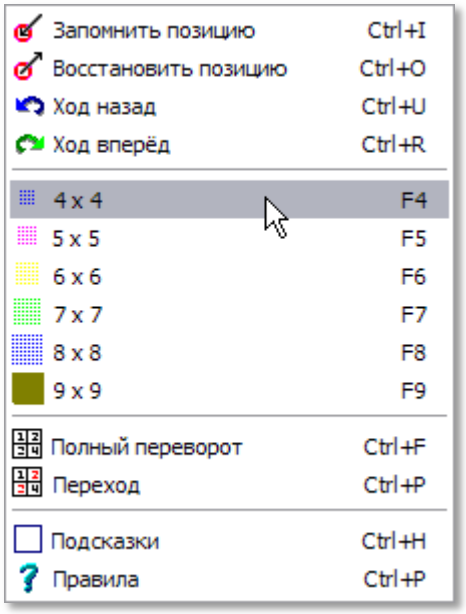
Управляющие кнопки:

	- начать новую игру, на игровое поле выводится новое задание,
	- загрузить задачу с диска,
	- сохранить позицию на диске,
	- сделать все шарики на игровом поле фиолетовыми,
	- изменить цвет всех шариков игрового поля на противоположный,
1	- установить глубину поиска решения задачи,
	-найти решение задачи,
	- сделать все шарики правого поля зелёными,
	- изменить цвет всех шариков правого поля на противоположный,

 ***- остановить поиск решения задачи.***

Всплывающее меню:

Вы также можете управлять программой с помощью всплывающего меню. Чтобы вызвать его, нажмите правую кнопку мыши.



Первая группа команд поможет вам самостоятельно найти короткое решение задачи. Вы можете запомнить любую позицию на поле, а затем восстановить её, а также вернуться назад на любое число ходов (или вернуть уже сделанные ходы при откате).
Вторая группа команд устанавливает размеры полей. Разумеется, чем больше размер поля, тем труднее будет решить задачу.
У игры имеется две разновидности – полный переворот, когда все шарики на игровом поле нужно сделать зелёными, и переход, когда следует «перекрасить» шарики на игровом поле так, как на правом поле-образце. В данной версии программы реализован только полный переворот.
Если включен режим подсказки, то программа будет отмечать те клетки игрового поля, на которые нужно ходить. В данной версии программы этот режим отсутствует.

Не забудьте выделить *заголовки* с помощью стилей, чтобы они составили оглавление справки! Также между всеми разделами необходимо вставить символ конца страницы (меню *Главная|Вставка|Разрыв страницы*).

Закончив работу с текстом справки, сохраните файл с расширением *RTF*.

Теперь вы можете откомпилировать его в какой-нибудь программе, предназначенной для создания справочных файлов. Например, в

программе *Help & Manual* вы сможете получить справку в различных форматах – *HLP, CHM, EXE, HTML, PDF*.

Пусть это будет *выполняемый* файл *ShortGame_Help.exe* в папке *Help*. Тогда процедура вызова справки будет такой:

```
//ПОКАЗАТЬ ФАЙЛ СПРАВКИ
procedure TForm1.miHelpClick(Sender: TObject);
var
  SEInfo: TShellExecuteInfo;
  ExitCode: DWORD;
  ExecuteFile: string;
Begin
  hide;
  ExecuteFile := ' Help\ShortGame_Help';
  FillChar(SEInfo, SizeOf(SEInfo), 0);
  SEInfo.cbSize := SizeOf(TShellExecuteInfo);
  with SEInfo do begin
    fMask := SEE_MASK_NOCLOSEPROCESS;
    Wnd := Application.Handle;
    lpFile := PChar(ExecuteFile);
    nShow := SW_SHOWNORMAL;
  end;
  if ShellExecuteEx(@SEInfo) then begin
    repeat
      Application.ProcessMessages;
      GetExitCodeProcess(SEInfo.hProcess, ExitCode);
    until (ExitCode <> STILL_ACTIVE) or Application.Terminated;
  end
  else ShowMessage('Error!');
  show;
end;  //miHelpClick
```

При выборе пункта меню *Правила* окно приложения исчезнет с экрана и появится текст справки. В левой части окна находится список разделов, соответствующий заголовкам в тексте справки. Щёлкнув на одной из строк, вы попадёте в нужное место справки (Рис. П5.40).

После того как вы закроете окно справки, на экране вновь появится окно приложения.



Рис. П5.40. Справка с оглавлением!



В операционной системе *Windows 7* справка работает неверно. Имейте это в виду!

Искусство побеждать

Казалось бы, правила игр *XorGame* и *ShortGame* настолько похожи, что и задачи должны решаться почти одинаково. Да не тут-то было! Ни одно из наших правил для игры *XorGame* здесь не действует, всё нужно начинать сначала!

Впрочем, стратегию игры найти нетрудно. Достаточно переворачивать *фиолетовые* фишки, начиная с верхнего ряда. Сделать это можно разными способами, и все они ведут к освобождению верхних рядов. Например, вы можете ходить последовательно в соседние справа или снизу клетки, но лучше подыскать более короткие комбинации уничтожения фиолетовых фишек.

Как бы то ни было, в итоге фишки останутся только в двух нижних рядах (легко избавиться и от предпоследнего ряда, но, если вы посмотрите на шаблон в начале раздела, то поймёте, что он занимает именно два ряда у нижней границы). Конечно, с равным успехом можно сгонять фишки к

верхней или боковой границе поля, но это ничего не изменит в решении задачи.

Оставшиеся в двух последних рядах фишки должны либо сразу составить нужную фигуру (*уголок* или букву *Т* вверх тормашками), либо легко уничтожаться, не «залезая» в верхние ряды. Для поля 4 х 4 клетки так всегда и получается, поэтому здесь вы не встретите большого сопротивления со стороны фиолетовых фишек, хотя, вероятно, вы придете к победе не самым коротким путём. А вот с другими полями всё не так просто! Произвольное уничтожение фиолетовых фишек обычно приводит к нерешаемым в пределах последних двух рядов расположениям фишек. Следовательно, переворачивать фишки в *предыдущих* рядах нужно так, чтобы получить *выигрышную* заключительную позицию. Если вам удастся найти правила выбора промежуточных ходов, то искусство побеждать станет наукой, а ваше имя навсегда останется в памяти благодарных игроков. Дерзайте!



Исходный код программы находится в папке **Short**.

Другие игры-оборотни

В игре *Flat Game* (версия 1.0, 2001 г.) (Рис. П5.41) Дмитрия Шумского вместе с «эпицентрической» клеткой изменяют цвет и все остальные клетки ряда – только вертикального или только горизонтального (а не обоих сразу, как в игре *Флип-Флоп*).

Поскольку щелчок на клетке поля не может однозначно идентифицировать ваши намерения, то щёлкать следует на *оцифровке* вокруг поля (так что она может пригодиться не только при записи ходов!). Других сложностей ни в реализации правил, ни в решении задач, как будто, нет.



Рис. П5.41. *Flat Game*

Гораздо более изощрённое правило перекрашивания клеток вы найдёте в игре *Камешки* (версия 1.7, 1999 г.) Здесь предлагается изменять цвет клеток по периметру квадрата 3 x 3 клетки (исключая центральную).

Задания выводятся последовательно из встроенной библиотеки, выбирать самостоятельно вы их не сможете. Другое неудобство игры – поле фиксированных размеров 7 x 5 клеток (Рис. П5.42).

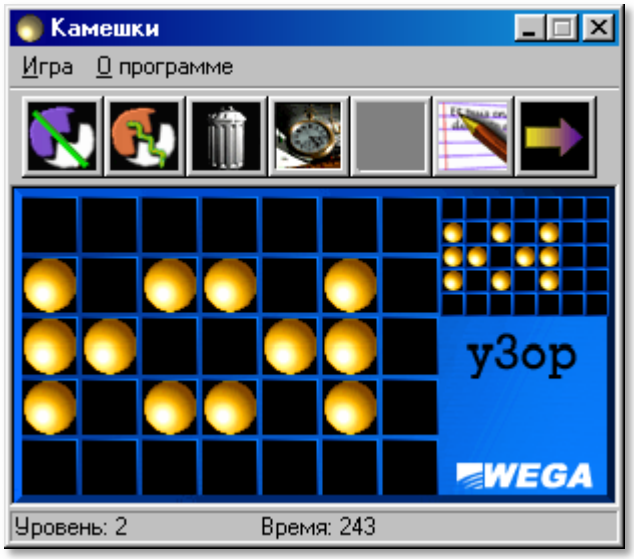


Рис. П5.42. *Камешки*

Взяв поле б***о***льших размеров и набор разных шаблонов переворачивания фишек, вы сможете получить гораздо более интересные игры.

Вот, пожалуй, и все сколько-нибудь заметные игры-оборотни (возможно, вам известны и другие). Как видите, сделано не много, так что программисты головоломок в большом долгу перед человечеством. Зато каждый из вас может отличиться на поприще создания новых игр!

Чёт и нечет

Числа правят миром.

Пифагорейское

Проверка на чётность – достаточно простой приём, который позволяет, однако, решать сложные задачи. Например, Евклид (или даже его предшественники из пифагорейской школы) с его помощью доказал иррациональность квадратного корня из двойки. Нам тоже довелось воспользоваться проверкой на чётность при программировании игры *XorGame*. Кроме того, она бывает полезна при нахождении решений перестановочных и топологических задач, а также многих головоломок.



«Проверкой на чётность» занимаются и девушки, гадающие на ромашке - «любит-не любит», и сатирик-юморист Михаил Задорнов, убеждённый, что женщинам следует дарить исключительно *нечётное* число цветов, и «менеджеры по продажам», уверенные, что покупатель с б***о***льшим удовольствием выложит 35 рублей, нежели 32.

Вообще, люди с удивительным упорством предпочитают одни числа другим, например *тройка* и *семёрка* явно предпочтительнее двойки или восьмёрки. Очень любимы числа, оканчивающие на 0 и 5 (вспомните юбилейные даты). Далее следуют числа, оканчивающиеся на 2, 3, 7, 6, 4, 9, 1. Причина тёплого отношения к «круглым» числам, конечно, заключается в десятичной системе счисления, а вот с остальными всё более загадочно. Впрочем, и природа в целом не одинаково относится к чётным и нечётным числам. Так, большинство животных (исключая иглокожих и головоногих) имеют *чётное* количество конечностей (следствие симметричности их тела), химические элементы с нечётными порядковыми номерами обычно более редки, чем соседние с ними чётные, а потому и более дороги. Теплота плавления углеводородов, начиная с гептана, имеющих чётное число атомов углерода в цепочке, заметно выше, чем у соседних с ними нечётных углеводородов. И эти примеры можно продолжать и продолжать…

Но вернёмся к нашим заботам. Для плавного перехода к теме рассмотрим шуточный *фокус* с тремя пустыми бокалами (годятся и стаканы, и другая посуда). Поставьте их на стол так, чтобы два бокала стояли ножками

(донышками) *вверх*, а между ними находился один бокал ножкой *вниз* (Рис. П5.43).



Рис. П5.43. Ножки - на ширине плеч

Необходимо за три «хода», переворачивая каждый раз *одновременно* два бокала, поставить все бокалы ножками вниз (Рис. П5.44).



Рис. П5.44. Правильное положение

Сделать это нетрудно.

1. Переворачиваем *первый* и *второй* бокалы (Рис. П5.45).



Рис. П5.45. Первый ход

2. Переворачиваем *первый* и *третий* бокалы (Рис. П5.46).



Рис. П5.46. Второй ход

3. Переворачиваем *первые два* бокала – задача решена (Рис. П5.47).



Рис. П5.47. Третий ход

Теперь незаметно переверните средний бокал ножкой вверх (Рис. П5.48)



Рис. П5.48. Хитрый трюк

и предложите кому-нибудь повторить фокус.

Если вы сравните последнюю картинку с первой, то легко заметите, что положение бокалов изменилось, и теперь головоломку невозможно решить за любое число ходов. И вот почему.

В первом случае в правильном положении (ножкой вниз) находится 1 бокал, а в конечном положении – 3. Оба эти числа *нечётные*. Во втором случае в правильном положении находятся 2 бокала – *чётное* число, а в конечном положении те же 3 бокала - *нечётное* число. Но при одновременном переворачивании двух бокалов чётность «системы» не изменяется.

Если мы перевернём из положения (Рис. П5.49)



Рис. П5.49. Новое исходное положение

два бокала ножками вверх (первый и третий), то в правильном положении будет находиться 0 бокалов – *чётное* число (будем считать, что это так). Если перевернуть другую пару бокалов, то ситуация вообще не изменится. Легко проверить, что и следующие ходы не нарушат чётности позиции, а это значит, что она никогда не станет нечётной и фокус повторить не удастся.

Если вы замените бокалы нашими фишками с двумя сторонами, то сразу поймёте, что этот фокус - всего лишь разновидность головоломок-оборотней, в которой все фишки вытянуты в одну линию.

Пусть в начальной позиции все бокалы стоят правильно – ножками вниз (Рис. П5.50)



Рис. П5.50. Правильное положение

а в конечной – все ножками вверх (Рис. П5.51).



Рис. П5.51. Конечное положение

Первая позиция *нечётная*, вторая – *чётная*, так что «фокус не удастся». Но попробуем решить *общую* задачу (см. журнал *Наука и жизнь*, №10 за 1982 год). Пусть имеется *m* бокалов ножками вниз, из которых за один раз можно перевернуть *n* любых бокалов. Требуется за минимальное число ходов получить позицию, в которой все бокалы находятся ножками вверх. Например, решим такую задачу: *m = 7, n = 5* (Рис. П5.52).



Рис. П5.52. Задачка!

Для удобства мы напишем небольшую программу для решения таких головоломок, ведь они могут быть интересны и вашим знакомым.

Ножки вверх!, или Головоломка с бокалами

Начните новый проект. Подготовьте картинки с бокалами в двух положениях и двух цветов. Как обычно, начните с подготовки *формы* (Рис. П5.53).

```
  Left = 14
  Top = 119 (расположите форму на экране так, чтобы были видны все
бокалы)
  Width = 770
  Height = 238 (размеры формы будут подгоняться под компоненты при
```

```
запуске программы, поэтому сильно не старайтесь)
  BorderIcons = [biSystemMenu]
  Caption = ' Бокалы'
  Color = clWhite
  Icon.Data – нарисуйте значок программы
  OnCreate = FormCreate
```

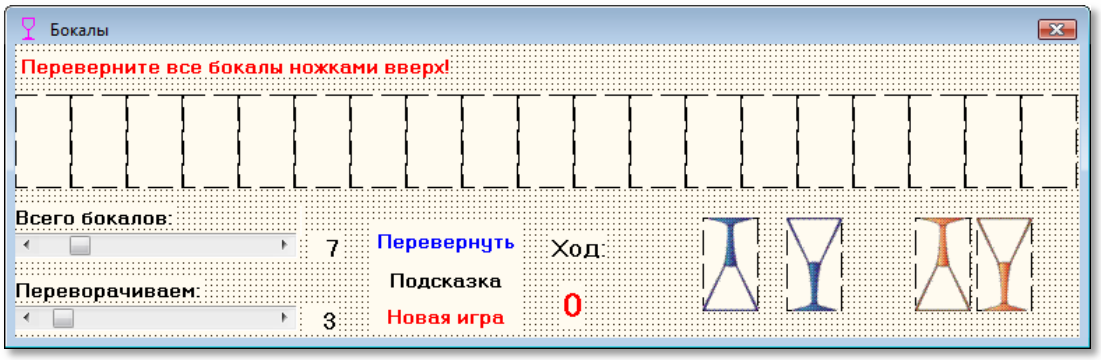
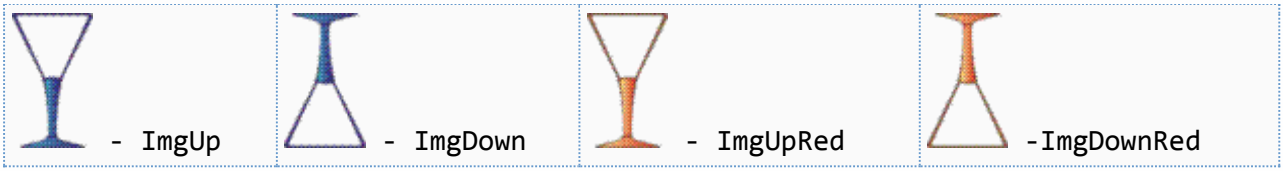


Рис. П5.53. Интерфейс приложения

Поместите картинки на хранение в компоненты *TImage*:




```
Width = 40
Height = 67 (размеры картинок)
AutoSize = True
Transparent = True
Visible = False
```

«Игровые» бокалы загрузите в компоненты *Image1.. Image19*:

```
Tag = 1, 2, 3, …
Left = 0, 40, 80, …
Top = 36
Width = 40
Height = 67
AutoSize = True
Transparent = True
OnClick = Image1Click (для всех компонентов).
```

Над бокалами установите метку *Label2* с напоминанием:

```
Left = 4
Top = 8
Width = 308
Height = 16
```

```
Caption = 'Переверните все бокалы ножками вверх!'
Font.Color = clRed    Font.Style = [fsBold]
```

Изменять количество бокалов на «столе» мы будем с помощью *скроллера sbBok* типа *TScrollBar* (вы можете найти и более изящное решение, если сочтёте нужным):

```
Left = 0
Top = 136
Width = 201
Height = 16
Max = 19 (максимальное число бокалов)
Min = 5    (минимальное число бокалов)
Position = 7 (значение по умолчанию)
OnChange = sbBokChange
```

Текущее значение числа бокалов можно увидеть в *метке lblBok*:

```
Left = 212
Top = 136
Width = 21
Height = 20
Alignment = taRightJustify
AutoSize = False
Caption = '7'
Font.Height = -16
Font.Style = [fsBold]
```

Аналогично мы будем задавать и число одновременно переворачиваемых бокалов *скроллером sbPerev*:

```
Left = 0
Top = 188
Width = 201
Height = 16
Max = 18
Min = 2
Position = 3
OnChange = sbPerevChange
```

и показывать в *метке lblPerev*:

```
Left = 208
Top = 188
Width = 23
Height = 20
Alignment = taRightJustify
AutoSize = False
Caption = '3'
Font.Height = -16
Font.Style = [fsBold]
```

Общее число бокалов мы будем хранить в глобальной *переменной:*

```
  //всего бокалов:
  nBok: integer;
```

А число одновременно переворачиваемых бокалов – *в переменной*:

```
  //переворачиваем за один раз:
  nPerev: integer;
```

При перемещении ползунка скроллера *sbPerev* в метке *lblPerev* выводится новое значение числа переворачиваемых бокалов, а также оно запоминается в переменной *nPerev*. Так как число переворачиваемых бокалов должно быть *меньше* общего числа бокалов, то значение *nPerev* устанавливается в допустимых пределах:

```
//ИЗМЕНИТЬ ЧИСЛО ПЕРЕВОРАЧИВАЕМЫХ БОКАЛОВ
procedure TForm1.sbPerevChange(Sender: TObject);
begin
  lblPerev.Caption:= inttostr(sbPerev.Position);
  nPerev:= sbPerev.Position;
  while nPerev >= nBok do sbPerev.Position:= sbPerev.Position-1;
  NewPlay;
end;  //sbPerevChange
```

Процедура обработки изменения позиции ползунка второго скроллера ещё проще:

```
//ИЗМЕНИТЬ ЧИСЛО БОКАЛОВ
procedure TForm1.sbBokChange(Sender: TObject);
begin
  lblBok.Caption:= inttostr(sbBok.Position);
  nBok:= sbBok.Position;
  //скорректировать число переворачиваемых бокалов:
  sbPerevChange(Self);
end; //sbBokChange
```

После любого изменения условия задачи следует начинать *новую* игру, так как все предыдущие ходы в этом случае теряют всякий смысл. В процедуре подготовки к новой игре снимаются значения со скроллеров и обнуляется число ходов:

```
//НОВАЯ ИГРА
procedure TForm1.NewPlay;
begin
   //число бокалов:
  lblBok.Caption:= inttostr(sbBok.Position);
  nBok:= sbBok.Position;
  ChangeBok;
```

```
  //переворачиваем за 1 раз:
  lblPerev.Caption:= inttostr(sbPerev.Position);
  nPerev:= sbPerev.Position;
  //обнулить ходы:
  Hod:= 0;
  lblHod.Caption:= '0';
end; //NewPlay
```

Бокалы могут находиться в одном из следующих *положений*:

Up – «горлышком» вверх,
Down – «горлышком» вниз,
UpRed – «горлышком» вверх, выделенное состояние,
DownRed – «горлышком» вниз, выделенное состояние,
None – бокал с заданным номером отсутствует.

Для описания положения бокалов введём новый *тип*

```
type stBok= (Up, Down, UpRed, DownRed, None);
```

и объявим глобальный *массив*

```
  //массив бокалов:
  masBok: array[1..19] of stBok;
```

В начале игры первые *nBok* бокалов находятся в правильном положении *Up*, а все остальные бокалы от *nBok+1* до *19* вообще в игре участия не принимают (*None*):

```
//ПЕРЕВЕСТИ БОКАЛЫ В ИСХОДНОЕ ПОЛОЖЕНИЕ
procedure TForm1.ChangeBok;
var
  i: integer;
begin
  for i:= 1 to 19 do begin
    if i <= nBok then
      masBok[i]:= Up
    else
      masBok[i]:= None;
  end;
  DrawBok;
end; //ChangeBok
```

Сначала все изменения на «столе» заносятся в массив *masBok*, а затем отображаются на экране в процедуре *DrawBok*. В ней последовательно перебираются все (в том числе и невидимые) компоненты *TImage* с бокалами и, соответственно значению в массиве *masBok*, в каждый компонент выводится нужная картинка:

```
//ПОКАЗАТЬ БОКАЛЫ
procedure TForm1.DrawBok;
var
  i: integer;
  comp: TComponent;
begin
  for i:= 1 to 19 do begin
    comp:= FindComponent('Image'+inttostr(i));
    (comp as Timage).top:= 36;
    case masBok[i] of
      None: (comp as Timage).visible:= False;
      Up: begin
            (comp as Timage).visible:= True;
            (comp as Timage).Picture.Assign(ImgUp.Picture);
          end;
      Down: begin
            (comp as Timage).visible:= True;
            (comp as Timage).Picture.Assign(ImgDown.Picture);
          end;
      UpRed: begin
            (comp as Timage).visible:= True;
            (comp as Timage).Picture.Assign(ImgUpRed.Picture);
          end;
      DownRed: begin
            (comp as Timage).visible:= True;
            (comp as Timage).Picture.Assign(ImgDownRed.Picture);
          end;
    end;
  end;
end; //DrawBok
```

Число выполненных ходов хранится в глобальной *переменной*

```
  //ходы:
  Hod: integer=0;
```

А для вывода числа ходов нам потребуются *метки Label1*:

```
    Left = 384
    Top = 136
    Width = 40
    Height = 20
    Caption = 'Ход:'
    Font.Height = -16
    Font.Style = [fsBold]
```

и *lblHod:*

```
    Left = 392
    Top = 172
    Width = 15
```

```
    Height = 29
    Alignment = taCenter
    Caption = '0'
    Font.Color = clRed
    Font.Height = -24
    Font.Style = [fsBold]
```

При запуске программы устанавливается в *True* свойство формы *AutoSize*, чтобы её размеры при любом числе бокалов были оптимальными, а затем выводится *начальная* позиция, определяемая положением движков скроллеров:

```
//СОЗДАТЬ ФОРМУ
procedure TForm1.FormCreate(Sender: TObject);
begin
  form1.AutoSize:= True;
  NewPlay;
end; //FormCreate
```

Поскольку вы можете переворачивать одновременно *любые* бокалы (но не любое их количество!), то их нужно уметь *выделять*. Для этого удобно просто щёлкать на нужных бокалах - их цвет будет меняться с *синего* (положение *Up* и *Down*) на *красный* (*UpRed* и *DownRed*), и наоборот:

```
//ВЫДЕЛИТЬ БОКАЛ
procedure TForm1.Image1Click(Sender: TObject);
var i: integer;
begin
  //номер бокала:
  i:= (Sender as TImage).tag;
  case masBok[i] of
    Up: masBok[i] := UpRed;
    Down: masBok[i] := DownRed;
    UpRed: masBok[i] := Up;
    DownRed: masBok[i] := Down
  end;
  DrawBok;
end; //Image1Click
```

Не забудьте обработчики события *OnClick* всех компонентов *TImage* связать с этой процедурой!

Сам ход выполняется при нажатии на *кнопку sbtPerev* типа *TSpeedButton*:

```
    Left = 256
    Top = 128
    Width = 105
    Height = 26
    Cursor = crHandPoint
    Caption = 'Перевернуть'
```

```
    Flat = True
    Font.Color = clBlue
    Font.Style = [fsBold]
    OnClick = sbtPerevClick
```

```
//ПЕРЕВЕРНУТЬ БОКАЛЫ
procedure TForm1.sbtPerevClick(Sender: TObject);
var i, n: integer;
begin
  //проверить ход:
  n:= 0;
  for i:= 1 to 19 do
    if (masBok[i]= UpRed) or (masBok[i]= DownRed)
      then inc(n);
  if n <> nPerev then begin
    application.MessageBox('Столько бокалов вам не перевернуть!',
NAME_PROG,IDOK);
    exit
  end;
  //выполнить ход:
  for i:= 1 to 19 do
  case masBok[i] of
    UpRed: masBok[i] := Down;
    DownRed: masBok[i] := Up
  end;
  DrawBok;
  inc(Hod);
  lblHod.Caption:= inttostr(Hod);

  //все бокалы перевёрнуты?
  n:= 0;
  for i:= 1 to 19 do
    if masBok[i]= Down
      then inc(n);
  if n = nBok then begin
    application.MessageBox('Отличная работа!', NAME_PROG,IDOK);
    NewPlay;
  end;
end; //sbtPerevClick
```

Сначала нужно проконтролировать правильность выполнения хода – число выделенных бокалов должно в точности равняться *nPerev*. Если всё верно, выделенные бокалы переворачиваются, и метод проверяет, не решена ли задача.

В любой момент вы можете начать *новую* игру, нажав *кнопку sbtNewPlay*:

```
    Left = 256
    Top = 184
    Width = 105
    Height = 22
```

```
    Cursor = crHandPoint
    Caption = 'Новая игра'
    Flat = True
    Font.Color = clRed
    Font.Style = [fsBold]
    OnClick = sbtNewPlayClick
```

```
//НАЧАТЬ НОВУЮ ИГРУ
procedure TForm1.sbtNewPlayClick(Sender: TObject);
begin
  NewPlay;
end; //sbtNewPlayClick
```

Как решить задачу?

Давайте вспомним о нашей задаче с семью бокалами, которые нужно переворачивать по три штуки за один раз.

Если вы станете бессистемно вертеть бокалы, то рано или поздно вы, конечно, эту задачу решите (не такая уж она и сложная), да что толку – со следующей вы будете мучиться ничуть не меньше. Лучше сразу всё хорошенько обдумать, а затем играть «со смыслом»!

Прежде всего, вы должны для себя отметить, что, сколько бы раз вы бокалы ни переворачивали, всё равно общее число перевёрнутых бокалов будет кратно трём (мы ведь переворачиваем по три бокала!). Но семь на три не делится, поэтому просто так их и не перевернуть. Легко за два раза перевернуть 6 бокалов, но тогда останется только один «неправильный» бокал, и вместе с ним нам придётся перевернуть и два правильных. Они станут неправильными и, возвращая их в нужное положение, мы опять прихватим один правильный бокал. В общем, играть нужно не так!

Первый ход очевиден – мы должны перевернуть ножками вверх любые три бокала (Рис. П5.54).



Рис. П5.54. Первый ход

Столь же очевиден и *последний* ход – нам придётся перевернуть ножками вверх какие-то три бокала (Рис. П5.55).



Рис. П5.55. Последний ход

Все сложности - в *промежуточных* ходах, точнее, даже в одном ходе. Попробуем его вычислить.

Ясно, что за два хода задачу не решить, так как мы сумеем перевернуть только 6 бокалов. За три хода мы перевернём 9 бокалов, а нужно 7. Но 7 – это те бокалы, которые стояли вначале ножками *вниз*, а ведь после первого хода появляются и бокалы, стоящие ножками *вверх*. Если один из них вернуть в исходное положение (ножкой вниз), то по ходу решения мы перевернём уже 8 бокалов. Но мы обязаны перевернуть его ещё один раз, иначе он так и будет стоять ножкой вниз! Таким образом, при этих операциях мы добавили к семи переворачиваемым бокалам ещё два. Итого получилось ровно 9, что и требуется!

Теперь всё предельно ясно: делая *второй* ход, мы должны возвратить в исходное положение один из тех бокалов, что был перевёрнут на первом ходу (Рис. П5.56).



Рис. П5.56. Второй ход

В расчётах мы не ошиблись – осталось ровно *три* бокала ножками *вниз*, которые и нужно перевернуть на *третьем* ходу.

Решим более сложную задачу. Пусть теперь имеется 17 бокалов, а переворачивать нужно по 5 бокалов одновременно. В лучшем случае задачу можно решить за 4 хода, перевернув 20 бокалов. То есть к 17 уже имеющимся следует добавить ещё три. А это невозможно: мы показали, что дополнительные бокалы берутся из уже перевёрнутых ножками вверх, при этом любой из них придётся переворачивать два раза. Но три на два не делится, поэтому никакие ухищрения не помогут. Придётся накинуть ещё один ход, тогда мы перевернём 25 бокалов, из которых 25 – 17 = 8

«лишних». Их могут дать 8 : 2 = 4 возвращаемых в исходное положение бокалов.

Итак, для решение задачи потребуется 5 ходов. На первом и последнем мы будем переворачивать по 5 бокалов, стоящих ножками вниз. А вот во время оставшихся трёх ходов нам и следует возвращать в исходное положение 4 бокала. Причём совершенно безразлично, как вы разделите эти бокалы между ходами – 4+0+0, 3+1+0, 2+2+0, … Стало быть, задача имеет *несколько* вариантов решения длиной в 5 ходов.

Нам осталось изложить открытый нами *алгоритм* на «дельфийском» языке и препоручить *кнопке sbtHint*:

```
Left = 256
Top = 156
Width = 105
Height = 25
Caption = 'Подсказка'
Flat = True
Font.Style = [fsBold]
OnClick = sbtHintClick
```

всю заботу о выводе подсказки:

```
//ПОДСКАЗКА
procedure TForm1.sbtHintClick(Sender: TObject);
begin
  Hint;
end;

//ВЫВЕСТИ ПОДСКАЗКУ
procedure TForm1.Hint;
var
  i, n, m: integer;
begin
  //если число бокалов нечётное, а переворачиваем чётное
  //число бокалов, то задача неразрешима:
  if odd(nBok) and not odd(nPerev) then
  begin
    showmessage('Задача решений не имеет!');
    exit
  end;

  m:= nBok;
  n:= nPerev;
  i:= 0;
  while (n*i< m) or ((n*i - m) mod 2 <> 0) do inc (i);

  showmessage('На решение задачи потребуется ходов - '+ inttostr
(i)+ #10#13+              'Число возвращаемых бокалов = '+
```

```
inttostr ((n * i - m) div 2));
end;  //Hint
```

Вычисления проводятся в цикле

```
while (n*i< m) or ((n*i - m) mod 2 <> 0) do inc (i);
```

Он совершается до тех пор, пока число, кратное числу переворачиваемых бокалов (*n*i*), меньше общего числа бокалов или разность (*n*i - m*) не делится на два. О необходимости (и достаточности) выполнения этих условий мы только что говорили.

Теперь вы сможете не только успешно переворачивать бокалы, но и делать это непринуждённо!

Конечно, вам уже кажется, что нет таких задач, с которыми вы бы не справились. Тогда попробуйте такую.

В ряд расставлены 6 бокалов, причём первые три с вином, последние три пустые (Рис. П5.57). Вы можете взять в руку только *один* бокал. Что нужно сделать, чтобы бокалы с вином и пустые чередовались?



Рис. П5.57. Невинная винная задача

Ответ вы найдете на следующей странице.



1. Можно ли решить «бокальную» задачу при условии, что допускается одновременно переворачивать только идущие **подряд** бокалы? Тривиальные сочетания **m** и **n** типа 6 и 3 не учитывайте.

2. Условие то же, что и в предыдущем пункте, но бокалы располагаются кольцом, то есть первый и последний бокалы стоят *рядом*.

3. *Вероятно,* задача неразрешима при условиях 1 и 2 (попробуйте доказать или опровергнуть это предположение!), поэтому поищите задачи, в которых начальное положение бокалов отличается от стандартного (все ножками вниз). Очевидно, что существует множество таких задач (их легко получить из конечной позиции, делая

ходы тем или иным способом). Вопрос в том, могут ли при этом возникнуть *интересные* расположения бокалов или *необычные* последовательности ходов.

Ответ неожиданно прост (быстро ли вы до него додумались?) – нужно перелить вино из *второго* бокала в *пятый* и вернуть его на место. Оп-ля!



Исходный код программы находится в папке **Bokaly**.

Занятие 6. Живописная основа, или класс TCanvas

Мостовая пусть качнется, как очнется!
Пусть начнется, что еще не началось!
Вы рисуйте, вы рисуйте,
вам зачтется...
Что гадать нам:
удалось — не удалось?

Булат Окуджава. *Живописцы*

Канва (**canvas**) выполняет в *Delphi* ту же роль, что и холст в настоящей живописи. Именно на ней мы до сих пор рисовали простые и замысловатые фигуры и картинки, ведь канва – это и есть тот самый маловразумительный **контекст устройства** в *Windows* (**DC, device context**), без которого не могут обойтись графические функции *API*. Если вы будете рисовать только средствами *Delphi*, то можете забыть об этих премудростях *Windows*. Если же вы склонны к «рискованному» программированию, то вам поможет в этом благородном деле дескриптор контекста устройства канвы, который задаётся её свойством *property Handle: HDC*. Как и всякий контекст устройства, канва – это не только поверхность для рисования, но и сами «художественные» инструменты: перо, кисть и шрифты.

Пуантилизм, или Ставим точки

Самым простым геометрическим объектом является **точка**. Все поверхности, и канва в том числе, состоят из точек. Вот только геометрическая поверхность – это бесконечное множество точек, а канва – всегда ограниченное, так как каждая точка должна отображаться на экране дисплея, а он «не резиновый». И на экране, и на канве точка называется **пикселем.** Канву можно представить в виде двухмерного массива пикселей, каждый из которых характеризуется координатами (индексами массива) и цветом. Чтобы узнать, какой цвет имеет тот или иной пиксель на канве, достаточно обратиться к свойству канвы *property Pixels[X, Y: Integer]: TColor* (если заданная точка выходит за границы канвы, то возвращаемое значение равно -1). Аналогично можно окрасить любой пиксель в нужный цвет, присвоив этому свойству соответствующее значение.

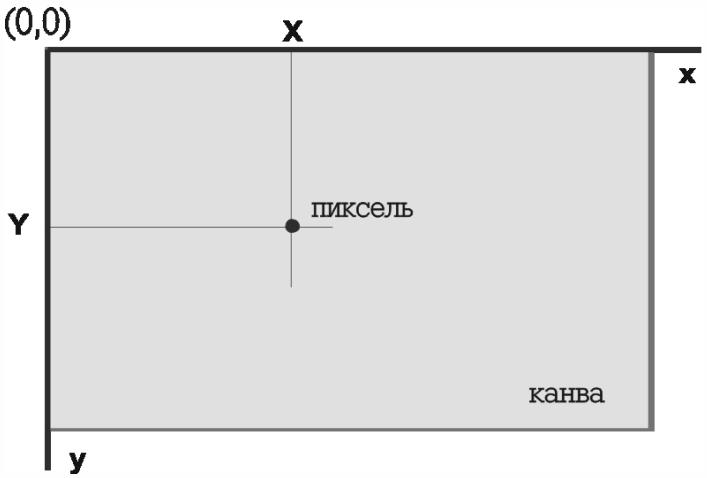


Рис. П6.1. Компьютерная система координат



Для тех, кто учился в школе, расположение координатных осей на канве покажется странным: ось **Y** (ордината) направлена вниз, а не вверх, поэтому чем *больше* значение **Y**, тем *ниже* будет точка на экране (Рис. П6.1). Если вы планируете задавать координаты точек в программе, то следует учитывать эту «тонкость».

Конечно, рисовать точками лучше в каком-нибудь графическом редакторе (*Microsoft Paint* или *Image Editor*, например, вполне годятся для подготовки значков *ICO* или картинок *BMP*), чем в программе, но – все фигуры построены из точек, поэтому напишем простенькую программу, которая будет самостоятельно выводить на канву формы приложения точки случайно выбранного цвета (Рис. П6.2). Эстетического удовольствия никакого, но зато мы хорошенько познакомимся со свойством **Pixels** (иногда бывает полезно вовремя поставить точку, к примеру, это может пригодиться при заливке области методом **FloodFill**).

Запустите *Delphi*, установите на форме **таймер** (он стоит первым на вкладке *System* в *Палитре компонентов*), щёлкните на нём и заполните процедуру **Timer1Timer**. Таймер нужен нам для автоматического обновления картинки. Вот весь текст программы:

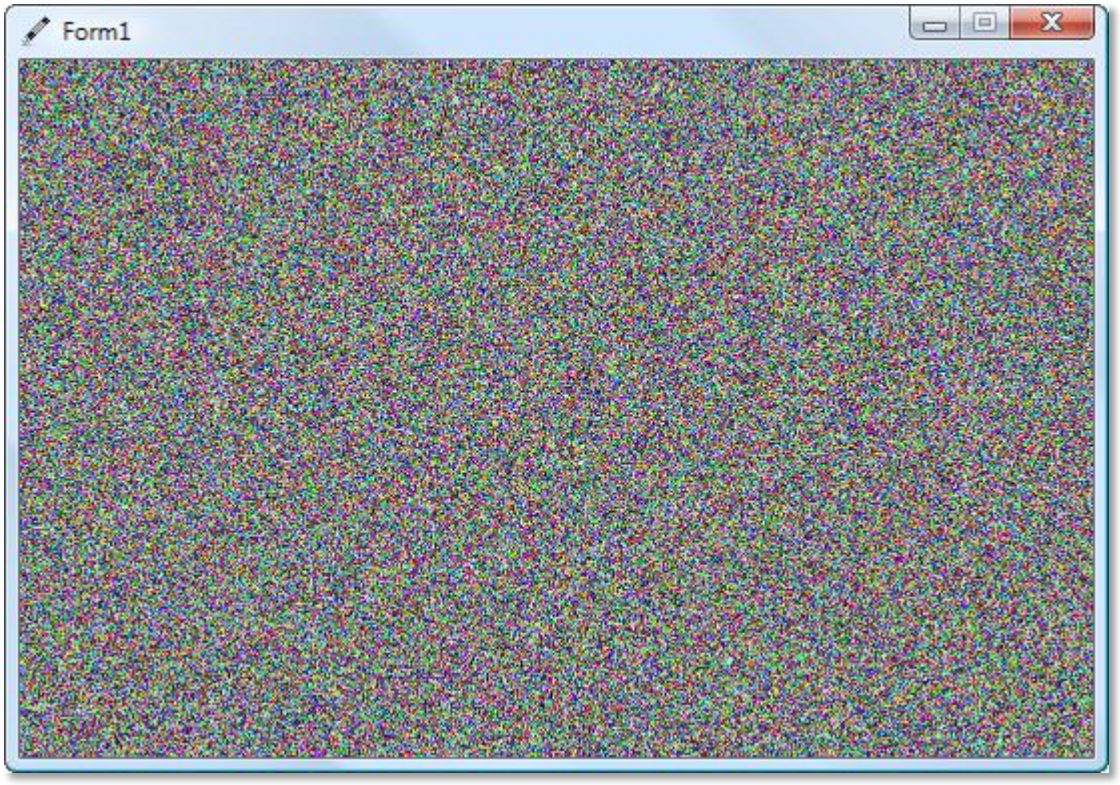


Рис. П6.2. Пикселизация формы прошла успешно!

```
unit pixel0;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
Dialogs,
  ExtCtrls;

type
  TForm1 = class(TForm)
    Timer1: TTimer;
    procedure FormDestroy(Sender: TObject);
    procedure Timer1Timer(Sender: TObject);
  private
    { Private declarations }
    procedure PaintPixel;
  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation
{$R *.DFM}
```

```
//ЗАПОЛНЯЕМ ФОРМУ РАЗНОЦВЕТНЫМИ ТОЧКАМИ
procedure TForm1.PaintPixel;
Var
  x, y:integer;
  clr: TColor;
begin
  for y:=0 to height-1 do
    for x:=0 to width-1 do
    begin
      clr:= rgb (random(256),random(256),random(256));
      Canvas.Pixels[x,y]:= clr;
   end;
end;

//ПЕРЕРИСОВЫВАЕМ КАРТИНКУ В КАЖДОМ ЦИКЛЕ ТАЙМЕРА
procedure TForm1.Timer1Timer(Sender: TObject);
begin
  PaintPixel;
  Timer1.Interval:= 2000
end;

//УНИЧТОЖИТЬ ФОРМУ
procedure TForm1.FormDestroy(Sender: TObject);
begin
  timer1.Enabled:= false;
end;

end.
```



Исходный код программы находится в папке **pixel0**.

Запустите программу и убедитесь, что мы сумели раскрасить все пиксели канвы. Также вы, наверное, обратили внимание, что этот способ задания цвета пикселям очень медленный, поэтому пустимся на хитрость: создадим канву в памяти, там её разрисуем, а уже потом быстренько выведем на экран (то есть на канву формы).

В разделе *implementation* введём переменную **bmp** типа *TBitMap*, в которой канва присутствует в качестве свойства (впрочем, это верно и для других компонентов, так как сама по себе канва существует только в абстрактном виде):

```
var
  bmp: TBitMap;
```

Дважды щёлкните на форме и заполните раскрывшуюся заготовку процедуры кодом:

```
//СОЗДАТЬ ФОРМУ
procedure TForm1.FormCreate(Sender: TObject);
begin
   bmp:=TBitMap.Create;        //создать в памяти битовую карту
   bmp.pixelformat:=pf24bit;   //глубина цвета - 24 бита
   bmp.width:=Clientwidth;      //размер битовой карты - по размеру
клиентской области формы
   bmp.height:=Clientheight;
end;
```

В результате будет создана не только форма, но и битовая карта (в памяти), на канве которой мы и будем рисовать.

В процедуре **PaintPixel;** замените строку `Canvas.Pixels[x,y]:= clr;` строкой `bmp.Canvas.Pixels[x,y]:= clr;` - теперь точки будут окрашиваться на канве битовой карты в памяти, что делается значительно быстрее (поэтому в процедуре **Timer1Timer** удалите строчку `Timer1.Interval:= 2000;` – изображение будет обновляться каждую секунду). После того как канва битовой карты будет заполнена точками, перерисовываем её на канву формы методом *canvas.draw(0,0,bmp);* (добавьте эту строчку в самый конец процедуры **PaintPixel**). Осталось только в процедуре **FormDestroy** очистить память от вспомогательной битовой карты:

```
bmp.free; //удалить битовую карту
```

Запустите приложение – после модернизации программы картинка появляется на экране очень быстро, мы уже не видим, как она прорисовывается. Этот приём увеличения скорости рисования применяется очень часто, поэтому на него следует «положить» глаз.



Исходный код программы находится в папке **pixel01**.

Если же для вас и это медленно, то стоит подружиться со свойством

```
property ScanLine[Row: Integer]: Pointer;
```

того же компонента **TBitmap**, который открывает доступ к полным строкам пикселей. В параметре **Row** мы задаём номер нужной строки, а в ответ получаем указатель на строку с данными в битовой карте. Для работы со **ScanLine** нам потребуется переменная **P** - указатель на массив битов. Так

как мы выбрали формат пикселей `bmp.pixelformat:=pf24bit`, то цвет каждой точки описывается 8-ю битами (по одному байту на каждую составляющую цвета). Поэтому на точку приходится 3 байта в массиве **pByteArray**, в каждый из которых мы записываем случайное значение интенсивности цвета. В целом процедура **PaintPixel** теперь должна быть такой:

```
//ЗАПОЛНЯЕМ ФОРМУ РАЗНОЦВЕТНЫМИ ТОЧКАМИ
procedure TForm1.PaintPixel;
Var
  x, y:integer;
  P : pByteArray;
begin
  //перебираем все строки битовой карты:
  for y:=0 to bmp.height-1 do
  begin
    //получаем уазатель на очередную строку:
    p:=bmp.scanline[y];
    for x:=0 to bmp.width-1 do
    begin
      p[x*3]:= random(256);
      p[x*3+1]:= random(256);
      p[x*3+2]:= random(256);
    end;
  end;
  canvas.draw(0,0,bmp);
  timer1.Interval:=30
end;
```

Промежуток между сменой картинок теперь можно сильно уменьшить – до 30 миллисекунд, больше ничего в программе изменять не нужно. Запустите программу – и точки начнут мелькать на экране до чёртиков в глазах. Вероятно, вам хотелось бы ещё понаслаждаться свойством **ScanLine**, но она из другой «оперы» (это свойство *TBitmap*, а не *TCanvas*), поэтому придётся с ней расстаться.



Исходный код программы находится в папке **pixel**.

Постараемся чуточку облагородить картину на экране, придав ей известную долю симметричности. Сделать это просто: будем выводить точки в левом верхнем квадранте канвы, а ещё три точки расставим симметрично в остальных трёх квадрантах. Вот только размеры пикселей маловаты – картинка всё равно получится весьма хаотичной. И дабы труды наши в этот раз не пропали даром, сделаем «пипетку», с помощью которой можно определять цвет любого пикселя. Подобный «медицинский» инструмент присутствует во многих графических редакторах, а у нас пипеткой порабо-

тает по совместительству мышиный курсор: мы будем «брать» цвет той точки, на которую он указывает. В этот цвет будим красить метку **lblColor,** так как на экране цвет пикселя разглядеть очень трудно. Одновременно в двух других метках будем выводить числовые значения цвета - в 16-ричном виде и как **R, G, B**-составляющие. Установите у правой границы формы 3 компонента **TLabel** (вкладка **Standard** в *Палитре компонентов*) шириной 125 пикселей. Процедуру вывода точек придётся, конечно, изменить:

```
//ЗАПОЛНЯЕМ ФОРМУ РАЗНОЦВЕТНЫМИ ТОЧКАМИ
procedure TForm1.PaintPixel;
Var
  x, y, hc, wc:integer;
  clr: TColor;
begin
  hc:= (bmp.height-1) div 2;
  wc:= (bmp.width-1)  div 2;
  for y:=0 to hc do
    for x:=0 to wc do
      begin
        clr:= rgb (random(256),random(256),random(256));
        bmp.Canvas.Pixels[wc+x,hc+y]:= clr;
        bmp.Canvas.Pixels[wc-x,hc+y]:= clr;
        bmp.Canvas.Pixels[wc+x,hc-y]:= clr;
        bmp.Canvas.Pixels[wc-x,hc-y]:= clr;
      end;
  canvas.draw(0,0,bmp);
end;
```

Таймер выбросим (иначе картина на экране сменится раньше, чем мы протестируем точку), а процедуру **Timer1Timer** заменим другой, чтобы картинка не исчезала при манипуляциях с формой:

```
procedure TForm1.FormPaint(Sender: TObject);
begin
  PaintPixel
end;
```

Заменять одну картинку другой мы будем сами, нажимая кнопку мышки:

```
//МЕНЯЕМ КАРТИНКУ
procedure TForm1.FormMouseDown(Sender: TObject; Button: TMouseBut-
ton;
  Shift: TShiftState; X, Y: Integer);
begin
  PaintPixel
end;
```

Осталось добавить к нашему проекту *пипетку*:

```
//ПИПЕТКА - ОПРЕДЕЛЯЕМ ЦВЕТ ПОД КУРСОРОМ
procedure      TForm1.FormMouseMove(Sender:      TObject;      Shift:
TShiftState; X,
  Y: Integer);
var
  clr: TColor;
begin
  clr:= bmp.Canvas.Pixels[x,y];
  label1.Caption:= colortostring(clr);
  label2.Caption:= ('R= ' + inttostr(GetRValue(clr)) + ' ' +
                    'G= ' + inttostr(GetGValue(clr)) + ' ' +
                    'B= ' + inttostr(GetBValue(clr)));
  lblColor.Color:= clr;
end;
```

И, раз уж мы битый час говорим о картинках, пора показать товар лицом. Вот так всё это выглядит в натуре (Рис. П6.3).

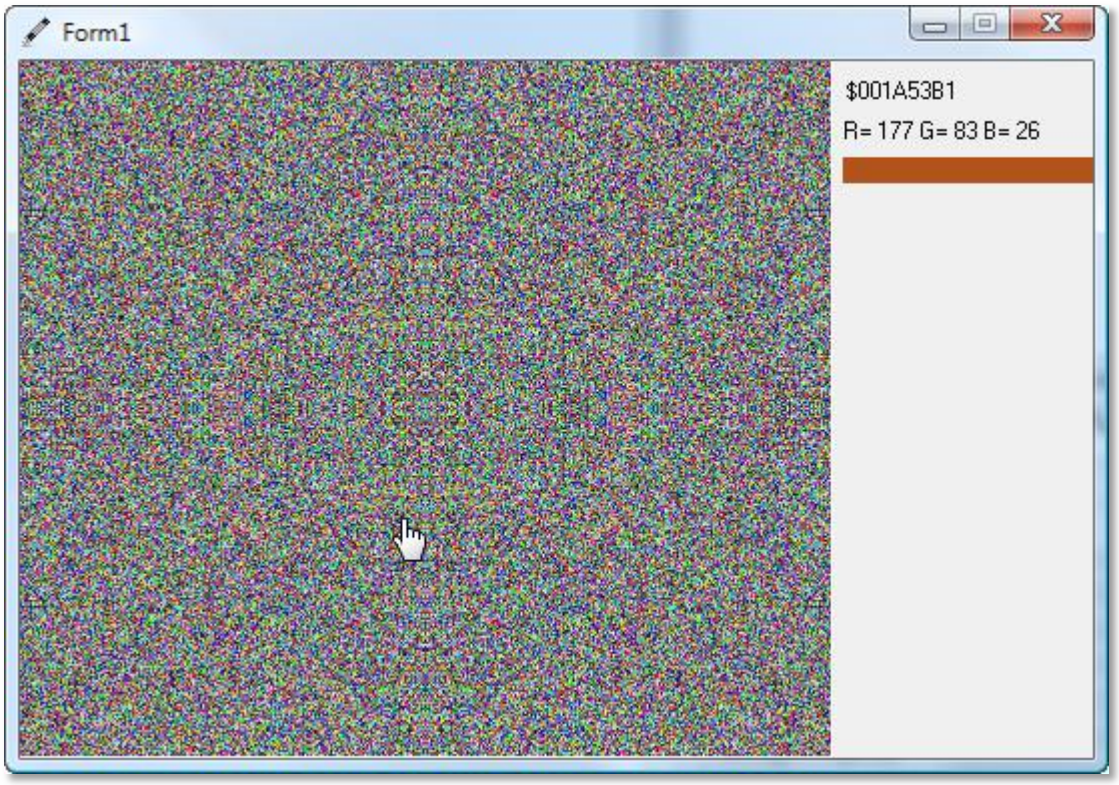


Рис. П6.3. Пипеточное приложение

Не скажешь, конечно, что глаз не оторвать, но ученье – не всегда весёлое занятие. Ну да ладно – дальше будет забавнее.



Исходный код программы находится в папке **pixel2**.

Геометрические фантазии, или Методы *TCanvas*

Одна точка, как мы убедились, - унылое зрелище. А вот через две точки уже можно провести прямую, что, безусловно, может согреть душу эстета.

Начальная точка прямой задаётся посредством метода

```
MoveTo(X, Y: Integer);
```

Если он вам не по нраву, можете воспользоваться свойством канвы

```
property PenPos: TPoint;
```

которое содержит текущие координаты пера **Pen** и позволяет не только устанавливать перо в нужную позицию, но и считывать текущую позицию пера.
Сказав «А», нужно сказать и «Б»: координаты второй, конечной точки прямой указываются непосредственно в методе

```
LineTo(X, Y: Integer);
```

После вызова этого метода текущей позицией пера станет конечная точка прямой, а на канве появится долгожданная, хотя и весьма прямолинейная, фигура. Её цвет и толщина определяются свойствами пера, о котором речь впереди.

Как всякий кулинарный рецепт начинается со слов «возьмите чистую тарелку», так и у нас то же самое: запустите *Delphi* и на чистой форме, у нижней её границы поставьте главную кнопку **sbtClear** типа *TSpeedButton*, щёлкните на ней от души и напишите что следует:

```
//ОЧИСТИТЬ ПОЛЕ
procedure TForm1.sbtClearClick(Sender: TObject);
begin
  //сплошная заливка
  canvas.Brush.Style:= bsSolid;
  //белым цветом
  canvas.Brush.Color:= clWhite;
  //залить поле для рисования
  canvas.FillRect(Rect(0,0,CanvasWidth-1,CanvasHeight-1))
end;
```

Эта процедура будет добросовестно стирать с канвы то, что мы намараем, - достаточно нажать кнопку (типа сказочного принципа «дёрни за верёвоч-

ку – дверь и откроется»). Она-то сделает, а мы ещё не того – не объявили в типе формы две переменные:

```
private
    { Private declarations }
  CanvasWidth, CanvasHeight: integer;
```

И этого мало: нужно при запуске программы устанавливать размеры поля и подготавливать его к работе, окрашивая в белый цвет. Первого зайца убьём при создании формы:

```
//СОЗДАТЬ ФОРМУ
procedure TForm1.FormCreate(Sender: TObject);
begin
    //размер канвы:
    CanvasWidth:=Clientwidth;
    CanvasHeight:=Clientheight-40;
end;
```

(О размерах канвы следует сказать честно, как о налоге в России: её размеры хранятся в свойстве

```
property ClipRect: TRect;
```

и устанавливаются системой в зависимости от контекста устройства, с которым связана канва, мы можем только узнать, что она натворила, изменить размеры канвы нам не дано. Хотя никто не может заставить нас рисовать на всей канве – режьте, сколько нужно.)

А вдогонку – второго грызуна:

```
procedure TForm1.FormPaint(Sender: TObject);
begin
  sbtClearClick(Self);
end;
```

Памятуя о том, что точка – это очень короткая линия, опять же понаставим точек. Но теперь это будет не пиксельная мелочь, а «крупная рыба»! Рядышком с кнопкой *sbtClear* подсаживаем её подружку **sbtPoints** и описываем сопутствующую ей процедуру:

```
//СТАВИМ ТОЧКИ
procedure TForm1.sbtPointsClick(Sender: TObject);
Var
  x1, y1:integer;
  clr: TColor;
  i :integer;
begin
```

```
  randomize;
  //размер точки:
  Canvas.Pen.Width:= 5;
  for i:= 1 to 10000 do
  begin
    //выбираем случайный цвет
    clr:= rgb (random(256),random(256),random(256));
    Canvas.Pen.Color:=clr;
    //координаты точки
    x1:= random(CanvasWidth-Canvas.Pen.Width div 2 - 1);
    y1:= random(CanvasHeight-Canvas.Pen.Width div 2 - 1);
    //задаём позицию точки
    Canvas.MoveTo(x1,y1);
    Canvas.LineTo(x1,y1);
  end;
end;
```

Собравшись с духом, жмите кнопку – и вот уже посыпались на канву разноцветные точки! Правда, квадратные, что не очень совпадает с нашими понятиями о форме точек (вообще-то геометрические точки никакой формы не имеют, так как «безразмерны»), поэтому увеличьте толщину пера до 15, 25 пикселей (`Canvas.Pen.Width:= …;`**).** Запустите программу – и тут начнётся: уже не банальные точки, а праздничные конфетти порадуют вас не только цветом, но и формой. А вот точки размером в 1 пиксель этим способом нарисовать не удастся – да и ни к чему это…

Но, как говорится, не за тем пришли, пора заняться делом, то есть прямыми. А для этого поставим на форму третью кнопку – **sbtLines**, и дальше – как положено:

```
//ЧЕРТИМ ЛИНИИ
procedure TForm1.sbtLinesClick(Sender: TObject);
Var
  x1, y1, x2, y2 :integer;
  clr: TColor;
  i :integer;
begin
  randomize;
  Canvas.Pen.Width:= 2;
  for i:= 1 to 500 do
  begin
    //выбираем случайный цвет линии
    clr:= rgb (random(256),random(256),random(256));
    Canvas.Pen.Color:=clr;
    //координаты первой точки
    x1:= random(CanvasWidth-Canvas.Pen.Width div 2 - 1);
    y1:= random(CanvasHeight-Canvas.Pen.Width div 2 -1);
    //задаём позицию пера
    Canvas.MoveTo(x1,y1);
    //координаты второй точки
```

```
    x2:= random(CanvasWidth-Canvas.Pen.Width div 2 -1);
    y2:= random(CanvasHeight-Canvas.Pen.Width div 2 -1);
    //проводим линию
    Canvas.LineTo(x2,y2);
    sleep(30);
  end;
end;
```

Тут уж ничего не добавишь: картина ещё краше, чем с точками. А мало этого, увеличьте толщину пера до 20 – и смотрите сами.

А дальше и того пуще: рисуем прямоугольники! Для этого имеется полезный во всех отношениях метод

`Rectangle(X1, Y1, X2, Y2: Integer);`

который рисует прямоугольник, верхний левый угол которого задаётся координатами (**X1, Y1**), а нижний правый - координатами (**X2, Y2**). Цвет контура определяется цветом пера, а заливки – цветом кисти **Brush**, о которой речь тоже впереди. Ставьте на форму подходящую кнопку – и вперёд:

```
//РИСУЕМ ПРЯМОУГОЛЬНИКИ
procedure TForm1.sbtRectsClick(Sender: TObject);
Var
  x1, y1, x2, y2 :integer;
  clr: TColor;
  i :integer;
begin
  randomize;
  Canvas.Pen.Width:= 2;
  Canvas.Brush.Style:= bsSolid;
  for i:= 1 to 500 do
  begin
    //выбираем случайный цвет линии
    clr:= rgb (random(256),random(256),random(256));
    Canvas.Pen.Color:=clr;
    //выбираем случайный цвет кисти:
    clr:= rgb (random(256),random(256),random(256));
    Canvas.Brush.Color:=clr;
    //координаты верхнего левого угла прямоугольника:
    x1:= random(CanvasWidth-1); y1:= random(CanvasHeight-1);
    //координаты верхнего левого угла прямоугольника:
    x2:= random(CanvasWidth-1); y2:= random(CanvasHeight-1);
    //чертим прямоугольник с границей и заливкой:
    Canvas.Rectangle(x1,y1,x2,y2);
    sleep(30);
  end;
end;
```

А теперь можно и визуально порадоваться нашим «канвасным» достижениям (Рис. П6.4, 5, 6).

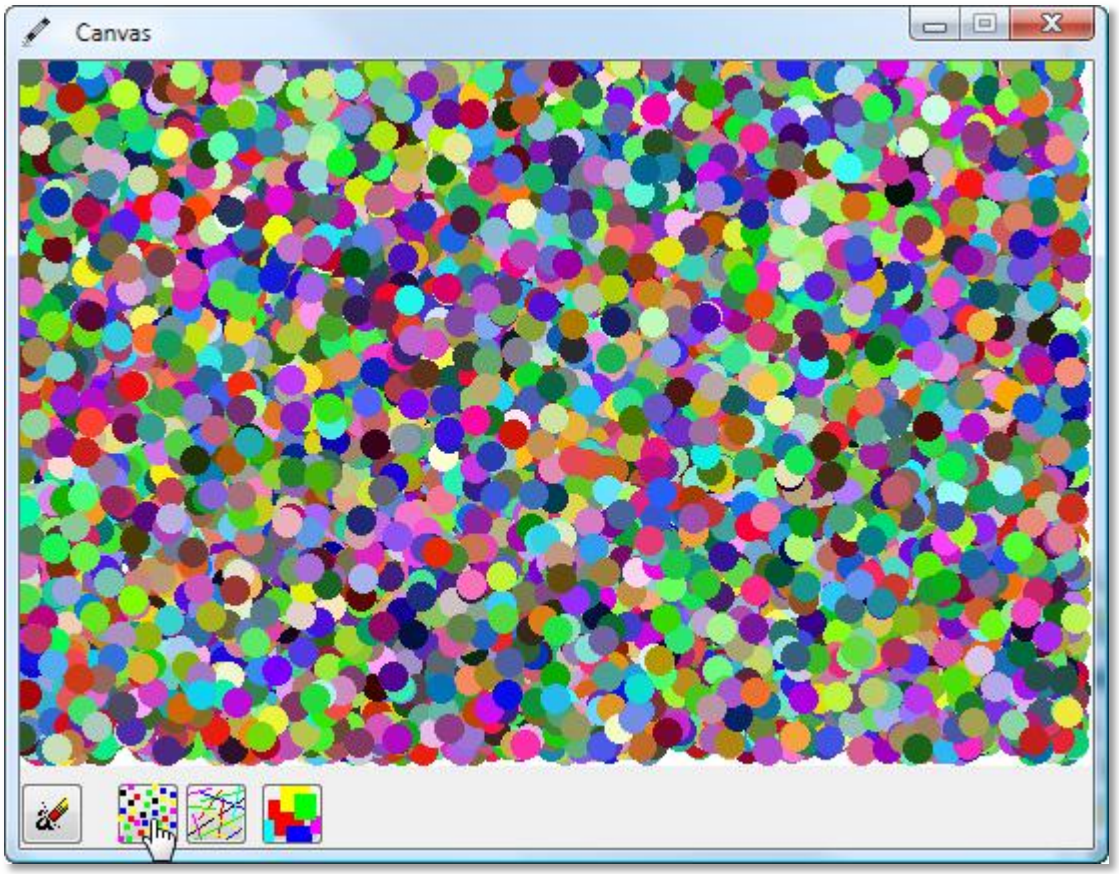


Рис. П6.4. Цветные точки

Кажется, неплохо получилось, а сколько ещё впереди!



Исходный код программы находится в папке **Methods**.

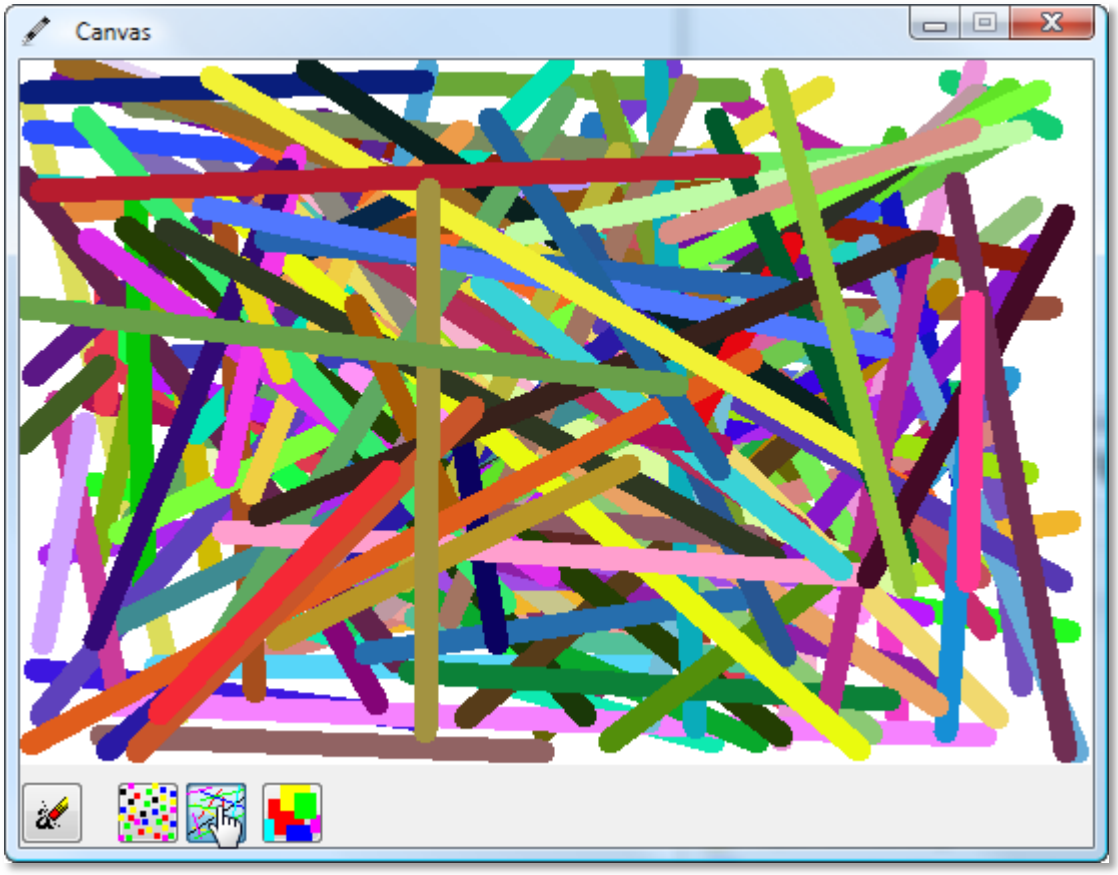


Рис. П6.5. Цветные линии

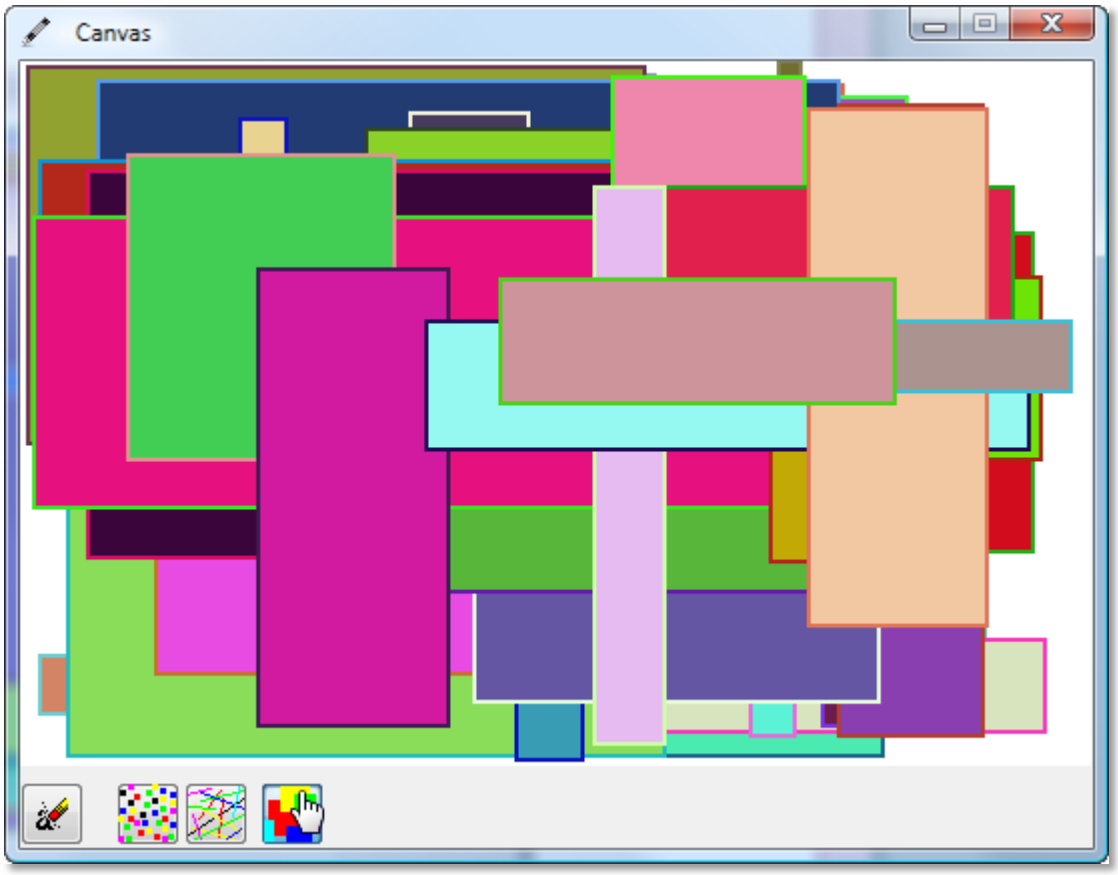


Рис. П6.6. Цветные прямоугольники

Попробуем отделить мух от супа: нарисуем отдельно контур прямоугольника и прямоугольник без контура. Для первого экзерсиса сгодится метод

FrameRect(const Rect: TRect);

Он вычерчивает контурный прямоугольник текущей **кистью**, толщина линии всегда равна 1 пикселю, а пространство внутри контура не закрашивается. Не мудрствуя лукаво, заменим в процедуре **sbtRectsClick** строку `Canvas.Rectangle(x1,y1,x2,y2);:`

```
//чертим контурный прямоугольник цветом кисти:
Canvas.FrameRect(Rect(x1,y1,x2,y2));
```

Вторую цель нетрудно одолеть с помощью метода

FillRect(const Rect: TRect);

Он закрашивает прямоугольную область текущей кистью, без оконтуривания. Вместо предыдущей строки наберите новую:

```
//чертим прямоугольник без границы:
Canvas.FillRect(Rect(x1,y1,x2,y2));
```

Для рисования контурного «точечного» прямоугольника можно воспользоваться методом

DrawFocusRect(const Rect: TRect);

Он действует аналогично методу *FrameRect*, но контур прорисовывается точками, как на элементе управления, имеющем фокус ввода, для чего он и предназначен - прямоугольник выводится на канву с использованием операции **XOR**, поэтому повторный вызов этого метода с тем же параметром удалит прямоугольник с экрана. Чтобы посмотреть, как действует этот метод, замените предыдущую строку:

```
//чертим контурный точечный прямоугольник:
Canvas.DrawFocusRect(Rect(x1,y1,x2,y2));
```

Затем немного подождём и сотрём его:

```
sleep(100);
//стираем контурный точечный прямоугольник:
Canvas.DrawFocusRect(Rect(x1,y1,x2,y2));
```

Можно несколько сгладить угловатость прямоугольника, закруглив ему углы. Тут нам на подмогу придёт метод

RoundRect(X1, Y1, X2, Y2, X3, Y3: Integer);

Здесь, как и раньше, (**X1, Y1**) и (**X2,Y2**) – координаты левого верхнего и правого нижнего углов прямоугольника (с обычными углами), а **X3** и **Y3** – размеры горизонтальной и вертикальной осей эллипса, сегментом которого является каждое из закруглений. Контур вычерчивается текущим пером, а поверхность закрашивается текущей кистью.

Для визуализации эффекта опять же набираем новую строку:

```
//чертим прямоугольник с закруглёнными углами:
Canvas.RoundRect(x1,y1,x2,y2, random(50),random(50));
```

Пойдём ещё дальше и закруглим прямоугольник настолько, что он превратится в симпатичный эллипс. Для вычерчивания этих штучек припасён метод

Ellipse (X1, Y1, X2, Y2: Integer);

Он вычерчивает эллипс текущим пером и закрашивает текущей кистью. Координаты (**X1, Y1**) и (**X2,Y2**) – координаты левого верхнего и правого нижнего углов описанного вокруг эллипса прямоугольника. Продолжаем идти проторённой дорожкой и набираем код:

```
//чертим эллипс:
Canvas.Ellipse(x1,y1,x2,y2);
```

Для визуальных гурманов в *Windows API* имеется функция

InvertRect(DC: HDC; RECT : TRECT): BOOL;

Она *инвертирует* цвета пикселей в указанной прямоугольной области, используя логическую операцию **NOT**. Допишите в конце процедуры рисования прямоугольников такой код:

```
  for i:=1 to 10 do
  begin
    InvertRect(Canvas.Handle, Rect(0,0, CanvasWidth-1, Can-
vasHeight-1));
    sleep(200);
    InvertRect(Canvas.Handle, Rect(0,0, CanvasWidth-1, Can-
vasHeight-1));
    sleep(200);
  end
```

- и после вывода на экран выбранных фигур они вам подмигнут на прощанье. Но мы не прощаемся, а, прежде чем заняться более замысловатыми фигурами, устроим себе рекламную паузу, то есть напишем программу, которая будет нас некоторое время развлекать вычерчиванием прямоугольников и эллипсов на экране.



Исходный код программы находится в папке **Methods**.

Запустите *Delphi* и растяните форму до хороших размеров, в нижнем правом углу поставьте пару кнопок: одна будет запускать программу, другая – останавливать её. Обрабатывать нажатие первой кнопки будет процедура **BitBtn1Click**:

```
//РИСУЕМ ФИГУРЫ
procedure TForm1.BitBtn1Click(Sender: TObject);
var
  xmax, ymax: integer;
  x1, y1, x2, y2: integer;
  x10, y10: integer;
  i, j: integer;
begin
  randomize;
  //размеры области рисования
  xmax:= ClientWidth- 80;
  ymax:= ClientHeight-1;
  x10:= 0; y10:= 0;
  canvas.Pen.Width:= 2; //толщина линий
  canvas.Brush.Style:= bsClear;
  repeat
    for i:= 6 downto 1 do
      for j:= 0 to 5 do
      begin
        //стартовый прямоугольник
        x1:= x10; y1:= y10; x2:= xmax; y2:= ymax;
        canvas.Pen.Color:= random($1000000); //задать цвет линий
        while (x1<= xmax) and (y1<= ymax) do
        begin
          //нарисовать прямоугольник
          Canvas.Rectangle(x1, y1, x2, y2);
          //нарисовать эллипс
          //Canvas.Ellipse(x1, y1, x2, y2);
          sleep(5); //задержка
          application.ProcessMessages;
          if flgCancel= True then  //если нажата кнопка Cancel,
          begin                    //то закончить рисование
            flgCancel:= False;
            exit
          end;
```

```
          //новые координаты вершин прямоугольника
          x1:= x1+i; y1:= y1+j;
          x2:= x2-i; y2:= y2-j;
        end;
    end;
  until false;
end;
```

А остановить шоу можно второй кнопкой:

```
//ОСТАНОВИТЬ РИСОВАНИЕ
procedure TForm1.BitBtn2Click(Sender: TObject);
begin
  flgCancel:= True;
end;
```

Опишите переменную **flgCancel** после раздела *uses*, она будет сигнализировать о вашем желании остановить бесконечный цикл в предыдущей процедуре:

```
var
  flgCancel: boolean = false;
```

Для разнообразия можете в скучном месте повторно нажимать первую кнопку для запуска ещё одного цикла (тогда и для остановки придётся нажимать вторую кнопку столько же раз).

Если заменить прямоугольники более ладными эллипсами, то картина будет ещё более завораживающая! Конечно, динамичную смену форм на экране невозможно передать статичной картинкой, но вы увидите примерно вот что (Рис. П6.7-8).

Вот так, простыми средствами можно получить удивительный результат. Блестящим подтверждением этому может служить игра *Жизнь*, которой мы в своё время займёмся, а пока с помощью всего одного метода и одного свойства **TCanvas**, правда вручную, нарисуем удивительные картины.



Исходный код программы находится в папке **Rectangular**.

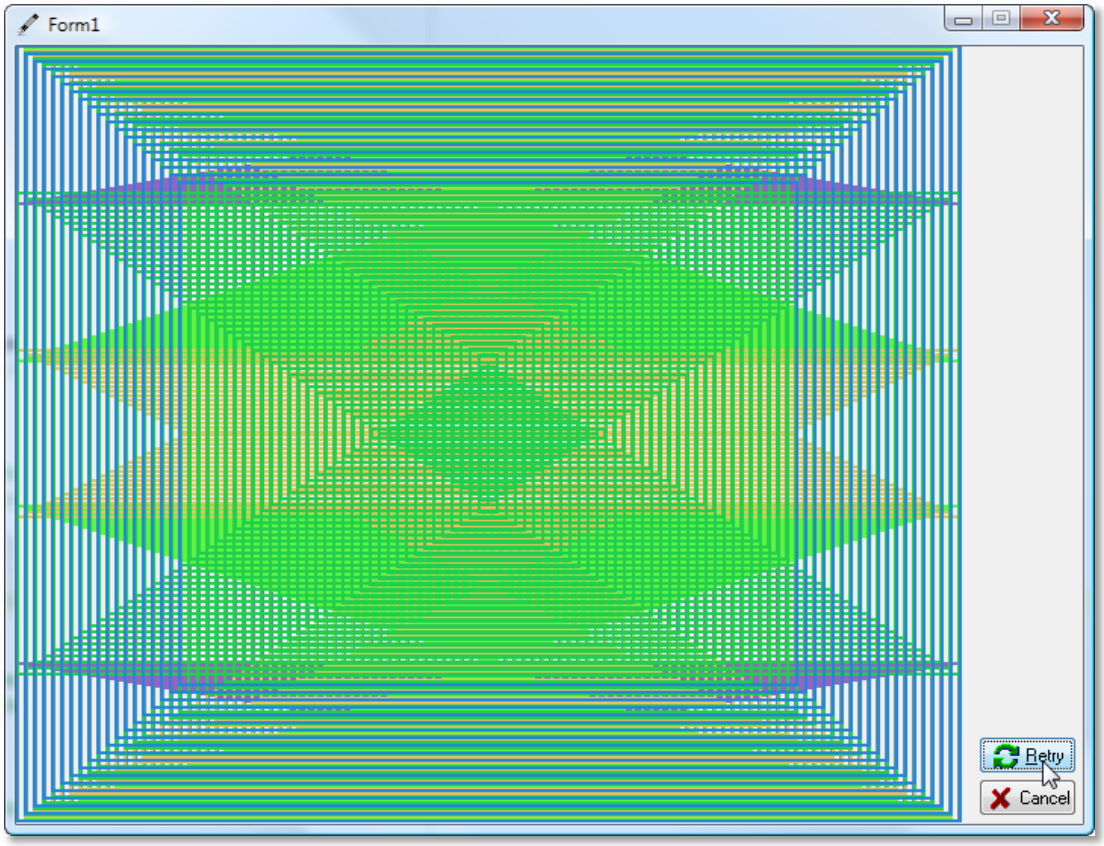


Рис. П6.7. Цветные прямоугольники

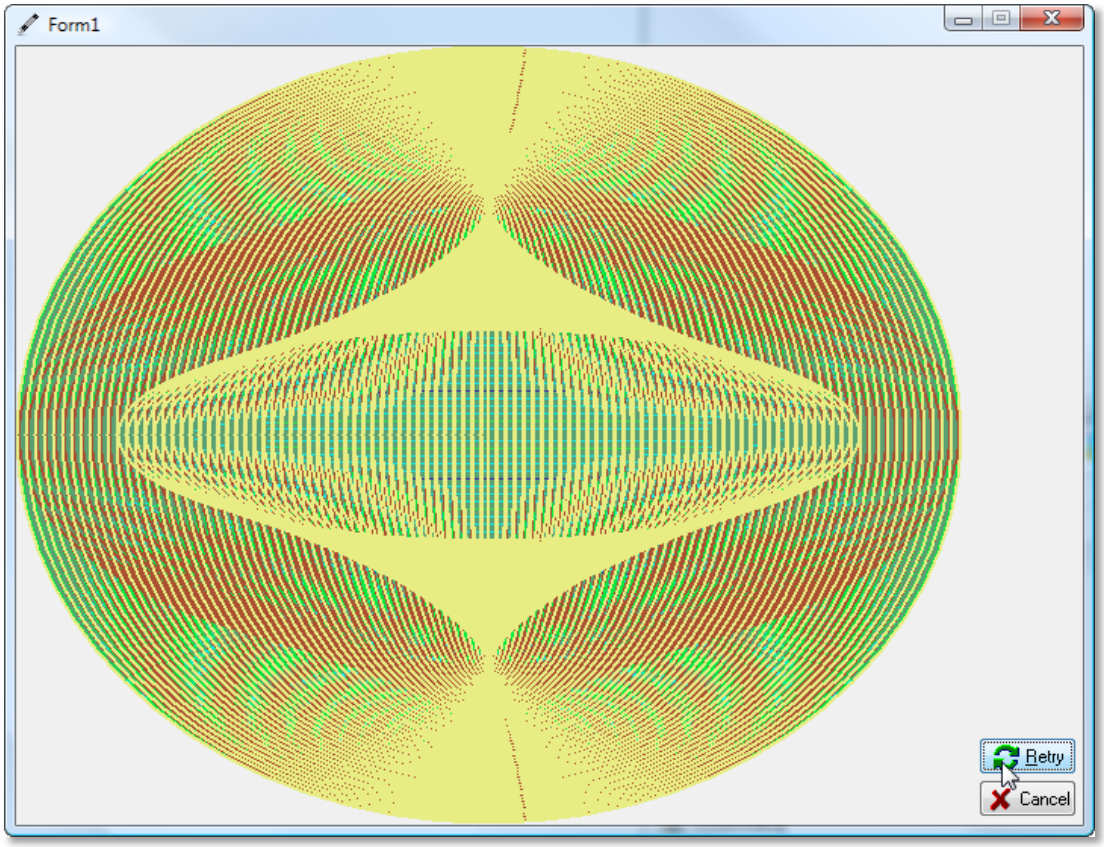
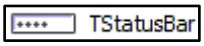


Рис. П6.8. Цветные эллипсы

Начните новый проект и установите свойства **формы**:

BorderIcons → biMaximize= false
BorderStyle=bsSingle (не нужно портить картинку, изменяя размеры формы)
Caption=' Liner'
Width=745
Height=553
Name=frmLiner
Position=poDesktopCenter)

На неё установите компонент **StatusBar1** 

(вкладка **Win32** *Палитры компонентов*), для вывода подсказок и компонент **TImage**, на котором мы и будем рисовать хитрые загогулины. То же самое можно изобразить и непосредственно на канве формы, но **TImage** всё-таки удобнее, так как можно легко настроить область рисования. Задайте свойствам **TImage** такие значения:

Cursor=crHandPoint
Height=465
Left=0
Name=PaintBox
Width=737

В верхний правый угол поставьте кнопку **sbtExit** типа *TSpeedButton*:

Cursor=crHandPoint - для всех остальных кнопок установите такое же значение
Flat=True - остальные кнопки пусть тоже будут плоскими
Glyph → подберите подходящую картинку
Height=32
Width=32 - все кнопки такого размера
Hint=Выход|Выход из программы
Name=sbtExit
ShowHint=True - также у всех кнопок. Щёлкните дважды на кнопке и напишите:

```
 //ЗАКРЫТЬ ПРОГРАММУ
procedure TfrmLiner.sbtExitClick(Sender: TObject);
begin
  close
end;
```

Впрочем, сначала нужно как следует попользоваться программой, поэтому на вкладке **Events** в *Инспекторе объектов* при выбранной форме **frmLiner** щёлкните на **OnShow** - появится заготовка для обработки этого события. В ней наберите этот код:

```
//НАЧАЛЬНЫЕ УСТАНОВКИ
```

```
procedure TfrmLiner.FormShow(Sender: TObject);
begin
    //размеры области рисования:
   pbWidth:= PaintBox.Width;
   pbHeight:=  PaintBox.Height;
   //фон - белый:
   PaintBox.Canvas.FillRect(PaintBox.ClientRect);
end;
```

Объявите *переменные* в разделе *private* типа формы:

```
pbWidth, pbHeight: integer;
```

они нужны, собственно, только для сокращения писанины.

Если вы теперь запустите программу, то увидите только белое поле, что, согласитесь, после увиденного ранее, скучновато, поэтому нажмите на единственную кнопку и закройте программу. Правильно – надо учиться рисовать. Делается это обычной мышкой, причём действия всегда одни и те же: нажимаем кнопку – ставим начальную точку, двигаем мышь – рисуем, отпускаем кнопку мыши – заканчиваем рисование. Чтобы отличать простое перемещение мышки от рисования, введём переменную **Drawing** (объявите её там же, где и первые две:

```
Drawing: Boolean;
```

Если мы рисуем, то устанавливаем **Drawing** в *True*, заканчиваем – в *False*. Естественно, нужно запомнить ту точку, в которой находился курсор, когда мы нажали кнопку мыши, - она нам потребуется в дальнейшем для рисования закреплённых линий. Для хранения координат этой точки заведём массив **Origin**, который нужно объявить аналогично переменным:

```
Origin, BeginPt: array[1..8] of TPoint;
```

Такой же массив **BeginPt** будет хранить текущие начальные точки при рисовании незакреплённых линий. Остаётся открытым вопрос, почему для одной точки потребовался массив из 8 элементов. А всё потому, что одиночная линия – не такая уж и красавица, гораздо интереснее симметричные картинки (вспомните калейдоскоп), значит, нужно основную линию отражать относительно осей – а у новых линий тоже есть точки **Origin** и **BeginPt**. Если ограничиться простыми видами симметрии, то 8 точек как раз и хватит. Вот и вся премудрость – если забыть о вычислении координат симметричных точек. Эта «глухомань» встречается нередко, поэтому нужно бы прояснить ситуацию.

Нажав кнопку мыши в области рисования, мы устанавливаем начальную точку **Origin[1]**, которая по совместительству является и текущей начальной точкой **BeginPt[1]**. Их координаты равны координатам курсора (**X,Y**) в области рисования. Симметрично относительно оси **y** ставим точку **Origin[2]**, а симметрично относительно оси **x** – точку **Origin[3]**. Этих точек вполне достаточно, чтобы вычерчивать линии с вертикальной или горизонтальной симметрией. Но, выбрав одновременно оба вида симметрии, вы обнаружите, что правый нижний угол выпадает: там кривая отсутствует. Поэтому специально для этого случая ставим точку **Origin[4]**. На рисунке все эти точки обозначены буквой **P** с соответствующим индексом (Рис. П6.9).

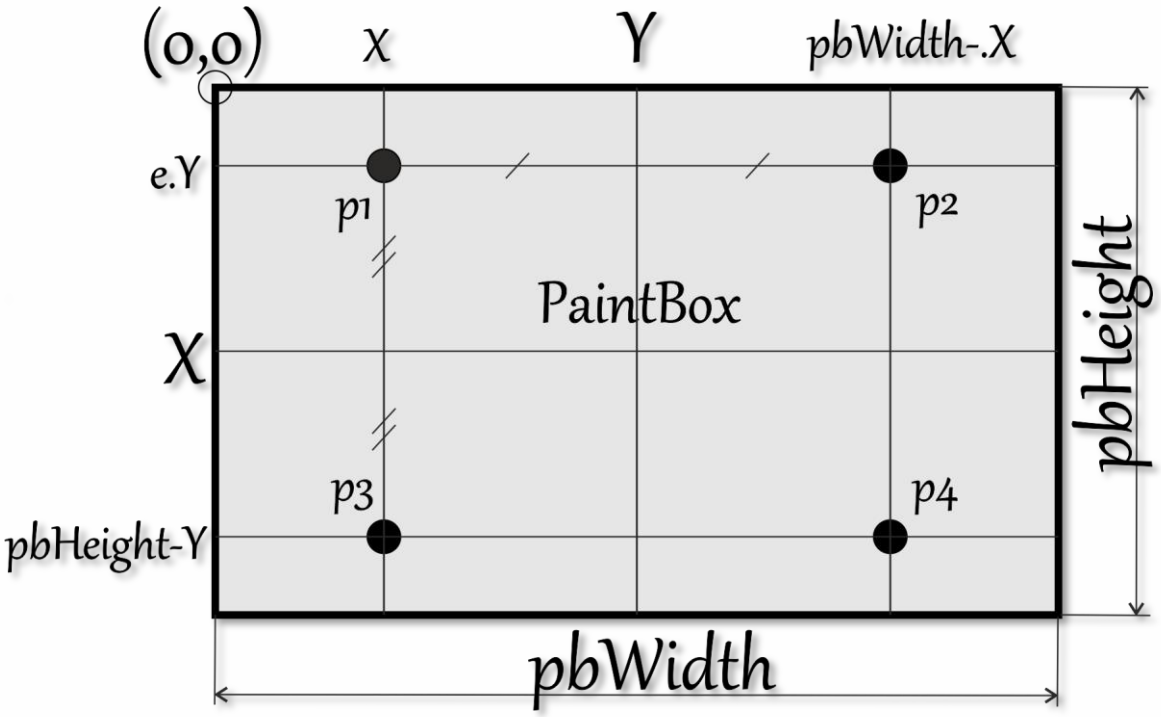


Рис. П6.9. Вычисляем координаты четырёх точек

Чтобы увеличить «упорядоченность» картинок, ставим ещё 4 точки. Отражаем **P1** относительно диагонали и получаем точку **P5**. Легко заметить, что при этом координаты точки **P1(X,Y)** меняются местами - **P5(Y, X).** Далее находим остальные 3 точки – **P6, P7, P8** – точно так же, как и в первом случае (Рис. П6.10).

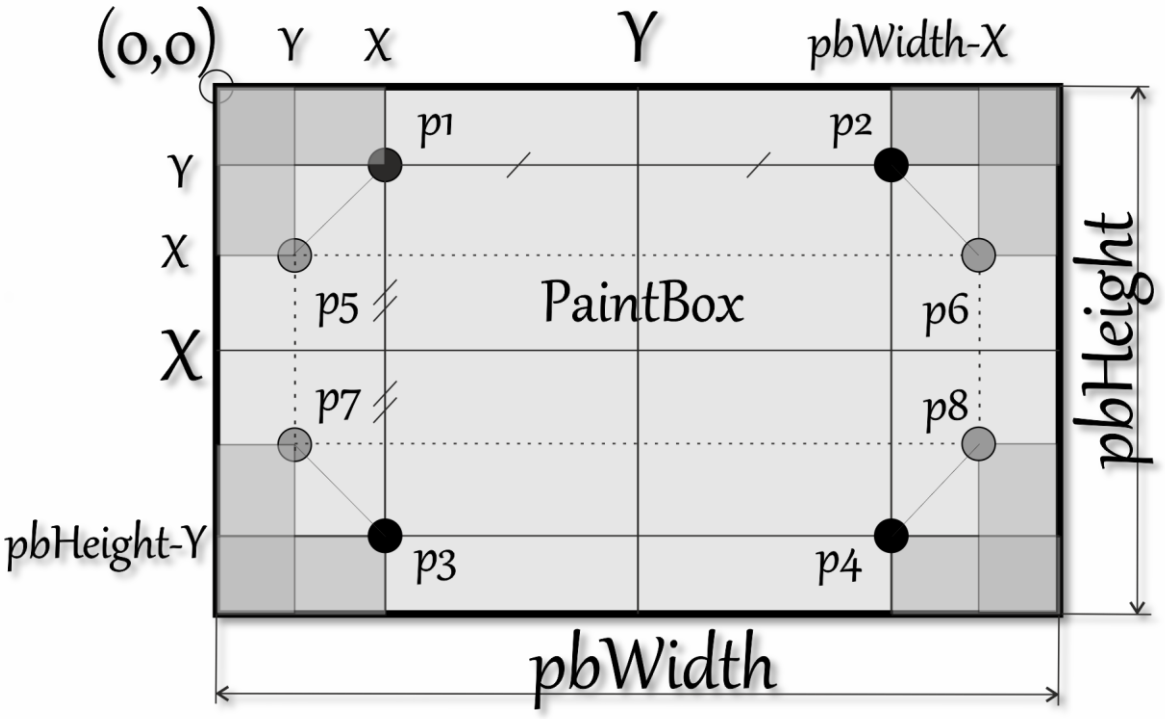


Рис. П6.10. Вычисляем координаты восьми точек

Далее все наши геометрические изыскания излагаем на дельфийском языке:

```
//НАЧАТЬ РИСОВАНИЕ
procedure TfrmLiner.PaintBoxMouseDown(Sender: TObject;
  Button: TMouseButton; Shift: TShiftState; X, Y: Integer);
begin
  Drawing := True;  //начинаем рисовать
  //PaintBox.Canvas.MoveTo(X, Y);    //устанавливаем перо в начальную точку
  Origin[1] := Point(X, Y);        //запоминаем начальную точку
  BeginPt[1] := Origin[1];          //она же пока - конечная
  //запоминаем точку, симметричную относительно оси X:
  Origin[2] := Point(pbWidth-X, Y);
  BeginPt[2] := Origin[2];
  //запоминаем точку, симметричную относительно оси Y:
  Origin[3] := Point(X,pbHeight-Y);
  BeginPt[3] := Origin[3];
  //запоминаем точку, симметричную относительно центра:
  Origin[4] := Point(pbWidth-X,pbHeight-Y);
  BeginPt[4] := Origin[4];

  //запоминаем точки, симметричные первым четырём относительно диагоналей
  Origin[5] := Point(Y, X);
  BeginPt[5] := Origin[5];
  Origin[6] := Point(pbWidth-Y, X);
  BeginPt[6] := Origin[6];
  Origin[7] := Point(Y, pbHeight-X);
  BeginPt[7] := Origin[7];
```

```
  Origin[8] := Point(pbWidth-Y, pbHeight-X);
  BeginPt[8] := Origin[8];
end;
```

Не отпуская кнопку, выводим на канве разнообразные узоры. Выбор большой: это может быть одиночная прямая, или две линии, симметричные относительно вертикальной или горизонтальных осей, или четыре линии, если заданы оба режима симметрии, или, наконец, восемь симметричных линий. Конечно, мы должны как-то включать и выключать эти режимы. Легче всего вычертить одну линию:

```
//РИСУЕМ
procedure TfrmLiner.PaintBoxMouseMove(Sender: TObject; Shift:
TShiftState;
  X, Y: Integer);
var
  i: integer;
begin
  if Drawing then
  begin
    PaintBox.Canvas.PenPos:= BeginPt[1];
    //провести линию из текущей точки в конечную
    PaintBox.Canvas.LineTo(x,y);
    //запомнить конечную точку
    BeginPt[1] :=  Point(X, Y);
  End
End
```

Если включён режим рисования, то проводим линию из **BeginPt[1]** в точку, где находится мышь, и запоминаем эту точку как текущую.

Если задана горизонтальная симметрия, то проводим вторую линию:

```
    //горизонтальная симметрия
    if sbtSymH.down=true then
    begin
      PaintBox.Canvas.PenPos:= BeginPt[2];
      PaintBox.Canvas.LineTo(pbWidth-x, y);
      //запомнить конечную точку
      BeginPt[2] :=  Point(pbWidth-X, Y);
    end;
```

И вот тут нам никак не обойтись без кнопки **sbtSymH**, которая в нажатом положении будет сигнализировать о включённом режиме горизонтальной симметрии. Из всего вышесказанного однозначно следует, что потребуются ещё две кнопки для переключения режимов – **sbtSymV** и **sbtSymA**, которые будут отвечать за вертикальную и суперсимметрию. Все эти кнопки – типа *TSpeedButton*. Чтобы их можно было фиксировать в нажатом поло-

жении, их свойство **GroupIndex** нужно установить, соответственно, в **3, 4, 5**. А чтобы их можно было выключать из нажатого положения, свойство **AllowAllUp** должно равняться *True*. Все эти кнопки действуют независимо друг от друга, но при нажатой кнопке **sbtSymA** должны включаться и кнопки **sbtSymH** и **sbtSymV**, так как следует рисовать все восемь линий:

```
//СУПЕРСИММЕТРИЯ
procedure TfrmLiner.sbtSymAClick(Sender: TObject);
begin
  //включить вертикальную и горизонтальную симметрии:
  sbtSymH.Down:= true;
  sbtSymV.Down:= true;
end;
```

Нужно также запретить выключать эти кнопки, пока нажата **sbtSymA**:

```
//ГОРИЗОНТАЛЬНАЯ СИММЕТРИЯ
procedure TfrmLiner.sbtSymHClick(Sender: TObject);
begin
  //нельзя отключить, если суперсиметрия:
  if sbtSymA.Down then sbtSymH.Down:= True
end;

//ВЕРТИКАЛЬНАЯ СИММЕТРИЯ
procedure TfrmLiner.sbtSymVClick(Sender: TObject);
begin
  //нельзя отключить, если суперсиметрия:
  if sbtSymA.Down then sbtSymH.Down:= True
end;
```

Пришло время вспомнить о точках **Origin**, которые мы запомнили, но нигде пока не использовали. Они нам пригодятся для особого режима рисования, когда конечная точка прямой соединяется не с текущей начальной **BeginPt**, а с закреплённой точкой **Origin**. Для включения этого режима нужна кнопка **sbtLock** (**GroupIndex**=1, **AllowAllUp**= True).

Теперь вполне можно рисовать *симметричные* узоры:

```
//РИСУЕМ
procedure   TfrmLiner.PaintBoxMouseMove(Sender:   TObject;   Shift:
TShiftState;
  X, Y: Integer);
begin
  if Drawing then
  begin
    //если начальная точка закреплена, то начинаем линию из неё
    if sbtLock.Down
      then PaintBox.Canvas.PenPos:= Origin[1]
    else PaintBox.Canvas.PenPos:= BeginPt[1];
```

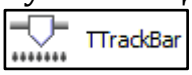
```
    //провести линию из текущей точки в конечную
    PaintBox.Canvas.LineTo(x,y);
    //запомнить конечную точку
    BeginPt[1] :=  Point(X, Y);

    //горизонтальная симметрия
    if sbtSymH.down=true then
    begin
      if sbtLock.Down
        then PaintBox.Canvas.PenPos:= Origin[2]
      else PaintBox.Canvas.PenPos:= BeginPt[2];
      PaintBox.Canvas.LineTo(pbWidth-x, y);
      //запомнить конечную точку
      BeginPt[2] :=  Point(pbWidth-X, Y);
    end;
  //вертикальная симметрия
  if sbtSymV.down then
  begin
    if sbtLock.Down
       then PaintBox.Canvas.PenPos:= Origin[3]
     else PaintBox.Canvas.PenPos:= BeginPt[3];
     PaintBox.Canvas.LineTo(x,pbHeight- y);
     //запомнить конечную точку
     BeginPt[3] :=  Point(x,pbHeight- y);
  end;

  //вертикальная  и горизонтальная симметрия
  if (sbtSymH.down) and (sbtSymV.down) then
  begin
    if sbtLock.Down
      then PaintBox.Canvas.PenPos:= Origin[4]
    else PaintBox.Canvas.PenPos:= BeginPt[4];
    PaintBox.Canvas.LineTo(pbWidth-X,pbHeight- y);
    //запомнить конечную точку
    BeginPt[4] :=  Point(pbWidth-X,pbHeight- y);
  end;

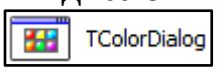
  //суперсимметрия
  if sbtSymA.down then
  begin
    if sbtLock.Down
      then PaintBox.Canvas.PenPos:= Origin[5]
    else PaintBox.Canvas.PenPos:= BeginPt[5];
      PaintBox.Canvas.LineTo(Y, X);
    BeginPt[5] :=  Point(Y, X);
    if sbtLock.Down
      then PaintBox.Canvas.PenPos:= Origin[6]
    else PaintBox.Canvas.PenPos:= BeginPt[6];
    PaintBox.Canvas.LineTo(pbWidth-Y, X);
    BeginPt[6] :=  Point(pbWidth-Y, X);
    if sbtLock.Down
      then PaintBox.Canvas.PenPos:= Origin[7]
    else PaintBox.Canvas.PenPos:= BeginPt[7];
```

```
      PaintBox.Canvas.LineTo(Y, pbHeight-X);
      BeginPt[7] :=  Point(Y, pbHeight-X);
      if sbtLock.Down
        then PaintBox.Canvas.PenPos:= Origin[8]
      else PaintBox.Canvas.PenPos:= BeginPt[8];
      PaintBox.Canvas.LineTo(pbWidth-Y, pbHeight-X);
      BeginPt[8] :=  Point(pbWidth-Y, pbHeight-X);
    end;
end;
```

Всё бы хорошо, да линии толщиной в 1 пиксель не кажутся убедительными. Конечно, можно задать в программе линии и посолиднее, но лучше потрудиться и сделать так, чтобы можно было устанавливать любую толщину линии. Для этого рядом с кнопками установим компонент **TTrackBar1** 

(вкладка **Win32** *Панели компонентов*) и придадим ему необходимые свойства: **Max**=40 (максимальная толщина линий), **Min**=1 (минимальная толщина линий). При изменении положения движка будет изменяться и толщина линий:

```
//ЗАДАТЬ ТОЛЩИНУ ЛИНИЙ
procedure TfrmLiner.TrackBar1Change(Sender: TObject);
begin
  PaintBox.Canvas.Pen.Width:= TrackBar1.Position;
  //показать числовое значение
  TrackBar1.Hint:= inttostr(TrackBar1.Position);
end;
```

Движок позволяет изменять толщину линий, но они всегда однообразно чёрные. Это угнетает, поэтому добавим кнопку **sbtColor**, которая будет вызывать стандартный диалог выбора цвета (добавляем на форму компонент **TColorDialog1** 

- вкладка **Dialogs** *Панели компонентов*). Действует она так:

```
//ВЫБРАТЬ ЦВЕТ ПЕРА
procedure TfrmLiner.sbtColorClick(Sender: TObject);
begin
  if ColorDialog1.execute then
  begin
    PaintBox.Canvas.Pen.Color:= ColorDialog1.Color;
    label1.Color:= ColorDialog1.Color
  end
end;
```

Чтобы видеть установленный цвет, справа от кнопки установим метку **Label1**.

А душа всё равно хочет праздника с его яркими красками. Вспомним о градиентной заливке и закрепим её за кнопкой **sbtGrad.** Цветовой переход вы можете выбрать любой из уже освоенных нами, в данном примере цвет линии при рисовании будет изменяться от циана к синему, а затем в обратную сторону. Направление изменение цвета будет храниться в переменной **Grad**, объявите её там же, где и **Drawing**:

```
Grad: Boolean;
```

Она равняется *TRUE*, если интенсивность увеличивается, и *FALSE*, если уменьшается. Процедура, обрабатывающая нажатие на кнопку:

```
//ГРАДИЕНТНОЕ ИЗМЕНЕНИЕ ЦВЕТА ЛИНИЙ
procedure TfrmLiner.sbtGradClick(Sender: TObject);
begin
  if sbtGrad.down then
  begin
    grad:= true;
    PaintBox.Canvas.Pen.Color:= rgb(0,255,255);
  end
end;
```

Ещё более цветастыми получатся линии, если их цвет изменять случайным образом. Так как одновременно эти режимы действовать не могут, то для кнопки **sbtGrad** и **sbtRandomColor** нужно установить одинаковые значения свойства **GroupIndex**=2, **AllowAllUp**= True, тогда при включении одной кнопки вторая автоматически выключится, причём могут быть выключены обе кнопки, чтобы рисовать одноцветными линиями.

Покончив с этой нудной нервотрёпкой, допишем в конец процедуры **TfrmLiner.PaintBoxMouseMove** код, обрабатывающий нажатие «цветных» кнопок:

```
  //задать случайный цвет линии:
  if sbtRandomColor.Down then
    PaintBox.Canvas.Pen.Color:= Random($1000000);
  end;

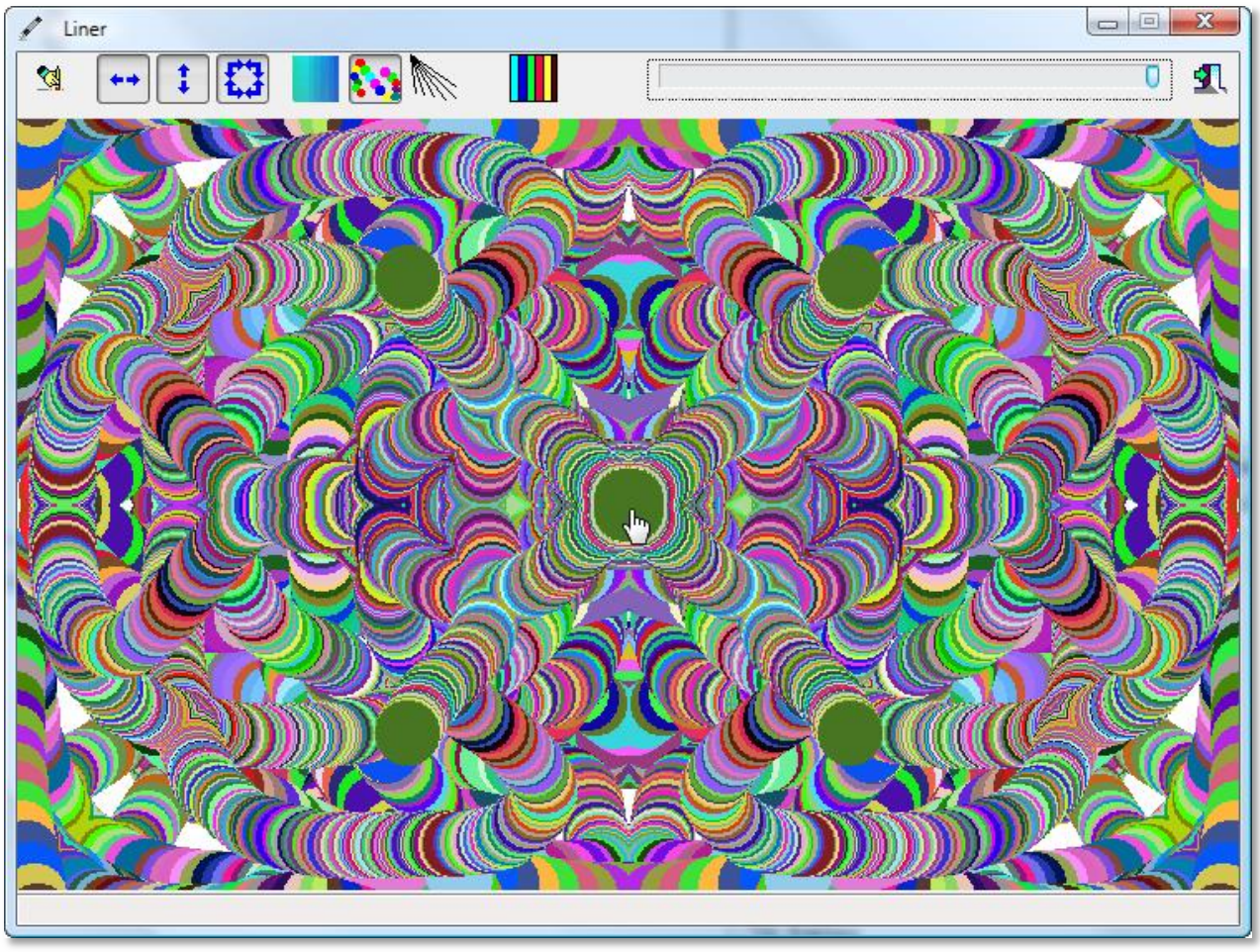
  //градиентое изменение цвета линий от циана к синему
  if sbtGrad.Down then
  begin
    //текущая интенсивность зелёной составляющей:
    i := GetGValue(PaintBox.Canvas.Pen.Color);
    if grad then //интенсивность цвета увеличивается
      begin
        inc(i);
        if i> 255 then
        begin
```

```
            i:= 255;
            grad:=not grad
          end
        end
      else          //интенсивность цвета уменьшается
        begin
          dec(i);
          if i< 0 then
          begin
            i:= 0;
            grad:=not grad
          end
        end;
      PaintBox.Canvas.Pen.Color:= rgb(0,i,255);
    end;  //Drawing
```

Не забудьте объявить переменную **i** в этой процедуре:

```
var
  i: integer;
```

Теперь ничто не помешает вам художничать от всей души - запускайте программу и радуйтесь жизни! Тут, конечно, многое зависит от мастерства, но рисунки будут выглядеть примерно так (Рис. П6.11).



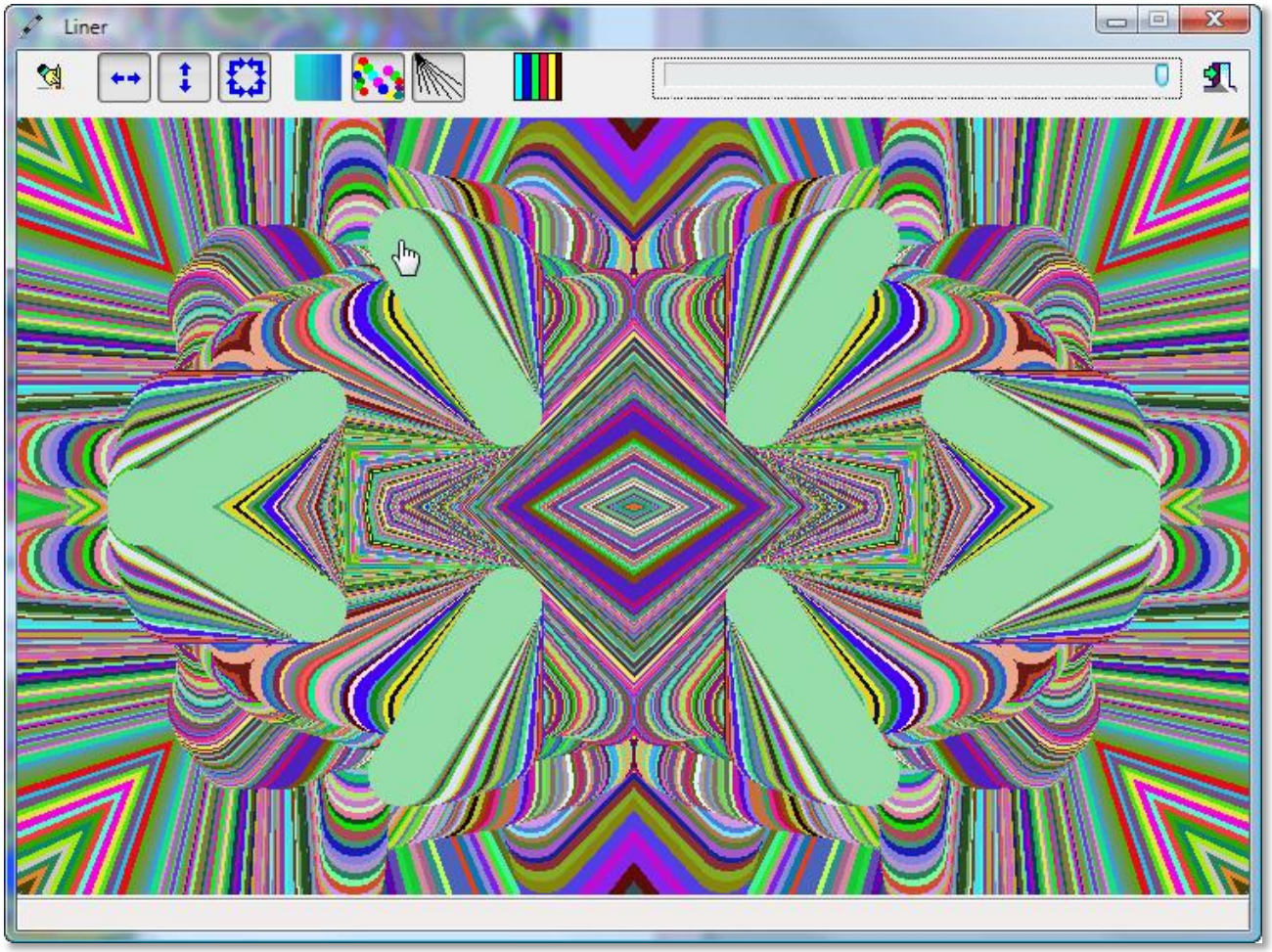


Рис. П6.11. Симметрично!

Когда вы чертите линии, мышка у вас занята этим благородным делом, поэтому нет возможности динамически изменять толщину линий, поэтому привлечём к работе клавиши **X** и **C**. Нажимая первую, мы **уменьшаем** толщину линий, вторую – **увеличиваем** (Рис. П6.12). Разумно это сделать в обработчике события нажатия клавиш на клавиатуре:

```
//ИЗМЕНИТЬ ТОЛЩИНУ ЛИНИЙ
procedure  TfrmLiner.TrackBar1KeyDown(Sender:  TObject;  var  Key:
Word;
  Shift: TShiftState);
begin
with TrackBar1 do
  if (key=88) and (Position>1) then Position:= Position-1
  else if key=67 then Position:= Position+1
end;
```



Надо отметить, что толщина линий изменяется очень быстро, поэтому следовало бы предусмотреть возможность плавного изменения толщины линий…

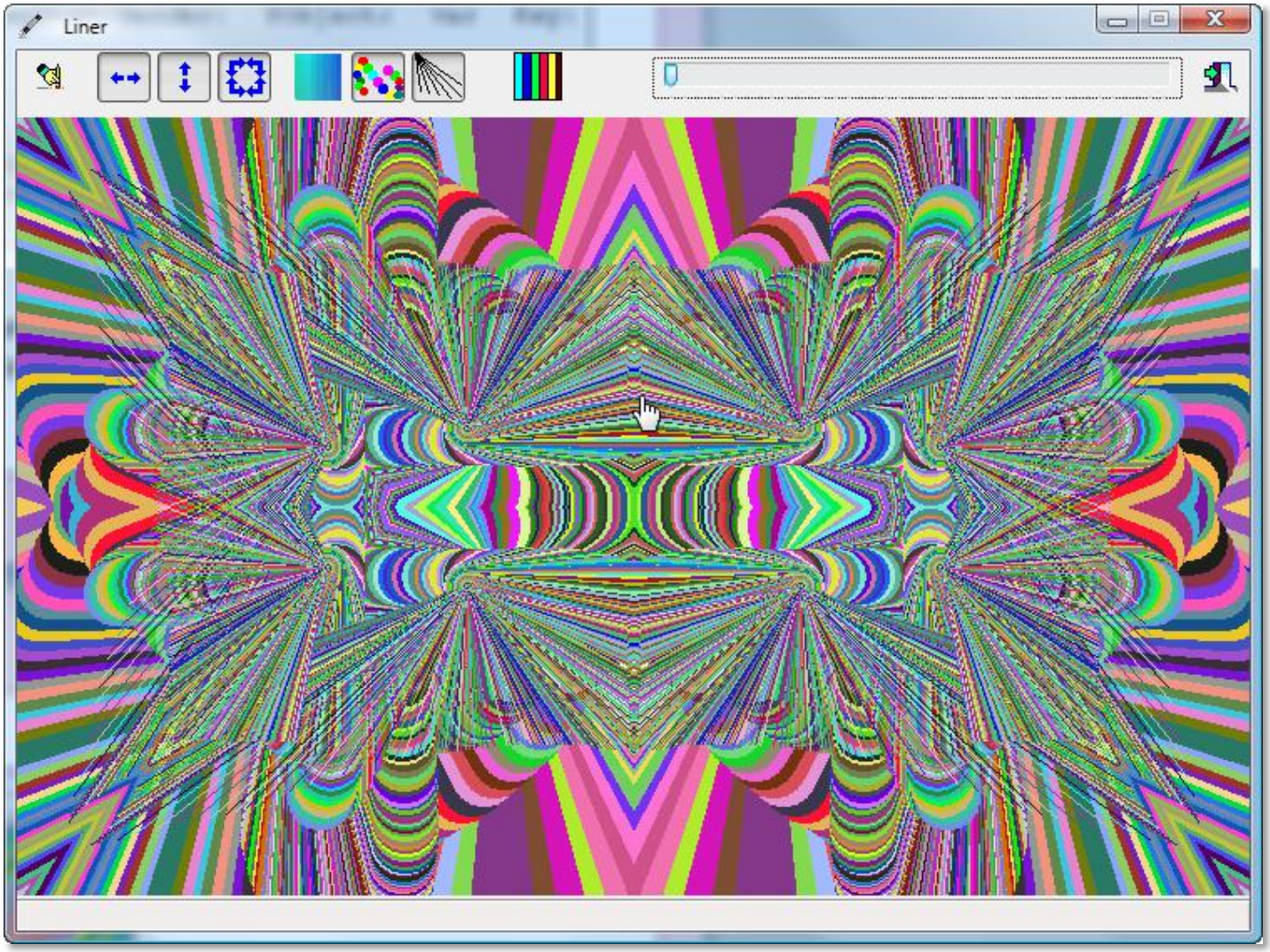


Рис. П6.12. Изменяем толщину линий

Пёстрое зрелище! Осталось только поставить кнопку **sbtClear**, чтобы время от времени стирать неудачную мазню:

```
//ОЧИСТИТЬ ПОЛЕ
procedure TfrmLiner.sbtClearClick(Sender: TObject);
begin
  PaintBox.Canvas.Brush.Color := clWhite;
  PaintBox.Canvas.FillRect(PaintBox.ClientRect);
end;
```



Исходный код программы находится в папке **Liner**.

Заливная рыба, или метод *FloodFill*

Какая гадость эта ваша заливная рыба!

Из сетований несчастного Ипполита

Кто не любил в детстве разрисовывать обои, мамины журналы или папину спортивную газету? Тот, кто не любил, спокойно может пропустить этот раздел – у него нет тяги к прекрасному. Всем остальным, коих, можно надеяться, подавляющее большинство, следует обратить внимание на один из методов канвы, который позволит вспомнить детские забавы:

```
FloodFill(X, Y: Integer; Color: TColor; FillStyle: TFillStyle);
```

Он производит заливку области канвы текущей кистью. Эта область может иметь любую форму; процесс начинается с точки, координаты которой – (**X, Y**), и продолжается до тех пор, пока есть соседние точки заданного цвета **Color** – если задан режим заливки **fsSurface**. Если выбран режим заливки f**sBorder**, то закраска прекращается на точках цвета **Color**. Увидеть различия в этих режимах поможет наша следующая программа - **Fill**.

Откройте новый проект и установите свойства формы:

```
BorderIcons=[biSystemMenu,biMinimize]
BorderStyle=bsSingle
Caption='  Раскраска'
Name=frmFill
Position=poDesktopCenter
Width=576, Height=504.
```

Рисовать мы будем по-прежнему на канве компонента **TImage**:

```
Autosize=True
Height=477
Left=0
Name=Listok
Width= 530
Top=0.
```

Кнопку для выхода из программы возьмём из предыдущего проекта:

```
//ЗАКРЫТЬ ПРОГРАММУ
procedure TfrmFill.sbtExitClick(Sender: TObject);
begin
  close
end;
```

Действие кнопки **sbtClear** тоже ничем не отличается:

```
//ОЧИСТИТЬ ПОЛЕ
procedure TfrmFill.sbtClearClick(Sender: TObject);
begin
  Listok.Canvas.Brush.Color := clWhite;
  Listok.Canvas.FillRect(Listok.ClientRect);
end;
```

Так как закрашивать мы будем уже готовые картинки, то нужно позаботиться об их загрузке. Отвечать за это будет кнопка, которую нужно «посадить» на форму в правом нижем углу. Правда, одной кнопкой тут не обойдёшься, потребуется ещё диалоговая форма для открытия графических файлов **OpenPictureDialog1** (компонент *TOpenPictureDialog* 

со вкладки **Dialogs** *Панели компонентов*). При нажатии кнопки **sbtLoadPicture** откроется окно со списком графических файлов, из которого, как обычно, нужно выбрать нужный:

```
//ЗАГРУЗИТЬ КАРТИНКУ
procedure TfrmFill.sbtLoadPictureClick(Sender: TObject);
begin
  if OpenPictureDialog1.Execute then
    Listok.Picture.LoadFromFile(OpenPictureDialog1.FileName);
end;
```

После этого появится картинка, которую можно раскрасить во все цвета радуги. Для выбора цвета кисти можно воспользоваться тем же компонентом **TColorDialog**, что и раньше, да не очень-то удобно всякий раз открывать диалоговую форму. Лучше установить на форме другой компонент – **TColorGrid** 

со страницы **Samples**. Он может иметь разную форму, но будет сподручнее, если расположить его вертикально:

BackgroundEnabled=False (нам нужен только один цвет - **Foreground**)
GridOrdering=go1x16
Height=336
Left=536
Top=88
Width=32.

Щёлкнув на ячейке с нужным цветом, мы установим цвет заливки, то есть кисти:

```
//ВЫБРАТЬ ЦВЕТ ДЛЯ ЗАКРАШИВАНИЯ
procedure TfrmFill.ColorGrid1Change(Sender: TObject);
begin
  Listok.Canvas.Brush.Color := ColorGrid1.ForegroundColor
end;
```

Эту же строку надо добавить в конец процедуры **sbtLoadPictureClick**, чтобы при загрузке новой картинки сохранялся цвет кисти:

```
  //установить цвет для закрашивания:
  Listok.Canvas.Brush.Color := ColorGrid1.ForegroundColor
```

Так как рисовать «позволено» только на картинке формата *BMP*, то нужно ограничить выбор типов графических файлов при загрузке. Сделать это можно, например, при создании формы:

```
//НАЧАЛЬНЫЕ УСТАНОВКИ
procedure TfrmFill.FormShow(Sender: TObject);
begin
  //фон - белый:
  Listok.Canvas.FillRect(Listok.ClientRect);
  //кисть - чёрная:
  Listok.Canvas.brush.color:= clBlack;
  //загружать будем только картинки BMP:
  OpenPictureDialog1.DefaultExt := GraphicExtension(TBitmap);
  OpenPictureDialog1.Filter := GraphicFilter(TBitmap);
end;
```

Заодно здесь же подготовимся к работе: окрасим холст в белый цвет, а цвет кисти по умолчанию должен быть чёрным.

Не лишним будет установить подходящую форму курсоров на канве (банка с краской) и на «палитре» (пипетка). Удобнее всего эти курсоры поместить в файл **MyCursors.res**. Он будет загружен при запуске приложения, если после строчки `{$R *.DFM}` написать:

```
{$R MyCursors.res}
```

После раздела **uses** нужно задать индексы наших курсоров (это могут быть любые целые положительные числа):

```
//загружаемые курсоры:
const
  crFill= 14;
  crEyedrop= 15;
```

Подключаем их в той же процедуре **FormShow**:

```
  //курсоры:
  Screen.Cursors[crFill] := LoadCursor(hinstance,'FILL');
  Screen.Cursors[crEyedrop] := LoadCursor(hinstance,'EYEDROP');
  Listok.Cursor := crFill;
  ColorGrid1.Cursor:= crEyedrop;
```



'FILL' и **'EYEDROP'** – названия курсоров в файле ресурсов **MyCursors.res**.

На сладкое осталась самая интересная процедура:

```
//ЗАКРАШИВАЕМ ОБЛАСТЬ ВЫБРАННЫМ ЦВЕТОМ
procedure TfrmFill.ListokMouseDown(Sender: TObject;
  Button: TMouseButton; Shift: TShiftState; X, Y: Integer);
var
  clr: TColor;
begin
  //цвет точки под мышкой:
  clr:= Listok.Canvas.Pixels [X, Y];
  Listok.Canvas.FloodFill(X, Y, clr, fsSurface);
end;,
```

где и появляется во всей красе метод **FloodFill**. Запустите программу, загрузите рисунок, выберите подходящий цвет, подведите курсор к нужной области рисунка и щёлкните мышкой. Так как выбран метод закрашивания **fsSurface**, то в цвет кисти окрасятся все точки выбранной области, которые имеют тот же цвет, что и точка, в которой находилось при щелчке перекрестие курсора. Всё просто и удобно, но если вы промахнётесь и попадёте в контурную линию чёрного цвета, то безвозвратно испортите картинку: весь контур будет закрашен текущим цветом. На этот случай имеется второй режим закрашивания – **fsBorder**. Чтобы им воспользоваться, немного подправьте процедуру **ListokMouseDown**:

```
begin
  //цвет точки под мышкой:
  {clr:= Listok.Canvas.Pixels [X, Y];
  Listok.Canvas.FloodFill(X, Y, clr, fsSurface);}
  Listok.Canvas.FloodFill(X, Y, clBlack, fsBorder);
end;
```

На этот раз закрашивание будет таким же весёлым, но более безопасным: контур даже при неудачном «выстреле» останется без изменений (побочное действие этого способа – вам не удастся перекрасить ни одну чёрную область, хотя этого и делать-то ни к чему).

А вот и обещанная заливная рыба, роль которой с успехом исполнит курьёзная рыбка – морской конёк (Рис. П6.13).

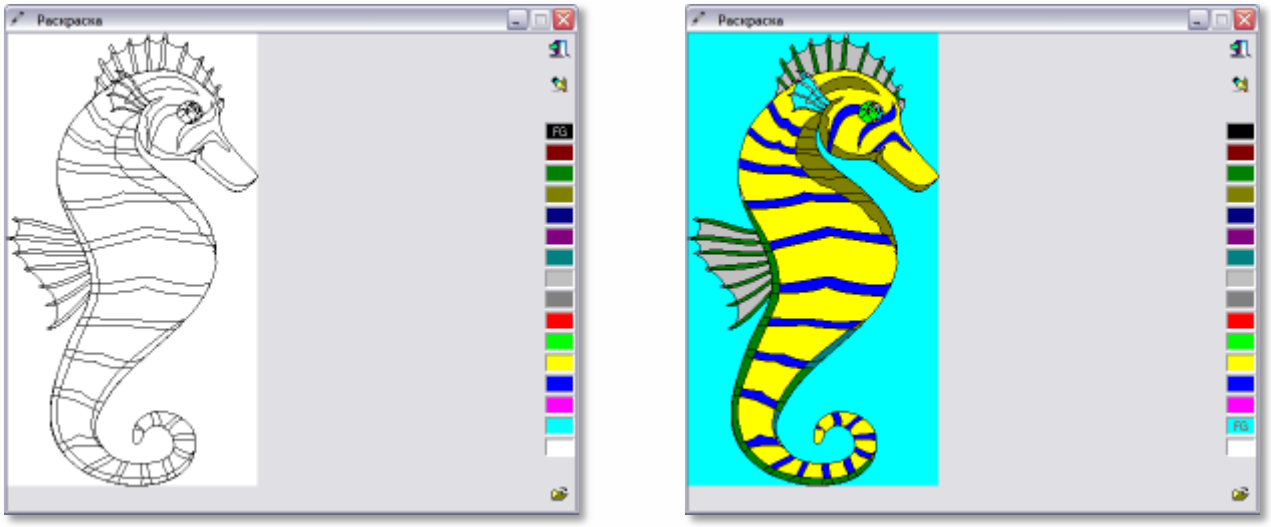


Рис. П6.13. На первом рисунке – заготовка, на втором – раскрашенная картинка. Как говорится, почувствуйте разницу!



Исходный код программы находится в папке **Fill**.

Чините перья, или класс *TPen*

Откройте новый проект и немного поработайте над **формой**:

```
Name=PaintForm,
BorderIcons =[biSystemMenu,biMinimize],
BorderStyle=bsSingle,
Caption=' Pen & Brush',
Height=448, Width=525,
Position=poDesktopCenter.
```

Начинаем наполнять форму содержанием, то есть компонентами. Для начала поставим уже хорошо знакомый нам компонент для вывода подсказок **StatusBar1**:

```
Align=alBottom (пусть будет у нижней границы формы, там ему самое
место),
Autohint=True.
```

В *Редакторе панелей* создайте **StatusPanel** с **Width**=200. Для рисования нам потребуется компонент **TImage**:

```
Name=PaintBox,
Cursor=crUpArrow,
```

```
Left=4
Top=36,
Height=365
Width=509.
```

Подготовимся к рисованию. Дважды щёлкните на форме и заполните заготовку процедуры:

```
//СОЗДАТЬ ФОРМУ
procedure TPaintForm.FormCreate(Sender: TObject);
begin
  //фон листа - белый:
  PaintBox.Canvas.FillRect(ClientRect);
end;
```

Чтобы время от времени стирать результаты своей деятельности, потребуется кнопка **sbtClear** типа *TSpeedButton*. Эта и все остальные кнопки будут плоскими (**Flat**=True), размером 32 x 32 пикселя. Задача у нашей первой кнопки самая простая:

```
//ОЧИСТИТЬ ПОЛЕ
procedure TPaintForm.sbtClearClick(Sender: TObject);
begin
  PaintBox.Canvas.Brush.Color:= clWhite;
  PaintBox.Canvas.Brush.Style:= bsSolid;
  PaintBox.Canvas.FillRect(ClientRect);
end;
```

В джентльменский набор входит также кнопка, закрывающая программу, - **sbtExit**:

```
//ЗАКРЫТЬ ПРОГРАММУ
procedure TPaintForm.sbtExitClick(Sender: TObject);
begin
  Close
end;
```

А вот теперь нужно обзавестись целой обоймой *кнопок* – для рисования различных фигур:

- **sbtLine** (прямых линий),
- **sbtSquare** (прямоугольников),
- **sbtCircle** (эллипсов),
- **sbtRoundRect** (прямоугольников с закруглёнными углами),
- **sbtArc** (дуг),
- **sbtChord** (сегментов эллипса),
- **sbtPie** (секторов эллипса),
- **sbtBezier** (кривых Безье),

- **sbtPolygon** (многоугольников),
- **sbtPolyline** (ломаных линий).

Одновременно в нажатом положении может находиться одна и только одна из этих кнопок, поэтому свойство **AllowAllUp** у этих кнопок должно быть установлено в *False* (все кнопки не могут быть одновременно отжаты), **Cursor**=crHandPoint, у кнопки **sbtLine** свойство **Down**=True (она нажата – чертим линии), **GroupIndex**=1 у всех кнопок (тогда при включении одной кнопки отжимается нажатая раньше). Картинки **Glyph** подберите сами, подсказки – тоже, свойство **Name** установите по назначению кнопок.

В итоге после запуска программы перед вашими очами должна открыться такая очаровательная форма (Рис. П6.14).

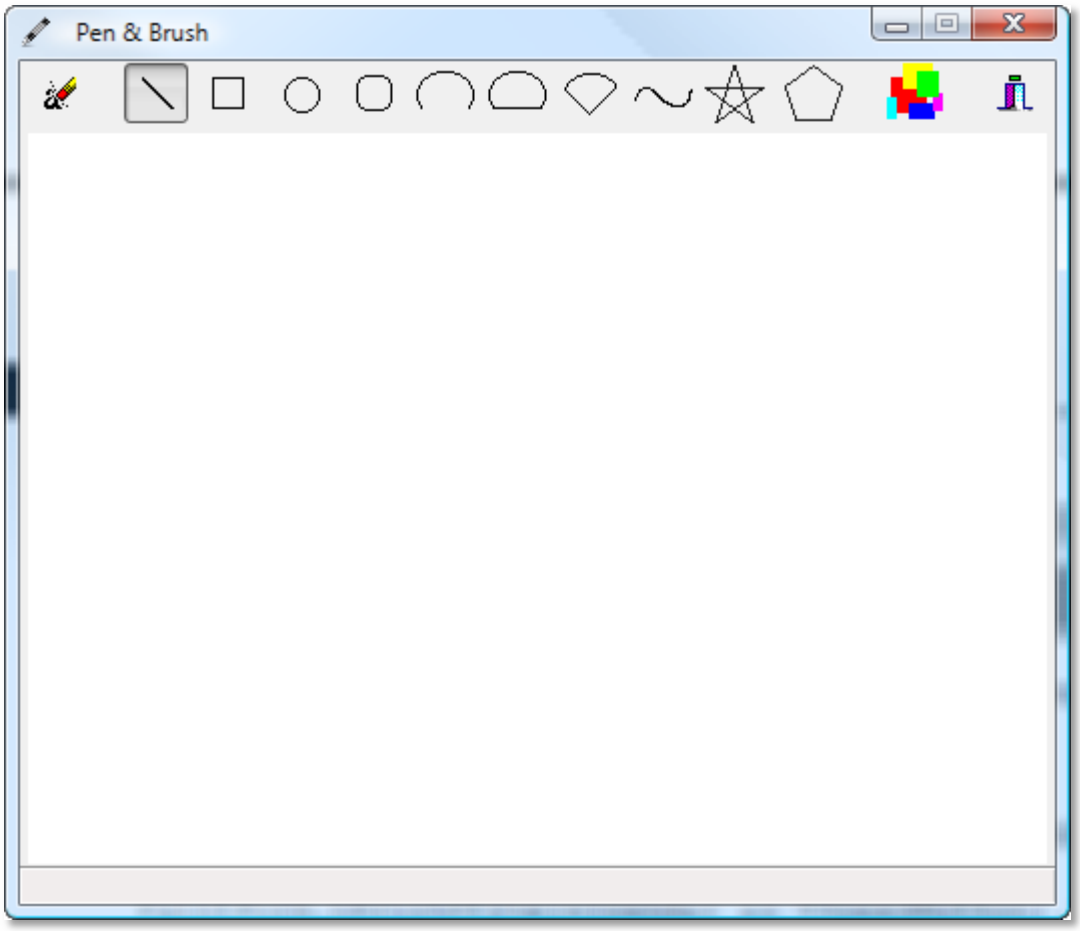


Рис. П6.14. Перьевой интерфейс

В переменной **nFigure** мы будем хранить номер выбранной фигуры, определяемой нажатой кнопкой. Объявим её в разделе *private* типа формы:

```
    nFigure: integer;
```

Чтобы различать, на какой кнопке был щелчок, свойству **tag** кнопок придадим значения от **0** до **9**, соответственно. Не забудьте также связать событие **OnClick** кнопок 1-9 с процедурой **sbtLineClick** (это делается на вкладке **Events** *Инспектора объектов*).

```
//ВЫБРАТЬ ФИГУРУ ДЛЯ РИСОВАНИЯ
procedure TPaintForm.sbtLineClick(Sender: TObject);
begin
  nFigure:=(Sender as TSpeedButton).tag;
end;
```

Далее действуем, как и в предыдущей программе: переменной **Drawing** - объявите её там же, где и **nFigure**:

```
Drawing: Boolean;
```

- присваиваем значение *True*. В переменных **Origin** и **MovePt** хранятся координаты начальной и текущей точки - добавьте их объявление:

```
Origin, MovePt: TPoint;
```

В этом тоже нет ничего нового:

```
//НАЖАТЬ КНОПКУ МЫШИ - НАЧИНАЕМ РИСОВАНИЕ
procedure TPaintForm.PaintBoxMouseDown(Sender: TObject; Button:
TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  Drawing := True;
  //PaintBox.Canvas.MoveTo(X, Y);
  Origin := Point(X, Y);
  MovePt := Origin;
end;
```

При перемещении мыши мы дважды выводим фигуру: сначала стираем её в прежнем положении (для этого используем растровую операцию **pmNotXor**), а затем рисуем в новом, определяемом положением курсора.

```
//РИСУЕМ ВЫБРАННУЮ ФИГУРУ
procedure TPaintForm.PaintBoxMouseMove(Sender: TObject; Shift:
TShiftState; X,
  Y: Integer);
begin
  if Drawing then
  begin
    DrawShape(Origin, MovePt, pmNotXor);
    MovePt := Point(X, Y);
    DrawShape(Origin, MovePt, pmNotXor);
  end;
```

```
end;
```

При отпускании кнопки мыши рисование заканчивается:

```
//ЗАКАНЧИВАЕМ РИСОВАНИЕ
procedure TPaintForm.PaintBoxMouseUp(Sender: TObject; Button:
TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  if Drawing then
  begin
    //выводим фигуру в конечном положении:
    DrawShape(Origin, Point(X, Y), pmCopy);
    Drawing := False;
  end;
end;
```

Растровая операция **pmCopy** рисует фигуру поверх уже имеющихся на «листе».

Осталась ещё одна процедура, в которой и происходит рисование всех фигур – **DrawShape**. Объявите её после всех переменных в разделе **private**:

```
  procedure DrawShape(TopLeft, BottomRight: TPoint; AMode:
TPenMode);
```

Точки **TopLeft** и **BottomRight** находятся в левом верхнем и нижнем правом углу прямоугольника, описывающего рисуемую фигуру, **AMode** – код растровой операции.

Выглядеть она будет так:

```
//СТИРАЕМ СТАРУЮ ФИГУРУ, РИСУЕМ НОВУЮ
procedure TPaintForm.DrawShape(TopLeft, BottomRight: TPoint;
AMode: TPenMode);
var
  ptPoly: array[1..5] of TPoint;
  ptPent: array[1..6] of TPoint;
  xMid, yMid: Integer;
  dxyRadius: Integer;
  i: Integer;
begin
  with PaintBox.Canvas do
  begin
    //Brush.Color := nBrushColor;
    //Pen.Color := nPenColor;
    Pen.Mode := AMode;
    //pen.width:= nPenWidth;
    //Brush.Style := TBrushStyle(nBrushStyle);
```

```
    //Pen.Style := TPenStyle(nPenStyle);
    case TFigure(nFigure) of
        dtLine:
        begin
          MoveTo(TopLeft.X, TopLeft.Y);
          LineTo(BottomRight.X, BottomRight.Y);
        end;
        dtRectangle: Rectangle(TopLeft.X, TopLeft.Y, Bottom-
Right.X, BottomRight.Y);
        dtEllipse: Ellipse(Topleft.X, TopLeft.Y, BottomRight.X,
          BottomRight.Y);
        dtRoundRect: RoundRect(TopLeft.X, TopLeft.Y, Bottom-
Right.X,
          BottomRight.Y, (TopLeft.X - BottomRight.X) div 2,
          (TopLeft.Y - BottomRight.Y) div 2);
        dtArc: Arc(TopLeft.X, TopLeft.Y, BottomRight.X, Bottom-
Right.Y,
           BottomRight.X, TopLeft.Y,TopLeft.X,TopLeft.Y);
        dtChord: Chord(TopLeft.X, TopLeft.Y, BottomRight.X, Bot-
tomRight.Y,
           BottomRight.X, TopLeft.Y,TopLeft.X,TopLeft.Y);
        dtPie: Pie(TopLeft.X, TopLeft.Y, BottomRight.X, Bottom-
Right.Y,
           BottomRight.X, TopLeft.Y,TopLeft.X,TopLeft.Y);
        dtPolyBezier: PolyBezier([point(TopLeft.X, TopLeft.Y),
                                  point(BottomRight.X, Bottom-
Right.Y),
                                  point(width div 2, height div
2),
                                  point(width div 2 - 10, height
div 2 - 10)]);
        dtPolygon: //рисуем звёзды
          begin
            {xMid := TopLeft.X+(BottomRight.X-TopLeft.X) div 2;
            yMid := TopLeft.Y+(BottomRight.Y-TopLeft.Y) div 2;
            dxyRadius := (BottomRight.X-TopLeft.X) div 2;}
            xMid := origin.x;
            yMid := origin.y;
            //dxyRadius := abs (BottomRight.X-origin.x)div 2;
            dxyRadius := (BottomRight.X-origin.x)div 2;
            if dxyRadius=0 then exit;
            //вычисляем точки:
            For i := 1 To 5 do begin
              ptPoly[i].x := xMid + round((Cos(i * radStar- pi /
10) * dxyRadius));
              ptPoly[i].y := yMid+ round((Sin(i * radStar- pi /
10) * dxyRadius));
            end;
            Polygon(ptPoly);
          end;
        dtPolyline: //рисуем пятиугольники
          begin
            xMid := origin.x;
```

```
            yMid := origin.y;
            dxyRadius := (BottomRight.X-xMid)div 2;
            if dxyRadius=0 then exit;
            //вычисляем точки:
            For i := 1 To 5 do begin
              ptPent[i].x := xMid + round((Cos(i * radPentagon- pi
/ 10) * dxyRadius));
              ptPent[i].y := yMid+ round((Sin(i * radPentagon- pi
/ 10) * dxyRadius));
            end;
            //соединить начальную и конечную точки
            ptPent[6]:= ptPent[1];
            //рисуем пятиугольник
            Polyline(ptPent);
          end;
    end;
  end
end;
```

Наворочено изрядно, так что начнём разгребать. Для удобства кодирования, чтобы не запутаться в номерах фигур, объявим перед типом формы, в том же разделе *type* новый тип – инструментов для рисования, соответствующих фигурам:

```
  TFigure = (dtLine, dtRectangle, dtEllipse, dtRoundRect, dtArc,
    dtChord, dtPie, dtPolyBezier, dtPolygon, dtPolyline);
  TPaintForm = class(TForm)
…
```

Итак, нажав кнопку мыши, мы устанавливаем начальную точку **Origin** для рисования, но пока никаких изменений на экране не будет. Не отпуская кнопку, перемещаем курсор – и тут начинается: сначала выводится маленькая фигурка, затем она стирается и перерисовывается уже с новыми размерами. Вы можете двигать мышку как угодно – фигурка будет послушно перерисовываться на экране. Если вы не переключали кнопки, то будет нарисована прямая линия с помощью методов канвы **MoveTo(X,Y)** и **LineTo(X,Y).** С ними мы уже хорошо знакомы, так что с лёгким сердцем можно переключиться на другую фигуру. У нас это прямоугольник. Метод **Rectangle(X1, Y1, X2, Y2)** тоже не новинка. На третьей кнопке сидит также известный нам по прежним программам метод **Ellipse(X1, Y1, X2, Y2)**. Ничем не сможет удивить нас и метод **RoundRect(X1, Y1, X2, Y2, X3, Y3)**.

И только на четвёртой кнопке мы обнаружим свежий метод

Arc(X1, Y1, X2, Y2, X3, Y3, X4, Y4: Integer) –

он рисует дугу эллипса. Сам эллипс определяется прямоугольником (**X1, Y1, X2, Y2**), описывающим его. Дуга рисуется из начальной точки в конечную против часовой стрелки. Начальная точка является пересечением эллипса и линии, проведённой из центра эллипса через точку (**X3, Y3**). Конечная точка является пересечением эллипса с прямой, проведённой из центра эллипса через точку (**X4, Y4**).

На следующей кнопке мы найдём метод

```
Chord(X1, Y1, X2, Y2, X3, Y3, X4, Y4: Integer).
```

Он рисует сегмент эллипса и закрашивает его текущей кистью. Все точки и другие характеристики фигуры совпадают с оными в методе **Arc**.

Ещё дальше расположился метод

```
Pie(X1, Y1, X2, Y2, X3, Y3, X4, Y4: Longint),
```

который вычерчивает сегмент эллипса, вписанного в прямоугольник (**X1, Y1, X2, Y2**). Стороны сектора лежат на прямых, проходящих из центра эллипса через точки (**X3, Y3**) и (**X4, Y4**).

На восьмой кнопке находится метод

```
PolyBezier(const Points: array of TPoint),
```

с помощью которого можно строить удивительные кривые Безье. Правда, в нашем исполнении кривая Безье смотрится неубедительно, поэтому мы сделаем для неё исключение в виде небольшой программы. Это будет «завтра», а пока нам остаётся довольствоваться такой «загогулиной». Массив, задающий кривую, должен содержать не менее четырёх точек: первая и четвёртая точки являются начальной и конечной точками кривой, а вторая и третья – контрольные точки, определяющие форму кривой. Кривая Безье может содержать несколько сегментов, тогда последняя точка одного сегмента служит начальной для второго, затем в массиве следуют две контрольные точки и конечная точка второго сегмента, и так до бесконечности.

Метод

```
Polygon(Points: array of TPoint),
```

кнопка 9, вычерчивает многоугольник, используя массив точек. Прямыми линиями последовательно соединяются две соседние точки, в том числе

начальная и конечная. Образовавшаяся замкнутая фигура закрашивается текущей кистью. В данной программе роль многоугольника исполнила звезда о пяти лучах. Сначала вычисляются и заносятся в массив **ptPoly** координаты звезды, а затем вызывается метод **Polygon(ptPoly)** с этим массивом в качестве аргумента. *Константа*

```
radStar: Double = 144 * PI / 180;
```

задаёт угол в радианах между соседними точками звезды на описывающей окружности. Для звезды этот угол равен 144 градуса. Точки **xMid** и **yMid** определяют центр окружности (в этой точке была нажата кнопка мыши).

И последний из рассматриваемых здесь методов канвы –

Polyline(Points: array of TPoint).

В отличие от предыдущего метода ломаная линия автоматически не замыкается и фигура, даже и замкнутая, не закрашивается. Если массив **Points** содержит только 2 точки, то будет нарисована прямая линия. Такие же фигуры можно построить, используя методы **MoveTo** и **LineTo**, но в этом случае изменится значение **PenPos**, а при вызове **Polyline** – нет. В этом приложении вычерчивается правильный пятиугольник, но для задания его координат потребуется **6** точек (последняя точка совпадает с первой, чтобы замкнуть фигуру) и, соответственно, другой *массив*:

```
ptPent: array[1..6] of TPoint;
```

для их хранения. Координаты точек вычисляются аналогично звезде, но угол между соседними точками пятиугольника на окружности не 144, а 72 градуса, что и зафиксировано в *константе*:

```
radPentagon: Double = 72 * PI / 180;
```

Пора вспомнить о таком свойстве пера, как **Width**, - оно определяет толщину пера в пикселях. По умолчанию создаётся перо толщиной в 1 пиксель, поэтому все наши фигуры до сих пор рисовались именно такими линиями. Мы сами должны позаботиться о том, чтобы изменить толщину пера, то есть значение **Width**.

Заведём для пера отдельную форму, которая будет служить плавающей панелью. Нажмите в *Менеджере проекта* правую кнопку мыши на названии исполняемого файла и в раскрывшемся меню щёлкните на строчке **Add New > VCL Form** (Рис. П6.15).

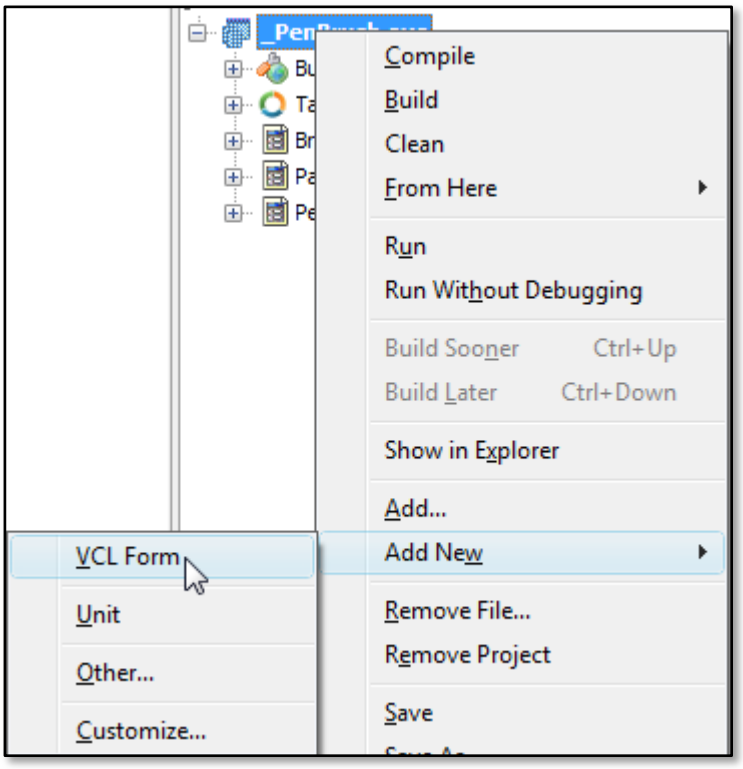


Рис. П6.15. Панельная форма

Придайте появившейся форме надлежащий вид:

Name=**frmPen**,
BorderIcons – всё установите в **False** (кнопки в заголовке нам не потребуются),
BorderStyle=bsSingle (изменять размеры формы нам также незачем),
Caption=' Перо',
FormStyle=fsStayOnTop (форма всегда должна располагаться поверх других форм),
Height=60,
Width=440.

Запишите модуль пера под именем **PenUnit**.

Чтобы использовать эту форму, в главной программе после {$R *.DFM} добавьте:

```
uses
  PenUnit;
```

А также заведите новую *процедуру*:

```
//ПОКАЗАТЬ ВСЕ ФОРМЫ
procedure TPaintForm.FormShow(Sender: TObject);
begin
  frmPen.show;
```

```
end;
```

Теперь при запуске программы форма пера появится – да и только. Для изменения толщины пера подойдёт компонент движок **TrackBar1** типа *TTrackBar*:

Height=25,
Left=288,
Top=4,
Max=10,
Min=1 (пределы изменения толщины линий, вы можете установить толщину пера и меньше 1, но она всё равно будет равна 1 пикселю),
Position=1 (толщина линий по умолчанию).

Вся хитрая механика по изменению толщины линий скрывается в *процедуре*:

```
//ИЗМЕНИТЬ ТОЛЩИНУ ПЕРА
procedure TfrmPen.TrackBar1Change(Sender: TObject);
begin
  PaintForm.nPenWidth:= TrackBar1.Position
end;
```

Её заготовка откроется, если дважды щёлкнуть на движке. Переменную, хранящую толщину пера, нужно объявить в главной программе, в разделе **public** типа формы:

```
  public
    { Public declarations }
    nPenWidth: integer;
```

В процедуре **TPaintForm.DrawShape** после `Pen.Mode := AMode;` допишите строку:

```
pen.width:= nPenWidth;
```

в которой будет устанавливаться толщина пера. Запустите программу – сейчас вы можете наслаждаться изменением толщины линий. Вот только линии по-прежнему останутся скучно-чёрными, а ведь у пера есть колоритное свойство **Color**, устанавливающее его цвет. С этим мы уже знакомы, поэтому быстро установим на форме компонент **ColorDialog1,** кнопку **sbtColor** и метку **Label1** (**Caption**='). Продолжаем идти проторённым путём:

```
//ИЗМЕНИТЬ ЦВЕТ ПЕРА
procedure TfrmPen.sbtColorClick(Sender: TObject);
begin
  if ColorDialog1.execute then
  begin
```

```
    PaintForm.nPenColor:= ColorDialog1.Color;
    label1.Color:= ColorDialog1.Color
  end
end;
```

В **PaintUnit** после `nPenWidth: integer;` объявите *переменную*:

```
nPenColor: TColor;
```

в которой будет храниться выбранный цвет пера. Чтобы задействовать установку, в процедуре **TPaintForm.DrawShape** после строки `Pen.width:= nPenWidth;` напишем:

```
Pen.Color := nPenColor;.
```

Это небольшое усовершенствование программы позволяет выбирать для линий любой цвет.

Впрочем, с этими свойствами пера мы уже встречались много раз. Но у пера имеется и «новое» свойство – **Style**, которое определяет стиль линии, рисуемой пером. По умолчанию создаётся непрерывное перо (**psSolid**), поэтому все линии были сплошными. Вы можете также рисовать штриховые линии (**psDash**), пунктирные (**psDot**), штрихпунктирные (**psDashDot**), штрихпунктирные с двумя точками между штрихами (**psDashDotDot**). Если выбрать стиль **psClear**, то линии вообще не будет. Стиль **psInsideFrame** аналогичен **psSolid**, но позволяет выбирать большее количество цветов пера.

Для выбора любого из этих стилей потребуется 7 кнопок: **sbtPSolid**, **sbtPDash**, **sdtPDot**, **sbtPDashDot**, **sbtPDashDotDot**, **sbtPClear**, **sbtPInsideFrame**. Свойство **GroupIndex** у всех кнопок должно быть одинаково и отлично от нуля, например 1, чтобы одновременно в нажатом положении фиксировалась только одна из этих кнопок. Щёлкните на первой кнопке и напишите:

```
//ИЗМЕНИТЬ СТИЛЬ ПЕРА
procedure TfrmPen.sbtPSolidClick(Sender: TObject);
begin
  PaintForm.nPenStyle:= (sender as TSpeedButton).tag;
end;
```

В этой процедуре переменной **nPenStyle** (объявите **nPenStyle**: integer; там же, где и nPenColor) будет задаваться значение стиля, равное свойству **tag** нажатой кнопки (поэтому в *Инспекторе объектов* запишите числа 0-6 в поле свойства **tag** этих кнопок, соответственно). В процедуре **TPaintForm.DrawShape** допишите строку:

```
Pen.Style := TPenStyle(nPenStyle);
```

Этого достаточно для того, чтобы вы могли рисовать фигуры линиями любого из доступных стилей. Но есть и ложка дёгтя в этой бочке «стильного» мёда: если толщина линий больше 1 пикселя, то все линии будут сплошными. Это ограничение **самой** ОС *Windows*, и тут уж ничего не поделаешь!

В итоге должна получиться вот такая форма для работы с пером (Рис. П6.16).

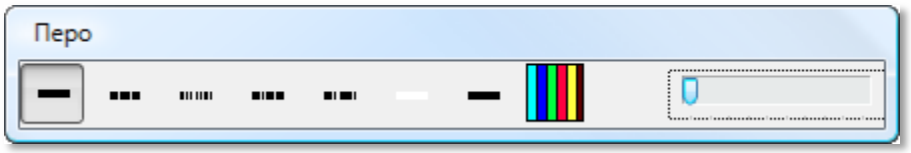


Рис. П6.16. Перьевая форма

У пера есть ещё одно интересное свойство – **Mode**, оно определяет, как будет сочетаться цвет пера с цветом канвы. В этой программе уже использовались две из возможных растровых операций: **pmCopy** - линия будет иметь выбранный цвет, **pmNotXor** – комбинация цвета пера и цвета фона канвы, очень полезная операция при рисовании перемещающихся фигур, сначала фигура будет нарисована, а затем полностью стёрта, так что изображение под фигурой будет восстановлено. Если **Mode** = pmBlack, то линии всегда будут чёрными, независимо от выбранного цвета пера. При Mode = **pmWhite** линии всегда будут белыми. Режим **pmNop** не изменяет цвет линий. Если вы хотите посмотреть действие каждой растровой операции, то просто замените в процедуре **TPaintForm.PaintBoxMouseUp** строку `DrawShape(Origin, Point(X, Y), pmCopy);`, например, так:

```
 DrawShape(Origin, Point(X, Y), pmNop);
```

Запустите программу и рисуйте фигуры – это лучший способ разобраться в хитросплетениях растровых операций. Можно сделать этот процесс ещё более наглядным. Возьмём из программы **methods** кнопку **sbtRects** и немного подправим её обязанности:

```
//РИСУЕМ ПРЯМОУГОЛЬНИКИ
procedure TPaintForm.sbtRectsClick(Sender: TObject);
Var
  x1, y1, x2, y2 :integer;
  clr: TColor;
  i :integer;
begin
```

```
  with PaintBox.Canvas do begin
    randomize;
    Pen.Width:= 2;
    Pen.Mode:=pmCopy;
    Brush.Style:= bsSolid;
    for i:= 1 to 50 do
    begin
      //выбираем случайный цвет линии
      clr:= rgb (random(256),random(256),random(256));
      Pen.Color:=clr;
      //выбираем случайный цвет кисти:
      clr:= rgb (random(256),random(256),random(256));
      Brush.Color:=clr;
      //координаты верхнего левого угла прямоугольника:
      x1:= random(ClientWidth-1); y1:= random(ClientHeight-1);
      //координаты верхнего левого угла прямоугольника:
      x2:= random(ClientWidth-1); y2:= random(ClientHeight-1);
      //чертим прямоугольник с границей и заливкой:
      Rectangle(x1,y1,x2,y2);
    end;
  end
end;
```

После запуска программы вы можете «засеять» канву разноцветными прямоугольниками, нажав эту кнопку, а на их фоне будет легче наблюдать взаимодействие цветов при той или иной растровой операции.

- Операция **pmNot** инвертирует (изменяет на противоположный) цвета канвы там, где проходит линия, а **pmNotCopy** инвертирует цвет самого пера.

- **pmMergePenNot** – комбинация цвета пера и инвертированных цветов канвы.

- **pmMaskPenNot** – комбинация цветов, общих для пера и инвертированных цветов фона канвы.

- **pmMergeNotPen** – комбинация цветов фона канвы и инвертированного цвета пера.

- **pmMaskNotPen** - комбинация цветов, общих для фона канвы и инвертированного пера.

- **pmMerge** - комбинация цвета пера и цветов канвы.

- **pmNotMerge** - инвертированный цвет *pmMerge*: комбинация цвета пера и цвета фона канвы.

- **pmMask** – комбинация цветов, общих для пера и цветов канвы.

- **pmNotMask** - инвертированный цвет *pmMask*: комбинация цветов, общих для пера и фона канвы.

- **pmXor** – комбинация цветов пера и фона канвы. Рассмотренная ранее операция **pmNotXor** даёт инвертированный по отношению к *pmXor* цвет.

Прежде чем продолжить наращивать мощь программы **PenBrush**, вспомним о долгах своих и закончим «разборки» с кривыми Безье. Вы вволю сможете поиграть одной подобной кривулькой в следующей программе:

```
unit Unit1;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
Dialogs,
  ExtCtrls, Buttons;

type
  TForm1 = class(TForm)
    PaintBox: TImage;
    sbtExit: TSpeedButton;
    sbtClear: TSpeedButton;
    procedure sbtExitClick(Sender: TObject);
    procedure sbtClearClick(Sender: TObject);
    procedure FormCreate(Sender: TObject);
    procedure PaintBoxMouseDown(Sender: TObject; Button: TMouse-
Button;
      Shift: TShiftState; X, Y: Integer);
    procedure PaintBoxMouseMove(Sender: TObject; Shift:
TShiftState; X,
      Y: Integer);
    procedure PaintBoxMouseUp(Sender: TObject; Button: TMouseBut-
ton;
      Shift: TShiftState; X, Y: Integer);
  private
    { Private declarations }
    apt: array[0..3] of TPoint;
    Btn: TMouseButton;
    procedure DrawBezier;
    procedure MoveBezier(Button: TMouseButton; X, Y: Integer);
  public
    { Public declarations }
  end;

var
```

```
  Form1: TForm1;

implementation

{$R *.DFM}

//ЗАКРЫТЬ ПРОГРАММУ
procedure TForm1.sbtExitClick(Sender: TObject);
begin
  Close
end;

//ОЧИСТИТЬ ПОЛЕ
procedure TForm1.sbtClearClick(Sender: TObject);
begin
  PaintBox.Canvas.Brush.Color:= clWhite;
  PaintBox.Canvas.Brush.Style:= bsSolid;
  PaintBox.Canvas.FillRect(ClientRect);
end;

//СОЗДАТЬ ФОРМУ
procedure TForm1.FormCreate(Sender: TObject);
begin
  //левая и правая кнопки мыши не нажаты:
  Btn:= mbMiddle;
  //фон листа - белый:
  PaintBox.Canvas.FillRect(ClientRect);
  //цвет линий - красный:
  PaintBox.Canvas.pen.color:= clRed;
  //инициализация массива точек для кривой Безье:
  //начальная точка:
  apt[0].X := PaintBox.Width div 2;
  apt[0].Y := PaintBox.Height div 10;
  //контрольные точки:
  apt[1].X := PaintBox.Width div 4;
  apt[1].Y := PaintBox.Height div 2;
  apt[2].X := PaintBox.Width * 3 div 4;
  apt[2].Y := PaintBox.Height div 2;
  //конечная точка:
  apt[3].X := PaintBox.Width div 2;
  apt[3].Y := PaintBox.Height * 9 div 10;
  //нарисовать кривую:
  DrawBezier;
  //  showmessage(inttostr(PaintBox.Height))
end;

//НАРИСОВАТЬ КРИВУЮ БЕЗЬЕ И УПРАВЛЯЮЩИЕ ЛИНИИ
procedure TForm1.DrawBezier;
begin
  with PaintBox.Canvas do begin
    //нарисовать сплошной линией
    Pen.Style := psSolid;
    //кривую Безье
```

```
    PaintBox.Canvas.PolyBezier(apt);
    //нарисовать пунктирной линией
    Pen.Style := psDot;
    //направляющие линии
    MoveTo (apt[0].X, apt[0].Y);
    LineTo (apt[1].X, apt[1].Y);
    MoveTo (apt[2].X, apt[2].Y);
    LineTo (apt[3].X, apt[3].Y);
  end;
end;

//ИЗМЕНЕНИЕ КРИВОЙ
procedure TForm1.MoveBezier(Button: TMouseButton; X, Y: Integer);
begin
    PaintBox.Canvas.pen.Color :=   PaintBox.Canvas.Brush.Color;
    DrawBezier;
    If Button = mbLeft Then begin
      apt[1].X := x;
      apt[1].Y := y;
    End;
    If Button = mbRight Then begin
      apt[2].X := x;
      apt[2].Y := y
    End;
    PaintBox.Canvas.pen.Color := clRed;
    DrawBezier
End;

//НАЖАТЬ КНОПКУ МЫШИ - НАЧАТЬ ИЗМЕНЕНИЕ КРИВОЙ
//левая кнопка перемещает верхнюю контрольную точку,
//правая - нижнюю
procedure TForm1.PaintBoxMouseDown(Sender: TObject; Button:
TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
   Btn := Button;
   MoveBezier(Btn, X, Y)
end;

//ДВИГАТЬ МЫШЬ - ИЗМЕНЯТЬ КРИВУЮ
procedure TForm1.PaintBoxMouseMove(Sender: TObject; Shift:
TShiftState; X,
  Y: Integer);
begin
  MoveBezier (Btn, X, Y)
end;

//ОТПУСТИТЬ КНОПКУ - ЗАКОНЧИТЬ ИЗМЕНЕНИЕ КРИВОЙ
procedure TForm1.PaintBoxMouseUp(Sender: TObject; Button: TMouse-
Button;
  Shift: TShiftState; X, Y: Integer);
begin
  //левая и правая кнопки мыши не нажаты:
```

```
  Btn:= mbMiddle
end;

end.
```

После того, что с нами уже было, вам не составит труда разобраться в этом нехитром коде. Пожалуй, самое интересное в этой программе – перерисовка линий. Делается это проще, чем с помощью растровой операции **pmNotXor**, - здесь линия сначала рисуется красным цветом, а затем стирается цветом фона канвы (он же – цвет кисти). Конечно, этот способ применим, только когда фон окрашен в один цвет. Интерфейс программы выглядит так (Рис. П6.17).

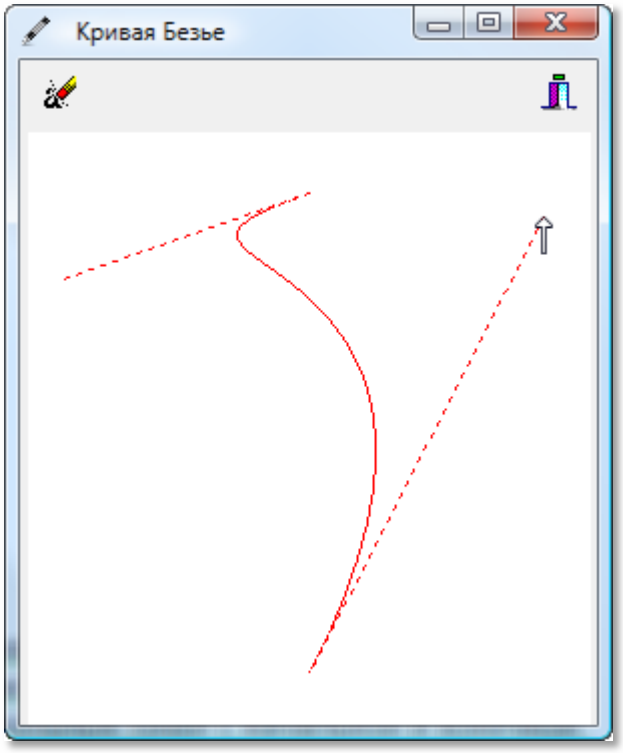


Рис. П6.17. Вот такая загогулина!

Нажав **левую кнопку** мыши, вы можете перемещать верхнюю (первую) контрольную точку, а, нажав **правую**, - нижнюю (вторую) контрольную точку. Так как контрольные линии (они соединяют соответствующие контрольные точки с начальной и конечной точками кривой) являются касательными к кривой Безье в крайних точках, то в зависимости от положения контрольных точек будет изменяться и форма самой кривой. Наблюдать за этими метаморфозами весьма любопытно. Вы легко можете произвольным образом менять начальные значения точек в массиве, чтобы разнообразить вид кривых. Вместо метода **PolyBezier** вы можете использовать аналогичный метод **PolyBezierTo** (const Points: array of TPoint);, кото-

рый отличается от первого только тем, что переносит **PenPos** в конечную точку кривой.



Придуманная Пьером Безье в 60-е годы прошлого века техника рисования сильно облегчает работу с кривыми. В программах на *Delphi* эти преимущества оценить трудно, но в любом **векторном** графическом редакторе (например, *CorelDraw*) вы неминуемо столкнётесь с этими кривыми. Там точки соединения сегментов кривой называются узелками, работе с ними вы можете научиться по любому руководству к соответствующей программе, мы же навсегда прощаемся и с Безье, и с его кривыми потомками, *ибо* - Бендеровское тупиковое словечко! - в наших дальнейших программах такие кривые никогда не встретятся, так что продолжайте эксперименты сами.



Исходный код программы находится в папке **BEZIER**.

Наше вам с кисточкой, или класс *TBrush*

Вторым источником вдохновения для рисования на канве служит **кисть**. Она всегда создаётся вместе с канвой и поначалу выглядит довольно бледно – она заливает всё что можно сплошным белым цветом (**Color** = clWhite, **Style** = bsSolid).

Так как кисть неразлучная подруга пера, создадим по его образу и подобию плавающую панель, которую нагрузим свойствами кисти.

Заведём для кисти отдельную форму, которая будет служить плавающей панелью. Добавьте к проекту новую форму и доработайте её:

```
Name=frmBrush,
BorderIcons = [],
BorderStyle=bsSingle,
Caption=' Кисть',
FormStyle=fsStayOnTop,
Height=60,
Width= 400.
```

Запишите модуль пера под именем **BrushUnit**.

Чтобы использовать эту форму, в главной программе после {$R *.DFM} добавьте:

```
uses
  PenUnit, BrushUnit;
```

А также обновите процедуру:

```
//ПОКАЗАТЬ ВСЕ ФОРМЫ
procedure TPaintForm.FormShow(Sender: TObject);
begin
  frmPen.show;
  frmBrush.show;
end;
```

Теперь при запуске программы на поверхности главной формы будут «плавать» две панели – пера и кисти. Вторая, правда, первобытно чиста, что не способствует работе над собой – начинаем её «грузить».

Главное свойство кисти – её цвет:

```
property Color: TColor;
```

Для доступа к этому свойству действуем, как умеем: устанавливаем на форме точно такой же компонент **TColorDialog**, как и на форме пера. В комплект входит также кнопка **sbtColor** и метка **Label1** (её свойство **Color** установите в clWhite, так как по умолчанию кисть имеет белый цвет). Процедура, обрабатывающая щелчок на кнопке **sbtColor** тоже нам знакома:

```
//ВЫБРАТЬ ЦВЕТ КИСТИ
procedure TfrmBrush.sbtColorClick(Sender: TObject);
begin
  if ColorDialog1.execute then
  begin
    PaintForm.nBrushColor:= ColorDialog1.Color;
    label1.Color:= ColorDialog1.Color
  end
end;
```

Перед строкой {$R *.DFM} в **BrushUnit** добавьте строку:

```
uses PaintUnit;
```

а в процедуре **TPaintForm.DrawShape** – пару строк:

```
//установить цвет кисти:
Brush.Color := nBrushColor;
```

Остаётся объявить *переменную*:

```
nBrushColor: TColor;
```

- и нам дозволено заливать фигуры по полной программе, то есть вгонять их практически в любую краску.

Второе свойство кисти

```
property Style:TBrushStyle;
```

отвечает за рисунок закраски поверхности фигур. Например, установленный по умолчанию стиль **bsSolid** даёт сплошное закрашивание (заливку). А также:

- **bsClear** вообще не закрашивает фигуру.
- **bsHorizontal** закрашивает фигуру горизонтальными линиями,
- **bsVertical** – вертикальными,
- **bsCross** – и теми, и другими, то есть в клеточку.
- **bsBDiagonal** – закрашивает восходящими диагоналями,
- **bsFDiagonal** – нисходящими,
- **bsDiagCross** – опять же теми и другими, то есть в ромбик.

Цвет линий «узора» - тот же **Brush.Color**, а цвет фона канвы не изменяется.

Как и в случае с пером, нам потребуются кнопки для включения нужного стиля закраски. Тут нет ничего нового - ставим **кнопки**:

- **sbtBSolid** (tag=0),
- **sbtBClear** (tag=1),
- **sbtBHoriz** (tag=2),
- **sbtBVert** (tag=3),
- **sbtBFDiag** (tag=4),
- **sbtBBDiag** (tag=5),
- **sbtBCross** (tag=6),
- **sbtBDiagCross** (tag=7).

В итоге должна получиться такая картина (Рис. П6.18).

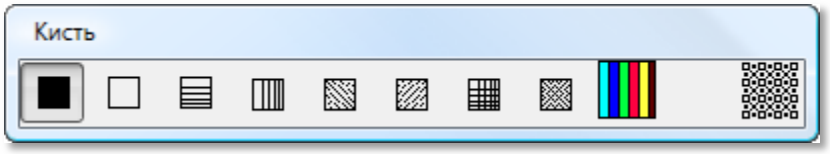


Рис. П6.18. Кистевая форма

Щёлкните на первой кнопке и напишите код, обрабатывающий переключение стильных кнопок:

```
//ЗАДАТЬ СТИЛЬ КИСТИ
procedure TfrmBrush.sbtBSolidClick(Sender: TObject);
begin
  PaintForm.nBrushStyle:=(sender as TSpeedButton).tag;
end;
```

Опишите где следует *переменную*:

```
nBrushStyle: integer;
```

а также добавьте в процедуру **TPaintForm.DrawShape** строчки:

```
//установить стиль кисти:
Brush.Style := TBrushStyle(nBrushStyle);
```

В процедуре **TPaintForm.FormCreate** нужно присвоить начальное значение переменной:

```
nBrushColor:= clWhite;
```

чтобы правильно устанавливался цвет в **TPaintForm.DrawShape** до того, как мы его изменили.

Запустите программу и рисуйте фигуры – они будут закрашиваться по вашему желанию (все замкнутые фигуры, исключая **Polyline**, которая имеет только контур!). Но есть в этом, казалось бы, незамысловатом процессе и свои чудеса: некоторые растровые операции пера **Pen.Mode** влияют на результат закрашивания фигур: узор может быть прозрачным, инвертированным и с изменёнными цветами. Вероятно, это происходит потому, что для закрашивания фигур используется перо со всеми его «наворотами». Любителям колоратурных каламбуров тут есть над чем поразмыслить! Необузданные фантазии в этом вопросе могут дать поразительные результаты.

И последнее интересное свойство кисти –

property Bitmap: TBitmap,

в котором содержится битовая карта, используемая в качестве узора при раскраске. Если это свойство определено, то первые два свойства кисти (цвет и стиль) не действуют. Всё бы хорошо, но для шаблона используется только верхний левый квадратик картинки размером 8 х 8 пикселей. Вы можете взять и монументальное полотно, но это ничего не изменит – будет задействован только его ничтожный фрагмент. Из чего, кстати, следует, что лучше сразу вычерчивать такой шаблон хотя бы в **Image Editor** (Рис. П6.19). Так как шаблон маленький, то очень сложный рисунок заливки с его помощью не получится. Впрочем, при известных художественных способностях можно приготовить и красивую заливку.

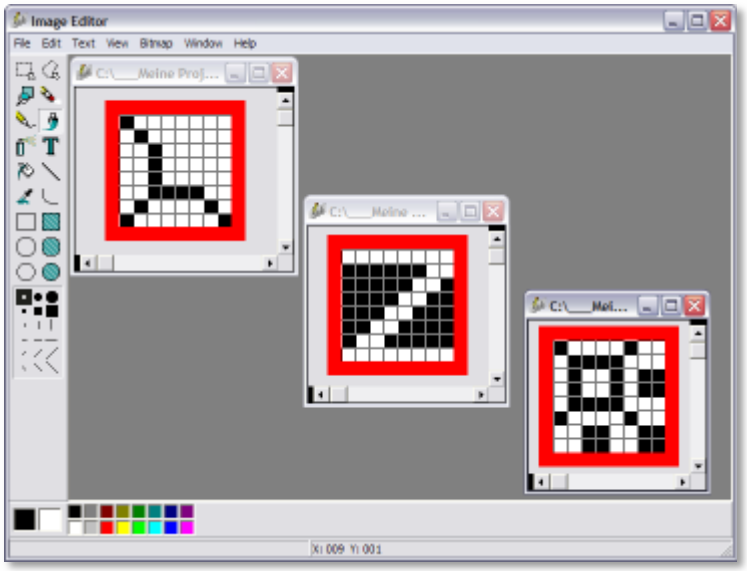


Рис. П6.19. Рисуем шаблон

Начнём внедрять искусство в жизнь. Добавим с правого краю кнопку **sbtBitmap** (**AllowAllUp**=True, **GroupIndex**=2). В нажатом положении она будет сигнализировать о том, что закрашивать фигуру нужно узором. Это действо нужно отразить в процедуре **TPaintForm.DrawShape**:

```
    //установить стиль кисти:
    Brush.Style := TBrushStyle(nBrushStyle);
    //если есть подходящая картинка и кнопка нажата,
    //использовать её для закраски
    if (bmp<> nil) and frmBrush.sbtBitmap.Down then
      Brush.Bitmap:= bmp;
```

Откуда же берётся нужная картинка? – Откуда угодно, но проще её просто загрузить из файла в процедуре **TPaintForm.FormCreate**:

```
  //загрузить рисунок для закрашивания:
  bmp:= TBitmap.Create;
  bmp.LoadFromFile('pattern1.bmp');
  //bmp.LoadFromFile('pattern2.bmp');
```

```
  //bmp.LoadFromFile('pattern3.bmp');
```

Как обычно, нужно объявить свежую *переменную*:

```
    nBrushStyle: integer;
    bmp: Tbitmap;
```

Особо «чистоплотные» могут присовокупить к программе процедуру, выносящую сор из избы:

```
procedure TPaintForm.FormClose(Sender: TObject; var Action:
TCloseAction);
begin
  PaintBox.Canvas.Brush.Bitmap:= nil;
  bmp.Free;
end;
```

На этом мы заканчиваем мучить программу **PenBrush**. Тем же, кто не прочь «поузорничать», позволит проявить себя программа **Pattern**.



Исходный код программы находится в папке **PenBrush**.

Графические шаблоны

Дабы не ходить вокруг да около, сразу представим товар лицом, то есть, по-нашему, интерфейсом (Рис. П6.20).

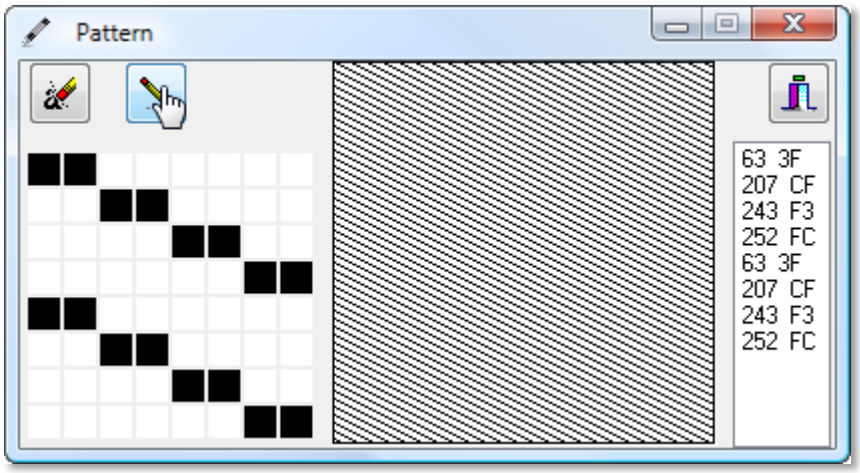


Рис. П6.20. Рисуем шаблон

Здесь вы можете не только придумывать узоры, но и наглядно видеть плоды своего творчества.

Форма:

```
BorderIcons=[biSystemMenu,biMinimize],
BorderStyle= bsSingle,
Caption='  Pattern',
Height= 220, Width= 415.
```

Кнопка **sbtExit** просто закрывает программу:

```
//ЗАКРЫТЬ ПРОГРАММУ
procedure TForm1.sbtExitClick(Sender: TObject);
begin
  close
end;
```

Живописать мы будем на нашем излюбленном компоненте **TDrawGrid**:

```
Name= dgPole,
BorderStyle= bsNone,
ColCount= 8,
RowCount=8,
Cursor= crUpArrow,
DefaultColWidth=16,
DefaultRowHeight=16,
DefaultDrawing=False  (за  прорисовку  ячеек  сетки  будем  отвечать
мы),
FixedCols=0,
FixedRows=0,
GridLineWidth=2,
Height= 144,
Width= 144,
Left=4,
Top=46,
ScrollBars= ssNone.
```

Нам потребуется сетка 8 х 8 клеток, так как узор кисти имеет такие размеры. Каждая клетка сетки – это 1 бит в массиве битов, используемых в качестве узора при закраске поверхностей.
Для визуализации абстрактных размышлений сгодится компонент **Image1** типа *TImage*:

```
Height= 192,
Width= 192,
Left= 156,
Top=0.
```

Данные о цвете клеток в сетке удобно хранить в массиве **masPole**, который нужно объявить перед типом формы:

```
var
```

```
  masPole: array[0..7,0..7]of word;
type
  TForm1 = class(TForm)
```

Введём для удобства работы **константы** для белой и чёрной клеток (объявите их перед *masPole*):

```
const
  NUM_PUSTO = 1;     //белая клетка в masPole
  NUM_KLETKA = 0;//чёрная клетка в masPole
```

Кнопка **sbtClearPole** отвечает за чистоту на поле:

```
//ОЧИСТИТЬ ПОЛЕ
procedure TForm1.sbtClearPoleClick(Sender: TObject);
var
  i,j: integer;
  area: tRect;
begin
  dgPole.Canvas.Brush.Color:= clWhite;
  For i := 0 To 7 do
    for j := 0 to 7 do begin
      area:=form1.dgPole.CellRect(i,j);
      dgPole.Canvas.FillRect(area);
      masPole[i,j]:= NUM_PUSTO;
    end;
  image1.Canvas.Brush.Color:= clWhite;
  image1.Canvas.FillRect(image1.Canvas.ClipRect);
end;
```

После нажатия на эту кнопку все ячейки сетки станут белыми, а все числа в массиве **masPole** будут иметь значение **NUM_PUSTO**, что соответствует цвету ячеек. Прямоугольник с образцом закраски тоже будет белым.

Эту же процедуру необходимо проделать и при запуске программы:

```
//СОЗДАТЬ ФОРМУ
procedure TForm1.FormCreate(Sender: TObject);
begin
  //очистить поле:
  sbtClearPoleClick(self);
end;
```

Начинаем придумывать узоры. Щёлкнем раз – клетка станет чёрной, щёлкнем второй раз – опять белой. Соответственно изменяем значение чисел в массиве **masPole**:

```
//КЛИКНУТЬ НА КЛЕТКЕ ПОЛЯ - перекрасить клетку
```

```
procedure TForm1.dgPoleSelectCell(Sender: TObject; ACol, ARow: In-
teger;
  var CanSelect: Boolean);
var
  area: tRect;
begin
  area:= dgPole.CellRect(ACol, ARow);
  //белую клетку закрасить чёрным:
  if masPole[ACol, ARow]= NUM_PUSTO then
  begin
    dgPole.Canvas.Brush.Color:= clBlack;
    //занести в массив цвет клетки
    masPole[ACol, ARow]:= NUM_KLETKA;
  end
  else //чёрную - сделать белой
  begin
    dgPole.Canvas.Brush.Color:= clWhite;
    //занести в массив цвет клетки
    masPole[ACol, ARow]:= NUM_PUSTO;
  end;
  dgPole.Canvas.FillRect(area);
end;
```

Нам придётся также самим заботиться о том, чтобы рисунок на **dgPole** восстанавливался при его перекрытии другими окнами и в других «несчастных» случаях:

```
//ОТРИСОВАТЬ ПОЛЕ
procedure TForm1.dgPoleDrawCell(Sender: TObject; ACol, ARow: Inte-
ger;
  Rect: TRect; State: TGridDrawState);
var
  area: tRect;
begin
  area:=form1.dgPole.CellRect(ACol, ARow);
  if masPole[ACol, ARow]= NUM_PUSTO then
    dgPole.Canvas.Brush.Color:= clWhite
  else
    dgPole.Canvas.Brush.Color:= clBlack;
  dgPole.Canvas.FillRect(area);
end;
```

И вот мы целиком удовлетворены нащёлканным рисунком – жмём последнюю из немногочисленных кнопок - **sbtPaint**. Вот тут-то, собственно, и начинаются интересные события. Сначала заполняем массив битов, используемый кистью. Лучше вынести эту «канитель» в отдельную процедуру:

```
//ЗАПОЛНИТЬ МАССИВ БИТОВ
procedure TForm1.SetBits;
```

```
var
  i,j: integer;
  b: word;
begin
  listbox1.Clear;
  For j := 0 To 7 do begin //номер байта
    b:= 0;
    for i := 0 to 7 do          //номер бита в байте
      if masPole[i,j]= NUM_PUSTO then
        b:= b or bit[i];
    bits[j]:= b;
    listbox1.Items.Add(inttostr(b) + '  ' + format('%x', [b]));
  end;
end;
```

Всего в узоре 8 байтов по 8 битов каждый. Просматриваем сетку сверху вниз и слева направо. Собирать каждый байт будем в переменной **b**. Сначала все биты сброшены в 0: `b:= 0`. Если встретилась белая клетка, то устанавливаем соответствующий бит в 1: `b:= b or bit[i];` (нулевой бит будет давать чёрную точку в узоре, единичный – белую, - как в сетке). Сделать это можно многими способами, но «дешевле» всего завести массив **bit**, в котором и хранить соответствующие значения. Объявите его там же, где **NUM**_*KLETKA*:

```
 bit: array[0..7] of byte= ( $80, $40, $20, $10, $08, $04, $02, $01
);
```

Обработав строку сетки, мы получим байт, соответствующий установленным и сброшенным битам (цвету ячеек сетки), который сохраним в массиве **bits**. Объявите его там, где *masPole*:

```
 bits: array[0..7] of
word=($0ff,$0ff,$0ff,$0ff,$0ff,$0ff,$0ff,$0ff);
```

Первоначально массив заполнен байтами **FF**, что даёт просто закраску фигуры белым цветом.

Значение каждого байта узора выводится в список **listbox1**: во-первых, интересно, во-вторых, полезно. Вы можете записать значения байтов понравившегося узора и использовать в своей программе, заполнив массив, подобный **bits**. Тогда вам не придётся загружать узоры из файла. Обработав все 8 строк сетки, мы сформируем необходимый массив битов – можно идти дальше:

```
//ЗАКРАСИТЬ ПРЯМОУГОЛЬНИК
procedure TForm1.sbtPaintClick(Sender: TObject);
var
```

```
  hbm: HBITMAP;
  brush: HBRUSH;
  brushOld: HBRUSH;
  dc: HDC;
begin
  SetBits;
  //очистить поле образца:
  image1.Canvas.Brush.Color:= clWhite;
  image1.Canvas.FillRect(image1.Canvas.ClipRect);
  dc:= image1.Canvas.Handle;
  //создаём монохромную битовую карту:
  hbm:=CreateBitmap(8,8,1,1,@bits);
  //создаём узорную кисть:
  brush:=CreatePatternBrush(hbm);
  //и выбираем её в контексте устройства:
  brushOld:= SelectObject(dc, brush);
  //закрашиваем прямоугольник:
  Rectangle(dc,0,0,image1.width-1,image1.height-1);
  //очистить память:
  SelectObject(dc, brushOld);
  DeleteObject(brush);
end;
```

Закрашивать узором будем прямоугольник на канве компонента **Image1**, поэтому устанавливаем соответствующий контекст устройства: `dc:= image1.Canvas.Handle;` Далее создаём чёрно-белую битовую карту с нашим узором: `hbm:=CreateBitmap(8,8,1,1,@bits);` Конечно, никто вам не запретит изготовить цветной узор, но с ним возни больше, а результат вряд ли оправдает ваши труды.

Для раскрашивания узором нужна подходящая кисть: `brush:=CreatePatternBrush(hbm);`, которую нужно выбрать затем в контексте устройства: `brushOld:= SelectObject(dc, brush);` Этой кистью закрашиваем образец и, уходя, наводим порядок. На сегодня это всё, как говорится в известном рекламном клипе. Кстати, о клипах читайте дальше.



Исходный код программы находится в папке **Pattern**.

Траектории мысли, или *Paths*

Ты выбирай путём,
каким идти путём!

Автокаламбур

В *Windows API* можно найти много «заморочек», и одна из них – траектории. По сути, это те же линии, но с дополнительными возможностями. Применений им можно найти уйму, но мы разумно ограничим себя в поисках неизведанных удовольствий.

При чрезмерной скрытности характера можно сделать себе сюрприз, нарисовав нужные фигуры невидимыми линиями, а потом в удобное время вывести всё творение целиком. Для демонстрации этого феномена нам потребуется, конечно же, форма:

```
BorderIcons=[biSystemMenu,biMinimize],
BorderStyle=bsSingle,
Caption= ‘  Paths’,
Heght=274,
Width=528.
```

На ней установим компонент **Image1** для рисования:

```
Height=241,
Width=437,
Left=4,
Top=4
```

и две кнопки – **sbtClearPole** и **sbtExit**. Трудятся они согласно квалификации:

```
//ОЧИСТИТЬ ПОЛЕ
procedure TForm1.sbtClearPoleClick(Sender: TObject);
begin
  image1.Canvas.Brush.Color:= clWhite;
  image1.Canvas.Brush.Style:= bsSolid;
  image1.Canvas.FillRect(image1.ClientRect)
end;
//ЗАКРЫТЬ ПРОГРАММУ
procedure TForm1.sbtExitClick(Sender: TObject);
begin
  close
end;
```

Для удобства размеры поля хранятся в переменных **w** и **h**:

```
  private
```

```
    { Private declarations }
    w, h: integer;
```

Значение им присваивается при старте программы:

```
//СОЗДАТЬ ФОРМУ
procedure TForm1.FormCreate(Sender: TObject);
begin
  //размеры поля:
  w:= image1.Width-1;
  h:= image1.Height-1;
  //очистить поле:
  sbtClearPoleClick(Self);
end;
```

Ещё одна кнопка – **sbtPaths** – будет демонстрировать механизм записи и предъявления траекторий:

```
//ПОКАЗАТЬ ТРАЕКТОРИЮ
procedure TForm1.sbtPathsClick(Sender: TObject);
Const radStar: Double = 144 * PI / 180;
var
  dc: HDC;
  xMid, yMid, R: integer;
  ptPoly: array[1..5] of TPoint;
  i: integer;
begin
  //закрашиваем жёлтым цветом:
  image1.Canvas.Brush.Color:= clYellow;
  //перо:
  image1.Canvas.Pen.Width:= 3;
  image1.Canvas.Pen.Color:= clGreen;
  //центр многоугольника:
  xMid := w div 2;
  yMid := h div 2 + 10;
  //радиус описанной окружности:
  R := yMid;
  //контекст устройства:
  dc:= image1.Canvas.Handle;
  //начинаем запись траектории:
  BeginPath(dc);
    //вычисляем координаты вершин многоугольника:
    For i := 1 To 5 do begin
      ptPoly[i].x := xMid + round((Cos(i * radStar- pi / 10) *
R));
      ptPoly[i].y := yMid+ round((Sin(i * radStar- pi / 10) * R));
    end;
    image1.Canvas.Polygon(ptPoly);
  //заканчиваем запись траектории:
  EndPath(dc);
  //задержка перед показом траектории:
```

```
  sleep(1000);
  //отрисовать траекторию:
  //StrokePath(dc);
  //закрасить траекторию текущей кистью:
  //FillPath(dc);
  //обвести и закрасить траекторию:
  StrokeAndFillPath(dc);
end;
```

В роли многоугольника выступит наша знакомая звезда. Как обычно, при обращении к функциям *Windows API* нам потребуется контекст устройства, на котором мы будем рисовать: `dc:= image1.Canvas.Handle;` В нашем случае это канва компонента **image1**. Запись траектории для выбранного **DC** всегда начинается с вызова функции **BeginPath(dc)**, а заканчивается вызовом функции **EndPath(dc)**. Между этими «скобками» вы можете нарисовать всё что угодно (справедливости ради следует отметить, что Windows 9x несколько ущербны в этом вопросе: они позволяют рисовать все фигуры, но записывают только *LineTo, MoveTo, PolyBezier, PolyBezierTo, Polygon, Polyline, PolylineTo, PolyPolygon, PolyPolyline* и *TextOut*, плюс функция CloseFigure, замыкающая открытые фигуры; ни прямоугольников, ни эллипсов начертить нельзя), но на экране вы никаких изменений не увидите: траектория только записана – чтобы её увидеть, нужно обратиться к одной из функций: **StrokePath(dc)** очертит траекторию выбранным пером, **FillPath(dc)** закрасит её текущей кистью, без контура, и, наконец, **StrokeAndFillPath(dc)** сделает все малярные работы. В нашей программе введена секундная задержка между записью траектории и выводом её на экран, чтобы каждый мог подивиться увиденному.

Второе применение траекторий – создание областей. На кнопку **sbtPoly** мы посадим процедуру, которая будет создавать комбинацию областей и закрашивать её текущей кистью. Одну область сотворим, как и раньше, с помощью функций **CreateRectRgn** или **CreateEllipticRgn**, а затем запишем траекторию, превратим её в область и скомбинируем с первой, используя один *режимов*:

- **RGN_AND** - общая часть двух областей (наложение),
- **RGN_COPY** - первая область,
- **RGN_DIFF** - из первой области исключается вторая,
- **RGN_OR** - сумма областей,
- **RGN_XOR** - сумма областей, исключая область наложения.

Выбор режима сочетания областей можно закрепить за компонентом список **ListBox1**:

```
Height=73,
Top=0,
Left=444,
Width=73.
```

Свойство **Items** заполните в *Редакторе* строками:

```
RGN_AND
RGN_COPY
RGN_DIFF
RGN_OR
RGN_XOR
```

Текущий режим будем хранить в переменной **RGN_MODE**, а её значение выбирать из массива **mode**. Их объявление нужно сделать так:

```
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
Dialogs,
  Buttons, ExtCtrls, StdCtrls;
const
  mode:                array[0..4]                of                integer=(RGN_AND,RGN_COPY,RGN_DIFF,RGN_OR,RGN_XOR);
var
  RGN_MODE: integer= RGN_AND;
```

Режим будет выбран при щелчке на соответствующей строке списка:

```
//ВЫБОР РЕЖИМА
procedure TForm1.ListBox1Click(Sender: TObject);
begin
  RGN_MODE:= mode[ListBox1.ItemIndex];
end;
```

Запускайте программу, выбирайте режим и жмите на кнопку **sbtPoly** – результат не заставит себя долго ждать:

```
//МНОГОУГОЛЬНИК
procedure TForm1.sbtPolyClick(Sender: TObject);
Const radPentagon: Double = 72 * PI / 180;
var
  dc: HDC;
  rgn, rgn2: HRGN;
  i: integer;
  x, y: integer;
  R: integer;
  xMid, yMid: integer;
  ptPent: array[1..5] of TPoint;
begin
  //центр многоугольника:
  xMid := w div 2;
```

```
  yMid := h div 2 + 10;
  //радиус описанной окружности:
  R := yMid;
  //очистить поле:
  sbtClearPoleClick(Self);
  //кисть синяя ромбиком:
  image1.Canvas.Brush.Color:= clBlue;
  image1.Canvas.Brush.Style:= bsDiagCross;
  //контекст устройства:
  dc:= image1.Canvas.Handle;
  //создаём эллиптическую область:
  rgn:= CreateEllipticRgn(20,20, w-20,h-20);
  //начинаем запись траектории:
  BeginPath(dc);
    //вычисляем точки:
    For i := 1 To 5 do begin
      ptPent[i].x := xMid + round((Cos(i * radPentagon- pi / 10) *
R));
      ptPent[i].y := yMid + round((Sin(i * radPentagon- pi / 10) *
R));
    end;
    image1.Canvas.Polygon(ptPent);
    //Polygon(dc,ptPent,5);
  //заканчиваем запись траектории:
  EndPath(dc);
  //превращаем траекторию в область:
  rgn2:=PathToRegion(dc);
  //находим комбинацию двух областей:
  CombineRgn(rgn,rgn,rgn2,RGN_MODE);
  //закрашиваем текущей кистью:
  FillRgn(dc, rgn, image1.Canvas.Brush.Handle);
end;
```

Последняя из рассматриваемых нами возможностей траекторий – это создание хитроумных областей отсечения: всё, что выходит за границы области отсечения на экране не появится. Областью отсечения в нашем примере будет комбинация прямоугольной области, созданной вызовом функции **CreateRoundRectRgn** (вы можете заменить её другой подобной или вообще не создавать – тогда закомментируйте строку `SelectClipRgn(dc, rgn);`), по размеру поля (или строки) и траектории, созданной из этой строки, которая комбинируется с первой одним из рассмотренных ранее способов. Отвечать за действо мы назначим кнопку **sbtText:**

```
//ОБЛАСТЬ ОТСЕЧЕНИЯ
procedure TForm1.sbtTextClick(Sender: TObject);
var
  dc: HDC;
  rgn: HRGN;
  i: integer;
```

```
  x, y: integer;
  r: integer;
  text: string;
begin
  //Очистить поле:
  sbtClearPoleClick(Self);
  //строка, используемая для создания траектории:
  text:='ТРАЕКТОРИЯ';
  with image1.canvas do begin
    //контекст устройства:
    dc:= Handle;
    //выбираем шрифт:
    Font.Name:= 'Arial';
    Font.Style:= Canvas.Font.Style+[fsBold];
    Font.Size:= 48;
    //создаём область по размеру строки:
    //rgn:= CreateRectRgn(0,0, TextEx-
tent(Text).cx,TextExtent(Text).cy);
    //создаём область по размеру поля:
    //rgn:= CreateRectRgn (0, 0, w, h);
    rgn:= CreateRoundRectRgn(0,0, w,h,50,50);
    //выбираем её в качестве области отсечения:
    SelectClipRgn(dc, rgn);
    //начинаем запись траектории:
    BeginPath(dc);
      TextOut(0,0, text);
    //заканчиваем запись траектории:
    EndPath(dc);
    //область отсечения = комбинации первой области и траектории:
    SelectClipPath(dc, RGN_MODE);
    //чертим радиусы окружности:
    r:= w+h;
    for i:=0 to 180 do begin
      MoveTo(0,0);
      x:= round(r*cos(pi*i / 360.0));
      y:= round(r*sin(pi*i / 360.0));
      LineTo(x, y);
    end;
  end
end;
```

Чтобы увидеть, как действует область отсечения, из левого верхнего угла поля проведём радиусы окружности – на экране они будут видны только в области отсечения (Рис. П6.21).

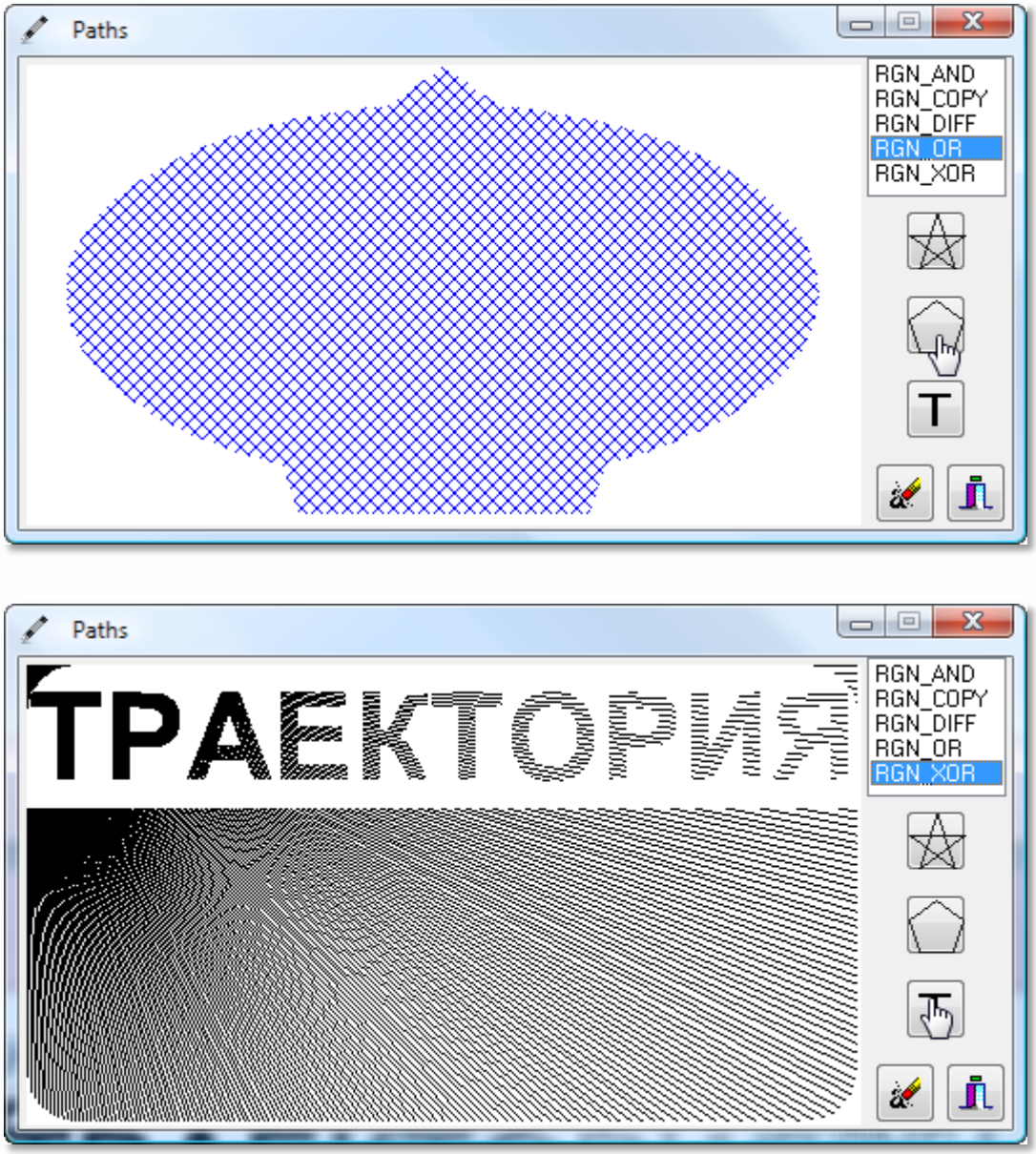


Рис. П6.21. Искусство есть отсечение лишнего (Роден)

Стоит обратить внимание на область отсечения, созданную текстом, - это не буквы, а прямоугольник по размеру строки за вычетом самих букв.

На этом наши одиссеи с линиями и фигурами завершаются, мы благополучно добрались до свойств канвы, связанных с картинками.



Исходный код программы находится в папке **Paths**.

Картинки с выставки, или Художественные методы канвы

Все линии и геометрические фигуры годятся лишь для создания областей и простейших рисунков. Чтобы нарисовать хорошую картинку, потребуется графический редактор. А вот затем её можно использовать в программе на *Delphi*.

Вначале мы рассмотрим 2 метода, которые умеют перерисовывать объект **TGraphic** на канву. Он может содержать картинки четырёх основных графических форматов *Windows*:

- растровую картинку **BMP** (класс **TBitmap**),
- сжатую растровую картинку **JPG**, **JPEG** (**TJPEGImage**),
- значок **ICO** (**TIcon**),
- метафайл **WMF**, **EMF** (**TMetafile**).

Первый метод

```
Draw(X, Y: Integer; Graphic: TGraphic);
```

просто перерисовывает содержимое *Graphic* на канву. Размеры картинки при этом не изменяются, а верхний левый угол картинки располагается в точке канвы с координатами (**X, Y**).

Второй метод

```
StretchDraw(const Rect: TRect; Graphic: TGraphic );
```

похитрее: он делает то же, что и **Draw**, но подгоняет картинку под заданный прямоугольник **Rect** на канве. Если он по размерам не совпадает с картинкой, то она масштабируется. Вид картинок при переносе на канву не изменяется, за исключением формата *BMP*, для которого можно установить прозрачный цвет и задать в методе **CopyMode** спецэффекты при копировании, и **JPG**, который можно выводить в цветном и сером вариантах. Некоторыми из этих возможностей мы воспользуемся в нашей программе, остальным придётся подождать своего часа.

А теперь создаём **форму** – как обычно. Из нового:

```
Caption=' Draw & StretchDraw',
Height=458,
Width=435.
```

Кнопка **sbtClearPole** будет очищать поле, кнопка **sbtExit** закрывать программу, а кнопка **sbtStop** останавливать рисование спиралей (спирали из значков выводятся на экран очень быстро, так что для них эта кнопка не нужна, а вот метафайлы вычерчиваются значительно медленнее, так что у вас вполне может возникнуть желание закончить это «продолговатое» действо). Последняя пара кнопок - **sbtDraw** и **sbtStretchDraw** – потребуется для переключения методов рисования.

Ползунком **TrackBar1** устанавливается коэффициент масштабирования картинок при рисовании методом **StretchDraw**, метка **Label1** показывает текущий коэффициент. По привычке рисовать будем на канве компонента **ImgPole** типа **TImage**:

```
Height=458,
Width=435.
```

Исходные картинки различных графических форматов будем хранить в таких же компонентах, но меньшего размера – 50 х 50 пикселов:

```
ImgJpg (tag=0),
ImgBmp (tag=1),
ImgIco (tag=2),
ImgWMF (tag=3),
ImgEMF (tag=4).
```

В них нужно предварительно поместить картинки нужных форматов (для этого в режиме проектирования щёлкните дважды на компоненте и загрузите файл; чтобы картинка точно вписалась в размеры компонента, его свойство **Stretch** должно быть установлено в True). Ещё две картинки (в примере это обезьяна и тыква) используются при вычерчивании спиралей, в верхней - **ImgSpiral** - находится значок, в нижней - **ImgSpiralEMF** – метафайл.

С помощью ползунка **TrackBar2** можно изменять угол между ветвями спирали, его значение показывает метка **Label2**. В итоге интерфейс программы должен получиться таким (Рис. П6.22).



Рис. П6.22. Высокохудожественная самодеятельность!

Код программы несложный, вы сами сможете в нём разобраться:

```
unit DrawUnit;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
Dialogs,
  ExtCtrls, Buttons, jpeg, ComCtrls, StdCtrls;
var
  flgStop: Boolean= False;
type
  TDrawMode=(dmDraw, dmStretchDraw);
  TForm1 = class(TForm)
    ImgPole: TImage;
    sbtClearPole: TSpeedButton;
    sbtExit: TSpeedButton;
    ImgJpg: TImage;
    ImgBmp: TImage;
    ImgIco: TImage;
    ImgWMF: TImage;
    ImgEMF: TImage;
    TrackBar1: TTrackBar;
    sbtDraw: TSpeedButton;
    sbtStretchDraw: TSpeedButton;
    Label1: TLabel;
    ImgSpiral: TImage;
```

```
    TrackBar2: TTrackBar;
    Label2: TLabel;
    ImgSpiralEMF: TImage;
    sbtStop: TSpeedButton;
    procedure sbtClearPoleClick(Sender: TObject);
    procedure sbtExitClick(Sender: TObject);
    procedure FormCreate(Sender: TObject);
    procedure ImgPoleMouseDown(Sender: TObject; Button: TMouseBut-
ton;
      Shift: TShiftState; X, Y: Integer);
    procedure ImgPoleMouseMove(Sender: TObject; Shift:
TShiftState; X,
      Y: Integer);
    procedure ImgPoleMouseUp(Sender: TObject; Button: TMouseBut-
ton;
      Shift: TShiftState; X, Y: Integer);
    procedure ImgJpgClick(Sender: TObject);
    procedure TrackBar1Change(Sender: TObject);
    procedure sbtDrawClick(Sender: TObject);
    procedure ImgSpiralDblClick(Sender: TObject);
    procedure TrackBar2Change(Sender: TObject);
    procedure sbtStopClick(Sender: TObject);
    procedure ImgBmpDblClick(Sender: TObject);
    procedure ImgJpgDblClick(Sender: TObject);
  private
    { Private declarations }
    pic: TGraphic;
    Drawing: boolean;
    Mas: real;
    mode: TDrawMode;
  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation

{$R *.DFM}

//СОЗДАТЬ ФОРМУ
procedure TForm1.FormCreate(Sender: TObject);
begin
  //очистить поле:
  sbtClearPoleClick(Self);
  //картинка по умолчанию:
  pic:=imgJpg.Picture.Graphic;
  //исходный масштаб:
  mas:= 1;
end;

//ЗАКРЫТЬ ПРОГРАММУ
```

```
procedure TForm1.sbtExitClick(Sender: TObject);
begin
  close
end;

//ОЧИСТИТЬ ПОЛЕ
procedure TForm1.sbtClearPoleClick(Sender: TObject);
begin
  imgPole.Canvas.Brush.Color:= clWhite;
  imgPole.Canvas.Brush.Style:= bsSolid;
  imgPole.Canvas.FillRect(imgPole.ClientRect)
end;

//ЩЁЛКНУТЬ НА КАРТИНКЕ - ВЫБРАТЬ ЕЁ
procedure TForm1.ImgJpgClick(Sender: TObject);
begin
  case (Sender as TImage).tag of
    0: pic:= imgJpg.Picture.Graphic;
    1: pic:= imgBmp.Picture.Graphic;
    2: pic:= imgIco.Picture.Graphic;
    3: pic:= imgWMF.Picture.Graphic;
    4: pic:= imgEMF.Picture.Graphic;
  end;
end;

//ИЗМЕНИТЬ МАСШТАБ КАРТИНКИ
procedure TForm1.TrackBar1Change(Sender: TObject);
begin
  case TrackBar1.Position of
    1: Mas:= 1.0;
    2: Mas:= 0.25;
    3: Mas:= 0.5;
    4: Mas:= 0.75;
    5: Mas:= 1.5;
    6: Mas:= 2.0;
    7: Mas:= 3.0;
    8: Mas:= 4.0;
    9: Mas:= 6.0;
    10: Mas:= 8.0
  end;
  label1.Caption:= FloatToStr(mas);
end;

//ВЫБРАТЬ МЕТОД КАНВЫ ДЛЯ РИСОВАНИЯ
procedure TForm1.sbtDrawClick(Sender: TObject);
begin
  if sbtDraw.Down then
  begin
    mode:= dmDraw;
    //в методе Draw картинка не масштабируется:
    TrackBar1.Enabled:= False;
    Label1.Enabled:= False;
  end;
```

```
  if sbtStretchDraw.Down then
  begin
    mode:= dmStretchDraw;
    TrackBar1.Enabled:= True;
    Label1.Enabled:= True;
  end;
end;

//ПЕРЕКЛЮЧИТЬ: ПРОЗРАЧНЫЙ РАСТР - НЕПРОЗРАЧНЫЙ РАСТР
procedure TForm1.ImgBmpDblClick(Sender: TObject);
begin
  //установить прозрачный цвет:
  imgBMP.Picture.Bitmap.TransparentMode := tmAuto;
  //или:
  //imgBMP.Picture.Bitmap.TransparentMode := tmFixed;
  imgBMP.Transparent := not imgBMP.Transparent;
end;

//ВЫВОДИТЬ КАРТИНКУ ЦВЕТНУЮ - В СЕРЫХ ТОНАХ
procedure TForm1.ImgJpgDblClick(Sender: TObject);
begin
  TJPEGImage(ImgJPG.Picture.Graphic).Grayscale:= not
  TJPEGImage(ImgJPG.Picture.Graphic).Grayscale;
end;

//НАЖАТЬ КНОПКУ - СКОПИРОВАТЬ КАРТИНКУ
procedure TForm1.ImgPoleMouseDown(Sender: TObject; Button: TMouse-
Button;
  Shift: TShiftState; X, Y: Integer);
var
  rec: Trect;
begin
  Drawing:= True;
  case mode of
    dmDraw:          //метод Draw
      imgPole.Canvas.Draw(X, Y, pic);
    dmStretchDraw:  //метод StretchDraw
    begin
      //масштабировать картинку:
      rec:= Rect(X, Y, X+round(pic.Width * mas),
Y+round(pic.Height* mas));
      imgPole.Canvas.StretchDraw(rec, pic);
    end
  end
end;

//ПЕРЕМЕЩАЕМ МЫШЬ - РИСУЕМ КАРТИНКОЙ, КАК КИСТЬЮ
procedure TForm1.ImgPoleMouseMove(Sender: TObject; Shift:
TShiftState; X,
  Y: Integer);
var
  rec: Trect;
begin
```

```
  if drawing then
  begin
    case mode of
      dmDraw:
        imgPole.Canvas.Draw(X, Y, pic);
      dmStretchDraw:
      begin
        if mas> 1 then exit; //большая картинка медленно рисуется!
        rec:= Rect(X, Y, X+round(pic.Width * mas),
Y+round(pic.Height* mas));
        imgPole.Canvas.StretchDraw(rec, pic);
      end
    end
  end
end;

//ОТПУСТИТЬ КНОПКУ МЫШИ - ЗАКОНЧИТЬ РИСОВАНИЕ
procedure TForm1.ImgPoleMouseUp(Sender: TObject; Button: TMouse-
Button;
  Shift: TShiftState; X, Y: Integer);
begin
  Drawing:= False;
end;

////////////////////////////////////////////////////////////////
//////////////////////////////  СПИРАЛЬ ////////////////////////
////////////////////////////////////////////////////////////////

//ЗАДАТЬ УГОЛ СПИРАЛИ
procedure TForm1.TrackBar2Change(Sender: TObject);
begin
  label2.Caption:= inttoStr(TrackBar2.Position);
end;

//НАРИСОВАТЬ СПИРАЛЬ ИЗ ЗНАЧКА ImgSpiral ИЛИ МЕТАФАЙЛА
ImgSpiralEMF
procedure TForm1.ImgSpiralDblClick(Sender: TObject);
var
  dxSrc, dySrc: integer;
  xInc, yInc, xSize, ySize, x, y: integer;
  radCur: Single;
  degCur, angInc: Integer;
  pict: TGraphic;
begin
  //выбрать картинку:
  case (Sender as TImage).tag of
    0: pict:= ImgSpiral.Picture.Graphic;
    1: pict:= ImgSpiralEMF.Picture.Graphic;
  end;
  //размеры источника:
  dxSrc := pict.Width;
  dySrc := pict.Height;
  //параметры спирали:
```

```
  xInc:= 1; yInc:= 1;
  xSize:= 3; ySize:= 3;
  degCur:= 0;
  angInc:= TrackBar2.Position; //91 71 89 90 181
  //рисуем спираль:
  with pict do
  begin
    //начинаем из центра:
    x:= (imgPole.Width div 2) - (dxSrc div 2);
    y:= (imgPole.Height div 2) - (dySrc div 2);
    repeat
      //выводим картинку в текущей позиции:
      imgPole.Canvas.Draw(x,y,Pict);
      //вычисляем угол в радианах:
      radCur:= (degCur - 90) * (PI / 180);
      //вычисляем следующие x и y:
      x := x + Round(xSize * Cos(radCur));
      y := y + Round(ySize * Sin(radCur));
      //расширение спирали:
      xSize := xSize + xInc;
      ySize := ySize + yInc;
      //изменение угла:
      degCur := (degCur + angInc) Mod 360;
      //если нажата кнопка sbtStop, то закончить рисование:
      if flgStop= True then
      begin
        flgStop:= False;
        exit
      end;
      //sleep(3); //пауза
      application.ProcessMessages;
    until (x < 0) or (x > imgPole.Width) or
          (y < 0) or (y > imgPole.Height);
  end;
end;

//ОСТАНОВИТЬ РИСОВАНИЕ
procedure TForm1.sbtStopClick(Sender: TObject);
begin
  flgStop:= True;
end;

end.
```

Запускайте программу, выбирайте картинку для рисования – для этого просто один раз щёлкните на ней. А, щёлкнув дважды на верхней картинке, вы превратите её из цветной в серую. То же действие на второй картинке сверху превратит её в прозрачную: белый фон не будет перерисовываться на канву.

Выбрав картинку, щёлкните мышкой в нужном месте поля для рисования – появится выбранная вами картинка (если нажата кнопка **sbtStretchDraw**, то можно изменить её масштаб). Вы можете даже рисовать картинкой, передвигая мышку по полю с нажатой кнопкой.

Двойной щелчок на двух нижних картинках вызовет интересный спецэффект: будет нарисована «спиральная галактика» из значка или метафайла (они всегда прозрачные, поэтому не требуют никаких дополнительных манипуляций). Вид галактик можно изменять, перемещая нижний ползунок. Не пожалейте времени – поищите свою галактику (Рис. П6.23)!



Рис. П6.23. Спиралепипед

Исходный код программы находится в папке **Draw**.

Следующая пара методов позволяет работать только с растровыми картинками, но зато даёт возможность копировать любой прямоугольный фрагмент из исходной картинки на канву. Это может пригодиться при

разработке спецэффектов (один из них вы найдёте в программе *Copy*), игр типа «Пятнашек» или различных мозаик.

Метод

```
CopyRect(Dest: TRect; Canvas: TCanvas; Source: TRect);
```

просто копирует заданный прямоугольный фрагмент картинки с одной канвы на другую. Если размеры исходного и целевого прямоугольников различаются, то картинка масштабируется (подгоняется под целевой прямоугольник).

Интерфейс этой программы мало чем отличается от предыдущего (Рис. П6.24).

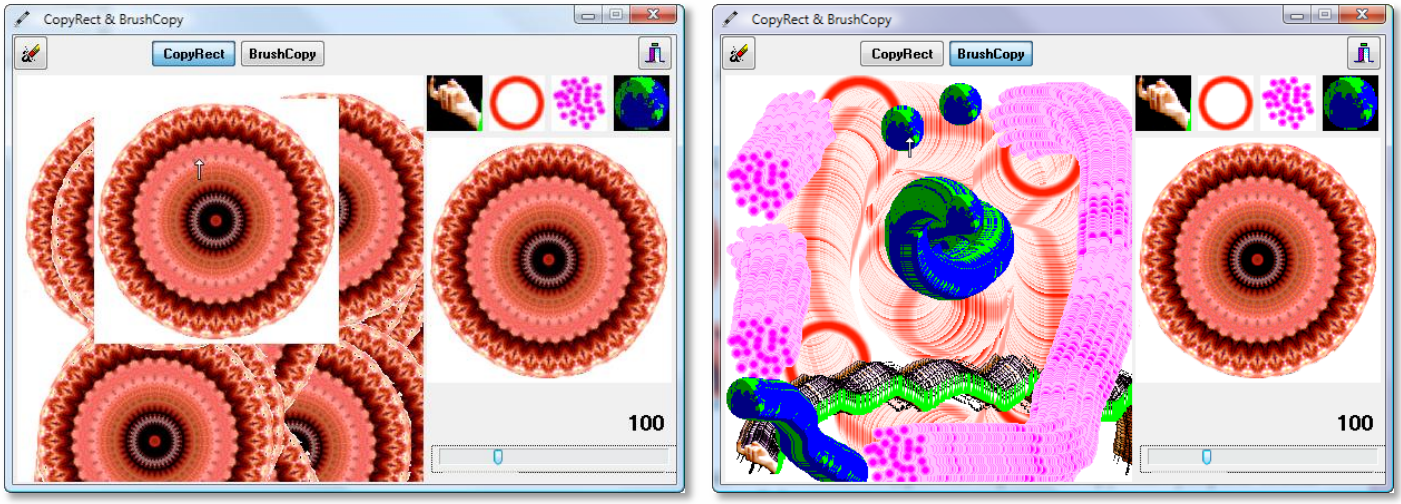


Рис. П6.24. Нехудожественная самодеятельность!

Справа находится большая квадратная картинка с круглым узором. Чтобы выбрать нужный фрагмент, нажмите кнопку мыши на картинке и растягивайте прямоугольник до требуемой величины (если вы хотите скопировать картинку целиком, просто щёлкните на ней). С такой техникой рисования прямоугольников мы уже встречались, так что здесь нет ничего нового. Затем этот фрагмент вы можете копировать сколько угодно раз на левом прямоугольном поле, для этого достаточно щёлкнуть кнопкой мыши в любой его точке. Масштаб картинки на поле изменяется с помощью ползунка (он внизу справа). По умолчанию он равен 100%, то есть картинка копируется без изменений. Щёлкните на поле правой кнопкой мыши – и тот же рисунок появится со спецэффектом: он выводится от краёв к центру.

Второй метод из этой парочки

```
BrushCopy(const Dest: TRect;
          Bitmap: TBitmap;
          const Source: TRect; Color: TColor);
```

поинтереснее. Он может не только копировать фрагменты растровой картинки, как **CopyRect**, но и кое-что ещё. Например, вы можете заменить цвет **Color** на исходной картинке цветом текущей кисти канвы. Сделайте цвет текущей кисти красным:

```
imgPole.Canvas.Brush.Color:=clRed;,
```

а заменяемый цвет белым (цвет фона картинки):

```
imgPole.Canvas.BrushCopy (dstRect, image1.picture.bitmap, srcRect,
clWhite);
```

и картинка будет копироваться на канву в красном обрамлении (не забудьте переключиться на этот метод рисования, нажав кнопку с надписью *BrushCopy*). Другое забавное свойство этого метода – вы можете вообще убрать фон и копировать прозрачную картинку. Стиль текущей кисти канвы должен быть установлен в **bsClear**, а прозрачный цвет на картинке может быть выбран несколькими способами (смотри программу), например, `image1.Picture.Bitmap.TransparentMode:=tmAuto;` Теперь фрагменты картинки будут копироваться на канву вообще без фона.

Таким же способом можно превратить любой рисунок в кисть. Над большим узором вы найдёте 4 рисунка. Щёлкните на любом из них, и он станет кистью. Нажмите **правую** кнопку мыши на поле и передвигайте её – за ней останется след из прозрачных рисунков. Меняя скорость движения мыши, можно получить весьма эффектные траектории. В режиме проектирования вы можете заменить эти кисточки своими. С помощью некоторые из них можно нарисовать оригинальный фон для дальнейшего использования.

Код программы не столь сложен, чтобы на нём останавливаться. Если где-то придётся поднатужиться, так это пойдёт вам только на пользу!

```
unit CopyUnit;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
  ExtCtrls, Buttons, ComCtrls, StdCtrls;
```

```
type
  TDrawMode=(dmCopyRect, dmBrushCopy);
  TForm1 = class(TForm)
    ImgPole: TImage;
    sbtClearPole: TSpeedButton;
    sbtExit: TSpeedButton;
    ImgBmp1: TImage;
    sbtCopyRect: TSpeedButton;
    sbtBrushCopy: TSpeedButton;
    TrackBar2: TTrackBar;
    Label2: TLabel;
    Image1: TImage;
    ImgBmp2: TImage;
    imgBmp3: TImage;
    imgBmp4: TImage;
    procedure sbtClearPoleClick(Sender: TObject);
    procedure sbtExitClick(Sender: TObject);
    procedure FormCreate(Sender: TObject);
    procedure ImgPoleMouseDown(Sender: TObject; Button: TMouseBut-
ton;
      Shift: TShiftState; X, Y: Integer);
    procedure ImgPoleMouseMove(Sender: TObject; Shift:
TShiftState; X,
      Y: Integer);
    procedure ImgPoleMouseUp(Sender: TObject; Button: TMouseBut-
ton;
      Shift: TShiftState; X, Y: Integer);
    procedure ImgBMP1Click(Sender: TObject);
    procedure sbtCopyRectClick(Sender: TObject);
    procedure TrackBar2Change(Sender: TObject);
    procedure Image1MouseDown(Sender: TObject; Button: TMouseBut-
ton;
      Shift: TShiftState; X, Y: Integer);
    procedure Image1MouseMove(Sender: TObject; Shift: TShiftState;
X,
      Y: Integer);
    procedure Image1MouseUp(Sender: TObject; Button: TMouseButton;
      Shift: TShiftState; X, Y: Integer);
  private
    { Private declarations }
    _canvas, bmp_canvas: TCanvas;
    bmp: TBitmap;
    Drawing: boolean;
    bmpDrawing: boolean;
    Origin, MovePt: TPoint;
    masSrc: integer;
    mode: TDrawMode;
    procedure Spiral(x,y: integer);
  public
    { Public declarations }
  end;
```

```
var
  Form1: TForm1;

implementation

{$R *.DFM}

//СОЗДАТЬ ФОРМУ
procedure TForm1.FormCreate(Sender: TObject);
begin
  //очистить поле:
  sbtClearPoleClick(Self);
  //картинка по умолчанию:
  _canvas:=image1.Canvas;
  //кисточка:
  bmp_canvas:= imgBmp1.Canvas;
  bmp:=imgBmp1.Picture.Bitmap;
  //исходный масштаб:
  masSrc:=100;
  //точечная линия для прямоугольника-выделения области картинки:
  image1.Canvas.Pen.Style:= psDot;
end;

//ЗАКРЫТЬ ПРОГРАММУ
procedure TForm1.sbtExitClick(Sender: TObject);
begin
  close
end;

//ОЧИСТИТЬ ПОЛЕ
procedure TForm1.sbtClearPoleClick(Sender: TObject);
begin
  imgPole.Canvas.Brush.Color:= clWhite;
  imgPole.Canvas.Brush.Style:= bsSolid;
  imgPole.Canvas.FillRect(imgPole.ClientRect)
end;

//ЩЁЛКНУТЬ НА КАРТИНКЕ - ВЫБРАТЬ ЕЁ В КАЧЕСТВЕ КИСТИ
procedure TForm1.ImgBMP1Click(Sender: TObject);
begin
  case (Sender as TImage).tag of
    0: begin
         bmp_canvas:= imgBmp1.Canvas;
         bmp:=imgBmp1.Picture.Bitmap;
       end;
    1: begin
         bmp_canvas:= imgBmp2.Canvas;
         bmp:=imgBmp2.Picture.Bitmap;
       end;
    2: begin
         bmp_canvas:= imgBmp3.Canvas;
         bmp:=imgBmp3.Picture.Bitmap;
       end;
```

```
    3: begin
         bmp_canvas:= imgBmp4.Canvas;
         bmp:=imgBmp4.Picture.Bitmap;
       end;
  end;
end;

//ВЫБРАТЬ МЕТОД КАНВЫ ДЛЯ РИСОВАНИЯ
procedure TForm1.sbtCopyRectClick(Sender: TObject);
begin
  if sbtCopyRect.Down then mode:= dmCopyRect;
  if sbtBrushCopy.Down then mode:= dmBrushCopy;
end;

//НАЖАТЬ КНОПКУ - СКОПИРОВАТЬ КАРТИНКУ
procedure TForm1.ImgPoleMouseDown(Sender: TObject; Button: TMouse-
Button;
  Shift: TShiftState; X, Y: Integer);
var
  dstRect, srcRect: Trect;
  dstW, dstH, srcW, srcH: integer;
begin
  //если нажата правая кнопка мыши в режиме dmCopyRect,
  //то вывести картинку со спецэффектом:
  if (button= mbRight) and (mode=dmCopyRect) then
  begin
    Spiral(X, Y);
    exit
  end;
  Drawing:= True;
  //если нажата правая кнопка мыши в режиме dmBrushCopy,
  //то рисовать картинкой:
  if (button= mbRight) then exit;
  //размеры выделенного прямоугольника:
  srcRect:= rect(Origin.X, Origin.Y, MovePt.X, MovePt.Y);
  srcW:= abs(MovePt.X - Origin.X);
  srcH:= abs(MovePt.Y - Origin.Y);
  //не был выделен прямоугольник на картинке:
  if srcW*srcH=0 then
  begin
    srcW:= image1.Width;
    srcH:= image1.Height;
    srcRect:= rect(0,0, srcW,srcH)
  end;
  //масштабировать прямоугольное выделение:
  dstW:= srcW * masSrc div 100;
  dstH:= srcH * masSrc div 100;
  dstRect:= Rect(X, Y, X+dstW, Y+dstH);
  case mode of
    dmCopyRect:    //метод CopyRect
      imgPole.Canvas.CopyRect(dstRect, _canvas, srcRect);
    dmBrushCopy:   //метод BrushCopy
    begin
```

```
      //цвет, заменяющий clWhite в картинке:
      //imgPole.Canvas.Brush.Color:=clRed;

      //если TransparentColor = заменяемому цвету и
      //кисть прозрачная, то заменяемый цвет также будет прозрач-
ным
      imgPole.Canvas.Brush.Style:=bsClear;
      //установить прозрачный цвет:
      //image1.Picture.Bitmap.TransparentColor:=clWhite;
      //или:
      image1.Picture.Bitmap.TransparentMode := tmAuto;
      //или:
      //imgBMP.Picture.Bitmap.TransparentMode := tmFixed;
      //перенести картинку на канву поля:
      imgPole.Canvas.BrushCopy(dstRect, image1.picture.bitmap,
                               srcRect, clWhite);
    end
  end
end;

//ПЕРЕМЕЩАЕМ МЫШЬ - РИСУЕМ КАРТИНКОЙ, КАК КИСТЬЮ
procedure TForm1.ImgPoleMouseMove(Sender: TObject; Shift:
TShiftState; X,
  Y: Integer);
var
  srcRect, dstRect: Trect;
  clr: TColor;
  w,h: integer;
begin
  if drawing and (mode=dmBrushCopy) and (ssRight in Shift)then
  begin
    srcRect:= bmp_canvas.ClipRect;
    w:=bmp.width;
    h:=bmp.Height;
    dstRect:= rect(X,Y,X+w,Y+h);
    //кисть - прозрачная:
    imgPole.Canvas.Brush.Style:=bsClear;
    clr:= bmp_Canvas.Pixels[0,h-1];
    Bmp.TransparentColor := clr;
    //скопировать картинку в позицию курсора:
    imgPole.Canvas.BrushCopy(dstRect, bmp, srcRect, clr);
  end
end;

//ОТПУСТИТЬ КНОПКУ МЫШИ - ЗАКОНЧИТЬ РИСОВАНИЕ
procedure TForm1.ImgPoleMouseUp(Sender: TObject; Button: TMouse-
Button;
  Shift: TShiftState; X, Y: Integer);
begin
  Drawing:= False;
  //восстановить кисть канвы:
  imgPole.Canvas.Brush.Color:=clWhite;
  imgPole.Canvas.Brush.Style:=bsSolid;
```

```
end;

//НАЧАТЬ ВЫДЕЛЕНИЕ ПРЯМОУГОЛЬНОГО ФРАГМЕНТА КАРТИНКИ
procedure TForm1.Image1MouseDown(Sender: TObject; Button: TMouse-
Button;
  Shift: TShiftState; X, Y: Integer);
begin
  bmpDrawing := True;
  Origin := Point(X, Y);
  MovePt := Origin;
end;

//ВЫДЕЛЯЕМ ПРЯМОУГОЛЬНУЮ ОБЛАСТЬ КАРТИНКИ
procedure TForm1.Image1MouseMove(Sender: TObject; Shift:
TShiftState; X,
  Y: Integer);
begin
  if bmpDrawing then
  begin
    Image1.Canvas.Pen.Mode := pmNotXor;
    Image1.Canvas.Rectangle(Origin.X, Origin.Y, MovePt.X,
MovePt.Y);
    MovePt := Point(X, Y);
    Image1.Canvas.Rectangle(Origin.X, Origin.Y, MovePt.X,
MovePt.Y);
  end;
end;

//ЗАКОНЧИТЬ ВЫДЕЛЕНИЕ ФРАГМЕНТА КАРТИНКИ
procedure TForm1.Image1MouseUp(Sender: TObject; Button: TMouseBut-
ton;
  Shift: TShiftState; X, Y: Integer);
begin
  if bmpDrawing then
  begin
    //стираем выделение:
    Image1.Canvas.Pen.Mode := pmNotXor;
    Image1.Canvas.Rectangle(Origin.X, Origin.Y, MovePt.X,
MovePt.Y);
    bmpDrawing := False;
  end;
end;

////////////////////////////////////////////////////////////////
//                СПИРАЛЬНЫЙ ВЫВОД КАРТИНКИ                   //
////////////////////////////////////////////////////////////////

//ИЗМЕНИТЬ МАСШТАБ
procedure TForm1.TrackBar2Change(Sender: TObject);
begin
  masSrc:= TrackBar2.Position;
  label2.Caption:= inttoStr(TrackBar2.Position);
end;
```

```
//РИСУЕМ КАРТИНКУ НА ПОЛЕ
procedure TForm1.Spiral(x,y: integer);
var
  xLeft, xRight, yTop, yBottom: integer;
  dxSrc, dySrc: integer;
  xInc, yInc: integer;
  dstRect, srcRect: TRect;
  xOff, yOff: integer;
begin
    //смещение координат картинки относительно координат на поле:
    xOff:= -x; yOff:= -y;
    //размеры исходной картинки:
    dxSrc := image1.Width;
    dySrc := image1.Height;
    //размеры элементарных прямоугольников:
    xInc := dxSrc div 40; //xInc := 5;
    yInc := dySrc div 40; //yInc := 5;
    //координаты левого верхнего угла целевого прямоугольника:
    xLeft := x; yTop := y;
    //координаты правого нижнего угла целевого прямоугольника:
    xRight := xLeft + dxSrc - xInc;
    yBottom := yTop + dySrc - yInc;
    //очистить поле:
    sbtClearPoleClick(Self);
    //выводим все стороны спирально
    While (xLeft <= xRight) And (yTop <= yBottom) Do
    begin
      //верхняя сторона:
      x := xLeft;
      repeat
        dstRect:= Rect(x,y,x+xInc, y+ yInc);
        srcRect:= Rect(x+xOff,y+yOff,x+xOff+xInc, y+ yOff + yInc);
        imgPole.Canvas.CopyRect(dstRect, _canvas, srcRect);
        x:= x+ xInc;
      until x > xRight;
      x := x - xInc;
      yTop := yTop + yInc;
      if (x+xOff) > (dxSrc div 2) then applica-
tion.ProcessMessages;
      //правая сторона:
      y := yTop;
      repeat
        dstRect:= Rect(x,y,x+xInc, y+ yInc);
        srcRect:= Rect(x+xOff,y+yOff,x+xOff+xInc, y+yOff + yInc);
        imgPole.Canvas.CopyRect(dstRect, _canvas, srcRect);
        y:= y+ yInc;
      until y > yBottom;
      y := y - yInc;
      xRight := x - xInc;
      if (x+xOff) > (dxSrc div 2) then applica-
tion.ProcessMessages;
      //нижняя сторона:
```

```
      x := xRight;
      repeat
        dstRect:= Rect(x,y,x+xInc, y+ yInc);
        srcRect:= Rect(x+xOff,y+yOff,x+xOff+xInc, y+yOff + yInc);
        imgPole.Canvas.CopyRect(dstRect, _canvas, srcRect);
        x:= x - xInc;
      until x < xLeft;
      x := x + xInc;
      yBottom := y - yInc;
      if (x+xOff) > (dxSrc div 2) then applica-
tion.ProcessMessages;
      //левая сторона:
      y := yBottom;
      repeat
        dstRect:= Rect(x,y,x+xInc, y+ yInc);
        srcRect:= Rect(x+xOff,y+yOff,x+xOff+xInc, y+yOff + yInc);
        imgPole.Canvas.CopyRect(dstRect, _canvas, srcRect);
        y:= y - yInc;
      until y < yTop;
      y := y + yInc;
      xLeft := xLeft + xInc;
      sleep(10);  //задержка
      //if (x+xOff) > (dxSrc div 2) then applica-
tion.ProcessMessages;
    end;
end;

end.
```



Исходный код программы находится в папке **Copy**.

Великий комбинатор, или Свойство *CopyMode*

Усердно копируя из пустого в порожнее, нам трудно было догадаться о существовании такого свойства канвы, как **CopyMode**. А между тем, при создании канвы этому свойству присваивается значение **cmSrcCopy**, которое незримо присутствует при копировании картинок методами **Draw** и **CopyRect**. Когда значение свойства **CopyMode** равняется **cmSrcCopy**, исходная картинка просто замещает собой изображение на канве, то есть это свойство как бы ничем себя и не проявляет.

Но это только один из возможных способов комбинирования изображений. Значений у **CopyMode** целых 15. А всего их в *Windows API* и того больше – аж 256! Называются они **кодами растровых операций**, сокращённо – **ROP**. Растровые операции определяют правила, по которым комбинируются пиксели (а они, в свою очередь, состоят из битов) исходной картинки, фона целевой канвы и узора на нём.

Биты пикселей комбинируются с помощью логических операций **AND, COPY, NOT, OR, XOR**. Их сочетание в **ROP** определяется соответствующим кодом. Как уже говорилось, всего возможно 256 различных сочетаний пикселей источника, приёмника и узора, но практически используются только 15 из них (все они есть в *Delphi*). Для удобства они, кроме кодов, имеют и собственные названия. Дальше вы найдёте их вместе с кратким описанием их действия.

- **cmBlackness** - закрашивает целевой прямоугольник на канве чёрным цветом;
- **cmDstInvert** - инвертирует картинку на канве, игнорируя источник; повторённая дважды, операция сохраняет приёмник без изменений;
- **cmMergeCopy** - комбинирует картинку на канве и исходный растр, используя логическую операцию **AND**; результат – как с растровой операцией *cmSrcCopy*, но копируется узор кисти, если кисть не сплошная;
- **cmMergePaint -** комбинирует инвертированный исходный растр с картинкой на канве, используя логическую операцию **OR**; белый фон становится прозрачным;
- **cmNotSrcCopy** - копирует инвертированный исходный растр на канву;
- **cmNotSrcErase -** комбинирует картинку на канве и исходный растр, используя логическую операцию **OR**, затем результирующая картинка инвертируется;
- **cmPatCopy** - копирует узор кисти на канву;
- **cmPatInvert** - комбинирует узор кисти с картинкой на канве, используя логическую операцию **XOR**; белый фон растра прозрачный, если кисть белая; повторённая дважды, операция сохраняет приёмник без изменений;

- **cmPatPaint** - комбинирует инвертированный исходный растр с узором, используя логическую операцию **OR**, затем комбинирует получившуюся картинку с картинкой на канве, используя ту же логическую операцию;
- **cmSrcAnd** - комбинирует картинку на канве и исходный растр, используя логическую операцию **AND**; белый фон становится прозрачным;

- **cmSrcCopy** - копирует растровую картинку на канву, режим по умолчанию;

- **cmSrcErase** - инвертирует картинку на канве, а затем комбинирует её с исходным растром, используя логическую операцию **AND**; повторённая дважды, операция делает белый фон растра прозрачным;
- **cmSrcInvert** - комбинирует картинку на канве и исходный растр, используя логическую операцию **XOR**; повторённая дважды, операция сохраняет приёмник без изменений;

- **cmSrcPaint** - комбинирует картинку на канве и исходный растр, используя логическую операцию **OR**;

- **cmWhiteness** - закрашивает целевой прямоугольник на канве чёрным цветом.

Конечно, ни по названию, ни даже по описанию невозможно точно представить себе результат действия той или иной растровой операции, поэтому рассмотрим всё это на практике, то есть «программным путём».

Откроем новый проект (назовём его **CopyMode**, не мудрствуя лукаво).

Свойства **формы**:

```
BorderIcons=[biSystemMenu,biMinimize],
BorderStyle=bsSingle,
Caption='  CopyMode',
Height=319, Width=477,
Position=poDesktopCenter.
```

На форме разместим две обязательные кнопки – **sbtClearPole** и **sbtExit**. Первая из них копирует на «страницу» фоновый рисунок из **Image1**:

```
//ОЧИСТИТЬ ПОЛЕ
procedure TForm1.sbtClearPoleClick(Sender: TObject);
begin
```

```
  imgPole.Picture.Bitmap.assign(image1.Picture.Bitmap)
end; //sbtClearPoleClick
```

Вторая – закрывает программу:

```
//ЗАКРЫТЬ ПРОГРАММУ
procedure TForm1.sbtExitClick(Sender: TObject);
begin
  close
end;  //sbtExit
```

А заниматься хитрыми комбинациями будем в комфорте – на канве компонента **ImgPole** типа *TImage*:

```
Cursor=crUpArrow,
Height=200,
Width=365,
Left=4,
Top=36.
```

Фоном для наших росписей будет служить растровая картинка, которую мы загрузим в компонент **Image1**:

```
Stretch=True,
Visible=False.
```

Во время работы программы он будет невидим (как и картинка), так что его размеры и расположение на форме не имеют значения. Саму картинку, которая хранится в свойстве **Picture**, мы будем копировать в компонент **ImgPole**, когда потребуется его очистить от следов наших экспериментов. Под ним разместим 4 картинки, которые будем всячески накладывать на фон, - **ImgBmp1**, **ImgBmp2**, **ImgBmp3**, **ImgBmp4**. Их размеры не очень важны, поэтому можно сделать их квадратными и одинаковыми: **Height**=50, **Width**=50. Это могут быть любые изображения, проще всего их позаимствовать из предыдущего проекта.

Справа от поля для рисования установите список **ListBox1**:

```
  Left = 376,
  Top = 32
  Width = 89,
  Height = 205
  Cursor = crHandPoint
  Hint = 'Режим копирования'
  Items.Strings = (
      'cmBlackness'
      'cmDstInvert'
      'cmMergeCopy'
      'cmMergePaint'
```

```
      'cmNotSrcCopy'
      'cmNotSrcErase'
      'cmPatCopy'
      'cmPatInvert'
      'cmPatPaint'
      'cmSrcAnd'
      'cmSrcCopy'
      'cmSrcErase'
      'cmSrcInvert'
      'cmSrcPaint'
      'cmWhiteness')
  ShowHint = True
  OnClick = ListBox1Click
```

Заполните его в *Редакторе строк* названиями режимов копирования. Теперь, чтобы выбрать любую растровую операцию, достаточно будет щёлкнуть на нужной строке в списке:

```
//ВЫБРАТЬ РАСТРОВУЮ ОПЕРАЦИЮ
procedure TForm1.ListBox1Click(Sender: TObject);
begin
  imgPole.Canvas.CopyMode:= c_mode[ListBox1.ItemIndex];
  imgPole.Canvas.Brush.Style:= bsCross;
end; //ListBox1Click
```

Для перевода номера строки в списке в код операции потребуется глобальный массив констант с кодами операций:

```
const
  c_mode: array[0..14] of integer=(cmBlackness,
                                   cmDstInvert,
                                   cmMergeCopy,
                                   cmMergePaint,
                                   cmNotSrcCopy,
                                   cmNotSrcErase,
                                   cmPatCopy,
                                   cmPatInvert,
                                   cmPatPaint,
                                   cmSrcAnd,
                                   cmSrcCopy,
                                   cmSrcErase,
                                   cmSrcInvert,
                                   cmSrcPaint,
                                   cmWhiteness);
```

В результате должна получиться примерно такая интерфейсная «конструкция» (Рис. П6.25).

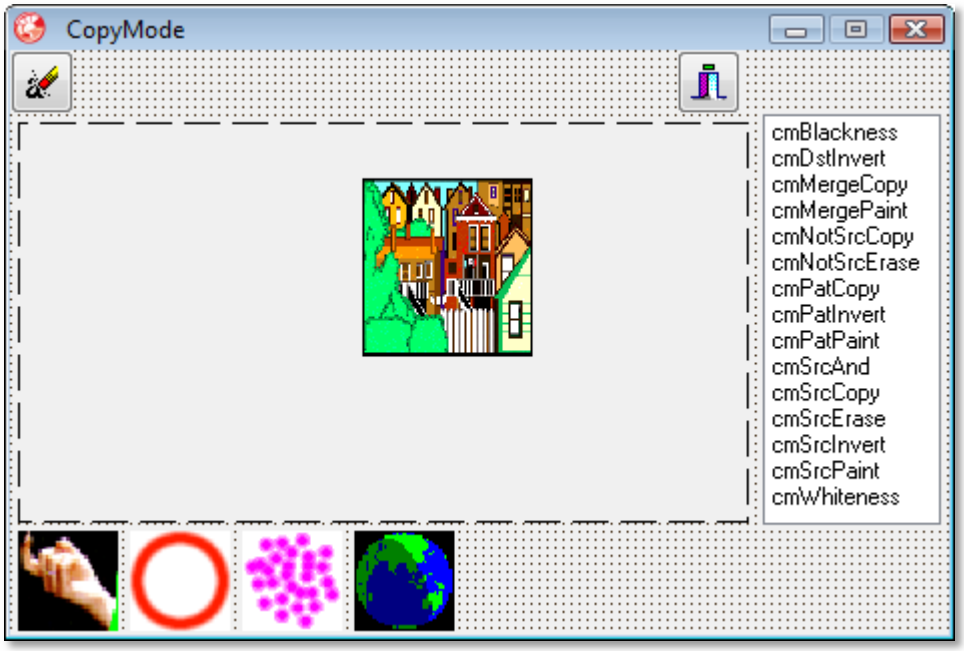


Рис. П6.25. Копировальный интерфейс

Текущая картинка для рисования будет храниться в *переменной*:

```
  private
    { Private declarations }
    bmp_canvas: TCanvas;
```

При запуске программы на чистую страницу переносится фоновый рисунок, выбирается текущая картинка для копирования и устанавливается режим копирования, при котором на страницу выводится чёрный прямоугольник вместо картинки:

```
//СОЗДАТЬ ФОРМУ
procedure TForm1.FormCreate(Sender: TObject);
begin
  //скопировать фон на поле:
  sbtClearPoleClick(Self);
  //картинка по умолчанию:
  bmp_canvas:= imgBmp1.Canvas;
  //установить первый режим копирования - cmBlackness:
  listbox1click(self)
end; //FormCreate
```

Если выбрать последнюю строку списка, то чёрный квадрата будет заменён белым, так что эти два режима для нас не очень интересны.

Чтобы выбрать картинку для копирования, щёлкните на любой из них, и она станет текущей:

```
//ЩЁЛКНУТЬ НА КАРТИНКЕ - ВЫБРАТЬ ЕЁ В КАЧЕСТВЕ КИСТИ
procedure TForm1.ImgBMP1Click(Sender: TObject);
begin
  case (Sender as TImage).tag of
    0: bmp_canvas:= imgBmp1.Canvas;
    1: bmp_canvas:= imgBmp2.Canvas;
    2: bmp_canvas:= imgBmp3.Canvas;
    3: bmp_canvas:= imgBmp4.Canvas;
  end;
end; //ImgBMP1Click
```

Само копирование картинки на поле происходит после нажатия кнопки мыши в любой его точке:

```
//НАЖАТЬ КНОПКУ - СКОПИРОВАТЬ КАРТИНКУ
procedure TForm1.ImgPoleMouseDown(Sender: TObject; Button: TMouse-
Button;
  Shift: TShiftState; X, Y: Integer);
var
  dstRect, srcRect: Trect;
  W, H: integer;
begin
  Drawing:= True;
  srcRect:= bmp_canvas.ClipRect;
  w:= srcRect.Right - srcRect.Left ;
  h:= srcRect.Bottom - srcRect.Top ;
  dstRect:= Rect(X,Y,X+w,Y+h);
  //скопировать картинку в позицию курсора:
  imgPole.Canvas.CopyRect(dstRect, bmp_canvas, srcRect);
end;  //ImgPoleMouseDown
```

Обратите внимание на переменную **Drawing**:

```
  private
    { Private declarations }
    bmp_canvas: TCanvas;
    Drawing: boolean;
```

Когда кнопка мыши нажата (**Drawing**= *True*), вы можете рисовать картинкой как кистью:

```
//ПЕРЕМЕЩАЕМ МЫШЬ - РИСУЕМ КАРТИНКОЙ КАК КИСТЬЮ
procedure TForm1.ImgPoleMouseMove(Sender: TObject; Shift: TShiftState; X,
  Y: Integer);
var
  srcRect, dstRect: Trect;
  w,h: integer;
begin
  if drawing then
  begin
    srcRect:= bmp_canvas.ClipRect;
    w:= srcRect.Right - srcRect.Left ;
```

```
    h:= srcRect.Bottom - srcRect.Top ;
    dstRect:= rect(X,Y,X+w,Y+h);
    //скопировать картинку в позицию курсора:
    imgPole.Canvas.CopyRect(dstRect, bmp_canvas, srcRect);
  end
end; //ImgPoleMouseMove
```

Отпустив мышь на свободу, вы закончите рисование:

```
//ОТПУСТИТЬ КНОПКУ МЫШИ - ЗАКОНЧИТЬ РИСОВАНИЕ
procedure TForm1.ImgPoleMouseUp(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  Drawing:= False;
end; //ImgPoleMouseUp
```

Вот и вся программа! Ничего хитрого, зато она поможет вам разобраться в хитростях взаимодействия картинки, канвы и кисти! Словами это описать невозможно, поэтому посвятите несколько минут экспериментам с режимами копирования (Рис. П6.26). Может быть, вы обнаружите кое-что полезное для своих будущих программ.

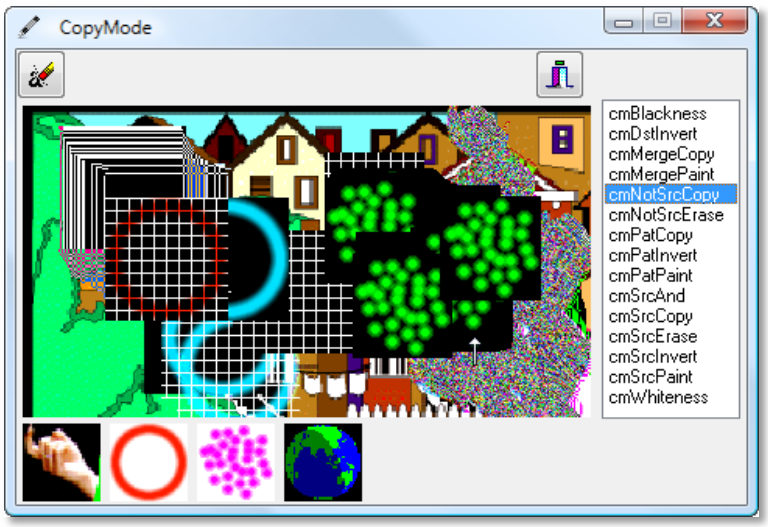


Рис. П6.26. Режимное рисование



Исходный код программы находится в папке **CopyMode**.

Наш аниматограф

Есть кто живой?

Один из коренных русских вопросов

Любой мультфильм состоит из множества кадров, которые последовательно сменяют друг друга. Рисовать их – дело непростое, поэтому часто делают так: герой перемещается, а фон остаётся на месте. Если фон одноцветный (что бывает нечасто), то можно просто закрашивать одно изображение, а затем рисовать следующее. С фактурным фоном всё гораздо сложнее – его нужно восстанавливать в тех местах, где он был закрыт движущимися предметами. При создании мультфильма их рисуют на прозрачной плёнке и накладывают на фон в другом месте. Так как фон при этом не затрагивается, то всё сводится к рисованию фигурок.

У нас целлулоида нет, поэтому попробуем перемещать изображение по фону так, чтобы он не изменялся. Для этого есть несколько способов, но мы сначала попробуем применить наши знание о растровых операциях к созданию простейшей динамической картинки.

За основу возьмём предыдущий проект, в котором изменим парочку процедур:

```
//НАЖАТЬ КНОПКУ - СКОПИРОВАТЬ КАРТИНКУ
procedure TForm1.ImgPoleMouseDown(Sender: TObject; Button: TMouse-
Button;
  Shift: TShiftState; X, Y: Integer);
var
  dstRect, srcRect: Trect;
  W, H: integer;
begin
  Drawing:= True;
  Origin := Point(X, Y);
  MovePt := Origin;
  srcRect:= bmp_canvas.ClipRect;
  w:= srcRect.Right - srcRect.Left ;
  h:= srcRect.Bottom - srcRect.Top ;
  dstRect:= rect(MovePt.X,MovePt.Y,MovePt.X+w,MovePt.Y+h);
  //скопировать картинку в позицию курсора:
  imgPole.Canvas.CopyRect(dstRect, bmp_canvas, srcRect);
end;

//ПЕРЕМЕЩАЕМ МЫШЬ - РИСУЕМ КАРТИНКОЙ, КАК КИСТЬЮ
procedure TForm1.ImgPoleMouseMove(Sender: TObject; Shift:
TShiftState; X,
  Y: Integer);
```

```
var
  srcRect, dstRect: Trect;
  w,h: integer;
begin
  if drawing then
  begin
    srcRect:= bmp_canvas.ClipRect;
    w:= srcRect.Right - srcRect.Left ;
    h:= srcRect.Bottom - srcRect.Top ;
    dstRect:= rect(MovePt.X,MovePt.Y,MovePt.X+w,MovePt.Y+h);
    //скопировать картинку в позицию курсора:
    imgPole.Canvas.CopyRect(dstRect, bmp_canvas, srcRect);
    MovePt := Point(X, Y);
    dstRect:= rect(MovePt.X,MovePt.Y,MovePt.X+w,MovePt.Y+h);
    imgPole.Canvas.CopyRect(dstRect, bmp_canvas, srcRect);
  end
end;
```

При перемещении выбранного рисунка по фону он дважды выводится в одно и то же место, поэтому при операциях **cmDstInvert**, **cmPatInvert**, **cmSrcInvert** фон не изменяется, а сама картинка действует как лупа – только она не увеличивает рисунок фона, а инвертирует его цвета. Эффект, конечно, интересный и где-то может и пригодиться, но для анимации он не подходит. Если же вместо картинки взять чёрный фон, то с помощью операций **cmSrcErase** и **cmSrcInvert** можно получить очень хорошее движущееся изображение глобуса. Из этого следует, что, прежде чем накладывать картинку на фон, нужно подложить под неё чёрную маску, тогда картинка сохранит все свои цвета. Примерно так получается прозрачное изображение значка **TIcon**. Но это уже совсем другая история, поэтому мы лучше рассмотрим очень любопытный пример «анимации» линиq, который сослужит вам хорошую службу во многих программах.



Исходный код программы находится в папке **Animation**.

Движущиеся линии, или «Классная» программа

Иа-Иа продолжал стоять,
понуро глядя в землю,
и Винни-Пух обошел вокруг него.
-- Ой, что это случилось с твоим хвостом? --
спросил он удивленно.
-- А что с ним случилось? -- сказал Иа-Иа.
-- Его нет!
-- Ты не ошибся?
-- Хвост или есть, или его нет.
По-моему, тут нельзя
ошибиться. А твоего хвоста нет.
-- А что же тогда там есть?
-- Ничего.

Алан Милн. *Винни-Пух и Все-Все-Все*

Зрелище вас и вправду ожидает великолепное, но программа не только поэтому называется классной – в ней мы напишем парочку классов для рисования точек и фигур. Напомним, что классом в *Delphi* называется тип данных, описывающий какой-либо объект – форму, компонент, карточную колоду или геометрическую фигуру…

Чтобы создать **объект**, или **экземпляр** класса, нужно объявить *переменную* этого типа. Так, каждая форма проекта начинается с описания типа формы:

```
type
  TForm1 = class(TForm)
 …
end;
```

А затем вводится переменная этого типа:

```
var
  Form1: TForm1;
```

Описание класса содержит поля, методы и свойства. В полях хранятся данные, характеризующие конкретный экземпляр класса – его размеры, положение, цвет … Эти данные обычно отличаются у разных экземпляров класса (представьте себе, что вы установили на форме несколько кнопок, - у них могут быть разные размеры, надписи, подсказки …).

Методы – это процедуры и функции, которые оперируют с полями класса, поэтому у всех экземпляров класса методы **одинаковые**.

Чтобы создать любой объект, нужно вызвать его метод **Create**, или **конструктор**, в котором также производится его инициализация. К примеру, если вы кликнете на значке компонента в *Палитре*, то он появится в центре формы с размерами и другими свойствами, заданными в его конструкторе (их легко увидеть в исходном коде компонента; вы можете открыть его в *Редакторе кода*, если установите курсор мыши на названии типа компонента в описании формы, нажмёте правую кнопку мыши и в открывшемся меню выберете пункт *Find Declaration*: `sbtLine: TSpeedButton;`).

Чтобы нам было удобнее работать с классами, отведём под каждый из них отдельный модуль. В первом объявим класс **TOnePoint**, который описывает точку в трёхмерном пространстве. Вообще-то у нас все точки будут находиться на плоскости, но этот класс пригодится нам для задания случайного цвета линий. Любой оттенок можно представить себе как точку в трёхмерном пространстве **RGB**, поэтому по случайным значениям трёх составляющих цвета **R, G, B** мы получим один из возможных цветов. Аналогично этот класс используется и в некоторых других случаях.

```
unit PointUnit;

interface
uses
  Windows, Classes;

type T3DPoint= record
  x: integer;
  y: integer;
  z: integer;
end;

type TOnePoint= class
  pt: T3DPoint;
    constructor create;
  public
    function PRandomize(minX, maxX,minY, maxY, minZ, maxZ: inte-
ger): T3DPoint;
end;

implementation

//СОЗДАТЬ ТОЧКУ
constructor TOnePoint.create;
begin
  pt.x:=0;  pt.y:=0;  pt.z:=0;
end;

//ВЫЧИСЛИТЬ СЛУЧАЙНЫЕ КООРДИНАТЫ ТОЧКИ
```

```
function TOnePoint.PRandomize(minX, maxX,minY, maxY, minZ, maxZ:
integer): T3DPoint;
begin
  pt.x:= Random(maxX-minX+1)+minX;
  pt.y:= Random(maxY-minY+1)+minY;
  if minZ<> maxZ then pt.z:= Random(maxZ-minZ+1)+minZ;
end;

end.
```

Чтобы вам стало понятно, о чём мы, собственно, говорим, сразу посмотрите на окно работающей программы (Рис. П6.27).

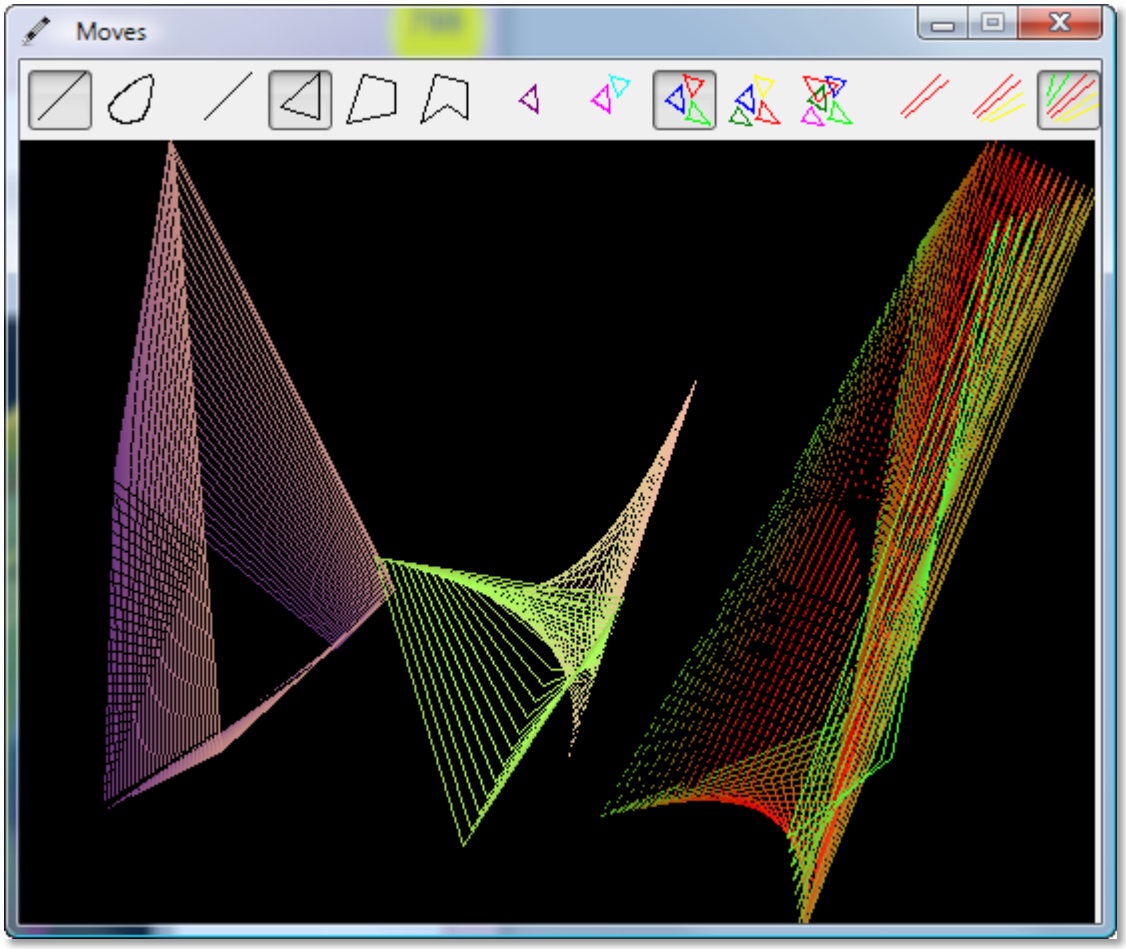


Рис. П6.27. Движущиеся треугольники

На чёрном фоне перемещаются фигуры одного из следующих видов:

- прямые линии,
- треугольники,
- четырёхугольники,
- пятиугольники,
- кривые Безье.

Разновидность фигур можно задать первыми двумя кнопками на форме. Если утоплена первая, то можно выбрать любую из четырёх «прямолинейных» фигур, если вторая – то на экране будут кривые Безье (они очень простые, так как задаются только тремя точками). Число фигур на экране - от одной до пяти – определяется состоянием пяти кнопок с треугольничками.

Каждая из фигур последовательно рисуется со смещением координат и цвета линий, поэтому за ней как бы остаётся шлейф, длину которого можно изменить с помощью трёх следующих кнопок. Например, если длина шлейфа равна 10, то сначала рисуется фигура в начальном, произвольном положении на чёрном поле, цвет линий также устанавливается произвольно. Затем вычисляются смещённые координаты фигуры и цвет линий, а фигура выводится в новом положении. И так до тех пор, пока на экране не окажется 10 фигур. Тогда последняя фигура в шлейфе (самая «старая») стирается и рисуется новая – но не на её месте, конечно, а по ходу движения. Таким образом, «голова» и «хвост» шлейфа перемещаются по экрану, а его длина остаётся постоянной. При движении линии изменяют не только положение, но и величину и цвет, а также отражаются от границ поля. Всё вместе это создаёт настоящее цветовое шоу (сейчас всё – «шоу»!).

Для описания фигур введём новый класс (не забудьте отвести под него отдельный модуль) **TCluster**:

```
unit ClusterUnit;

interface

uses
  Windows, Classes, PointUnit, SysUtils, Graphics;

const
  //смещение координат вершин фигуры:
  VELOCITY_RANGE=8;
  //смещение цвета линий:
  COLOUR_VELOCITY_RANGE=5;

//вид фигур - из отрезков прямых или кривых Безье:
type TClusterType=(TypeLine, TypeBezier);

//фигура:
type TCluster=class
  //вид фигур:
  _clusterType: TClusterType;
  //число точек в фигуре:
  _points: integer;
  //цвет линий:
  _colour: TOnePoint;
```

```
  //смещение цвета точек:
  _colourVelocity: TOnePoint;
  //высота поля:
  _height: integer;
  //ширина поля:
  _width: integer;
  //длина шлейфа:
  _trailLength: integer;
  //индексы в буфере:
  _trailReadIndex: integer;
  _trailCurrIndex: integer;
  _trailWriteIndex: integer;
  //указывает заполнение буфера:
  _trailBufferFull: boolean;
  //буфер с координатами точек фигуры:
  _Buffer: array[0..5*31-1] of TPoint;
  //смещение координат точек фигуры:
  _velocity: array[0..4]  of TOnePoint;

  constructor Create(ClusterType: TClusterType; points, length,
width, height: integer);
  procedure Move;
  procedure Draw;
  procedure DrawCluster(p: array of TPoint; sm, points: integer);
end;
```

В нём вы найдёте всё то, о чём мы только что говорили, плюс буфер для хранения координат точек фигуры **_Buffer** и индексов в нём, - но о них немного позже.

При создании фигуры в конструктор **Create** передаётся её вид («прямолинейная» или кривая Безье), число точек в фигуре, длина шлейфа и размеры поля. Эти значения копируются в соответствующие поля, обнуляются указатели в буфере координат, а затем в него заносятся случайные координаты **x, y** точек фигуры, а также вычисляются случайные значения для смещения координат, для цвета линий и изменения цвета при движении:

```
implementation

uses
  MovesUnit;

//СОЗДАТЬ ФИГУРУ
constructor TCluster.Create(ClusterType: TClusterType;
              points, length, width, height: integer);
var i: integer;
begin
  //вид фигур:
  _clusterType:= ClusterType;
  //число точек в фигуре:
```

```
  _points:= points;
  //высота поля:
  _height:= height;
  //ширина поля:
  _width:= width;
  //длина шлейфа:
  _trailLength:= length*points;
  //указатели в буфере координат:
  _trailReadIndex:=0;
  _trailWriteIndex:=0;
  _trailCurrIndex:=0;
  //буфер пуст:
  _trailBufferFull:= FALSE;

  //назначить случайные координаты и величину смещения фигур:
  for i:=0 to points-1 do
  begin
    //заносим в буфер:
    inc(_trailWriteIndex);
    _Buffer[i].x:= Random(width);
    _Buffer[i].y:= Random(height);
    _velocity[i]:= TOnePoint.create;
    _velocity[i].PRandomize(-VELOCITY_RANGE, VELOCITY_RANGE,
                            -VELOCITY_RANGE, VELOCITY_RANGE,0,0);
  end;

  //цвет линий:
  _colour:= TOnePoint.create;
  _colour.PRandomize(0,255,0,255,0,255);
  //задать смещение цвета:
  _colourVelocity:= TOnePoint.create;
  _colourVelocity.PRandomize(-COLOUR_VELOCITY_RANGE,
COLOUR_VELOCITY_RANGE,
                             -COLOUR_VELOCITY_RANGE,
COLOUR_VELOCITY_RANGE,
                             -COLOUR_VELOCITY_RANGE,
COLOUR_VELOCITY_RANGE);
end;  //Create
```

Теперь займёмся конструированием формы программы.

Размеры формы вы можете взять любыми, но не стоит делать её слишком большой:

```
    Width = 548
    Height = 465
    BorderIcons = [biSystemMenu, biMinimize]
    Caption = '  Moves'
    Position = poDesktopCenter
    OnClose = FormClose
    OnCreate = FormCreate
```

Для рисования фигур потребуется поле **imgPole** типа *Timage*, которое занимает почти всю форму:

```
    Left = 0
    Top = 40
    Width = 537
    Height = 397
```

Первые две **кнопки**, переключают вид фигур:

sbtLine: *TSpeedButton* (все остальные кнопки того же типа):

```
    Left = 4
    Top = 4 (все кнопки находятся на одном уровне)
    Width = 32
    Height = 32 (все кнопки одного размера)
    Cursor = crHandPoint (курсор у всех кнопок одинаковый)
    Hint = 'Прямые линии'
    GroupIndex = 1 (первые кнопки работают в паре, поэтому значение этого свойства у них одинаково; из двух кнопок нажатой может быть только одна)
    Down = True (по умолчанию рисуются прямые линии)
    Flat = True (все кнопки плоские)
    ShowHint = True
    OnClick = sbtLineClick
```

sbtBezier:

```
    Left = 40
    Hint = 'Кривые Безье'
    GroupIndex = 1
    OnClick = sbtLineClick (нажатие на обе кнопки обрабатывается в одной процедуре)
```

Пора написать немного кода. Сначала объявите несколько констант и переменных, необходимых для работы программы. В данной версии скорость движения фигур не изменяется, но для вас оставлена такая возможность:

```
unit MovesUnit;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
Dialogs,
  StdCtrls,ClusterUnit, PointUnit, ExtCtrls, ComCtrls, Buttons;

//type Speed=(SpeedSlower, SpeedSlow, SpeedFast, SpeedFaster);
```

```
const
  //скорость движения фигур:
  SpeedSlower=250;
  SpeedSlow=125;
  SpeedFast=70;
  SpeedFaster= 10;
  //длина шлейфа:
  TrailShort=5;
  TrailLong=15;
  TrailLonger=30;

var
  //длина шлейфа:
  trailLength:  array[0..2]  of  integer=(TrailShort,  TrailLong,
TrailLonger);
  //задержка межу обновлениями позиций:
  _speed: integer;
  //число точек в фигуре:
  _points: integer;
  //число фигур:
  _clusters: integer;
  //длина шлейфа:
  _trailLength: integer;
  //тип фигур:
  _clusterType: TClusterType;
  //скорость движения фигур:
  //Speed: array[1..4] of integer= (SpeedSlower, SpeedSlow, Speed-
Fast, SpeedFaster);
  _clusterList: array[1..5] of TCluster;
```

При запуске приложения устанавливаются значения по умолчанию и создаются фигуры – сначала это одна прямая линия:

```
//СОЗДАТЬ ФОРМУ
procedure TForm1.FormCreate(Sender: TObject);
begin
  randomize;
  //по умолчанию чертим прямые линии:
  _clusterType:= TypeLine;
  _points:= 2;
  //число фигур:
  _clusters:= 1;
  //длина шлейфа:
  _trailLength:= TrailShort;
  //запустить генератор фигур:
  MistifyCreate;
end;  //FormCreate
```

В процедуре **MistifyCreate** создаются все фигуры заданного вида. Толщина линий устанавливается в 1 пиксель, но вы можете сделать их «солиднее». Поле заливается чёрным цветом – на нём цветные линии выглядят кон-

трастнее. Скорость передвижения фигур всегда выбирается максимальной. И, наконец, создаётся нужное число фигур и запускается таймер (не забудьте установить компонент **timer1**).

```
//СОЗДАТЬ ФИГУРЫ
procedure TForm1.MistifyCreate;
var
  i: integer;
begin
  timer1.Enabled:= FALSE;
  //толщина линий фигур:
  imgPole.Canvas.Pen.Width:=1;
  //фон - чёрный:
  imgPole.Canvas.Brush.Color := clBlack;
  imgPole.Canvas.FillRect(imgPole.ClientRect);
  //задержка между обновлениями позиций:
  _speed:= SpeedFast;
  //создать массив (список) фигур:
  for i:=1 to _clusters do
    _clusterList[i]:=Tcluster.create(_clusterType, _points,
_trailLength, imgPole.width, imgPole.Height);
  //настраиваем и включаем таймер:
  timer1.Interval:= _speed;
  timer1.Enabled:= TRUE;
end; //MistifyCreate
```

Теперь вы с лёгкостью поймёте, как работают первые кнопки:

```
//ВЫБРАТЬ ФОРМУ ЛИНИЙ - ПРЯМЫЕ ЛИНИИ ИЛИ КРИВЫЕ БЕЗЬЕ
procedure TForm1.sbtLineClick(Sender: TObject);
begin
  if sbtLine.Down then  //- прямые
  begin
    _clusterType:= TypeLine;
    sbtP2.Enabled:= TRUE;
    sbtP3.Enabled:= TRUE;
    sbtP4.Enabled:= TRUE;
    sbtP5.Enabled:= TRUE;
    //режим по умолчанию:
    sbtP2.down:= TRUE;
    //число точек в фигуре:
    _points:= 2;
  end
  else                  //- кривые Безье
  begin
    _clusterType:= TypeBezier;
    sbtP2.Enabled:= FALSE;
    sbtP3.Enabled:= FALSE;
    sbtP4.Enabled:= FALSE;
    sbtP5.Enabled:= FALSE;
    //кривые Безье задаются тремя точками:
```

```
    _points:=3;
  end;
  //создать фигуры:
  MistifyCreate;
end;  //sbtLineClick
```

Следующие 4 кнопки также работают в одной упряжке и позволяют вам выбрать разновидность фигур, образованных **прямыми линиями**.

sbtP2:

```
    Tag = 2 (это свойство указывает число точек, необходимых для рисования фигуры)
    Left = 88
    Hint = 'Линии'
    GroupIndex = 2
    Down = True
    OnClick = sbtP2Click
```

sbtP3:

```
    Tag = 3
    Left = 124
    Hint = 'Треугольники'
    GroupIndex = 2
    OnClick = sbtP2Click (нажатие всех кнопок обрабатывается в одной процедуре)
```

sbtP4:

```
    Tag = 4
    Left = 160
    Hint = 'Четырёхугольники'
    GroupIndex = 2
    OnClick = sbtP2Click
```

sbtP5:

```
    Tag = 5
    Left = 196
    Hint = 'Пятиугольники'
    GroupIndex = 2
    OnClick = sbtP2Click
```

По значению свойства **tag** мы определяем, сколько точек необходимо для построения фигуры и создаются новые фигуры:

```
//ЗАДАТЬ ЧИСЛО ВЕРШИН В ФИГУРАХ
procedure TForm1.sbtP2Click(Sender: TObject);
begin
```

```
  _points:=(Sender as TSpeedButton).tag;
  MistifyCreate;
end; //sbtP2Click
```

Следующие пять **кнопок** отвечают за количество фигур на поле.

sbtC1:

Tag = 1 (количество фигур)
Left = 244
Hint = 'Одна фигура'
GroupIndex = 3
Down = True (по умолчанию выводится одна фигура)
OnClick = sbtC1Click (нажатие всех кнопок обрабатывается в этой процедуре)

sbtC2:

Tag = 2
Left = 280
Hint = 'Две фигуры'
GroupIndex = 3

sbtC3:

Tag = 3
Left = 316
Hint = 'Три фигуры'

sbtC4:

Tag = 4
Left = 352
Hint = 'Четыре фигуры'

sbtC5:

Tag = 5
Left = 388
Hint = 'Пять фигур'

```
//ЗАДАТЬ ЧИСЛО ФИГУР 1..5
procedure TForm1.sbtC1Click(Sender: TObject);
begin
  _clusters:= (Sender as TSpeedButton).tag;
  MistifyCreate;
end; //sbtC1Click
```

Последние три **кнопки** управляют длиной шлейфа.

sbtT1:

```
    Left = 436
    Hint = 'Короткий шлейф'
    GroupIndex = 4
    Down = True
    OnClick = sbtT1Click
```

sbtT2:

```
    Left = 472
    Hint = 'Средний шлейф'
    GroupIndex = 4
    OnClick = sbtT1Click
```

sbtT3:

```
    Left = 508
    Hint = 'Длинный шлейф'
    GroupIndex = 4
    OnClick = sbtT1Click
```

```
//ЗАДАТЬ ДЛИНУ ШЛЕЙФА: КОРОТКИЙ, СРЕДНИЙ, ДЛИННЫЙ
procedure TForm1.sbtT1Click(Sender: TObject);
begin
  _trailLength:= trailLength[(Sender as TSpeedButton).tag];
  MistifyCreate;
end; //sbtT1Click
```

Как вы помните, после создания фигур запускается таймер. Через заданный промежуток времени вызывается процедура:

```
//ТАЙМЕР
procedure TForm1.Timer1Timer(Sender: TObject);
begin
  MistifyMoveClusters;
end; //Timer1Timer
```

в которой фигуры перемещаются в новое положение:

```
//ПЕРЕМЕЩЕНИЕ ФИГУР
procedure TForm1.MistifyMoveClusters;
var
  i: integer;
begin
  for i:=1 to _clusters do
  begin
    _clusterList[i].Move;
    _clusterList[i].draw;
  end;
end; //MistifyMoveClusters
```

При выходе из программы таймер выключается:

```
//ЗАКРЫТЬ ПРОГРАММУ
procedure TForm1.FormClose(Sender: TObject; var Action: TCloseAc-
tion);
begin
  timer1.Enabled:= FALSE;
end; //FormClose
```

На этом все процедуры главного модуля исчерпаны, и мы возвращаемся в модуль **ClusterUnit**, в котором нам нужно написать две важные процедуры.

В первой вычисляются новые координаты фигуры и цвет линий:

```
//ВЫЧИСЛИТЬ НОВЫЕ КООРДИНАТЫ И ЦВЕТ ФИГУР
procedure TCluster.Move;
var i: integer;
begin
  //передвинуть все точки фигуры:
  for i:= 0 to _points-1 do
  begin
    //новые координаты точки:
    _Buffer[_trailWriteIndex+i].x :=
      _Buffer[_trailCurrIndex+i].x + _velocity[i].pt.x;
    _Buffer[_trailWriteIndex+i].y :=
      _Buffer[_trailCurrIndex+i].y + _velocity[i].pt.y;
    //проверить новые координаты:
    if _Buffer[_trailWriteIndex+i].x > _width then
    begin
      _Buffer[_trailWriteIndex+i].x:= _width;
      _velocity[i].pt.x := - Random(VELOCITY_RANGE)-1
    end
    else if _Buffer[_trailWriteIndex+i].x < 0 then
    begin
      _Buffer[_trailWriteIndex+i].x := 0;
      _velocity[i].pt.x := Random(VELOCITY_RANGE)+1
    end;
    if _Buffer[_trailWriteIndex+i].y > _height then
    begin
      _Buffer[_trailWriteIndex+i].y := _height;
      _velocity[i].pt.y := - Random(VELOCITY_RANGE)-1
    end
    else if _Buffer[_trailWriteIndex+i].y < 0 then
    begin
      _Buffer[_trailWriteIndex+i].y := 0;
      _velocity[i].pt.y := Random(VELOCITY_RANGE)+1
    end;
    //изменить цвет линий:
    _colour.pt.x:= _colour.pt.x + _colourVelocity.pt.x;
    _colour.pt.y:= _colour.pt.y + _colourVelocity.pt.y;
    _colour.pt.z:= _colour.pt.z + _colourVelocity.pt.z;
```

```
    //проверка значений составляющих цвета:
    if _colour.pt.x > 255 then
    begin
      _colour.pt.x := 255;
      _colourVelocity.pt.x:= - Random(COLOUR_VELOCITY_RANGE)-1
    end
    else if _colour.pt.x < 0 then
    begin
      _colour.pt.x := 0;
      _colourVelocity.pt.x:= Random(COLOUR_VELOCITY_RANGE)+1
    end;
    if _colour.pt.y > 255 then
    begin
      _colour.pt.y := 255;
      _colourVelocity.pt.y:= - Random(COLOUR_VELOCITY_RANGE)-1
    end
    else if _colour.pt.y < 0 then
    begin
      _colour.pt.y := 0;
      _colourVelocity.pt.y:= Random(COLOUR_VELOCITY_RANGE)+1
    end;
    if _colour.pt.z > 255 then
    begin
      _colour.pt.z := 255;
      _colourVelocity.pt.z:= - Random(COLOUR_VELOCITY_RANGE)-1
    end
    else if _colour.pt.z < 0 then
    begin
      _colour.pt.z := 0;
      _colourVelocity.pt.z:= Random(COLOUR_VELOCITY_RANGE)+1
    end
  end;
  //новый текущий индекс в буфере:
  _trailCurrIndex:= _trailWriteIndex;
  //новый индекс для записи в буфер:
  _trailWriteIndex:= _trailWriteIndex + _points;
  //скорректировать индекс для записи в буфер, если он заполнен:
  if _trailWriteIndex > _trailLength then
  begin
    _trailWriteIndex:= 0;
    _trailBufferFull:= TRUE
  end;
end; //Move
```

Во второй фигура выводится в новом положении:

```
//ПЕРЕРИСОВАТЬ ФИГУРУ
procedure TCluster.Draw;
begin
  //стереть фигуру в предыдущем положении:
  if _trailBufferFull then
  begin
```

```
    form1.imgPole.Canvas.Pen.Color:= clBlack;
    DrawCluster(_Buffer,_trailReadIndex, _points);
    _trailReadIndex:= _trailReadIndex + _points;
    if _trailReadIndex > _trailLength then _trailReadIndex:=0;
  end;
  //нарисовать фигуру в текущем положении:
  form1.imgPole.Canvas.Pen.Color:= RGB(_colour.pt.x, //цвет линий
                                       _colour.pt.y,
                                       _colour.pt.z);
  DrawCluster(_Buffer,_trailCurrIndex, _points);
end; //Draw
```

Если шлейф достиг заданной длины (*trailBufferFull= True*), то самая «старая» фигура в шлейфе стирается.

```
//НАРИСОВАТЬ ФИГУРУ
procedure TCluster.DrawCluster(p: array of TPoint; sm, points: in-
teger);
var
  i,j: integer;
  _start, _end: TPoint;
begin
  j:= sm;
  case _ClusterType  of
    TypeLine:  //- фигура из прямых линий
      begin
        for i:= 0 to _points-1 do
        begin
          _start:= Point(p[i+j].x, p[i+j].y);
          if i< _points-1 then
            _end:= Point(p[i+1+j].x, p[i+1+j].y)
          else
            _end:= Point(p[j+0].x, p[j+0].y);
          form1.imgPole.Canvas.MoveTo(_start.x, _start.y);
          form1.imgPole.Canvas.LineTo(_end.x, _end.y);
        end;
      end;
    Typebezier:              //- кривые Безье
      with Form1.imgPole do
        Canvas.PolyBezier([Point(p[j].x, p[j].y),
                           Point(p[j+1].x, p[j+1].y),
                           Point(p[j+2].x, p[j+2].y),
                           Point(p[j].x, p[j].y)]);
  end;
end; //DrawCluster
```

Необходимость в буфере координат объясняется просто – как только шлейф достигает заданного размера, нужно стирать фигуру в «хвосте» шлейфа. Так как все фигуры выводятся на чёрный фон, то достаточно нарисовать фигуру в том же месте чёрным пером. А для этого мы должны знать координаты «фигурообразующих» точек.

При создании фигуры все индексы устанавливаются в **0**, то есть указывают на начало буфера (Рис. П6.28).

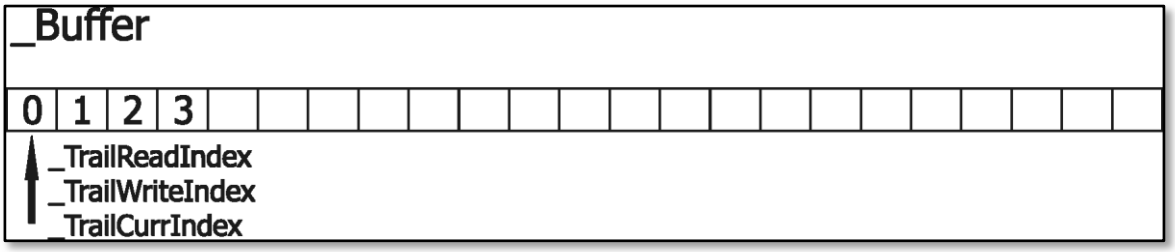


Рис. П6.28. Начальный буфер

Затем случайным образом выбираются точки для построения фигуры. Рассмотрим это на примере прямой линии. Чтобы однозначно определить линию, нужны две её точки. В процедуре **Create** координаты этих двух точек помещаются в буфер, а указатель **_trailWriteIndex** перемещается на два индекса вправо (Рис. П6.29).

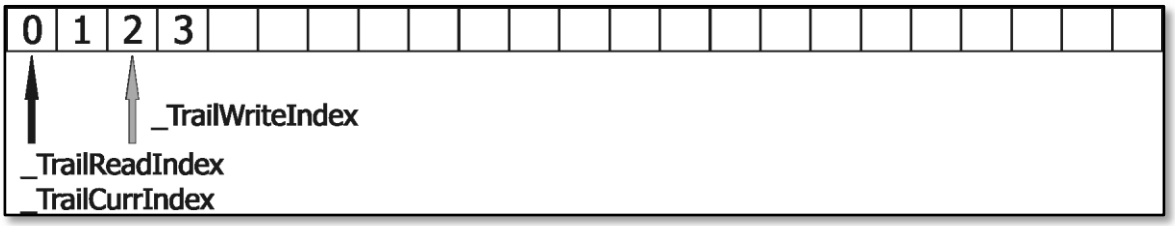


Рис. П6.29. Точка - в буфере

Таким образом, этот индекс всегда указывает на то место в буфере, куда будут записываться координаты очередной фигуры.

В процедуре **Move** вычисляются и записываются в буфер координаты следующей фигуры, при этом указатель смещается ещё на две позиции, а текущим индексом **_trailCurrIndex** становится предыдущий указатель **_trailWriteIndex** (Рис. П6.30).

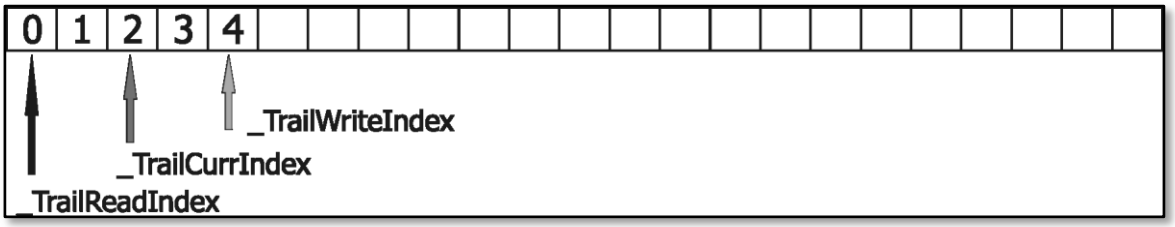


Рис. П6.30. Фигура - в буфере

Как только в буфер будут записаны координаты заданного количества фигур, определяемого длиной шлейфа и числом точек в фигуре, указатель **_trailWriteIndex** снова будет установлен на начало буфера, а переменная

_trailBufferFull принимает значение *TRUE* и сигнализирует о том, что шлейф достиг заданной длины. В этом случае перед выводом новой фигуры в процедуре **Draw** сначала стирается самая последняя фигура в шлейфе. На её координаты указывает индекс **_trailReadIndex**, который затем перемещается на две позиции вправо – к следующей «престарелой» фигуре. На координаты очередной фигуры для вывода на экран (голова шлейфа) указывает индекс **_trailCurrIndex**.

Таким образом, все три указателя циклически перемещаются по буферу. По индексу **_trailWriteIndex** будут записываться координаты следующей фигуры. Индекс **_trailCurrIndex** указывает на координаты текущей фигуры для вывода в голову шлейфа, а **_trailReadIndex** указывает на координаты стираемой фигуры.



Исходный код программы находится в папке **Moves1**.

Если вы запустите программу и **внимательно** понаблюдаете за движением фигур, то скоро обнаружите небольшой изъян в этом способе «анимации». Пока фигура движется в середине поля, всё нормально, но после её отражения от границы поля стираемые линии портят часть линий в теле шлейфа, так как они накладываются друг на друга. Часть цветных точек заменяется чёрными, так что в фигуре образуются «потёртости». Положение ещё более усугубляется, если на поле дрейфует несколько сложных фигур. В результате они стирают не только себя, но и ни в чём неповинных сограждан. Конечно, ничего криминального в таком поведении нет, но эстетическое восприятие страдает. Ну что ж, попробуем воспользоваться другим методом «анимации» - будем целиком формировать картинку в памяти компьютера, а затем переносить её на экран. В этом случае все линии будут только рисоваться, но не стираться, поэтому никто не пострадает!

Для начала перепишите все файлы проекта в новую папку, а затем мы внесём необходимые изменения в программу. Постараемся сделать это с наименьшими потерями для кода (это значит, что вы можете его улучшить, забыв о наследии прошлого).

Модуль **PointUnit** можно оставить без изменений, а в модуле **MovesUnit** потребуются небольшие изменения.
На этот раз мы обойдёмся без таймера, и вы можете удалить его с формы. Это позволит увеличить скорость движения фигур, поэтому нужно изменить несколько *констант*:

```
const
  //скорость движения фигур:
  {SpeedSlower=250;
  SpeedSlow=125;
  SpeedFast=70;
  SpeedFaster= 10;}
  //длина шлейфа:
  TrailShort=12;
  TrailLong= 40;
  TrailLonger= 70;
```

и добавить процедуру:

```
//ЗАДЕРЖКА
procedure Delay(MSecs: Longint);
var
  FirstTickCount, Now: Longint;
begin
  FirstTickCount := GetTickCount;
  repeat
    Application.ProcessMessages;
    Now := GetTickCount;
  until (Now - FirstTickCount >= MSecs) or (Now < FirstTickCount);
end; //Delay
```

Потребуются также две глобальные *переменные*:

```
  bmp: TBitmap;
  flgStop: Boolean= False;
```

В первой мы будем формировать картинку с последующим выводом на экран, вторая будет сигнализировать о вашем желании закрыть программу.

В процедуру **FormCreate** следует добавить код, создающий растр в памяти компьютера по размерам поля:

```
  bmp:= TBitmap.create;
  bmp.Height:= imgPole.Height;
  bmp.Width:= imgPole.Width;
```

Процедуру закрытия программы придётся переписать заново. Теперь в ней из памяти удаляется растр и устанавливается флаг остановки программы:

```
//ЗАКРЫТЬ ПРОГРАММУ
procedure TForm1.FormClose(Sender: TObject; var Action: TCloseAc-
tion);
```

```
begin
  bmp.Free;
  flgStop:= True;
end; //FormClose
```

Так как таймер теперь отсутствует, то фигуры будут стартовать в процедуре активации формы:

```
//ЗАПУСТИТЬ ДВИЖЕНИЕ ФИГУР
procedure TForm1.FormActivate(Sender: TObject);
begin
  MistifyMoveClusters;
end;  //FormActivate
```

Иными словами, здесь управление передаётся процедуре **MistifyMoveClusters**, в которой фигуры перемещаются в новое положение:

```
//ПЕРЕМЕЩЕНИЕ ФИГУР
procedure TForm1.MistifyMoveClusters;
var
  i: integer;
  tick, t: cardinal;
  max: cardinal;
begin
  max:=0;
  //окрасить вторичную поверхность в чёрный цвет:
  bmp.Canvas.Brush.Color:= clBlack;

  //перерисовывать фигуры в новом положении,
  //пока не установлен флаг flgStop:
  repeat
    bmp.Canvas.FillRect(clientRect);
    for i:=1 to _clusters do
      begin
        _clusterList[i].Move;
        _clusterList[i].draw;
      end;
    imgPole.Picture.Bitmap.Assign(bmp);
    imgPole.Invalidate;}
    //небольшая задержка:
    delay(1);
  until flgStop= True;
end; //MistifyMoveClusters
```

Движение фигур теперь будет организовано иначе. Они будут выводиться до тех пор, пока пользователь не закроет программу (при этом флаг **flgStop** устанавливается в *TRUE*). Скорость движения определяется задержкой вывода фигур в новом положении:

```
delay(1);
```

Если вы хотите замедлить движение фигур, то увеличьте число в скобках.

Вся картинка формируется в модуле **ClusterUnit**, а затем переносится на канву поля, поэтому здесь нас ждёт большая работа! Сначала подправьте значения констант:

```
const
  //смещение координат вершин фигуры:
  VELOCITY_RANGE= 6;
  //смещение цвета линий:
  COLOUR_VELOCITY_RANGE=3;
```

В описании типа формы нужно увеличить размер буфера соответственно длине шлейфа (лучше сразу сделать его больше, чтобы не исправлять значение верхнего предела массива при изменении длины шлейфа). А также добавьте новый буфер – для хранения цвета каждой фигуры. Это нужно потому, что раньше нам не нужно было знать цвет каждой фигуры, ведь мы рисовали только последнюю из них. А теперь нам всякий раз наам придётся заново перерисовывать все фигуры в шлейфе, и без знания их цвета никак не обойтись!

```
  //буфер с координатами точек фигуры:
  _Buffer: array[0..999] of TPoint;
  //буфер, в котором хранится цвет линий:
  _Color: array[0..999] of TColor;
```

В процедуру **Move** после проверки значений составляющих цвета вставьте строку:

```
  //запомнить цвет фигуры:
  _Color[trailWriteIndex] := rgb(colour.pt.x,_colour.pt.y,
_colour.pt.z);
```

В неё мы и будем заносить данные о цвете в массив **_Color**.

Рисование фигур по-новому осуществляется в процедуре **Draw** (Рис. П6.31). Начинаем рисование с последней фигуры в шлейфе и последовательно переходим к начальной. Цвет фигуры берём из массива **_Color**, а координаты, как и раньше, из буфера.

```
//ПЕРЕРИСОВАТЬ ФИГУРУ
procedure TCluster.Draw;
var
  i,n: integer;
begin
  //индекс последней фигуры в шлейфе:
```

```
  n:= _trailReadIndex;
  //нарисовать все фигуры в шлейфе:
  for i:= 1 to _trailLength do begin
    //цвет фигуры:
    bmp.Canvas.Pen.Color:= _Color[n];
    //нарисовать очередную фигуру:
    DrawCluster(_Buffer,n, _points);
    //перейти к следующей фигуре:
    n:= n+ _points;
    if n > _trailLength then n:=0;
  end;
  //новый индекс последней фигуры в шлейфе:
  _trailReadIndex:= _trailReadIndex+ _points;
  if _trailReadIndex > _trailLength then _trailReadIndex:=0;
end; //Draw
```

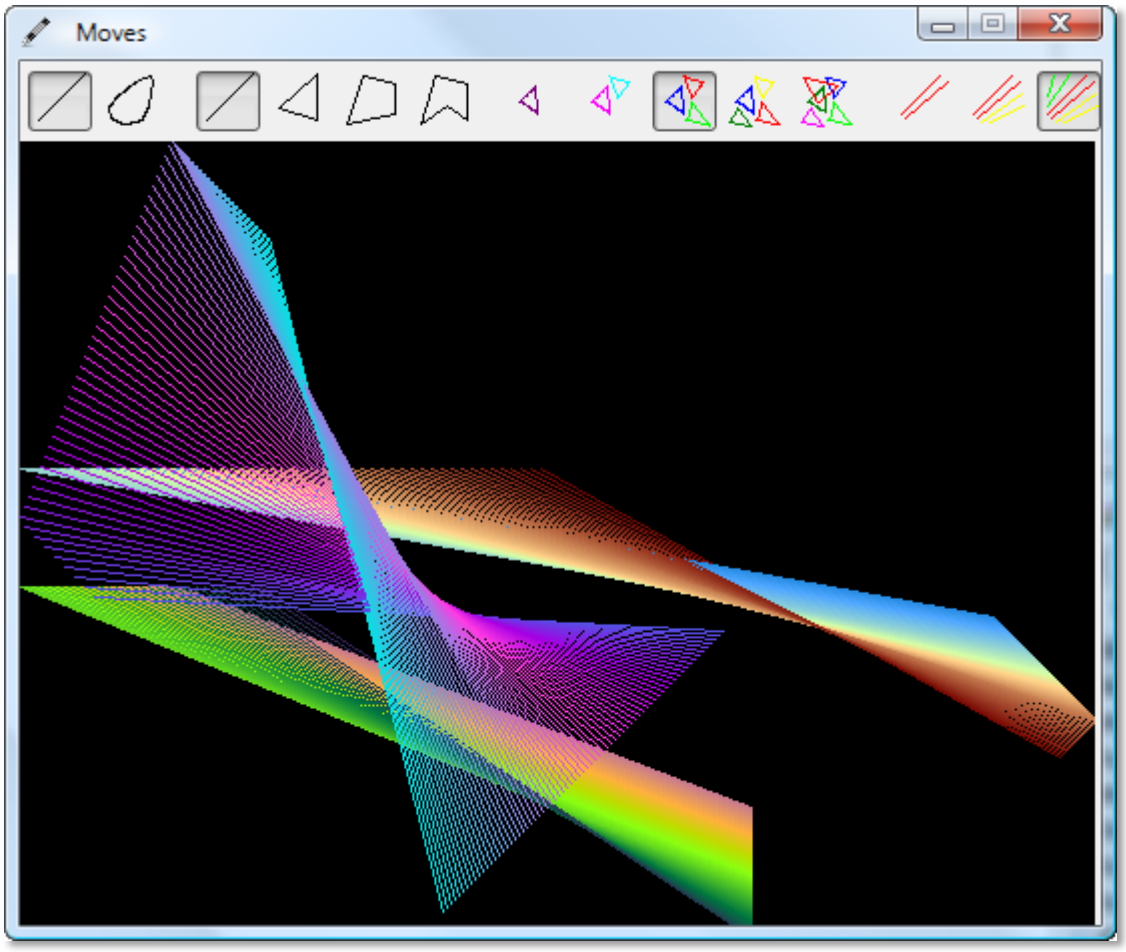


Рис. П6.31. Тоже красиво!



Исходный код программы находится в папке **Moves2**.

AVIация, или Клепаем клипы

Есть ещё одна возможность использовать анимацию в своих проектах. Среди стандартных компонентов *Delphi* на странице *Win32* вы найдёте **TAnimate** 

, который позволяет прокручивать видеоролики в формате **AVI**. Например, довольно часто при запуске программ подобным образом выводится название игры – оно появляется в сопровождении видео- и звуковых эффектов.

Создать свой клип можно в одной из множества программ для анимации: *Ulead Cool 3D, MatchWare Visual GIF Animator* и других. Но имейте в виду, что файл получится весьма приличных размеров. И хуже того: не все видеофайлы компонент **TAnimate** воспроизводит правильно. Для более экономичных форматов, таких как *GIF* и *SWF*, предназначенных для *Интернета*, в *Delphi* нет стандартных компонентов.

Занятие 7. Компот из компонентов, или «А мне всегда чего-то не хватает…»

- А у вас нет точно такого,
только с перламутровыми пуговицами?
- Нет.
- Будем искать!

Один из вопросов
Семён Семёныча Горбункова

А, правда – что делать, если возможности стандартных компонентов вас уже не удовлетворяют? – Написать свои собственные или «подправить» существующие. Эти две возможности мы *вкратце* и рассмотрим.

Откройте подменю **Component** *Главного меню* (Рис. П7.1).

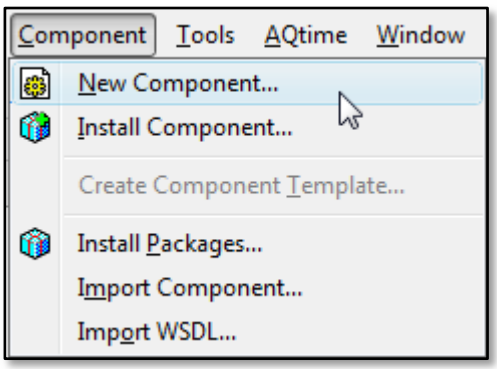


Рис. П7.1. Компоненты в меню

Оказывается, вы смело можете создавать новые компоненты – **New Component...**, а затем и устанавливать их - **Install Component...** и **Install Packages...**! Возрадовавшись, этим и займёмся.

Просто кнопка, или *Как пройти в библиотеку?*

Всё гениальное – просто.

Избитая истина

Создание новых компонентов – дело весьма хлопотное (есть даже полновесные книги об этом священнодействии), поэтому мы будем считать, что исходный код компонента у нас уже имеется. А сейчас мы займёмся установкой самой простой, но очень полезной кнопки, к написанию кода которой автор книги приложил руку и сердце. С равным успехом вы можете

воспользоваться и другими «исходниками», коих немало можно отыскать в просторах *Интернета*.

Начинаем новый проект типа **Package** (Рис. П.7.2).

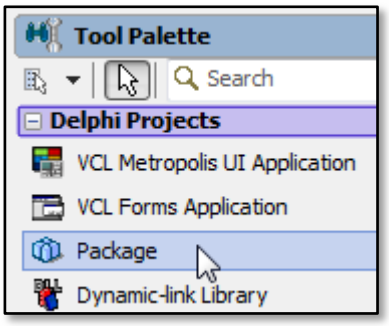


Рис. П7.2. Компонентный проект

Для этого пару раз щёлкнем по соответствующей надписи в *Палитре проектов*. Тут же в окне *Менеджера проекта* появится пустая заготовка для нашего проекта (Рис. П7.3).

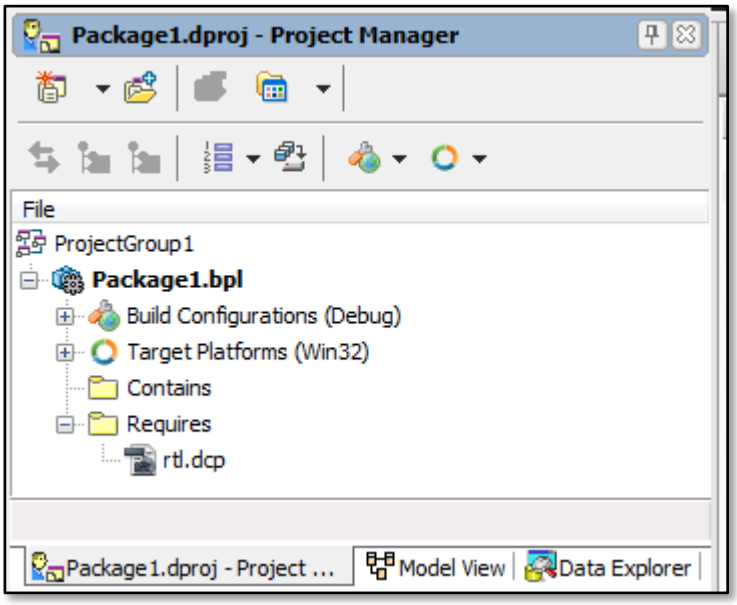


Рис. П7.3. Заготовились!

Добавим в него модуль **rvEasyButton2006.pas** и переименуем библиотеку компонентов (Рис. П7.4).

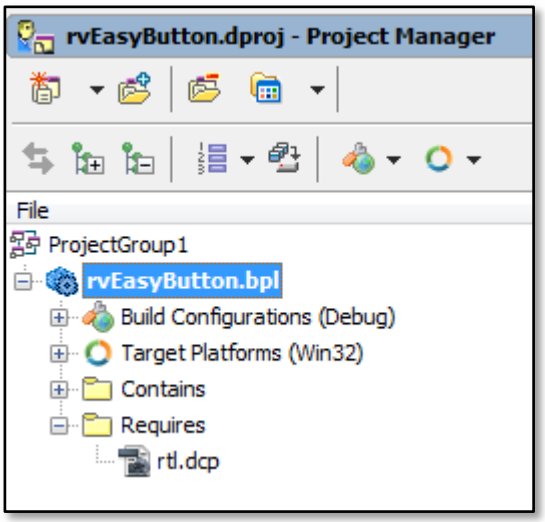


Рис. П7.4. Добавили домашнюю заготовку

Осталось нажать кнопку **Install** в контекстном меню (Рис. П7.5)

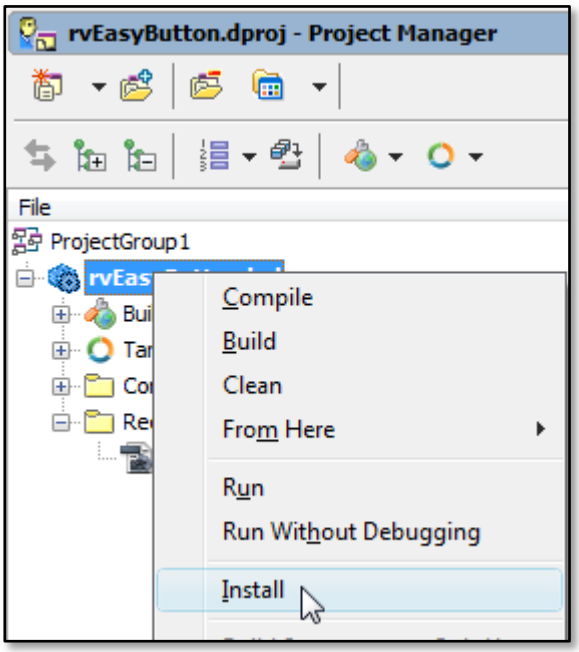


Рис. П7.5. Инсталлируем кнопку!

- и новый компонент появляется на странице **Samples** (Рис. П7.6).

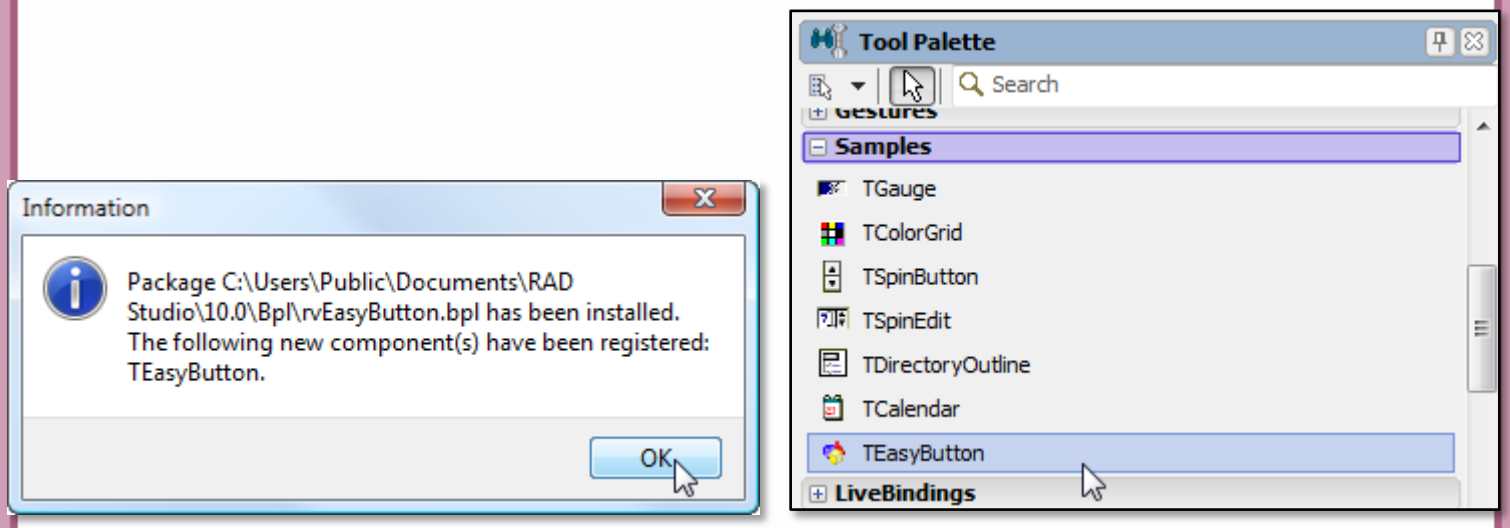


Рис. П7.6. Кнопка успешно добавлена!

Теперь мы можем пользоваться нашей кнопкой точно так же, как и другими компонентами. Проверим это утверждение!

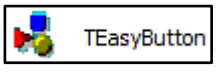


Если при запуске программы возникнут ошибки, скопируйте в папку

C:\Program Files\Embarcadero\RAD Studio\10.0\lib\win32\release

следующие файлы:

rvEasyButton2006.pas
rvEasyButton2006.dcu

Начнём новый проект. Щёлкнем дважды по значку нового компонента 

в *Палитре компонентов*, а затем загрузим две любые картинки формата *BMP* в два свойства кнопки (я, не мудрствуя лукаво, взял картинки с флагами из программы *Викторина*):

- **GlyphNormal** – картинка видна, когда курсор **не находится** на кнопке (Рис. П7.7);
- **GlyphHover** – картинка видна, когда курсор **находится** на кнопке (Рис. П7.8).

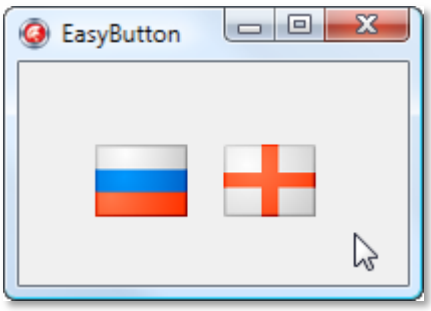


Рис. П7.7. Курсор не на кнопке

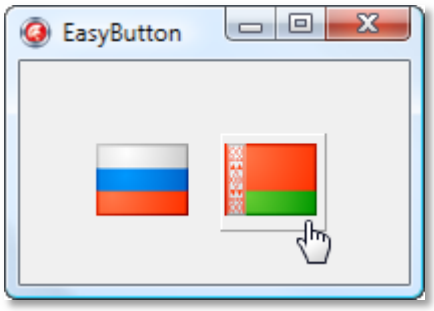


Рис. П7.8. Курсор на кнопке

Запускаем «приложение» и убеждаемся, что утверждение верно и кнопка работает прекрасно.

Вы можете использовать эту кнопку в своих проектах, а в главе, посвящённой игре *Цветные линии* вы найдёте подходящий к этому случаю пример.

В меню **Component** на вкладке **Install Packages...** находится список всех подключенных библиотек (Рис. П7.9).

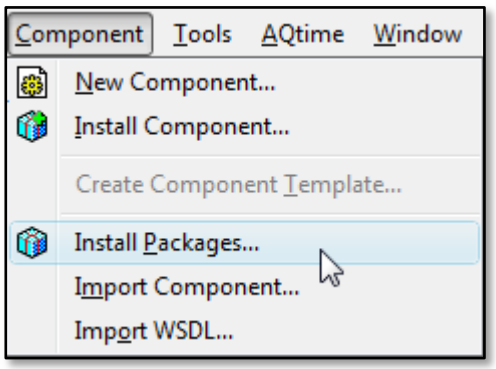


Рис. П7.9. Идём в библиотеку

В том числе и наша кнопка (Рис. П7.10).

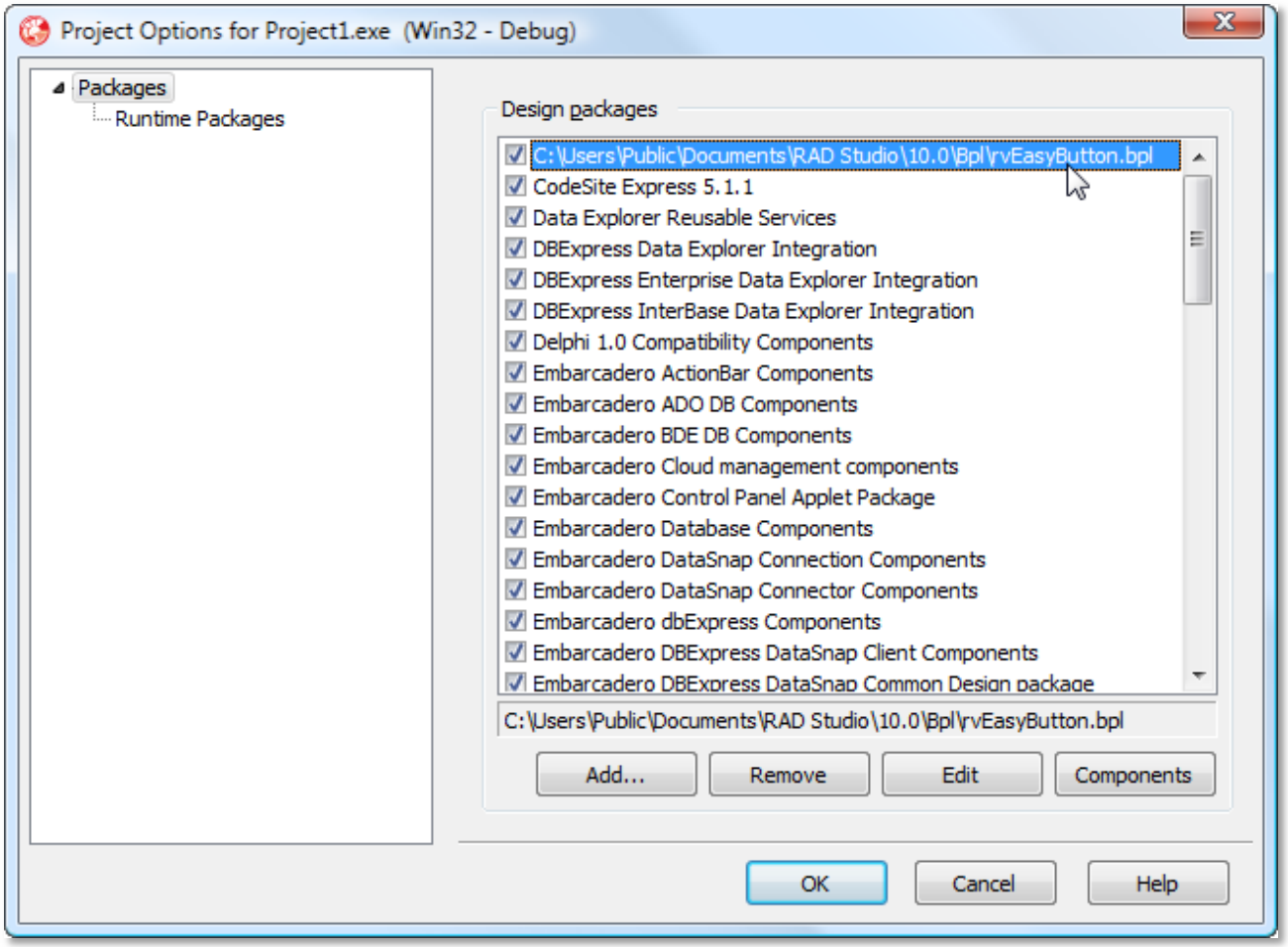


Рис. П7.10. Наша кнопка тоже тут!

Вы можете легко удалить её из списка, выделив строку с названием и нажав кнопку **Remove**. Так как при этом сама библиотека остаётся на диске, то в любое время её можно вернуть на место. Нажмите кнопку **Add** и укажите папку (так что перед тем как удалять библиотеку, запомните полный путь к ней! Или перенесите в свою папку с библиотеками), в которой хранится файл с расширением *bpl*.



Исходный код программы находится в папке **rvEasyButton**.

Копай с умом!, или Пятнадцать человек на сундук мертвеца

Не прячьте ваши денежки по банкам и углам.

Песенка из приключений Буратино

К счастью, новые компоненты приходится создавать нечасто, обычно достаточно «доработать» стандартные компоненты, придав им дополнительные полезные свойства и качества.

Посмотрим, как это делается на примере игры **Кладоискатель**. Это почти *Сапёр*, только наоборот: там игрок, обнаружив мину, взрывался вдребезги, а здесь, найдя клад, он сказочно богатеет. Это более современно и гуманно.

Однако я не просто так упомянул о *Сапёре*, дело в том, что мы воспользуемся (с любезного согласия автора книги *Занимательное программирование* Максима Мозгового) хитроумным, «сапёрным» способом создания компонентов.

Впрочем, обо всё по порядку.

Правила игры

На квадратном поле спрятано несколько кладов. **Ваша задача** – за наименьшее число попыток найти все клады. Попытка, или ход, состоит в том, что вы щёлкаете по какой-либо клетке поля, и в ней появляется число (а также и клад, если повезёт!), которое показывает, сколько всего кладов находится на 4-х прямых линиях (вертикаль, горизонталь и две диагонали – полный комплект), проходящих через эту клетку (нагляднее так: сколько кладов «видно» из данной клетки во всех направлениях; при этом клад в самой клетке, если он и есть, не виден!). Поскольку вашей жизни и даже умственному здоровью ничего не угрожает, то клады рано или поздно вы всё равно найдёте. Но это скучно, поэтому, пользуясь полученной в результате хода информацией, старайтесь «вычислить», где же «собака зарыта». Вот это уже интересно!

Программирование начнём с конца: вот так выглядит готовая программа в действии (Рис. П7.11).

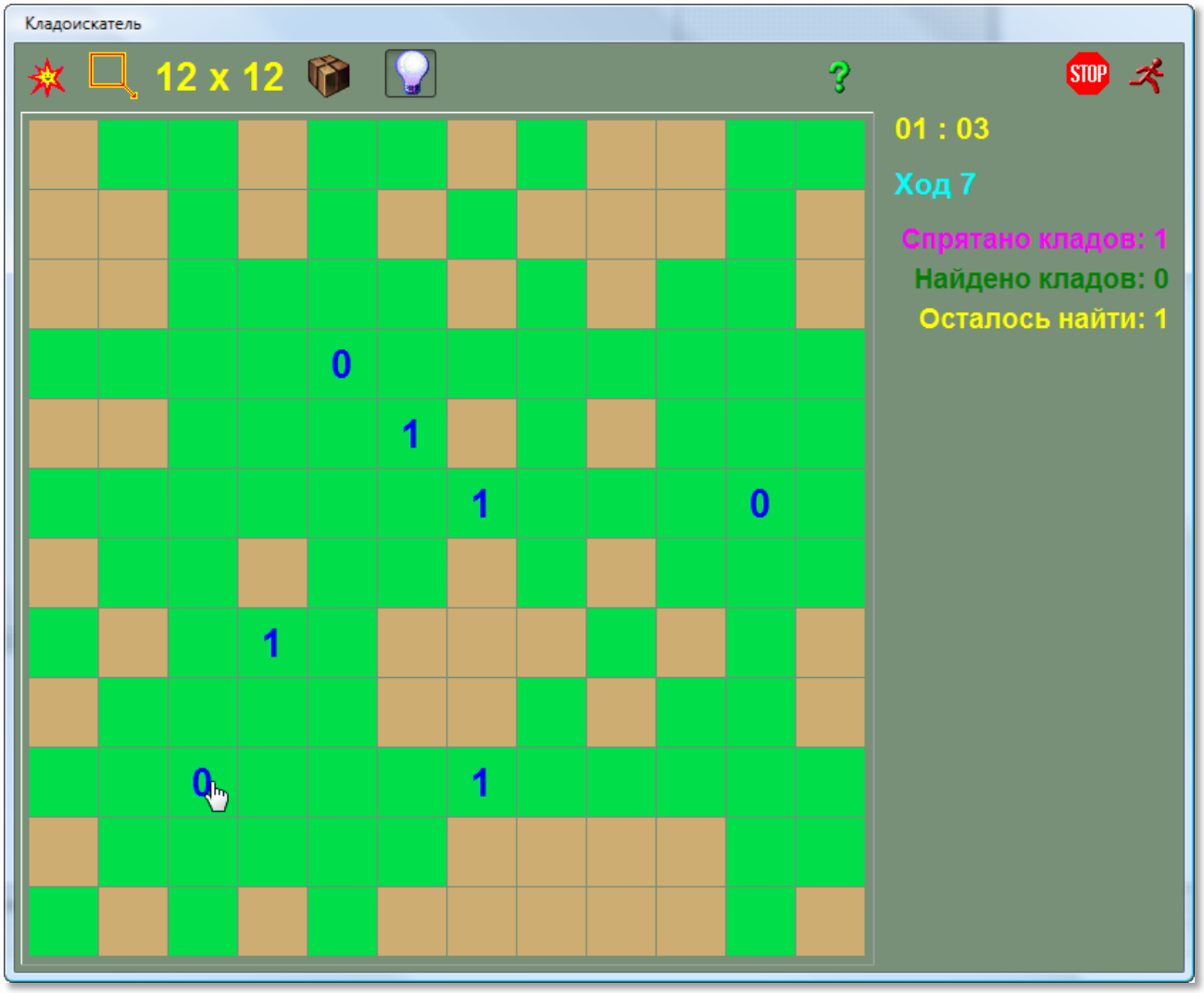


Рис. П7.11. Ищем клады!

Только что запущенная программа отличается тем, что размеры поля - 4 х 4, все клетки – коричневые и информация о ходе игры – по умолчанию.

Что же происходит при запуске программы? – Всего-навсего подготовка к игре:

```
//-------------------- ПОДГОТОВКА К ИГРЕ -------------------------

//СОЗДАТЬ ФОРМУ - ПОДГОТОВИТЬСЯ К ИГРЕ
procedure TForm1.FormCreate(Sender: TObject);
begin
  //установить цвета компонентов:
  Color:= rgb(120,144,120);
  Panel1.Color:= rgb(34,94,54);
  Panel3.Color:= Color;
  Randomize;
  //подготовить поле для игры:
  PrepareField;
```

```
end;

//ПОДГОТОВИТЬ ПОЛЕ К НОВОЙ ИГРЕ
procedure PrepareField;
var i, j : Integer;
begin
  //показать размеры поля:
  form1.lblWH.Caption:= inttostr(FieldWidth)+ ' x '+ int-
tostr(FieldHeight);
  //создать поле:
  MakeField;
  //вывести картинки в клетки:
  for i := 0 to FieldWidth - 1 do
    for j := 0 to FieldHeight - 1 do
      begin
        //картинок нет:
        Field[i, j].Picture := nil;
        //пока содержимое всех клеток нам не известно -
        //поместим в них коричневый квадрат:
        Field[i, j].Picture:= form1.imgNoKnow.picture;
        //клетка не нажата:
        Field[i, j].clicked:= false;
        //и клада в ней пока нет:
        Field[i, j].is_treasure:= false;
      end;
end;
```

Самое интересное для нас, для чего мы, собственно, и затеяли эту игру, происходит в процедуре **MakeField**, где мы динамически создаём поле из компонентов **TMyImage**. Не ищите их в *Палитре компонентов* – их там нет. В принципе, можно было бы удовольствоваться и обычными компонентами **TImage**, но тогда пришлось бы добавить ещё несколько массивов, что не очень удобно.

Поэтому мы «изготовим» новый компонент на основе компонента-картинки **TImage**, дополнив его новыми свойствами и переписав обработчик события **OnClick**, то есть щелчка на компоненте:

```
//Клетка поля на основе компонента TImage
type TMyImage = class(TImage)
  public
    x, y: Integer;              // - координаты клетки на игровом
поле
    clicked : Boolean;          // = TRUE, если клетка уже "кликну-
та"
    is_treasure: Boolean;       // = TRUE, если в этой клетке нахо-
дится клад
    nSosed: Integer;            // - число кладов на прямых, прохо-
дящих через клетку
    procedure Click; override; //- обработчик события OnClick
```

```
  end;
```

Так как наш компонент существует «только на бумаге», то его нужно реально создать при формировании поля. Игровое поле состоит из отдельных квадратиков размером *CELL_WIDTH x CELL_HEIGHT:*

```
const
  NAME_PROG='КЛАДОИСКАТЕЛЬ 2007';
  //Максимальные размеры поля:
  MAX_FIELDWIDTH = 12;
  MAX_FIELDHEIGHT = 12;
  //Размеры клеток поля:
  CELL_WIDTH = 48;
  CELL_HEIGHT = 48;
```

Каждый квадратик – это отдельный экземпляр компонента **TMyImage**, расположенный на панели **Panel1**. Можно располагать клетки поля и непосредственно на форме, но решение с панелью мне кажется более привлекательным с эстетической точки зрения. Все клетки поля объединены в массив **Field**:

```
var
  Form1: TForm1;
  //Игровое поле - массив клеток:
  Field : array[0..MAX_FIELDWIDTH - 1, 0..MAX_FIELDHEIGHT - 1] of TMyImage;
```

В процедуре **MakeField** мы создаём нужное число экземпляров компонента **TMyImage**, задаём им необходимые свойства – и поле готово:

```
//СОЗДАТЬ ИГРОВОЕ КНОПОК ИЗ КОМПОНЕНТОВ TMyImage
procedure MakeField;
var i, j: Integer;
begin
  //отцентрировать клетки относительно панели
  //в зависимости от размеров поля:
  offsetX:= (Form1.Panel1.Width - CELL_WIDTH*FieldWidth-
FieldWidth+1) div 2;
  offsetY:= (Form1.Panel1.Height - CELL_HEIGHT*FieldHeight-
FieldHeight+1) div 2;
  //создать новые компоненты TMyImage по размерам поля:
  for i := 0 to FieldWidth - 1 do
    for j := 0 to FieldHeight - 1 do
      begin
        Field[i, j] := TMyImage.Create(nil);
          //задать свойства новому компоненту:
          with Field[i, j] do
            begin
              //расположение на панели:
              Left := i * (CELL_WIDTH+1) + offsetX;
              Top := j * (CELL_HEIGHT+1) + offsetY;
              //размеры:
```

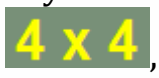
```
                Width := CELL_WIDTH;
                Height := CELL_HEIGHT;
                Cursor:= crHandPoint;
                Transparent:= FALSE;
                //родителя:
                Parent := Form1.Panel1;
                //индексы в массиве поля:
                x:= i;
                y:= j;
                //"соседние" клетки ещё не подсчитаны:
                nSosed:= -1;
              end;
          end;
end;
```

При первой загрузке поле имеет минимальные размеры 4 x 4 клетки, что определяется значениями переменных **FieldWidth** и **FieldHeight** по умолчанию:

```
  //ширина поля:
  FieldWidth: Integer = 4;
  //высота поля:
  FieldHeight: Integer = 4;
```

Конечно, это поле слишком мало для настоящих игроков, поэтому мы позаботимся о его солидном увеличении - до 12 x 12 клеток:

```
const
  NAME_PROG='КЛАДОИСКАТЕЛЬ 2007';
  //Максимальные размеры поля:
  MAX_FIELDWIDTH = 12;
  MAX_FIELDHEIGHT = 12;
```

Для этого мы установим кнопку **sbtCreateNewSizeGame** , а рядом метку **lblWH** 

, в которой и будем показывать актуальные размеры поля. Их удобнее выбирать в меню, раскрывающемся при нажатии на кнопку (Рис. П7.12).

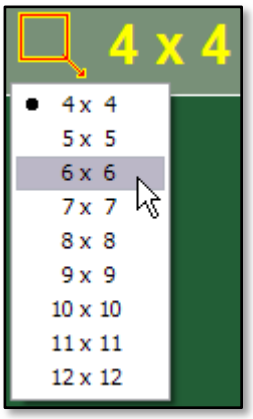


Рис. П7.12. Размерное меню

Искать **один** клад – тоже невесело, поэтому подумаем о том, чтобы на поле можно было закопать *разумное* число кладов. За это будет отвечать кнопка **sbtNumTreasures** . Здесь под «разумными» мы будем понимать такие значения:

```
//ЧТОБЫ ЗАДАТЬ ЧИСЛО КЛАДОВ НА ПОЛЕ,
//НУЖНО НАЖАТЬ КНОПКУ sbtNumTreasures
//И УСТАНОВИТЬ ДВИЖОК НА НУЖНОЕ ЗНАЧЕНИЕ
procedure TForm1.sbtNumTreasuresMouseDown(Sender: TObject; Button:
TMouseButton;
  Shift: TShiftState; X, Y: Integer);
var i: integer;
begin
  if GameStatus=PLAY then exit;//- игра уже идёт!

  //задать максимальное число кладов в зависимости от размера сет-
ки:
  i:= 4;
  case FieldWidth of
    4: i:= 4;
    5: i:= 6;
    6: i:= 9;
    7: i:=12;
    8: i:=16;
    9: i:=20;
   10: i:=25;
   11: i:=30;
   12: i:=36;
  end;

  //настроить движки на форме frmNumTreasures
  //в соответствии с числом кладов:
  frmNumTreasures.tbNumTreasures.Max:= i;
  frmNumTreasures.tbNumTreasures.Position:= 1;
  //расположить форму frmNumTreasures
  //под кнопкой:
```

```
  frmNumTreasures.Left:= form1.left + sbtNumTreasures.Left;
  frmNumTreasures.Top:=  form1.top + sbtNumTrea-
sures.Top+sbtNumTreasures.Height+28;
  //frmNumTreasures.ShowModal;
  //показать форму frmNumTreasures
  //(счезает с экрана после щелчка на ней):
  frmNumTreasures.Show;
end;
```

Дабы не загромождать форму чуждыми ей элементами управления, движок вынесем на отдельную форму – **frmNumTreasures**. Так она выглядит в окне *Конструктора формы* и в запущенном приложении (Рис. П7.13).

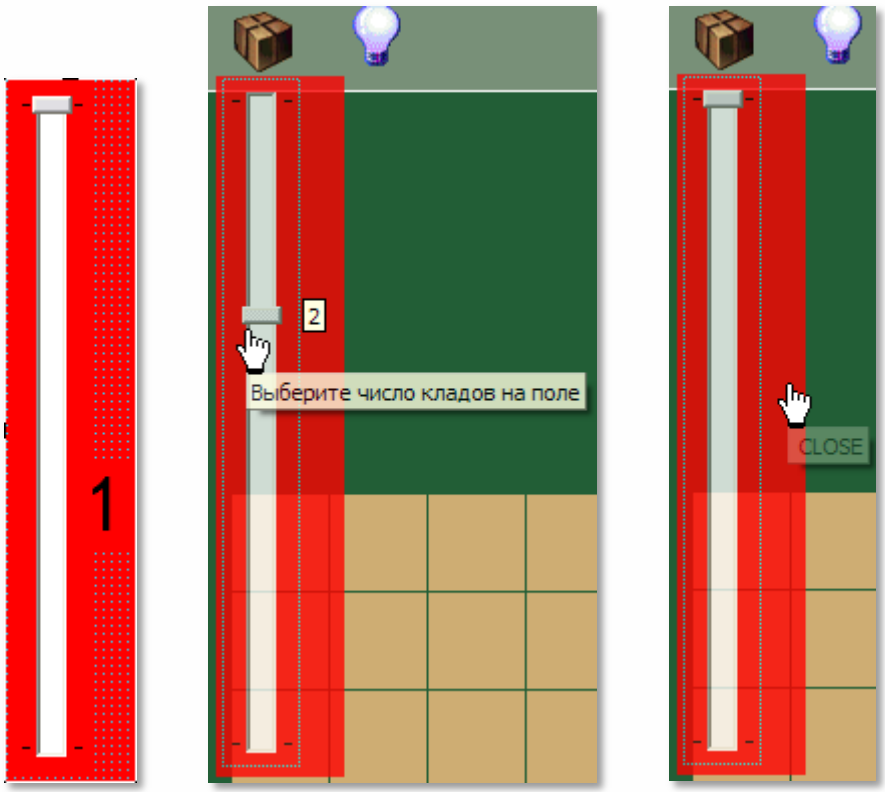


Рис. П7.13. Кладовый движок

Я окрасил её в ярко-красный цвет и разместил сразу под кнопкой, чтобы она не затерялась на форме.

```
unit NumTreasuresUnit;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Con-
trols, Forms,
  Dialogs, ComCtrls, StdCtrls, Buttons;
```

```
type
  TfrmNumTreasures = class(TForm)
    tbNumTreasures: TTrackBar;
    lblNumTreasures: TLabel;
    procedure tbNumTreasuresChange(Sender: TObject);
    procedure FormClick(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  frmNumTreasures: TfrmNumTreasures;

implementation
uses TreasureUnit;
{$R *.dfm}

//ЗАДАТЬ ЧИСЛО КЛАДОВ НА ПОЛЕ
procedure TfrmNumTreasures.FormClick(Sender: TObject);
begin
  nTreasures:= frmNumTreasures.tbNumTreasures.Position;
  //ни один клад не найден:
  nMyTreasures:=0;
  PrintTreasures([tpmTreasure]+[tpmMyTreasure]+[tpmRest]);
  close;
end;

end.
```

При перемещении ползунка изменяется число кладов на поле. Установив нужное значение, щёлкните на форме **frmNumTreasures** – она и закроется. Теперь при старте **новой** игры на поле будет зарыто достаточное число кладов!

Пора начинать игру (сама она не начнётся!). Для этого достаточно нажать кнопку :

```
//-------------------- НАЧАЛО ИГРЫ --------------------------


//ИГРА НАЧИНАЕТСЯ ПОСЛЕ НАЖАТИЯ НА КНОПКУ СТАРТ!
procedure TForm1.sbtStartNewGameClick(Sender: TObject);
begin
  //уничтожить предыдущее поле, так как новое
  //может иметь другие размеры, и компоненты
  //придётся создавать вновь:
  DestroyField;
```

```
  //запустить новую игру:
  StartGame;
end;

//НАЧИНАЕМ ИГРУ!!!
procedure StartGame;
var i,j,n : Integer;
begin
  //подготовить поле:
  PrepareField;
  //обнулить ходы:
  nMove:= 0;
  Form1.lblHod.Caption:= 'Ход '+inttostr(nMove);
  //обнулить счётчик найденных кладов:
  nMyTreasures:=0;
  PrintTreasures([tpmTreasure]+[tpmMyTreasure]+[tpmRest]);

   //спрятать клады:
   n:=0;
   //всего прячем nTreasures кладов
   //(по умолчанию = 1, изменяется кнопкой sbtNumTreasures)
   while n< nTreasures do
   begin
     //выбираем случайную клетку поля:
     i:= Random(FieldWidth);
     j:= Random(FieldHeight);
     //если там клада нет
     //(в клетке не может быть больше одного клада!),
     //то прячем его в этой клетке:
     if not Field[i, j].is_treasure then
     begin
       Field[i, j].is_treasure:= TRUE;
       inc(n);
     end;
   end;

   //все клады спрятаны!!!
   sound('newgame');
   //включить таймер
   Form1.Timer1.Enabled:= TRUE;
   //играем!!!
   GameStatus:= PLAY;
   //засечь время начала тестирования:
   time0:=GetTickCount();
end;
```

Время пошло:

```
//ВЫВОДИМ ВРЕМЯ ИГРЫ И ПРОВЕРЯЕМ, НЕ ЗАКОНЧЕНА ЛИ ОНА
procedure TForm1.Timer1Timer(Sender: TObject);
var
  s: string;
```

```
  min, sec: integer;
begin
  //остановить игру
  if GameStatus= STOP then begin
    Timer1.Enabled:= False;
    exit
  end;

  //время игры:
  time:=(GetTickCount()-time0)/1000;
  min:= Round(time) div 60;
  if min> 59 then begin //- время закончилось!
    Timer1.Enabled:= False;
    sound('gameover');
    Application.MessageBox(pChar(s), NAME_PROG, IDOK);
    GameStatus:= STOP;
    exit
  end;

  //вывести время игры:
  sec:= Round(time) mod 60;
  s:= inttostr(sec);
  if sec< 10 then s:= '0' + s;
  if min< 10 then s:= '0' + inttostr(min)+ ' : ' + s
  else s:= inttostr(min)+ ' : ' + s;
  lblTime.Caption:= s;
end; //Timer1Timer
```

Не пугайтесь: время игры не учитывается, хотя и разумно ограничено 1 часом.

Кнопка помощи  должна показывать файл справки *Help.htm*, написание которого целиком лежит на вашей совести!

```
//ПОКАЗАТЬ ПРАВИЛА ИГРЫ
procedure TForm1.sbtHelpClick(Sender: TObject);
var s: string;
begin
  s:= extractfilepath(application.exename)+'Help.htm';
  //ShellExecute(handle, 'open','iexplore',pChar(s),'',
SW_SHOWMAXIMIZED);
end;  // sbtHelpClick
```

В назначении кнопки 

ошибиться трудно – она останавливает начатую игру, если она вас утомила. После этого вы можете изменить параметры игры и начать новую партию:

```
//ОСТАНОВИТЬ ИГРУ
procedure TForm1.sbtStopClick(Sender: TObject);
var s: string;
```

```
begin
    GameStatus:= STOP;
    sound('end');
    s:='Игра остановлена!';
    Application.MessageBox(pChar(s), NAME_PROG, MB_OK);
end;
```

Последняя кнопка -  - просто закрывает программу:

```
//ЗАКРЫТЬ ПРОГРАММУ
procedure TForm1.sbtExitClick(Sender: TObject);
begin
  close;
end;
```

Осталось самое важное – игра как таковая. Как вы помните, сначала поле (или карта) совершенно пустое, то есть лишено какой бы то ни было информации. Она появляется и обновляется после каждого клика на клетке поля (правда, **повторно** клетку кликнуть можно, но программа проигнорирует **неразумные** действия):

```
//-------------------- ИГРА --------------------------

//НАЖАТЬ КНОПКУ МЫШИ НА КЛЕТКЕ ПОЛЯ - СДЕЛАТЬ ХОД
procedure TMyImage.Click;
var i, j,k,l    : Integer;
    s: string;

  //вычислить число "соседей" у клетки (x,y)
  //(то есть число кладов, находящихся
  //на прямых, проходящих через клетку (x,y))
  function CalcTreasure(x,y: integer): integer;
  var i,j: integer;
  begin
    Result:= 0;
    //ищем по всему полю:
    for j:= 0 to FieldHeight-1 do
      for i:= 0 to FieldWidth-1 do
        //нашли клетку с кладом:
        if Field[i, j].is_treasure then begin
          //если она находится на одной линии
          //с клеткой (x,y), то эта клетка "соседняя":
          if (x <> i) and (y=j) then inc(Result);
          if (y <> j) and (x=i) then inc(Result);
          if (abs(x-i) = abs(y-j)) and (x<>i)then inc(Result);
        end;
   end;

  //вычислить число соседей у нажатой клетки (x,y)
  function CalcTreasureClicked(x,y: integer): integer;
```

```
  var i,j: integer;
  begin
    Result:= 0;
    //ищем по всему полю:
    for j:= 0 to FieldHeight-1 do
      for i:= 0 to FieldWidth-1 do
        //нашли нажатую клетку с кладом:
        if (Field[i, j].is_treasure) and (Field[i, j].clicked)
then begin
          if (x <> i) and (y=j) then inc(Result);
          if (y <> j) and (x=i) then inc(Result);
          if (abs(x-i) = abs(y-j)) and (x<>i)then inc(Result);
        end;
   end;


begin //TMyImage.Click

  //если игра закончена или клетка уже была нажата,
  //то ничего не делать:
  if (GameStatus=STOP) or (clicked) then Exit;

  //если нажата клавиша CTRL,
  //то обозначить клетку как пустую - зелёного цвета:
  if (GetKeyState(VK_CONTROL) and $100 <> 0) then begin
    Picture:= form1.imgEmpty.picture;
    exit;
  end;

  //клетка нажата:
  clicked := true;
  sound('click2');
  //сделан ещё 1 ход:
  inc(nMove);
  Form1.lblHod.Caption:= 'Ход '+inttostr(nMove);

  //в этой клетке находится клад:
  if is_treasure then
  begin
    sound('buljk');
    //нашли ещё 1 клад:
    inc(nMyTreasures);
    PrintTreasures([tpmMyTreasure]+[tpmRest]);
    //показать в клетке картинку с кладом:
    Picture:= form1.imgTreasure.picture;
    //цифры жёлтого цвета:
    Canvas.Font.Color:= clYellow;
  end
  else begin  //пустая клетка:
    Picture:= form1.imgEmpty.picture;
    //цифры синего цвета:
    Canvas.Font.Color:= clBlue;
  end;
```

```
  //подсчитать, сколько кладов ("соседей") находится
  //на линиях, проходящих через клетку(x,y):
  nSosed:=CalcTreasure(x,y);
  //напечатать в клетке заданным цветом
  //число кладов:
  Canvas.Font.Name:= 'Arial';
  Canvas.Font.Height:= 32;
  Canvas.Font.Style:= Font.Style + [fsBold];
  Canvas.Brush.Style:= bsClear;
  if nSosed < 10 then Canvas.TextOut(16,9,inttostr(nSosed))
  else Canvas.TextOut(8,9,inttostr(nSosed));

  //если найдены все клады, то игра закончена:
  if nTreasures=nMyTreasures then begin
    GameStatus:= STOP;
    sound('Quest');
    s:='Вы нашли все клады! ('+ inttostr(nTreasures)+')';
    Application.MessageBox(pChar(s), NAME_PROG, MB_OK);
    exit;
  end;

  //найдены не все клады - игра продолжается

  //если у кликнутой клетки нет соседей
  //и нажата кнопка sbtHint, то отметить все пустые клетки:
  if (nSosed=0) and (Form1.sbtHint.Down) then
    for j:= 0 to FieldHeight-1 do
      for i:= 0 to FieldWidth-1 do
        if (i=x) or (j=y) or (i+j=x+y) or (i-j=x-y) then begin
          if not Field[i, j].is_treasure then Field[i,
j].Picture:= form1.imgEmpty.picture;
            if (Field[i, j].nSosed>-1)  then
              with Field[i, j] do begin
                Canvas.Font.Name:= 'Arial';
                Canvas.Font.Height:= 32;
                Canvas.Font.Style:= Font.Style + [fsBold];
                Canvas.Brush.Style:= bsClear;
                if is_treasure then Canvas.Font.Color:= clYellow
                else Canvas.Font.Color:= clBlue;
                if nSosed < 10 then Can-
vas.TextOut(16,9,inttostr(nSosed))
                else Canvas.TextOut(8,9,inttostr(nSosed));
              end;
          end;

  //если для какой-либо клетки уже найдены все клады
  //и нажата кнопка sbtHint,
  //то остальные соседние клетки можно отметить как пустые:
  if (Form1.sbtHint.Down) then
    //для каждой клетки поля:
    for j:= 0 to FieldHeight-1 do
      for i:= 0 to FieldWidth-1 do
```

```
        //если для неё определено число соседей
        if Field[i, j].nSosed > 0 then
          if CalcTreasureClicked(i,j)= Field[i, j].nSosed then
          //по всем кладам:
          for k:= 0 to FieldHeight-1 do
            for l:= 0 to FieldWidth-1 do
              if (i=l) or (j=k) or (i+j=l+k) or (i-j=l-k) then
begin
                if not Field[l, k].is_treasure then Field[l,
k].Picture:= form1.imgEmpty.picture;
                if (Field[l, k].nSosed>-1)  then
                with Field[l, k] do begin
                  Canvas.Font.Name:= 'Arial';
                  Canvas.Font.Height:= 32;
                  Canvas.Font.Style:= Font.Style + [fsBold];
                  Canvas.Brush.Style:= bsClear;
                  if is_treasure then Canvas.Font.Color:= clYellow
                  else Canvas.Font.Color:= clBlue;
                  if nSosed < 10 then Can-
vas.TextOut(16,9,inttostr(nSosed))
                  else Canvas.TextOut(8,9,inttostr(nSosed));
                end;
              end;

end;
```

Для пущего комфорта добавим такую приятную возможность, как закраска заведомо пустых клеток зелёным цветом. В этом случае нужно удерживать клавишу *CTRL* при щелчке на клетке:

```
  //если нажата клавиша CTRL,
  //то обозначить клетку как пустую - зелёного цвета:
  if (GetKeyState(VK_CONTROL) and $100 <> 0) then begin
    Picture:= form1.imgEmpty.picture;
    exit;
  end;
```

Если в кликнутой клетке находится клад, то выводим в неё картинку с кладом :

```
  //в этой клетке находится клад:
  if is_treasure then
  begin
    sound('buljk');
    //нашли ещё 1 клад:
    inc(nMyTreasures);
    PrintTreasures([tpmMyTreasure]+[tpmRest]);
    //показать в клетке картинку с кладом:
    Picture:= form1.imgTreasure.picture;
```

```
    //цифры жёлтого цвета:
    Canvas.Font.Color:= clYellow;
  end
  else begin  //пустая клетка:
    Picture:= form1.imgEmpty.picture;
    //цифры синего цвета:
    Canvas.Font.Color:= clBlue;
  end;
```

Если же в клетке ничего нет, то «принудительно» окрашиваем её в зелёный цвет 

. Кроме того, в клетке показываем число кладов, которые «видны» из неё:

```
  //подсчитать, сколько кладов ("соседей") находится
  //на линиях, проходящих через клетку(x,y):
  nSosed:=CalcTreasure(x,y);
  //напечатать в клетке заданным цветом
  //число кладов:
  Canvas.Font.Name:= 'Arial';
  Canvas.Font.Height:= 32;
  Canvas.Font.Style:= Font.Style + [fsBold];
  Canvas.Brush.Style:= bsClear;
  if nSosed < 10 then Canvas.TextOut(16,9,inttostr(nSosed))
  else Canvas.TextOut(8,9,inttostr(nSosed));
```

Вычислять количество «соседей» у заданной клетки (я не придумал более вразумительного термина для кладов, находящихся на 8-ми прямых, проходящих через клетку) по-своему интересно:

```
  //вычислить число "соседей" у клетки (x,y)
  //(то есть число кладов, находящихся
  //на прямых, проходящих через клетку (x,y))
  function CalcTreasure(x,y: integer): integer;
  var i,j: integer;
  begin
    Result:= 0;
    //ищем по всему полю:
    for j:= 0 to FieldHeight-1 do
      for i:= 0 to FieldWidth-1 do
        //нашли клетку с кладом:
        if Field[i, j].is_treasure then begin
          //если она находится на одной линии
          //с клеткой (x,y), то эта клетка "соседняя":
          if (x <> i) and (y=j) then inc(Result);
          if (y <> j) and (x=i) then inc(Result);
          if (abs(x-i) = abs(y-j)) and (x<>i)then inc(Result);
        end;
   end;
```

Мы по очереди отыскиваем клетки с кладом (можно и просто запомнить их координаты в списке) и проверяем, не находятся ли они на одной линии с кликнутой клеткой. Если это так, то добавляем 1 к числу «соседей». Если мы нашли все клады, то игра закончена:

```
  //если найдены все клады, то игра закончена:
  if nTreasures=nMyTreasures then begin
    GameStatus:= STOP;
    sound('Quest');
    s:='Вы нашли все клады! ('+ inttostr(nTreasures)+')';
    Application.MessageBox(pChar(s), NAME_PROG, MB_OK);
    exit;
  end;
```

Ноль в клетке ясно говорит о том, что у неё «соседей» нет, то есть все клетки во всех направлениях «пустые», без кладов, а их мы договорились окрашивать в зелёный цвет. Вы это можете сделать самостоятельно (удерживая клавишу *CTRL*, как это было описано выше) либо предоставить эту рутину программе. О ваших намерениях говорит кнопка **sbtHint** . Если она нажата, подсказки выводятся, в противном случае – нет:

```
  //если у кликнутой клетки нет соседей
  //и нажата кнопка sbtHint, то отметить все пустые клетки:
  if (nSosed=0) and (Form1.sbtHint.Down) then
    for j:= 0 to FieldHeight-1 do
      for i:= 0 to FieldWidth-1 do
        if (i=x) or (j=y) or (i+j=x+y) or (i-j=x-y) then begin
          if not Field[i, j].is_treasure then Field[i,
j].Picture:= form1.imgEmpty.picture;
            if (Field[i, j].nSosed>-1)  then
              with Field[i, j] do begin
                Canvas.Font.Name:= 'Arial';
                Canvas.Font.Height:= 32;
                Canvas.Font.Style:= Font.Style + [fsBold];
                Canvas.Brush.Style:= bsClear;
                if is_treasure then Canvas.Font.Color:= clYellow
                else Canvas.Font.Color:= clBlue;
                if nSosed < 10 then Can-
vas.TextOut(16,9,inttostr(nSosed))
                else Canvas.TextOut(8,9,inttostr(nSosed));
              end;
          end;
```

Другая подсказка похитрее. Если, например, в клетке стоит **1** и этот клад уже найден, то очевидно, что все остальные клетки по 8-ми направлениям должны быть зелёного цвета (кладов в них точно нет):

```
  //если для какой-либо клетки уже найдены все клады
```

```
  //и нажата кнопка sbtHint,
  //то остальные соседние клетки можно отметить как пустые:
  if (Form1.sbtHint.Down) then
    //для каждой клетки поля:
    for j:= 0 to FieldHeight-1 do
      for i:= 0 to FieldWidth-1 do
        //если для неё определено число соседей
        if Field[i, j].nSosed > 0 then
          if CalcTreasureClicked(i,j)= Field[i, j].nSosed then
          //по всем кладам:
          for k:= 0 to FieldHeight-1 do
            for l:= 0 to FieldWidth-1 do
              if (i=l) or (j=k) or (i+j=l+k) or (i-j=l-k) then
begin
                if not Field[l, k].is_treasure then Field[l,
k].Picture:= form1.imgEmpty.picture;
                if (Field[l, k].nSosed>-1)  then
                with Field[l, k] do begin
                  Canvas.Font.Name:= 'Arial';
                  Canvas.Font.Height:= 32;
                  Canvas.Font.Style:= Font.Style + [fsBold];
                  Canvas.Brush.Style:= bsClear;
                  if is_treasure then Canvas.Font.Color:= clYellow
                  else Canvas.Font.Color:= clBlue;
                  if nSosed < 10 then Can-
vas.TextOut(16,9,inttostr(nSosed))
                  else Canvas.TextOut(8,9,inttostr(nSosed));
                end;
              end;
```

Число кладов подсчитываем, как и раньше:

```
  //вычислить число соседей у нажатой клетки (x,y)
  function CalcTreasureClicked(x,y: integer): integer;
  var i,j: integer;
  begin
    Result:= 0;
    //ищем по всему полю:
    for j:= 0 to FieldHeight-1 do
      for i:= 0 to FieldWidth-1 do
        //нашли нажатую клетку с кладом:
        if (Field[i, j].is_treasure) and (Field[i, j].clicked)
then begin
          if (x <> i) and (y=j) then inc(Result);
          if (y <> j) and (x=i) then inc(Result);
          if (abs(x-i) = abs(y-j)) and (x<>i)then inc(Result);
        end;
   end;
```

Этим, естественно, возможности подсказывания не исчерпываются, и вы

можете добавить и свои находки по части решения задач (смотрите также дальше). А тут, как говорится, и сказочке конец!



Исходный код программы находится в папке **rvTreasure**.

Однако, вернёмся к теме этой главы и создадим из «виртуального» компонента **TMyImage** «реальный», который будет находиться в *Палитре компонентов* среди себе подобных. Это полезно не только для пустого экспериментирования, но и для практических нужд – подобным образом вы можете создать сколько угодно своих компонентов, которые не нужно создавать динамически и потому удобно расставлять на форме в режиме проектирования.

Начинаем создавать новый компонент (Рис. П7.14).

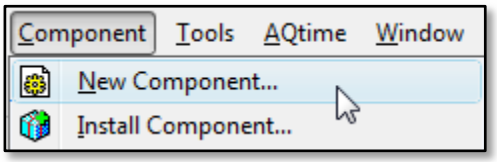


Рис. П7.14. Начинаем!

Так как наша модернизированная картинка наследует **TImage**, то именно этот компонент и следует выбрать в списке (Рис. П7.15).

Нажимаем кнопку **Next** и выбираем имя для нового класса и страницу в *Палитре компонентов*, куда будет помещён наш компонент (Рис. П7.16).

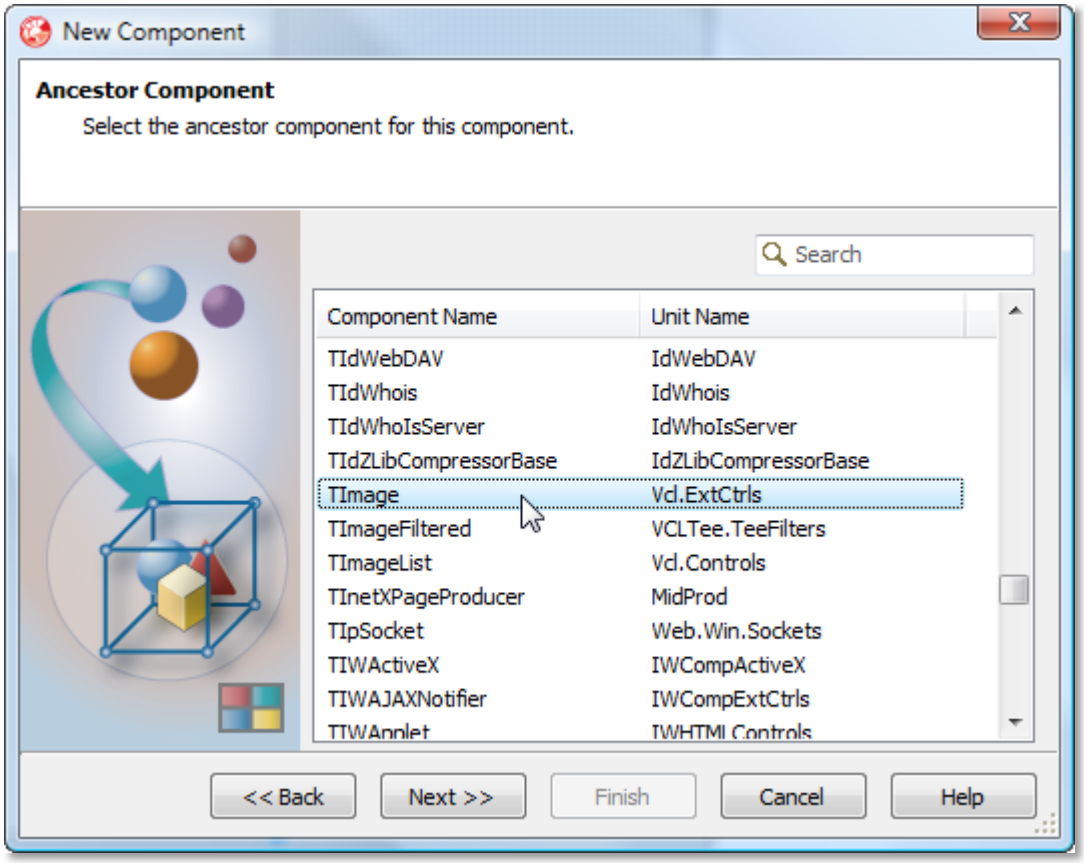


Рис. П7.15. Выбираем предка

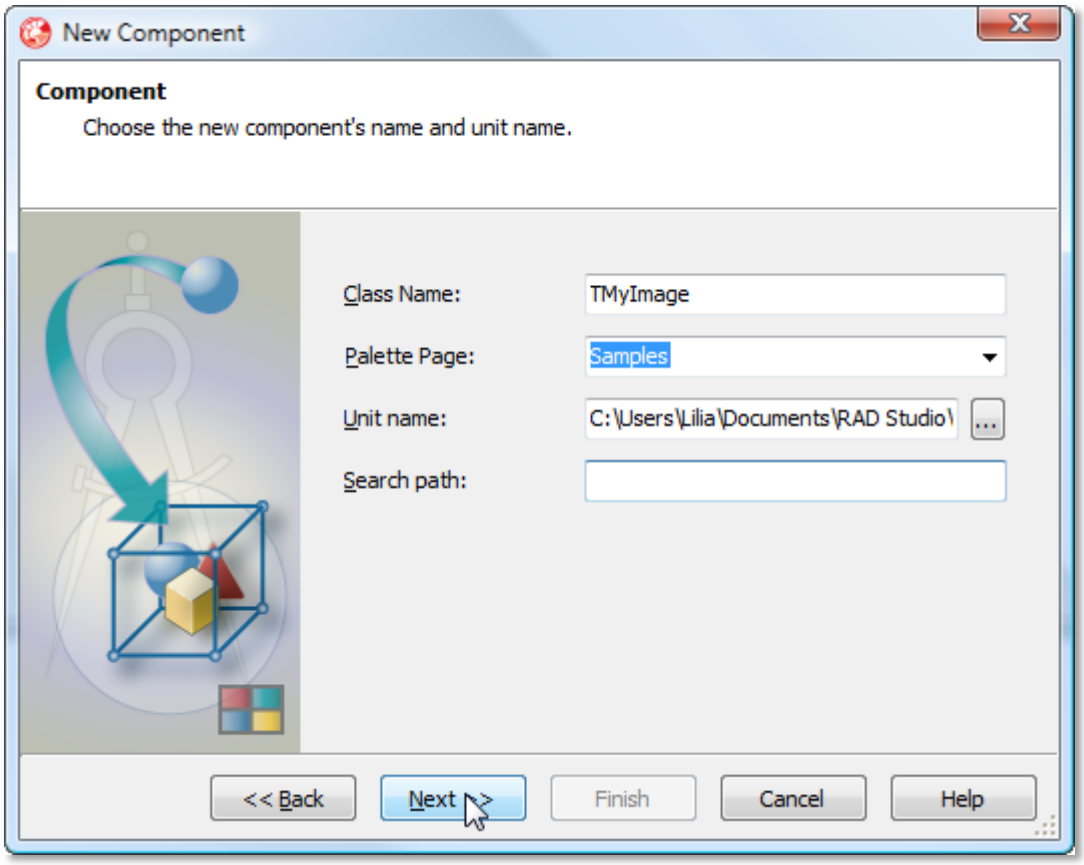


Рис. П7.16. Выбираем имя и место жительства

Нажимаем кнопку **Finish** (Рис. П7.17).

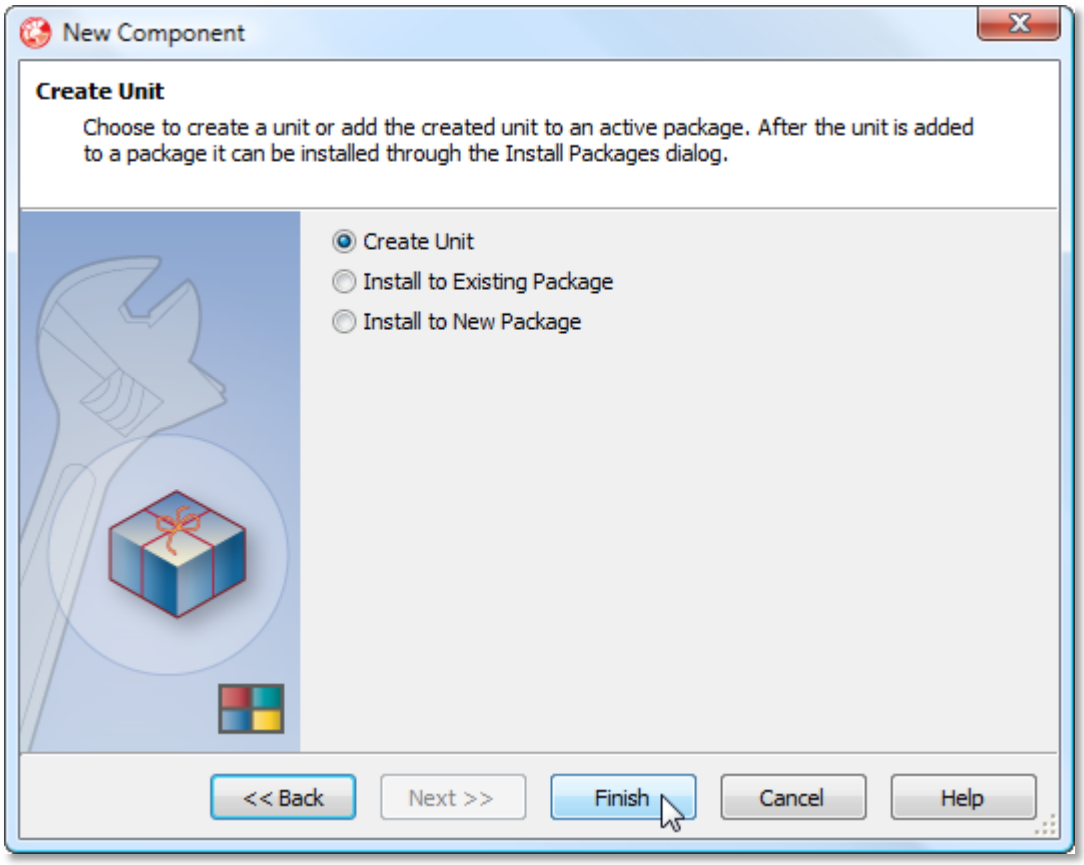


Рис. П7.17. Выбираем имя и место жительства

И получаем «пустую» заготовку для компонента:

```
unit Image1;

interface

uses
  System.SysUtils, System.Classes, Vcl.Controls, Vcl.ExtCtrls;

type
  TMyImage = class(TImage)
  private
    { Private declarations }
  protected
    { Protected declarations }
  public
    { Public declarations }
  published
    { Published declarations }
  end;

procedure Register;
```

```
implementation

procedure Register;
begin
  RegisterComponents('Samples', [TMyImage]);
end;

end.
```

Скопируем в неё код нашей картинки:

```
unit MyImage;

interface

uses
  SysUtils, Classes, Controls, ExtCtrls;

type
  TMyImage = class(TImage)
  private
    { Private declarations }
  protected
    { Protected declarations }
  public
    { Public declarations }
    x, y: Integer;                // - координаты клетки на игровом
поле
    clicked : Boolean;            // = TRUE, если клетка уже "клик-
нута"
    is_treasure: Boolean;           // = TRUE, если в этой клетке
находится клад
    nSosed: Integer;              // - число кладов на прямых, про-
ходящих через клетку
    //procedure Click; override; //- обработчик события OnClick
   published
    { Published declarations }
  end;

procedure Register;

implementation

procedure Register;
begin
  RegisterComponents('Samples', [TMyImage]);
end;

end.
```

Как видите, пришлось выбросить процедуру-обработчик щелчка на картинке, так как он содержит переменные основной программы.

Дальше создаём новую библиотеку компонентов, включаем в неё модуль **MyImage.pas** и устанавливаем её так, как было описано выше. На странице **Samples** появляется новый компонент, который можно устанавливать на форму в режиме проектирования (Рис. П7.18).

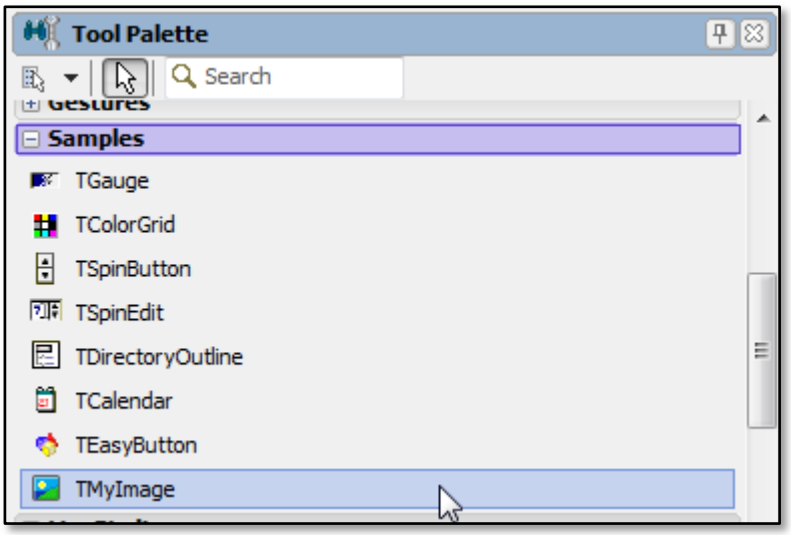


Рис. П7.18. С новым компонентом!

Для нового компонента вы можете написать код в его процедуре-обработчике любого события, как для всех других компонентов:

```
procedure TForm1.MyImage1Click(Sender: TObject);
var n: integer;
begin
  MyImage1.x:= 12;
  n:= MyImage1.x;
  listbox1.Items.Add(inttostr(n));
  MyImage1.y:= 11;
  n:= MyImage1.y;
  listbox1.Items.Add(inttostr(n));
  MyImage1.clicked:= TRUE;
  listbox1.Items.Add(booltostr(MyImage1.clicked));
end;
```

Для проверки компонента создадим новый проект и поместим его на форме вместе со списком (Рис. П7.19).

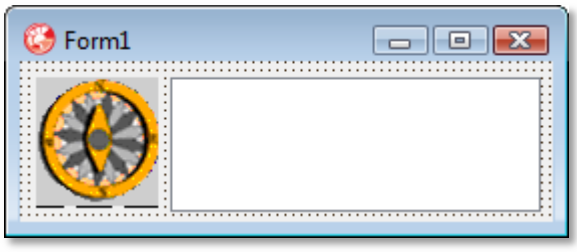


Рис. П7.19. Новый проект с новым компонентом

Запускаем программу, щёлкаем по картинке и получаем правильную реакцию (Рис. П7.20).

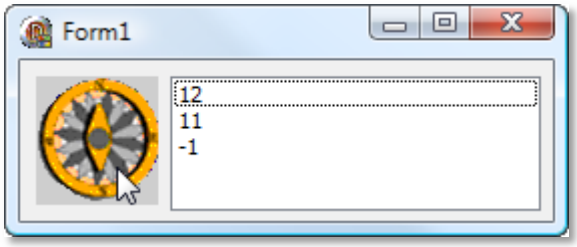


Рис. П7.20. Новый компонент в работе

Аналогично вы можете плодить наследников от любых компонентов, включая собственные! В случае необходимости их можно создавать и динамически, поэтому новый компонент можно использовать и в нашей программе.



Исходный код программы находится в папке **MyImage**.

Прячем клады, или Статистика знает всё!

Этот раздел – для особо любознательных, поэтому вы можете пропустить его, если к таковым себя не относите.

Подсчитаем число вариантов расположения кладов на поле.

Пусть размеры поля - **N x N** клеток, а число кладов равно **n**. Для **n=1** всё просто: клад можно поместить в любую клетку, то есть всего существует **R = N x N** способ спрятать 1 клад. Если **n=2**, то первый клад можно спрятать в любой из **N x N** клеток поля, и второму остаётся **N x N - 1** клетка. Таким образом, для двух кладов имеется **R= (N x N) * (N x N -1)** способов. Но все ли они различны? Посмотрим на рисунок (Рис. П7.21).

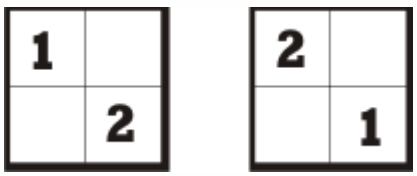


Рис. П7.21. Прячем 2 клада

Клады у нас не нумерованы, поэтому обе «карты сокровищ» совершенно одинаковы. Это значит, что мы должны разделить **R** на число возможных

перестановок кладов, то есть на **n факториал**. Для **n=2** имеем **R = 4*3/2= 6**. Все карты легко нарисовать и пересчитать (Рис. П7.22).

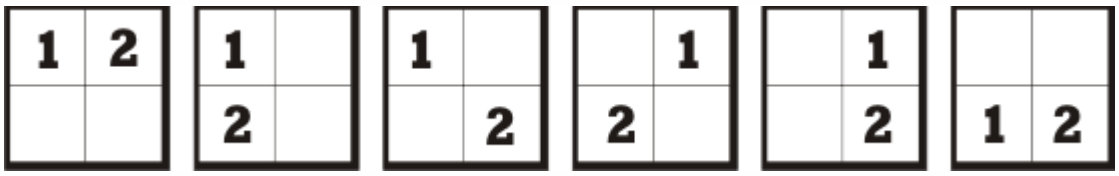


Рис. П7.22. Считаем варианты

Обозначим **K = N * N**, тогда **R (K,n) = K * (K-1)*(K-2)…* … (K-n+1) / n!** Или компактнее

R (K,n) = K! / (K-n)! n!

Например, R (9,1) = 9! / 8! 1! = 9;
R (9,2) = 9! / 7! 2! = 36, и так далее.

Для **n=2** можно найти и более красивую формулу, так как число способов зарыть клад образует арифметическую прогрессию **1..K-1**, откуда

R (K,2) = K * (K-1) / 2.

Конечно, считать по таким формулам вручную не очень приятно, поэтому напишем небольшую программу, которая будет делать это за нас.

Считать варианты разумнее по длинной формуле, иначе придётся вычислять большой факториал **K!** (число **K! / (K-n**)!), понятно, почти всегда меньше).

Сначала найдём произведение **K * (K-1)*(K-2)…* … (K-n+1)** ,

```
  for i := nCell downto nCell - nTreasures+1 do begin
    nVarFloat:= nVarFloat * i;
    nVar:= nVar * i;
  end;
```

затем разделим его на **n**!

```
  for i := 2 to nTreasures do begin
    nVarFloat:= nVarFloat / i;
    nVar:= nVar div i;
  end;
```

Здесь для большего понимания **nCell = K, nTreasures = n, nVar** и **nVarFloat = R**. Легко представить, что количество вариантов может достигать огромных величин, для которых в *Delphi* нет соответствующих типов це-

лых чисел, поэтому лучше перемежать умножение и деление - так мы немного «продлим удовольствие». Впрочем, ненадолго, так что параллельно будем считать варианты и «приблизительно». В итоге получится такая процедура:

```
//ПОДСЧИТАТЬ ЧИСЛО ВАРИАНТОВ РАЗМЕЩЕНИЯ
//КЛАДОВ НА ПОЛЕ
procedure TForm1.btnNumVarClick(Sender: TObject);
var i: integer;
    nVar: int64;
    nVarFloat: Extended;
    nDiv: integer;
begin
   nVar:= 1;
   nVarFloat := 1;
   nDiv:= 1;
   for i := nCell downto nCell - nTreasures+1 do begin
     nVarFloat:= nVarFloat * i;
     nVar:= nVar * i;
     if nDiv<= nTreasures then begin
       nVarFloat:= nVarFloat / nDiv;
       nVar:= nVar div nDiv;
       inc(nDiv);
     end;
   end;
   lblNumVar.Caption:= inttostr(nVar);
   lblNumVarFloat.Caption:= floattostr(nVarFloat);
end;
```

Так можно вычислить, например, **точное** число способов размещения 12 кладов на поле 12 х 12 клеток (**1036192938247073 88**), после чего потенциал *Delphi* оказывается полностью исчернанным. Можно воспользоваться услугами арифметики длинных чисел, но ничего нового мы всё равно не узнаем – в большинстве случаев количество способов зарыть клады просто огромно!

Остался неразрешённым ещё один интересный вопрос (по крайней мере, для нас он важнее): найти не только **число** способов зарыть клады, но и **все** эти способы. «В свете последних событий» понятно, что в данном случае речь может идти только о таких комбинациях размеров поля и числа кладов, когда количество вариантов не слишком велико.

Чтобы не изобретать «велосипед» (читай: алгоритм), воспользуемся тем, что имеется в главе, посвящённой *Японскому кроссворду*. У нас этот алгоритм получится даже проще, поскольку все «группы» одинаковые и содержат только 1 клетку-клад. Также мы должны учесть, что между кладами не обязательно должен быть «зазор» в виде пустой клетки.

Нажимаем кнопку **Разместить клады**:

```
//НАЙТИ ВСЕ РАЗМЕЩЕНИЯ КЛАДОВ НА ПОЛЕ
procedure TForm1.btnRazmClick(Sender: TObject);
var i,j,n: integer;
begin
  for j:= 0 to MAX_FIELDHEIGHT - 1 do
    for i:= 0 to MAX_FIELDWIDTH - 1 do
      Pole[i,j]:= NUM_GRAY;

  n:= TestingOneLine(nTreasures,PoleWidth*PoleHeight);
  lblNumVar.Caption:= inttostr(n);
end;
```

И в процедуре-обработчике окрашиваем все клетки поля в серый цвет. Это нужно для того, чтобы «посторонние клетки» не мешали наблюдать за полем заданных размеров (так как программа «несерьёзная», то мы не станем её «усугублять» лишними «удобствами»: таблица-поле всегда будет максимальных размеров *MAX_FIELDWIDTH x MAX_FIELDHEIGHT*):

```
const
  NAME_PROG = 'Variants';
  MAX_FIELDWIDTH = 12;
  MAX_FIELDHEIGHT = 12;
  MAX_CELLNUM = MAX_FIELDWIDTH*MAX_FIELDHEIGHT;
  NUM_GRAY = 0;
  NUM_WHITE = 1;
  NUM_BLACK = 2;
```

Затем мы вызываем доработанную функцию **TestingOneLine**, которая считает и показывает варианты сокрытия кладов:

```
//------ ищем подходящие комбинации клеток в ряду --------
function TestingOneLine(nGroup: integer; LenLine: integer): inte-
ger;
  //функция возвращает количество возможных расстановок кладов на
поле
  //nGroup: integer;  // число (0..) кладов на поле
  //LenLine: integer; // число клеток на поле

  label MoveLastGroup,  PredMove;
  var
    ptr: integer; // номер (0..nGroup-1) сдвигаемой группы
    pos: array[-1..99] of integer; //позиция клада на поле
    nVar: int64;  //число вариантов размещения
    i: integer;
    s: string;

    //показать расстановку:
    function WriteVariant(): Boolean;
```

```
    var i, j: integer;
    begin
      Result:= TRUE;
      //выставить белые клетки:
      for i:= 0 to LenLine-1 do
        Pole[i mod PoleWidth, i div PoleWidth]:= NUM_WHITE;
      //расставить все чёрные клетки - клады:
      for i:= 0 to nGroup-1 do
        Pole[pos[i] mod PoleWidth, pos[i] div PoleWidth]:=
NUM_BLACK;

      form1.dgPole.Invalidate;
      s:='Вариант: '+ inttostr(nVar);
      if form1.cbo1.Checked then
         application.MessageBox(PChar(s),NAME_PROG, MB_OK);

      form1.lstProtokol.Items.Add(s);
      For j:= 0 to PoleHeight-1 do begin
        s:='';
        For i:= 0 to PoleWidth-1 do
          if Pole[i, j]= NUM_WHITE
          then s:= s+ '.'
          else s:= s+ 'X';
        form1.lstProtokol.Items.Add(s);
      end;
      form1.lstProtokol.Items.Add('');

      if pos[0]> 0 then begin
         pos[nGroup]:= nCell;
         pos[-1]:= 0;
         s:=inttostr(nCell)+' = ';
         For i:= -1 to nGroup-1 do begin
           s:= s+ inttostr(pos[i+1]- pos[i]);
           if i < nGroup-1 then s:= s+ '+';
         end;
      end;

      form1.lstProtokol.Items.Add(s);
      form1.lstProtokol.Items.Add('');
    end; // WriteVariant

begin
  //расставляем клады в исходном положении - вплотную слева:
  For i:= 0 to nGroup do pos[i]:= i;
  //начальная расстановка групп - 1-й вариант:
  nVar:=1;
  //записать расстановку:
  WriteVariant;

  //начинаем сдвигать последнюю группу -
  MoveLastGroup:
    ptr:= nGroup-1;
    Inc(pos[ptr]);
```

```
    //передвигаем группу в след. клетку -
  while pos[ptr]+ 1<=LenLine do begin
      inc(nVar);
      //записать расстановку:
      WriteVariant;
      Inc(pos[ptr]);
  end;

  //сдвигать нельзя -->
  //переходим к предыдущей группе:
  PredMove:
    dec (ptr);
    if ptr<0 then //- все группы сдвинуты до края
    begin
      if form1.cbo1.Checked then begin
         s:='Все варианты: '+ inttostr(nVar);
         application.MessageBox(PChar(s),NAME_PROG, MB_OK)
      end;
      Result:= nVar;
      exit
    end;
    //ищем дальше -->
    //перемещаем начало текущей группы:
    inc(pos[ptr]);
    //проверяем:
    if pos[ptr]+ 1>LenLine then
    begin
      if form1.cbo1.Checked then begin
         s:='Все варианты: '+ inttostr(nVar);
         application.MessageBox(PChar(s),NAME_PROG, MB_OK)
      end;
      Result:= nVar;
      exit
    end;
    //расставляем следующие группы сразу после текущей:
    For i:= ptr+1 to nGroup-1 do
    begin
      pos[i]:= pos[i-1]+1;
      //проверить:
      if pos[i]+ 1>LenLine then //- для след. групп не хватает ме-
ста
        goto PredMove;
    end;
    //всё нормально - записать новую расстановку:
    inc(nVar);
    WriteVariant;
    //ищем след. растановку:
    Goto MoveLastGroup;
  end; // function TestingOneLine()
```

Первоначально поле имеет минимальные размеры 2 x 2 клетки, и клад спрятан только 1:

```
var
  Form1: TForm1;
  //Массив поля:
  Pole: array[0..MAX_FIELDWIDTH-1] of array[0..MAX_FIELDHEIGHT] of
integer;
  //Ширина поля:
  PoleWidth: integer=2;
  //Высота поля:
  PoleHeight: integer=2;
  //Количество клеток на поле:
  nCell: integer=4;
  //Количество кладов на поле:
  nTreasures: integer=1;
```

Для удобства экспериментов установим два движка. Одним будем изменять размеры поля:

```
//ИЗМЕНИТЬ РАЗМЕРЫ ПОЛЯ
procedure TForm1.tbWHChange(Sender: TObject);
begin
  PoleWidth:= tbWH.Position;
  PoleHeight:=PoleWidth;
  nCell:= PoleHeight * PoleWidth;
  lblWH.Caption:= inttostr(PoleWidth) + ' x '
+inttostr(PoleHeight);
  tbNumTreasures.Max:= nCell div 2;
end;
```

Другим – число кладов:

```
//ЗАДАТЬ ЧИСЛО КЛАДОВ
procedure TForm1.tbNumTreasuresChange(Sender: TObject);
begin
  nTreasures:= tbNumTreasures.Position;
  lblNumTreasures.Caption:= inttostr(nTreasures);
end;
```

Одновременно полезную информацию мы будем выводить в список **lstProtokol** (Рис. П7.23).

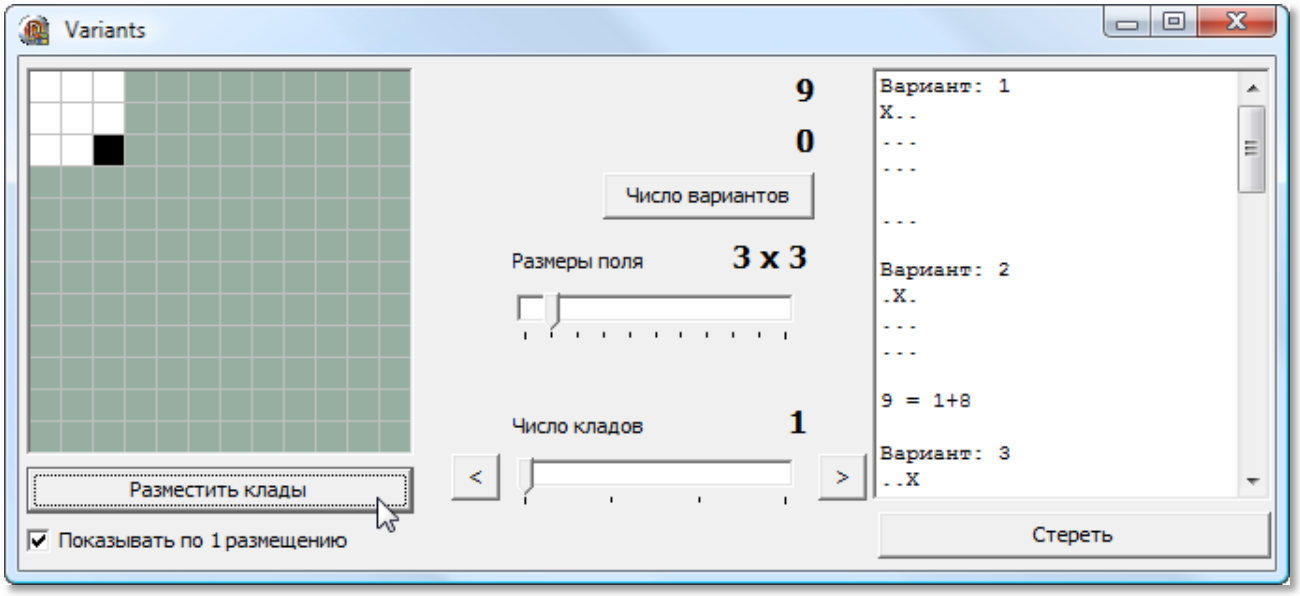


Рис. П7.23. Считаем варианты

Если внимательно присмотреться, то можно заметить, что одновременно мы решили ещё одну задачу – нашли все способы разбиения числа **nCell** на (**nTreasures+1**) слагаемых. Например, 13-й вариант размещения двух кладов на поле 3 x 3 клетки выглядит так (Рис. П7.24).



Рис. П7.24. Один из вариантов размещения двух кладов

Имеем: до первой чёрной клетки – одна белая, от первой чёрной клетки (включая её) до второй – 5 и от второй чёрной (включая её) до конца поля – ещё 3. Таким образом, получаем разбиение **9 = 1 + 5 + 3**. По ходу сокрытия кладов мы найдём и все остальные разбиения. Конечно, лучше было бы написать отдельную процедуру для этого, чтобы не перемешивать решения двух разных задач. Вот и сделайте!



Исходный код программы находится в папке **Variants**.

Шиворот-навыворот

Представим себе вздорную ситуацию, будто бы у нас имеется полная числовая информация о кладах на поле. Сможем ли мы определить, где находятся клады?

Начнём с самого простого случая: клад только один (**nTreasures=1**). Поскольку клетка с кладом при подсчёте кладов не учитывается, то в ней должен стоять нуль. Если «нулевая» клетка единственная, то клад в ней и находится. Если же таких клеток несколько, то последовательно предполагаем, что клад находится в ней, и проверяем числа во всех остальных клетках поля. Если они при этом размещении клада верные, то место клада найдено, иначе переходим к следующей клетке.

Проанализируем такую задачу (Рис. П7.25).

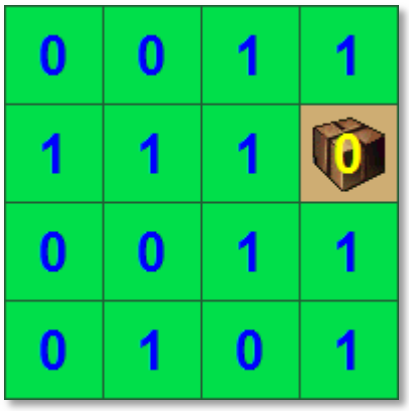


Рис. П7.25. Кладовая задача

Пусть клад находится в верхней левой клетке, тогда в следующей должна стоять единица, а не нуль. По этой же причине не годится и вторая клетка. В клетке с кладом, естественно, клад может находиться. Спускаемся на третью строку. Ни первый, ни второй нуль не годятся. Проверка в четвёртой строке приводит к тем же печальным результатам. Вывод: клад может находиться (а значит, и находится!) только в клетке, где находится картинка с кладом. Решение задачи, как видите, простое и единственное.

Очевидно, что решение принципиально не изменится и на поле других размеров. Правда, нулей придётся проверить поболее!

Возьмёмся за задачку посложнее, то есть уроем на поле 2 клада (**nTreasures=2**) (Рис. П7.26).

2	1	0	1
1	1	2	1
2	2	0	0
1	1	2	1

Рис. П7.26. Где-то здесь 2 клада!

Совершенно очевидно, что клад может находиться только в клетках с нулём и единицей, но никак не с двойкой (обоснование смотри выше). Закрасим эти клетки зелёным – в них кладов нет (Рис. П7.27).

2	1	0	1
1	1	2	1
2	2	0	0
1	1	2	1

Рис. П7.27. В зелёных клетках кладов нет

Чем ещё мы можем себе помочь? – Из условий игры следует, что в клетках, расположенных на прямых, проходящих через «нулевые» клетки, кладов нет. Их также можно закрасить (Рис. П7.28).

2	1	0	1
1	1	2	1
2	2	0	0
1	1	2	1

Рис. П7.28. И здесь их нет!

Осталось только 2 клетки на 2 клада – в них-то клады и зарыты (Рис. П7.29).



Рис. П7.29. Нашли!

Задача решена! Но она оказалась слишком простой, поэтому ещё больше испортим себе жизнь – спрячем на поле 5 х 5 клеток 3 клада (Рис. П7.30).



Рис. П7.30. Сложная задача!

«Вычёркиваем» тройки и нули (Рис. П7.31).



Рис. П7.31. Отмечаем пустые клетки

Да – кладов три, а клеток осталось 6. Вы можете сразу определить, где они находятся? – А вот я не могу. Предположим, что клад находится в первой же клетке – с двойкой (Рис. П7.32).

2	1	2	1	2
3	1	2	2	2
2	1	3	1	1
1	3	0	2	0
1	2	1	2	2

Рис. П7.32. Может быть, здесь?

Тогда в клетках с единицей, которые находятся в той же горизонтали, вертикали и диагонали кладов уже быть не может (Рис. П7.33).

2	1	2	1	2
3	1	2	2	2
2	1	3	1	1
1	3	0	2	0
1	2	1	2	2

Рис. П7.33. Продолжаем поиски!

Казалось бы, всё – задача решена, ведь осталось только 2 клетки. Но не торопитесь! Давайте проверим, выполняются ли условия задачи (Рис. П7.34).

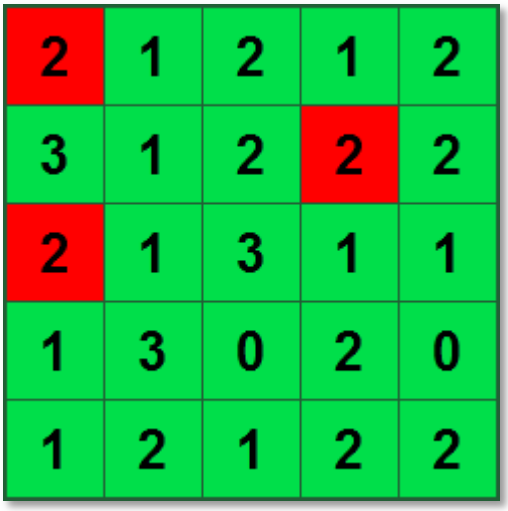


Рис. П7.34. Проверяем!

Достаточно попробовать первую строку, чтобы найти несуразности. В четвёртой клетке стоит 1, а указывает она на 2 клада! Значит, наше предположение оказалось ошибочным – в первой клетке клада нет (Рис. П7.35).



Рис. П7.35. В первой клетке клада нет!

Проверим вторую клетку (Рис. П7.36).



Рис. П7.36. Проверяем вторую клетку

Пока всё нормально, идём дальше (Рис. П7.37).

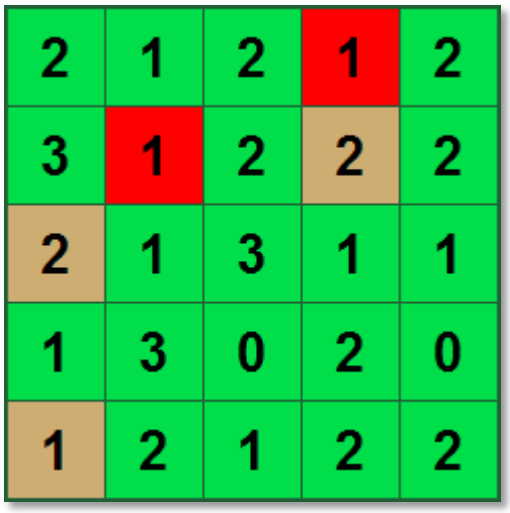


Рис. П7.37. Продолжаем проверку

Тогда в следующей клетке с двойкой клада нет. Опускаемся в третью строку. Если в первой клетке есть клад, то все клады расставлены, но эта клетка указывает только на 1 клад (Рис. П7.38).

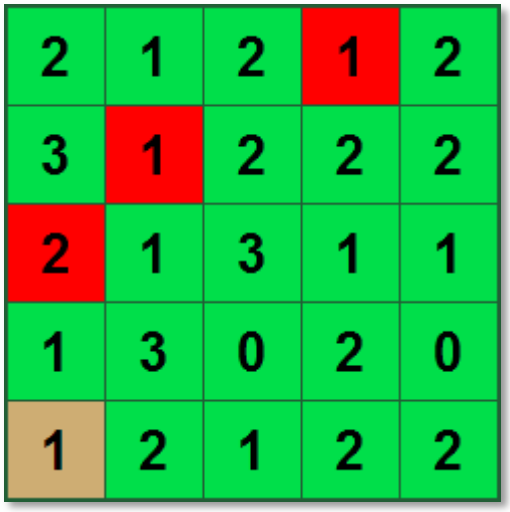


Рис. П7.38. Не сходится!

«Фокус» опять не удался, стало быть, и второе предположение было неудачным. Окрашиваем единицу в первой строке в зелёный цвет (Рис. П7.39).

2	1	2	1	2
3	1	2	2	2
2	1	3	1	1
1	3	0	2	0
1	2	1	2	2

Рис. П7.39. Единица должна быть зелёной!

Идём проторённым путём (Рис. П7.40).

2	1	2	1	2
3	1	2	2	2
2	1	3	1	1
1	3	0	2	0
1	2	1	2	2

Рис. П7.40. Размышления продолжаются

И дальше (Рис. П7.41).

2	1	2	1	2
3	1	2	2	2
2	1	3	1	1
1	3	0	2	0
1	2	1	2	2

Рис. П7.41. И продолжаются…

Двойку в третьей строке мы опять окрасили неверно, тогда так (Рис. П7.42).

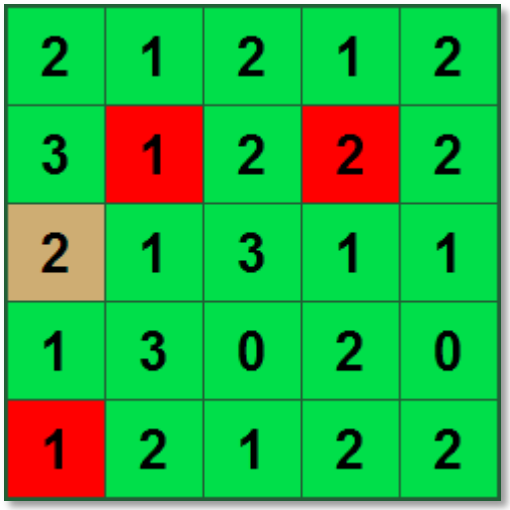


Рис. П7.42. Клады найдены!

Теперь всё красиво! Осталось проверить, нет ли других решений (Рис. П7.43).

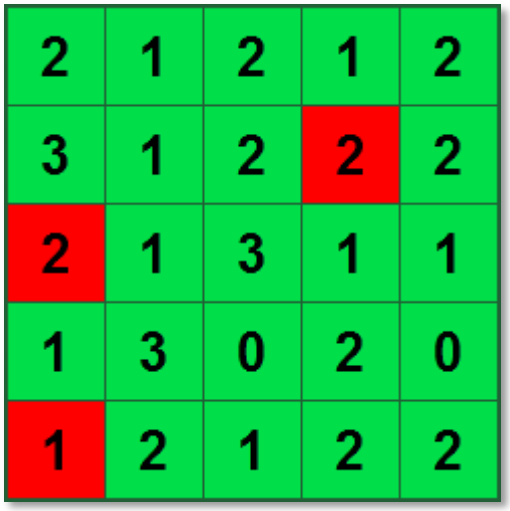


Рис. П7.43. Другое решение?

Эта расстановка не удовлетворяет условиям задачи, поэтому задача имеет *единственное* решение (Рис. П7.44).



Рис. П7.44. Правильное и единственное решение

Такое решение годится для людей систематических и усидчивых (мы, конечно, не такие, но и нам оно пригодится), для смекалистых есть решение короче. Двойка во второй клетке последней строки указывает ровно на две клетки, в которых могут быть клады – так они там и должны быть (Рис. П7.45).

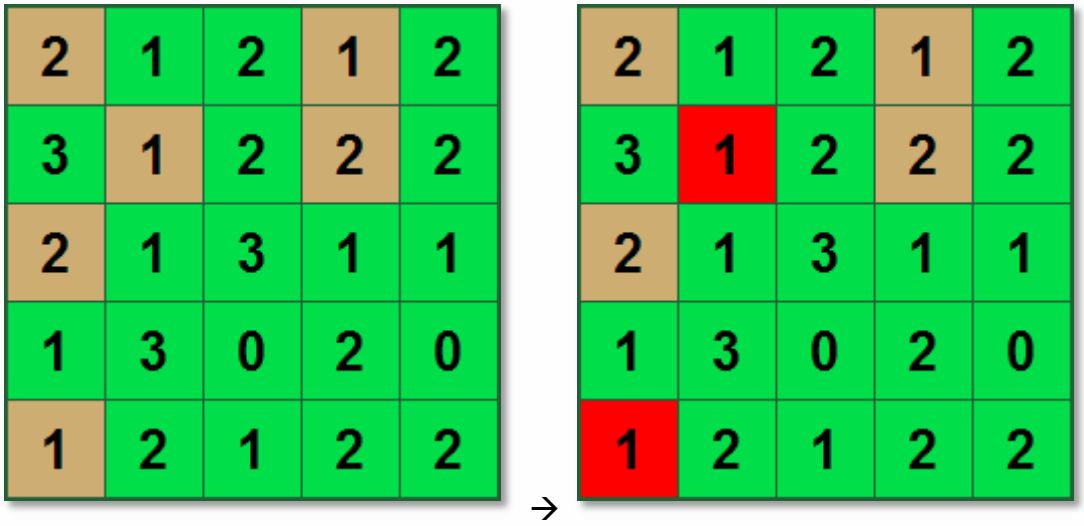


Рис. П7.45. Решаем по-хитрому**!**

Замечаем, что в первой колонке есть клетка с единицей и условие для неё уже выполнено, следовательно, остальные клетки в первой колонке – зелёные. Точно так же и на восходящей диагонали (Рис. П7.46).

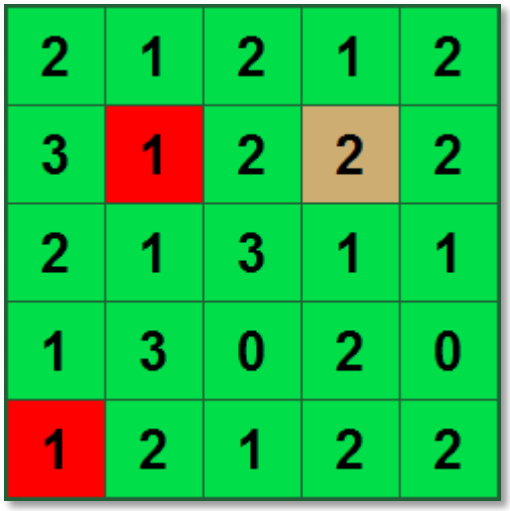


Рис. П7.46. Смотри зорко!

Для последнего клада осталась только одна клетка – задача решена (Рис. П7.47).

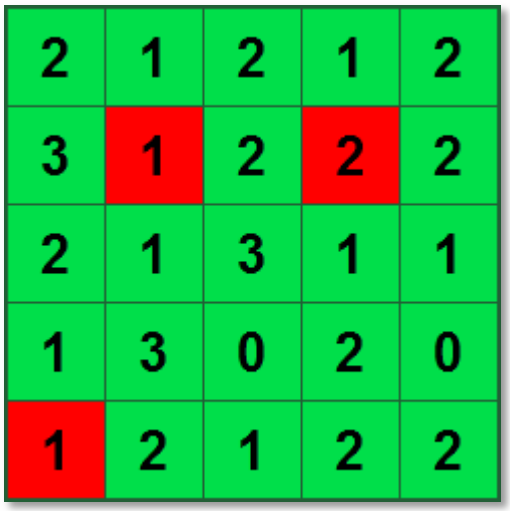


Рис. П7.47. Вот и вся задачка!

Логическое решение оказалось не просто интереснее и короче – оно избавило нас от проверки на единственность решения. «Мораль сей басни такова»: вручную задачу следует решать головой (парадокс!), а перебирать варианты должен компьютер в лице соответствующей программы!

К сожалению, мы уже и так далеко ушли от заявленной темы главы, поэтому закончите эту программу и напишите полноценный искатель кладов самостоятельно. Первая задача не очень сложная, а вот вторая…



Исходный код программы находится в папке **Patience**.

Часть III. Факультатив

Факультатив, как известно, дело серьёзное, поэтому здесь мы не будем останавливаться на мелочах и частностях, резонно полагая, что основы языка программирования и интегрированной среды разработки программ вы уже выучили назубок.

На первых трёх уроках вы более основательно, чем в первой части книги, познакомитесь с интерфейсом программы **Delphi XE3**, со структурой приложений и их файлами, научитесь настраивать среду программирования по своему вкусу, освоите главный компонент всех приложений – форму.

Следующие четыре урока факультативных занятий полностью отданы программированию игр и головоломок. Для этого вам вполне будет достаточно тех знаний, которые вы получили при чтении первых двух частей книги и изучении исходного кода программ-примеров. А здесь мы создадим **четыре законченных проекта** интереснейших игр. Написание этих программ не только занимательно само по себе, но и чрезвычайно полезно. В них вы найдёте очень много того, что пригодится вам в дальнейшем при разработке *собственных* проектов: как сохранить игру на диске, как загрузить её с диска, как добавить комментарии к текущей игре, как вести протокол игры, как вернуть неудачный ход, а – самое главное! – как придумать хороший алгоритм для решения головоломки…

И, наконец, последний, **девятый урок** познакомит вас с новой платформой **FireMonkey**, которая позволяет создавать быстрые кроссплатформенные приложения с удивительно красивым и легко изменяемым интерфейсом пользователя, а также приложения с **трёхмерной** графикой.

Факультатив 1. Фейсом к интерфейсу, или *Дельфи, открой личико!*

Кролик надел очки.
- С чего начинать, Ваше Величество? – спросил он.
- Начни с начала, - важно ответил Король, - и продолжай, пока не дойдёшь до конца.
Как дойдёшь – кончай!

Льюис Кэрролл.
Приключения Алисы в Стране Чудес

Создание и отладка программ (приложений) выполняется в **интегрированной среде разработки** (**ИСР**, или **IDE**) *Delphi*, которая состоит из нескольких окон с различными управляющими элементами (Рис. Ф1.1).

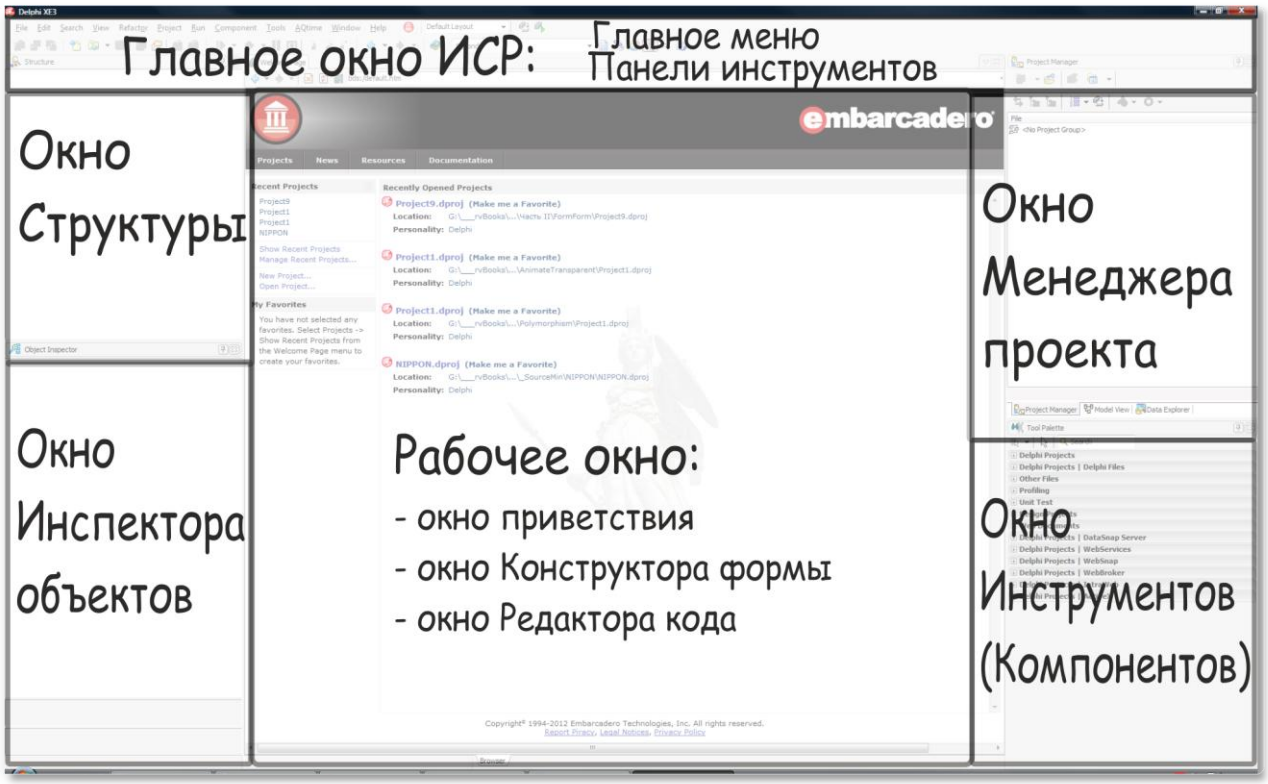


Рис. Ф1.1. Общий вид ИСР *Delphi XE3*

Сверху находится **Главное окно**, на нём размещаются **Главное меню** программы и **Панели инструментов**. В левой и правой частях экрана располагаются **панели**, а центральную часть занимает **Рабочее окно**.

Кроме представленных окон, в нижней части экрана в режиме проектирования и отладки приложения могут появляться окна сообщений и отлад-

ки. О них речь пойдёт дальше, а сейчас давайте серьёзно познакомимся с теми, что нам уже известны.

Главное окно

В строке заголовка *Главного окна* выводится название открытого файла.

При сворачивании *Главного окна* минимизируются и все остальные окна среды, при закрытии *Главного окна* работа с *Delphi* завершается.

Главное окно состоит из *Главного меню* и нескольких плавающих *Панелей инструментов*, которые можно передвигать и превращать в отдельное окно (щёлкните дважды на полосках слева; чтобы вернуть окно в исходное положение, подведите его к краю другой панели *Главного окна* и отпустите).

Главное меню (Main Menu)

Находится сразу под заголовком *Главного окна* и содержит несколько подменю с множеством команд (Рис. Ф1.2).



Рис. Ф1.2. Главное меню

File
- загрузка, сохранение файлов и другие операции (Рис. Ф1.3).

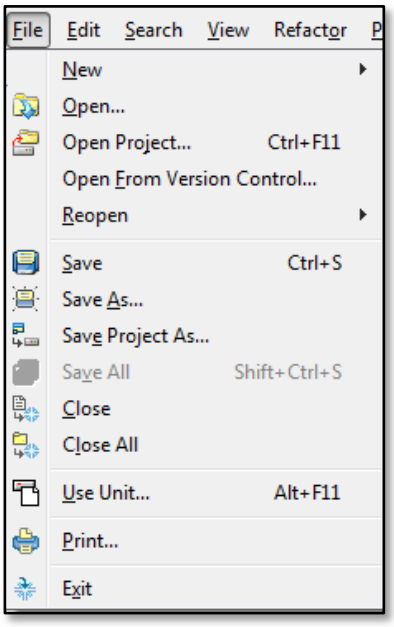


Рис. Ф1.3. Меню *File*

- **New**
  – начинает новый проект (Рис. Ф1.4).

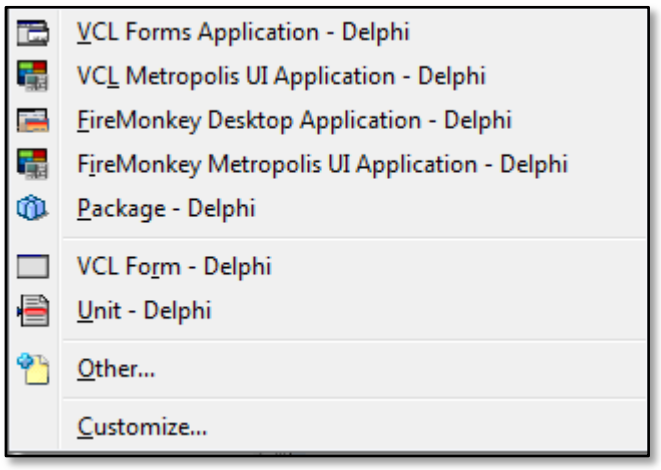


Рис. Ф1.4. Подменю *New*

  - **VCL Forms Application – Delphi** – начинает новый проект, при этом текущий закрывается, так как *Delphi* не позволяет работать одновременно с несколькими проектами.
  - **VCL Metropolis UI Application – Delphi** – начинает новый проект в стиле *Microsoft Windows 8*.
  - **FireMonkey Desktop Application – Delphi** – начинает новый проект *FireMonkey*.
  - **FireMonkey Metropolis UI Application – Delphi** – начинает новый проект *FireMonkey* в стиле *Microsoft Windows 8*.
  - **Package – Delphi** – начинает новую библиотеку компонентов.
  - **VCL Form – Delphi** – добавляет новую пустую форму к проекту.
  - **Unit – Delphi** – добавляет ещё один модуль к проекту.

- **Open**
  – открывает стандартное диалоговое окно загрузки файлов (Рис. Ф1.5).

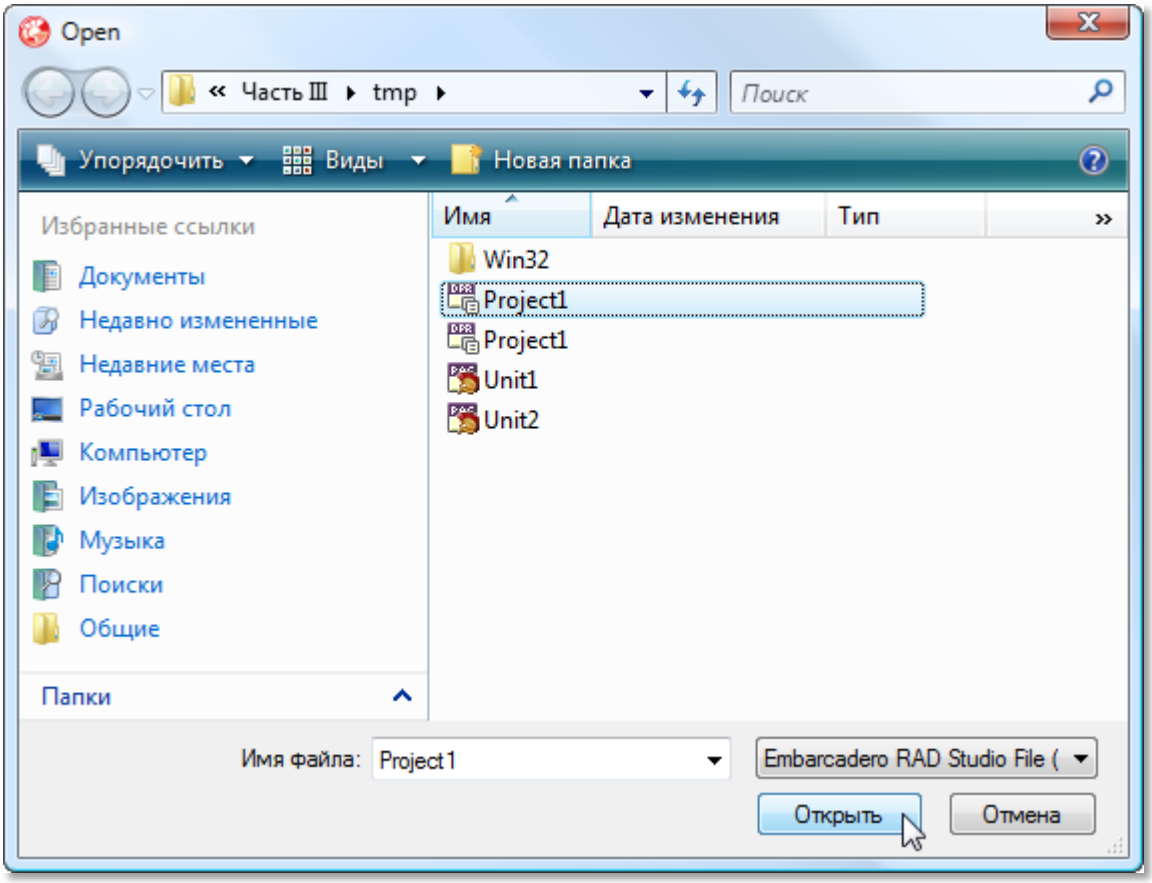


Рис. Ф1.5. Диалоговое окно выбора файлов

В нём можно выбрать существующий проект, форму, модуль, текст и загрузить их в среду *Delphi*.

- **Open Project** - загружает существующий проект *Delphi*.
- **Reopen** – очень полезная команда! Открывает список с названиями последних проектов (верхняя часть списка) и модулей, с которыми вы недавно работали, так что вам не придётся искать их по излюбленным папкам операционной системы *Windows*.
- **Save** – записывает на диск текущий файл (формы и модуля), находящийся в окне *Редактора кода* под текущим именем.
- **Save As** – позволяет ввести в диалоговом окне новое имя для текущего файла и сохранить его на диске.
- **Save Project As** – записывает текущий проект под новым именем.
- **Save All** – записывает все открытые файлы проекта.
- **Close** – закрывает текущий проект и все связанные с ним файлы.

- **Close All** – закрывает все открытые файлы.

- **Use Unit** – добавляет к активному (текущему) модулю выбранный модуль. Команда может пригодиться, когда вы открываете в проекте новую форму, которая упоминается в другом модуле проекта.

- **Print** – печатает текущий модуль (форму) или его часть на принтере. Полезно распечатать «упрямую» процедуру, когда не удаётся отладить её за компьютером, чтобы подумать в спокойной обстановке. Этот приём часто помогает отыскать ошибки в алгоритме или коде. Вместо этой команды можно просто перенести нужную часть кода в буфер обмена и распечатать в любом текстовом редакторе (в этом случае выделять синтаксические особенности (цвет элементов кода) программы на *Delphi* вам придётся самостоятельно).

  В диалоговом окне вы можете задать параметры печати и настроить принтер.

- **Exit** – закрывает проект и *Delphi*.

Edit

– редактирование кода и компонентов на форме (Рис. Ф1.6).

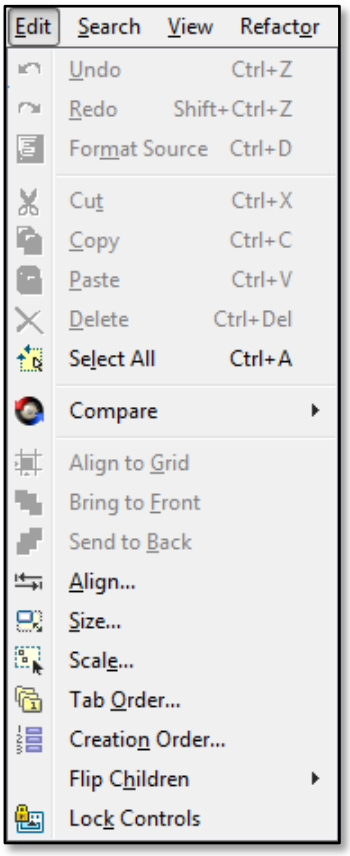


Рис. Ф1.6. Меню *Edit*

Команды редактирования используются при разработке программ очень часто, поэтому желательно запомнить клавиатурные сокращения для них, тем более что они точно такие же, как и в большинстве текстовых редакторов.

Остальные команды этого меню служат для работы с компонентами на форме, но гораздо удобнее открыть контекстное меню *Редактора кода*, в котором присутствуют все эти команды (впрочем, потребность в них не очень большая).

Search

– поиск и замена последовательности символов (текста) в исходном коде (Рис. Ф1.7).

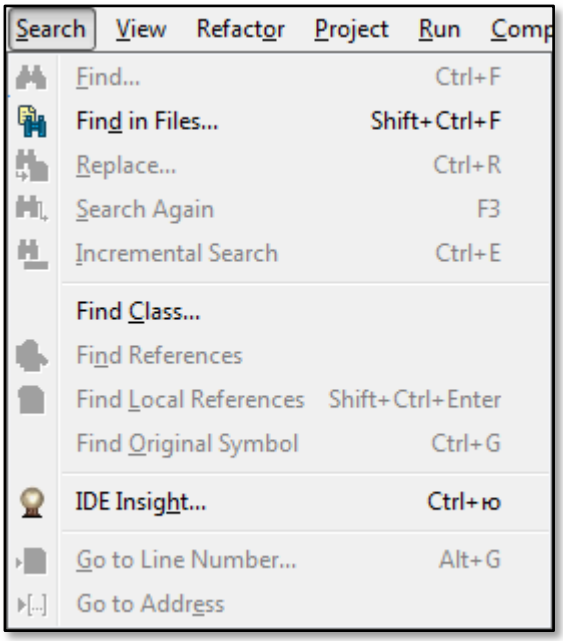


Рис. Ф1.7. Меню Search

- **Find** – отыскивает заданный текст в окне *Редактора кода*. Очень полезная команда, которая подробно рассматривается дальше. Чтобы эта команда всегда была под рукой, можно посоветовать установить кнопку **Find** на *Панель инструментов*.

- **Find in Files...** - отыскивает заданный текст в указанном файле или группе файлов на диске и выводит найденные строки в окне *Редактора кода*.

- **Replace** – заменяет одну группу символов на другую. Полезна при изменении названия компонента, процедуры и других элементов кода.

- **Search Again** – повторяет (продолжает) предыдущий поиск.
- **Go to Line Number** – переходит на строку текста с указанным номером.

View
– работа с окнами и панелями ИСР (Рис. Ф1.8).

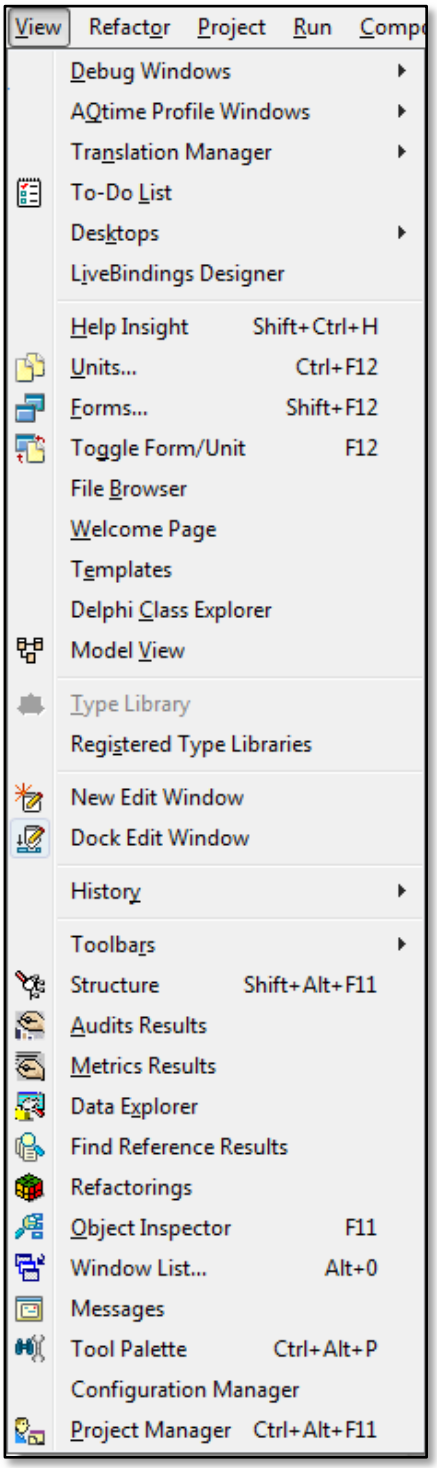


Рис. Ф1.8. Меню *View*

- **Project Manager** (**Ctrl+Shift+F11**) – выводит на экран окно *Менеджера проекта*.
- **Tool Palette** (**Ctrl+Alt+P**) – выводит на экран *Палитру компонентов*.
- **Object Inspector** (**F11**) – открывает окно *Инспектора объектов*.
- **Window List...** (**Alt+0**) – выводит на экран список открытых окон.
- **Debug Windows** - открывает отладочные окна (Рис. Ф1.9).

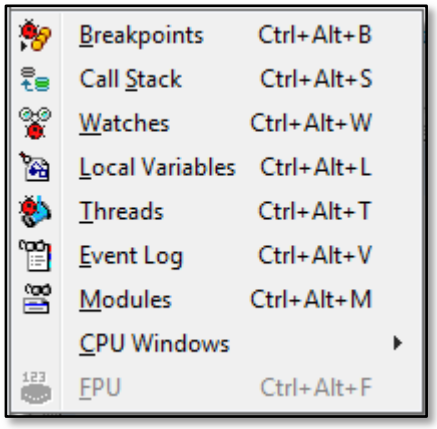


Рис. Ф1.9. Подменю *Debug Windows*

Через команды меню и отладочные окна можно управлять работой встроенного отладчика *Delphi*. Так, команда **Breakpoints** (**Ctrl+Alt+B**) позволяет устанавливать точки останова, при достижении которых запущенная программа будет приостанавливаться с выводом указанной информации. С помощью команды **Watches** (**Ctrl+Alt+W**) можно отслеживать значения переменных в окне просмотра. Установка параметров встроенного отладчика осуществляется в диалоговом окне, которое вызывается при выполнении команды *Главном меню* **Tools > Options > Debugger Options**.

- **Desktops** – сохраняет текущий вид (раскладку окон и панелей) ИСР.
- **Units**– выводит на экран список модулей.
- **Forms**– выводит на экран список форм.
- **Toggle Form/Unit (F12)** – переключается между окнами *Конструктора формы* и *Редактора кода*.
- **Welcome Page** – выводит на экран *Страницу приветствия*.

- **New Edit Window** – откреплять окна *Редакторе кода* и *Конструктора формы* от *Главного окна*.

- **Dock Edit Window** – прикреплять окна *Редакторе кода* и *Конструктора формы* к *Главному окну*.

- **Toolbars** – удаляет или отображает указанные *Панели инструментов* в *Главном окне*.

Refactor

- рефакторинг исходного кода приложения: переименование переменных и другие операции по изменению текста (Рис. Ф1.10).

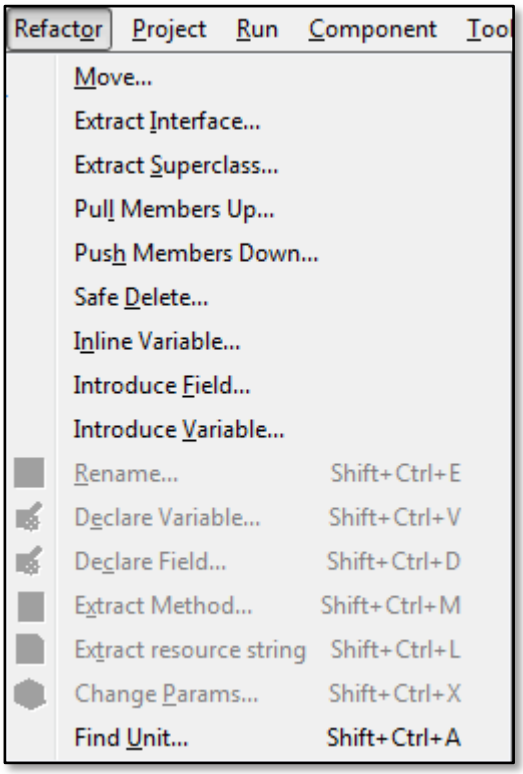


Рис. Ф1.10. Меню *Refactor*

Project

– работа с файлами проекта (Рис. Ф1.11).

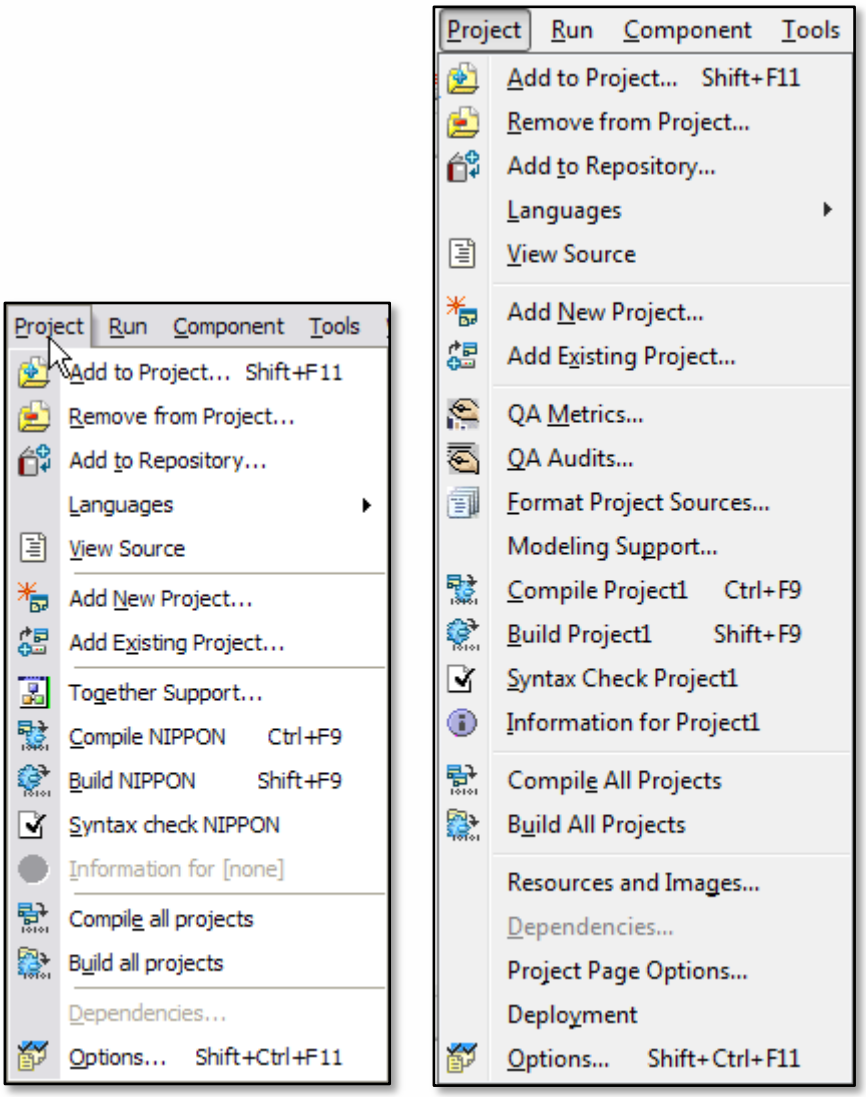


Рис. Ф1.11. Меню *Project*

- **Add to Project... (Shift+F11)** – добавляет файл к проекту.
- **Remove from Project...** – удаляет файл из проекта.
- **Add to Repository...** – добавляет проект в *Репозиторий*.
- **View Source** – выводит в окне *Редактора кода* файл проекта.
- **Add New Project...** – добавляет новый проект в *Менеджер проекта*.
- **Add Existing Project...** – добавляет существующий проект в *Менеджер проекта*.
- **Compile...** (**Ctrl+F9**) – компилирует только те файлы текущего проекта, которые были изменены со времени предыдущей компиляции. Компилировать проект можно на любой стадии разработки проекта

– для проверки правильности кода и просмотра вида формы в действующей программе. При компиляции проекта для каждого модуля создаётся файл с расширением *DCU*, после чего компилируется файл проекта и формируется исполняемый файл с именем файла проекта и расширением *EXE*.

- **Build…** (**Shift+F9**) – создаёт приложение с компиляцией **всех** модулей проекта – как изменённых со времени последней компиляции, так и оставшихся без изменения.

- **Options…** (**Ctrl+Shift+F11**) – открывает диалоговое окно настройки параметров проекта. О нём более подробно мы поговорим в следующей главе.

Run

– запуск и отладка программы в среде *Delphi* (Рис. Ф1.12).

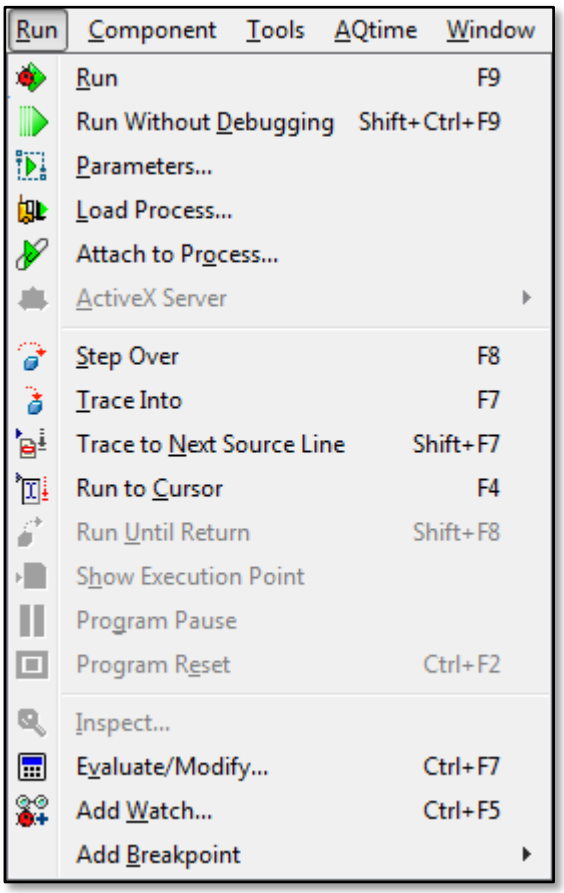


Рис. Ф1.12. Меню *Run*

- **Run** (**F9**) – компилирует (если были внесены изменения в файлы проекта после предыдущего старта программы) и запускает приложение. Скомпилированное приложение можно запустить как из сре-

ды *Delphi*, так и из среды *Windows*. При этом в работе приложения будут некоторые отличия. Так, **при запуске из Delphi** программа выполняется медленнее; нельзя запустить копию приложения; при зависании программы завершать её нужно только средствами *Delphi* – выполнить команду *Главного меню* **Run > Program Reset** (или нажать комбинацию клавиш **Ctrl+F2**). Если ваша программа «склонна к зависанию», то безопаснее её запускать из среды *Windows*.

- **Attach to Process...** – выводит список запущенных программ, в котором можно выбрать нужную для отладки.

- **Step Over (F8)** – запускает программу и выполняет по одной строке кода без захода в процедуры. Используется при отладке программ.

- **Trace Into (F7)** – действует аналогично предыдущей, но с заходом в процедуры.

- **Trace To Next Source Line (Shift+F7)** - построчно выполняет исходный код программы.

- **Run To Cursor (F4)** – выполняет программу с обычной скоростью, пока не подойдёт очередь строки с исполняемым кодом, в которой находился курсор перед стартом этой команды. Здесь программа покажет эту строку в *Редакторе кода*, и можно будет узнать значения переменных, а затем продолжить выполнение программы командами **Step Over** и **Trace Into**.

- **Run Until Return (Shift+F8)** – используется, когда программа остановлена для отладки командой **Program Pause**.

- **Program Pause** – приостанавливает исполнение программы для отладки. Затем процесс может быть продолжен командами **Run**, **Step** или **Trace**.

- **Program Reset (Ctrl+F2)** – заканчивает работу программы и выгружает её из памяти компьютера. Команда очень полезна при отладке программы, когда она зацикливается или зависает.

- **Inspect...** – открывает диалоговое окно **Inspect** (сначала остановите программу для отладки командой *Program Pause* или предварительно добавьте точку останова в исходном коде), в котором вы можете ввести выражение или идентификатор и, нажав кнопку *OK*, получить их текущее значение (Рис. Ф1.13).

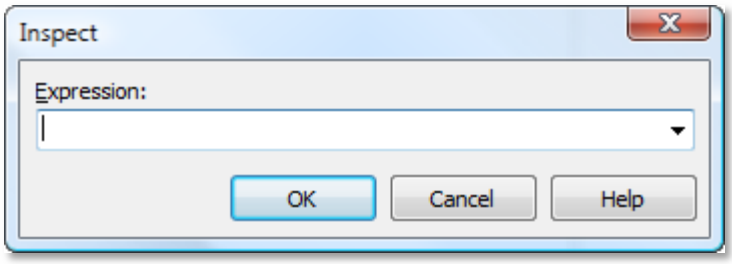


Рис. Ф1.13. Диалоговое окно *Inspect*

Продолжить выполнение программы можно, как для **Program Pause**. Эту же команду можно вызвать из контекстного меню *Редактора кода* – **Debug > Inspect**.

- **Evaluate/Modify (Ctrl+F7)** – действует, как предыдущая команда, но предоставляет больше возможностей. Например, вы можете изменить значение переменной.

- **Add Watch... (Ctrl+F5)** – открывает диалоговое окно **Watch Properties**, в котором можно создать и изменить переменные или выражения. Их значения затем будут отслеживаться при отладке программы. Используйте также команду контекстного меню *Редактора кода* **Debug > Add Watch at Cursor**.

- **Add Breakpoint** – открывает диалоговое окно, в котором вы можете создать или изменить точку останова.

Component

– работа с компонентами (Рис. Ф1.14).

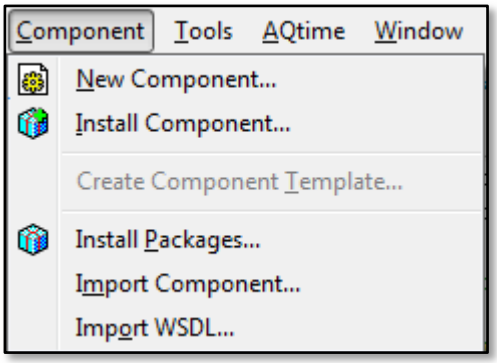


Рис. Ф1.14. Меню *Component*

- **New Component...** – открывает диалоговое окно для создания нового компонента.

- **Install Component…** – открывает диалоговое окно для установки нового компонента.

- **Create Component Template…** – позволяет создать новый компонент из уже имеющихся на форме.

- **Install Packages…** – даёт возможность установить новые библиотеки компонентов.

Tools
– инструменты и настройки (Рис. Ф1.15).

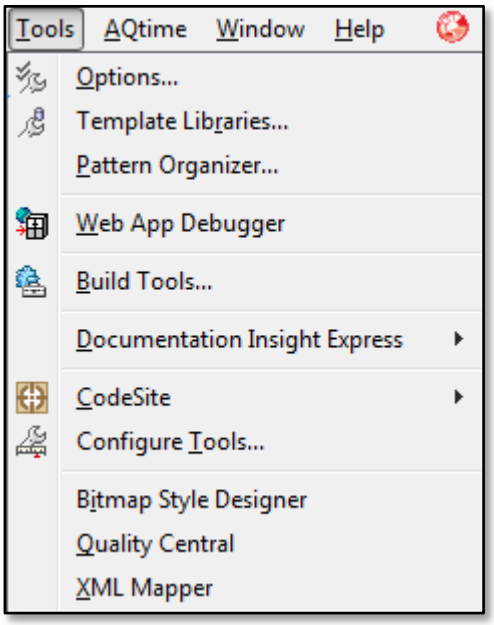


Рис. Ф1.15. Меню *Tools*

- **Options… -** открывает окно с несколькими группами настроек:

  - **Environment Options** – настройки ИСР.

  - **Editor Options** – настройки *Редактора кода*.

  - **Debugger Options** – настройки отладчика в диалоговом окне (Рис. Ф1.16).

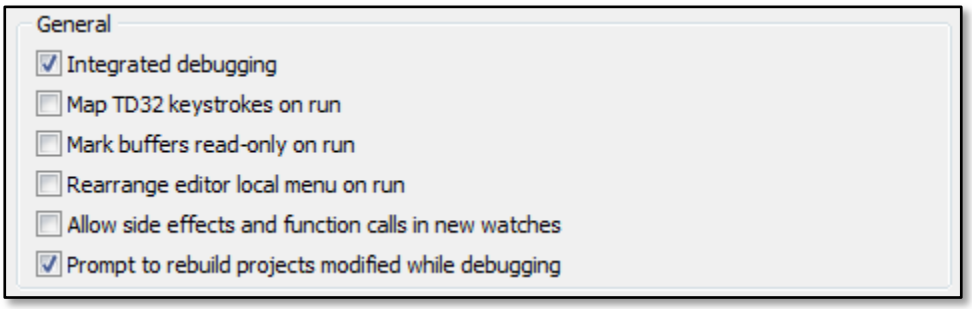


Рис. Ф1.16. Диалоговое окно *Debugger Options*

При установленном флажке **Integrated debugging** встроенный отладчик *Delphi* автоматически подключается к каждому приложению и фиксирует возникающие в процессе работы приложения ошибки. Если вы используете в программе собственные обработчики ошибок, то этот флажок следует сбросить.

- **Template Libraries...** – открывает диалоговое окно **Репозитория** (хранилища) объектов (Рис. Ф1.17).

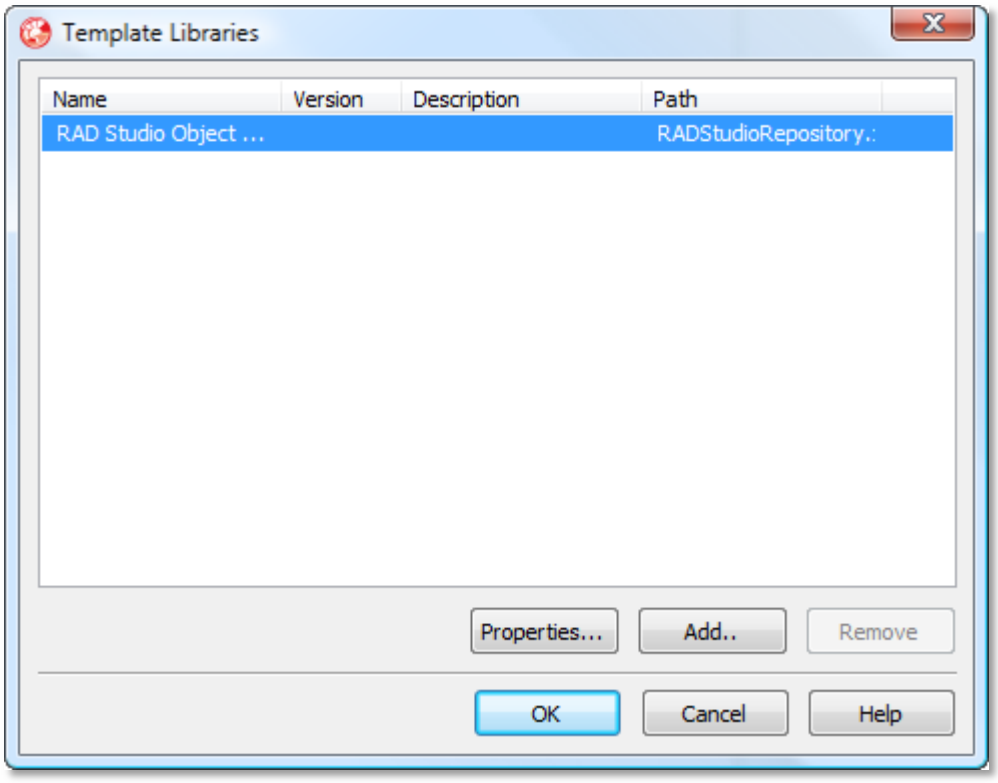


Рис. Ф1.17. Диалоговое окно *Template Libraries*

Нажмите кнопку **Properties...** и в диалоговом окне **Object Repository** выберите тип приложения – *MDI/SDI* (Рис. Ф1.17а).

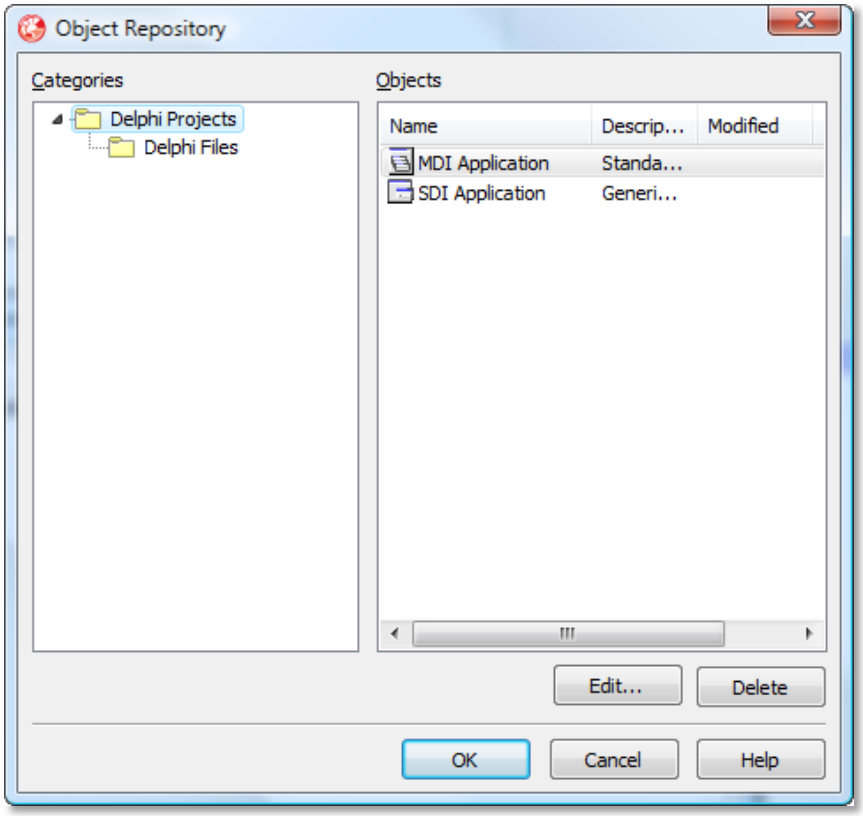


Рис. Ф1.17а. Диалоговое окно *Object Repository*

Вы также можете раскрыть папку **Delphi Files** (Рис. Ф1.17б) и выбрать нужный проект из списка.

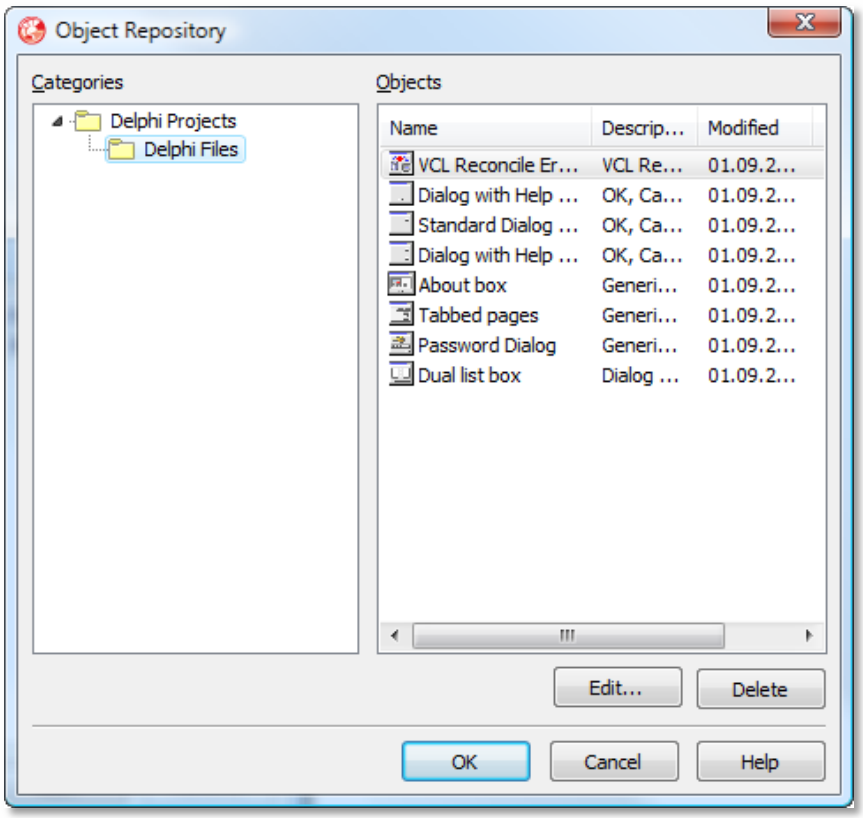


Рис. Ф1.17б. Папка *Delphi Files*

Delphi позволяет использовать объекты в качестве *шаблонов* при разработке других приложений. Для этого нужно сохранить выбранный объект в *Репозитории*. Сделать это можно из контекстного меню *Конструктора формы* (команда **Add to Repository...**) или из *Главного меню* командой **Project > Add to Repository...** При выполнении команд меню вам будет предложено ввести необходимую информацию в диалоговом окне (Рис. Ф1.18).

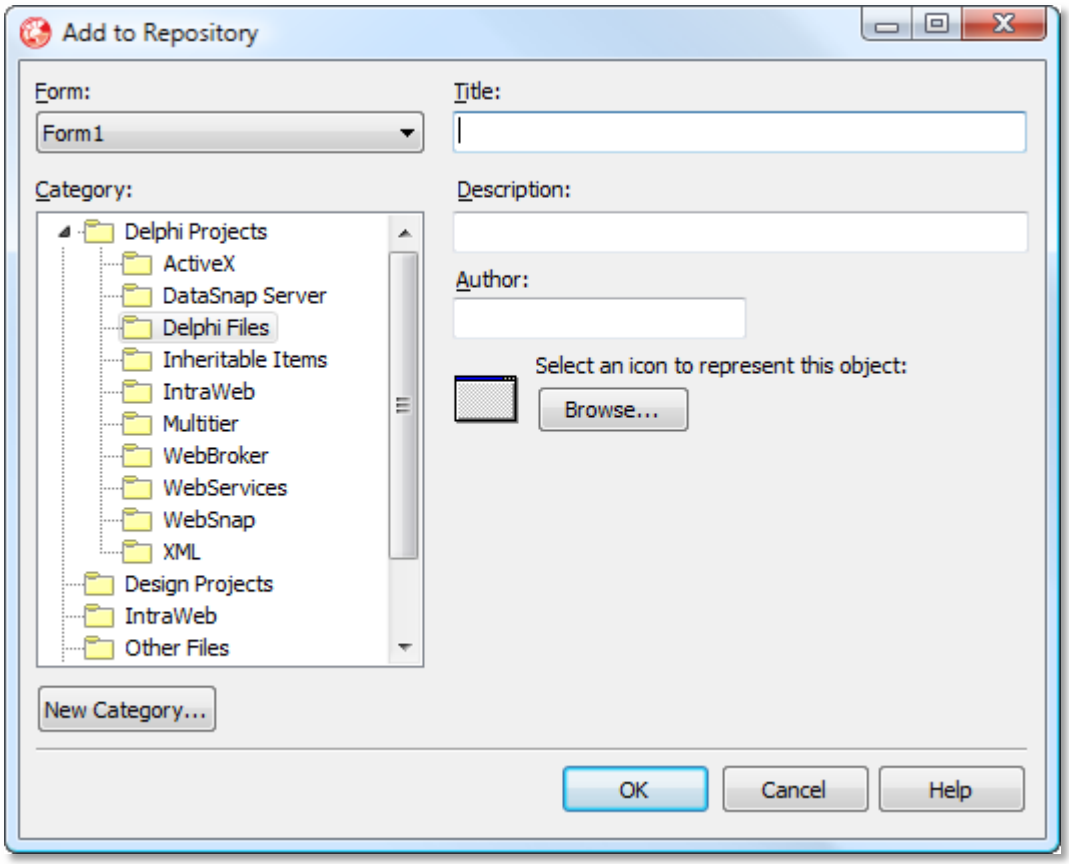


Рис. Ф1.18. Диалоговое окно *Add to Repository*

В *Репозитории* вы можете редактировать (**Edit...**) и удалять (**Delete**) объекты.

- **Configure Tools...** – открывает диалоговое окно **Tools Options**, в котором можно выбрать программы, используемые как инструментальные в среде программирования.

Window

- работа с окнами *Редактирования кода* (Рис. Ф1.19).

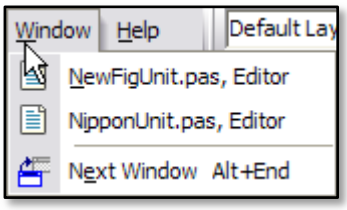


Рис. Ф1.19. Меню *Window*

Предоставляет возможность переключаться между окнами, когда исходный код модуля (или нескольких модулей) открыт в отдельном окне.

Help

– помощь (Рис. Ф1.20).

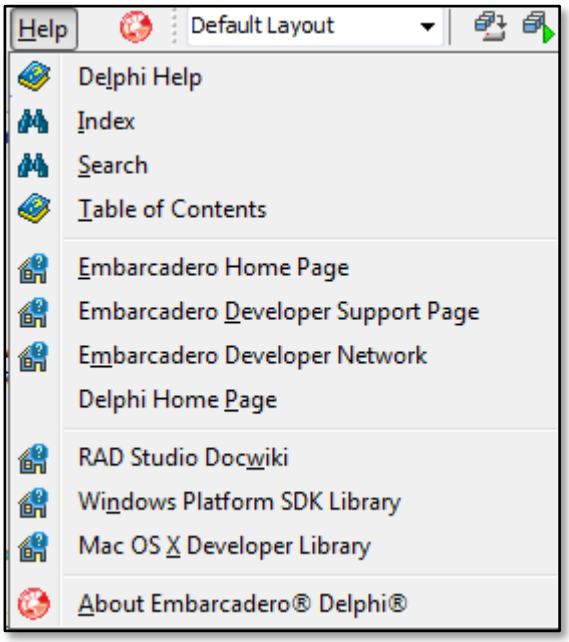


Рис. Ф1.20. Меню *Help*

- **Delphi Help** – открывает стандартное диалоговое окно справки. Пользуйтесь этой командой для получения необходимой информации по командам меню, процедурам, компонентам… Так как «помощь» чисто английская, то желательно знать хотя бы несколько слов из этого языка.

  Вы также легко сможете получить нужную информацию, если установите курсор на названии компонента или процедуры в *Редакторе кода* и нажмёте клавишу **F1**. Такой же результат вы получите, если выделите на форме компонент или свойство (событие) в *Инспекторе объектов* и опять же нажмёте **F1**. И вообще – не стесняйтесь попросить помощи у *Delphi* (**F1**!) при любой закавыке. Эта система доста-

точно «отзывчивая», в худшем случае вы получите **вежливый** отказ. Всё-таки приятно, что компьютеры не посылают вас подальше.

- **Embarcadero Developer Network** – открывает в *Рабочем окне* главную страницу сайта **Embarcadero** http:// edn.embarcadero.com/.

- **Delphi Home Page** – открывает в *Рабочем окне* страницу сайта **Embarcadero**, посвящённую *Delphi*.

Панели инструментов (Toolbars)

Находятся под *Главным меню* в левой части *Главного окна* (панель **Desktop** первоначально располагается справа от меню) и содержат кнопки, дублирующие наиболее часто используемые команды *Главного меню*.

Standard

– стандартная (Рис. Ф1.21).



Рис. Ф1.21. Панель инструментов *Standard*

-  - команда **New Items**. Очень часто применяется для присоединения новой формы или модуля к проекту.

-  - аналог команды **File > Reopen**. Позволяет быстро открывать сохранённые на диске проекты. Нажмите стрелочку справа от папки, чтобы раскрыть список последних проектов и файлов.

-  - аналог команды **File > Save** (быстрые клавиши - **Ctrl+S**).

-  - аналог команды **File > Save all** (быстрые клавиши – **Shift + Ctrl+S**).

-  - аналог команды **File > Open Project** (**Ctrl+F11**). Быстрое открытие сохранённого на диске проекта.

-  - аналог команды **Project > Add to Project** (**Shift+F11**).

-  - аналог команды **Project > Remove from Project**.

View

- просмотр (Рис. Ф1.22).



Рис. Ф1.22. Панель инструментов *View*

-  - аналог команды **View > Units...** (**Ctrl+F12**). Выводит на экран список всех модулей приложения, в котором можно выбрать нужный для отображения в окне *Редактора кодов*.

-  - аналог команды **View > Forms...** (**Shift+F12**). Выводит на экран список всех форм приложения. Выбранная форма появляется в окне *Конструктора формы*.

-  - аналог команды **View > Toggle Form/Unit** (**F12**). Переключается между окном *Редактора кода* и окном *Конструктора формы* (если в проекте несколько форм, то сначала выберите нужную предыдущей командой).

Debug

– отладка (Рис. Ф1.23).



Рис. Ф1.23. Панель инструментов *Debug*

-  - аналог команды **Run > Run Whitout Debugging** (**Ctrl+ Shift+F9**). Используется очень часто для запуска текущего проекта.

-  - аналог команды **Run > Run** (**F9**). Используется очень часто для запуска текущего проекта.

-  - аналог команды **Run > Program Pause**. Приостанавливает запущенную программу для отладки.

-  - аналог команды **Run > Program Reset (Ctrl+F2)**. Останавливает запущенную программу.

- - аналог команды **Run > Trace Into** (**F7**).
-  - аналог команды **Run > Step Over** (**F8**).
-  - аналог команды **Run > Run Until Return (Shift+F8)**.

Custom

- пользовательская.

-  - аналог команды **Help > Delphi Help**. Вход в **Справочную систему** *Delphi*.

Delphi

 - открывает диалоговое окно **About Embarcadero Delphi**.

Desktop

– рабочий стол (Рис. Ф1.24).



Рис. Ф1.24. Панель инструментов *Desktop*

- **Pick list** – позволяет переходить на ранее сохранённую или стандартную раскладку среды программирования.
- - сохраняет текущие настройки.
- - позволяет установить настройки для отладки.

Настройки можно сохранить и в *Главном меню* (**View > Desktops > Save Desktop**). Там же можно удалить настройки (**View > Desktops > Delete**) и выбрать настройки для отладки (**View > Desktops > Set Debug Desktop...**).

Вы можете легко убрать из *Главного окна* любую *Панель инструментов* и так же легко вернуть её на место с помощью контекстного меню, которое «всплывает» при нажатии правой кнопки мыши, когда она находится в об-

ласти *Главного меню* или *Панелей инструментов* (или выполнив команду *Главного меню* **View > Toolbars > Customize...**). Отображаются только те панели, против которых в контекстном меню стоит галочка (Рис. Ф1.25).

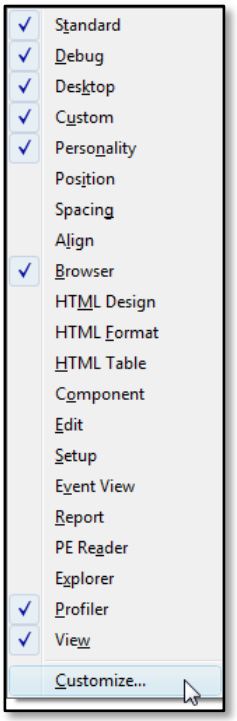


Рис. Ф1.25. Контекстное меню

Последняя команда контекстного меню, **Customize...** (многоточие во всех меню означает, что команда вызывает диалог или список возможных значений), позволяет настроить *Панели инструментов* и *Главное меню*. Выбрав её, вы увидите диалоговое окно с тремя страницами (Рис. Ф1.26).

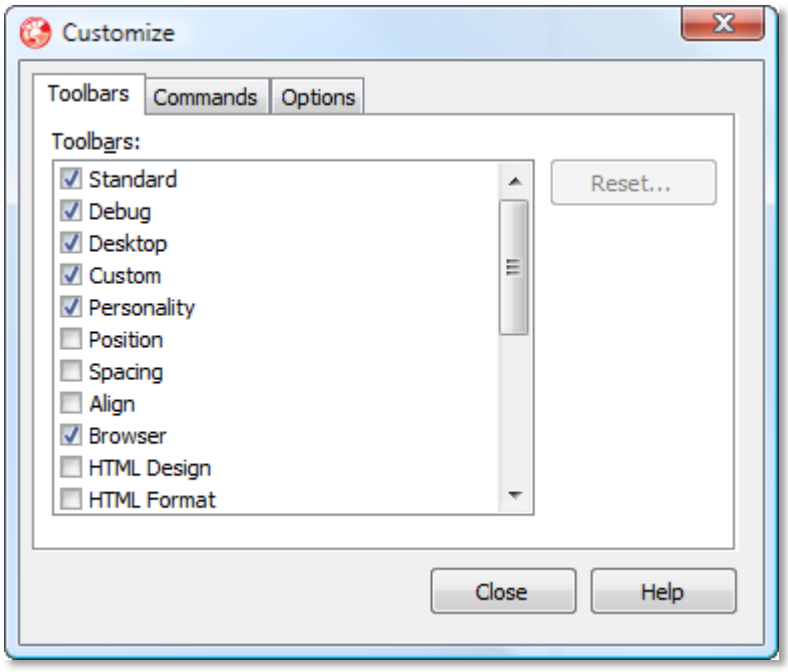


Рис. Ф1. 26. Диалоговое окно *Customize*

Первая страница, **Toolbars**, повторяет команды контекстного меню, управляющие видимостью *Панелей инструментов*. Кроме того, выбрав мышкой одну или несколько панелей (для этого удерживайте клавишу **Ctrl** или **Shift** и щёлкайте мышкой), вы увидите, что кнопка **Reset...** стала доступной. Вы можете нажать её, чтобы восстановить настройки по умолчанию для выделенных *Панелей инструментов* (если вы убрали кнопки из «джентльменского набора», то они снова появятся на панели, а все добавленные вами исчезнут). Эта радикальная чистка панелей может потребоваться, если вы нечаянно порезвитесь в настройках.

Страница **Commands** позволяет добавлять кнопки на любую панель инструментов и удалять их оттуда (Рис. Ф1.27).

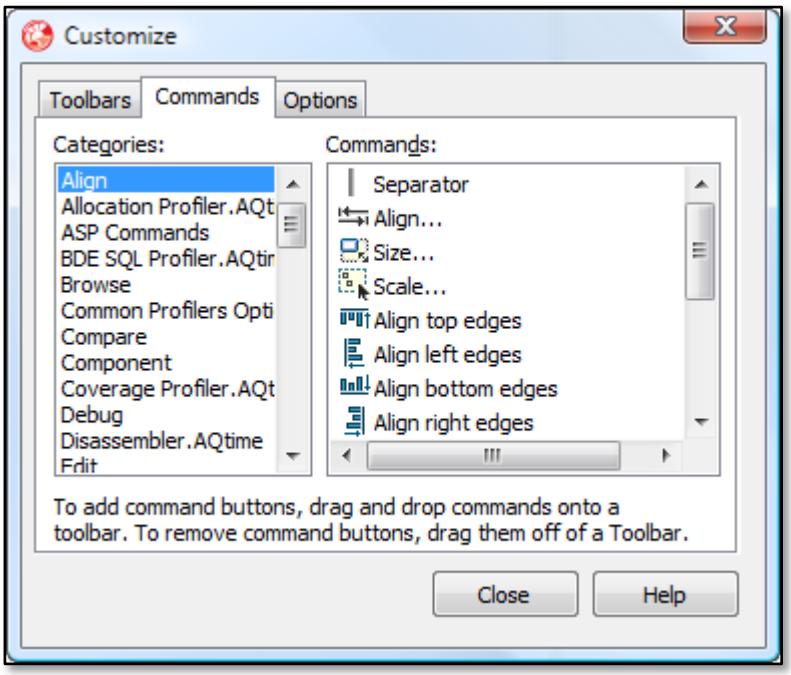


Рис. Ф1.27. Страница *Commands*

В списке **Categories** находятся названия пунктов *Главного меню*, а в списке **Command** – все команды из заданного меню. Чтобы поставить кнопку (она будет выполнять те же действия, что и соответствующая команда *Главного меню*) на панель, возьмите её мышкой и перетащите на нужное место в выбранной панели инструментов, где и «бросьте» (модная нынче технология **drag and drop**). Убрать кнопку с панели и того проще: подхватите её мышкой и отбуксируйте за пределы *Главного окна*. Таким нехитрым способом вы можете скомплектовать панели по своему вкусу. Ну а если изрядно напортачите, возвращайтесь к первоисточнику!

На последней странице – **Options** - вы можете изменить вид подсказок (так называемые хинты, *hints*), которые появляются, когда мышка тормозит на какой-нибудь из кнопок панелей инструментов.

Палитра компонентов (Tool Palette)

Палитра компонентов была подробно разобрана на Семинаре 1.

Окно Менеджера проекта (Project Manager)

Расположено в правом верхнем углу ИСР и показывает все файлы текущего проекта (Рис. Ф1.28).

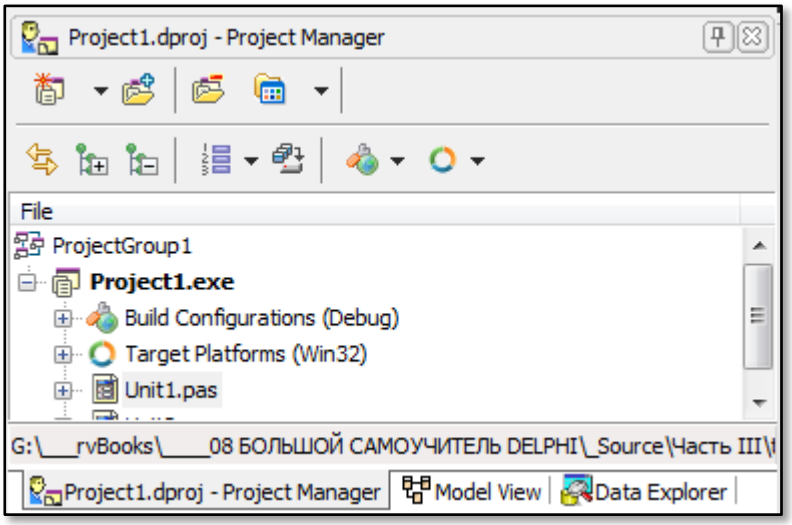


Рис. Ф1.28. Менеджер проектов

Часть информации находится в *свёрнутом* виде (Рис. Ф1.29).



Рис. Ф1.29. Свёрнутый список

Чтобы *развернуть* нужную часть списка, щёлкните на значке ⊞ (Рис. Ф1.30).

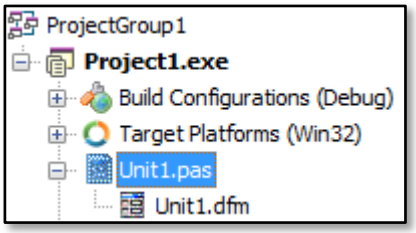


Рис. Ф1.30. Развёрнутый список

Менеджер проекта также облегчает работу с файлами проекта. Например, после нажатия правой кнопки мыши на строчке *Unit1.pas* (главный модуль проекта), на экране появится контекстное меню (Рис. Ф1.31).

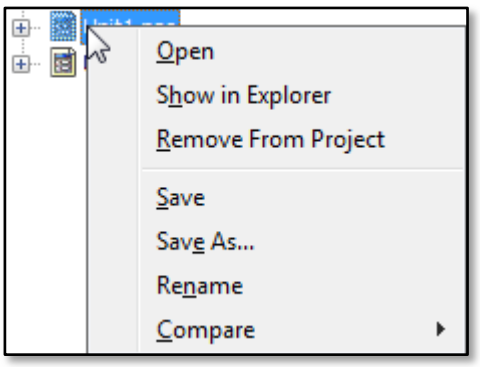


Рис. Ф1.31. Контекстное меню

В нём вы можете сохранить файл модуля на диске (команды **Save** или **Save As...**), переименовать его (**Rename**) и удалить из проекта (**Remove From Project**). Команда *Открыть* (**Open**) действует аналогично клавише **F12**, то есть открывает окно *Конструктора формы* (если курсор находится в строке *Unit1.dfm*) или окно *Редактора кода* (если в строке *Unit1.pas*).

Ещё больше возможностей предоставляет контекстное меню для строчки с названием выполняемого файла (в данном примере *Project1.exe*) (Рис. Ф1.32).

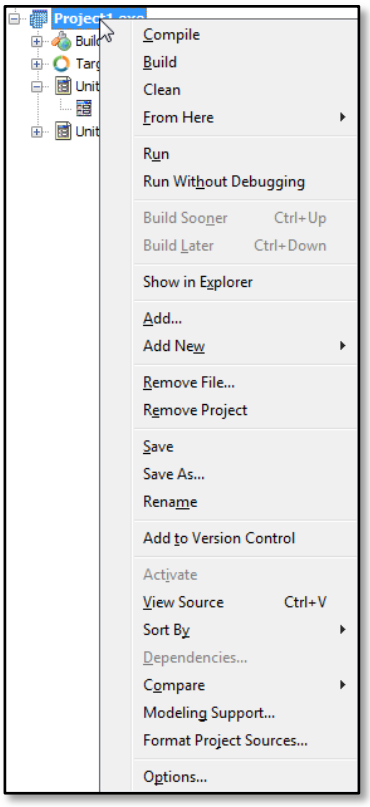


Рис. Ф1.32. Контекстное меню выполняемого файла

Из него вы можете добавлять новые элементы к проекту (**Add...** и **Add New**), удалять их (**Remove File...**) и переименовать проект (**Rename**), сохранять его на диске, а также создавать исполняемый файл проекта (**Compile** и **Build**).

Таким образом, если ваш проект содержит большое количество составных частей (элементов), **Менеджер проектов** обеспечит вам лёгкий доступ к каждой из них.

Окно Структуры проекта (Structure View)

Находится слева вверху. Оно отображает иерархическую структуру компонентов на форме (когда открыто окно *Конструктора формы*) или структуру кода программы (когда открыто окно *Редактора кода*) (Рис. Ф1.33).

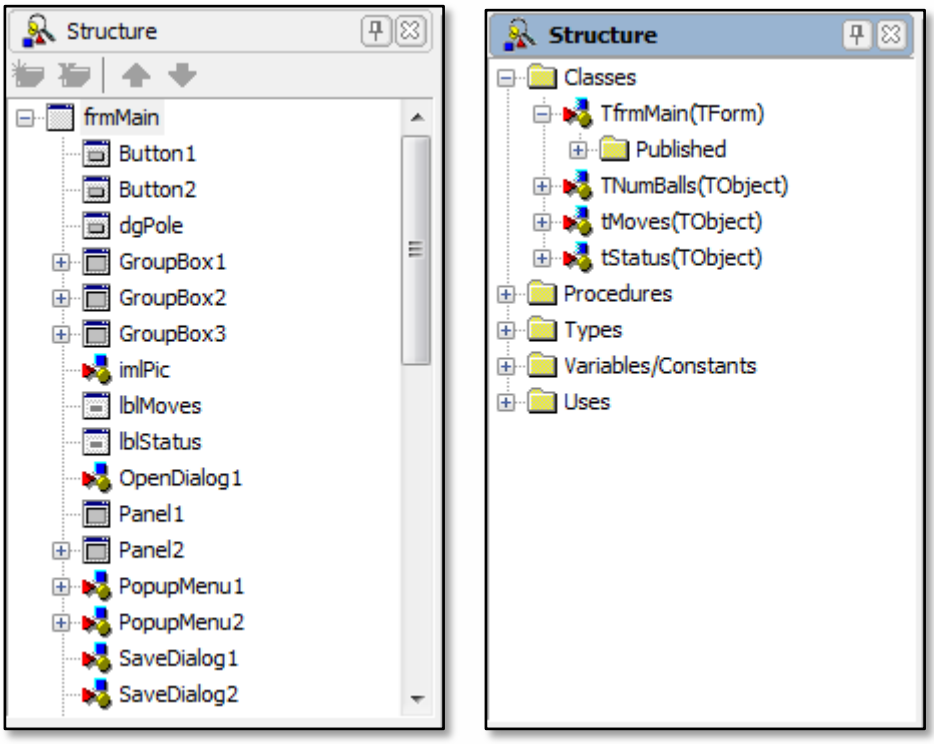


Рис. Ф1.33. Окно *Структуры проекта*

В окне *Структуры проекта* находится древовидный список (почему раньше этот режим и назывался *Object TreeView*) всех объектов модуля, выбранного в *Рабочем окне*. Здесь вы сможете легко найти все константы, переменные, типы, процедуры и функции, объявленные в модуле, и быстро перейти к ним.

Каждый вид объектов имеет в окне *Структуры проекта* собственный значок (Рис. Ф1.34).

-  - классы;
-  - используемые модули;
-  - константы и переменные;
-  - методы и процедуры;
-  - методы и функции;
-  - свойства;
-  - типы.

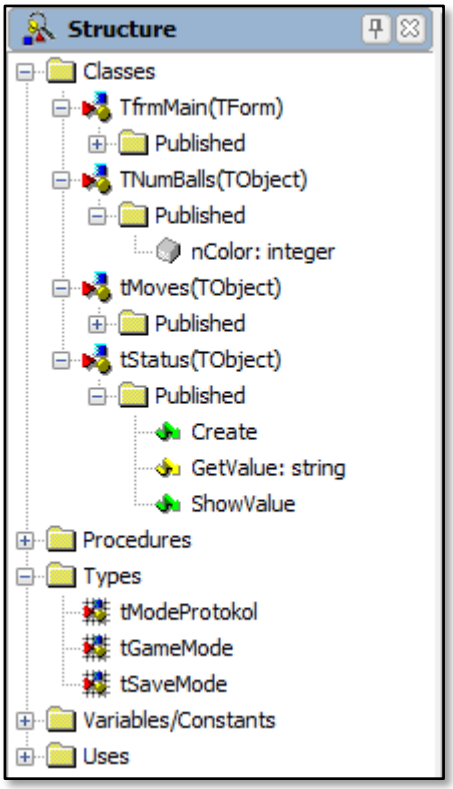


Рис. Ф1.34. Структурные значки

Если проект содержит большое количество компонентов и/или очень длинный исходный код, то из окна *Структуры проекта* вы сможете быстро переместиться к нужному компоненту или фрагменту исходного кода. Например, в режиме конструирования формы **одиночный щелчок** на названии компонента выделяет его на форме (Рис. Ф1.35).



Рис. Ф1.35. Одним щелчком мыши можно выделить кнопку на форме

Обратите внимание, что в окне *Инспектора объектов* этот компонент становится активным, и вы можете изменять его свойства.

Двойной щелчок автоматически открывает окно *Редактора кода* с обработчиком события компонента по умолчанию (Рис. Ф1.36).

```
  //ВЫХОД ИЗ ПРОГРАММЫ
procedure TfrmMain.sbtExitClick(Sender: TObject);
begin
759   Game_Exit;
760 end;  // sbtExitClick
```

Рис. Ф1.36. Код кнопки

В режиме редактирования кода одним щелчком мыши на названии обработчика события компонента или функции вы сразу перейдёте к нужному участку исходного кода (Рис. Ф1.37).

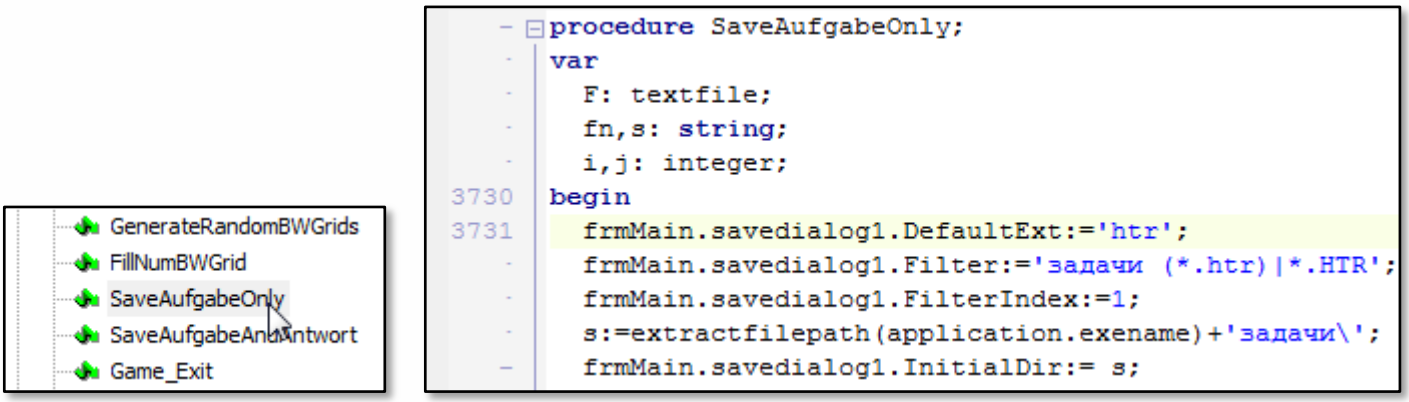


Рис. Ф1.37. Переход к коду

Кроме того, в окне *Структуры проекта* показываются все синтаксические и другие **ошибки** в исходном тексте программы. Например, исправим название переменной **s** на **s1** (Рис. Ф1.38).

```
  //записать :
    for j:= 0 to Pole.Height-1 do begin
3754  s1:='';
      for i:= 0 to Pole.Width-1 do begin
```

Рис. Ф1.38. Делаем ошибку

Сразу же в окне *Структуры проекта* появится новая папка **Errors** и описание ошибки с номером строки, в которой она находится (Рис. Ф1.39).

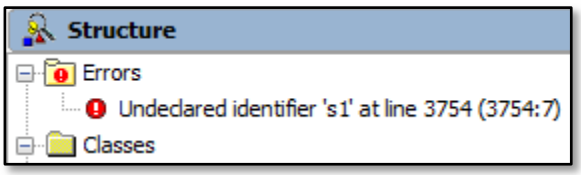


Рис. Ф1.39. Информация об ошибке

Достаточно дважды щёлкнуть по строке с описанием ошибки, и вы тут же окажетесь в нужном месте исходного кода, при этом ошибка также будет отмечена красной волнистой линией (вспомните школьные годы чудесные!).



Код программы проверяется на синтаксические ошибки уже при наборе нового текста.

Чтобы закрыть окно *Структуры проекта*, щёлкните на крестике в верхнем правом углу. А вернуть его на экран можно, выбрав команду **View > Structure** в *Главном меню*, либо нажав клавиши **Shift+Alt+F11**.

В контекстном меню окна *Структуры проекта* (Рис. Ф1.40)

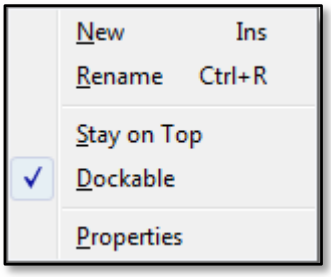


Рис. Ф1.40. Контекстное меню окна *Структуры проекта*

можно добавить (команда **New**) или переименовать (**Rename**) выбранный объект (лучше не злоупотреблять этой возможностью, так как в тексте программы останется и прежнее название), отделить окно **Структуры проекта** (сбросьте флажок у пункта **Dockable**), а также настроить окно *Структуры проекта* (**Properties**) в диалоговом окне **Explorer Options** (Рис. Ф1.41).

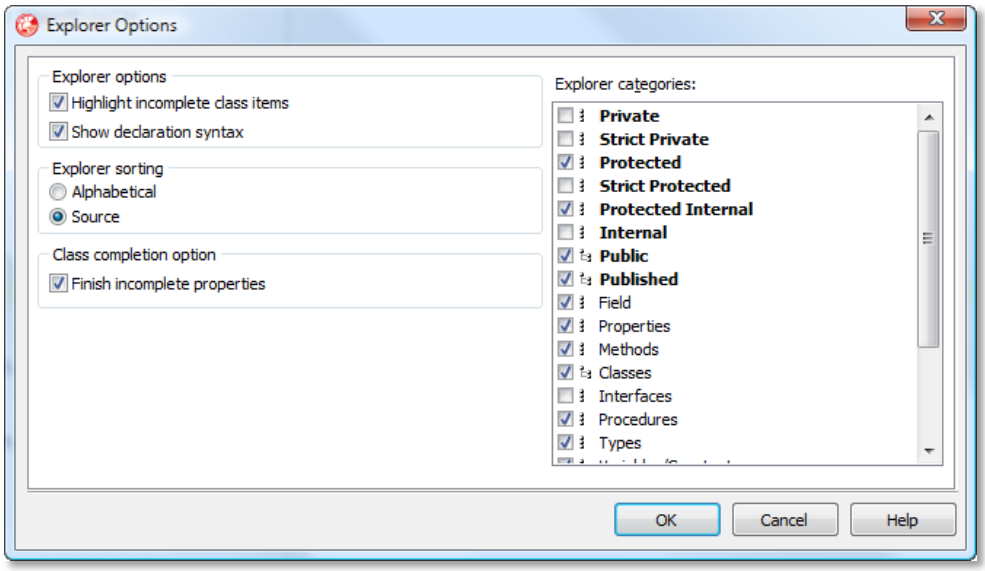


Рис. Ф1.41. Диалоговое окно *Explorer Options*

Если флажок **Show declaration syntax** сброшен, то в списке будут указаны только названия элементов кода (режим по умолчанию), если установлен, то выводится полная информация о них.

Переключатель **Explorer sorting** определяет порядок элементов в списке: **Alphabetical** – алфавитный, **Source** - как в программе.

Флажки в разделе **Explorer categories** позволяют выбрать те категории элементов кода, которые будут отображаться в окне *Структуры проекта*.

Рабочее окно

В более ранних версиях *Delphi* окна *Конструктора формы* и *Редактора кода* были выполнены в виде независимых панелей, частично или полностью перекрывавших друг друга. Теперь они оба находятся в *Рабочем окне* ИСР, которое представляет собой панель с вкладками. Первоначально вкладка только одна – **Welcome Page**, но после открытия нового проекта или после загрузки существующего количество вкладок увеличивается (Рис. Ф1.42).



Рис. Ф1.42. Страницы *Рабочего окна*

Чтобы открыть нужное окно, достаточно просто щёлкнуть по вкладке с его именем.

В режимах конструирования формы и редактирования кода под **Рабочим окном**, в строке состояния также появляются три вкладки (Рис. Ф1.43).



Рис. Ф1.43. Вкладки под *Рабочим окном*



Две вкладки – **Code** и **Design** – существуют уже давно, а вот вкладка **History** появилась только в **Delphi 2005**.

Окно Конструктора формы (Designer Surface, Form Designer)

Именно в этом окне выполняется **визуальное проектирование** формы вашего приложения. Здесь вы можете задать ей необходимые размеры, расставить на ней нужные для работы программы компоненты (как это сделать, мы уже говорили). Всю трудоёмкую и неприятную работу по пересчёту их размеров и координат берёт на себя *Конструктор формы*, который как настоящий друг находится в тени (в хорошем смысле этого слова) и ненавязчиво помогает вам избавиться от ненужной рутины в творческом процессе. Впрочем, о скромных героях быстро забывают, поэтому окно *Конструктора формы* часто называют *окном Формы*, а то и просто **формой.**

Форма - главный компонент всего приложения, но следует отметить, что форма в окне *Конструктора формы* и в работающей программе – вовсе не одно и то же. Например, все компоненты на форме в режиме проектирования видны, однако часть из них можно скрыть от посторонних глаз в работающей программе. Да и сама форма может измениться до неузнаваемости.

В **контекстном меню** *Конструктора формы* вы найдёте многие команды подменю **Edit** *Главного меню* (Рис. Ф1.44).

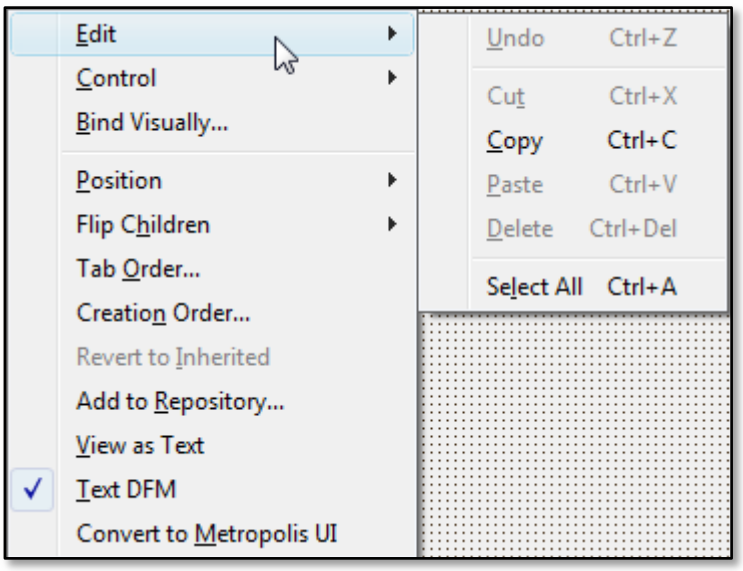


Рис. Ф1.44. Контекстное меню *Конструктора формы*

Дополнительно вы можете просмотреть атрибуты (свойства и события) формы и компонентов на ней в текстовом виде (команда **View As Text**; возврат к графическому отображению - командой **View As Form** контекстного меню ***Редактора кода*** или клавишами **Alt+F12**). Здесь можно

изменить любые значения свойств (но обычно это делают в *Инспекторе объектов*) или посмотреть только те свойства и события, которые были изменены вами при проектировании приложения, тогда как в *Инспекторе объектов* присутствует много «посторонней» информации. Можно также перенести текстовую информацию в программу или в её распечатку.
Если флажок **Text DFM** установлен (по умолчанию), то форма будет **з**аписана на диск в *текстовом* формате (как при выполнении команды **View As Text**). Если сброшен, то в двоичном. Обычно используют первый формат, а второй оставлен только для совместимости с прежними версиями *Delphi*.

Окно Редактора кода (Code Editor)

В окне **Редактора кода** вы можете создавать и редактировать код программы (Рис. Ф1.45).

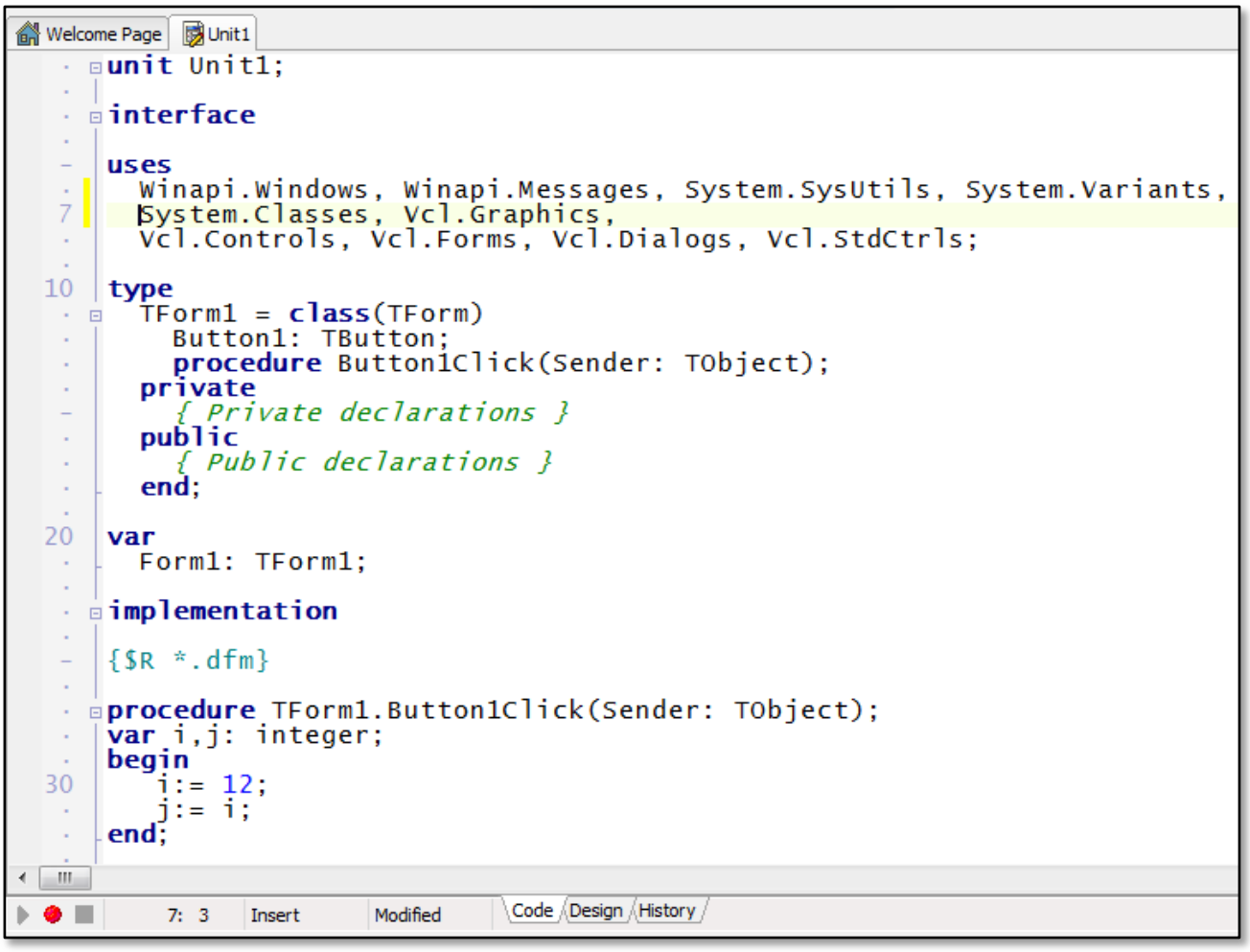


Рис. Ф1.45. Редактор кода

Редактор кода представляет собой обычный текстовый редактор. Сначала в нём открыта для редактирования только одна страница – с исходным

текстом (заготовкой) модуля формы **Form1** разрабатываемого приложения. Его название **Unit1.pas** находится в заголовке окна и на вкладке *Рабочего окна*. В редактор можно загрузить и другие подходящие файлы с диска. Под каждый из них в редакторе отводится отдельная страница, названия файлов написаны сверху, на вкладках (ярлычках, табуляторах). Переход на нужную страницу осуществляется щелчком на соответствующем ярлычке.

На самом деле *Редактор кода* не обычный, а весьма «умный» текстовый редактор! Например, он подсвечивает (выделяет цветом) синтаксические конструкции языка *Delphi*, отыскивает ошибки, самостоятельно формирует часть исходного текста модуля.

В отличие от более ранних версий, в *Delphi XE3* имеется **нумерация строк** (Рис. Ф1.46).

```
20  var
      Form1: TForm1;

    implementation

    {$R *.dfm}

    procedure TForm1.Button1Click(Sender: TObject);
    var i,j: integer;
    begin
30      i:= 12;
        j:= i;
    end;

    end.
```

Рис. Ф1.46. Строки нумеруются слева

Она включена по умолчанию. Вы можете отменить нумерацию строк или, наоборот, заставить редактор нумеровать все строки исходного текста. Сделать это можно в меню **Tools > Options > Editor Options** (Рис. Ф1.47).

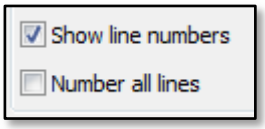


Рис. Ф1.47. Управляем нумерацией строк

Кроме того, номер строки с курсором и его позиция в строке выводятся в строке состояния: 

.

Если код модуля очень длинный, то смысловые участки кода (блоки) можно сворачивать и разворачивать, нажимая на значки ⊟ и ⊞ слева от кода, соответственно (Рис. Ф1.48-51).

```
  //Приём 7
  //Если в двух приугловых клетках находятся одинаковые числа,
2020 //то третья приугловая клетка должна быть белого цвета.
  //.....
  //XX...
  //.....
2024 function Algorithm_Static7(var x: integer; var y: integer): boolean; ...
2050
```

Рис. Ф1.48. Код процедуры свёрнут

```
  //Приём 7
  //Если в двух приугловых клетках находятся одинаковые числа,
2020 //то третья приугловая клетка должна быть белого цвета.
  //.....
  //XX...
  //.....
2024 function Algorithm_Static7(var x: integer; var y: integer): boolean;
    //var i, j, n: integer;

    function test(x1,y1,x2,y2,x3,y3,n: integer): boolean;
    begin
      Result := TRUE;
2030  if Pole.GetNum(x1,y1) = Pole.GetNum(x2,y2) then //- пара одинаковых чисел
        Result := Pole.SetColorEx(x3,y3, ccWhite, 'Static7.' + inttostr(n));
    end;

  begin
    Result:= FALSE;
    If not test(0,1,1,1,1,0,1) then exit; //- левый верхний угол h
    If not test(1,0,1,1,0,1,2) then exit; //- левый верхний угол v

    If not test(Pole.Width-2,1, Pole.Width-1,1, Pole.Width-2,0,3) then exit; //
2040 If not test(Pole.Width-2,0, Pole.Width-2,1, Pole.Width-1,1,4) then exit; //

    If not test(0,Pole.Height-2, 1,Pole.Height-2, 1,Pole.Height-1,5) then exit;
    If not test(1,Pole.Height-2, 1,Pole.Height-1, 0,Pole.Height-2,6) then exit;

    If not test(Pole.Width-2,Pole.Height-2, Pole.Width-1,Pole.Height-2, Pole.Wi
    If not test(Pole.Width-2,Pole.Height-2, Pole.Width-2,Pole.Height-1, Pole.Wi
    Result:= TRUE;
  end;

2050
```

Рис. Ф1.49. Код процедуры развёрнут

Аналогично действуют команды контекстного меню **Fold/Unfold**.

Вы можете также сворачивать произвольные участки кода, выделив их парной директивой **{$REGION} … {$ENDREGION}** .

```
{$REGION}
  flgChange:= TRUE;
  Pole.SetColor(x,y,clr);
  inc(Moves.Count);
2060
  Moves.ShowCount;
  case Clr of
    ccWhite: begin s:= ' White' end;
    ccBlack: begin  s:= ' Black' end;
    ccGray: begin  s:= ' Gray' end;
  end;
{$ENDREGION}
```

Рис. Ф1.50. Развёрнутая область кода

```
begin
2056 ...
  //frmProtokol
```

Рис. Ф1.51. Свёрнутая область кода

Так можно скрывать уже отлаженные участки исходного кода программы, экономя место на экране и получая быстрый доступ к остальному коду.

При наборе имени класса или объекта достаточно поставить точку, как на экране появляется список их доступных методов и свойств, из которого можно выбрать нужный. Кроме того, для процедур и функций указаны все параметры (Рис. Ф1.52).

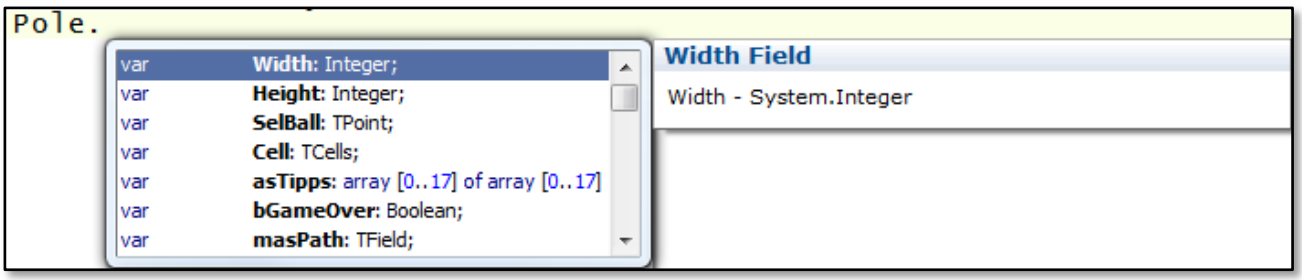


Рис. Ф1.52. Интеллектуальная подсказка

Если раньше это окно имело постоянные размеры из-за чего длинные строки в нём не помещались, то теперь вы можете самостоятельно изменять его и вширь и ввысь.

В *Delphi* можно очень легко **закомментировать** выделенный блок текста. Для этого нужно нажать комбинацию клавиш **Ctrl + #** (Рис. Ф1.53).

```
//  case masRowsNum[ACol, ARow].StatusGroup of
//    stYellow: masRowsNum[ACol, ARow].StatusGroup:= stWhit
//    stWhite: masRowsNum[ACol, ARow].StatusGroup:= stYello
//  end;
```

Рис. Ф1.53. Блок кода закомментирован

Применение этой команды к закомментированному выделенному блоку текста удаляет символы комментария (Рис. Ф1.54).

```
case masRowsNum[ACol, ARow].StatusGroup of
  stYellow: masRowsNum[ACol, ARow].StatusGroup:= stWhite;
  stWhite: masRowsNum[ACol, ARow].StatusGroup:= stYellow;
end;
```

Рис. Ф1.54. Раскомментированный блок текста

Аналогично действует команда контекстного меню **Toggle Comment.**

Контекстное меню *Редактора кода* появляется после щелчка правой кнопкой мыши в области окна (Рис. Ф1.55).

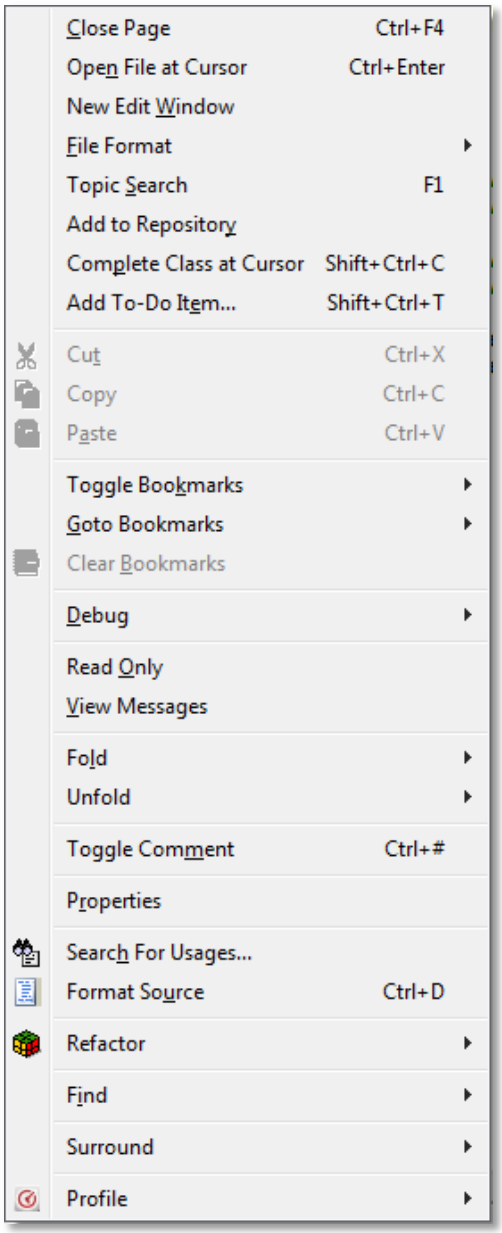


Рис. Ф1.55. Контекстное меню *Редактора кода*

В нём доступны следующие **команды**:

- **Close Page** (**Ctrl+F4**) – закрывает текущую страницу в *Редакторе кода*. Если страница была изменена, будет выдан запрос на сохранение текста или проекта.
- **Open File At Cursor** (**Ctrl+Enter**) – вставляет текстовый файл в позицию курсора.

- **New Edit Window** – дублирует команду *Главного меню* **View > New Edit Window**.

- **File Format** – позволяет задать кодировку файла при сохранении на диске.

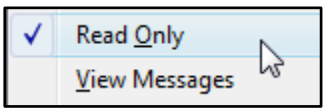
- **Topic Search** (**F1**) - дублирует команду *Главного меню* **Help > Delphi Help**.

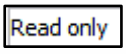
- **Cut, Copy, Paste** – дублируют команды *Редактора кода*.

- **Toggle Bookmarks** – устанавливает закладку.

- **Goto Bookmarks** – осуществляет переход к закладке (для быстрого перехода нажмите **Ctrl+номер закладки**).

- **Debug** – команды отладки.

- **Read Only** – при установленном флажке 

текст в окне *Редактора кода* нельзя изменить (в строке состояния появится соответствующая надпись: 

).

- **View Messages** – показывает окно сообщений 

под окном *Редактора кода*.

- **Fold** – *сворачивает* указанный блок кода (Рис. Ф1.56).

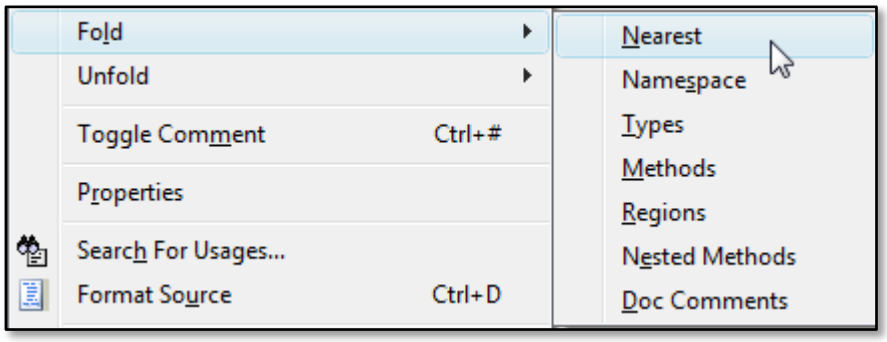


Рис. Ф1.56. Сворачиваем блок кода

- **Unfold** – *разворачивает* указанный блок кода (Рис. Ф1.57).

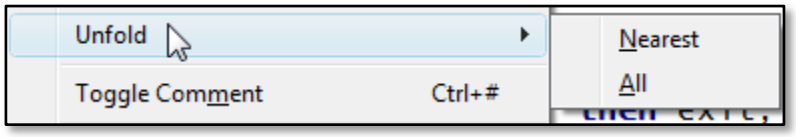


Рис. Ф1.57. Разворачиваем блок кода

- **Toggle Comment** – закомментировывает/раскомментировывает выделенный блок текста.

- **Properties** – дублирует команду *Главного меню* **Tools > Options**.

- **Surround** – вставляет заготовку кода (Рис. Ф1.58).

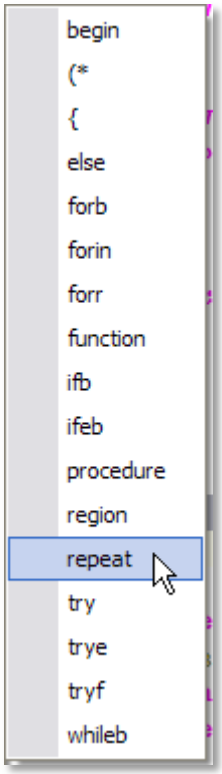


Рис. Ф1.58. Заготовки кода

Например, после выполнения команды **repeat** в исходный текст будут вставлены строки (Рис. Ф1.59).

```
repeat


until (True);
```

Рис. Ф1.59. Вставили заготовку

- **Refactor** – переименовывает переменную или процедуру.
- **Find Declaration** – переходит к объявлению переменной или процедуры.

Окно Инспектора объектов (Object Inspector)

Окно **Инспектора объектов** находится под панелью *Структуры проекта* и имеет две страницы (вкладки) - **Properties page** и **Events page**. Каждая из них разделена на две части вертикальной линией, которую можно передвигать мышкой. **Чтобы открыть нужную страницу**, нужно, как обычно, щёлкнуть на её ярлычке (Рис. Ф1.60).

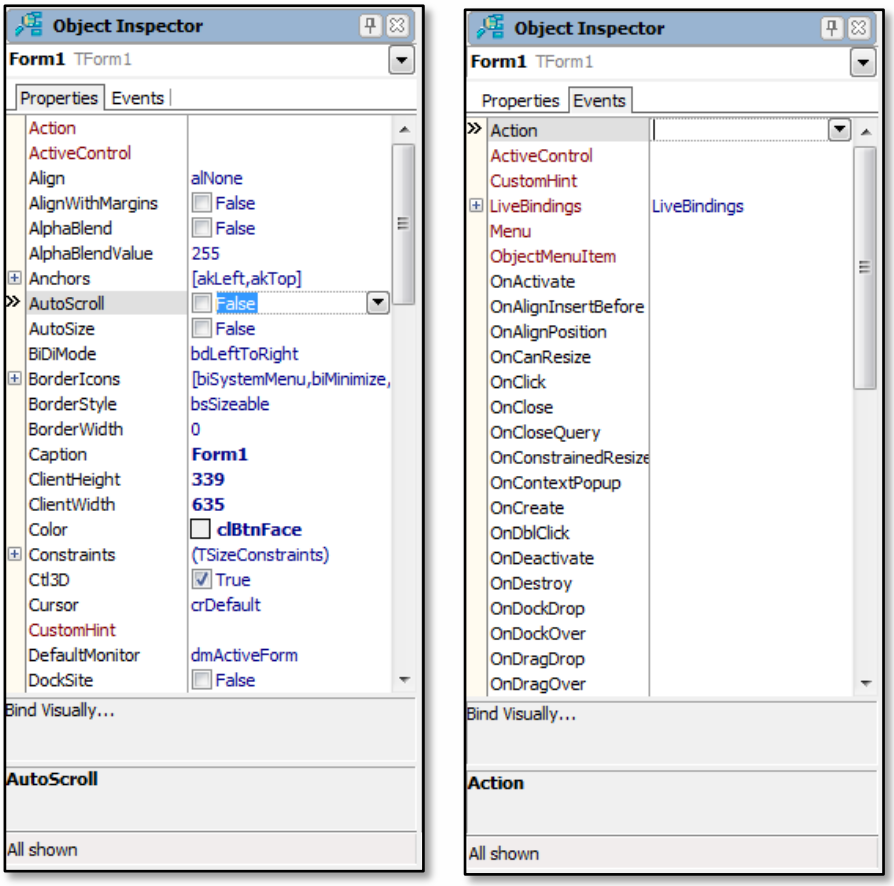


Рис. Ф1.60. Две страницы *Инспектора объектов*

На странице **Properties page** можно задавать **свойства** компонентов, установленных на форме (а также и самой формы), доступные во время разработки проекта (некоторые свойства можно изменить только в работающей программе). В начале работы над проектом в окне *Инспектора объектов* виден только список свойств и их значения для **формы**. Если вы добавите на форму какой-либо **компонент**, то перечень его свойств появится, если выделить его (щёлкните на нём мышкой) или выбрать название этого компонента в открывающемся списке, находящемся под заголовком окна.

На странице **Events page** вы найдёте все **события**, на которые реагирует текущий компонент. Например, все визуальные компоненты «откликаются» на щелчок мыши на нём. Названия событий начинаются с префикса **On** (**OnClick** – для щелчка). Как именно программа будет обрабатывать наступление того или иного события, зависит от вас. Для этого нужно задать процедуру-обработчик события. Чтобы получить заготовку такой процедуры, дважды щёлкните в пустом текстовом поле напротив названия события (Рис. Ф1.61).



Рис. Ф1.61. Создаём заготовку для метода

Например, заготовка для процедуры, обрабатывающей щелчок на форме, выглядит так:

```
procedure TForm1.Button1Click(Sender: TObject);
var i,j: integer;
begin

end;
```

Естественно, весь остальной код – это уже ваша забота.

Все компоненты имеют **«главное» событие** (или свойство) **по умолчанию**, поэтому достаточно дважды щёлкнуть **на самом объекте**, чтобы раскрылась заготовка процедуры для обработчика события (или диалоговое окно для ввода значения свойства).

Вы можете также выбрать уже имеющуюся процедуру-обработчик для текущего или другого компонента из выпадающего списка (узнаёте кнопку со стрелочкой в конце окна ввода названия процедуры?), тогда указанное событие будет связано с другим событием и будет обрабатываться его процедурой.



Кстати говоря, вы можете «не согласиться» со стандартным названием процедуры и ввести любое другое (только не запутайтесь в псевдонимах!).

Если для какого-либо события активного компонента существует процедура, то легко перейти на неё в окне *Редактора кода*, если дважды щёлкнуть по её названию на странице свойств.
Последовательность действий такая. Перейдите в окно *Конструктора формы*, выберите компонент, откройте в окне *Инспектора объектов* страницу событий и щёлкните на названии процедуры. На первый взгляд эти манипуляции довольно сложные, но, немного потренировавшись, вы убедитесь в их эффективности.

Как и другие окна, *Инспектор объектов* имеет своё **контекстное меню** (Рис. Ф1.62).

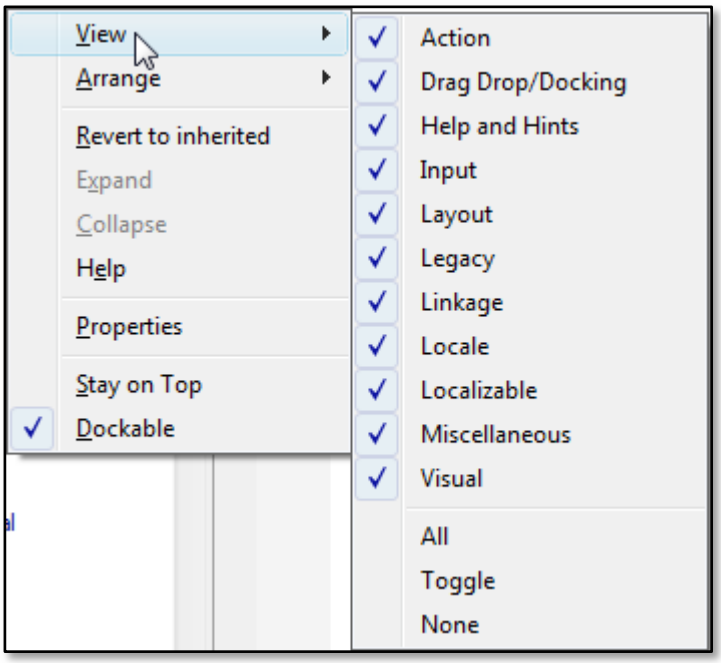


Рис. Ф1.62. Контекстное меню *Инспектора объектов*

В нём можно задать, какие категории свойств и событий будут отображаться в окне *Инспектора объектов* и в каком порядке. Иногда бывает полезно убрать те из них, которыми вы не пользуетесь в своём проекте.

Каждый компонент имеет свой, достаточно обширный набор свойств и событий, они подробно описаны при рассмотрении самих компонентов (в этой книге «повезло» только форме и сетке **TDrawGrid**, но многие свойства и события у большинства компонентов совпадают, так что вполне можно разобраться и с остальными).

По умолчанию свойства и события компонентов сгруппированы по категориям, что не всегда удобно (Рис. Ф1.63).

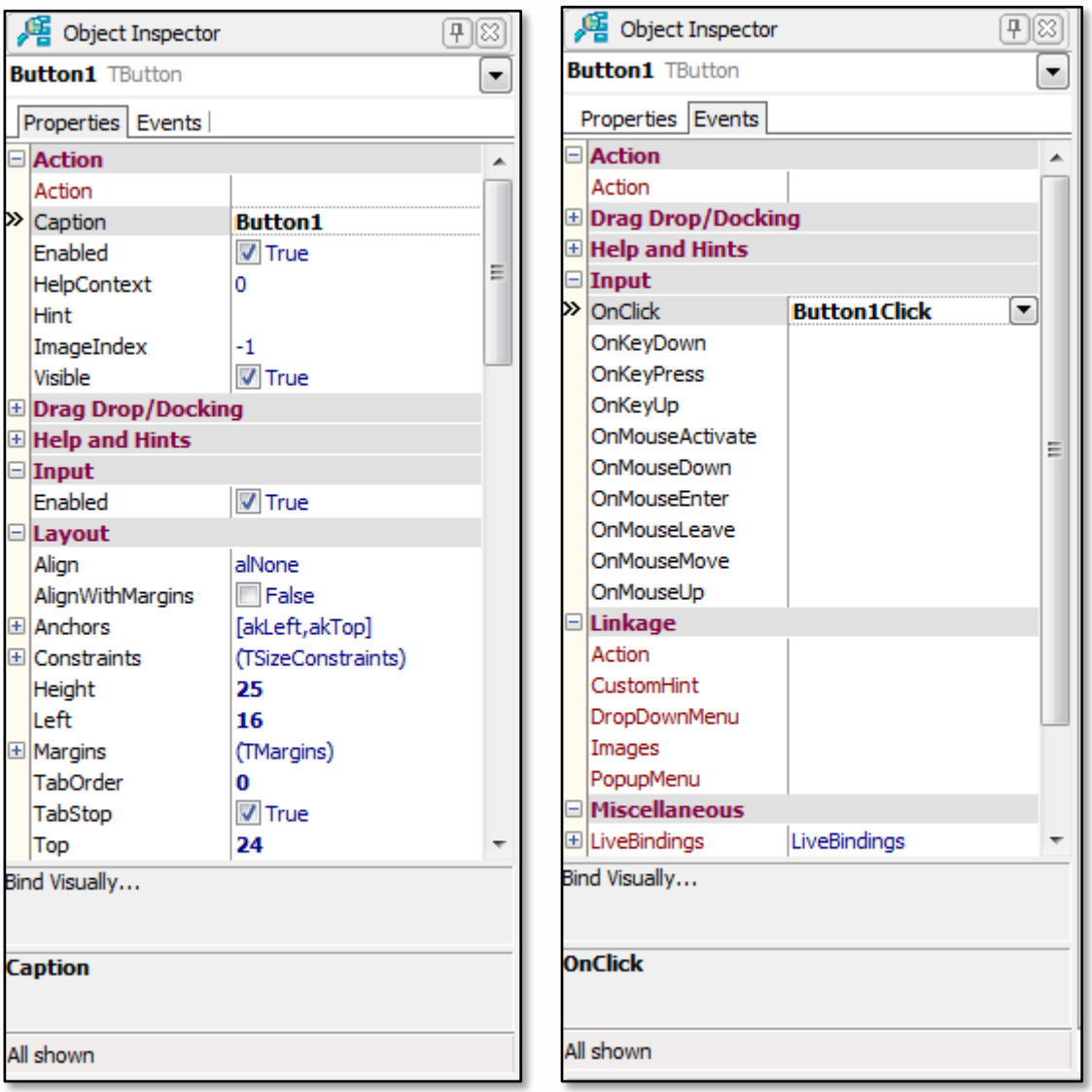


Рис. Ф1.63. *Категории* свойств и событий объекта

В прежних версиях *Delphi* свойства и события располагались на страницах свойств и событий **в алфавитном порядке**. Чтобы вернуться к более естественному представлению названий свойств и событий, нужно в контекстном меню окна *Инспектора объектов* выбрать команду **Arrange > by Name** (Рис. Ф1.64).

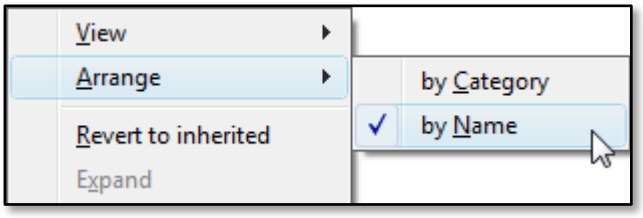


Рис. Ф1.64. Выбираем *алфавитный* порядок

Горячие клавиши ИСР

Во многих случаях удобнее пользоваться *горячими клавишами* (*клавиатурными сокращениями*), чем кнопками или пунктами меню. В следующей таблице вы найдёте наиболее употребительные **горячие клавиши**.

Вызов меню		
File	**Alt+F**	Открытие и сохранение файлов.
Edit	**Alt+E**	Редактирование файлов.
Search	**Alt+S**	Поиск в файлах.
View	**Alt+V**	Панели.
Refactor	**Alt+O**	Переименование объектов.
Project	**Alt+P**	Работа с файлами проекта.
Run	**Alt+R**	Запуск и отладка приложения.
Component	**Alt+C**	Работа с компонентами.
Tools	**Alt+T**	Настройки.
AQtime	**Alt+A**	Профилирование.
Window	**Alt+W**	Работа с окнами приложения.
Help	**Alt+H**	Справочник.
Команды меню File		
Ctrl+F11	**Open Project...**	**Открывает** сохранённый на диске проект.
Ctrl+S	**Save**	**Записывает** изменения в текущем модуле.
Shift+Ctrl+S	**Save All**	**Записывает** все изменения в текущем проекте.
Alt+F11	**Use Unit...**	Открывает диалоговое окно *Use Unit*, в котором можно выбрать из списка модуль для секции *uses* текущего модуля.
Команды меню Edit		
Ctrl+Z	**Undo**	**Отменяет** последнюю команду.
Shift+Ctrl+Z	**Redo**	**Возвращает** последнюю отменённую команду.
Ctrl+D	**Format Source**	**Форматирует** исходный код.
Ctrl+X	**Cut**	**Вырезает** выделенный блок кода или компонент(ы) (они удаляются, но помещаются в буфер обмена).
Ctrl+C	**Copy**	**Копирует** выделенный блок кода или компоненты (они помещаются в буфер обмена).

Ctrl+V	**Paste**	**Вставляет** удалённый или скопированный блок кода или компонент (из буфера обмена).
Ctrl+Del	**Delete**	**Удаляет** выделенный блок кода или компонент(ы).
Ctrl+A	**Select All**	**Выделяет** весь текст или все компоненты на форму.
Команды меню Search		
Ctrl+F	**Find…**	**Ищет** заданные слова в текущем файле.
Shift+Ctrl+F	**Find in Files…**	**Ищет** заданные слова в файлах на диске.
Ctrl+R	**Replace…**	**Заменяет** один текст другим.
F3	**Search Again**	**Продолжает** поиск.
Ctrl+E	**Incremental Search**	**Ищет** заданные слова в текущем файле. *При вводе* заданного слова в тексте выделяются все подходящие слова.
Shift+Ctrl+Enter	**Find References**	Составляет список всех **ссылок** в файлах проекта на заданный объект.
Ctrl+G	**Find Original Symbol**	Ищет заданный объект в файлах *Менеджера проекта*.
Ctrl+.	**IDE Insight…**	Открывает диалоговое окно *IDE Insight* для интерактивного поиска объектов ИСР.
Alt+G	**Go to Line Number**	Переходит к заданной строке текста в *Редакторе кода*.
Команды меню View		
Shift+Ctrl+H	**Help Insight**	Вызывает **подсказку** для выделенного объекта исходного кода.
Ctrl+F12	**Units…**	Открывает диалоговое окно *Search for units* со списком всех **модулей** проекта.
Shift+ F12	**Forms…**	Открывает диалоговое окно *Search for forms* со списком всех **форм** проекта.
F12	**Toggle Form/Unit**	Переключается между окнами *Редактора кода* и *Конструктора формы*.
Shift+ Alt+F11	**Structure**	Открывает окно **Структуры проекта**.
F11	**Object Inspector**	Открывает окно **Инспектора объ-**

		ектов.
Alt+0	**Window List...**	Открывает диалоговое окно *Window List* со списком окон проекта**.**
Ctrl+Alt+P	**Tool Palette**	Открывает окно **Палитры инструментов**.
Ctrl+Alt+F11	**Project Manager**	Открывает окно **Менеджера проекта**.
Ctrl+Alt+B	**Breakpoints**	Открывает окно **Breakpoints List** со списком всех точек останова.
Ctrl+Alt+B	**Call Stack**	Открывает окно **Call Stack** с вызовами стека.
Ctrl+Alt+W	**Watches**	Открывает окно **Watch List**.
Ctrl+Alt+L	**Local Variables**	Открывает окно **Local Variables**.
Ctrl+Alt+T	**Threads**	Открывает окно **Thread Status**.
Ctrl+Alt+V	**Event Log**	Открывает окно **Event Log**.
Ctrl+Alt+M	**Modules**	Открывает закладку **Modules**.
Команды меню Refactor		
Shift+Ctrl+E	**Rename...**	Открывает диалоговое окно для **переименования** заданного объекта.
Shift+Ctrl+V	**Declare Variable...**	Открывает диалоговое окно для **объявления локальных** переменных и массивов. Выделите в коде необъявленную переменную и выполните команду *Declare Variable*.
Shift+Ctrl+D	**Declare Field...**	Открывает диалоговое окно *Declare New Field* для **объявления** поля класса в разделе *private*.
Shift+Ctrl+M	**Declare Method...**	Открывает диалоговое окно для **создания нового метода** класса из выделенных строк исходного кода в разделе *private*.
Shift+Ctrl+A	**Find Unit...**	Открывает диалоговое окно *Find Unit.*
Команды меню Project		
Shift+F11	**Add to Project...**	Открывает диалоговое окно для **добавления** файла к проекту.
Ctrl+F9	**Compile ...**	**Компилирует** файлы текущего проекта, которые были изменены со времени предыдущей компиляции.
Shift+ F9	**Build ...**	**Создаёт** приложение с компиляцией **всех** модулей проекта.
Shift+Ctrl+F11	**Options...**	Открывает диалоговое окно для

		настройки параметров проекта.
Команды меню Run		
F9	**Run**	**Запускает** приложение в режиме отладки.
Shift+Ctrl+F9	**Run Without Debugging**	**Запускает** приложение без отладки.
F8	**Step Over**	**Выполняет** следующую строку исходного кода без захода в процедуру или функцию.
F7	**Trace Into**	**Выполняет** следующую строку исходного кода с заходом в процедуру или функцию.
Shift+F7	**Trace to Next Source Line**	**Выполняет** следующую строку исходного кода.
F4	**Run to Cursor**	**Выполняет** исходного кода до строки с курсором.
Shift+F8	**Run Until Return**	**Выполняет** исходного кода до строки с курсором или до возврата из функции.
Ctrl+F2	**Program Reset**	**Прерывает** выполнение программы в режиме отладки.
Ctrl+F7	**Evaluate/Modify...**	Открывает диалоговое окно для ввода **выражений** и новых значений **переменных**.
Ctrl+F5	**Add Watch...**	Открывает диалоговое окно *Watch Properties* для ввода **выражений.**

* * *

Итак, вооружившись с головы до пят дельфийскими инструментами, мы выходим на большую дорогу современной молодецкой удали, то есть трепанации программного кода.

Факультатив 2. Практические занятия, или Анатомия приложения

Во всём нужна сноровка,
Закалка, тренировка!

Наша песня

Чтобы **начать новый проект**, нажмите кнопку **New Items**  и в открывшемся диалоговом окне выберите **VCL Forms Application - Delphi**.

Теперь проект содержит пустую форму **Form1**, модуль формы **Unit1** и ещё несколько файлов. Как много *Delphi* уже сделал «за вас», вы можете убедиться, сохранив проект на диске. Откройте папку с проектом в каком-либо файловом менеджере и полюбуйтесь, сколько всего «натворил» *Delphi* почти без всякого «рукоприкладства» с вашей стороны. Вы найдёте такие файлы:

- **Project1.dproj** – файл описания проекта. В нём находится общая информация о проекте, изменять его не следует.

- **Project1.dpr** *(Main Project File)* – главный модуль проекта **Delphi**, в котором содержится текст модуля **Project1**. Вы можете посмотреть его в окне *Редактора кода*, выполнив команду *Главного меню* **Project > View Source**.

- **Project1.res** – файл ресурсов приложения, в котором сначала хранится только стандартный значок *Delphi*.

- **Unit1.dfm** (*Delphi Form File*) – файл формы приложения. Здесь в текстовом виде хранится описание формы. Как оно выглядит, вы можете посмотреть, выполнив команду контекстного меню окна *Конструктора формы* **View as Text**, а вернуться к обычному виду формы можно из контекстного меню *Редактора кода*, выбрав команду **View as Form** или нажав клавиши **Alt+F12**.

- **Unit1.pas** – файл модуля реализации.

При следующем сохранении файлов проекта (используйте кнопки *Панели инструментов*  или ) в папке **__history** появятся резервные (архивные) копии соответствующих файлов. К ним можно вернуться, чтобы

убрать внесённые в программу изменения. Для этого нужно просто стереть тильды в названии файла. Но вы должны учитывать, что так можно отменить только те изменения в программе, которые появились с момента последнего сохранения файлов. Так как проект следует сохранять довольно часто (по крайней мере, после каждой отладки кода), то далеко назад вернуться не удастся. Поэтому в папке проекта желательно завести ещё одну папку - для архивных файлов. В ней следует хранить все файлы проекта на той стадии разработки, когда значительная часть кода отлажена и не вызывает сомнений. Кроме того, список изменений вы можете увидеть на вкладке **History** 

Рабочего окна.



Если вы добавите к проекту одну или больше новых форм, то для каждой из них при сохранении проекта будут созданы файлы с расширением **DFM** и **PAS** (или один файл **PAS** при добавлении только нового модуля).

После того как вы сохранили файлы проекта под другими именами, они будут обновлены и в заголовках *Главного окна* и окна *Редактора кода*.

Вы уже сейчас можете **запустить** ваш проект. Нажмите кнопку  на *Панели инструментов*, и через некоторое время на экране появится форма, которую можно передвигать, сворачивать, разворачивать, изменять его размеры и закрывать.

Опять проверьте содержимое папки с проектом – в ней завелась парочка новых файлов:

Unit1.dcu *(Delphi compiled unit*) – откомпилированный файл модуля формы. Он используется при компиляции программы, после чего надобность в нём отпадает, и вы можете удалить его из папки (впрочем, это сизифов труд – он возникнет, как птица феникс, при следующем запуске программы).

Project1.exe – выполняемый код вашего проекта.



Имя проекта получат также файлы с расширениями **.exe**, **.dpr**, **.dproj**, .**cfg**, .**dsk**, а имя модуля – файлы **.pas**, **.dfm** и **.dcu**.

Файлы *Unit1.dcu и Project1.exe* ищите в папке **Win32 > Debug**. Или в папке **Win32 > Release**, если вы создали окончательнвую версию проектаю

Сохранённый на диске проект вы можете вновь **открыть** в *ИСР*.

Особенно удобна для этого кнопка **Open** на *Панели инструментов*.

Нажмите стрелочку справа от папки, чтобы раскрыть список последних проектов и файлов, и выберите нужный.

То же самое можно сделать и выполнив команду *Главного меню* **File > Re-open**.

С помощью команды **File > Open** вы сможете открыть любые доступные файлы, а команда **File > Open Project (Ctrl + F11)** «специализируется» на загрузке проектов и групп проектов (Рис. Ф2.1).

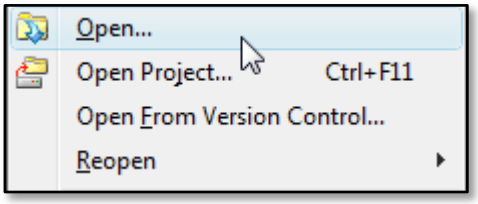


Рис. Ф2.1. Открывающие команды меню *File*

Так же действует кнопка на *Панели инструментов* **Open Project (Ctrl + F11)**.

В *Окне приветствия* имеется команда **Open Project**, а также списки последних проектов **(Recent Projects** и **Recently Opened Projects**), из которого очень удобно выбирать нужный.

Файл проекта

По сути, вся программа заключается в файле проекта (**dpr**-файле, *главном модуле проекта*), в котором выполняется инициализация приложения, создание форм и запуск приложения.

Для приложения, имеющего в своём составе только одну форму, файл проекта выглядит так (чтобы он появился в окне *Редактора кода*, выполните команду *Главного меню* **Project > View Source**):

```
program Project1;

uses
  Vcl.Forms,
  Unit1 in 'Unit1.pas' {Form1};
```

```
{$R *.res}

begin
  Application.Initialize;
  Application.MainFormOnTaskbar := True;
  Application.CreateForm(TForm1, Form1);
  Application.Run;
end.
```

Название проекта (программы) совпадает с именем файла проекта, под которым вы сохранили его на диске.

В разделе **uses** указывается подключаемый модуль **Vcl.Forms**, который является обязательным, если приложение имеет хотя бы одну форму, а также перечисляются все модули проекта (у нас это один модуль формы **Unit1**).

Директива **$R** присоединяет к проекту файл ресурсов. Как мы знаем, его имя совпадает с именем файла проекта, но имеет расширение **RES** (поэтому имя файла ресурсов всегда заменяется звёздочкой, которая и означает, что нужно подключить именно этот файл). Вы также можете подключить и другие файлы ресурсов (значки, курсоры, картинки), но обычно это делается в модуле главной формы проекта (примеры такого подключения вы найдёте во многих программах этой книги).

Следует учитывать, что ресурсы, подключённые к проекту директивой **$R**, будут включены в выполняемый код программы (**EXE**-файл) и могут сильно увеличить размер файла приложения. Поэтому в файлах ресурсов обычно помещают значки (**ICO**), курсоры (**CUR**) и маленькие картинки (**BMP**). Большие графические изображения лучше загружать из файла на диске при запуске программы (в этом случае вместе с выполняемым файлом в папке проекта (программы) должны находиться и графические файлы).

Файл проекта создаётся и изменяется автоматически. Если вы подключите к проекту новую форму, то в файл проекта будут добавлены две строки кода для неё. После сохранения проекта на диске файл проекта будет выглядеть так:

```
program Project1;

uses
  Vcl.Forms,
  Unit1 in 'Unit1.pas' {Form1},
  Unit2 in 'Unit2.pas' {Form2};
```

```
{$R *.res}

begin
  Application.Initialize;
  Application.MainFormOnTaskbar := True;
  Application.CreateForm(TForm1, Form1);
  Application.CreateForm(TForm2, Form2);
  Application.Run;
end.
```

Эти строчки будут так же автоматически исключены из файла проекта при удалении этой формы.

Так как *Delphi* самостоятельно формирует файл проекта, то обычно никакие изменения в него вручную не вносятся. По этой же причине в распечатках всех программ этой книги файл проекта не приводится.

Файлы формы

Delphi автоматически создаёт для нового проекта и два файла формы – **файл описания** (*form file*) с расширением **DFM** и **файл модуля** (файл реализации, *source file*) с расширением **PAS**. При конструировании формы все изменения в файл описания формы также вносятся автоматически (поэтому непосредственно этот файл редактируют редко).



Иногда приходится «вмешиваться» в содержимое **DFM**-файла. Например, вы использовали в проекте «нестандартный» компонент, который не работает в новой версии *Delphi*. Тогда вам достаточно удалить его из файла описания формы или заменить стандартным. При этом, вероятно, потребуются изменения и в файле реализации, но это уже не так страшно, как отсутствие целого компонента.

Вы также можете изменять свойства компонентов непосредственно в файле описания формы, но гораздо удобнее это делать в окне *Конструктора формы*!

Посмотрите файл формы в **текстовом** виде:

```
object Form1: TForm1
  Left = 0
  Top = 0
  Caption = 'Form1'
```

```
  ClientHeight = 339
  ClientWidth = 635
  Color = clBtnFace
  Font.Charset = DEFAULT_CHARSET
  Font.Color = clWindowText
  Font.Height = -11
  Font.Name = 'Tahoma'
  Font.Style = []
  OldCreateOrder = False
  PixelsPerInch = 96
  TextHeight = 13
end
```



В **контекстном меню** *Конструктора формы* выполните команду **View as Text**, и файл описания формы появится в окне *Редактора кода*.

Выполните команду **контекстного меню** *Редактора кода* **View as Form**, чтобы вернуться в *Конструктор формы*.

Вернитесь к обычному отображению формы, добавьте на неё какой-нибудь компонент (пусть это будет кнопка) и снова посмотрите файл описания формы – в нём появились строки, отражающие свойства и события (если для них задан обработчик) нового объекта формы:

```
object Form1: TForm1
  Left = 0
  Top = 0
  Caption = 'Form1'
  ClientHeight = 339
  ClientWidth = 635
  Color = clBtnFace
  Font.Charset = DEFAULT_CHARSET
  Font.Color = clWindowText
  Font.Height = -11
  Font.Name = 'Tahoma'
  Font.Style = []
  OldCreateOrder = False
  PixelsPerInch = 96
  TextHeight = 13
  object Button1: TButton
    Left = 288
    Top = 176
    Width = 75
    Height = 25
    Caption = 'Button1'
    TabOrder = 0
  end
end
```

Файл модуля формы

Файл модуля формы содержит информацию о форме и размещённых на ней компонентах. Состоит он из двух частей – **секции** (раздела) **описания** (**interface**) и **секции** (раздела) **реализации** (**implementation**).



Файл модуля формы может иметь и ещё две секции – **инициализации** и **финализации**, которые используются значительно реже.

В разделе **interface** находится список всех подключенных модулей и описание класса формы (с перечнем всех компонентов). Для пустой формы он выглядит так:

```
unit Unit1;

interface

uses
  Winapi.Windows,     Winapi.Messages,     System.SysUtils,     Sys-
tem.Variants, System.Classes, Vcl.Graphics,
  Vcl.Controls, Vcl.Forms, Vcl.Dialogs, Vcl.StdCtrls;

type
  TForm1 = class(TForm)
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation

{$R *.dfm}

end.
```

В разделе **implementation** к модулю с помощью директивы **$R** подключается файл описания формы (как и в файле проекта, его имя заменено звёздочкой, так как оно совпадает с именем модуля формы).

При размещении на форме какого-либо компонента или создании процедуры-обработчика событий для него *Delphi* дописывает в модуль формы необходимые строки. Опять установим кнопку и щёлкнем на событии **OnClick**. Теперь файл модуля формы будет выглядеть так:

```
unit Unit1;

interface

uses
  Winapi.Windows,     Winapi.Messages,     System.SysUtils,     Sys-
tem.Variants, System.Classes, Vcl.Graphics,
  Vcl.Controls, Vcl.Forms, Vcl.Dialogs, Vcl.StdCtrls;

type
  TForm1 = class(TForm)
    Button1: TButton;
    procedure Button1Click(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation

{$R *.dfm}

procedure TForm1.Button1Click(Sender: TObject);
begin

end;

end.
```

Легко заметить отличия: добавилась кнопка **Button1** типа **TButton** и обработчик нажатия мышкой. А в разделе **implementation** появилась заготовка для процедуры-обработчика.

Заготовка, как видите, совершенно пустая. К сожалению (а скорее, к счастью), *Delphi* не умеет «сочинять» программы. Вы сами должны выбирать и размещать на форме нужные компоненты, писать для них обработчики, вводить переменные, константы и заниматься всей остальной хитрой механикой (то есть программированием).

Главным образом проект разрабатывается в окне *Конструктора формы* (визуальная часть приложения) и в окне *Редакторе кода* (его функциональная часть), поэтому мы скоро и займёмся благоустройством этих окон.

Файлы модулей

В состав проекта могут входить и **отдельные модули**, не связанные ни с какой формой (но они тоже имеют расширение **PAS**). В них обычно помещают код процедур и других элементов программы на языке *Delphi*, которые могут использоваться в других модулях проекта (чтобы подключить к ним отдельные модули, достаточно указать их названия в разделе **uses** соответствующего модуля).

Подобные модули могут быть легко присоединены к любому другому проекту, поэтому в них целесообразно хранить часто используемые процедуры, константы, переменные.

Файлы ресурсов

При первом сохранении проекта на диске автоматически создаётся файл ресурсов проекта. Он имеет имя файла проекта, но с расширением **RES**. Сначала в нём находится только значок проекта, которым по умолчанию является пиктограмма *Delphi*. Затем значок проекта можно заменить или изменить.

Ресурс карман не тянет, или *Image Editor*

Всё своё ношу с собой.

Из мудрых мыслей

Для серьёзной работы с курсорами, иконками, растрами и файлами ресурсов есть удобные и мощные программы, но вот если вам нужно слегка подправить или создать несложный курсор, значок или картинку для кнопки, тогда **Image Editor** – то, что вам нужно: он очень быстро загружается и покладист в обращении (Рис. Ф2.2).



Редактор ресурсов находится в папке с программами для этой главы.

Чаще всего файл ресурсов проекта оставляют без изменения (значок программы можно заменить и проще), а для других ресурсов создают *отдельные* файлы. В наших программах мы будем использовать файл ресурсов **MyCursors.res** с курсорами.

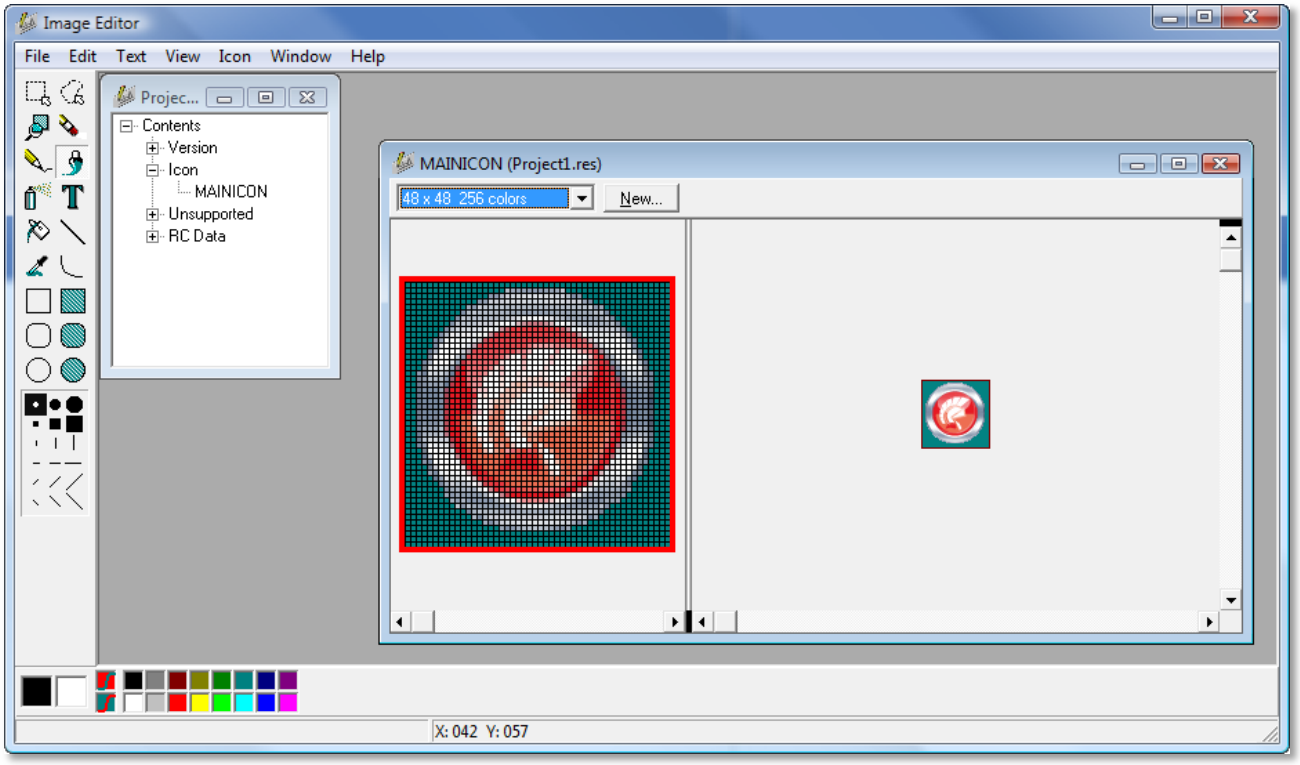


Рис. Ф2.2. Редактор ресурсов

А описание *Редактора ресурсов* мы начнём по привычке с **главного меню**.

File (Рис. Ф2.3)

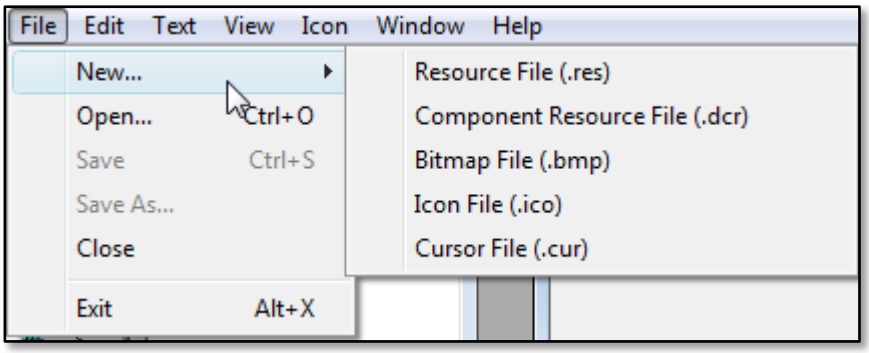


Рис. Ф2.3. Меню *File*

- **New** – создаёт новый ресурс (мы будем называть здесь ресурсом и сам файл ресурсов, и отдельные значки, курсоры и растры (картинки в формате **BMP**)).
- **Open** – открывает существующий ресурс.
- **Save** – сохраняет на диске текущий (находится в активном окне) ресурс под текущим именем.

- **Save As** – действует аналогично команде **Save**, но позволяет выбрать для ресурса новое имя.
- **Close** – закрывает окно текущего ресурса.
- **Exit** – закрывает все ресурсы и редактор.

Edit (Рис. Ф2.4)

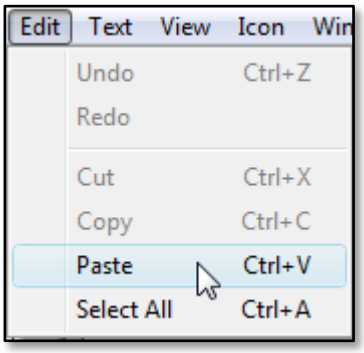


Рис. Ф2.4. Меню *Edit*

- **Undo** – отменяет последнее действие.
- **Redo** – возвращает последнее действие (отменяет команду **Edit > Undo**).
- **Cut** – вырезает выделенную область изображения и помещает её в буфер обмена (*clipboard*).
- **Copy** – копирует выделенную часть картинки.
- **Paste** – вставляет скопированную или вырезанную область в картинку (через буфер обмена можно передавать картинки, скопированные совсем в другом графическом редакторе).
- **Select All** – выделяет всю картинку.

Icon (Рис. Ф2.5)

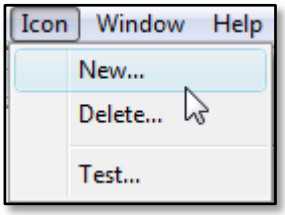


Рис. Ф2.5. Меню *Icon*

Этот пункт меню появляется тогда, когда вы редактируете значок (иконку, пиктограмму).

- **New** – создаёт пустую заготовку с другими свойствами (размер, количество цветов) для текущего значка.
- **Delete** – удаляет текущий значок с заданными свойствами.
- **Test** – проверяет, как будет смотреться значок на фоне выбранного цвета.

Resource (Рис. Ф2.6)

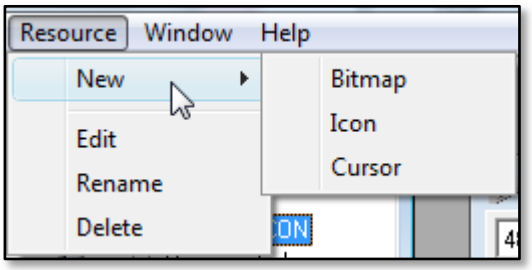


Рис. Ф2.6. Меню *Resource*

Этот пункт меню становится доступным, когда вы редактируете файл ресурсов.

- **New** – создаёт новый значок, курсор или растр для текущего файла ресурсов.
- **Edit** – позволяет редактировать выбранный значок, курсор или растр в отдельном окне.
- **Rename** – переименовывает выбранный значок, курсор или растр в составе файла ресурсов.
- **Delete** - удаляет выбранный значок, курсор или растр из файла ресурсов.

Text (Рис. Ф2.7)

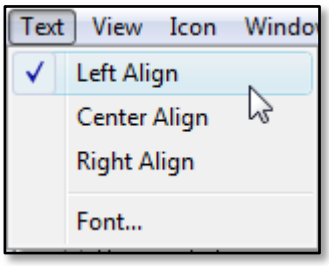


Рис. Ф2.7. Меню *Text*

Этот пункт меню появляется, когда вы добавляете к картинке текст.

- **Left Align, Center Align, Right Align** – задаёт способ выравнивания строки.
- **Font** – позволяет выбрать шрифт.

View (Рис. Ф2.8)

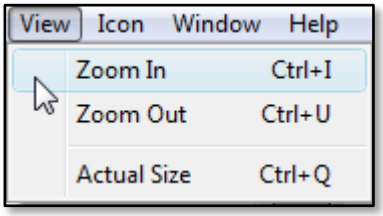


Рис. Ф2.8. Меню *View*

- **Zoom In** – увеличивает картинку (масштаб просмотра).
- **Zoom Out** – уменьшает картинку.
- **Actual Size** – восстанавливает действительный размер картинки.

Window (Рис. Ф2.9)

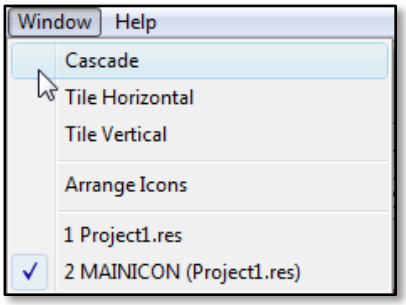


Рис. Ф2.9. Меню *Window*

Так как вы можете работать одновременно с несколькими ресурсами, для каждого из которых создаётся отдельное окно, то бывает полезно выстроить их так, чтобы все они были видны на экране:

- **Cascade** – располагает окна внахлёст.
- **Tile Horizontal** – располагает по горизонтали.
- **Tile Vertical** – располагает по вертикали.

Вы также можете выбрать из списка любое из открытых окон (сделать его текущим, активным).

Help (Рис. Ф2.10)

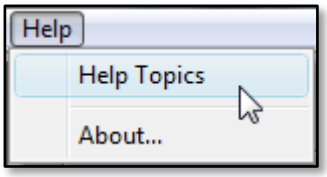


Рис. Ф2.10. Меню *About*

- **Topics** - открывает справочную систему **Редактора ресурсов**.
- **About...** - показывает текущую версию редактора (3.0).

А теперь давайте **создадим простейший файл ресурсов**.

1. Запустите **Image Editor** и выполните команду меню **File > New > Resource File**. Откроется новое окно, в заголовке которого находится имя файла по умолчанию (Рис. Ф2.11).

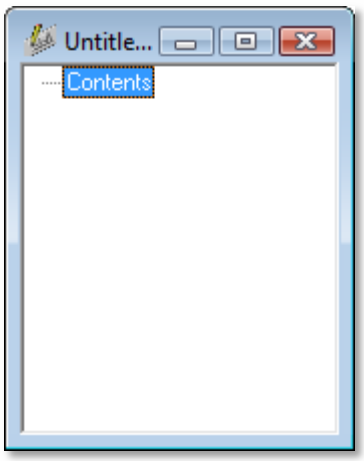


Рис. Ф2.11. Начинаем создавать файл ресурсов

2. Лучше сразу сохранить файл в папке с проектом под именем, например **MyRes**, командой **File > Save As** (затем достаточно будет выполнять команду **File > Save**). Проверьте - в папке появился новый файл ресурсов **MyRes.res**. Естественно, в нём пока ничего нет.

3. Добавим в файл растровую картинку. Для этого нужно выполнить команду меню **Resource > New > Bitmap** или команду **New > Bitmap** в контекстном меню окна файла ресурсов. В появившемся на экране диалоговом окне выберите размеры и число цветов для картинки (Рис. Ф2.12).

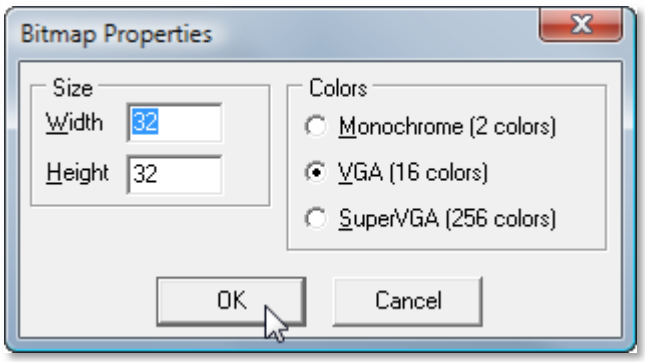


Рис. Ф2.12. Добавляем картинку

В окне файла ресурсов начнёт расти «дерево» ресурсов. Пока у него только одна ветка - с растрами, на которой всего один листок – початая нами картинка (Рис. Ф2.13).

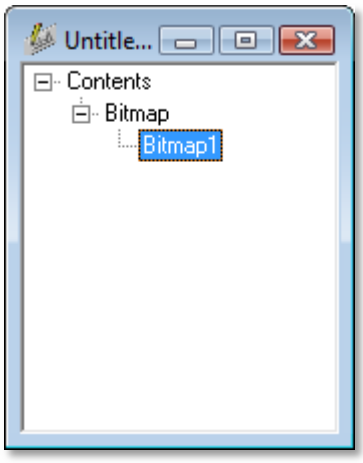


Рис. Ф2.13. С пополнением!

Правильнее будет заменить её название более «говорящим». На этой картинке мы нарисует крестик, который вполне уместно назвать **PLUS** (вы можете вводить название в любом регистре, всё равно все названия в файле ресурсов будут напечатаны заглавными буквами). Чтобы изменить название, выделите его мышкой и **осторожно** щёлкните на нём. Появится мигающий курсор и рамочка. Если же у вас неверная рука, то выберите в главном или контекстном меню команду **Rename**.

Дважды щёлкните на названии ресурса (**PLUS**) или выполните команду меню **Edit**. Откроется ещё одно окно – с вашей картинкой (Рис. Ф2.14).

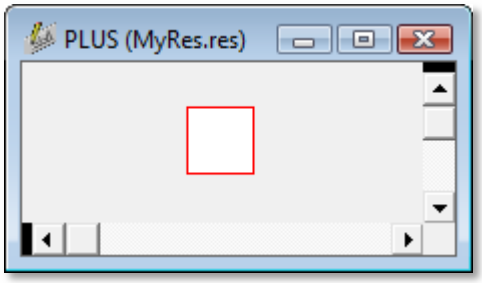


Рис. Ф2.14. Пустая картинка

Увеличьте картинку и окно так, чтобы появилась сетка (Рис. Ф2.15).

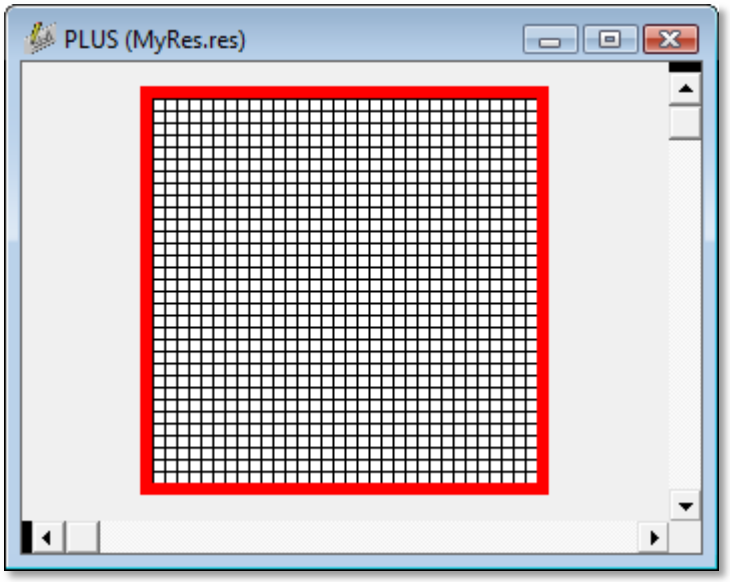


Рис. Ф2.15. Увеличенная картинка



Для изменения масштаба рисунка используйте команды главного меню **View > Zoom In**, **View > Zoom Out** и **Actual Size** (ещё удобнее их клавиатурные эквиваленты **Ctrl+I**, **Ctrl+U**, **Ctrl+Q**).

Чтобы **изменить размеры окна**, возьмите его мышкой за угол и двигайте в нужную сторону. Если размеры картинки большие, лучше развернуть главное окно редактора на весь экран.

Выберите в *Палитре инструментов* карандаш , подберите толщину линий (Рис. Ф2.16)



Рис. Ф2.16. Толщина линий

и их цвет (Рис. Ф2.17).

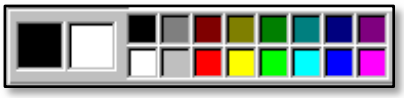


Рис. Ф2.17. Цвет линий

Нарисуйте крест или любой другой объект – насколько вам позволяют ваши художественные способности (Рис. Ф2.18).

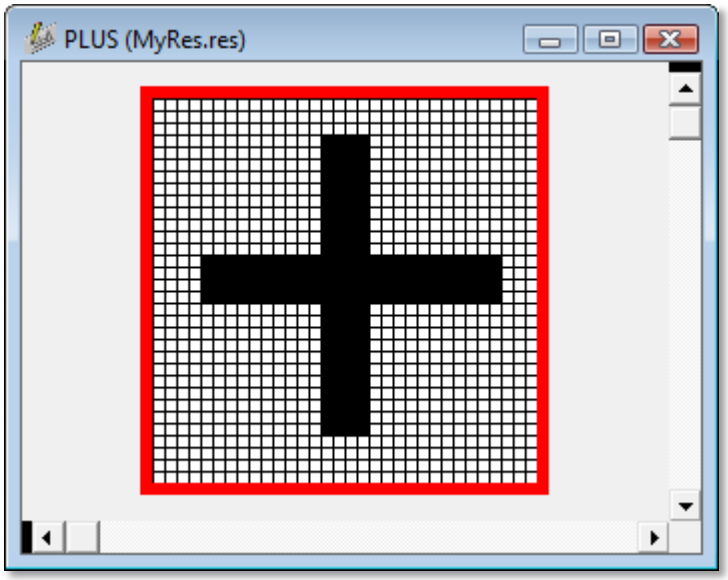


Рис. Ф2.18. Ставим крест

Щёлкните на окне с файлом ресурсов (чтобы сделать окно активным) и сохраните его на диске. Ваши труды не пропали даром: в файле ресурсов появился рисунок с крестиком-плюсиком. Вы можете подключить этот файл к своему проекту так, как это было описано выше, и использовать в программе, например, для вывода картинки на кнопку или в ячейку сетки. Подобные примеры вы найдёте в наших программах.

Точно так же вы можете добавить к файлу ресурсов ещё один растр (Рис. Ф2.19).

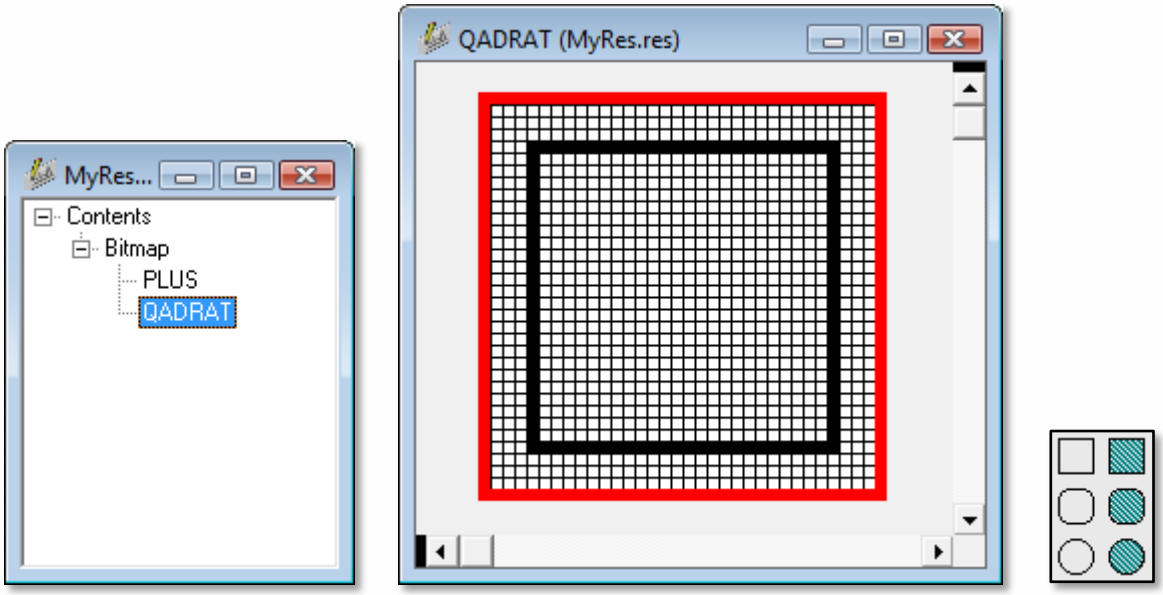


Рис. Ф2.19. Квадратная картинка

В той же *Палитре* вы найдёте инструменты для рисования прямоугольников (квадратов), прямоугольников с закруглёнными углами, эллипсов (кругов) – как контурных, так и с заливкой. С остальными инструментами вы легко разберётесь сами (Рис. Ф2.20).

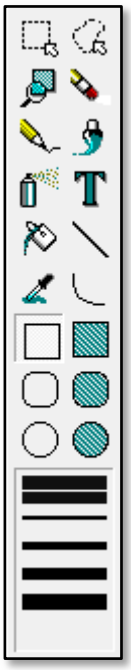


Рис. Ф2.20. Инструменты рисования

Особо следует отметить инструменты прямоугольного и произвольного выделений, находящиеся наверху *Палитры инструментов*. С их помощью вы сможете передвинуть рисунок или его фрагмент в окне, так как других средств для этого *Редактор ресурсов* не имеет.

4. Добавим к файлу рисунков значок, выполнив команду контекстного меню **New > Icon**. Назовём его **KRUG** и откроем в отдельном окне. С помощью тех же инструментов нарисуем мордочку посимпатичнее (возможно, шарж на вашего товарища) (Рис. Ф2.21).

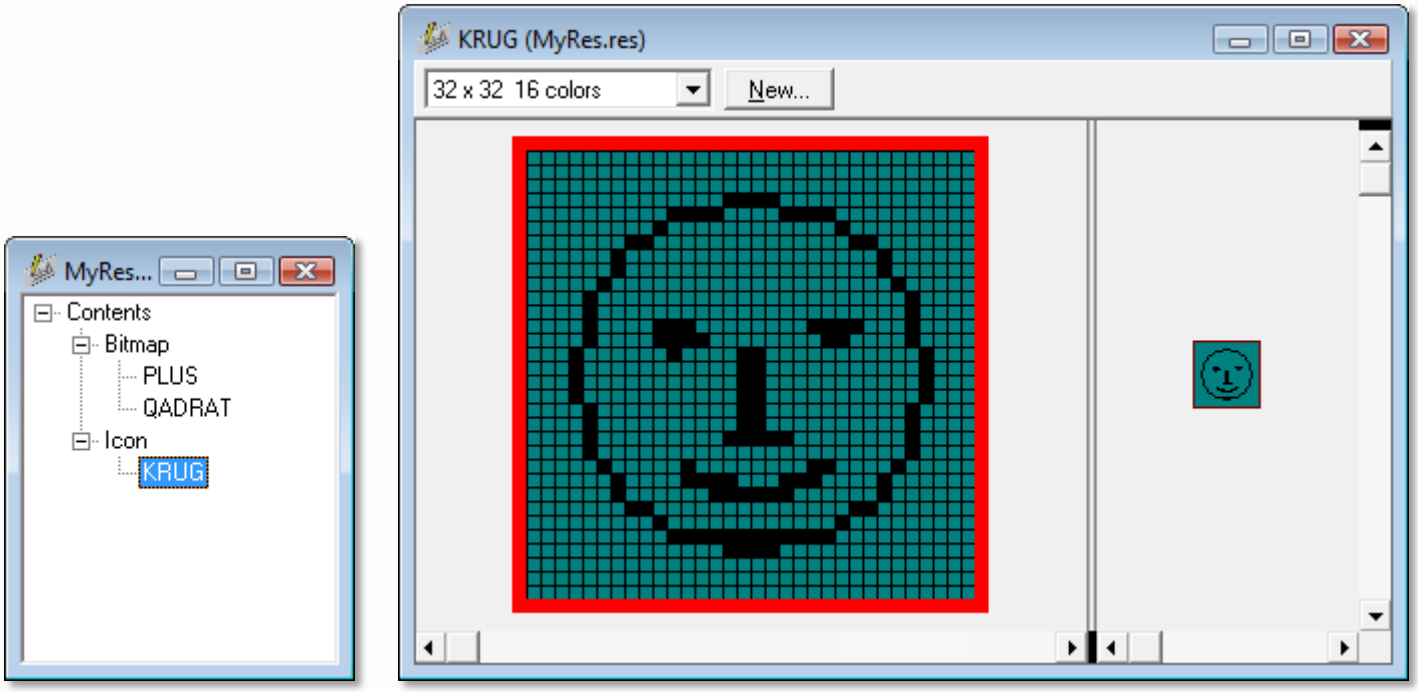


Рис. Ф2.21. Икона!

Можете посмотреть получившийся рисунок на любом фоне, воспользовавшись командой меню **Icon > Test...** Вдоволь насладившись содеянным, сохраните файл ресурсов на диске.

5. И последний эксперимент – присовокупим к нашим ресурсам курсор (**New > Cursor**) и назовём его **ARROW** (Рис. Ф2.22).

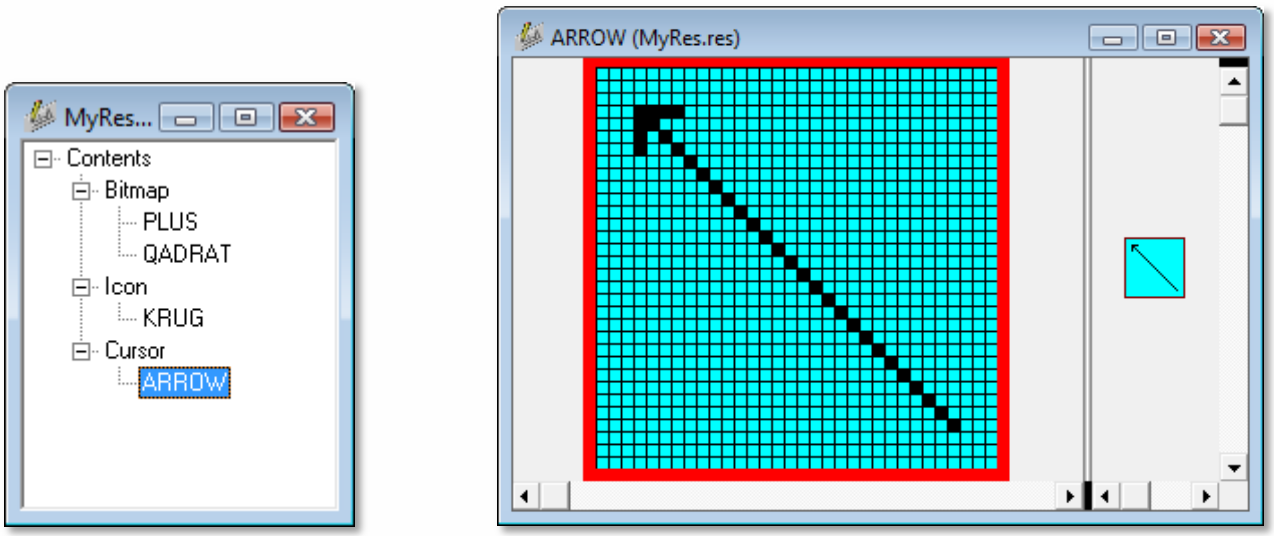


Рис. Ф2.22. Курсор-стрелка

Откроем окно для создания курсора и нарисуем диагональную стрелку. В отличие от предыдущих двух ресурсов, с курсором придётся повозиться подольше – для него нужно задать «горячую точку», координаты которой и будут передаваться в программу при движении курсора по окну приложения. В этом примере за горячую точку естественно принять координаты острия. Чтобы их узнать, подведите курсор *Редактора ресурсов* в нужную точку и посмотрите значения координат в строке состояния, которая находится внизу. Для нашего курсора вы там увидите "**X: 003 Y: 003**". Выполните команду меню (главного или контекстного) **Cursor > Set Hot Spot...** и в появившемся на экране диалоговом окне введите запомненные числа (Рис. Ф2.23).

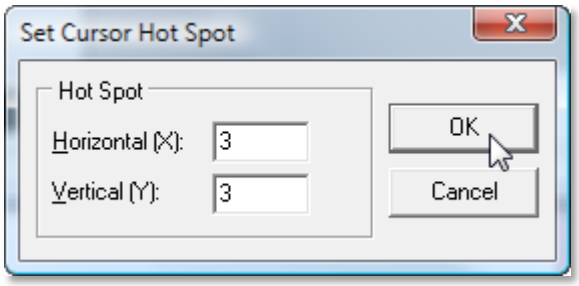


Рис. Ф2.23. Устанавливаем горячую точку

Не забудьте проверить созданный курсор в окне **Cursor Tester**, которое появится после выполнения команды меню **Cursor > Test...**

6. Сохраните файл проекта и закройте *Редактор ресурсов*.

Помимо собственноручно нарисованных картинок вы можете добавить в файл ресурсов и уже готовые. Сделать это очень просто. Откройте в отдельном окне графический файл нужного формата, скопируйте его целиком или частично в буфер обмена. Перейдите в окно с пустой сеткой для картинки файла ресурсов и вставьте изображение из буфера. Правда, тут есть одна тонкость. *Редактор ресурсов* не позволяет загружать растры, в которых используется более 256 цветов, поэтому их нужно предварительно в каком-нибудь графическом редакторе или просмотрщике (годится **ACDSee**) преобразовать в 256-цветный режим. Качество картинки при этом может сильно пострадать - тогда лучше подгружать её прямо из файла на диске при работе программы.

Вы можете копировать и вставлять картинки не обязательно одного и того же графического формата. Например, скопировав в буфер обмена растр, вы можете вставить его в окно значка или курсора. Если картинка при переносе исказится, то попробуйте конвертировать её в нужный формат в графическом редакторе. Таким образом вы сможете сделать из растра зна-

чок или курсор, что иногда бывает полезно. Через буфер обмена можно вставлять в окна ресурсов картинки и многих других графических форматов, если скопировать их туда из любого графического редактора.

Отдельные ресурсы создаются точно так же, как и те, что входят в состав файла ресурсов. Кроме того, в *Редакторе* можно создавать значки для компонентов наподобие тех, что вы видите в *Палитре компонентов*.

Параметры проекта

Для установки параметров проекта выполните команду *Главного меню* **Project > Options...** (**Shift+Ctrl+F11**). Перед вами откроется диалоговое окно **Project Options** (Рис. Ф2.24).

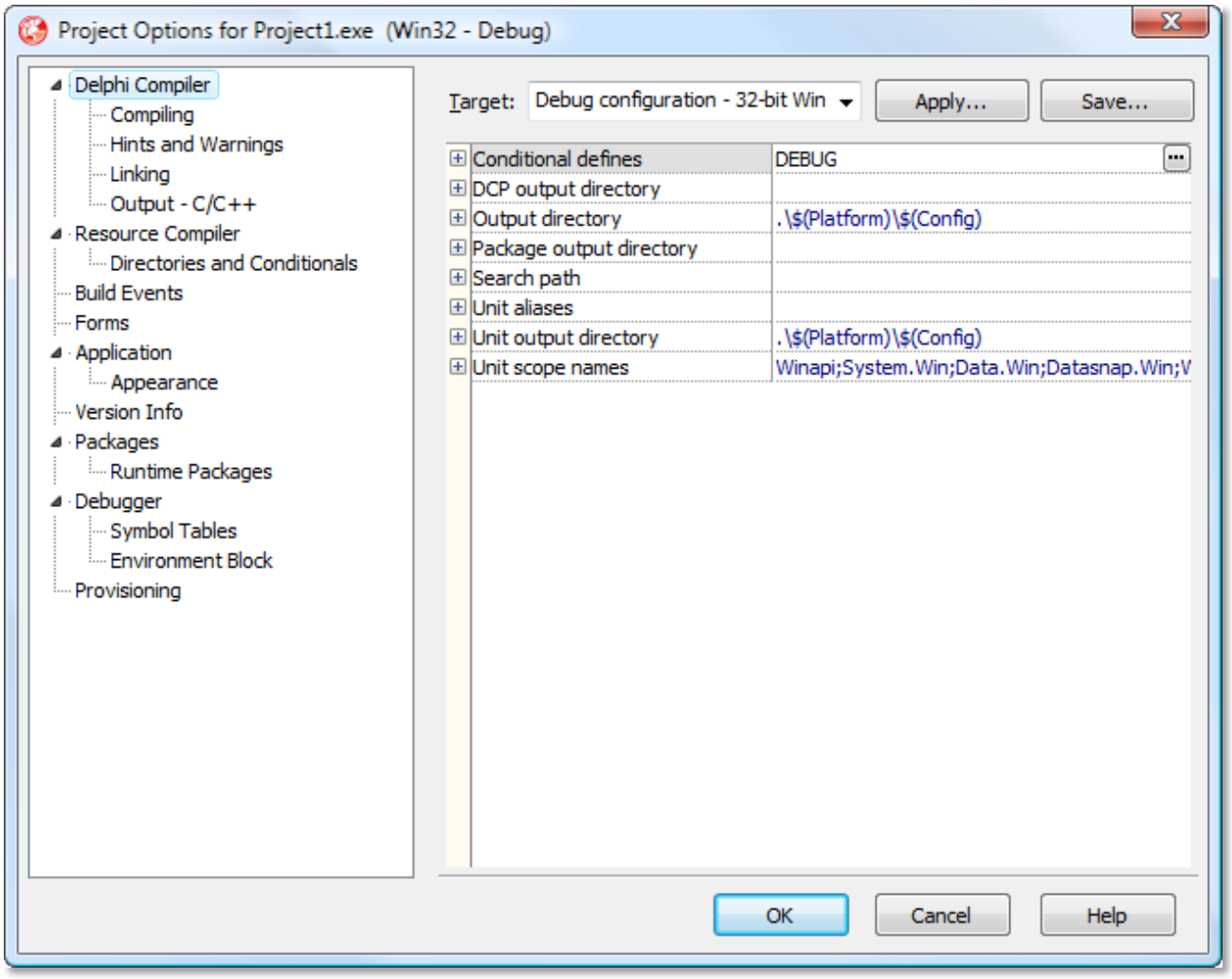


Рис. Ф2.24. Диалоговое окно *Project Options*

В большинстве страниц мы пока ничего менять не будем, для наших приложений вполне сгодятся установки умолчанию.

Выделив строку **Forms**, можно выбрать главную форму проекта (если их несколько), то есть ту форму, которая появится на экране после запуска приложения. Она всегда находится в начале списка форм в окне **Auto-create forms**. Обычно главной является та форма, с которой мы и начинаем работу в окне *Конструктора формы* (в нашем примере это **Form1**).

Формы, находящиеся в списке **Auto-create forms**, автоматически создаются при запуске приложения и находятся в памяти компьютера. Если в вашем проекте много больших форм, то вы можете создавать их самостоятельно по мере необходимости методом **Create**. В этом случае их нужно перенести в список **Available forms**. Для этого выделите название нужной формы в левом списке и нажмите кнопку . **Порядок форм** в списке вы можете легко изменить, подцепив одну из форм мышкой и перетащив на другое место.

Строка **Application** позволит вам выбрать значок (кнопка **Load Icon...**) для своего приложения, который украсит *Панель задач* и будет представлять ваш проект в списке файлов на диске (Рис. Ф2.25, сверху). Команда **Appearance** открывает новое диалоговое окно (Рис. Ф2.25, снизу), в котором вы можете задать имя приложения (**Title**), которое появляется в *Панели задач Windows*, когда приложение запущено, и добавить к проекту файл помощи (**Help file**).

После изменения параметров проекта они будут автоматически внесены в соответствующие файлы проекта. Например, параметры, установленные нами в опциях **Forms** и **Application**, появятся в файлах проекта и ресурсов.

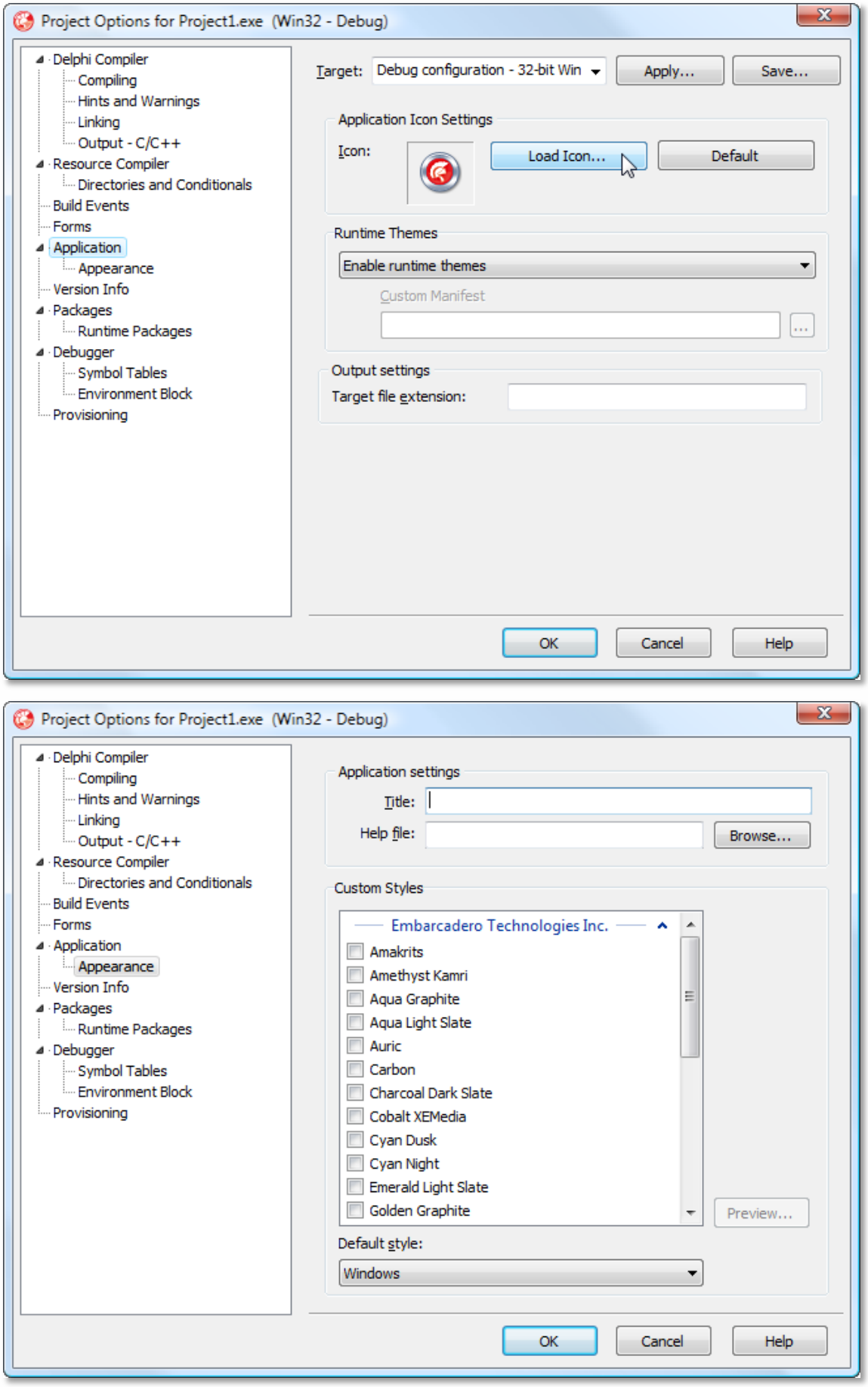


Рис. Ф2.25. Настройка параметров приложения

Как нам обустроить *Delphi*

Сделайте себе красиво!

Парафраз из Маяковского

Вы можете настроить среду разработки приложений *Delphi* так, чтобы вам было удобно и приятно ей пользоваться. Некоторые возможности мы уже рассмотрели – это аранжировка кнопок на *Панелях инструментов* и компонентов в *Палитре*. Но это ещё не всё!

До настроек среды разработки можно «добраться» из *Главного меню*: **Tools > Options...** (Рис. Ф2.26).

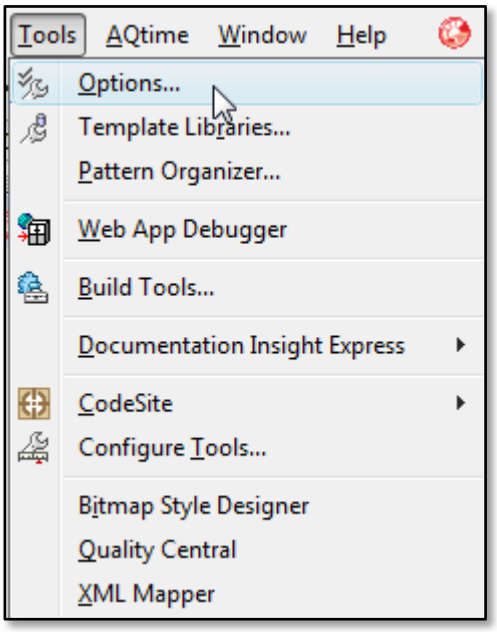


Рис. Ф2.26. Начинаем настройку ИСР

После выполнения этой команды на экране появится диалоговое окно с обширным меню (Рис. Ф2.27).

Далее мы рассмотрим только настройки непосредственно ИСР (**Environment Options**) и *Редактора кода* (**Editor Options**).

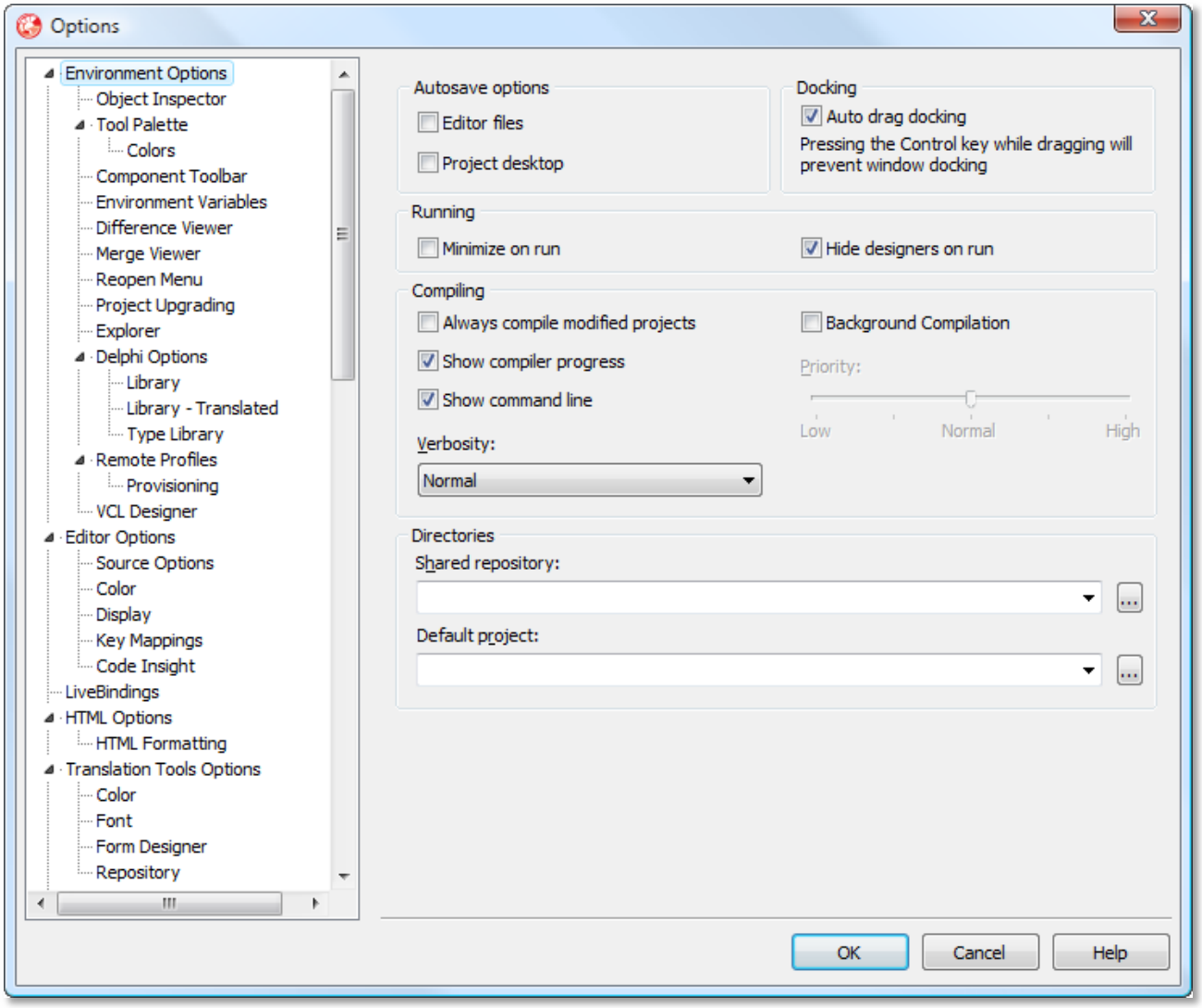


Рис. Ф2.27. Диалоговое окно *Options*

Редактор кода

Начнём с облагораживания *Редактора кода*. Щёлкните на строчке **Editor Options**, и перед вами возникнет диалоговое окно (Рис. Ф2.28).

Из него легко перейти на первую вкладку – **Source Options** (Рис. Ф2.29).

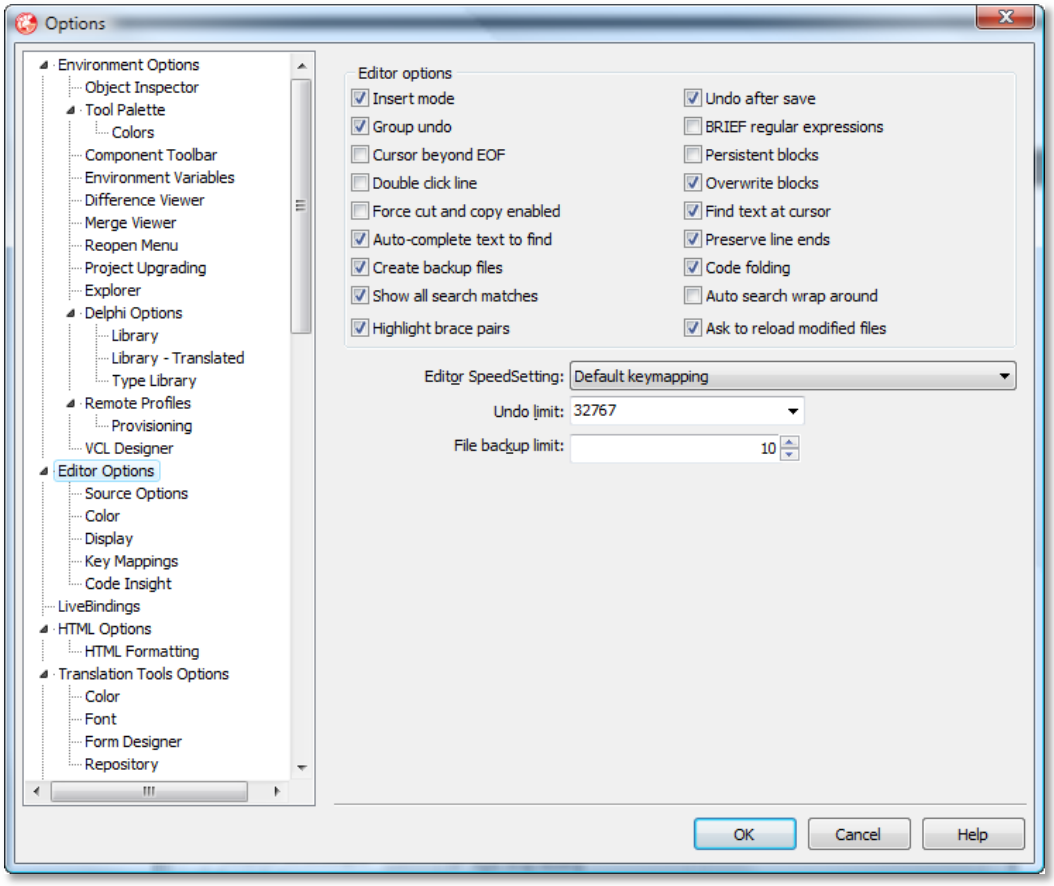


Рис. Ф2.28. Настраиваем *Редактор кода*

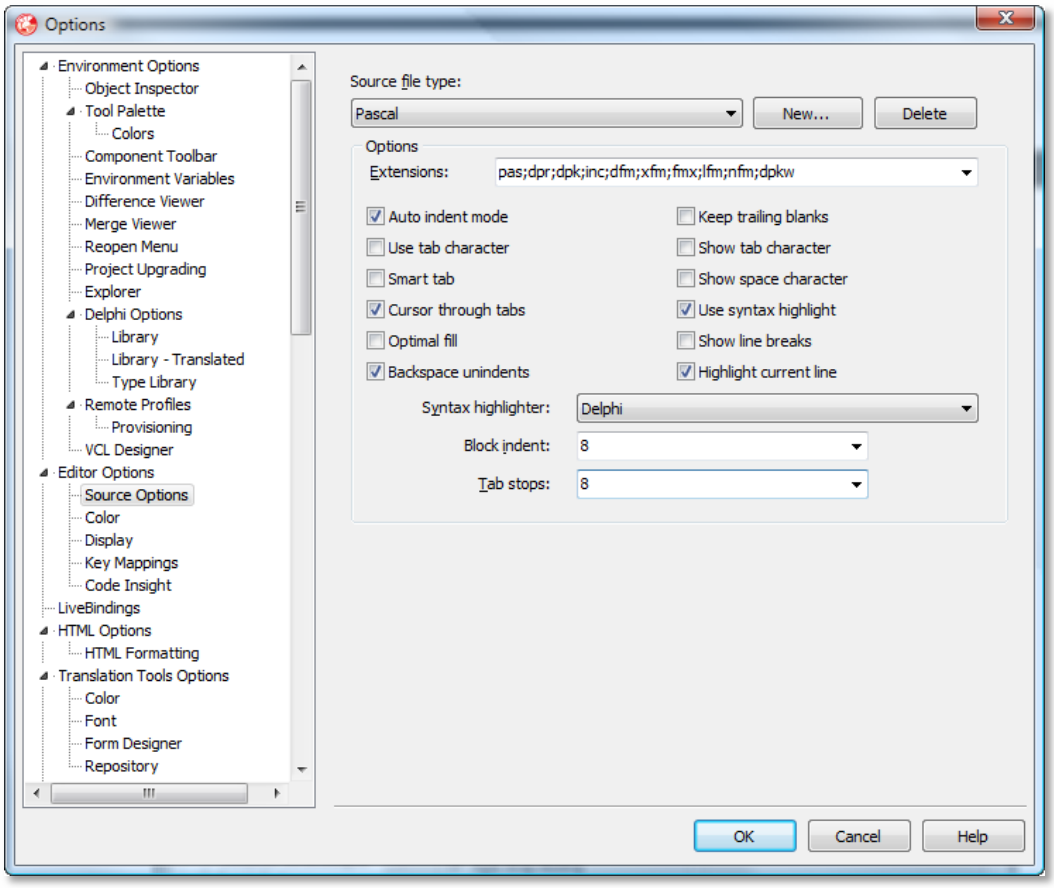


Рис. Ф2.29. Вкладка *Source Options*

В этих диалоговых окнах можно задать параметры сохранения и отображения текста на экране.

- **Insert Mode** – если флажок установлен, то новый текст раздвигает уже имеющийся, иначе он затирает старый (переключать эти режимы удобнее клавишей **Ins** на клавиатуре).

- **Use Tab Character** – вставляет символ табуляции, если флажок включен, или пробелы в противном случае (при включении **Smart Tab** этот флажок выключается).

- **Smart Tab** – табуляция до первого пробела (при включении **Use Tab Character** этот флажок выключается).

- **Double Click Line** – выделяет строку в *Редакторе кода* (флажок установлен) или слово (флажок сброшен) при двойном щелчке.

- **Use Syntax Highlighting** – выделяет синтаксис цветом (устанавливается на странице **Color**).

На странице **Display** устанавливаются параметры страницы и шрифт текста в *Редакторе кода* (Рис. Ф2.30).

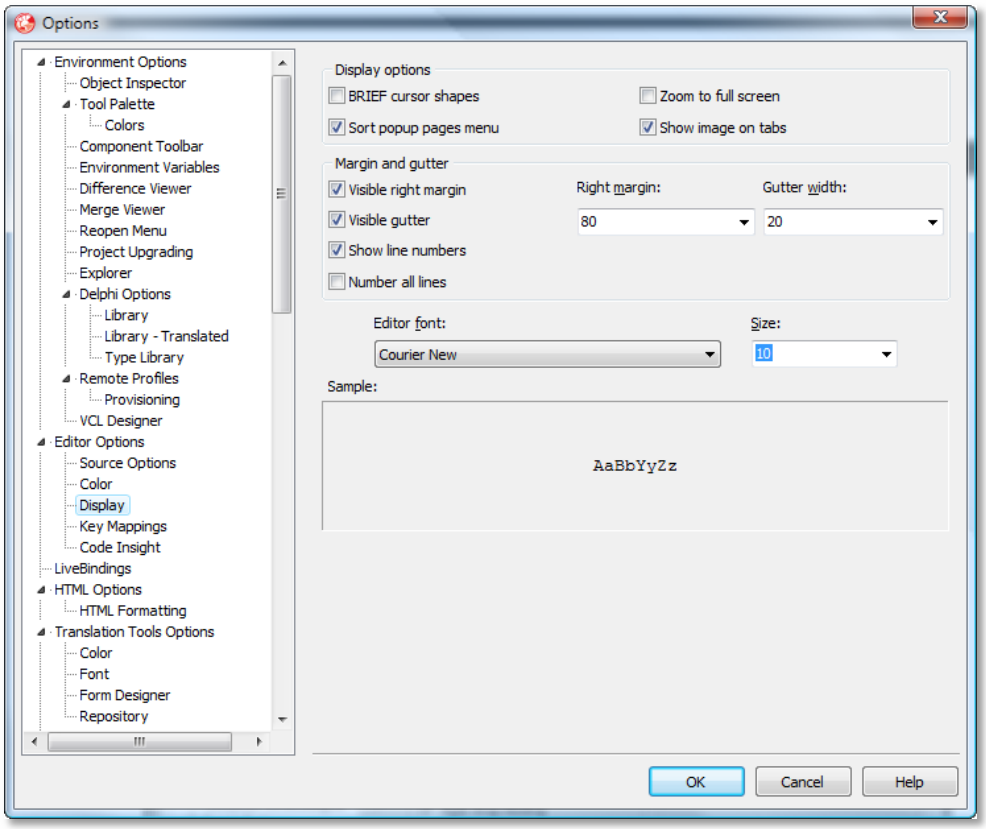


Рис. Ф2.30. Страница *Display*

- **BRIEF cursor shapes** - изменяет вид текстового курсора с вертикальной полоски на горизонтальную.

- **Zoom to full screen** – распахивает окно *Редактора кода* во весь экран.

- **Visible right margin** – показывает правую границу строки.

- **Right margin** – устанавливает правую границу строки (вертикальная пунктирная линия). По умолчанию длина строки равна 80 символам (допустимые значения 0..1024).

- **Visible gutter** – показывает вертикальную полоску слева от текста (нужна для установки точек останова и других целей).

- **Gutter width** – задаёт ширину (в пикселях) этой полоски. Если полоска очень узкая, то значки на ней будут усекаться!

- **Show line numbers** – показывает номера строк, оканчивающихся на ноль (через десять).

- **Number all lines** - показывает номера *всех* строк.

- **Editor font** – позволяет выбрать шрифт из списка доступных. *Редактор кода* использует только шрифты с символами равной ширины (не пропорциональные). Начертание шрифта можно посмотреть в окне **Sample**. Проверьте, содержит ли шрифт русские буквы!

- **Size** – устанавливает высоту шрифта.

Работать в окне **Color** интереснее всего! Здесь можно выбрать атрибуты шрифта (жирный (**Bold**), наклонный (*Italic*), подчёркнутый (Underline)), цвет символов (**Foreground color**) и фона (**Background color**) для всех элементов кода, которые выбираются поочерёдно из раскрывающегося списка **Element** (Рис. Ф2.31).

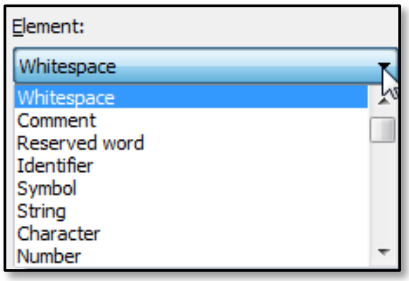


Рис. Ф2.31. Список элементов кода

Постарайтесь не переборщить с цветами, иначе исходный код будет напоминать своей пестротой тропического попугая!

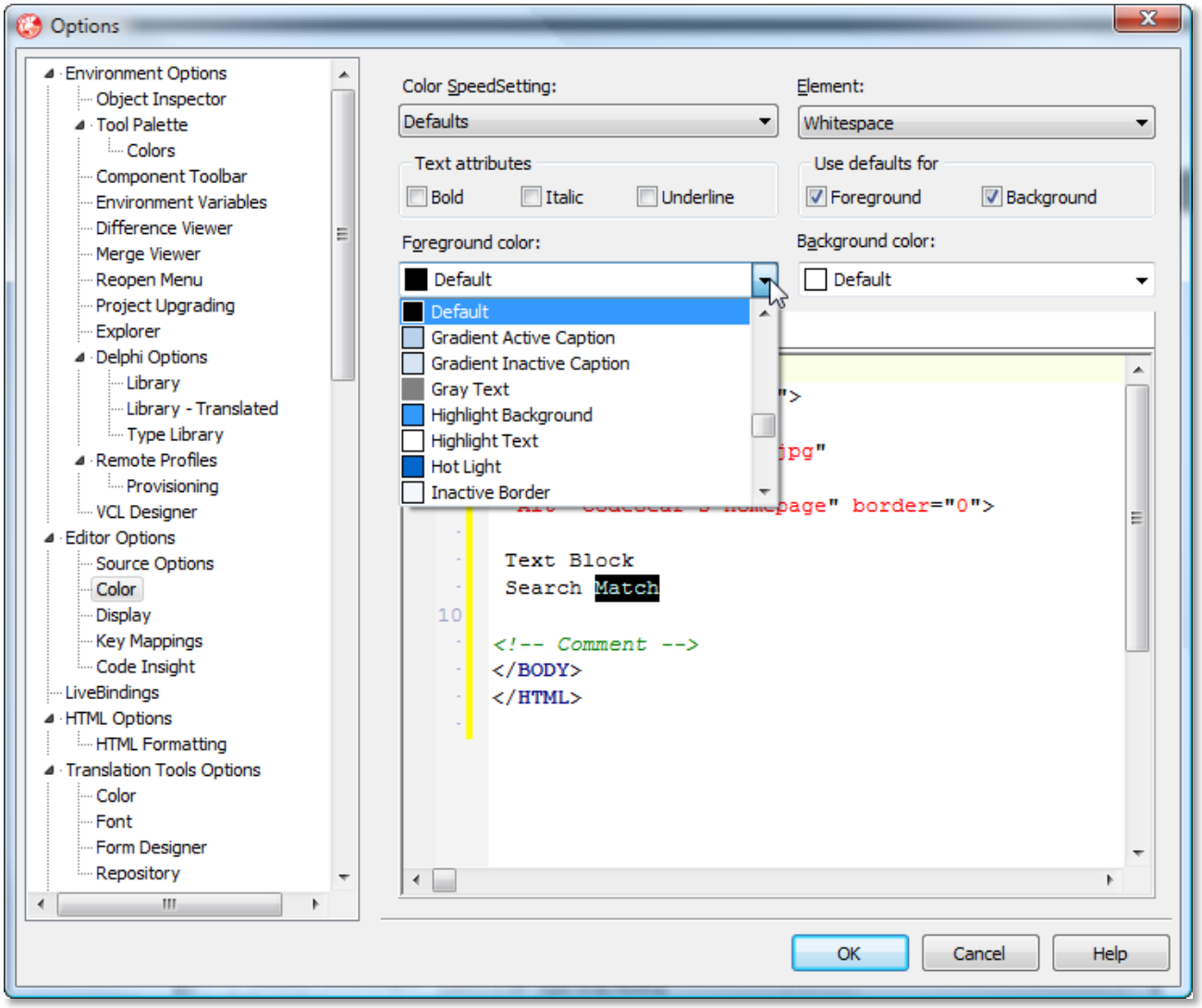


Рис. Ф2.32. Выбираем цвета

Можете довериться чужому вкусу и оставить все настройки по умолчанию или выбрать из выпадающего списка **Color SpeedSetting** какой-нибудь из имеющихся стилей (Рис. Ф2.32).

Если установлены флажки **Use defaults for**, то для выбранного элемента кода будет использоваться системный цвет *Windows* (символа и/или фона). Для изменения системных цветов используйте *Контрольную панель Windows*.

Много полезного вы найдёте на странице **Code Insight** (Рис. Ф2.33).

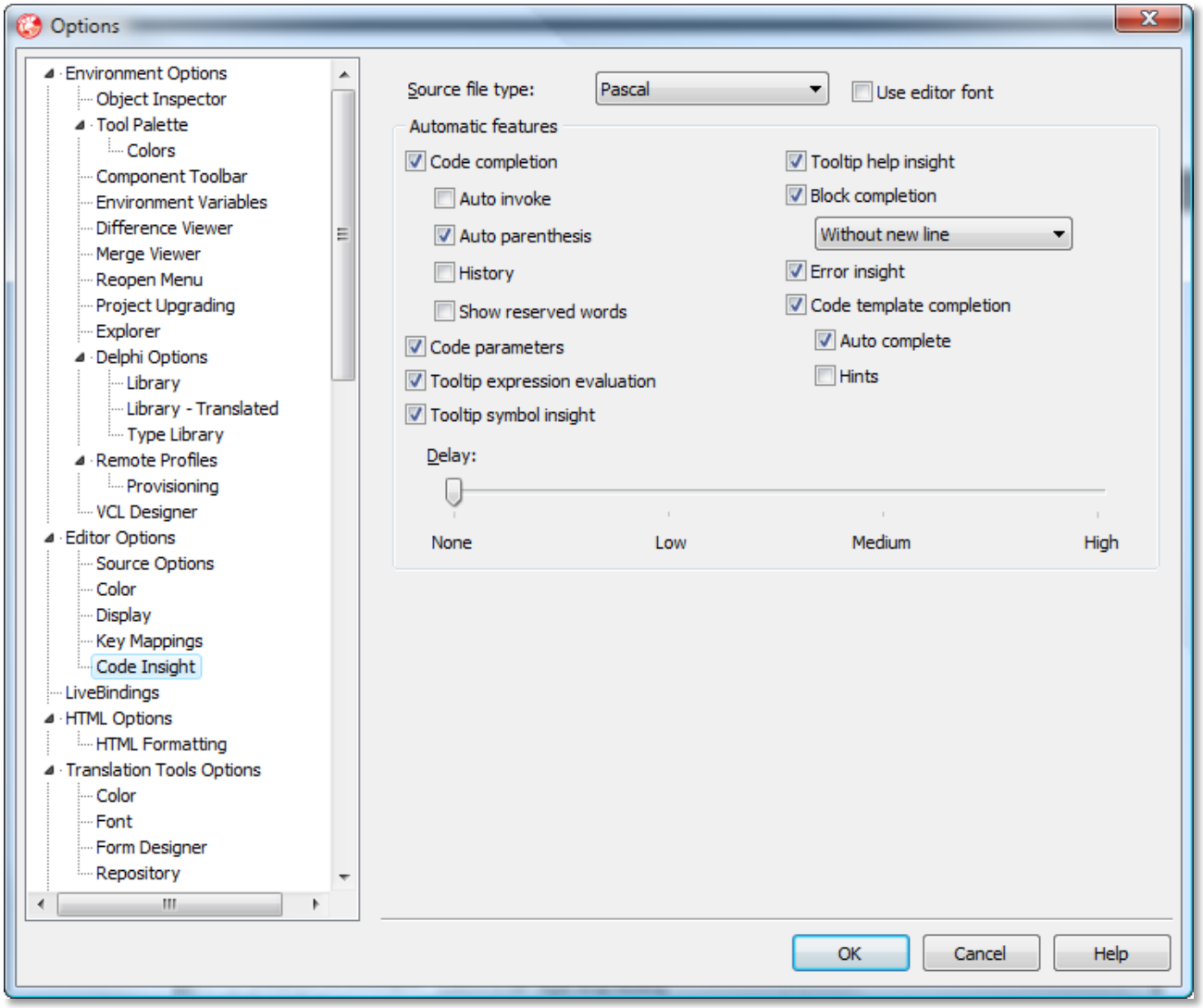


Рис. Ф2.33. Страница *Code Insight*

- **Code completion** – при вводе имени класса (или записи) и точки на экране появляется список доступных свойств, методов и событий для него. Выберите нужную строку и нажмите клавишу **Enter,** чтобы добавить её к программному коду.



Установите курсор на какой-либо элемент кода и нажмите **Ctrl+Spacebar** (пробел). Если для него имеется подсказка, она появится на экране. Таким образом можно вызывать подсказки и при сброшенном флажке **Code completion**.

- **Code parameters** – показывает синтаксис прототипа метода, когда вы набираете его имя.

- **Tooltip expression evaluation** – при остановке программы на отладку позволяет посмотреть значение переменной в *Редакторе кода*, если установить курсор на её название.

- **Tooltip Symbol Insight** – выводит подсказку (*hint*) для каждого идентификатора, на названии которого останавливается курсор.

- **Delay** – устанавливает задержку перед выводом подсказки **Tooltip Symbol Insight**.



Обратите внимание: в окне **Source file** type должен быть выбран **Pascal**!

Настройка параметров среды

Начинается с вызова диалогового окна **Environment Options** (Рис. Ф2.34).

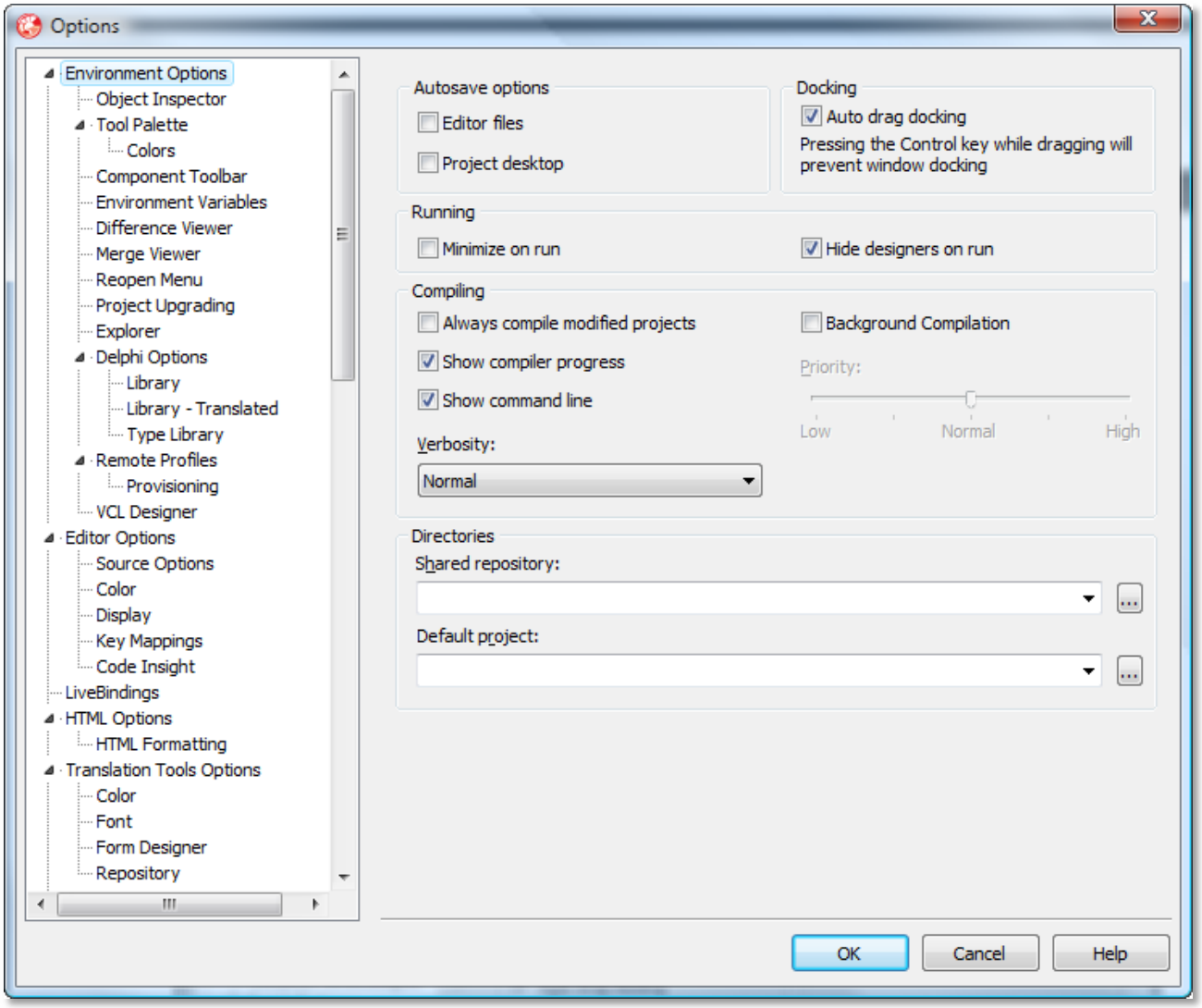


Рис. Ф2.34. Диалоговое окно *Environment Options*

В группе настроек **Autosave options** можно указать, какие файлы будут автоматически сохраняться на диске при запуске программы.

- **Editor Files** – если флажок установлен, то будут сохраняться все файлы, изменённые при выполнении команд **Run>Run, Run>Trace Into, Run>Step Over, Run>Run To Cursor**, а также при выходе из *Delphi*.

- **Project Desktop** – будут записаны все настройки рабочей среды при закрытии проекта или выходе из *Delphi*. Они будут восстановлены при следующей загрузке проекта.

Группа настроек **Compiling** и **Running** отвечает за компиляцию и запуск приложения.

- **Show Compiler Progress** – показывает ход компиляции (для визуальных гурманов).

- **Minimize On Run** - убирает все окна *Delphi* с экрана в *Панель задач Windows* при запуске программы командой **Run>Run**. Они восстанавливаются при закрытии приложения.

- **Hide Designers On Run** – скрывает окна *Конструктора форм* и *Инспектора объектов* при запуске приложения. Они восстанавливаются при закрытии приложения.

Страница **VCL Designer** позволяет управлять свойствами сетки на форме, которая служит для облегчения выравнивания компонентов на ней (Рис. Ф2.35).

- **Display Grid** – показывает на форме точки, образующие настроечную сетку.

- **Snap To Grid** - автоматически привязывает компоненты к ближайшей линии сетки. Если этот флажок установлен, то вы не сможете установить компонент в промежуточное положение между узлами сетки, перетаскивая его мышкой. В этом случае для точной настройки обратитесь к Инспектору объектов, в котором можно задать нужные координаты компонента на форме или ином контейнере.

- **Grid Size X** – шаг сетки по горизонтали (2..200).

- **Grid Size Y** – шаг сетки по вертикали (2..200).



Установленные по умолчанию значения - 8, 8 – не позволяют «тонко» устанавливать компоненты на форме, поэтому лучше уменьшить эти значения до 4 пикселей.

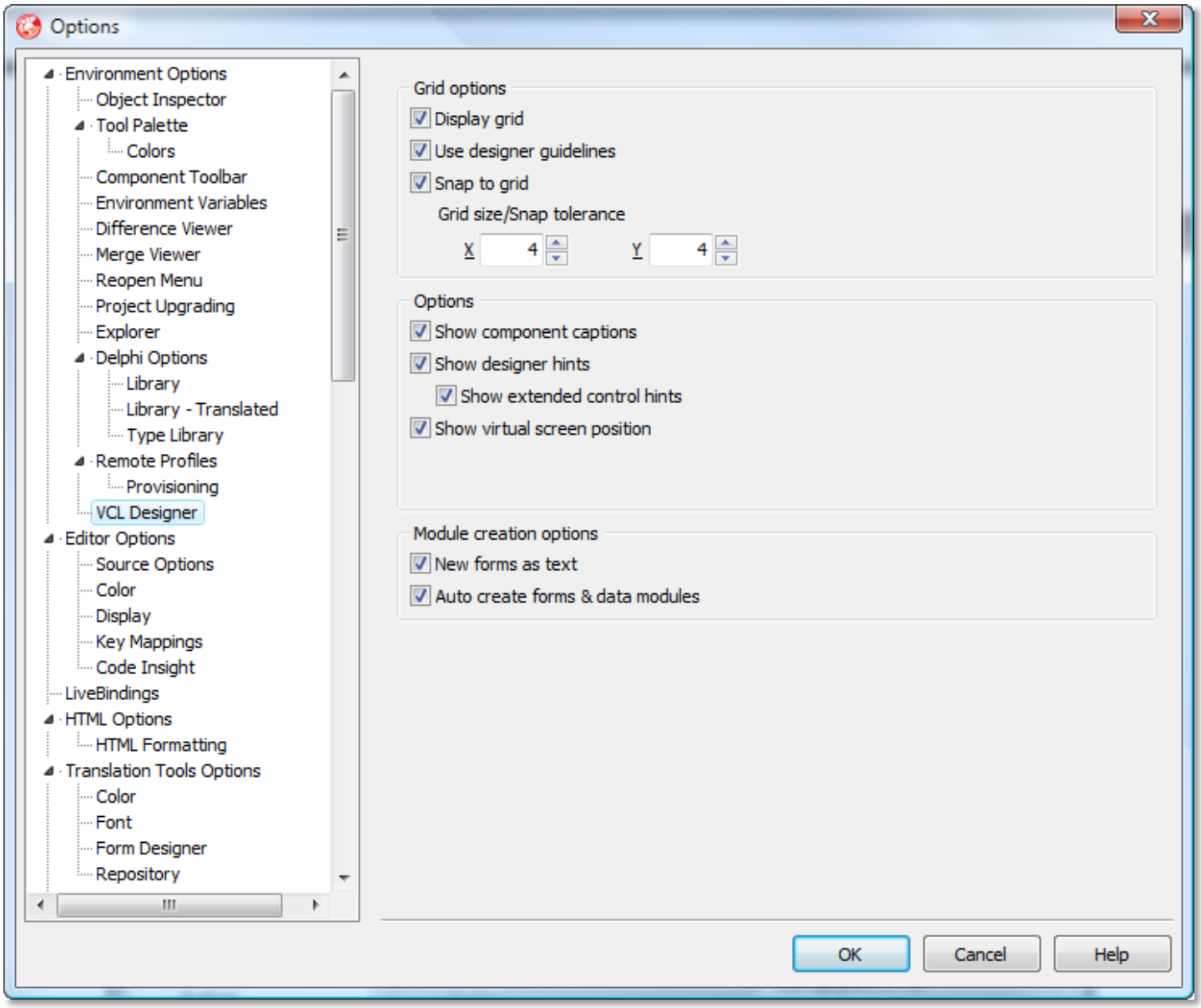


Рис. Ф2.35. Страница *VCL Designer*

А теперь приглашаю вас проследовать в следующую главу, в которой мы препарируем форму приложения вдоль и поперёк.

Факультатив 3. Формы и окна

Главное, натурально, форма!

Кредо Голохвастова, героя
комедии *За двумя зайцами*

Как редкая птица долетит до середины Днепра, так и редкая программа обходится без формы. Конечно, существуют в природе приложения *Windows*, которые работают, не показывая своего лица подобно Гюльчатай, или стыдливо прячут свой значок в *Панель задач*, чтобы оттуда исполнять свои скудные обязанности через всплывающее меню. Но нас такие эфемерные фантомы интересуют мало, так как заниматься мы будем исключительно важными делами.

При запуске программы на экране появляется *Главное окно* приложения, которое обеспечивает взаимодействие между пользователем и программой (интерфейс). Чаще всего главным окном приложения является то, которое находится в окне *Конструктора формы* при старте *Delphi* (или при открытии нового проекта). Но приложение может иметь и несколько форм - тогда любую из них можно «назначить» главной. Для этого откройте диалоговое окно **Project > Options...** (см. *Факультатив 2*, раздел Параметры проекта) и переместите мышкой название нужной формы в начало левого списка форм.

Все прочие окна приложения служат для организации диалогов с пользователем (получение от него какой-либо информации), для вывода сообщений и тому подобных действий. Вид этих окон можно выбирать, исходя из выполняемых ими функций.

Элементы стандартного окна

Любое «настоящее» приложение *Windows* «обязано» иметь своё окно, ибо всё управление программой (как пользователем, так и операционной системой) осуществляется именно через него. **Окном** в ОС *Windows* называется прямоугольная область экрана, имеющая необходимые для работы окна элементы.



Окно приложения создаётся даже тогда, когда оно не имеет его видимого воплощения на экране.

Окна могут быть не только прямоугольными, но и любой другой формы.

При запуске *Delphi* вы можете видеть окно будущего приложения в окне (ох, уж эти окна!) *Конструктора формы*. Впрочем, форма и окно это не совсем одно и то же. Можно сказать, что **форма** – это окно на стадии его проектирования (конструирования). Поэтому форма умеет делать многое из того, что окну совершенно не нужно. Например, форма должна взаимодействовать с окнами *Delphi* - такими, как *Инспектор объектов, Редактор кода, Менеджер проекта*, а также с *Панелями инструментов* и *Палитрой компонентов*. На поверхности формы можно увидеть разметочную сетку для выравнивания компонентов, которая абсолютно ни к чему работающему приложению. Зато некоторые свойства формы, хотя и задаются в *Инспекторе объектов*, появляются только в окне приложения (отсутствие заголовка или кнопок на нём, например). Дальше мы часто не будем делать различия между формой и окном, но вы всегда должны иметь в виду их особенности (Рис. Ф3.1).

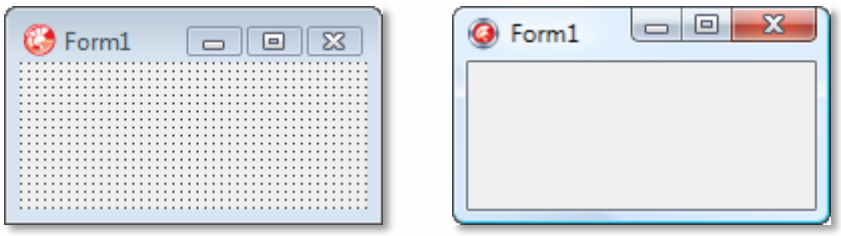
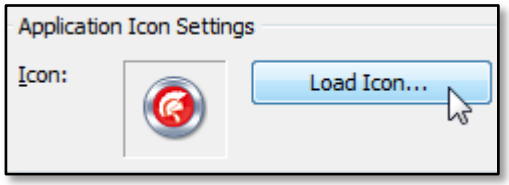


Рис. Ф3.1. Форма Окно

В верхней части окна находится полоска, которую обычно называют **заголовком** (*Title bar, Caption*). Он нужен для перемещения окна мышкой, а также на нём расположены:

- **Значок формы** – по умолчанию это стандартный значок *Delphi*, но вы можете заменить его своим, установив свойство **Icon** формы (просто загрузите нужный файл в формате *ICO* с диска). При запуске приложения в *Панели задач Windows* создаётся кнопка со *значком программы*, первоначально это стандартный значок приложений *Delphi* 

, изменить его можно, вызвав диалоговое окно командой *Главного меню* **Project > Options...** и на вкладке **Application** загрузив нужный значок (используйте кнопку **Load Icon...**) 

.

- **Заголовок формы** – по умолчанию там выводится название (значение свойства **Name**) формы. Чтобы его изменить, введите нужный

текст в окне свойства **Caption**. Обычно в заголовке главной формы помещают название программы, сведения о версии продукта и другую важную информацию, а в заголовке диалоговых окон – их назначение. В *Панели задач* при запуске приложения рядом со значком приложения появится и его название (по умолчанию это **Form1**).

- **Кнопка сворачивания окна (*минимизации*)**  – окно сворачивается до размера кнопки на *Панели задач*, которая из нажатого положения (активное окно) переходит в отжатое. Чтобы активизировать окно, достаточно щёлкнуть по его кнопке на *Панели задач*.
- **Кнопка разворачивания окна (*максимизации*)**  **–** окно раскрывается во весь экран (за исключением *Панели задач*, если не изменены её свойства), а вместо этой кнопки появляется **кнопка восстановления окна** , при нажатии на которую окно принимает прежние размеры.
- **Кнопка закрытия окна**  – окно исчезает с экрана. Если это главное окно приложения, то работа с ним завершается.

Граница окна (***Sizing Border***) – весь контур окна позволяет изменять его размеры. Для этого нужно поставить курсор мыши на границу окна и, когда курсор примет вид двойной стрелочки , потянуть её в нужную сторону.

Полоса главного меню окна – располагается сразу под заголовком окна, если оно имеет главное меню. На ней расположены названия разделов меню, при нажатии на которые открывается список пунктов меню (Рис. Ф3.2).

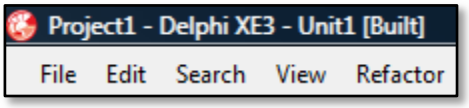


Рис. Ф3.2. Главное меню программы

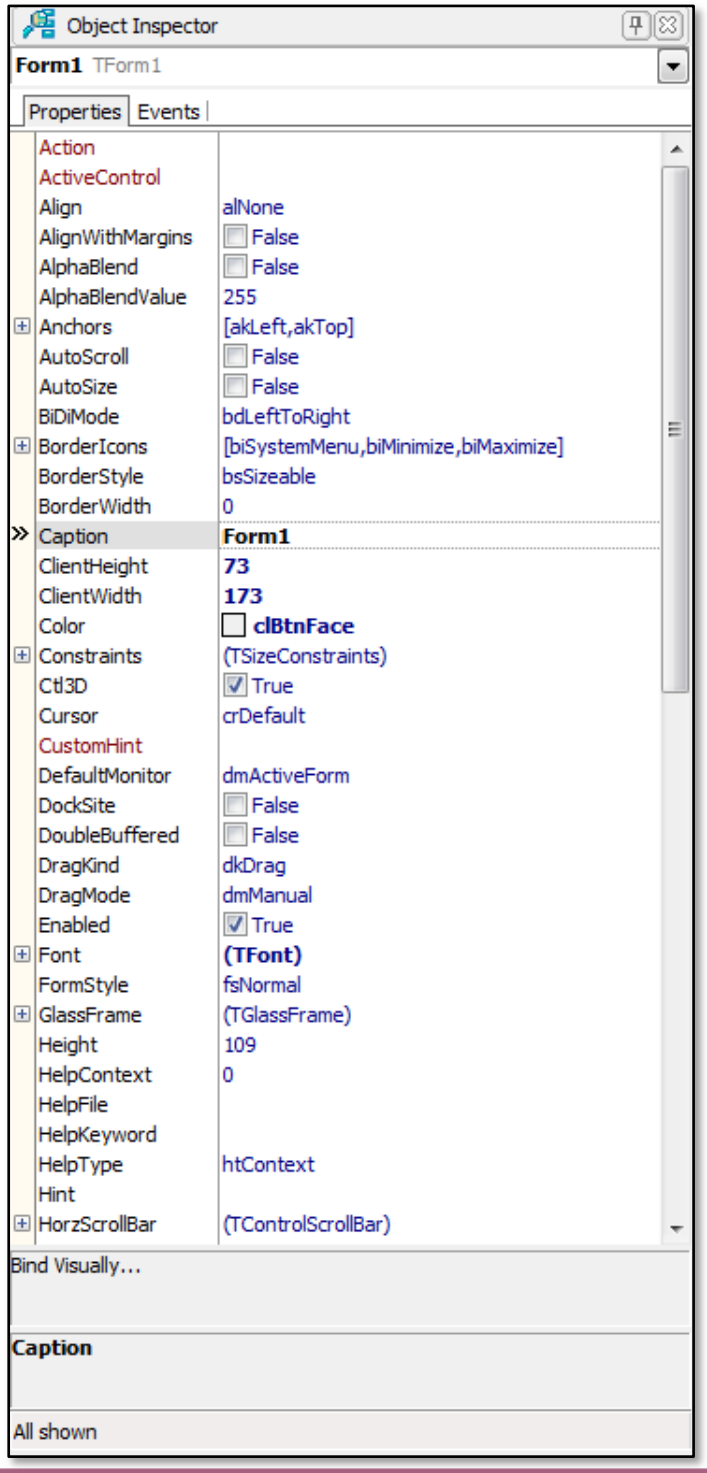
Клиентская область – всё остальное пространство окна. Именно здесь располагаются все компоненты формы, обеспечивающие интерфейс с пользователем.

Системное меню окна – появляется, когда вы нажимаете правую кнопку мыши на заголовке окна. Оно дублирует уже рассмотренные нами команды управления окном, но в него могут быть добавлены и другие пункты.

Свойства формы

Форма, в принципе, такой же компонент *Delphi*, как и все остальные. Однако в *Палитре компонентов* вы её не найдёте – роль формы в приложении настолько велика, что для добавления форм к проекту предназначены специальные команды *Главного меню* и *Панели инструментов* (мы их уже много раз рассматривали). Наиболее «популярные» формы можно отыскать (и сохранить) в *Репозитории*.

Как и другие компоненты, форма имеет множество свойств, большинство из которых устанавливается в *Инспекторе объектов* (то же самое можно сделать и программно, но это менее удобно и результаты будут видны только после запуска приложения), и лишь некоторые (например, свойство *Canvas*) доступны только в работающей программе.



Окно *Инспектора объектов*, если оно отсутствует на экране, можно открыть командой *Главного меню* **View > Object Inspector** или клавишей **F11**. На странице **Properties** в алфавитном порядке перечислены все (если вы не изменили установки *Инспектора объектов*) доступные во время проектирования свойства формы (Рис. Ф3.3).

Рис. Ф3.3. Свойства формы

Начиная с пятой версии *Delphi*, можно изменить формат перечисления свойств в *Инспекторе объектов.* Для вывода свойств не в алфавитном порядке, где разные по назначению свойства перемешаны между собой, а по **категориям** нужно включить флажок **by Category** в контекстном меню *Инспектора объектов* (Рис. Ф3.4).

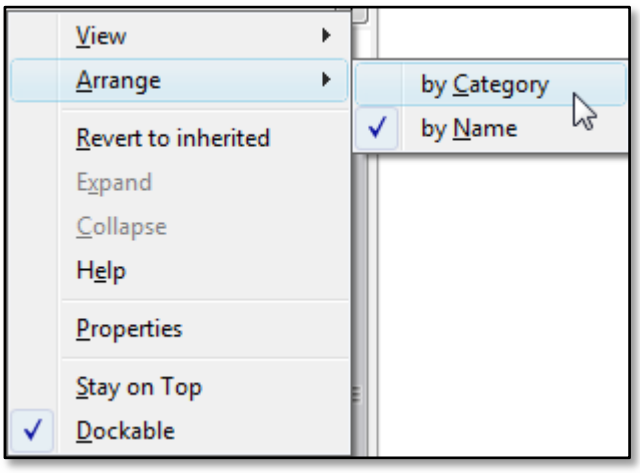
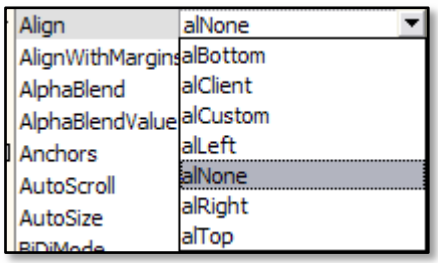


Рис. Ф3.4. Аранжировка свойств

Чтобы вернуться к алфавитному порядку, включите флажок **by Name**.

Рассмотрим **наиболее важные свойства формы** (подчёркнуты названия тех свойств, значения которых можно устанавливать в *Инспекторе объектов*):

Action	– определяет действие, связанное с формой.
ActiveControl	– указывает на оконный элемент управления, который имеет фокус ввода. Если значение равно *Nil*, то фокус имеет сама форма.
ActiveMDIChild	– указывает на активную дочернюю форму **MDI**-приложения.
AlphaBlend	– если флаг установлен, то прозрачность окна определяется значением свойства **AlphaBlendValue**.
AlphaBlendValue	= 0..255 – определяет прозрачность окна.
Align	– выравнивает окно по сторонам экрана. Значения свойства выбираются из выпадающего списка:  **alNone** – значение по умолчанию: выравнивание отсутствует. **alTop** – выравнивание по верхней стороне экрана. Форма переносится на

	верх экрана, ширина устанавливается по ширине экрана, а высота не изменяется. Это и другие свойства выравнивания имеют преимущества перед другими свойствами, изменяющими положение и размер формы. Поэтому если установлен этот вид выравнивания, вы сможете изменить только высоту формы, а передвинуть форму в другое место экрана или уменьшить её ширину вам не удастся. Для временного отключения действия свойства **Align** служит метод **DisableAlign**, а для повторного включения - метод **EnableAlign**. Эффект выравнивания проявляется только в работающей программе, в окне **Конструктора форм** вы никаких изменений не увидите. **alBottom** – выравнивание по нижней стороне экрана. **alLeft** - выравнивание по левой стороне экрана. **alRight** - выравнивание по правой стороне экрана. **alClient** – форма занимает всю рабочую область экрана.
Anchors	– фиксирует форму относительно сторон экрана. При изменении разрешения экрана расстояние между формой и «заякоренной» стороной остаётся неизменным.
ArrangeIcons	– упорядочивает значки свёрнутых дочерних окон **MDI**-приложения.
AutoScroll	– если значение свойства равно **True**, то полосы прокрутки на форме - при необходимости - появляются автоматически.
AutoSize	– автоматически подгоняет размеры формы под размещённые на ней компоненты. Установите значение этого свойства в **True**, если в процессе выполнения программы изменяются размеры компонентов (например, количество клеток в сетке).
BiDiMode	– задаёт направление вывода текста. По умолчанию свойство равно **bdLeftToRight** – текст выводится по-европейски, слева направо.
BorderIcons	– определяет набор кнопок в заголовке окна (результат будет виден только в работающей программе): **biSystemMenu –** форма имеет системное меню. Если установить это свойство в **False**, то исчезнут все кнопки из заголовка. **biMinimize** – сворачивает окно (минимизация). **biMaximize** – разворачивает окно (максимизация). **biHelp** - при нажатии этой кнопки курсор принимает форму вопросительного знака со стрелкой (**crHelp**). Подведите его к какому-либо элементу управления на форме и щёлкните мышкой – на экране появится информация о нём из файла помощи (если он подключен). Эта кнопка будет действовать, если **BorderStyle = bsDialog** или отсутствуют кнопки **biMinimize** и **biMaximize**. Кнопка **закрытия окна** всегда присутствует, если имеется заголовок и у формы есть системное меню.
BorderStyle	– определяет стиль обрамления окна: **bsDialog** – стандартное диалоговое окно, его размеры не изменяются, а в заголовке находится только кнопка закрытия окна. Обычно используются для получения информации от пользователя и вывода сообщений. Может исполнять и роль главного окна приложения, если вы не предполагаете

	изменять его размеры. **bsSingle** – как предыдущее окно, но набор кнопок в заголовке можно задать в **BorderIcons**. Почти все главные формы программ в этой книге имеют такой стиль. **bsNone** – как диалоговое окно, но без заголовка. Часто используется для вывода окон-заставок при загрузке программ. Смотрите дальше раздел «Форма-заставка». **bsSizeable** – стиль формы по умолчанию. Размеры формы можно изменять, а набор кнопок в заголовке установить через свойство **BorderIcons**. В этой книге практически не используется. **bsToolWindow** – как **bsSingle**, но с уменьшенной высотой заголовка. Используется для создания панелей инструментов. **bsSizeToolWin** - как **bsSizeable**, но с уменьшенной высотой заголовка. Также используется для создания панелей инструментов. (В приложениях **MDI** дочерние формы не будут иметь стили **bsDialog** и **bsNone**, даже если вы их установите.) Результат установки стиля окна будет виден только в работающей программе.
BorderWidth	– задаёт ширину бордюра по периметру клиентской области формы.
BoundsRect	– имеет тип **TRect** и описывает прямоугольник формы. Это свойство позволяет иначе обратиться к свойствам **Top**, **Left**, **Width**, **Height** и доступно только из программы (изменить размеры формы можно и с помощью метода **SetBounds** (ALeft, ATop, AWidth, AHeight: Integer);).
Brush	– определяет цвет и узор (pattern) для фона визуального элемента управления.
Canvas	– канва формы для вывода графики.
Caption	– заголовок формы
Cascade	– располагает дочерние формы **MDI**-приложения каскадом (со сдвигом вправо вниз).
ClientHandle	– описатель клиентского окна в **MDI**-приложении.
ClientHeight	– задаёт высоту клиентской области формы (без бордюра (**Border**) формы). Свойства клиентской области компонента обычно специально не задаются, вместо этого устанавливают нужные значения свойств **Height** и **Width**, а размеры клиентской области вычисляются автоматически.
ClientOrigin	– имеет тип **TPoint** и содержит координаты верхнего левого угла клиентской области формы. Свойство доступно только из программы.
ClientRect	– имеет тип **TRect** и описывает прямоугольник клиентской области. Свойство доступно только для чтения, из программы.
ClientWidth	– задаёт ширину клиентской области формы в пикселях.
Color	– устанавливает цвет рабочей области формы.
ComponentCount	– указывает число компонентов, владельцем которых является форма.
ComponentIndex	– индекс компонента в списке **Components** родителя. Отсчёт начинается с **0**.
Components	– индексированный список компонентов на форме-владельце.

ComponentState	– текущее состояние компонента. Это свойство используется средой разработки.
Constraints	– ограничивает изменение размеров формы при изменении разрешения экрана.
ControlCount	– содержит число элементов в списке **Controls**.
Controls	– содержит список неоконных и оконных дочерних элементов (компонентов) на форме.
ControlState	– показывает текущее состояние формы во время выполнения программы. Может принимать значения: **csLButtonDown** – левая кнопка мыши нажата и ещё не отпущена. Используется в событии **OnMouseDown**. **csClicked** – как **csLButtonDown**, но только если свойство **ControlStyle** содержит **csClickEvents**. Используется для определения события **OnClick**. **csPalette** – изменяется системная палитра. **csReadingState** – чтение значений свойств из потока. **csAlignmentNeeded** – осуществляется выравнивание. **csFocusing** – форма получает фокус управления (становится активной). **csCreating** – создаётся форма или компоненты на ней. **csCustomPaint** – форма отрисовывается нестандартным способом. **csDestroyingHandle** - уничтожается указатель на форму. **csDocking** – изменяется положения формы.
ControlStyle	– стиль формы.
Ctl3D	– определяет вид любого визуального компонента. При значении **True** (по умолчанию) он имеет объёмный вид, при **False** – плоский.
Cursor	– задаёт вид курсора на форме.
DefaultMonitor	– определяет, на каком из мониторов в многомониторной системе появится форма.
DockClientCount	– содержит число элементов управления на доке.
DockClients	– содержит список элементов управления на доке.
DockManager	– **COM**-интерфейс при операции **Drag-and-Dock**, позволяющий расширить возможности дока.
DockOrientation	– определяет, как будет размещаться переносимый объект на доке относительно других объектов.
DockSite	– если значение этого свойства равно **True**, то компонент может быть доком (то есть принимать компоненты, переносимые операцией **Drag-and-Dock**).
DragKind	– определяет, какая из операций будет выполняться при перетаскивании элемента управления: **dkDrag** – drag-and-drop. **dkDock** – drag-and-dock.

DragMode	– определяет способ включения механизмов перетаскивания **drag-and-drop** и **drag-and-dock**: **dmAutomatic** – включаются автоматически, когда пользователь нажимает кнопку мыши на элементе управления и начинает перетаскивание. **dmManual** – вызов метода **BeginDrag** должен осуществить программист. Это значение свойства по умолчанию.
DropTarget	– если значение равно **True**, то форма может быть приёмником объектов при их перетаскивании методом **drag-and-drop**.
Enabled	– при значении **True** форма работоспособна, в противном случае совершенно «неуправляема», как бы зависает.
Floating	– значение свойства равно **True**, если форма не перенесена на другую форму.
FloatingDockSiteClass	– определяет класс временного окна, создаваемого при переносе компонента.
Font	– задаёт шрифт, который может быть использован в других компонентах на форме.
FormState	– определяет состояние формы в данный момент.
FormStyle	– определяет разновидность окна: **fsNormal** – обычное окно. **fsMDIChild** – дочернее окно приложения **MDI**. **fsMDIForm** – главное окно приложения **MDI**. **fsStayOnTop** - окно всегда остаётся поверх других окон приложения, даже в неактивном состоянии.
Handle	– дескриптор окна, с помощью которого можно вызывать функции **Windows API**, что расширяет возможности **Delphi**.
Height	– задаёт высоту формы в пикселях.
HelpContext	– индекс, указывающий на раздел в файле справочной системы. Если компонент имеет фокус, то при нажатии клавиши **F1** на экран выводится информация, связанная с заданным контекстом. Если значение этого свойства равно **0**, то будет выведена (если она имеется) информация о родительском компоненте.
HelpFile	– имя файла помощи, который появляется на экране при нажатии кнопки **F1**.
Hint	– задаёт подсказку, появляющуюся при остановке курсора на форме.
HorzScrollBar	– устанавливает горизонтальную полосу прокрутки содержимого клиентской области формы. Автоматически появляется, если не все элементы управления видны при установленных размерах формы (значение свойства **AutoScroll** должно быть равно **True**). Можно сделать полосу прокрутки невидимой, если установить её свойство **Visible** в **False.**
HostDockSite	– указывает на док элемента управления.
Icon	– определяет значок окна.
KeyPreview	– если значение этого свойства равно **True**, то форма сначала перехватывает ввод с клавиатуры, адресованный элементу управления, имеющему фокус ввода, а затем пересылает сообщение этому компоненту.
Left	– задаёт **X**-координату верхнего левого угла формы в системе координат экрана. Начало отсчёта находится в верхнем левом углу экрана, а оси направлены вниз и вправо. Все координаты выражаются в пикселях.
MDIChildCount	– содержит количество дочерних форм в **MDI**-приложении.
MDIChildren	(property MDIChildren[I: Integer]: Tform;) – указывает на I-тую дочернюю форму **MDI**-приложения.
Menu	– задаёт главное меню окна.

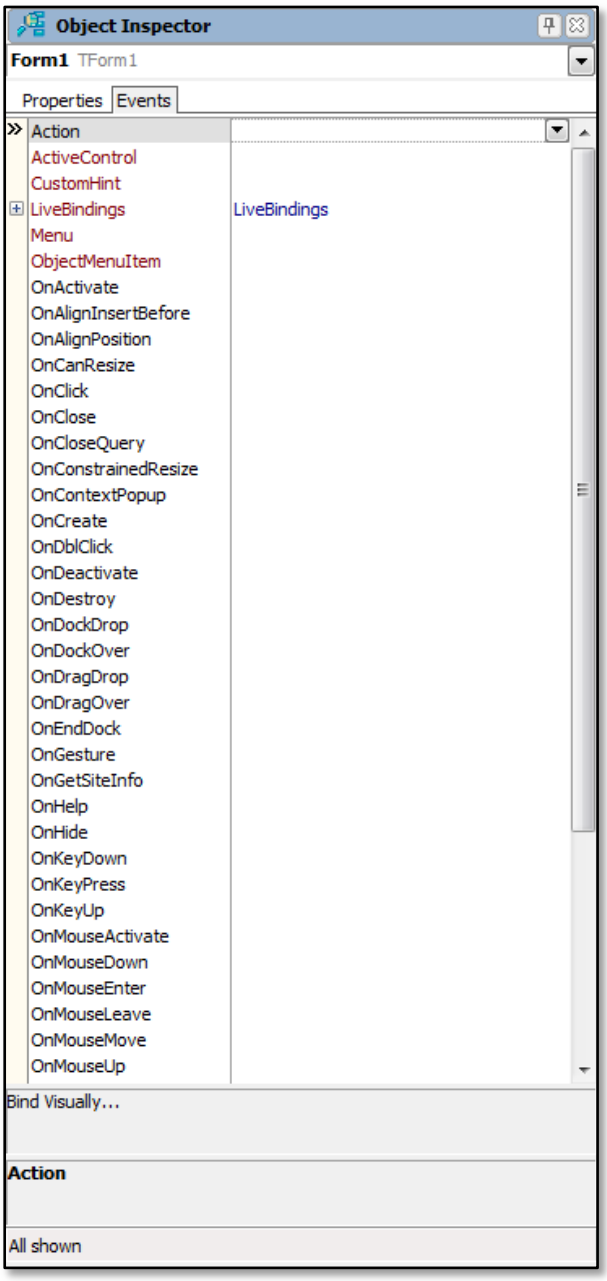
ModalResult	– значение, возвращаемое модальной формой при её закрытии. Может принимать значения: **mrNone** (0) – форма не закрыта, значение по умолчанию. **mrOk** (idOK) – выход с нажатием кнопки **OK**. **mrCancel** (idCancel) – выход с нажатием кнопки **CANCEL**. **mrAbort** (idAbort) - выход с нажатием кнопки **ABORT**. **mrRetry** (idRetry) - выход с нажатием кнопки **RETRY**. **mrIgnore** (idIgnore) – выход с нажатием кнопки **IGNORE**. **mrYes** (idYes) – выход с нажатием кнопки **YES**. **mrNo** (idNo) – выход с нажатием кнопки **NO**. Это свойство имеется и у кнопок **Button** и **BitBtn**, которые могут передавать его значение свойству формы. Если в программе вызывается метод **Close** для модальной формы, то свойство **ModalResult** получает значение **mrCancel**.
Monitor	– содержит характеристики текущего монитора.
Name	– устанавливает имя формы.
Next	– передаёт фокус ввода следующей в списке дочерней форме **MDI**-приложения.
Owner	– указывает на элемент-владелец компонента. Обычно владельцем является форма, на которой расположен компонент. Владелец компонента отвечает за его уничтожение при уничтожении самого владельца.
ParentFont	– если значение свойства равно **True**, то используется шрифт родительской формы. По умолчанию равно **False**.
PixelsPerInch	– задаёт разрешение экрана в точках на дюйм. Если свойство **Scaled = True**, то значение свойства **PixelsPerInch** считывается из текущих настроек **Windows**. Для стандартных шрифтов **PixelsPerInch** равно 96, для больших – 120. Если установить **PixelsPerInch** в **0**, то масштабирования формы не происходит (см. свойство **Scaled**).
PopupMenu	– задаёт контекстное (всплывающее) меню формы.
Position	**–** определяет положение и размер формы на экране. Может принимать значения: **poDesigned** – положение и размеры формы те же, что и в **окне Конструктора формы** (если оно откреплено**)**. Значение по умолчанию. Имейте в виду, что при переходе на меньшее разрешение экрана форма может исчезнуть «за горизонтом». **poDefault** – положение и размеры формы определяет операционная система **Windows**. **poDefaultPosOnly** - **Windows** устанавливает только положение формы на экране, не изменяя размеры. **poDefaultSizeOnly – Windows** устанавливает только размеры формы на экране, не изменяя её положения. **poScreenCenter** – форма помещается в центр экрана. Если в системе несколько мониторов, то форма будет видна на мониторе по умолчанию. **poDesktopCenter** – форма помещается в центре рабочего стола. **poMainFormCenter** – форма помещается в центре главной формы прило-

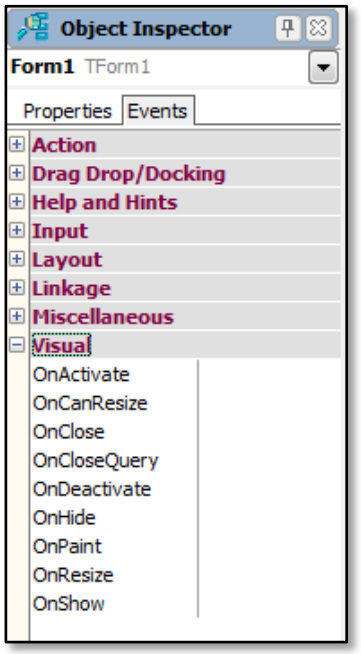
	жения. **poOwnerFormCenter** – форма помещается в центре формы-владельца.
Previous	– активизирует (передаёт фокус ввода) предыдущей дочерней форме **MDI**-приложения (если активной, например, была вторая форма, то фокус передаётся первой).
PrintScale	– масштабирует изображения формы при печати на принтере. Свойство может иметь такие значения: **poNone** – печать без масштабирования (точка в точку). **poProportional** – изображение формы на бумаге будет таким же, как на экране (получите то, что видите – WYSIWYG). **poPrintToFit** - форма будет напечатана так, как она выглядит на экране, но её размеры будут подогнаны к размеру листа.
Scaled	– если это свойства равно **True**, то при создании формы значение свойства **PixelsPerInch** считывается из текущих настроек системы **Windows**. Если это значение при старте программы отличается от значения свойства **PixelsPerInch** при конструировании формы, то форма и компоненты соответствующим образом масштабируются. Коэффициент масштабирования равен отношению разрешения экрана при работе программы к разрешению экрана при её разработке. Если **Scaled = False**, то масштабирования не происходит.
ShowHint	– если значение этого свойства равно **True**, то подсказка (**hint**) для формы будет появляться при остановке курсора на форме.
Showing	– если значение свойства равно **True**, то форма видна на экране. Свойство только для чтения!
TabOrder	– определяет порядок передачи фокуса между элементами управления на форме при нажатии клавиши **Tab**. Сначала фокус устанавливается на элементе с минимальным значением этого свойства, затем – по возрастанию значения.
TabStop	– если значение равно **False,** то компонент не принимает фокус управления при нажатии клавиши **Tab**.
Tag	– свойство, в котором можно сохранять целое число типа **Longint**. Позволяет определить элемент управления, который вызвал тот или иной обработчик.
Tile	– располагает дочерние окна **MDI**-приложения мозаикой, стиль которой определяется свойством **TileMode**.
TileMode	– определяет стиль расположения дочерних окон **MDI-**приложения: **tbHorizontal** – дочерние формы вписываются по ширине родительской формы. **tbVertical** – то же, но по высоте.
Top	– задаёт **Y**-координату верхнего левого угла формы.
TransparentColor	– если значение этого свойства равно **TRUE**, то цвет **TransparentColorValue** будет прозрачным.
TransparentColorValue	– задаёт прозрачный цвет формы.
UndockHeight	– содержит высоту объекта, переносимого с дока.
UndockWidth	– содержит ширину объекта, переносимого с дока.
UseDockManager	– если значение равно **True**, то используется **COM**-интерфейс при операции **drag-and-dock**.
VertScrollBar	– устанавливает вертикальную полосу прокрутки, см. **HorzScrollBar**.
Visible	– если значение равно **True**, то форма видна на экране, если **False** – форма невидима.

VisibleDockClientCount	– указывает число видимых объектов на доке.
Width	– устанавливает ширину формы в пикселях.
WindowMenu	– указывает на пункт меню главной формы **MDI**-приложения, к которому дописываются имена дочерних форм.
WindowState	– устанавливает состояние, в котором будет находиться окно при запуске приложения: **wsNormal** – нормальное. **wsMinimized** - свёрнутое. **wsMaximized** – развёрнутое.

События формы

События – это те же свойства, но процедурного типа. Возникают они в результате действий пользователя программы или элементов управления. Вы можете использовать возникновение любого события в своей программе. Для этого в процедуре-обработчике соответствующего события поместите необходимый код. Заготовка такой процедуры появляется в *Редакторе кода*, если дважды щёлкнуть на окошке ввода справа от названия события. Вы также можете назначить любой другой обработчик из уже имеющихся, выбрав его из списка, который открывается после щелчка на кнопке со стрелкой (в конце окна ввода).





Все возможные для текущего компонента события расположены на странице **Events** в *Инспекторе объектов* (Рис. Ф3.5).

Рис. Ф3.5. События формы

Как и страница *Properties*, она может быть представлена в двух разных видах – как простой **алфавитный** список (Рис. Ф3.5, слева) и с разбивкой по **категориям** (что удобнее, если у компонента длинный список событий) (Рис. Ф3.5, справа).

Дальше вы найдёте краткое **описание наиболее важных** для нас **событий формы**.

OnActivate	– форма получает фокус ввода сообщений, то есть становится активной. При активизации формы **Windows** посылает ей сообщение **WM_SIZE**. Событие **OnActivate** следует за **OnPaint**.
OnCanResize	(property OnCanResize: TCanResizeEvent;) – изменение размеров формы. type **TCanResizeEvent** = procedure(Sender: TObject; var NewWidth, NewHeight: Integer; var Resize: Boolean) of object; **Sender** – объект, размеры которого изменяются. **NewWidth** – возвращает новую ширину объекта. **NewHeight** – возвращает новую высоту объекта. **Resize** – возвращает **True**, если можно изменить размеры, и **False** в противном случае.
OnClick	– щелчок на форме.
OnClose	(property OnClose: TCloseEvent;) – обработчик этого события вызывается методом **Close** после **CloseQuery** **TCloseEvent** = procedure(Sender: TObject; var Action: TCloseAction) of object; Параметр **Action**: TCloseAction может принимать несколько значений, в зависимости от которых можно по-разному закрывать форму: **caNone** – не закрывать форму. **caHide** – форму не закрывать, а только сделать невидимой. Приложение имеет доступ к ней. Значение по умолчанию (для дочерних форм **MDI** – **caMinimize** или **caNone,** в зависимости от возможности их закрытия). **caFree** – закрыть форму и выгрузить из памяти компьютера. **caMinimize** – свернуть форму в значок на **Панели задач**.
OnCloseQuery	(property OnCloseQuery: TCloseQueryEvent;) – в обработчике этого события можно проверить, сохранены ли нужные значения на диске, действительно ли пользователь хочет выйти из программы и тому подобное type **TCloseQueryEvent** = procedure(Sender: TObject; var CanClose: Boolean) of object; Если параметр **CanClose** равен **True** (значение по умолчанию), то форму можно закрыть. В **MDI**-приложениях сначала должны быть закрыты все дочерние формы
OnConstrainedResize	– задать предельные размеры формы при их изменении.
OnContextPopup	– это событие возникает перед тем, как контекстное меню будет выведено на экран.
OnCreate	– процедура-обработчик, связанная с этим событием, вызывается при получении программой сообщения **WM_CREATE**, которое **Windows** посылает перед созданием каждого нового окна приложения. В этой процедуре удобно выполнять инициализацию переменных, массивов, установку свойств компо-

	нентов и другие действия.
OnDblClick	– двойной щелчок на форме.
OnDeactivate	– форма теряет фокус ввода.
OnDestroy	– событие возникает при уничтожении формы. Обработчик этого события вызывается в деструкторе (**Destroy**) формы и может использоваться для сохранения настроек и другой информации на диске.
OnDockDrop	– компонент отпущен над доком.
OnDockOver	– перемещение компонента над доком.
OnDragDrop	(property OnDragDrop: TDragDropEvent;) – событие возникает при отпускании левой кнопки мыши над элементом управления, который может стать приёмником для перетаскиваемого элемента (доком). Type **TDragDropEvent** = procedure(Sender, Source: TObject; X, Y: Integer) of object;
OnDragOver	(property OnDragOver: TDragOverEvent;) – событие, которое возникает при перетаскивании одного элемента управления над другим. Type **TDragOverEvent** = procedure(Sender, Source: Tobject; X, Y: Integer; State: TdragState; var Accept: Boolean) of object; Здесь **Sender** – элемент, над которым находится перетаскиваемый элемент **Source**. **X, Y** – координаты курсора в клиентской области элемента **Sender**. **State** – одно из трёх состояний перетаскивания: **dsDragEnter** - курсор мыши заходит на элемент управления. **dsDragLeave** - курсор мыши ушёл с элемента управления (или отпущена кнопка). **dsDragMove** - курсор мыши перемещается над элементом управления. **Accept** – если значение параметра равно **True**, то элемент управления **Sender** может принять **Source**.
OnEndDock	– завершение операции перетаскивания **Drag-and-Dock** элемента управления.
OnEndDrag	– завершение операции **Drag-and- Drop**.
OnGetSiteInfo	(property OnGetSiteInfo: TgetSiteInfoEvent;) – пользователь нажал кнопку мыши на переносимом (dock) компоненте и начал его сдвигать. Type **TGetSiteInfoEvent** = procedure(Sender: TObject; DockClient: TControl; var InfluenceRect: TRect; MousePos: TPoint; var CanDock: Boolean) of object; **DockClient** – переносимый объект, **MousePos** – координаты мыши, **Sender** – док, **InfluenceRect** – прямоугольная область дока, на которой может разместиться **DockClient**, **CanDock** – равен **True**, если док может принять **DockClient**.
OnHide	- форма становится невидимой.
OnKeyDown	- нажата клавиша на клавиатуре.
OnKeyPress	- клавиша нажата и отпущена.

OnKeyUp	- клавиша отпущена.
OnMouseDown	- нажата кнопка мыши на компоненте.
OnMouseMove	- мышка перемещается по компоненту.
OnMouseUp	- отпущена кнопка мыши на компоненте.
OnMouseWheel **OnMouseWheelDown** **OnMouseWheelUp**	- события для колёсика мыши.
OnPaint	- отрисовка формы после того, как она стала видимой. Обработчик этого события вызывается перед рисованием компонентов на форме (и после изменения размеров формы), поэтому нарисованная в этом обработчике картинка может служить для них фоном.
OnResize	- сигнализирует о завершении масштабирования формы. Обработчик этого события может быть использован для изменения размеров формы и компонентов на ней.
OnShortCut	- событие возникает, когда пользователь нажимает клавишу. Это событие происходит раньше, чем **OnKeyDown**.
OnShow	- событие возникает при появлении формы на экране.
OnStartDock	- событие возникает непосредственно перед посадкой компонента на док.
OnUndock	– уход компонента с дока и его размещение в другом месте.

Методы формы

Шура, это же не наш метод!

Обращение Верзилы Феди
к карающему его Шурику

Методы – это процедуры и функции, описанные внутри класса (в данном случае класса формы **TForm**) и предназначенные для операций над его полями (свойствами). В отличие от обычных процедур и функций им передаётся указатель на тот объект, который их вызвал.

Чаще других в программах используются следующие **методы формы**.

Activate	- переводит окно в активное состояние, вызывая обработчик события **OnActivate.**
AfterConstruction	– вызывается сразу после создания формы.
Assign	(procedure Assign(Source : TPersistent)) – копирует содержимое объекта **Source** в объект, вызвавший этот метод (**Self**).
BeforeDestruction	– вызывается непосредственно перед уничтожением формы.
BeginDrag	(procedure BeginDrag(Immediate: Boolean; Threshold: Integer = -1);) - начало перетаскивания компонента.

	Метод вызывается при ручном управлении операцией **Drag-and-Drop**. Если параметр **Immediate** равен **True**, то перетаскивание начинается немедленно, если **False** – только после перемещения мыши на расстояние не менее величины, заданной в **Threshold** (в этом случае можно использовать щелчок на элементе управления и «по прямому назначению»).
BringToFront	- устанавливает компонент над другими (ближе к нам) компонентами в контейнере, заслоняя их при перекрывании.
Broadcast	- рассылает всем дочерним элементам из списка **Controls** сообщение **Message**.
CanFocus	- возвращает **True**, если форма может получить фокус ввода (для этого она и все её родители должны быть активны (**Enabled = True**) и видимы (**Visible = True**)).
ClassName	- возвращает строку, содержащую название типа компонента (**TForm** для формы).
ClientToScreen	(function ClientToScreen(const Point: TPoint): TPoint;) - преобразует координаты точки в клиентской области компонента к координатам экрана.
Close	– закрывает форму.
CloseQuery	(function CloseQuery: Boolean;) - этот метод вызывается перед методом **Close**. Форма будет закрыта, когда он вернёт значение **True**. Для проверки возможности закрытия формы вызывается обработчик события **OnCloseQuery**.
ContainsControl	(function ContainsControl(Control: TControl): Boolean;) - возвращает **True**, если в списке **Controls** есть элементы.
ControlAtPos	(function ControlAtPos(const Pos: TPoint; AllowDisabled: Boolean, AllowWinControls:Boolean = False): TControl;) – эта функция отыскивает в списке **Controls** элемент, которому принадлежит заданная точка **Pos**. Если флаг **AllowWinControls** установлен, то поиск ведётся и среди пассивных (их свойство **Enabled=False**) элементов.
Create	- создаёт и инициализирует новую форму.
CreateNew	- создаёт и инициализирует новую форму, не используя файл **.DFM**.
DefocusControl	(procedure DefocusControl(Control: TWinControl; Removing: Boolean);) - забирает фокус у визуального компонента на форме. Если параметр **Removing** равен **True**, то фокус передаётся элементу-родителю.
Destroy	– удаляет форму из памяти. Вызывается методом **Free**.
DestroyComponents	– уничтожает все компоненты, подчинённые данному.
DisableAlign	– используется для временного отключения выравнивания компонентов.
DisableAutoRange	– выключает автоматическое прокручивание.
Dispatch	– ищет динамический метод-обработчик по индексу сообщения **Windows**.
DockDrop	– метод вызывает обработчик события **OnDockDrop**.
DragDrop	– метод вызывает обработчик события **OnDragDrop**.
Dragging	(function Dragging: Boolean;) - возвращает **True**, если элемент управления в настоящее время перетаскивается.
DrawTextBiDiModeFlags	- возвращает значение флагов для использования в функции **DrawText**.
EnableAlign	- включает выравнивание после его отключения методом **DisableAlign**.
EnableAutoRange	– включает автоматическое прокручивание.
EndDrag	– метод вызывает событие **OnDragDrop** (если возможен приём элемента управления) и **OnEndDrag**.

ExecuteAction	– вызывается автоматически при необходимости выполнить действие, назначенное для этого компонента.
FindChildControl	(function FindChildControl(const ControlName: string): TControl;) - отыскивает дочерний элемент по его имени.
FindComponent	(function FindComponent(const AName: string): TComponent) – отыскивает компонент с именем **AName**. Позволяет обращаться к компонентам одного типа как к массиву.
FocusControl	– передаёт фокус управления визуальному компоненту на форме.
Focused	(function Focused: Boolean;) – возвращает **True**, если форма имеет в данный момент фокус ввода (может получать и обрабатывать входные сообщения).
Free	– уничтожает форму и, если необходимо, выгружает её из памяти.
GetControlsAlignment	– возвращает установленное значение выравнивания (alignment).
GetFormImage	(function GetFormImage: TBitmap;) - возвращает растровую картинку с изображением формы.
GetNamePath	- возвращает имя объекта для использования в **Инспекторе обьектов**.
GetParentComponent	- возвращает **nil** для компонента типа **TComponent**.
Hide	- скрывает форму; метод устанавливает свойство **Visible** в **False**.
InsertComponent	(procedure InsertComponent (AComponent: TComponent)) - вставляет компонент **AComponent** в список **Components**.
InsertControl	- процедура, вставляет элемент в конец списка **Controls**.
Invalidate	- перерисовывает форму.
IsShortCut	- возвращает **True**, если нажатые клавиши являются клавиатурным сокращением.
ManualDock	- программный перенос (**Dock**) объекта на док.
ManualFloat	- программный перенос объекта с дока.
PaintTo	(procedure PaintTo(DC: HDC; X, Y: Integer);) – рисует оконный элемент управления на контексте устройства **DC**.
Perform	– передаёт сообщение **Windows** непосредственно в функцию окна формы.
Print	– распечатывает на принтере изображение формы, полученное с помощью метода **GetFormImage**.
Refresh	- перерисовывает компонент, вызывая метод **Repaint**.
Release	- уничтожает форму и выгружает её из памяти. Этот метод посылает окну сообщение **CM_RELEASE**.
RemoveComponent	(procedure RemoveComponent (AComponent: TComponent)) - удаляет компонент **AComponent** из списка **Components**.
RemoveControl	- удаляет элемент из списка **Controls**.
Repaint	- перерисовывает компонент, вызывая методы **Invalidate** и **Update**.
ReplaceDockedControl	- переносит объект с одного дока на другой.
ScaleBy	(procedure ScaleBy(M, D: Integer);) - масштабирует компонент в отношении M/D.
ScreenToClient	– конвертирует экранные координаты точки в координаты клиентской области компонента.
ScrollBy	(procedure ScrollBy(DeltaX, DeltaY: Integer);) - прокручивает содержимое клиентской области формы на **DeltaX**, **DeltaY** пикселей по горизонтали и/или вертикали. С помощью этого метода можно прокручивать окно без

	полос прокрутки.
ScrollInView	(procedure ScrollInView(AControl: TControl)) - помещает элемент управления **AControl** в видимую часть клиентской области формы, при этом изменяется положение бегунков полос прокрутки. Этот метод автоматически вызывают элементы управления, находящиеся в невидимой части клиентской области формы, когда им передаётся фокус вода.
SendToBack	– устанавливает компонент позади (дальше от нас) других компонентов на контейнере.
SetBounds	(procedure SetBounds(ALeft, ATop, AWidth, AHeight: Integer);) - устанавливает положение и размеры формы.
SetFocus	– передаёт форме фокус ввода (она должна быть активной и видимой).
SetFocusedControl	– передаёт фокус управляющему элементу на форме.
Show	– показывает форму; действует аналогично **Visible=True**. Форма становится видимой и помещается поверх других окон приложения.
ShowModal	– показывает модальную форму. Модальными обычно являются диалоговые окна. При их выводе на экран они получают фокус ввода и не позволяют переключиться на другие формы, пока они не закрыты.
Update	– отрисовывает форму, не дожидаясь обработки других сообщений **Windows**.

Факультатив 4. Японский кроссворд, или Восхождение на Фудзияму мысли

Банзай!

Клич японских фанатов оригами

Японцы, конечно, молодцы – придумали такое количество отличных головоломок, что о них можно написать целое собрание сочинений! Впрочем, наши помыслы не устремляются так далеко, поэтому мы ограничимся только одной из них, но весьма интересной и полезной для ума. Самое удивительное в ней то, что эта забава, которую мы называем *Японским кроссвордом* (а то и сканвордом!) или *Японской головоломкой* не имеет никакого отношения к словесным задачам – в ней вообще нет никаких слов.



Сами японцы называют её *Edel*. Поскольку головоломка очень известна во все мире, то у нее, как и у закоренелого рецидивиста, имеются «клички» - *Нонограмма* и *Hanjie*.

Это и понятно: японцы пишут иероглифами, то есть целыми словами, из которых кроссворд составить довольно затруднительно, что, собственно, и послужило толчком для создания этой головоломки. Сначала в заданиях требовалось найти загаданный иероглиф, но потом стали использовать и обычные чёрно-белые картинки. В России, конечно, распространён именно последний вариант головоломки…

Просветление, или Что придумали японцы

-Вы неправильно коня поставили,
товарищ гроссмейстер,- залебезил одноглазый.
– Конь так не ходит.

Ильф и Петров,

Двенадцать стульев

Прежде чем перейти к самому интересному (а этим для нас, безусловно, является решение головоломок с помощью компьютера), нужно разобраться в сущности японского кроссворда.

Все подобные задачи имеют вот такой вид (Рис. Ф4.1).

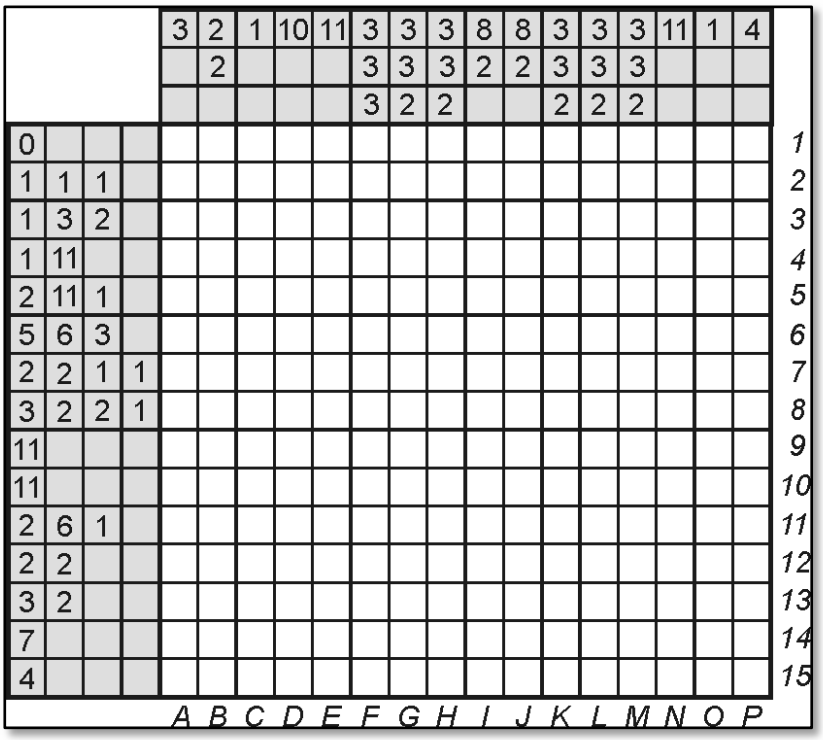


Рис. Ф4.1. Японский кроссворд

Большой белый прямоугольник, разделённый на квадратные клетки, - это *игровое поле*. Сначала оно пустое, то есть все клетки *белые*. Рисунка на нём, как на ещё не проявленной фотографии, не видно. А чтобы он «проявился», часть клеточек нужно *закрасить*. Любой непрерывный горизонтальный или вертикальный ряд чёрных клеток называется *группой*. Друг от друга группы отделяются по крайней мере одной *белой* клеткой (белые клетки могут находиться и по краям строки или столбца на игровом поле, то есть до первой и за последней группой).

Сверху и слева от игрового поля размещаются поля с числами (на нашем рисунке они выделены *серым* цветом). Мы будем называть их *числовыми полями*. Числа показывают длину каждой группы соответствующего ряда, поэтому в каждом ряду на игровом поле столько групп, сколько чисел в числовом поле. Например, на Рис. Ф4.2 ряд содержит 3 группы чёрных клеток – в левом числовом поле находится 3 числа. В первой группе 2 клетки, во второй – 6, в третьей – 1, то есть первое число слева (или сверху – для верхнего числового поля) задаёт длину первой группы, второе – второй и так далее. Положение групп в ряду может и отличаться от того, что на рисунке, - всё зависит от условия задачи.



Рис. Ф4.2. Ряд чисел и клеток



Надо заметить, что, не нарушая последовательности чисел, их можно записывать либо от левого (верхнего), либо от правого (нижнего) края числового поля. Чаще встречается второй вариант, что нередко приводит к ошибкам при решении задачи, так как начальные группы рядов располагаются на разных уровнях. Именно поэтому мы всегда будем придерживаться второго варианта.

Для наглядности дальнейших рассуждений все ряды игрового поля обозначены числами и буквами, как в шахматах (правда, у шахматистов принято нумеровать горизонтали снизу вверх, а нам удобнее делать это сверху вниз, что более естественно). В реальных задачах этот «элемент оформления» отсутствует.

Таким образом, для каждого ряда игрового поля нам известно: число групп в нём и длина каждой из групп. Дело за малым (этот малый, конечно, вы) – правильно «расставить» белые клетки. Не надейтесь, что вам удастся сделать это играючи, а лучше внимательно прочитайте правила решения японских кроссвордов и разберитесь в примере решения одного из них, не самого сложного, но и не самого простого. Даже если ваша рука и не тянется к перу, а перо – к бумаге, всё равно наши экзерсисы пригодятся вам при работе над грядущей программой.

Правила хорошего моветона

Лошадью ходи!

Один из дурацких Фединых советов

Начинать решение следует с тех рядов, которые сами в руки просятся. Думать придётся много, но это впереди.

Золотые правила решения японских кроссвордов

Правило 1: *Пустой ряд.* Перво-наперво поищите ряды (горизонтальные и вертикальные), в которых стоит только число *0*. Такая ситуация встречается нечасто, но нужно быть готовым ко всему. Так вот, это число неопровержимо свидетельствует о том, что *весь ряд состоит только из белых клеток* (в печатных изданиях все клетки изначально белые, поэтому «обелить» их не удастся, придётся прибегать ко всяческим уловкам, чтобы их отметить, - например, точками или крестиками). Найдя *0*, вы смело можете поставить на этом ряде крест (Рис. Ф4.3).

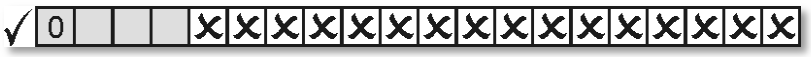


Рис. Ф4.3. Пустой ряд из белых клеток

Так как весь ряд разгадан, то слева от него мы ставим *галочку*, дабы он более не обременял нас размышлениями.

Правило 2: *Полный ряд*. Значительно чаще в задачах попадаются ряды, которые также описываются единственным числом, в точности совпадающим с длиной ряда на игровом поле. Тут, как говорится, без вариантов – весь ряд нужно «очернить». Действуем аналогично первому случаю, но в негативном исполнении (Рис. Ф4.4).



Рис. Ф4.4. Полный ряд черных клеток

Первые два правила просты и очевидны, как очевидно и то, что с их помощью можно решить только *Чёрный квадрат* Малевича или его бледного родственника. В японских кроссвордах такие шедевры не встречаются, поэтому перейдём к более сложным комбинациям клеток.

Правило 3: *Полная сумма*. Это третья – и последняя – возможность разом покончить с целым рядом. Так как между группами чёрных клеток должна находиться хотя бы одна белая, то все они займут как минимум столько клеток, сколько в сумме составляют все группы, плюс по одной клетке между ними (то есть число групп в ряду за вычетом единицы). Если мы найдём ряд с такой суммой чисел, то его участь предрешена – он будет беспощадно раскрашен (Рис. Ф4.5).



Рис. Ф4.5. Полная сумма

При этом нам потребуется уже известная сноровка, но это приятные хлопоты. Как и раньше, отмечаем ряд галочкой. Третьим правилом придётся пользоваться довольно часто, но и его маловато будет для решения даже самой простой задачи.



Ясно, что *Правило 2* является частным случаем более общего *Правила 3*. При ручном разгадывании кроссворда этот факт ничего не даёт, но при программировании он нам ещё послужит.

Правило 4: *Длинная группа*. Больше нам не удастся решить целиком весь ряд, придётся брать частями (подобный вид дележа чужих денег был не по вкусу Остапу Бендеру, он предпочитал всё сразу, но мы вынуждены идти на компромисс). Если в числовом поле стоит *единственная группа*, длина которой больше половины длины ряда на игровом поле, то, как бы ни располагалась эта группа, часть клеток в середине ряда всегда будет занята чёрными клетками. Эти клетки вычисляются очень просто – достаточно отсчитать от края игрового поля столько клеток, сколько должен занять этот ряд, и посмотреть, сколько клеток останется ещё не закрытыми с противоположного края. Так как разгаданные чёрные клетки должны располагаться симметрично относительно границ ряда, то столько же неразгаданных клеток должно быть и в начале ряда. Число этих клеток можно узнать, если из длины ряда вычесть длину группы. Мы пока не знаем, какого цвета будут граничные клетки, а вот остальные обязательно будут *чёрными* (Рис. Ф4.6).



Рис. Ф4.6. Длинная группа

Чтобы лучше понять, почему так происходит, нарисуем группу в *крайних* положениях (вы можете сделать то же самое и для всех промежуточных положений группы, но это ничего не изменит) (Рис. Ф4.7).

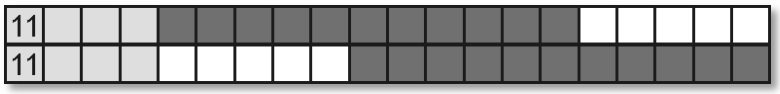


Рис. Ф4.7. Нахлёст

Отчётливо видно, что первые и последние 5 клеток ряда (16 – 11 = 5) могут быть как чёрными, так и белыми, поэтому мы не можем пока однозначно определить их цвет. А вот 6 клеток в центре группы (11 – 5 = 6) обязательно будут *чёрными*, как ни крути!

Это правило следует хорошенько запомнить, так как пользоваться им придётся очень часто.

Его действие можно распространить и на тот случай, когда в ряду не одна, а *две* группы. Вычтите из длины ряда длину одной из групп и ещё одну клетку для «зазора» между группами. Если длина второй группы больше половины полученной разности, то часть клеток второй группы обязательно будет *чёрной*. Например, если длина ряда 9 клеток и в нём две группы из 4 и 3 клеток, то даже в крайнем правом положении вторая группа «выбьет» из ряда 4 клетки. На долю первой группы останется 5 клеток,

и мы легко можем подсчитать, что 3 клетки ряда, начиная со второй, обязаны быть *чёрными* (Рис. Ф4.8).

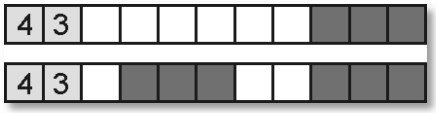


Рис. Ф4.8. Двойной нахлест

Кстати говоря, правила составления классических японских кроссвордов запрещают и пустые, и полные строки, и полные суммы, но у нас на это дело - по привычке - смотрят шире, чем в «узкоглазой» Японии, то есть у нас можно всё. И ещё о запретном: размеры игрового поля в классическом кроссворде должны быть кратны пяти. Но судьба этого запрета в России столь же печальна, как и предыдущих…

Железные правила решения японских кроссвордов

Первые два правила можно применить только однократно, в самом начале игры, третье и четвёртое могут пригодиться и дальше, если учитывать уже разгаданные клетки ряда. Все следующие правила нужно проверять после *каждого* изменения ситуации на игровом поле.

Правило 5: *Нахлёст*. Если в ряду осталась только *одна* неразгаданная группа, длина которой больше половины оставшихся неразгаданными клеток, то часть клеток в середине группы обязательно будет *чёрными*. Их количество и положение легко установить, пользуясь *Правилом 1* (Рис. Ф4.9).



Рис. Ф4.9. Нахлёст

Правило 6: *Габариты.* Важно отмечать не только заведомо чёрные, но и заведомо *белые* клетки. Проверяйте, какие клетки наверняка не могут быть чёрными, и, значит, они белые. Такое положение часто возникает, если в ряду уже отгаданы чёрные клетки и мы можем точно установить предельные клетки чёрной группы. На верхнем рисунке (Рис. Ф4.10) показана позиция на игровом поле, когда в ряду уже разгадана одна чёрная клетка. Если предположить, что она является начальной для чёрной группы, то, отсчитав влево 11 клеток, мы убедимся, что последние две клетки не могут входить в эту группу, поэтому они непременно *белые* (Рис. Ф4.10).

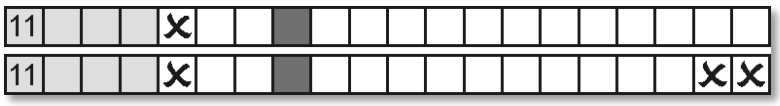


Рис. Ф4.10. Габариты

Правило 7: *Слитная группа*. Если в ряду осталась неразгаданной только *одна* группа и имеются *чёрные* клетки, разделённые неразгаданными, то все они принадлежат одной группе и все неразгаданные клетки между чёрными тоже *чёрные* (Рис. Ф4.11).

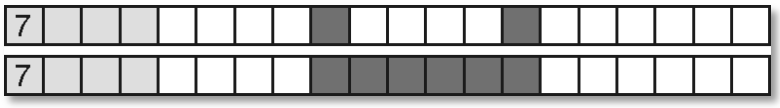


Рис. Ф4.11. Слитная группа

Правило 8: *Автогруппа*. Проверяйте, не образовались ли после закрашивания клеток в одних рядах полные группы в других. Если необходимо, обрамляйте их с двух сторон *белыми* клетками (Рис. Ф4.12).

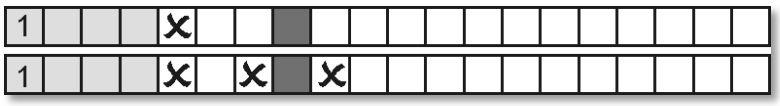


Рис. Ф4.12. Автогруппа

Правило 9: *Граничные клетки*. Если на поле есть *чёрная* клетка, граничащая с *белой*, то она является либо начальной, либо конечной клеткой одной из групп ряда (Рис. Ф4.13).



Рис. Ф4.13. Граничные клетки

Правило 10: *Вилка*. Если между двумя белыми клетками (или между белой клеткой и краем поля) не может разместиться ни одна группа, то все клетки между ними *белые* (Рис. Ф4.14).

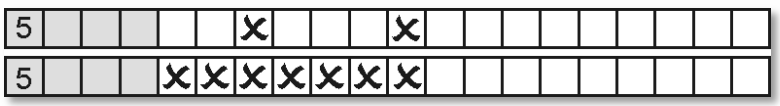


Рис. Ф4.14. Вилка

Теперь вы вполне сможете «правильно» решать задачи. Но имейте в виду, что невозможно описать все положения, могущие возникнуть на игровом поле, поэтому подходите к применению правил творчески. К примеру, многими из них можно воспользоваться и в более сложных случаях, чем это описано.

Решительный момент, или Берём быка за бока

Тореадор, смелее в бой!

Оперный призыв

Давайте решим для примера задачу, которую вы уже видели в начале главы.

1. Следуя нашим правилам, мы легко обнаруживаем **ряд 1**, в котором для нас заботливо поставлен *нолик*, и решаем его без всяких усилий (**Правило 1**).
2. К сожалению, на этом блага цивилизации заканчиваются, так как никто не позаботился воплотить **Правило 2** в жизнь. Зато – мир не без добрых людей! – **Правило 3** представлено во всей красе: сразу два ряда – **5** и **6** - подчиняются ему, так что мы с лёгким сердцем можем разом покончить с обоими.
3. В столбцах **N** и **E** притаились *длинные* ряды. Так как верхняя клетка уже известна (она *белая*), то группу из 11 клеток нужно разместить на 14 клетках. По **Правилу 5**, определяем, что 8 клеток, 5-12, *чёрные*.
4. **D** – опять *длинный* ряд. Верхняя клетка разгадана, и группа из 10 клеток должна разместиться на тех же 14 клетках. Если отсчитать 10 клеток сверху (от белой) и снизу (от края поля), то окажется, что 7 клеток, 5-11, в середине ряда *чёрные*. По **Правилу 6**, отсчитываем вниз от **D5** 10 клеток и выясняем, что последняя клетка ряда - **D15** - может быть только *белой*.

После всех манипуляций игровое поле должно выглядеть так (Рис. Ф4.15).

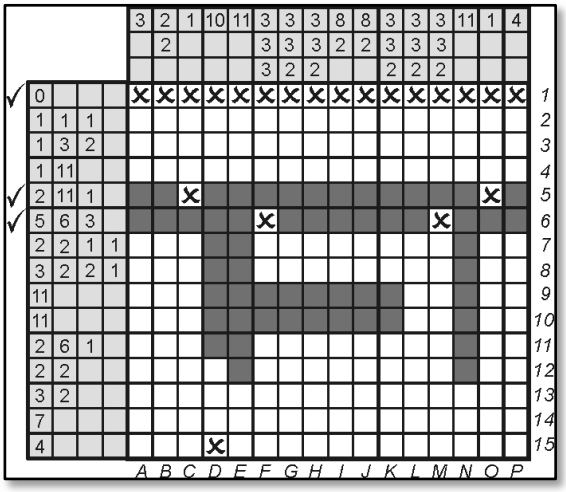


Рис. Ф4.15. Четыре шага позади

5. **Строки 9, 10**. Воспользуемся **Правилом 7** и закрасим клетки **L**, **M**. Так как теперь группы на поле состоят из 11 клеток, что равно заданной длине групп, то эти группы решены. Других групп в этих рядах нет, поэтому весь ряд решён. Оставшиеся клетки помечаем как *белые*.
6. **Столбцы F, M.** Клетка **5** может быть только *конечной* для первой группы из 3 клеток, поэтому можно закрасить клетки **3** и **4**. Группа разгадана, отмечаем её кружком. Клетка **2** *белая*.
7. **В столбце O** образовалась группа из 1 клетки, единственная в этом ряду, следовательно, весь ряд решён, отмечаем его галочкой. Оставшиеся клетки ряда – *белые*.
8. **Строка 4.** Согласно **Правилу 7**, клетки **F**, **M** принадлежат группе из 11 клеток, иначе эту группу невозможно разместить на поле. Все клетки между ними, **G-L**, *чёрные*. По **Правилу 10**, определяем, что клетка **P** – *белая*.
9. **Столбец P.** По **Правилу 9**, клетка **5** – *начальная* для группы, поэтому клетки **7** и **8** чёрные, а группа решена. Так как группа единственная в ряду, то решён и весь ряд. Отмечаем его галочкой, а оставшиеся клетки ряда помечаем как *белые*.

После этого хода на поле возникла такая забавная конфигурация клеток (Рис. Ф4.16).

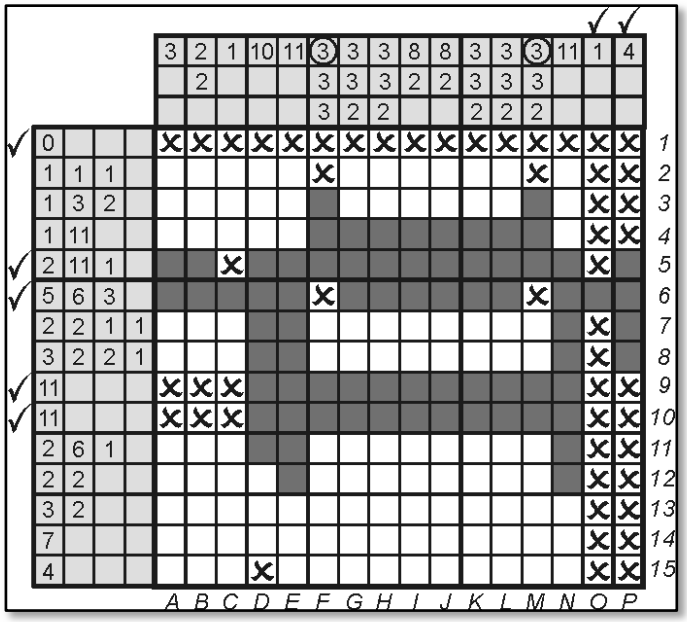


Рис. Ф4.16. И ещё пяток

10. **В строке 7** образовались две последние группы, отмечаем их кружком, а в клетку **M** ставим крестик - она *белая*.
11. **В строке 8** вырисовалась последняя группа, а также известна конечная клетка предпоследней группы, следовательно, клетка **M** *чёрная*, а **L** – *белая*. Отмечаем обе группы кружком.
12. **В столбцах G, H, K, L** образовались первые группы из 3 клеток, отмечаем их кружком и обрамляем *белыми* клетками **3** и **4**.
13. **В строке 11** образовалась последняя группа из 1 клетки. Отмечаем её кружком, а в клетку **M** ставим *крестик*.

14. **В столбцах I, J** все чёрные клетки входят в группу из 8 клеток, поэтому неразгаданные клетки **7** и **8** между ними можно закрасить.
15. **В строке 7** получились последние две группы - ряд решён. Оставшиеся неразгаданными клетки помечаем как *белые*.
16. **Столбец C** решился сам собой – ура, то есть банзай!!!
17. **В строке 8** возникла ещё одна группа из 2 клеток, а клетка **D** – *начальная* для группы из 3 клеток, поэтому с чистой совестью мы закрашиваем клетку **F**. Со всеми группами этого ряда мы успешно справились, помечаем его *галочкой*, а оставшиеся клетки *крестиком*.

Позиция, возникшая на «доске» после 17-ого хода (следите за мыслью!) (Рис. Ф4.17).

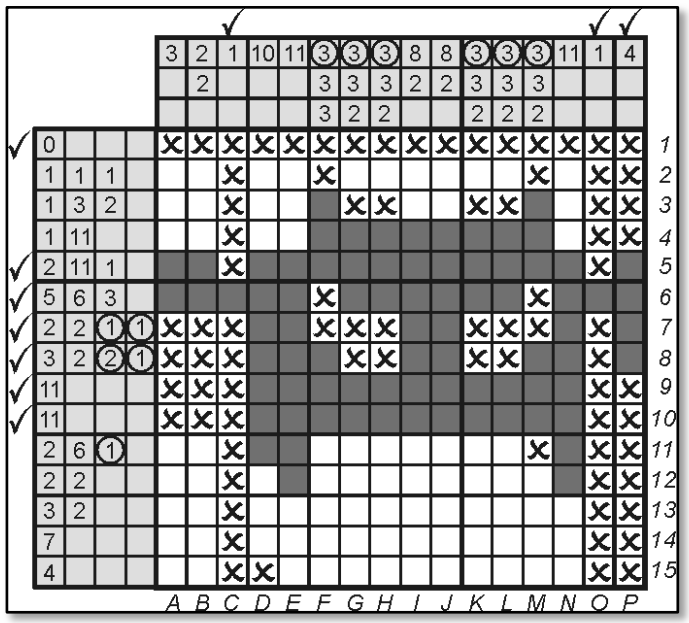


Рис. Ф4.17. И ещё чуток

18. **В столбцах F, M** образовались вторые группы из 3 клеток, отмечаем их *кружком*. В клетку **F11** ставим *крестик*.
19. **Столбцы G, H, K, L** – клетка **9** начинает вторую группу из 3 клеток, закрашиваем клетку **11** и ставим крестик в поле **12**. Отмечаем, что группа решена. Клетка **2** может быть только *белой*, так как первая группа уже разгадана.
20. **В строке 11** образовалась группа из 2 клеток, отмечаем её *кружком*. Чёрные клетки **F, G** и **K, L** могут принадлежать только группе из 6 клеток, помечаем клетки **I, J** как *чёрные*. Группа разгадана. Разгадан полностью и весь ряд, ставим слева *галочку*. Оставшиеся клетки **A, B**, разумеется, *белые*.
21. **В столбцах I, J** образовались группы из 8 клеток, обрамляем их *крестиками*. Клетка **2** тоже *белая*. Фиксируем свои успехи.
22. **В строке 4** между белыми клетками **C** и **O** ровно 11 клеток, поэтому группа из 11 клеток решена, клетки **D, E, N** *закрашиваем*.
23. **В строке 3** клетка **M** – *начальная* для группы из 2 клеток, поэтому клетка **N** – *чёрная*. Клетка **F** – *конечная* для группы из 3 клеток, значит, клетки **D, E** также *чёрные*. Отмечаем обе группы как решённые.
24. **Столбец A.** Клетка **6** – *конечная* для группы из 3 клеток, поэтому клетка **4** *чёрная*. Группа решена, а других в этом ряду и нет. Отмечаем его *галочкой*.

25. **В столбце B** получилась одна из групп из 2 клеток, следовательно, клетка **4** - *белая*.

Ситуация на поле складывается явно в нашу пользу (Рис. Ф4.18).

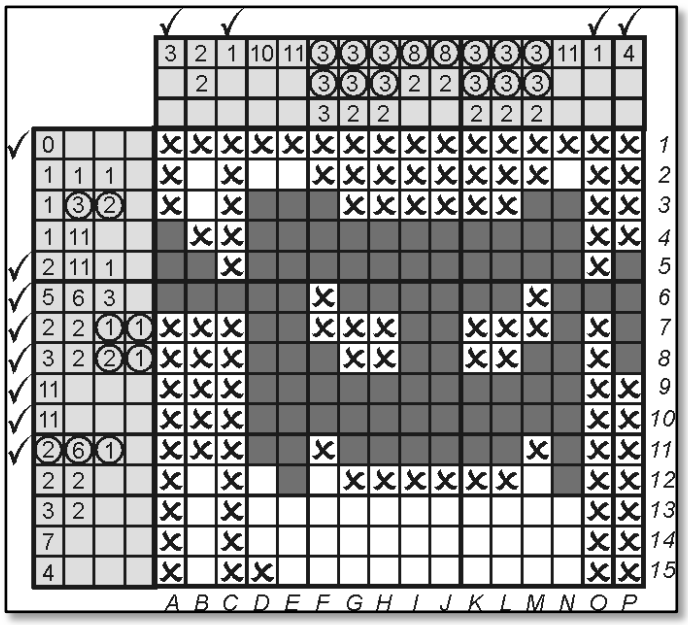


Рис. Ф4.18. Победа близка!

26. **В строке 4** образовалась первая группа, теперь все группы решены, а с ними и весь ряд.
27. **В строках 12, 13, 14, 15** между белыми клетками **A** и **C** не сможет разместиться ни одна группа, стало быть, клетка **B** – *белая*.
28. **В столбце B** остались только две не разгаданных клетки – как раз для первой группы. И с этим рядом покончено.
29. **В строке 2** образовалась первая группа (в клетке **B**). Последняя группа может занять только клетку **N**, так как на остальных клетках (**D, E**) не могут разместиться 2 группы.
30. **В столбце N** сама собой возникла единственная группа из 11 клеток, что позволяет нам считать клетки **13**, **14**, **15** *белыми*. И с этим рядом мы управились.
31. **В строке 12** клетка **N** - *конечная* для второй группы из 2 клеток, поэтому предыдущая клетка также *чёрная*.
32. **В столбце M** клетка *12* - начальная для группы из 2 клеток, следовательно, клетка **13** также *чёрная*. Все группы этого ряда решены, осталось поставить крестики в клетки **14** и **15**.
33. **В строке 3** образовалась группа из 1 клетки, с чем мы можем себя поздравить – теперь весь ряд решён.
34. **В строке 13** клетка **M** – *конечная* для группы из 2 клеток, значит, клетка **L** *чёрная*, а предыдущая – *белая*.
35. **В строке 14** остались неразгаданными 9 клеток (**D – L**), в которых должна разместиться группа из 7 клеток, и мы должны признать, что клетки **F-J** неизбежно *чёрные*.
36. **В столбце K** хватает места только для последней группы, и мы должны отметить свой очередной успех.
37. **В строке 14** клетка **D** *белая,* по *Правилу габаритов***.**

38. **В столбце L** клетка **13** – *начальная* для группы из 2 клеток. Следующую за ней клетку мы *закрашиваем*, в последнюю ставим *крестик* – и ряд приказал долго жить.

Положение пресловутой японской головоломки критическое (Рис. Ф4.19).

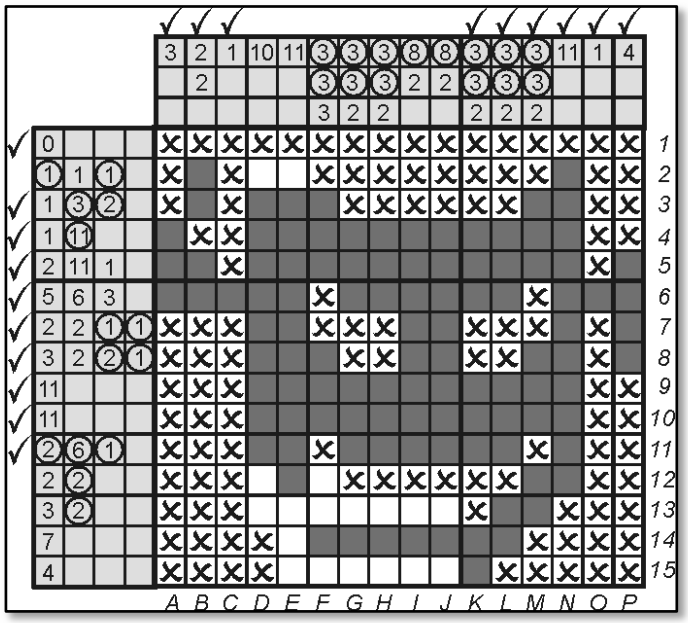


Рис. Ф4.19. И мы близки к победе!

39. **В строке 15** клетка **K** – *конечная* для группы из 4 клеток. И этому ряду конец.
40. **В столбцах F, G** клетка **14** – *конечная* для последней группы. Оба ряда – «туды их в качель».
41. **В строке 14** образовалась единственная группа, так что и ряду тоже *kaput*. (Не забудьте на радостях поставить ему *крестик* в клетку **E**.)
42. **В строке 12** возникла группа из 2 клеток. Клетка **D** – *белая*, а ряд благополучно *того*.
43. Со **столбцом D** разобраться проще пареной репы: клетка **13** *белая*, а клетка **2**, напротив, *чёрная*.
44. **В строке 2** образовалась последняя группа, нам осталось наградить клетку **E** *крестом* (посмертно).
45. **В столбце E** ставим жирную чёрную точку в несчастливую клетку **13**, и пусть этот ряд покоится с миром.
46. **В строке 13** получилась группа из 3 клеток. Ставим последние *кресты* в этой задаче. Аминь.

Все клетки разгаданы. Теперь, коли ещё остались силы, вы можете полюбоваться зрелыми плодами своей интеллектуальной деятельности (Рис. Ф4.20).

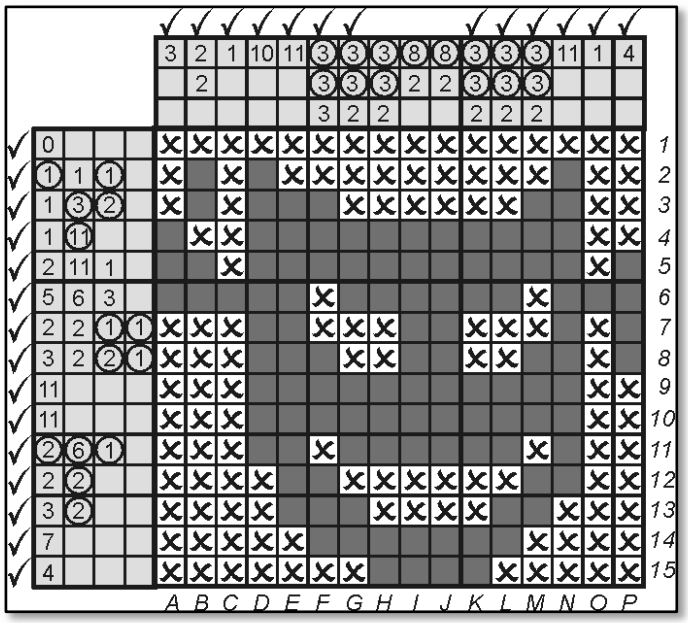


Рис. Ф4.20. Разделали в пух и прах!

Аналогично «раскрашиваются» и более сложные рисунки – правила решения одинаково верны для всех задач. Однако, если рисунок имеет размеры побольше нашего (а чаще всего так оно и есть), да плюс ко всему множество мелких групп, то и потрудиться придётся преизрядно. Не удивительно, что где-то можно ошибиться и вместо чёрной клетки поставить белую (или наоборот), из-за чего всё решение пойдёт дальше шиворот-навыворот и вы напрасно потратите остаток своей жизни на тщетные поиски коварных заблуждений. Более того, некоторые задачи садистского толка вообще устроены так, что их невозможно решить «чисто логически», не делая предположений насчёт цвета отдельных клеток. Скорее всего, одно из них окажется неверным, и это сильно огорчит вас, потому что придется несолоно хлебавши возвращаться назад, а, значит, что вам не избежать «протокола», в котором вы должны будете фиксировать позицию на поле перед тем, как наугад пометить клетку. Тут уж вы повеселитесь на полную катушку!

Кибернетическое убийство времени, или Компью-тер-поводырь

Заграница нам поможет!

Один из посулов Остапа Бендера

Если вы любитель «рукоприкладства» в решении головоломок, но тяготитесь бумажной рутиной, то с помощью нашей будущей программы сможете решать японские головоломки на экране монитора. Хотя надо иметь ангельское терпение, чтобы часами решать на компьютере задачи вручную, когда тот же компьютер справится с ними в «уно моменто». Я надеюсь на ваше благоразумие, поэтому ручному решению японских кроссвордов в нашей программе мы уделим меньше внимания, чем автоматическому. Если же вас привлекает именно ручная, тонкая работа, то я могу порекомендовать вам несколько программ, которые помогут вам убедиться в бренности всего сущего.

1. Вас, несомненно, порадует «японским» дизайном *игра Cross Image* компании *Xdyne* (Рис. Ф4.21).

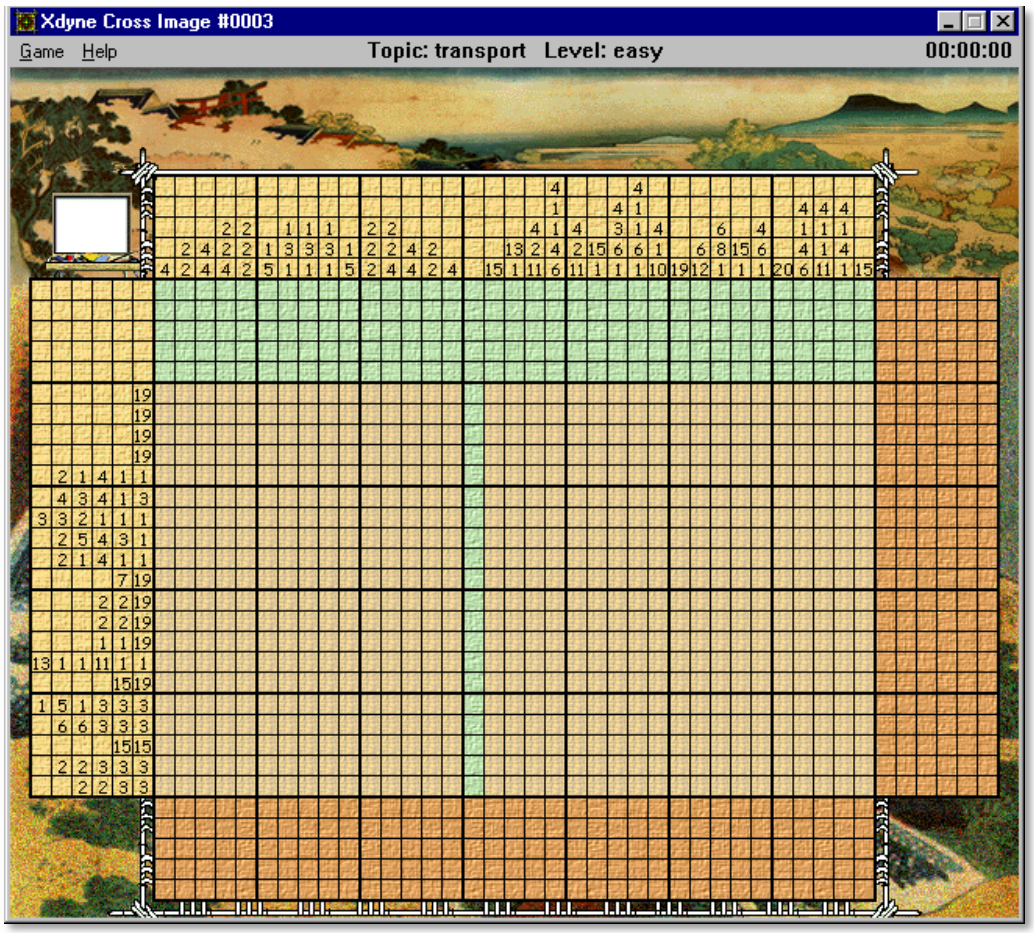


Рис. Ф4.21. Японисто!

Впрочем, радоваться вам придётся недолго: справившись с тремя прилагаемыми задачами, вы задумаетесь, где бы достать новые. К сожалению, ввести свои задачи вам не удастся, так что эта программа почти бесполезная.

2. Значительно «правильнее» программа Романа Гантверга *Японский кроссворд. Игра и редактор* с библиотекой из нескольких десятков заданий (Рис. 3.22, слева), к тому же вы сможете вводить и свои собственные. Для автоматического решения задач вполне уместно использовать программу *Soluter* Гантверга и Самсонова (Рис. Ф4.22, справа).

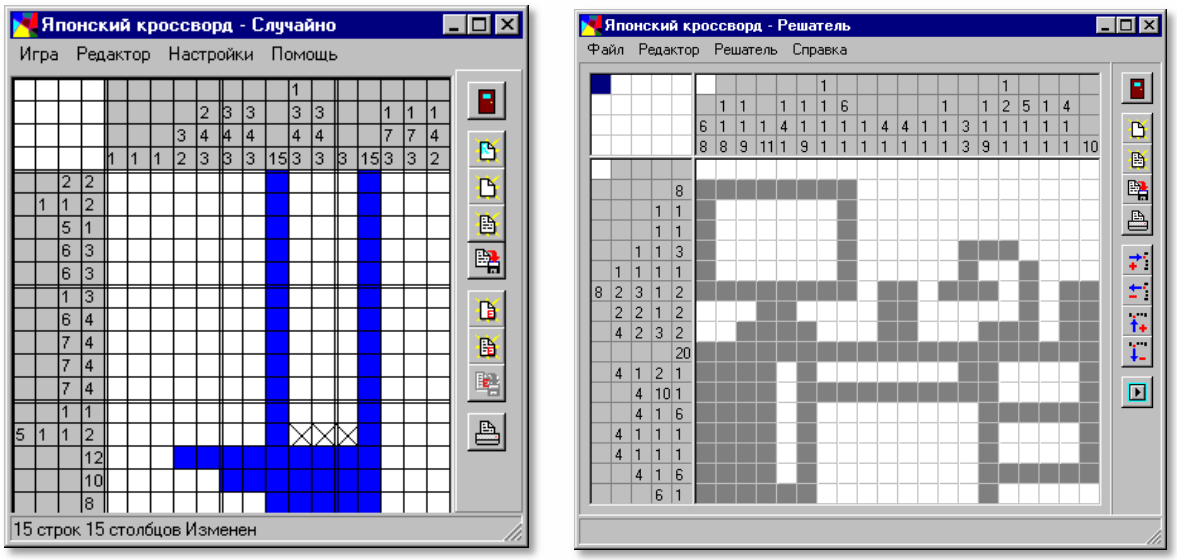


Рис. Ф4.22. Программы Гантверга и Самсонова

3. Не менее интересна программа *Японский кроссворд 2000* Егоркина Игоря Владимировича, которая включает 315 задач. Вы можете и самостоятельно вводить новые задания, а программа в трудную минуту придёт вам на помощь и дорешает задачу, если силы оставят вас (Рис. Ф4.23).

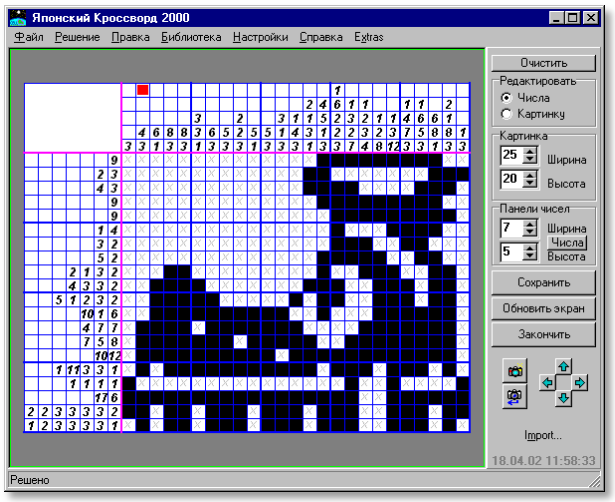


Рис. Ф4.23. Японский кроссворд 2000 в действии

4. А в программе Антона Головина *Японская головоломка* вы, как Фигаро или Труффальдино из Бергамо, сможете одновременно биться на нескольких фронтах, решая несколько задач в отдельном окне (Рис. Ф4.24).

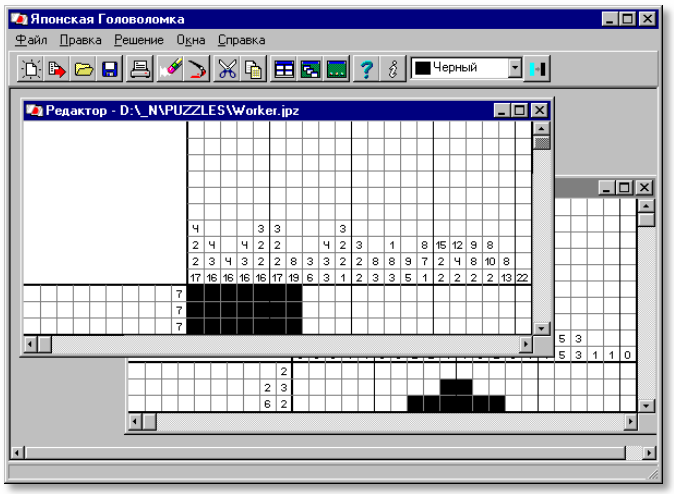


Рис. Ф4.24. Японская головоломка

5. Беднее по графическому исполнению, но близка по возможностям к первой программе игра *JpcWin* Антона Ильяшенко (Рис. Ф4.25).

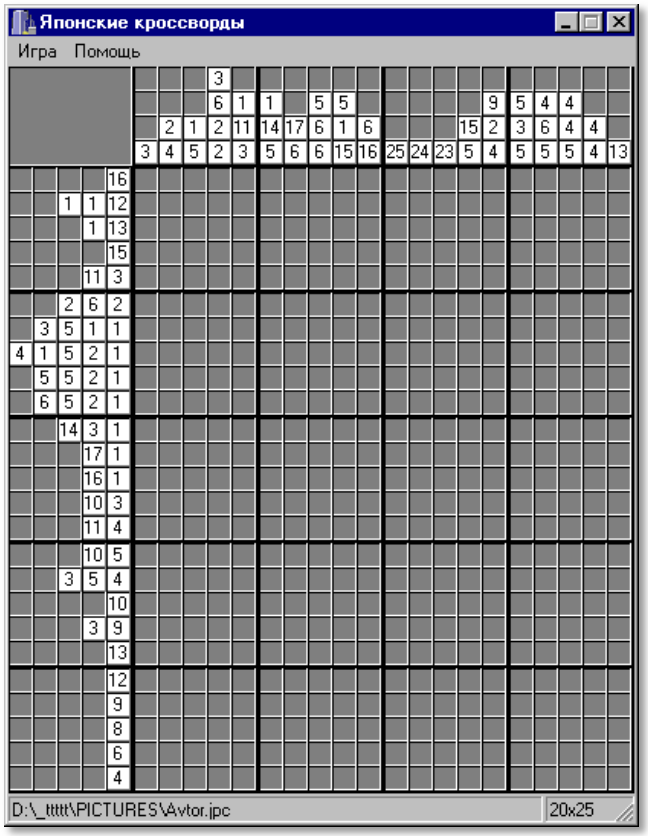


Рис. Ф4.25. Программа *JpcWin*

6. Очень мощная программа *Cross+A* Сергея Кутасова и Ильи Морозова умеет решать не только черно-белые (Рис. 3.26, слева), но и *цветные* задачки (Рис. Ф4.26, справа).

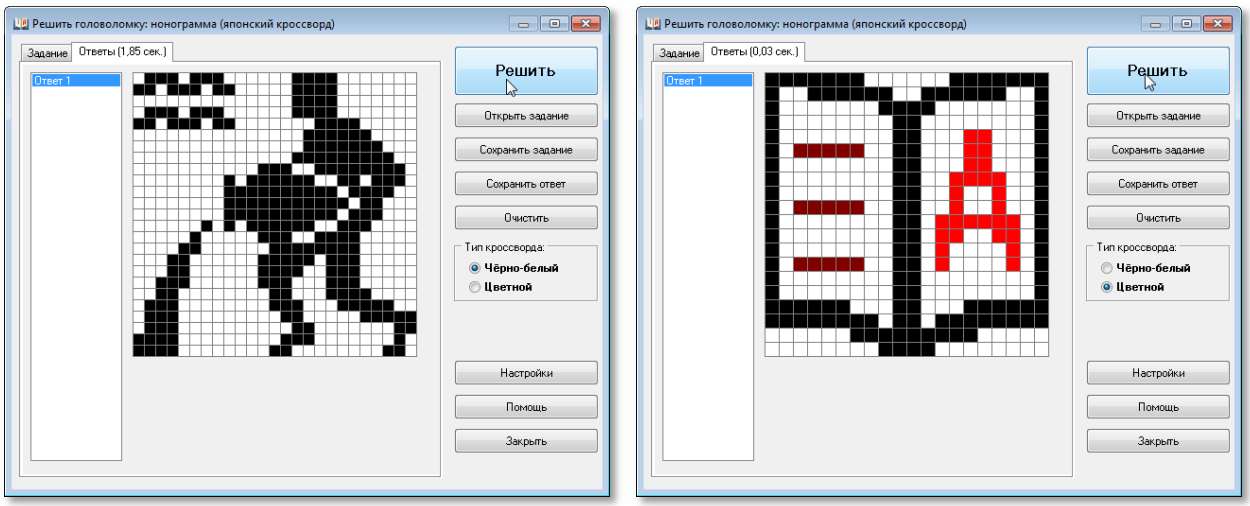


Рис. Ф4.26. *Cross+A* решает *Водолейную* задачу

К тому же она позволяет создавать и редактировать собственные задачи, а также поддерживает файлы разных форматов.

7. И последняя программа – *Gunn's Japanese Puzzle* – также предлагает решать только свои собственные задачи. А их прилагается не очень много…

Решительный штурм, или Как расколоть японские кроссворды

Взялся за гуж, не говори, что не дюж.

Из русского фольклора

Каждый любитель головоломок погорячее, как и былинный витязь на распутье, должен выбрать одну из дорог: решать задачу «вручную» или с помощью компьютера. Оба пути не из лёгких (бытует заблуждение, что на компьютере задачу и дурак решит), а потому по-своему интересны (а зачем вообще браться за простые задачи, например, сканворды?). Но тот, кто предпочтёт «сотрудничество» с компьютером, получит двойное удовольствие – и задачу решит (и не только одну, но и все подобные), и программу напишет, что само по себе во много раз интереснее любой головоломки. Когда *ваша* программа начнёт щёлкать сложнющие задания как орешки, вас охватит ни с чем несравнимая радость творца, из «ничего» создавшего целый мир, в котором всё движется и шевелится, покоряясь его воле.

Мы напишем программу, которой будет по зубам любой японский кроссворд. Она может послужить основой для вашего творчества – изучайте, изменяйте и улучшайте её – до тех пор, пока не удовлетворитесь полученным результатом.

А начнём мы наш новый проект по привычке с соблюдения формальностей, то есть создадим форму солидных размеров, так как многие задачи имеют до 50-80 клеток в ширину и высоту. Свойства **формы**:

Width=729
Height=576
BorderIcons= [biSystemMenu,biMinimize]
BorderStyle=bsSingle
Caption=’ ЯПОНСКИЙ РИСУНОК’
Position=poDesktopCenter

По образу и подобию «бумажных» задач мы разместим на форме 3 сетки, роль которых исполнят фавориты наших программ – компоненты *TDrawGrid*.

dgPole (игровое поле), его свойства:

ColCout=50
DefaultColWidth=14 (все ячейки будут иметь размер 14 x 14 пикселей, иначе цифры будут слишком мелкими и плохо различимыми на экране)
DefaulDrawig=False (выводить содержимое ячеек будем сами)
DefaultRowHeight=14
FixedCols=0
FixedRows=0
Height=419
Left=40
Options=[goVertLine,goHorzLine] (оставим только вертикальные и горизонтальные линии, другие возможности компонента нам не понадобятся)
RowCout=50
ScrollBars=ssNone
Top=76
Width=449

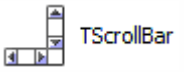
dgColsNum (верхнее числовое поле) со свойствами:

ColCout=10
DefaultColWidth=14
DefaulDrawig=False
DefaultRowHeight=14
FixedCols=0
FixedRows=0
Height=33
Left=44

```
Options=[goVertLine,goHorzLine]
RowCout=4
ScrollBars=ssNone
Top=76
Width=449
```

dgRowsNum (левое числовое поле):

```
ColCout=4
DefaultColWidth=14
DefaulDrawig=False
DefaultRowHeight=14
FixedCols=0
FixedRows=0
Height= 421
Left=4
Options=[goVertLine,goHorzLine]
RowCout=10
ScrollBars=ssNone
Top=76
Width=33
```

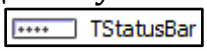
Так как размеры полей могут превышать возможности формы, то нам придётся позаботиться об их *прокрутке*, чтобы добраться и до скрытых от нашего пытливого взора частей рисунка. Для воплощения этой мечты удобнее всего воспользоваться услугами компонента *TScrollBar* 

со страницы *Standard Панели компонентов*. Нетрудно догадаться, что их потребуется ровно два – для вертикальной - *sbVert* - и горизонтальной - *sbHorz* – прокрутки. Большинство их свойств можно не изменять в *Инспекторе объектов*, так как полосы прокрутки нам придётся устанавливать на форме в соответствии с размерами полей. Но о некоторых, постоянных свойствах лучше позаботиться сразу. Для *sbVert* установите:

```
Cursor= crHandPoint (точно такой же курсор устанавливайте и для
других компонентов, на которые придётся «нажимать»)
Kind= sbVertical
LargeChange=5
PageSize=5
```

Для *sbHorz*:

```
Kind= sbHorizontal
LargeChange=5
PageSize=5
```

Для вывода актуальной информации мы установим на форме компонент *StatusBar1* 

:

```
Align=alBottom
Autohint=True
Height=25
Showhint=True
```

Дважды щёлкните на нём и в *Редакторе* разметьте панели:

```
 0 (в крайней левой панельке будут выводиться подсказки (hint):
Alignment=taLeftJustify
Width=196;
 1 (следующая панелька покажет размеры загруженной картинки):
Alignment=taCenter
Text=' 28 x 28'
Width=60;
 2 (далее следует информация о числе клеток в картинке):
Alignment=taCenter
Text=' Клеток: 0'
Width=90;
 3 (горизонтальная координата курсора на игровом поле):
Alignment=taCenter
Text=' X= 0'
Width=40;
 4 (вертикальная координата курсора на игровом поле):
Alignment=taCenter
Text=' Y= 0'
Width=40;
 5 (режим работы программы: ожидает ли она наших действий или ре-
шает задачу):
Alignment=taCenter
Text=' ОЖИДАНИЕ'
Width=90;
 6 (можете приспособить для своих нужд).
```

Под заголовком формы разместите кнопки *TSpeedButton*:

sbtNewFig  – служит для изменения размеров полей:

```
Height=36
Width=36 (все остальные кнопки точно такие же квадратные)
Hint=' Размеры|Изменить размеры полей'
Left=4
Showhint=True (установите у всех кнопок)
Top=0 (тоже у всех кнопок));
```

sbtClearPole  – очищает все поля, игровое поле становится серым, верхнее и левое – белыми, без чисел:

```
Hint=' Очистить| Очистить поля';
```

sbtWhiteGrid  – очищает все поля, игровое поле становится белым, что удобно для рисования новых картинок:

```
Hint=' Белая сетка|Белая сетка для рисования';
```

sbtNumbers  – оцифровывает нарисованную вами картинку:

```
Caption='№'
Hint=' Оцифровать|Оцифровать рисунок';
```

sbtLoadFig  – загружает сохранённые ранее картинки:

```
Hint=' Загрузить | Загрузить рисунок'
```

Для её работы необходим компонент *TOpenDialog* со страницы *Dialogs Панели компонентов*;

sbtOpenPicture  – загружает растровые рисунки и значки для создания из них новых рисунков:

```
Hint=' Импорт|Импорт растров и значков'
```

Для её работы установите компонент *TOpenPictureDialog* 

с той же страницы;

sbtSaveFig  – сохраняет на диске задачу (данные из числовых полей):

```
Hint=' Записать|Записать задачу')
```

Потребуется компонент *TSaveDialog* 

, который вы найдёте там же;

sbtSavePic  – сохраняет на диске рисунок (из игрового поля):

```
Hint=' Записать рисунок|Записать рисунок';
```

sbtStart  – начинает автоматическое решение задачи:

```
Hint=' Решить|Решить задачу';
```

sbtStop 

– останавливает решение задачи, если оно затягивается:

```
Hint= ' Остановить|Остановить поиск';
```

sbtExit  – заканчивает работу с программой:

```
Hint=' Выход|Выход из программы';
```

Ещё три кнопки служат *переключателями*.

sbtSound  – включает и выключает звуковые эффекты:

```
AllowAllUp=True
GroupIndex=1
Hint=' Звук|Звук вкл-откл';
```

sbtMove  – включает и выключает режим перемещения рисунка:

```
AllowAllUp=True
GroupIndex=2
Hint=' Двигать|Двигать рисунок';
```

sbtDraw  – включает и выключает режим рисования картинки и ручного решения задачи:

```
AllowAllUp=True
GroupIndex=3
Hint=' Рисование, решение|Рисовать картинку, решать задачу'.
```

И последнее, что нам осталось, - уютно разместить на форме две *метки* (вы можете легко обойтись и без них):

lblReady – показывает число разгаданных клеток рисунка (при автоматическом решении):

```
Hint=' Разгадано|Число разгаданных клеток';
```

lblLevel – показывает сложность решения задачи (число предположений, «сомнительных» клеток:

```
Hint=' Уровень|Сложность решения задачи'.
```

В итоге перед вашим взором должна открыться такая картина (Рис. Ф4.27).

Вы можете переставить «мебель» по своему вкусу, да только зачем? Сейчас для нас куда важнее наполнить форму нужным содержанием, чтобы она не только радовала взор, но и славно на нас работала.

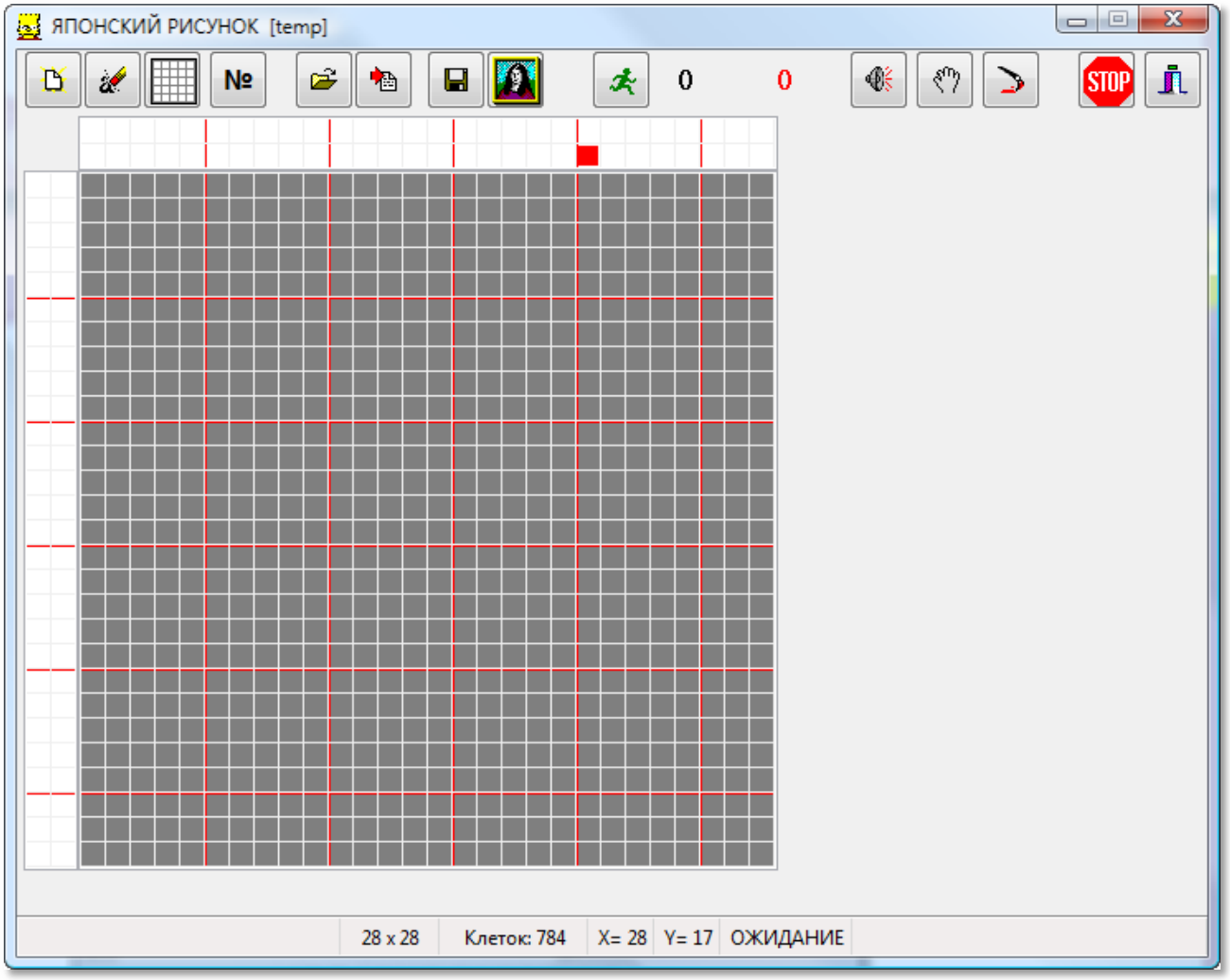


Рис. Ф4.27. Лицевая сторона нашего приложения

Занимательная механика, или За кулисами формы

Природа не терпит пустоты.

Сказал один мудрец

Но часто наполняет её всякой дрянью.

Добавил другой

Предположим, что вас сильно огорчила некая японская задача, которая спутала вам все карты и жизненные планы, а вы не такой человек, чтобы безропотно сносить удары судьбы. Тогда вооружитесь компьютером и нанесите ответный удар.

Итак, ваш противник в виде несгибаемой задачи строит вам фигуры со станицы газеты, а то и журнала. Компьютер газет не читает (да и журна-

лов тоже), поэтому нужно подать ему головоломку на *блюдечке с голубой каёмочкой*. Сервировать блюдо можно по-разному. Первое что приходит в голову – просто перенести числа из газеты в числовые поля нашей программы.

Компонент *TDrawGrid* вполне уместен для отображения информации, но хранить данные он не умеет, поэтому каждой сетке мы сопоставим *массив*. Перед определением типа формы запишем (все остальные переменные, не относящиеся к процедурам, мы также будем объявлять в этом разделе):

```
var
  //массив, в котором хранится копия поля
  masPole : TPole;
```

Так как подобные массивы пригодятся нам и в других случаях, то перед разделом переменных мы опишем *тип игрового поля*:

```
//данные игрового поля
type  TPole  =  array[0..MAX_POLE_WIDTH-1,  0..MAX_POLE_HEIGHT-1]of
Integer;
```

Верхние границы массива лучше задать в виде *констант* – и понятнее, и легче изменить в случае необходимости. Константы объявляем сразу после раздела *uses*:

```
unit NipponUnit;

////////////////////////////////////////////////////////////////////////
//                                                                    //
//             ПРОГРАММА ДЛЯ РЕШЕНИЯ ЯПОНСКИХ ГОЛОВОЛОМОК             //
//                                                                    //
////////////////////////////////////////////////////////////////////////

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
Dialogs,
  ExtCtrls, StdCtrls, Grids, ComCtrls, Buttons, mmsystem, ExtDlgs;

const
  NAME_PROG = 'ЯПОНСКИЙ РИСУНОК';
  MAX_POLE_WIDTH = 70;   //макс. ширина поля в клетках
  MAX_POLE_HEIGHT = 50;  //макс. высота поля в клетках
```

Установленные предельные размеры сетки позволяют загрузить практически все существующие задачи. Если же вам этого недостаточно, вы можете увеличить их. Но имейте в виду, что большие и плохо составленные задачи могут потребовать достаточно много времени для решения!

В массиве игрового поля мы будем хранить данные о *цвете* клеток. Помните, как нам приходилось хитрить, чтобы отметить разгаданные белые клетки? И это всё из-за того, что не решённые ещё клетки на бумаге тоже белые. Но компьютер не газета, поэтому мы просто сделаем *неизвестные* клетки *серыми*, а *известные*, как и полагается, *белыми* и *чёрными*. Для удобства пользования и в целях профилактики склероза объявим соответствующие *константы*:

```
  NUM_WHITE = 2;         //белая клетка в masPole
  NUM_BLACK = 1;         //чёрная клетка в masPole
  NUM_GRAY = 0;          // серая клетка в masPole (фон)
```

Эти же клетки на игровом поле мы будем окрашивать в нужные цвета:

```
  COLOR_WHITE: TColor= $FFFFFF;  //белая клетка на поле
  COLOR_BLACK: TColor= $0;       //чёрная клетка на поле
  COLOR_GRAY: TColor= $7F7F7F;   //серая клетка на поле
```

Текущее значение размеров поля мы будем хранить в переменных:

```
  POLE_WIDTH: integer = 28;      //ширина поля в клетках
  POLE_HEIGHT: integer = 28;     //высота поля в клетках
```

Массивы для числовых полей отличаются от массива игрового поля не только размерами, но и структурой:

```
  //массив, в котором хранятся верхние числа:
  masColsNum   :   array[0..MAX_POLE_WIDTH-1,   -1..MAX_CELLNUM-1]of
TCell;
  //массив, в котором хранятся левые числа:
  masRowsNum   :   array[-1..MAX_CELLNUM-1,   0..MAX_POLE_HEIGHT-1]of
TCell;
```

В каждой ячейке массивов будет храниться *число*, записанное в соответствующей клетке сетки (нам понадобится не только само число, но и его представление в виде *строки*), а также *статус группы*, для описания которого мы введём новый *тип*:

```
//статус групп в числовых полях:
//stWhite - группа не решена,
//stYellow - группа решена,
//stGreen - ряд решён
type TStatusGroup=(stWhite, stYellow, stGreen);
//данные для каждой клетки числовых полей:
type TCell = Record
  sNum: String;           //строковое представление числа в клетке
  Num: integer;              //число в клетке
  StatusGroup: TStatusGroup; //статус группы
```

```
end;
```

Кроме того, в ячейках с индексом –1 мы будем хранить *число групп* в данном ряду.

Объявим также *константу*, ограничивающую размеры числовых полей:

```
MAX_CELLNUM = 14;       //макс. ширина/высота числовых полей
```

Этого значения вполне достаточно для большинства заданий (правильнее сказать, для всех *разумных* заданий). Впрочем, никто не помешает вам увеличить его по собственному разумению. О последствиях этого читайте выше.

Переменные для текущих (установленных) размеров числовых полей:

```
  LEFT_POLE_WIDTH: integer = 2; //ширина левого числового поля в
клетках
  TOP_POLE_HEIGHT: integer = 2; //высота верхнего числового поля в
клетках
```

Очевидно, что высота *левого* числового поля равна *высоте* игрового поля, а *ширина* верхнего числового поля – *ширине* игрового поля (если для вас это не очевидно, а невероятно, то просто внимательно посмотрите на картинку с формой).

Так как размеры экрана обычного монитора не позволяют целиком отобразить всё поле больших размеров, то мы должны ввести *константы* для предельных размеров «вырезки» поля:

```
  MAX_ALL_CELLS_HEIGHT = 30;     //макс. высота игрового и числово-
го полей
                                 //(видимая часть)
  MAX_ALL_CELLS_WIDTH = 46;      //макс. ширина игрового и числово-
го полей
```

Для пущего куражу введём несколько нестандартных *курсоров* (в разделе констант):

```
  //курсоры:
  crHand : integer= 4;
  crMove : integer= 5;
  crKistj : integer= 7;
```

Не забудьте подключить к проекту *файл ресурсов* со своими курсорами:

```
{$R *.DFM}
```

```
{$R MyCursors.res}
```

За *режим работы* программы будет отвечать переменная:

```
 status: string='';
```

Благодаря ей, нажимая любую кнопку (я надеюсь, вы понимаете, о каких кнопках речь), мы сможем узнать, что делает наша программа. Если она занята решением задачи, то беспокоить её не следует. Можно отключать все «лишние» кнопки на время решения задачи, но это потребует б***о***льших усилий.

Название картинки мы будем хранить в *переменной*:

```
  NameFig: string= 'temp';
```

Если картинка не загружалась и не сохранялась на диске, то имя ей - *temp*. Название картинки, чтобы не забыть оного, мы будем выводить в *заголовке* формы.

Чтобы лучше ориентироваться в «пространстве», мы будем следить за положением курсора в сетках:

```
  //координаты клетки с курсором:
  cellMouse: TPoint;     //игровое поле
  cellMouseTop: TPoint;  //верхнее числовое поле
  cellMouseLeft: TPoint; //левое числовое поле
```

Ещё две переменные будут следить за *количеством* клеток:

```
  AllCells: Integer=0;   //всего клеток на поле
  ReadyCells: Integer=0; //разгадано клеток
```

Теперь у нас есть всё для того, чтобы начать творить. Объявим две *процедуры*, с помощью которых можно вернуть массивам девственную чистоту:

```
    { Private declarations }
    procedure Clear_masColsNum;
    procedure Clear_masRowsNum;
```

Сами процедуры очень просты: мы стираем все числа в ячейках и указываем, что ряд не решён:

```
//ОЧИСТИТЬ МАССИВ ВЕРХНЕГО ЧИСЛОВОГО ПОЛЯ
procedure TForm1.Clear_masColsNum;
var
  i,j: integer;
begin
```

```
  For i := 0 To MAX_POLE_WIDTH-1 do
    for j := -1 to  MAX_CELLNUM - 1 do
    begin
      masColsNum[i,j].sNum:='';              // - числа нет
      masColsNum[i,j].StatusGroup:= stWhite;// - ряд не решён
    end
end;
//ОЧИСТИТЬ МАССИВ ЛЕВОГО ЧИСЛОВОГО ПОЛЯ
procedure TForm1.Clear_masRowsNum;
var
  i,j: integer;
begin
  For i := -1 To MAX_CELLNUM-1 do
    for j := 0 to  MAX_POLE_HEIGHT - 1 do
    begin
      masRowsNum[i,j].sNum:='';              // - числа нет
      masRowsNum[i,j].StatusGroup:= stWhite;// - ряд не решён
    end
end;
```

После этих процедур объявляем ещё одну:

```
    procedure InvalidateGrids;
```

Она будет «освежать» игровое и числовые поля после глобальных изменений:

```
//ОБНОВИТЬ ПОЛЯ
procedure TForm1.InvalidateGrids;
begin
  dgPole.Invalidate;
  dgColsNum.Invalidate;
  dgRowsNum.Invalidate;
end;
```

В типе формы объявите процедуру, которая будет *подготавливать* сетки заданных размеров и при необходимости добавлять полосы прокрутки:

```
procedure Prepare(ColCount, RowCount, TopCount, LeftCount: Inte-
ger);
```

```
//ПОДГОТОВИТЬ НОВУЮ ИГРУ
procedure TForm1.Prepare(ColCount, RowCount, TopCount, LeftCount:
Integer);
//ColCount - число колонок на игровом поле
//RowCount - число строк на игровом поле
//TopCount - число строк на верхнем поле
//LeftCount - число колонок на левом поле
var
  w,h,lw: integer;
  i,j,n: integer;
```

```
begin
  //скорректировать размеры полей:
  if ColCount< 2 then ColCount:= 2;
  if RowCount< 2 then RowCount:= 2;
  if TopCount< 1 then TopCount:= 1;
  if LeftCount< 1 then LeftCount:= 1;
  //
  //=============  игровое поле  ==============
  //
  //размер клетки в пикселах:
  w:= dgPole.DefaultColWidth;
  h:= dgPole.DefaultRowHeight;
  //толщина линий:
  lw:= dgPole.GridLineWidth;
  //размеры игрового поля в клетках:
  dgPole.ColCount:= ColCount;
  dgPole.RowCount:= RowCount;
  i:= ColCount;
  j:= RowCount;
  //если ширина игрового поля + ширина левого числового поля
  //превышают MAX_ALL_CELLS_WIDTH клеток,
  //то поставить вертикальную полосу прокрутки:
  if ColCount > MAX_ALL_CELLS_WIDTH  - LeftCount then begin
    i:= MAX_ALL_CELLS_WIDTH  - LeftCount;
    sbHorz.Visible:= true
    end
  else sbHorz.Visible:= false;
  //если высота игрового поля + высота верхнего числового поля
  //превышают MAX_ALL_CELLS_HEIGHT клеток,
  //то поставить горизонтальную полосу прокрутки:
  if RowCount > MAX_ALL_CELLS_HEIGHT  - TopCount then begin
    j:= MAX_ALL_CELLS_HEIGHT  - TopCount;
    sbVert.Visible:= true
    end
  else sbVert.Visible:= false;
  //размеры в пикселах видимой части игрового поля:
  dgPole.Width:= 3 + (w + lw)* i+1;
  dgPole.Height:= 3 + (h + lw)* j+1;
  //
  //===================  числовые поля  ========================
  //
  //размеры клеток числовых полей = размерам клеток игрового поля:
  dgColsNum.DefaultColWidth:= w;
  dgColsNum.DefaultRowHeight:= h;
  dgRowsNum.DefaultColWidth:= w;
  dgRowsNum.DefaultRowHeight:= h;
  //размеры числовых полей:
  dgColsNum.ColCount:= ColCount; // = ширине игрового поля
  dgColsNum.Width:= dgPole.Width;
  dgColsNum.RowCount:= TopCount; // задаётся
  //высота верхнего числового поля в пикселах:
  dgColsNum.Height:= 3 + (h + lw)* TopCount;
```

```
   //высота левого числового поля в клетках = высоте игрового по-
ля:
  dgRowsNum.RowCount:= dgPole.RowCount;
  //высота левого числового поля в пикселах:
  dgRowsNum.Height:= dgPole.Height;
  dgRowsNum.ColCount:= LeftCount; // задаётся
  //ширина левого числового поля в пикселах::
  dgRowsNum.Width:=3 + (h + lw)* LeftCount;
  //положение на форме:
  dgColsNum.left:= dgRowsNum.Left + dgRowsNum.Width;
  dgRowsNum.top:=  dgColsNum.top +  dgColsNum.Height;
  dgPole.top:= dgRowsNum.top;
  dgPole.Left:=dgColsNum.Left;
  //
  //==============  полосы прокрутки  ================
  //
  //разместить полосы прокрутки рядом с игровым полем:
  if sbVert.visible then
  begin
    //коорд. левой стороны верт. полосы прокрутки:
    sbVert.Left:= dgPole.Left+ dgPole.Width+ 5;
    //её высота:
    sbVert.Height:= dgPole.Height;
    //коорд. верхней стороны:
    sbVert.Top := dgPole.top;
    //макс. позиция:
    n:= dgPole.RowCount - (MAX_ALL_CELLS_HEIGHT-
dgColsNum.RowCount);
    if n< sbVert.LargeChange then n:= sbVert.LargeChange;
    sbVert.Max := n+4;
  end;
  if sbHorz.visible then
  begin
    //коорд. верхней стороны гориз. полосы прокрутки:
    sbHorz.Top := dgPole.top+ dgPole.Height+ 5;
    //ширина:
    sbHorz.Width:= dgPole.Width;
    //коорд. левой стороны:
    sbHorz.Left:= dgPole.Left;
    //макс. позиция:
    n:= dgPole.ColCount - (MAX_ALL_CELLS_WIDTH-
dgRowsNum.ColCount);
    if n< sbHorz.LargeChange then n:= sbHorz.LargeChange;
    sbHorz.Max:= n;
  end;
  //вывести размеры фигуры в панели:
  statusbar1.Panels[1].text:= inttostr(ColCount)+ ' x ' + intto-
str(RowCount);
  //подсчитать число клеток в фигуре:
  AllCells:= ColCount * RowCount;
  statusbar1.Panels[2].text:='Клеток: '+inttostr(AllCells);
end; //Prepare
```

Если картинка не видна на экране целиком, то её можно перемещать с помощью полос прокрутки. Хитрость же заключается в том, что при сдвиге *игрового* поля по горизонтали одновременно должно сдвигаться и *верхнее* числовое поле, а при сдвиге по вертикали – *левое* числовое поле, иначе данные в них сместятся по отношению к рядам игрового поля.

```
//СИНХРОННО СДВИГАТЬ ПОЛЯ ПО ГОРИЗОНТАЛИ
procedure    TForm1.sbHorzScroll(Sender:    TObject;    ScrollCode:
TScrollCode;
  var ScrollPos: Integer);
begin
  dgPole.LeftCol:= ScrollPos; dgColsNum.LeftCol:= dgPole.LeftCol;
end;
//СИНХРОННО СДВИГАТЬ ПОЛЯ ПО ВЕРТИКАЛИ
procedure    TForm1.sbVertScroll(Sender:    TObject;    ScrollCode:
TScrollCode;
  var ScrollPos: Integer);
begin
  dgPole.TopRow := ScrollPos; dgRowsNum.TopRow:= dgPole.TopRow;
end;
```

Щёлкните по носу кнопку *sbtClearPole* и напишите:

```
//ОЧИСТИТЬ ПОЛЯ
procedure TForm1.sbtClearPoleClick(Sender: TObject);
var
  i,j: integer;
begin
  if status='ПОИСК' then exit;
  //все клетки поля - серые:
  For i := 0 To POLE_WIDTH-1 do
    for j := 0 to  POLE_HEIGHT - 1 do
      masPole[i,j]:= NUM_GRAY;
  //очистить цифровые поля:
  Clear_masColsNum;
  Clear_masRowsNum;
  InvalidateGrids;
  //фигура не загружена:
  NameFig:='temp';
  //вывести в заголовке формы имя временного файла:
  caption:= NAME_PROG + '  [' + NameFig + ']';
  //координаты мыши:
  cellmouse.x:=-1; cellmouse.y:=-1;
end;
```

Нажимайте на неё всякий раз, когда вам потребуется очистить *все* поля.

Аналогично поступите и с кнопкой *sbtWhiteGrid*:

```
//ОЧИСТИТЬ ПОЛЕ ДЛЯ РИСОВАНИЯ
procedure TForm1.sbtWhiteGridClick(Sender: TObject);
```

```
var
  i,j: integer;
begin
  if status='ПОИСК' then exit;
  //все клетки - белые:
  dgPole.Canvas.Brush.Color:=COLOR_WHITE;
  For i := 0 To MAX_POLE_WIDTH-1 do
    for j := 0 to  MAX_POLE_HEIGHT - 1 do
      masPole[i,j]:= NUM_WHITE;
  //очистить цифровые поля:
  Clear_masColsNum;
  Clear_masRowsNum;
  InvalidateGrids;
  //фигура не загружена:
  NameFig:='temp1';
  //вывести в заголовке формы имя загруженного файла:
  caption:= NAME_PROG + '  [' + NameFig + ']';
  //координаты мыши:
  cellmouse.x:=-1; cellmouse.y:=-1;
end;
```

Она пригодится нам в скором светлом будущем, когда мы начнём *рисовать* свои картинки на игровом поле. Обе эти процедуры различаются только одной строкой, поэтому правильнее будет объединить их в одну (домашнее задание!).

Все данные о группах и цвете клеток на игровом поле хранятся в массивах, и в сетках на форме ничего не появится, пока мы сами не позаботимся об этом. Как обычно, прорисовка клеток всех сеток происходит при вызове их метода *DrawCell*.

Для *игрового* поля это делается так. Определяем, какое число стоит в массиве на соответствующем месте, и закрашиваем клетку нужным цветом. Затем для лучшей ориентировки на поле проводим *красные линии* (если они на вас действуют, как красная тряпка на быка, просто сотрите часть кода).

```
//НАРИСОВАТЬ КЛЕТКУ ИГРОВОГО ПОЛЯ
procedure TForm1.dgPoleDrawCell(Sender: TObject; ACol, ARow: Inte-
ger;
  Rect: TRect; State: TGridDrawState);
var  area: tRect;
begin
  //закрасить клетку нужным цветом:
  area:=dgPole.CellRect(ACol, ARow);
  case masPole[ACol, ARow] of
    NUM_WHITE: dgPole.Canvas.Brush.Color:=COLOR_WHITE;
    NUM_BLACK: dgPole.Canvas.Brush.Color:=COLOR_BLACK;
    NUM_GRAY: dgPole.Canvas.Brush.Color:=COLOR_GRAY;
```

```
  end;
  dgPole.Canvas.FillRect(area);
  //проводим красные линии через 5 клеток:
  if (ACol>0) and (ACol mod 5 = 0) then begin
    dgPole.Canvas.Pen.Color:= clRed;
    dgPole.Canvas.MoveTo(Rect.Left,Rect.Top);
    dgPole.Canvas.LineTo(Rect.Left,Rect.Bottom)
  end;
  if (ARow>0) and (ARow mod 5 = 0) then begin
    dgPole.Canvas.Pen.Color:= clRed;
    dgPole.Canvas.MoveTo(Rect.Left,Rect.Top);
    dgPole.Canvas.LineTo(Rect.Right,Rect.Top)
  end;
end; //dgPoleDrawCell
```

В *числовых* полях мы действуем аналогично, но нам нужно ещё вывести стоящие в клетках *числа*:

```
//ПРОРИСОВАТЬ ЯЧЕЙКУ
procedure TForm1.dgColsNumDrawCell(Sender: TObject; ACol, ARow:
Integer;
  Rect: TRect; State: TGridDrawState);
var
  sNum: string;
begin
  //если столбец решён, отмечаем его зелёным цветом:
  case masColsNum[ACol,-1].StatusGroup of
    stGreen: dgColsNum.Canvas.Brush.Color:= RGB(0,255,0);
    else begin  //- столбец не решён
      //если группа решена, выделить её жёлтым цветом:
      if masColsNum[ACol,ARow].StatusGroup= stYellow then
        dgColsNum.Canvas.Brush.Color:= clYellow
      else dgColsNum.Canvas.Brush.Color:= clWhite; //- иначе белым
    end
  end;
  dgColsNum.Canvas.FillRect(Rect);
  //выводим число в клетке (ACol, ARow):
  sNum:= masColsNum[ACol, ARow].sNum;
  with Rect, dgColsNum.Canvas do
    textrect(Rect, left+(right-left-textwidth(sNum)) div 2,
             top+(bottom-top-textheight(sNum)) div 2, sNum);
  //нарисовать красные линии через 5 клеток:
  if (ACol>0) and (ACol mod 5 = 0) then begin
    dgColsNum.Canvas.Pen.Color:= clRed;
    dgColsNum.Canvas.MoveTo(Rect.Left,Rect.Top);
    dgColsNum.Canvas.LineTo(Rect.Left,Rect.Bottom)
  end;
  if (ARow>0) and (ARow mod 5 = 0) then begin
    dgColsNum.Canvas.Pen.Color:= clRed;
    dgColsNum.Canvas.MoveTo(Rect.Left,Rect.Top);
    dgColsNum.Canvas.LineTo(Rect.Right,Rect.Top)
  end;
```

```
end; // dgColsNumDrawCell

//ПРОРИСОВАТЬ ЯЧЕЙКУ
procedure  TForm1.dgRowsNumDrawCell(Sender:  TObject;  ACol,  ARow:
Integer; Rect: TRect;
  State: TGridDrawState);
var
  sNum: string;
begin
  //если столбец решён, отмечаем его зелёным цветом:
  case masRowsNum[-1,ARow].StatusGroup of
    stGreen: dgRowsNum.Canvas.Brush.Color:= RGB(0,255,0);
    else begin  //- столбец не решён
      //если группа решена, выделить её жёлтым цветом:
      if masRowsNum[ACol,ARow].StatusGroup= stYellow then
        dgRowsNum.Canvas.Brush.Color:= clYellow
      else dgRowsNum.Canvas.Brush.Color:= clWhite; //- иначе белым
    end
  end;
  dgRowsNum.Canvas.FillRect(Rect);
  sNum:= masRowsNum[ACol, ARow].sNum;
    with Rect, dgRowsNum.Canvas do
    textrect(Rect, left+(right-left-textwidth(sNum)) div 2,
              top+(bottom-top-textheight(sNum)) div 2, sNum);
  //нарисовать красные линии через 5 клеток:
  if (ACol>0) and (ACol mod 5 = 0) then begin
    dgRowsNum.Canvas.Pen.Color:= clRed;
    dgRowsNum.Canvas.MoveTo(Rect.Left,Rect.Top);
    dgRowsNum.Canvas.LineTo(Rect.Left,Rect.Bottom)
  end;
  if (ARow>0) and (ARow mod 5 = 0) then begin
    dgRowsNum.Canvas.Pen.Color:= clRed;
    dgRowsNum.Canvas.MoveTo(Rect.Left,Rect.Top);
    dgRowsNum.Canvas.LineTo(Rect.Right,Rect.Top)
  end;
end; // dgRowsNumDrawCell
```

Сделайте двойной щелчок на форме и в открывшейся заготовке процедуры напишите:

```
//СОЗДАТЬ ФОРМУ
procedure TForm1.FormCreate(Sender: TObject);
begin
  //очистить поля:
  sbtClearPoleClick(self);
  //вывести сетки заданных размеров:
  Prepare(POLE_WIDTH,POLE_HEIGHT,                    TOP_POLE_HEIGHT,
LEFT_POLE_WIDTH);
  //загрузить курсоры:
  Screen.Cursors[crKistj] := LoadCursor(hinstance,'kistj');
  Screen.Cursors[crHand] := LoadCursor(hinstance,'hand');
```

```
  Screen.Cursors[crMove] := LoadCursor(hinstance,'move');
  //установить курсор для игрового поля:
  dgPole.Cursor :=crHand;
end;
```

Осталось нагрузить делом кнопку *sbtExit*:

```
//ЗАКРЫТЬ ПРОГРАММУ
procedure TForm1.sbtExitClick(Sender: TObject);
begin
  close
end;
```

- и можно, наконец, запустить программу.

Работать она будет исправно: все поля и кнопки на месте и ждут ваших решительных действий. Но, скорее всего, нужные нам размеры полей не совпадут с теми, что установлены по умолчанию (смотрите объявления констант). Так что нам придётся позаботиться об изменении их размеров.

Так как на главной форме места уже не осталось, мы добавим к проекту *новую форму*, на которой и разместим все потребные для этого элементы управления (Рис. Ф4.28).

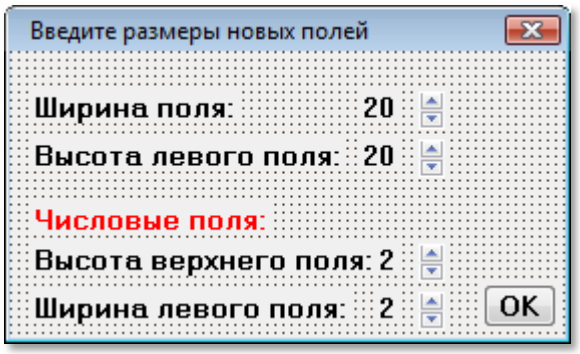


Рис. Ф4.28. Форма для изменения размеров сеток

Свойства формы:

```
BorderIcons=[biSystemMenu]
BorderStyle=bsDialog
Caption=' Введите размеры новых полей'
Name=frmNewFig
Label1: Caption=' Ширина поля:'
Label2: Caption=' Высота левого поля:'
Label3: Caption=' Числовые поля:'
Label4: Caption=' Высота верхнего поля:'
Label5: Caption=' Ширина левого поля:'
LblPoleWidth, lblPoleHeight: Caption='20'
LblTopHeight, lblLeftWidth: Caption='2'
```

Компоненты *TUpDown* 

возьмите на странице **Win32** *Палитры компонентов*:

UpDownPoleWidth, UpDownPoleHeigh:
Min=2
Orientation=udVertical
Position=2
UpDownTopHeight, UpDownLeftWidth:
Min=1

И кнопка *sbtOK* - для комплекта.

Процедуры, модуля *NewFigUnit* настолько просты, что любые комментарии к ним излишни:

```
unit NewFigUnit;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
Dialogs,
  StdCtrls, ComCtrls, Buttons;

type
  TfrmNewFig = class(TForm)
    Label1: TLabel;
    Label2: TLabel;
    Label3: TLabel;
    Label4: TLabel;
    Label5: TLabel;
    lblPoleWidth: TLabel;
    lblPoleHeight: TLabel;
    lblTopHeight: TLabel;
    lblLeftWidth: TLabel;
    UpDownPoleWidth: TUpDown;
    sbtOK: TSpeedButton;
    UpDownPoleHeight: TUpDown;
    UpDownTopHeight: TUpDown;
    UpDownLeftWidth: TUpDown;
    procedure  UpDownPoleWidthClick(Sender:  TObject;  Button:  TUD-
BtnType);
    procedure sbtOKClick(Sender: TObject);
    procedure  UpDownPoleHeightClick(Sender:  TObject;  Button:  TUD-
BtnType);
    procedure  UpDownTopHeightClick(Sender:  TObject;  Button:  TUD-
BtnType);
    procedure  UpDownLeftWidthClick(Sender:  TObject;  Button:  TUD-
BtnType);
  private
```

```
    { Private declarations }
  public
    { Public declarations }
  end;

var
  frmNewFig: TfrmNewFig;

implementation

uses NipponUnit;

{$R *.DFM}

//ИЗМЕНИТЬ ШИРИНУ ПОЛЯ
procedure TfrmNewFig.UpDownPoleWidthClick(Sender: TObject;
  Button: TUDBtnType);
begin
  lblPoleWidth.Caption:= inttostr(UpDownPoleWidth.Position);
end;
//ИЗМЕНИТЬ ВЫСОТУ ПОЛЯ
procedure TfrmNewFig.UpDownPoleHeightClick(Sender: TObject;
  Button: TUDBtnType);
begin
  lblPoleHeight.Caption:= inttostr(UpDownPoleHeight.Position);
end;
//ИЗМЕНИТЬ ВЫСОТУ ВЕРХНЕГО ЧИСЛОВОГО ПОЛЯ
procedure TfrmNewFig.UpDownTopHeightClick(Sender: TObject;
  Button: TUDBtnType);
begin
  lblTopHeight.Caption:= inttostr(UpDownTopHeight.Position);
end;
//ИЗМЕНИТЬ ШИРИНУ ЛЕВОГО ЧИСЛОВОГО ПОЛЯ
procedure TfrmNewFig.UpDownLeftWidthClick(Sender: TObject;
  Button: TUDBtnType);
begin
  lblLeftWidth.Caption:= inttostr(UpDownLeftWidth.Position);
end;
//УСТАНОВИТЬ НОВЫЕ РАЗМЕРЫ ПОЛЕЙ
procedure TfrmNewFig.sbtOKClick(Sender: TObject);
begin
  NipponUnit.POLE_WIDTH:= UpDownPoleWidth.Position;
  NipponUnit.POLE_HEIGHT:= UpDownPoleHeight.Position;
  NipponUnit.TOP_POLE_HEIGHT:= UpDownTopHeight.Position;
  NipponUnit.LEFT_POLE_WIDTH:= UpDownLeftWidth.Position;
  form1.Prepare(POLE_WIDTH,      POLE_HEIGHT,      TOP_POLE_HEIGHT,
LEFT_POLE_WIDTH);
  Close;
end;

end.
```

Вызываться эта форма вместе с «причиндалами» будет при нажатии на кнопку *sbtNewFig*:

```
//ИЗМЕНИТЬ РАЗМЕРЫ ПОЛЕЙ
procedure TForm1.sbtNewFigClick(Sender: TObject);
begin
  frmNewFig.lblPoleWidth.Caption:= inttostr(POLE_WIDTH);
  frmNewFig.lblPoleHeight.Caption:= inttostr(POLE_HEIGHT);
  frmNewFig.lblLeftWidth.Caption:= inttostr(LEFT_POLE_WIDTH);
  frmNewFig.lblTopHeight.Caption:= inttostr(TOP_POLE_HEIGHT);
  frmNewFig.UpDownPoleWidth.Position:= POLE_WIDTH;
  frmNewFig.UpDownPoleWidth.Max:= MAX_POLE_WIDTH;
  frmNewFig.UpDownPoleHeight.Position:= POLE_HEIGHT;
  frmNewFig.UpDownPoleHeight.Max:= MAX_POLE_HEIGHT;
  frmNewFig.UpDownLeftWidth.Position:= LEFT_POLE_WIDTH;
  frmNewFig.UpDownLeftWidth.Max:= MAX_CELLNUM;
  frmNewFig.UpDownTopHeight.Position:= TOP_POLE_HEIGHT;
  frmNewFig.UpDownTopHeight.Max:= MAX_CELLNUM;
  frmNewFig.showmodal;
end;
```

Упомяните добрым словом в *NipponUnit* новый модуль:

```
implementation

uses NewFigUnit;
```

Впрочем, если вы забудете об этом, *Delphi* напомнит вам о ваших обязанностях.

Последние нововведения позволят вам запросто изменять размеры полей, как вашей душе будет угодно. Но нам предстоит ещё научиться вводить числа в числовые поля. Есть несколько способов для этого. Мы постараемся сделать этот процесс удобнее – для себя ведь делаем!

Каждому сверчку по шестку, или Нумеруем клетки

Входит и выходит –
замечательно выходит.

Каламбур Винни-Пуха

Отметим клетку, в которую будут вводиться числа, *красным* цветом. Красный квадратик можно перемещать точно так же, как и обычный курсор, но при переходе с одной клетки на другую нам придётся самим восстанавливать содержимое «покинутой» им клетки. Чтобы этот «указатель» не мешал нам при разгадывании задачи (ведь вводить числа нам будет неза-

чем), проверяем, нажата ли кнопка *sbtDraw*. Если нажата, то указатель убираем.

```
//ПЕРЕМЕЩАТЬ ПОЗИЦИЮ ВВОДА В ВЕРХНЕМ ЧИСЛОВОМ ПОЛЕ
procedure TForm1.dgColsNumMouseMove(Sender: TObject; Shift:
TShiftState; X,
  Y: Integer);
var ACol, ARow: Integer;
  r: TRECT;
  text: string;
begin
  //ACol, ARow <-- клетка с курсором
  dgColsNum.MouseToCell(x,y,ACol, ARow);
  //если курсор переместился на другую клетку -
  if (ACol<>cellMouseTop.x) or (ARow<>cellMouseTop.y) then
  begin
    //установить фокус ввода на верхнем поле:
    dgColsNum.SetFocus;
    //перерисовать предыдущую клетку с курсором:
    r:= dgColsNum.CellRect(cellMouseTop.x, cellMouseTop.y);
    dgColsNumDrawCell(self, cellMouseTop.x, cellMouseTop.y, R,
[]);
    //выделить красным цветом текущую клетку:
    if not sbtDraw.Down then begin
      r:= dgColsNum.CellRect(ACol, ARow);
      InflateRect(r,-1,-1);
      dgColsNum.Canvas.Brush.Color:= RGB(255,0,0);
      dgColsNum.Canvas.FillRect(R);
      //восстановить число в клетке:
      text:= masColsNum[ACol, ARow].sNum;
      with R, dgColsNum.Canvas do
       textrect(R, left+(right-left-textwidth(text)) div 2,
                top+(bottom-top-textheight(text)) div 2, text);
    end;
  end;
  //новая позиция ввода:
  cellMouseTop.x:= ACol;
  cellMouseTop.y:= ARow;
end; // dgColsNumMouseMove

//ПЕРЕМЕЩАТЬ ПОЗИЦИЮ ВВОДА В ЛЕВОМ ЧИСЛОВОМ ПОЛЕ
procedure TForm1.dgRowsNumMouseMove(Sender: TObject; Shift:
TShiftState; X,
  Y: Integer);
var ACol, ARow: Integer;
  r: TRECT;
  text: string;
begin
  //ACol, ARow <-- клетка с курсором
  dgRowsNum.MouseToCell(x,y,ACol, ARow);
  if (ACol<>cellMouseLeft.x) or (ARow<>cellMouseLeft.y) then
```

```
  begin
    //установить фокус на левом поле:
    dgRowsNum.SetFocus;
    //перерисовать предыдущую клетку с курсором:
    r:= dgRowsNum.CellRect(cellMouseLeft.x, cellMouseLeft.y);
    dgRowsNumDrawCell(self, cellMouseLeft.x, cellMouseLeft.y, R,
[]);
    //выделить красным цветом текущую клетку:
    if not sbtDraw.Down then begin
      r:= dgRowsNum.CellRect(ACol, ARow);
      InflateRect(r,-1,-1);
      dgRowsNum.Canvas.Brush.Color:= RGB(255,0,0);
      dgRowsNum.Canvas.FillRect(R);
      text:= masRowsNum[ACol, ARow].sNum;
      with R, dgRowsNum.Canvas do
       textrect(R, left+(right-left-textwidth(text)) div 2,
                top+(bottom-top-textheight(text)) div 2, text);
    end;
  end;
  cellMouseLeft.x:= ACol;
  cellMouseLeft.y:= ARow;
end; // dgRowsNumMouseMove
```

Запустите программу и погуляйте с курсором по *числовым* полям – красная клетка, как верный пёс, всюду будет следовать за вами (Рис. Ф4.29).

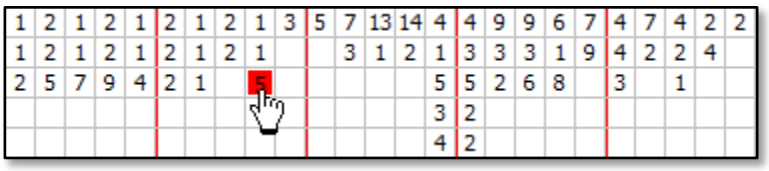


Рис. Ф4.29. Красная метка

Немного отдохнув, опять беритесь за дело. Для ввода чисел мы напишем процедуру, обрабатывающую событие *OnKeyPress*, возникающее при нажатии клавиши на клавиатуре. Прежде всего, проверим, не нажата ли кнопка *sbtDraw*, чтобы случайно не изменить числа при решении задачи. Затем нам нужно убедиться, что нажата именно *цифровая* клавиша, иначе возникнет ошибка при обработке введённых символов. Ну и последнее. Для подстраховки от неожиданных судорог и тиков мы ограничим длину вводимого числа *двумя* цифрами. Если всё в порядке, сохраняем число в массиве.

```
//ВВЕСТИ ЧИСЛО В ТЕКУЩУЮ КЛЕТКУ
procedure TForm1.dgColsNumKeyPress(Sender: TObject; var Key:
Char);
var
  s, text: string;
```

```
  rect: TRect;
begin
  if sbtDraw.Down then exit;
  if not(key in['0'..'9']) then exit;
  text:= Key;
  //если что-то уже есть в этой клетке -
  s:= masColsNum[cellMouseTop.x, cellMouseTop.y].sNum;
  //числа могут быть только однозначные и двузначные:
  if length(s)>=2 then exit;
  //записать число в клетку:
  masColsNum[cellMouseTop.x, cellMouseTop.y].sNum:= s+Key;
  rect:= dgColsNum.CellRect(cellMouseTop.x, cellMouseTop.y);
  dgColsNumDrawCell(self, cellMouseTop.x, cellMouseTop.y, Rect,
[]);
end; // dgColsNumKeyPress

//ВВЕСТИ ЧИСЛО В ЯЧЕЙКУ ЛЕВОГО ЧИСЛОВОГО ПОЛЯ
procedure TForm1.dgRowsNumKeyPress(Sender: TObject; var Key:
Char);
var
  s, text: string;
  rect: TRect;
begin
  if sbtDraw.Down then exit;
  if not(key in['0'..'9']) then exit;
  text:= Key;
  s:= masRowsNum[cellMouseLeft.x, cellMouseLeft.y].sNum;
  if length(s)>=2 then exit;
  masRowsNum[cellMouseLeft.x, cellMouseLeft.y].sNum:= s+Key;
  rect:= dgRowsNum.CellRect(cellMouseLeft.x, cellMouseLeft.y);
  dgRowsNumDrawCell(self, cellMouseLeft.x, cellMouseLeft.y, Rect,
[]);
end; // dgRowsNumKeyPress
```

А что делать, если вы запутались в цифрах? – Не мудрствуя лукаво, мы просто сотрём всё, что было в клетке, дважды щёлкнув на ней. Опять же для подстраховки проверяем, не решаем ли мы уже задачу, чтобы не усложнять себе и без того непростую жизнь.

```
//ОЧИСТИТЬ ТЕКУЩУЮ КЛЕТКУ
procedure TForm1.dgColsNumDblClick(Sender: TObject);
var Rect: TRECT;
begin
  if sbtDraw.Down then exit;
  masColsNum[cellMouseTop.x, cellMouseTop.y].sNum:='';
  rect:= dgColsNum.CellRect(cellMouseTop.x, cellMouseTop.y);
  dgColsNumDrawCell(self, cellMouseTop.x, cellMouseTop.y, Rect,
[]);
end;

//ОЧИСТИТЬ ТЕКУЩУЮ КЛЕТКУ
```

```
procedure TForm1.dgRowsNumDblClick(Sender: TObject);
var Rect: TRECT;
begin
  if sbtDraw.Down then exit;
  masRowsNum[cellMouseLeft.x, cellMouseLeft.y].sNum:='';
  rect:= dgRowsNum.CellRect(cellMouseLeft.x, cellMouseLeft.y);
  dgRowsNumDrawCell(self, cellMouseLeft.x, cellMouseLeft.y, Rect,
[]);
end;
```

Для проверки качества своей работы возьмём да и введём третью задачу из газеты *Поле чудес*, №3 за 1996 год. Это один из первых японских кроссвордов, появившихся у нас. Впрочем, вы можете взять и любой другой.

Запустите программу, нажмите самую первую кнопку (*sbtNewFig*) и в появившемся окне установите размеры всех трёх полей. Для нашего примера цифирь должна получиться такая (Рис. Ф4.30).

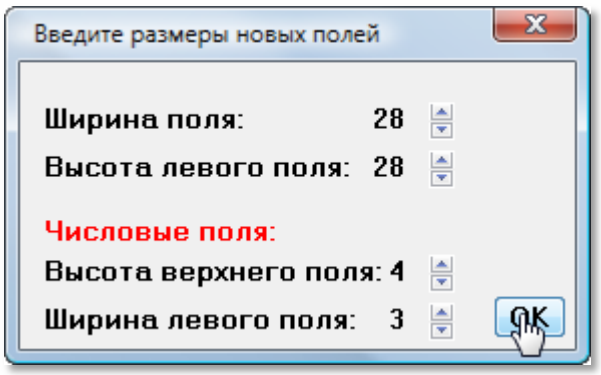


Рис. Ф4.30. Устнаваливаем размеры сеток

Нажав кнопку *ОК*, вы с удовольствием убедитесь, что игровое и числовые поля ужались и разжались как им велено. Теперь подводите красную метку к нужным клеткам (как сказал бы киношный Портос, *я вас умоляю*, не обижайте мышку, не кликайте ею всуе) и нажимайте на клавиатуре цифровые клавиши. Числа тут же появятся в клетках.

Как мы уже договорились, числа должны записываться в числовых полях от *верхнего* и *левого* краёв. А вот в газетах они, напротив, чаще записаны от *нижнего* и *правого*. Надо сказать, ввод чисел - удовольствие не из приятных, и вряд ли стоит усугублять его ещё и перетаскиванием чисел из одних клеток в другие. Поэтому мы сделаем для себя послабление и заставим программу гонять числа по полям, сами же будем спокойно вписывать числа в клетки, не придерживаясь твёрдых правил. Важно только сохранить последовательность чисел в рядах, а пустые клетки вы можете рас-

ставлять как угодно. Если у вас под рукой не оказалось первоисточника, то поверьте, что в итоге должна получиться такая «таблица» (Рис. Ф4.31).

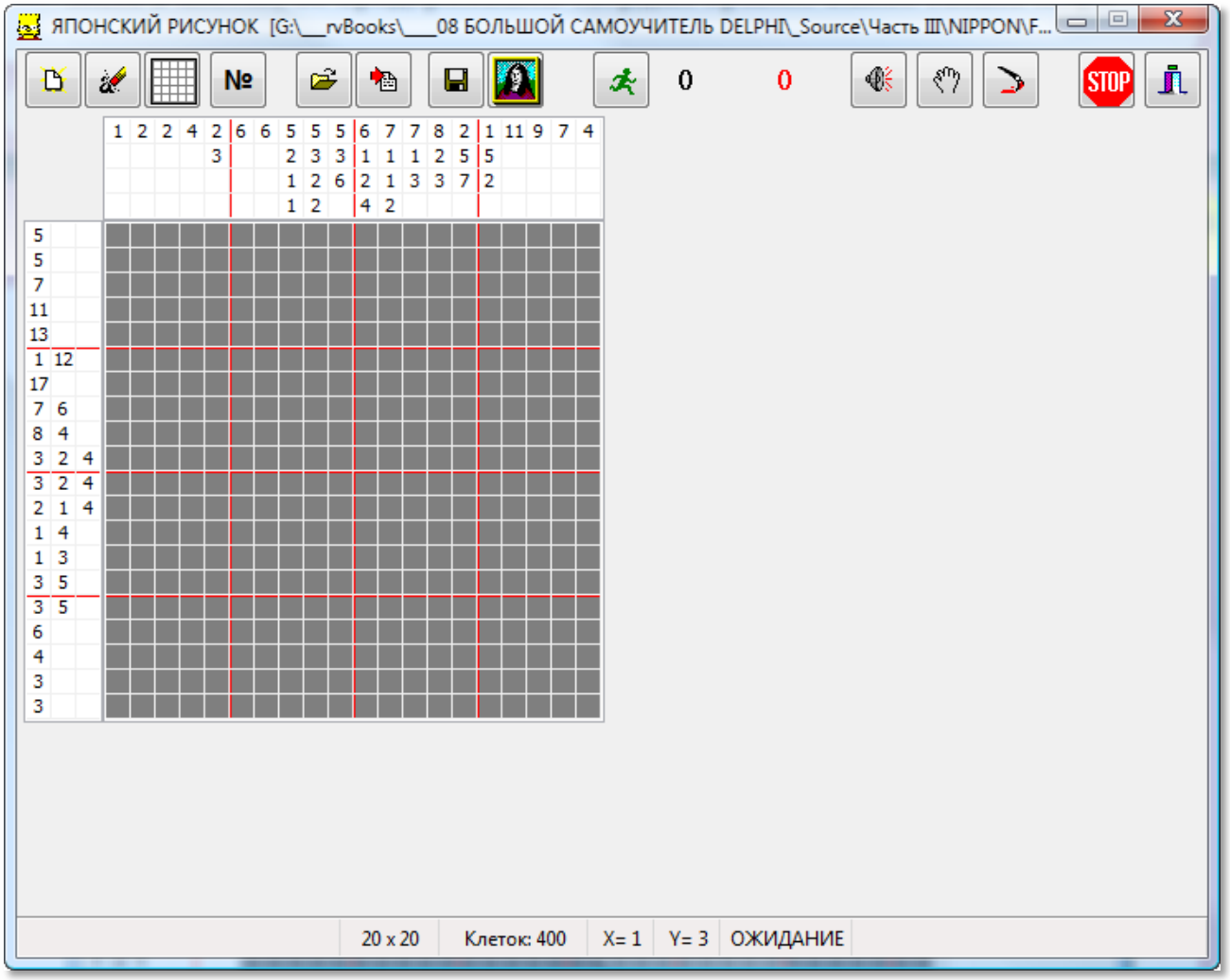


Рис. Ф4.31. Все числа – в клетках!

Этот способ ввода чисел, пожалуй, самый наглядный – легко сравнить задачу в газете с её «электронной копией», найти и исправить ошибки. Если же вам претит катать мышку по коврику, то вы можете набрать числа в обыкновенном текстовом редакторе, хотя бы в *Блокноте* или *WordPad'е*, которые входят в комплект операционной системы *Windows*.

Возможны *два* варианта создания текстовых файлов с условиями задачи. *Первый* - числа в строке должны отделяться друг от друга, по крайней мере, одним пробелом. *Второй* - числа пишутся слитно.

Рассмотрим сначала «пробельный» вариант. Формат файла такой. В *первой* строке - два числа: первое показывает *ширину* игрового поля, второе – его *высоту*. Во *второй* строке также два числа: первое задаёт *высоту верхнего*

числового поля, второе – *ширину левого* числового поля. Затем записываются данные для *верхнего* числового поля. Каждая строка содержит числа из одной колонки. После чего аналогично записываются данные для *левого* числового поля, но в строках перечисляются числа из горизонтальных рядов. Например, информация о таком кроссворде (Рис. Ф4.32)

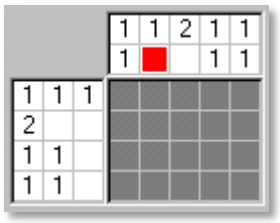


Рис. Ф4.32. «Образцовый» рисунок!

в текстовом исполнении будет выглядеть так:

```
5 4
2 3
1 1
1
2
1 1
1 1
1 1 1
2
1 1
1 1
```

Закончив ввод чисел, запишите файл как обычный текст без форматирования, а затем замените его расширение *TXT* другим - *JPN*. Конечно, можно оставить и *TXT*, но тогда в файл придётся вводить дополнительную информацию, чтобы отличить его при загрузке от обычного текста, иначе такие попытки будут приводить к ошибкам.

Чтобы не нажимать беспрестанно на клавишу *пробел*, вы можете воспользоваться другим способом ввода задач с клавиатуры. Формат файла в этом случае несколько иной (он используется в программе Гантверга и Самсонова *Soluter*, так что все претензии адресуйте им). *Первая* строка хранит *высоту* и *ширину* игрового поля. Информация о размерах числовых полей умалчивается. В следующих строках записаны числа – сначала из *левого* числового поля, затем из *верхнего*. Последовательность чисел в строках та же, что и раньше, но набираются они без пробелов:

```
45
111
2
11
```

```
11
11
1
2
11
11
```

Не забудьте изменить расширение файла на *JCP*.

Когда все числа в кроссворде меньше 10, этот формат, безусловно, удобнее для ввода задач. А вот б***о***льшие числа создают определённые трудности, так как число 10, например, при слитной записи будет трактоваться как два числа – 1 и 0. Так что придётся каждое число обозначать своим значком. В принципе, это делается просто: за цифрой *9* в таблице символов следует двоеточие - :, которое и будет заменять число 10. Следующие числа «шифруются» так:

```
11 - ;
12 - <
13 - =
14 - >
15 - ?
16 - @
17 - A
18 - B
19 - D
20 - E
```

И так далее. За всё надо платить - убрав пробелы, мы получили множество значков, в которых легко запутаться. Именно поэтому и был разработан первый формат текстового файла, который больше не используется ни в каких других программах, кроме нашей. Но вы можете придумать и что-нибудь своё – кто вам запретит!

Мотаем на ус, или Наш задачник

Наливай!

Из нашего

Вываливай!

Тоже

Теперь – тем или иным способом – вы сможете перенести задачу в компьютер. Встречаются же японские кроссворды почти в каждой головоломной газете. *Поле чудес*, *Магазин кроссвордов*, *Загадочная газета*, *Фаворит*, *Смекай!*, *Японский лабиринт* и многие другие редко обходятся без них. Имеются издания «с узкой специализацией», в которых ничего другого и нет. Например, газеты *Фудзияма*, *Японские кроссворды*, *Японский сканворд*. Но вы можете воспользоваться и готовыми задачами, которые прилагаются к уже упомянутым программам для самостоятельного решения. Причём большинство из них можно загрузить в нашу программу и решать непосредственно в ней. Правда, для этого нам сначала придётся создать необходимые процедуры для обработки «нестандартных» файлов. Так как это не относится непосредственно к нашей программе, то мы не будем подробно останавливаться на всех тонкостях чуждых нам форматов. При желании каждому из вас по силам самостоятельно в этом разобраться (а это будет и небесполезно!). Пока же просто дважды щёлкните по кнопке *sbtLoadFig* и заполните заготовку процедуры следующим кодом:

```
//ЗАГРУЗИТЬ НОВУЮ ФИГУРУ
procedure TForm1.sbtLoadFigClick(Sender: TObject);
var
  s: string;
  F: TextFile;
  ss: array[1..MAX_POLE_WIDTH + MAX_POLE_HEIGHT] of string;
  nLines: integer;  //счётчик считанных строк фигуры
  maxLen: integer;
  i,j:integer;
  flgComm: boolean; //= TRUE, если считываем комментарий к фигуре
  w,h: integer;

  //ЗАГРУЗИТЬ ФАЙЛ JC {файл картинки программы Gun's Japanese Crossword)
  procedure LoadJcFile;
  var
    F: file of Byte;
    w,h: integer;
    b: byte;
    i,j:integer;
  begin
    {$i-}
    AssignFile(F, NameFig);
    Reset(F);
    {$i+}
```

```
  if IOResult<>0 then begin   {ошибка при загрузке файла}
    application.MessageBox ('Такой фигуры нет!',NAME_PROG, MB_OK);
    exit
  end;
  //вывести в заголовке формы имя загруженного файла:
  form1.caption:= NAME_PROG + '  [' + NameFig + ']';
  //начинаем считывать файл
  //считываем размеры поля
  //ширина поля - 4-ый байт:
  Seek(F, 4); Read(F, b); w:= b;
  //высота поля - 6-ой байт:
  Seek(F, 6); Read(F, b); h:= b;
  //проверить размеры фигуры:
  if (h > MAX_POLE_HEIGHT) or (w > MAX_POLE_WIDTH)then begin
    application.MessageBox('Слишком большая фигура!',NAME_PROG, MB_OK);
    CloseFile(F); exit
  end;
  POLE_WIDTH:= w; POLE_HEIGHT:= h;
  //проверяем длину файла:
  i:= FileSize(f);
  j:= 8 + POLE_HEIGHT * POLE_WIDTH;
  if i < j then begin
    application.MessageBox('Неверный размер файла!',NAME_PROG, MB_OK);
    CloseFile(F); exit
  end;
  //загружаем данные для поля:
  for j:= 0 to POLE_HEIGHT-1 do
    for i:= 0 to POLE_WIDTH-1 do begin
      Seek(F, POLE_WIDTH*j+i + 8); Read(F, b);
       if b=0 then masPole[i,j]:= NUM_WHITE
      else masPole[i,j]:= NUM_BLACK;
    end;
  //закрыть файл - всё загружено:
  CloseFile(F);
  //оцифровать рисунок:
  sbtNumbersClick(Self);
  //очистить поле:
  for j:= 0 to POLE_HEIGHT-1 do
    for i:= 0 to POLE_WIDTH-1 do masPole[i,j]:= NUM_GRAY;
  Prepare(POLE_WIDTH,POLE_HEIGHT, TOP_POLE_HEIGHT, LEFT_POLE_WIDTH);
end; //LoadJcFile

//ЗАГРУЗИТЬ ФАЙЛ JCW
procedure LoadJcwFile;
var
  F: file of Byte;
  i,j,n:integer;
  w,h: integer;
  b, pred: byte;
  bi, predi: integer;
  off: integer;
  ng: integer; //число групп в ряду
  p: TPole;
begin
  {$i-}
  AssignFile(F, NameFig);
  Reset(F);
```

```
    {$i+}
    if IOResult<>0 then begin   {ошибка при загрузке файла}
      application.MessageBox ('Такой фигуры нет!',NAME_PROG, MB_OK);
      exit
    end;
    //вывести в заголовке формы имя загруженного файла:
    form1.caption:= NAME_PROG + '  [' + NameFig + ']';
    //начинаем считывать файл
    //считываем размеры поля
    //ширина поля - первый байт:
    Seek(F, 0); Read(F, b); w:= b;
    //высота поля - второй байт:
    Seek(F, 1); Read(F, b); h:= b;
    //проверить размеры фигуры:
    if (h > MAX_POLE_HEIGHT) or (w > MAX_POLE_WIDTH)then begin
      application.MessageBox('Слишком большая фигура!',NAME_PROG, MB_OK);
      CloseFile(F); exit
    end;
    POLE_WIDTH:= w; POLE_HEIGHT:= h;
    //проверяем длину файла:
    i:= FileSize(f);
    j:= 2 + POLE_HEIGHT * POLE_WIDTH;
    if i < j then begin
      application.MessageBox('Неверный размер файла!',NAME_PROG, MB_OK);
      CloseFile(F); exit
    end;
    //считываем данные для верхнего числового поля:
    off:= 2;
    h:=0; //высота верхнего числового поля
    for i := 0 to POLE_WIDTH -1 do //- по длине поля
    begin
      ng:= 0;   //число групп в ряду
      n:= 0;    //- длина группы
      pred:= 0; //- предыдущий байт
      //в каждом ряду должно быть хотя бы одно число:
      masColsNum[i,0].sNum:= '0';
      //для каждого столбца:
      for j := 0 to POLE_HEIGHT -1 do
      begin
        Seek(F, POLE_HEIGHT*i+j + off);
        //считать очередной байт:
        Read(F, b);
        //сохранить его:
        p[i,j]:= b;
        if b=2 then begin       //- чёрная клетка
          if pred<>2 then begin //- предыдущая клетка не чёрная -->
            //записать группу:
            masColsNum[i,ng-1].sNum:= inttostr(n);
            //начинается новая группа:
            inc(ng);
            n:= 0;
          end;
          //увеличить длину группы:
          inc(n);
        end;
        pred:= b;
      end;
```

```
    masColsNum[i,ng-1].sNum:= inttostr(n);
    masColsNum[i,-1].Num:= ng;
    if ng> h then h:= ng;
  end;
  //закрыть файл - всё загружено:
  CloseFile(F);
  //формируем данные для левого числового поля:
  w:=0; //ширина левого числового поля
  for j := 0 to POLE_HEIGHT -1 do //- по всем строкам поля
  begin
    ng:= 0;     //число групп в ряду
    n:= 0;      //- длина группы
    predi:= 0; //- предыдущий байт
    //в каждом ряду должно быть хотя бы одно число:
    masRowsNum[0,j].sNum:= '0';
    for i := 0 to POLE_WIDTH -1 do //- по длине строки
    begin
      //считать очередное число:
      bi:= p[i,j];
      if bi=2 then begin //- чёрная клетка
        if predi<>2 then begin //- предыдущая клетка не чёрная -->
          //записать группу:
          masRowsNum[ng-1,j].sNum:= inttostr(n);
          //начинается новая группа:
          inc(ng);
          n:= 0;
        end;
        //увеличить длину группы:
        inc(n);
      end;
      predi:= bi;
    end; // i
    masRowsNum[ng-1,j].sNum:= inttostr(n);
    masRowsNum[-1,j].Num:= ng;
    if ng> w then w:= ng;
  end; //j
  //проверить число групп:
  if (w > MAX_CELLNUM) or (h > MAX_CELLNUM) then begin
    application.MessageBox('Слишком много групп!',NAME_PROG, MB_OK);
    CloseFile(F); exit
  end;
  TOP_POLE_HEIGHT:= h;
  LEFT_POLE_WIDTH:= w;
  //вывести картинку на игровое поле:
  for j:= 0 to POLE_HEIGHT-1 do
    for i:= 0 to POLE_WIDTH-1 do
      case p[i,j] of
        2: masPole[i,j]:= NUM_BLACK;
        else masPole[i,j]:= NUM_WHITE;
      end;
  //вывести поля заданных размеров с числами:
  Prepare(POLE_WIDTH,POLE_HEIGHT, TOP_POLE_HEIGHT, LEFT_POLE_WIDTH);
end;// LoadJcwFile

//ЗАГРУЗИТЬ ФАЙЛ JPN
procedure LoadJpnFile;
var
```

```
    F: TextFile;
    i:integer;
    w,h: integer;
    ng: integer;   //число групп в ряду
    s: string;
    pos: integer;  //позиция в строке
    sNum: string;  //число
    //получить число из заданной строки
    function GetNumber(s: string; var pos: integer; var s2: string):
boolean;
    //= FALSE, если строка ещё не кончилась
    begin
      s2:= '';
      if pos> Length(s) then Result:= TRUE else Result:= FALSE;
      while (pos<= Length(s)) and (s[pos]<> ' ') do begin
        s2:= s2 + s[pos]; inc(pos)
      end;
      //пропустить пробелы:
      while (pos< Length(s)) and (s[pos]= ' ') do inc(pos);
      if (pos= Length(s)) and (s[pos]= ' ') then Result:= TRUE;
    end;
  begin
    {$i-}
    AssignFile(F, NameFig);
    Reset(F);
    {$i+}
    if IOResult<>0 then begin   {ошибка при загрузке файла}
      application.MessageBox ('Такой фигуры нет!',NAME_PROG, MB_OK);
      exit
    end;
    //вывести в заголовке формы имя загруженного файла:
    form1.caption:= NAME_PROG + '  [' + NameFig + ']';
    //начинаем считывать файл
    //считываем размеры поля
    //ширина поля - первое число
    //высота поля - второе:
    Readln(F, S);
    pos:= 1;
    GetNumber(S, pos, sNum); w:= strtoint(sNum);
    GetNumber(S, pos, sNum); h:= strtoint(sNum);
    //проверить размеры фигуры:
    if (h > MAX_POLE_HEIGHT) or (w > MAX_POLE_WIDTH)then begin
      application.MessageBox('Слишком большая фигура!',NAME_PROG, MB_OK);
      CloseFile(F);
      exit
    end;
    POLE_WIDTH:= w; POLE_HEIGHT:= h;
    //считываем размеры числовых полей
    //высота верхнего числового поля первое число,
    //ширина левого числового поля - второе:
    Readln(F, S);
    pos:= 1;
    GetNumber(S, pos, sNum); h:= strtoint(sNum);
    GetNumber(S, pos, sNum); w:= strtoint(sNum);
    //проверить число групп:
    if (w > MAX_CELLNUM) or (h > MAX_CELLNUM) then begin
      application.MessageBox('Слишком много групп!',NAME_PROG, MB_OK);
```

```
      CloseFile(F); exit
    end;
    TOP_POLE_HEIGHT:= h;
    LEFT_POLE_WIDTH:= w;
    //считываем данные для верхнего числового поля:
    For i:= 0 to POLE_WIDTH-1 do begin
      if eof(f) then begin
        s:='Не хватает данных!';
        application.MessageBox(PChar(s),NAME_PROG,MB_OK );
        exit
      end;
      //считать строку из файла:
      Readln(F, S);
      //выделить числа из строки:
      pos:= 1; ng:= -1;
      while GetNumber(S, pos, sNum)= FALSE do begin
        inc(ng);
        masColsNum[i, ng].sNum:= sNum;
      end;
    end;
    //считываем данные для левого числового поля:
    For i:= 0 to POLE_HEIGHT-1 do begin
      if eof(f) then begin
        s:='Не хватает данных!';
        application.MessageBox(PChar(s),NAME_PROG,MB_OK );
        exit
      end;
      //считать строку из файла:
      Readln(F, S);
      //выделить числа из строки:
      pos:= 1; ng:= -1;
      while GetNumber(S, pos, sNum)= FALSE do begin
        inc(ng);
        masRowsNum[ng,i].sNum:= sNum;
      end;
    end;
  //закрыть файл - всё загружено:
  CloseFile(F);
  //вывести поля заданных размеров с числами:
  Prepare(POLE_WIDTH,POLE_HEIGHT, TOP_POLE_HEIGHT, LEFT_POLE_WIDTH);
end;// LoadJpnFile

  //ЗАГРУЗИТЬ ФАЙЛ JPC из программы JpcWin Матвея Ильяшенко
  procedure LoadJpcFile;
  var
    F: file of Byte;
    i,j,n:integer;
    w,h: integer;
    b: byte;
    off: integer;
    function ReadWord(index: Longint): word;
    var
      b1, b2: byte;
    begin
      //считать первый байт:
      Seek(F, index); Read(F, b1);
      //считать второй байт:
```

```
      Seek(F, index+1); Read(F, b2);
      //вычислить слово:
      Result:= 256*b2+b1;
    end;
  begin
    //загрузить файл:
    {$i-}
    AssignFile(F, NameFig);
    Reset(F);
    {$i+}
    if IOResult<>0 then begin   {ошибка при загрузке файла}
      application.MessageBox ('Такой фигуры нет!',NAME_PROG, MB_OK);
      exit
    end;
    //вывести в заголовке формы имя загруженного файла:
    form1.caption:= NAME_PROG + '  [' + NameFig + ']';
    //начинаем считывать файл -
    //считываем размеры поля:
    w:= ReadWord(0);
    h:= ReadWord(2);
    //проверить размеры фигуры:
    if (h > MAX_POLE_HEIGHT) or (w > MAX_POLE_WIDTH)then begin
      application.MessageBox('Слишком большая фигура!',NAME_PROG, MB_OK);
      CloseFile(F); exit
    end;
    POLE_WIDTH:= w; POLE_HEIGHT:= h;
     //считываем размеры числовых полей:
    h:= ReadWord(4);
    w:= ReadWord(6);
    //проверить число групп:
    if (w > MAX_CELLNUM) or (h > MAX_CELLNUM) then begin
      application.MessageBox('Слишком много групп!',NAME_PROG, MB_OK);
      CloseFile(F); exit
    end;
    TOP_POLE_HEIGHT:= h;
    LEFT_POLE_WIDTH:= w;
    //проверяем длину файла:
    i:= FileSize(f);
    j:= 8+POLE_HEIGHT*LEFT_POLE_WIDTH + POLE_WIDTH* TOP_POLE_HEIGHT;
    if i<>j then begin
      application.MessageBox('Неверный размер файла!',NAME_PROG, MB_OK);
      CloseFile(F); exit
    end;
    //считываем данные для левого числового поля:
    off:= 8;
    for j := 0 to POLE_HEIGHT -1 do //сверху
    begin
      n:= 0;
      //в каждом ряду должно быть хотя бы одно число:
      masRowsNum[0,j].sNum:= '0';
      for i := LEFT_POLE_WIDTH - 1 downto 0 do //справа налево
      begin
        Seek(F, POLE_HEIGHT*i+j+ off);
        Read(F, b);
        if b<>0 then begin
          masRowsNum[n,j].sNum:= inttostr(b);
          inc(n);
```

```
      end;
    end;
  end;
  //считываем данные для верхнего числового поля:
  off:= LEFT_POLE_WIDTH * POLE_HEIGHT + 8;
  for i := 0 to POLE_WIDTH -1 do
  begin
    n:= 0;
    //в каждом ряду должно быть хотя бы одно число:
    masColsNum[i,0].sNum:= '0';
    for j := TOP_POLE_HEIGHT - 1 downto 0 do //снизу вверх
    begin
      Seek(F, POLE_WIDTH* j+ i+ off);
      Read(F, b);
      if b<>0 then begin
        masColsNum[i,n].sNum:= inttostr(b);
        inc(n);
      end;
    end;
  end;
  //закрыть файл - всё загружено:
  CloseFile(F);
  //вывести поля заданных размеров с числами:
  Prepare(POLE_WIDTH,POLE_HEIGHT, TOP_POLE_HEIGHT, LEFT_POLE_WIDTH);
end;// LoadJpcFile

//ЗАГРУЗИТЬ ФАЙЛ JCR из программы "Японский кроссворд 2000"
//Егоркина И.В.
procedure LoadJcrFile;
var
  F: file of Byte;
  i,j,n:integer;
  w,h: integer;
  b: byte;
  off: integer;
begin
  //загрузить файл:
  {$i-}
  AssignFile(F, NameFig);
  Reset(F);
  {$i+}
  if IOResult<>0 then begin   //ошибка при загрузке файла
    application.MessageBox ('Такой фигуры нет!',NAME_PROG, MB_OK);
    exit
  end;
  //вывести в заголовке формы имя загруженного файла:
  form1.caption:= NAME_PROG + '  [' + NameFig + ']';
  //начинаем считывать файл -
  //считываем размеры поля:
  Seek(F, 0); Read(f,b); w:= b;
  Seek(F, 1); Read(f,b); h:= b;
  //проверить размеры фигуры:
  if (h > MAX_POLE_HEIGHT) or (w > MAX_POLE_WIDTH)then begin
    application.MessageBox('Слишком большая фигура!',NAME_PROG, MB_OK);
    CloseFile(F); exit
  end;
  POLE_WIDTH:= w; POLE_HEIGHT:= h;
```

```
    //считываем размеры числовых полей:
    Seek(F, 2); Read(f,b); w:= b;
    Seek(F, 3); Read(f,b); h:= b;
    //проверить число групп:
    if (w > MAX_CELLNUM) or (h > MAX_CELLNUM) then begin
      application.MessageBox('Слишком много групп!',NAME_PROG, MB_OK);
      CloseFile(F); exit
    end;
    TOP_POLE_HEIGHT:= h;
    LEFT_POLE_WIDTH:= w;
    //проверяем длину файла:
    i:= FileSize(f);
    j:= 4+POLE_HEIGHT*LEFT_POLE_WIDTH + POLE_WIDTH* TOP_POLE_HEIGHT;
    if i<>j then begin
      application.MessageBox('Неверный размер файла!',NAME_PROG, MB_OK);
      CloseFile(F); exit
    end;
    //считываем данные для верхнего числового поля:
    off:= 4;
    for i := 0 to POLE_WIDTH -1 do
    begin
      n:= 0;
      //в каждом ряду должно быть хотя бы одно число:
      masColsNum[i,0].sNum:= '0';
      for j := TOP_POLE_HEIGHT - 1 downto 0 do //снизу вверх
      begin
        Seek(F, TOP_POLE_HEIGHT*i+j+ off);
        Read(F, b);
        if b<>0 then begin
          masColsNum[i,n].sNum:= inttostr(b);
          inc(n);
        end;
      end;
    end;
    //считываем данные для левого числового поля:
    off:= TOP_POLE_HEIGHT * POLE_WIDTH + 4;
    for j := 0 to POLE_HEIGHT -1 do //сверху вниз
    begin
      n:= 0;
      for i := LEFT_POLE_WIDTH - 1 downto 0 do
      begin
        Seek(F, LEFT_POLE_WIDTH *j+i+ off);
        Read(F, b);
        if b<>0 then begin
          masRowsNum[n,POLE_HEIGHT -1-j].sNum:= inttostr(b);
          inc(n);
        end;
      end;
    end;
    //закрыть файл - всё загружено:
    CloseFile(F);
    Prepare(POLE_WIDTH,POLE_HEIGHT, TOP_POLE_HEIGHT, LEFT_POLE_WIDTH);
  end;// LoadJcrFile

begin
  if status='ПОИСК' then exit;
  //"прокручиваем" все поля в начало:
```

```
dgPole.LeftCol:= 0; dgPole.TopRow:= 0;
dgColsNum.LeftCol:= 0; dgColsNum.TopRow:= 0;
dgRowsNum.LeftCol:= 0; dgRowsNum.TopRow:= 0;
//число разгаданных клеток = 0:
lblReady.Caption:='0';
//начнём решение задачи с нулевого уровня:
lblLevel.Caption:='0';
flgComm:= false;
frmMemo.memo1.Clear;
frmMemo.memo1.SetFocus;
//файлы по умолчанию имеют расширение jcp:
opendialog1.DefaultExt:='jcp';
opendialog1.Filter:=
  'Japan puzzle(*.jcp, *.jpc, *.jcw, *.jpn, *.jc, *.jcr)|'+
  '*.JCP;*.JPC;*.JCW; *.JPN; *.JC; *.JCR';
//ищем файлы в каталоге 'FIGURE':
s:=extractfilepath(application.exename)+'FIGURE\';
opendialog1.InitialDir:= s;
opendialog1.Title:='Загрузите новую фигуру';
if opendialog1.Execute then begin
  //очистить игровое и числовые поля:
  sbtClearPoleClick(self);
  //выбрали файл с именем NameFig=s:
  s:= opendialog1.filename;
  NameFig:=s;
  //файл формата JPC
  s:= ExtractFileExt(NameFig);
  if s= '.jpc' then begin LoadJpcFile; exit end
    //файл формата JCW - задания для самостоятельного решения
    //из программы Романа Гантверга и Дмитрия Самсонова Japan Crossword
    //Soluter (Японский кроссворд - Решатель), 1999
  else if s= '.jcw' then begin LoadJcwFile; exit end
    //обычный текстовый файл:
  else if s= '.jpn' then begin LoadJpnFile; exit end
    //файл картинки программы Gun's Japanese Crossword:
  else if s= '.jc' then begin LoadJcFile; exit end
  //файл задачи из программы "Японский кроссворд 2000":
  else if s= '.jcr' then begin LoadJcrFile; exit end;
  {$i-}
  AssignFile(F,NameFig);
  Reset(F);
  {$i+}
  if IOResult<>0 then begin  //ошибка при загрузке файла
    application.MessageBox ('Такой фигуры нет!',NAME_PROG, MB_OK);
    exit
    end;
  //начинаем считывать файл:
  nLines:=0; //считано строк фигуры
  while not eof(f) do begin
    //считать строку из файла:
    Readln(F, S);
    //комментарий к фигуре?
    if ((length(s)>1) and (s[1]='/') and (s[2]='/')) or
      (flgComm= true) then    //- комментарий
    begin
      if flgComm=false then   //- начало комментария
      begin
```

```
          flgComm:= true;
          s:=copy(s,3,length(s));
          frmMemo.memo1.Lines.Text:=s;
        end
        else
          frmMemo.memo1.Lines.Text:=frmMemo.memo1.Lines.Text+#10+s;
      end
      else begin //- строка фигуры
        inc(nLines);
        //сохранить строку в массиве:
        ss[nLines]:=s;
      end
    end;
    //закрыть файл - всё загружено:
    CloseFile(F);
    //размеры игрового поля -
    //высота поля в клетках:
    h:= ord(ss[1][1])-ord('0');
    //ширина поля в клетках:
    w:= ord(ss[1][2])-ord('0');
    //проверить размеры фигуры:
    if (h > MAX_POLE_HEIGHT) or (w > MAX_POLE_WIDTH)then begin
      application.MessageBox('Слишком большая фигура!',NAME_PROG, MB_OK);
      exit
    end;
    //количество строк для числовых полей должно соответствовать
    //размерам поля:
    if nLines-1 <> h + w then begin
      application.MessageBox('Неверное   количество   данных!',NAME_PROG,
MB_OK);
      exit
    end;
    //занести данные в masColsNum, masRowsNum:
    maxLen:=0;
    for j:=2 to h+1 do begin
      s:=ss[j];  //- очередная строка из массива
      //по длине очередной строки -
      for i:=1 to length(s) do begin
        if length(s)> maxLen then maxLen:= length(s);
        //проверить количество групп чисел в строке:
        if maxLen > MAX_CELLNUM then begin
          application.MessageBox('Слишком    много    групп!',NAME_PROG,
MB_OK);
          //очистить masRowsNum[]:
          Clear_masRowsNum;
          exit
        end;
        //занести число в массив:
        masRowsNum[i-1,j-2].sNum:= inttostr(ord(s[i])-ord('0'));
      end;
    end;
    //макс. число групп в левом числовом поле:
    LEFT_POLE_WIDTH:= maxLen;
    //начинаем заполнение массива верхнего числового поля:
    maxLen:=0;
    for j:=h+2 to nLines do begin
      //очередная строка:
```

```
      s:=ss[j];
      //по длине очередной строки -
      for i:=1 to length(s) do begin
        if length(s)> maxLen then maxLen:= length(s);
        //проверить количество групп чисел в строке:
        if maxLen > MAX_CELLNUM then begin
          application.MessageBox('Слишком    много    групп!',NAME_PROG,
MB_OK);
          //очистить masColsNum[]:
          Clear_masColsNum;
          exit
        end;
        //занести число в массив:
        masColsNum[j-h-2,i-1].sNum:= inttostr(ord(s[i])-ord('0'));
      end;
    end;
    //макс. число групп в верхнем числовом поле:
    TOP_POLE_HEIGHT:= maxLen;
    //размеры поля:
    POLE_WIDTH:= w; POLE_HEIGHT:= h;
    //вывести в заголовке формы имя загруженного файла:
    form1.caption:= NAME_PROG + '  [' + NameFig + ']';
    //вывести поля заданных размеров с числами:
    Prepare(POLE_WIDTH,POLE_HEIGHT, TOP_POLE_HEIGHT, LEFT_POLE_WIDTH);
  end; //opendialog1.Execute
end;//sbtLoadFigClick}
```

Следует заметить, что, как ни старайся последовательно излагать «материал», где-нибудь да и встретится анахронизм. Вот и в этой процедуре встречаются «воспоминания о будущем», поэтому давайте забежим чуточку вперёд и, нарушив логику событий, сделаем два добрых дела.

Во-первых, добавим к проекту ещё одну форму – *frmMemo*, и сверху слева поместим компонент *Image1*, в котором будет отображаться растровая картинка или значок – они послужат нам при составлении собственных задач. Как и прежде, в компоненте *Memo1* вы можете записывать свои комментарии к картинкам (например, сложность решения, авторство задачи, источник и тому подобное) - если будет желание. Они сохранятся с условием задания и будут вместе с ним загружаться. Расположение компонентов на форме большого значения не имеет, положитесь на свой вкус. Можно хотя бы так (Рис. Ф4.33).

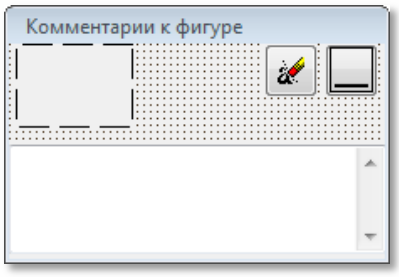


Рис. Ф4.33. Форма для комментариев

В тексте модуля *MemoUnit* нет ничего неожиданного для вас:

```
unit MemoUnit;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
Dialogs,
  StdCtrls, Buttons, ExtCtrls;

type
  TfrmMemo = class(TForm)
    Memo1: TMemo;
    sbtClear: TSpeedButton;
    sbtMinimize: TSpeedButton;
    Image1: TImage;
    procedure sbtClearClick(Sender: TObject);
    procedure sbtMinimizeClick(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  frmMemo: TfrmMemo;

implementation

uses NipponUnit;

{$R *.DFM}

procedure TfrmMemo.sbtClearClick(Sender: TObject);
begin
  frmMemo.memo1.Clear;
  frmMemo.memo1.SetFocus
end;

procedure TfrmMemo.sbtMinimizeClick(Sender: TObject);
begin
  frmMemo.WindowState:=wsMinimized
end;

end.
```

Во-вторых, обеспечим работой кнопку *sbtNumbers*. Она потребуется нам позднее, когда мы займёмся разработкой собственных задач. Нарисовав картинку, вы просто нажмёте эту кнопку, и программа автоматически установит нужные размеры числовых полей и заполнит их числами. Так

что вам никогда не придётся считать чёрные и белые клетки на своём рисунке: ваше дело – творчество, а считать должен компьютер! Действие этой процедуры достаточно прозрачно, что отнюдь не мешает вам «покопаться» в ней.

```
//ОЦИФРОВАТЬ РИСУНОК
procedure TForm1.sbtNumbersClick(Sender: TObject);
var
  i,j: integer;
  h,w: integer;
  n,ng: integer;
  Pred, len: integer;
begin
  //очистить цифровые поля:
  Clear_masColsNum;
  Clear_masRowsNum;
  //считываем данные для верхнего числового поля:
  h:=0; //- высота верхнего числового поля
  for i := 0 to POLE_WIDTH -1 do //- по длине поля
  begin
    ng:= 0;  //число групп в ряду
    len:= 0; //- длина группы
    Pred:= NUM_WHITE; //- предыдущая клетка
    //в каждом ряду должно быть хотя бы одно число:
    masColsNum[i,0].sNum:= '0';
    //для каждого столбца:
    for j := 0 to POLE_HEIGHT -1 do
    begin
      //считать очередной число:
      n:= masPole[i,j];
      if n= NUM_BLACK then begin //- чёрная клетка
          if pred<>NUM_BLACK then begin //- предыдущая клетка не чёрная -
->
            //записать группу:
            masColsNum[i,ng-1].sNum:= inttostr(len);
            //начинается новая группа:
            inc(ng);
            len:= 0;
          end;
          //увеличить длину группы:
          inc(len);
        end;
        pred:= n;
      end;
      masColsNum[i,ng-1].sNum:= inttostr(len);
      masColsNum[i,-1].Num:= ng;
      if ng> h then h:= ng;
    end;

    //формируем данные для левого числового поля:
    w:=0; //ширина левого числового поля
    for j := 0 to POLE_HEIGHT -1 do //- по всем строкам поля
    begin
      ng:= 0;   //- число групп в ряду
      len:= 0;  //- длина группы
      pred:= 0; //- предыдущее число
```

```
      //в каждом ряду должно быть хотя бы одно число:
      masRowsNum[0,j].sNum:= '0';
      for i := 0 to POLE_WIDTH -1 do //- по длине строки
      begin
        //считать очередное число:
        n:= masPole[i,j];
        if n= NUM_BLACK then begin //- чёрная клетка
          if pred<> NUM_BLACK then begin //-предыдущая клетка не чёрная -
->
            //записать группу:
            masRowsNum[ng-1,j].sNum:= inttostr(len);
            //начинается новая группа:
            inc(ng);
            len:= 0;
          end;
          //увеличить длину группы:
          inc(len);
        end;
        pred:= n;
      end; // i
      masRowsNum[ng-1,j].sNum:= inttostr(len);
      masRowsNum[-1,j].Num:= ng;
      if ng> w then w:= ng;
    end; //j
    //проверить число групп:
    if (w > MAX_CELLNUM) or (h > MAX_CELLNUM) then begin
      application.MessageBox('Слишком много групп!',NAME_PROG, MB_OK);
      exit
    end;
    TOP_POLE_HEIGHT:= h;
    LEFT_POLE_WIDTH:= w;
    //вывести поля заданных размеров с числами:
    Prepare(POLE_WIDTH,POLE_HEIGHT, TOP_POLE_HEIGHT, LEFT_POLE_WIDTH);
end; // TForm1.sbtNumbersClick
```

Если вы ввели данные для числовых полей, то было жалко потерять их после завершения работы с программой. Но нас уже давно дожидается временно безработная (но с определённым местом жительства!) кнопка *sbtSaveFig*. Она-то и будет переписывать условие задачи в файл.

Нажав эту кнопку, вы попадёте в диалог, где сможете выбрать *название* файла для своей задачи. Конечно, лучше со смыслом, чем с порядковым номером, иначе потом её трудно будет отыскать среди множества других. Вместе с данными в файл будет помещён и комментарий, поэтому у вас есть возможность добавить свои замечания к задаче, которые потом могут вам пригодиться.

Мы не будем изобретать собственного формата файла, а воспользуемся тем, что и в программе *Soluter* Гантверга и Самсонова - *JCP*. Это позволит вам решать свои задачи не только в рассматриваемой программе, но и в некоторых других. Польза от этого такая. Если вы будете совершенство-

вать *нашу* программу, то вам легко будет сравнить её «убойную силу» не только с ее предыдущими версиями, но и с чужими достижениями.

```
//ЗАПИСАТЬ ЗАДАЧУ
procedure TForm1.sbtSaveFigClick(Sender: TObject);
var
  F: textfile;
  fn,s: string;
  i,j: integer;
begin
  if status='ПОИСК' then exit;
  //расширение файлов фигур:
  savedialog1.DefaultExt:='jcp';
  savedialog1.Filter:='Japan puzzle (*.jcp)|*.JCP';
  //записываем в каталог 'FIGURE':
  s:=extractfilepath(application.exename)+'FIGURE\';
  savedialog1.InitialDir:= s;
  savedialog1.Title:='Запишите фигуру на диск';
  savedialog1.filename:= NameFig;
  if not savedialog1.Execute then exit;
  //имя конечного файла:
  fn:= savedialog1.filename;
  //изменить расширение файла, если при записи было выбрано другое имя:
  fn:=ChangeFileExt(fn, '.jcp');
  NameFig:=fn;
  assignfile(f,fn);
  rewrite(f);
  //записать фигуру -
  //высота и ширина фигуры:
  writeln (f, chr(POLE_HEIGHT + ord('0')) + chr(POLE_WIDTH+ ord('0')));
  //данные из левого числового поля-
  //очередная строка:
  for j:= 0 to POLE_HEIGHT-1 do begin
    for i:= 0 to LEFT_POLE_WIDTH-1 do
      if masRowsNum[i,j].sNum<>'' then
        write (f, chr(strtoint(masRowsNum[i,j].sNum) + ord('0')));
    writeln(f, '');
  end;
  //данные из верхнего числового поля -
  //очередная строка:
  for j:=0 to POLE_WIDTH-1 do begin
    for i:= 0 to TOP_POLE_HEIGHT-1 do
      if masColsNum[j,i].sNum<>'' then
        write (f, chr(strtoint(masColsNum[j,i].sNum) + ord('0')));
    writeln(f, '');
  end;
  //записать комментарий к фигуре:
  s:=frmMemo.memo1.Lines.Text;
  if s<>'' then begin
    s:='//'+s;
    writeln (f,s)
  end;
  closefile(f);
  //вывести в заголовке формы новое название файла:
  form1.caption:= NAME_PROG + '  [' + NameFig + ']';
  form1.Refresh ;
end; //Save
```

Делайте с нами, или Тонкие штучки на компьютере

Помнят руки!

Радостный возглас героя Юрия Никулина
из фильма *Когда деревья были большими*

Подготовив плацдарм, нам ничего другого не остаётся, как испытать свои силы в ручном решении – хотя и на компьютере – одного из японских кроссвордов. Вы можете начать с того самого рисунка, который был использован нами как пример решения задач. Тогда вы всегда сможете проконтролировать свои шаги и лучше усвоить *Правила*.

Само решение мы сделаем точно таким же, как и на бумаге, чтобы не морочить себе голову компьютерной премудростью.

1. Отыскав подходящий ряд, мы должны отметить разгаданные белые и чёрные клетки (напомню, что не разгаданные клетки - *серые*). Естественно использовать для этого левую и правую кнопки мыши. *Левой* мы будем ставить *чёрные* клетки, *правой* – *белые*. А вот если потребуется вернуть клетке её первоначальный *серый* цвет, то надо бы нажать *среднюю* кнопку мыши, да не все мышки так далеко продвинуты, поэтому серые клетки мы будем ставить той же *левой* кнопкой, но с нажатой клавишей **Shift**. Делать это придётся нечасто (не ошибайтесь при решении – и вовсе не придётся), так что смиритесь. Для пущего форсу вид мышиного курсора будет изменяться: при передвижении картинки он приобретёт вид загребущей *руки*, при рисовании – *кисти*. Причём передвигать картинку или рисовать на поле можно, только если нажата соответствующая кнопка в меню. Процесс рисования выглядит так: по координатам мышки в пикселях мы определяем, в какой клетке поля она находится, и заносим в массив код нужного цвета, после чего перерисовываем клетку в обновлённом виде.

```
//НАЖАТЬ КНОПКУ МЫШКИ
procedure TForm1.dgPoleMouseDown(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
var
  ACol,ARow: integer;
  area: tRect;
begin
  if status='ПОИСК' then exit;
  //изменить форму курсора
  //передвигаем картинку:
  if sbtMove.down then begin
    screen.Cursor:=TCursor(crMove);
    dgPole.Cursor :=TCursor(crMove);
    screen.Cursor:=TCursor(crDefault);
    exit end;
  //рисуем на поле:
  if not sbtDraw.Down then exit;
  screen.Cursor:=TCursor(crKistj);
  dgPole.Cursor :=TCursor(crKistj);
```

```
  screen.Cursor:=TCursor(crDefault);

  //координаты мыши:
  dgPole.MouseToCell(x,y,ACol,ARow);
  cellmouse.x:=ACol;
  cellmouse.y:=ARow;
  area:= dgPole.CellRect(ACol, ARow);
  //если кнопка мыши нажата вместе с клавишей Shift,
  //то поставить серую клетку:
  if ssShift in shift then begin
    masPole[ACol, ARow]:=NUM_GRAY;
    //закрасить клетку:
    dgPoleDrawCell(self, ACol, ARow, area, []);
    exit
  end;
  //если нажата только левая кнопка - чёрная клетка:
  if ssLeft in shift then
  begin
    //занести в массив цвет клетки
    masPole[ACol, ARow]:=NUM_BLACK;
    //закрасить клетку:
    dgPoleDrawCell(self, ACol, ARow, area, []);
 end;
  //если нажата правая кнопка - белая клетка:
  if ssRight in shift then
  begin
    //занести в массив цвет клетки:
    masPole[ACol, ARow]:= NUM_WHITE;
    //закрасить клетку:
    dgPoleDrawCell(self, ACol, ARow, area, []);
  end;
end; // TForm1.dgPoleMouseDown
```

При отпускании кнопки мыши мы просто возвращаем курсору обычный вид:

```
//ОТПУСТИТЬ КНОПКУ МЫШИ
procedure TForm1.dgPoleMouseUp(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  //изменить форму курсора:
  screen.Cursor:=TCursor(crHand);
  dgPole.Cursor :=TCursor(crHand);
  screen.Cursor:=TCursor(crDefault);
end;
```

В принципе, этого вполне достаточно для того, чтобы решать задачи. Вот только не очень удобно щёлкать на каждой клетке, если разгадана целая группа клеток, а то и весь ряд целиком. Поэтому наделим мышку, как и курьёзного Микки Мауса, способностью закрашивать клетки при передвижении по полю. События здесь происходят точно те же, что и при щелчке. Мы добавим в процедуру *dgPoleMouseMove* ещё и возможность *циклически сдвигать* рисунок. Пока нам это ни к чему, но, взявшись за рисование соб-

ственных кроссвордов, вы оцените её сполна: если к уже готовой физиономии вам захочется пририсовать усы, а места для них нет, то просто сдвиньте картинку и потешьте себя. Поскольку рассуждать о сдвигах - хотя бы и поля – сейчас не очень уместно, то разберитесь в этом вопросе самостоятельно, тем более что здесь всё просто (такие «миражи» и «вертижи» самомнения часто возникают, когда программа уже отлажена).

```
//ПЕРЕМЕЩАТЬ МЫШКУ ПО ПОЛЮ
procedure TForm1.dgPoleMouseMove(Sender: TObject; Shift:
TShiftState; X,
  Y: Integer);
var
  ACol,ARow: integer;
  area: TRect;

  //прокрутить поле влево
  procedure PoleLeft;
  var
    n, x, y: Integer;
  begin
    //сдвигаем:
    For y := 0 To POLE_HEIGHT-1 do begin
      //запомнить цвет первой клетки:
      n:= masPole[0,y];
      For x := 0 To POLE_WIDTH - 2 do
        masPole[x,y] := masPole[x+1,y];
      //последний ряд <-- первый ряд:
      masPole[POLE_WIDTH-1,y]:= n;
    end
  end; //PoleLeft
  //Сдвинуть поле вправо
  procedure PoleRight;
  var
    n, x,y: Integer;
    XR: Integer; //правый столбец
  begin
    XR:= POLE_WIDTH-1;
    For y := 0 To POLE_HEIGHT-1 do begin
      n:= masPole[XR, y];
      For x := XR downTo 1 do
        masPole[x,y]:= masPole[x-1,y];
      //первый ряд <-- последний ряд
      masPole[0, y]:=n;
    end
  End; //PoleRight
  //Сдвинуть поле вверх
  procedure PoleUp;
  var
    n, x, y: Integer;
  begin
    For x := 0 To POLE_WIDTH-1 do begin
```

```
      n:= masPole[x,0];
      For y := 1 To POLE_HEIGHT-1 do
        masPole[x,y-1] := masPole[x,y];
      //нижняя строка <-- верхняя строка:
      masPole[x, POLE_HEIGHT-1]:= n;
    end
  End; //PoleUp
  //Сдвинуть поле вниз
  procedure PoleDown;
  var
    n, x, y: Integer;
  begin
    For x := 0 To POLE_WIDTH-1 do begin
      n:= masPole[x, POLE_WIDTH-1];
      For y := POLE_HEIGHT-1 downTo 1  do
        masPole[x,y] := masPole[x,y-1];
      //верхняя строка <-- нижняя строка:
      masPole[x, 0]:= n;
    end
  End; //PoleDown

begin
  if status='ПОИСК' then exit;
  dgPole.SetFocus;
  //координаты мыши:
  dgPole.MouseToCell(x,y,ACol,ARow);
  statusbar1.Panels[3].text:='X= '+inttostr(ACol+1);
  statusbar1.Panels[4].text:='Y= '+inttostr(ARow+1);
  //циклически сдвигаем картинку:
  if (sbtMove.Down) and (ssLeft in shift)  then begin
    if cellmouse.x>ACol then PoleLeft;
    if cellmouse.x<ACol then PoleRight;
    if cellmouse.y>ARow then PoleUp;
    if cellmouse.y< ARow then PoleDown;
    dgPole.Invalidate;
    cellmouse.x:=ACol;
    cellmouse.y:=ARow;
  end;
  //рисуем:
  if not sbtDraw.Down then exit;
  area:= dgPole.CellRect(ACol, ARow);
  //если кнопка мыши нажата вместе с клавишей Shift,
  //то поставить серую клетку:
  if (ssShift in shift) and (ssLeft in shift) and
     ((cellmouse.x<>ACol) or (cellmouse.y<>ARow)) then
  begin
    //занести в массив цвет клетки:
    masPole[ACol, ARow]:=NUM_GRAY;
    //закрасить клетку:
    dgPoleDrawCell(Self, ACol, ARow, area, []);
    cellmouse.x:=ACol;
    cellmouse.y:=ARow;
    exit
```

```
  end;
  //если нажата левая кнопка - чёрная клетка
  if (ssLeft in shift) and
     ((cellmouse.x<>ACol) or (cellmouse.y<>ARow))then
  begin
    //занести в массив цвет клетки:
    masPole[ACol, ARow]:=NUM_BLACK;
    //закрасить клетку:
    dgPoleDrawCell(Self, ACol, ARow, area, []);
  end;
  if (ssRight in shift) and
     ((cellmouse.x<>ACol) or (cellmouse.y<>ARow))then
  begin
    //занести в массив цвет клетки:
    masPole[ACol, ARow]:= NUM_WHITE;
    //закрасить клетку:
    dgPoleDrawCell(Self, ACol, ARow, area, []);
  end;
  //новые координаты мыши:
  cellmouse.x:=ACol;
  cellmouse.y:=ARow;
end; // TForm1.dgPoleMouseMove
```

2. Помните, как в рассмотренном в начале главы примере мы отмечали разгаданные группы *кружком*? Казалось бы, мелочь, а сильно облегчает жизнь: сразу видно, какие группы ещё не разгаданы. Сейчас мы поступим иначе – кружки ставить не будем, а вместо этого пометим разгаданные группы *жёлтым* цветом. Иначе говоря, мы изменяем при этом *статус* группы – из состояния «неразгаданности» *stWhite*, мы переводим её в более приятное состояние *stYellow* (щёлкнув еще раз, вы вернёте клетке белый цвет). Так что процедуре, обрабатывающей нажатие кнопки в числовом поле, нужно просто занести нужное значение статуса в «кликнутую» клетку:

```
//ИЗМЕНИТЬ СТАТУС ГРУППЫ
procedure TForm1.dgColsNumClick(Sender: TObject);
var ACol, ARow: Integer;
  r: TRECT;
begin
  if not sbtDraw.Down then exit;
  ACol:= cellMouseTop.x; ARow:= cellMouseTop.y;
  case masColsNum[ACol, ARow].StatusGroup of
    stYellow: masColsNum[ACol, ARow].StatusGroup:= stWhite;
    stWhite: masColsNum[ACol, ARow].StatusGroup:= stYellow;
  end;
  r:= dgColsNum.CellRect(ACol, ARow);
  dgColsNumDrawCell(self, ACol, ARow, r, []);
end;

//ИЗМЕНИТЬ СТАТУС ГРУППЫ
procedure TForm1.dgRowsNumClick(Sender: TObject);
var ACol, ARow: Integer;
  r: TRECT;
```

```
begin
  if not sbtDraw.Down then exit;
  ACol:= cellMouseLeft.x; ARow:= cellMouseLeft.y;
  case masRowsNum[ACol, ARow].StatusGroup of
    stYellow: masRowsNum[ACol, ARow].StatusGroup:= stWhite;
    stWhite: masRowsNum[ACol, ARow].StatusGroup:= stYellow;
  end;
  r:= dgRowsNum.CellRect(ACol, ARow);
  dgRowsNumDrawCell(self, ACol, ARow, R, []);
end;
```

В примере мы дополнительно отмечали и *разгаданные* ряды, здесь мы этого делать не будем – и так всё хорошо видно. А кому не видно – измените процедуру так, чтобы при клике на жёлтой клетке она перекрашивалась в *зелёную* (переход в состояние *stGreen*).

Теперь решение задачи на компьютере мало чем отличается от обычного, поэтому можете любую задачу решать и так, и этак (Рис. Ф4.34).

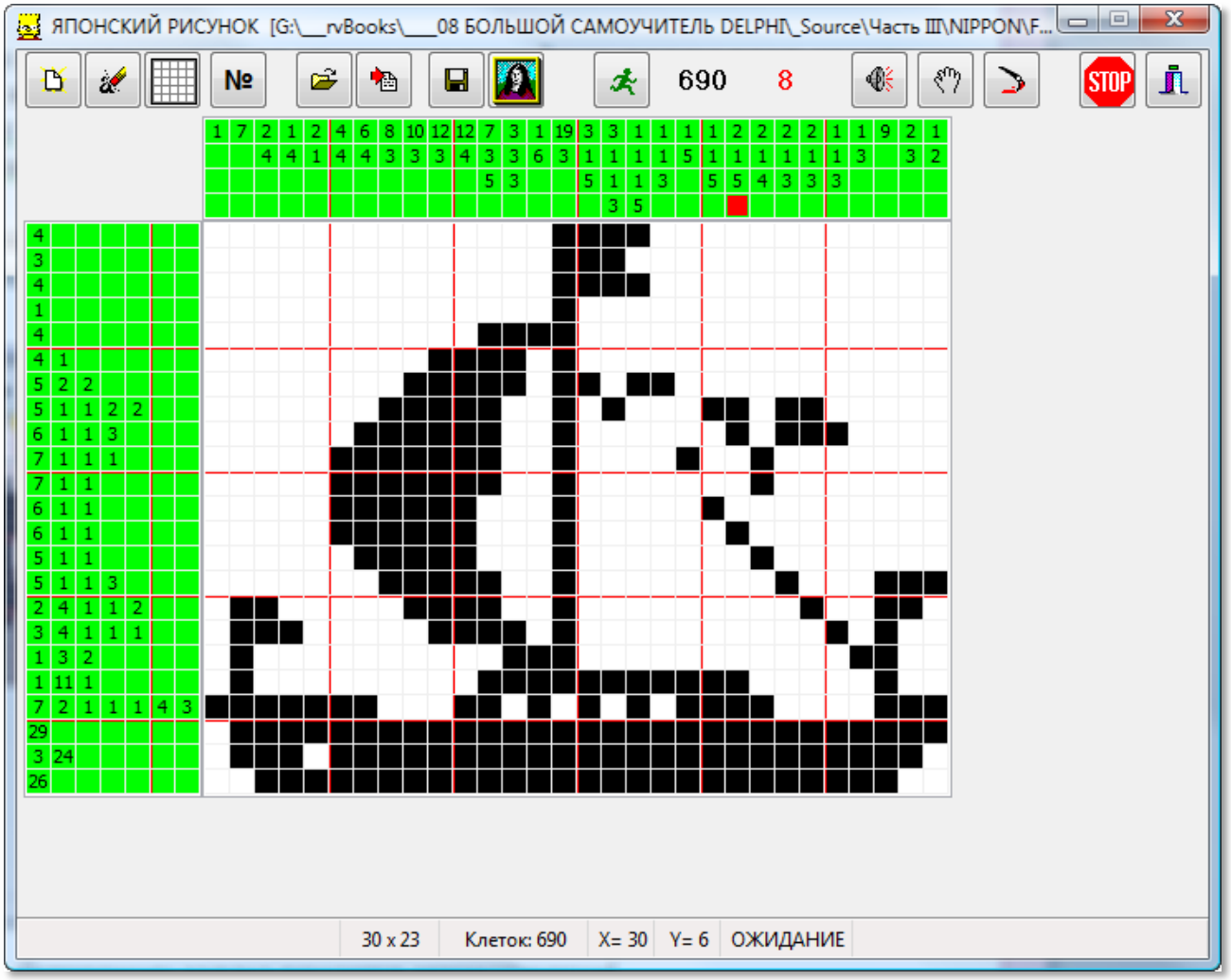


Рис. Ф4.34. Осторожно: задача решается автоматически!

На этом мы и остановимся: решать задачи вручную – это *не наш метод*, как сказал однажды верзила Федя вечному студенту Шурику. Если же эта мысль не оставляет вас в покое, то вам нужно позаботиться о запоминании позиции на поле после каждого хода (как это сделать, будет рассказано дальше) и о возврате назад, если вы попадёте в тупик. Клетку, в цвете которой вы не уверены, следует помечать иначе, чем разгаданные и не разгаданные, чтобы потом можно было бы вернуться к ней. Неплохо было бы подсчитывать количество уже разгаданных клеток. Не очень удобно решать на экране задачи больших размеров, так как ряды целиком не видны. Придётся постоянно двигать поле. Можно немного улучшить ситуацию, уменьшив размеры клеток в сетках и, соответственно, размер шрифта. Но тут уж берегите глаза – решение огромных задач потребует от вас нескольких часов работы и напряжённого вглядывания в экран. Так что если вас неудержимо тянет к монументальным формам, купите себе монитор подходящих размеров. Конечно, для воплощения всего этого придётся потрудиться, но – охота пуще неволи!

А нам пора дальше. Заставим компьютер делать всё, чему мы уже научились сами. А учить компьютер – вот где настоящая педагогическая поэма! Испытайте себя в роли учителя – и вы много нового откроете для себя (и в себе).

Компьютерная педагогика, или Пускай работает Иван

Не можешь – научим,
Не хочешь – заставим!

Из книги *Армейская жизнь как таковая*

Какое наслаждение наблюдать за тем,
как работают другие!

Из *Манифеста* Филона

Прежде чем с головой окунуться в пучину страсти, мы сначала должны проверить условие задачи, иначе всё решение пойдёт насмарку. Для этого мы напишем функцию *Testing*, которая будет возвращать значение *True*, если всё правильно, или *False*, если обнаружились ошибки.

Во-первых, она должна проверить, а загружена ли задача. Это легко установить по названию задачи, которое хранится в строковой переменной *NameFig*. Если её значение равно *'temp'*, то это ошибка: задача не загруже-

на. Эта же ситуация возникнет, если вы ввели данные задачи в числовые поля, но не сохранили её на диске под другим именем.

Во-вторых, если вы *вводили* числа в поля, то нужно переписать их так, чтобы они были записаны правильно, то есть от левого и верхнего краёв числовых полей. За комфорт при вводе чисел придётся расплачиваться написанием процедуры *MoveNums*. В ней все числа будут сдвинуты вверх и влево (естественно, если числа сразу были записаны верно, то ничего и не изменится). И не забудьте объявить процедуру в разделе *private* типа формы:

```
procedure MoveNums;
```

```
//ПЕРЕНЕСТИ ЧИСЛА В ЧИСЛОВЫХ ПОЛЯХ
procedure TForm1.MoveNums;
var
  i, j, n: integer;
  s: string;
begin
  //переносим все числа к верхней границе сетки
  for i:= 0 to POLE_WIDTH-1 do
  begin
    n:= 0;
    //каждый столбец:
    for j:= 0 to TOP_POLE_HEIGHT-1 do
    begin
      s:= masColsNum[i,j].sNum;
      if s <> '' then begin
        masColsNum[i,j].sNum:= '';
        masColsNum[i,n].sNum:= s;
        inc(n);
      end;
    end;
  end;
  //левое числовое поле:
  for i:= 0 to POLE_HEIGHT-1 do
  begin
    n:= 0;
    //в каждой строке:
    for j:= 0 to LEFT_POLE_WIDTH-1 do
    begin
      s:= masRowsNum[j,i].sNum;
      if s <> '' then begin
        masRowsNum[j,i].sNum:= '';
        masRowsNum[n,i].sNum := s;
        inc(n);
      end;
    end;
  end;
  Invalidategrids;
end; //TForm1.MoveNums;
```

В-третьих, кроме проверок, в этой же функции мы соединим приятное с полезным: запишем число групп в каждом ряду в массив, чтобы не пересчитывать их каждый раз, когда нам потребуется это знать.

В-четвёртых, из элементарной арифметики известно, что общее количество клеток в группах не зависит от того, будем ли мы их пересчитывать слева направо или сверху вниз. Поэтому суммы чисел в верхнем и в левом числовых полях должны быть *равны*. Если это не так, то задачу и решать не стоит. Правда, равенство этих сумм вовсе не гарантирует правильности условия задачи, то есть это условие необходимое, но не недостаточное. Например, если обе суммы равны нулю, то условие задачи (числа) просто отсутствует. Это положение может возникнуть, если вы не оцифровали свой рисунок, а хотите его решить (вообще-то его и решать не нужно, но вдруг вам захотелось «приколоться»). А вот более «хитрые» ошибки можно обнаружить, только уже решая задачу.

```
//проверить данные задачи
function Testing(): Boolean ;
var
  sumL, sumT: integer;
  i, j, n: integer;
begin
  Result:= False;
  //если задача не загружена, то нечего решать:
  if NameFig='temp' then
  begin
    WavError;
    application.MessageBox('Вы не загрузили задачу!',NAME_PROG, MB_OK);
    exit
  end;
  //нормализовать запись чисел:
  MoveNums;
  //записать число групп в каждом ряду в массив
  //и подсчитать сумму чисел -
  //верхнее числовое поле:
  sumT:= 0;
  for i:= 0 to POLE_WIDTH-1 do
  begin
    n:= 0;
    //считаем в каждом столбце:
    for j:= 0 to TOP_POLE_HEIGHT-1 do
    begin
      if masColsNum[i,j].sNum = '' then break; //- числа кончились
      inc(n);
      //записать соотв. число:
      masColsNum[i,j].Num := strtoint(masColsNum[i,j].sNum);
      sumT:= sumT + masColsNum[i,j].Num
    end;
    //записать число групп:
    masColsNum[i,-1].Num:= n;
  end;
  //левое числовое поле:
```

```
    sumL:= 0;
    for i:= 0 to POLE_HEIGHT-1 do
    begin
      n:= 0;
      //считаем в каждой строке:
      for j:= 0 to LEFT_POLE_WIDTH-1 do
      begin
        if masRowsNum[j,i].sNum = '' then break; //- числа кончились
        inc(n);
        //записать соотв. число:
        masRowsNum[j,i].Num := strtoint(masRowsNum[j,i].sNum);
        sumL:= sumL + masRowsNum[j,i].Num
      end;
      //записать число групп:
      masRowsNum[-1,i].Num:= n;
    end;
    //проверить суммы чисел:
    if sumT + sumL= 0 then begin
      WavError;
      application.MessageBox('Вы   не   оцифровали   рисунок!',NAME_PROG,
MB_OK);
      exit
    end;
    if sumT <> sumL then
    begin
      WavError;
      s:= 'Сумма чисел сверху (' +inttostr(sumT)+
      ')'#10#13'не равна сумме чисел слева (' +inttostr(sumL)+')!';
      application.MessageBox(PChar(s),NAME_PROG, MB_OK);
      exit;
    end;

    //если в ряду есть нулевая группа, то других быть не должно
    //верхнее числовое поле:
    for i:= 0 to POLE_WIDTH-1 do
      //проверяем каждый столбец:
      for j:= 0 to TOP_POLE_HEIGHT-1 do
      begin
        if masColsNum[i,j].sNum = '' then break; //- числа кончились
        if (masColsNum[i,j].sNum = '0') and (masColsNum[i,-1].Num>1)
        then begin
          WavError;
          s:='Неверные данные в столбце '+ inttostr(i+1)+'!';
          application.MessageBox(PChar(s),NAME_PROG, MB_OK);
          exit
        end;
      end;
    //левое числовое поле:
    for i:= 0 to POLE_HEIGHT-1 do
    //проверяем каждую строку:
      for j:= 0 to LEFT_POLE_WIDTH-1 do
      begin
        if masRowsNum[j,i].sNum = '' then break; //- числа кончились
        if (masRowsNum[j,i].sNum = '0') and (masRowsNum[-1,i].Num>1)
        then begin
          WavError;
          s:='Неверные данные в строке '+ inttostr(i+1)+'!';
```

```
          application.MessageBox(PChar(s),NAME_PROG, MB_OK);
          exit
        end;
      end;
    //всё нормально:
    Result:= True;
  end; // Testing
```

Если вы были дюже внимательны, то могли разглядеть в процедуре «инородное тело» - *WavError*. Оно внедрено сюда специально для тех, кто не равнодушен к звуковым галлюцинациям, иначе говоря, эффектам, которые преследуют каждого пользователя *Windows*, если он не выключил колонки. Но - каждый имеет право отравлять себе жизнь всеми доступными ему способами, в том числе и самыми изощрёнными, к коим можно смело отнести музицирование на компьютере. А готовится это зелье так. Объявите там же, где и функцию *Testing,* три *процедуры*:

```
    procedure WavError;
    procedure WavReadyLine;
    procedure WavPobeda;
```

Которые выглядят так:

```
//====================== ЗВУКОВЫЕ ЭФФЕКТЫ ==========================
//ОШИБКА!
procedure TForm1.WavError;
begin
  if not sbtSound.Down then exit;
  //воспроизвести звуковой эффект:
  sndPlaySound('wav\error.wav', SND_ASYNC or SND_FILENAME);
end;
//РЯД РЕШЁН!
procedure TForm1.WavReadyLine;
begin
  if not sbtSound.Down then exit;
  sndPlaySound('Wav\ReadyLine.wav', SND_ASYNC or SND_FILENAME);
end;
//ЗАДАЧА РЕШЕНА!
procedure TForm1.WavPobeda;
begin
  if not sbtSound.Down then exit;
  sndPlaySound('Wav\pobeda.wav', SND_ASYNC or SND_FILENAME);
end;
```

Все звуковые файлы хранятся в папке *Wav,* которая находится в той же папке (ох, уж эти папки!), что и выполняемый файл программы. Но вы можете засунуть эту музыку и куда подальше. Только не забудьте указать программе путь к ней, чтобы не вышла по вашему велению сказочная история, в которой герою предлагалось пойти туда не знаю куда.

Если вам этого звукового сопровождения (п)оказалось мало (а вдруг вы меломан), то вы можете добавить сколько угодно своих эффектов и расставить их в программе на каждом шагу. Тогда при решении задачи у вас не только глаза вылезут, но и ушки завернутся!

Но музыку, даже самую противную, любят не все, поэтому её услышит только тот, кто по неосторожности нажмёт кнопку *sbtSound* да так и оставит её без присмотра.

И вот условия задачи проверены, вся цифирь оказалась на месте, и нам уже не отвертеться от решения оной задачи. Сейчас мы сделаем много лишнего, что, однако, необходимо нам для отыскания общего алгоритма решения любых японских кроссвордов.

Давайте заставим компьютер выполнять все *Золотые правила*.

Они достаточно просты, и, конечно, каждый помнит их назубок.

1. Ряды, в которых есть *нулевые группы*, можно сразу закрасить *белым*. Процедура *TestingZero* с лёгкостью сделает то, с чего мы начинали решение задачи-примера. Она перебирает все ряды чисел в числовых полях и ищет в них *нули*. Найдя такой ряд, она отмечает его как *решённый* (статус такого ряда, как вы помните, *stGreen*), окрашивает все клетки ряда в *белый* цвет и, наконец, считает разгаданные клетки (их число хранится в переменной *ReadyCells*). Так как эти операции очень просты и хорошо известны, то вам легко будет понять их действие в программе. Потом вы можете и не пользоваться этой процедурой, так как общий алгоритм решения задачи обойдётся и без неё. С другой стороны, она ничуть и не мешает.

```
//отмечаем пустые строки
  procedure TestingZero;
  var
    i, j, n: integer;
  begin
    //проверяем верхнее числовое поле:
    for i:= 0 to POLE_WIDTH-1 do
      //проверяем каждый столбец:
      for j:= 0 to masColsNum[i,-1].Num-1 do
      begin
        if (masColsNum[i,j].sNum = '0') then //- нашли
        begin
          {s:='Есть 0 в столбце '+ inttostr(i+1)+'!';
          application.MessageBox(PChar(s),NAME_PROG, MB_OK);}
          //этот столбец решён:
          masColsNum[i,-1].StatusGroup:= stGreen;
          //закрашиваем столбец поля белым цветом:
          for n:= 0 to POLE_HEIGHT-1 do
          begin
            if masPole[i,n]=NUM_GRAY then //- эти клетки разгаданы?
            begin
```

```
            masPole[i,n]:= NUM_WHITE;
            inc(ReadyCells)
          end;
        end;
      end;
    end;
  //проверяем левое числовое поле:
  for i:= 0 to POLE_HEIGHT-1 do
  //проверяем каждую строку:
    for j:= 0 to masRowsNum[-1,i].num-1 do
    begin
      if (masRowsNum[j,i].sNum = '0')then //- ашли
      begin
        //эта строка решена:
        masRowsNum[-1,i].StatusGroup:= stGreen;
        {s:='Есть 0 в строке '+ inttostr(i+1)+'!';
        application.MessageBox(PChar(s),NAME_PROG, MB_OK);}
        //закрашиваем строку поля белым цветом:
        for n:= 0 to POLE_WIDTH-1 do
        begin
          if masPole[n,i]=NUM_GRAY then //- эти клетки разгаданы?
          begin
            masPole[n,i]:= NUM_WHITE;
            inc(ReadyCells)
          end;
        end;
      end;
    end;
  InvalidateGrids;
end; // TestingZero
```

2. По *Правилу 2*, ряды, в которых есть число, равное размеру поля, можно сразу закрасить *чёрным*. Кодируем *Правило 2* аналогично предыдущему. Единственное отличие – мы должны проверить, не «накладываются» ли чёрные клетки на белые (они могли появиться после выполнения *первого Правила*). Если это так, то в условии задачи имеются ошибки, решить её невозможно, а функция возвращает значение *False*.

```
//отмечаем полные строки
function TestingFullLine(): Boolean;
var
  i, n: integer;
begin
  Result:= TRUE;
  //проверяем верхнее числовое поле:
  for i:= 0 to POLE_WIDTH-1 do
    if masColsNum[i,0].Num = POLE_HEIGHT then //нашли
    begin
      //s:='Есть полный столбец '+ inttostr(i+1)+'!';
      //application.MessageBox(PChar(s),NAME_PROG, MB_OK);
      //закрашиваем столбец поля чёрным цветом:
      for n:= 0 to POLE_HEIGHT-1 do
      begin
        if masPole[i,n]<>NUM_GRAY then //- эти клетки уже разгаданы?
        begin
```

```
            s:='Неверные числа в столбце '+ inttostr(i+1)+#10#13+
            'и строке ' + inttostr(n+1) +'!';
            application.MessageBox(PChar(s),NAME_PROG, MB_OK);
            Result:= FALSE;
            exit
          end
          else
          begin
            masPole[i,n]:= NUM_BLACK;
            inc(ReadyCells)
          end;
        end;
    //этот столбец решён:
    masColsNum[i,-1].StatusGroup:= stGreen;
    end;
    //проверяем левое числовое поле:
    for i:= 0 to POLE_HEIGHT-1 do
      if masRowsNum[0,i].Num = POLE_WIDTH then //- нашли
      begin
        //s:='Есть полная строка '+ inttostr(i+1)+'!';
        //application.MessageBox(PChar(s),NAME_PROG, MB_OK);
          //закрашиваем строку поля чёрным цветом:
          for n:= 0 to POLE_WIDTH-1 do
          begin
            if masPole[n,i]=NUM_WHITE then //- эта клетка белая?
            begin
              s:='Неверные числа в строке '+ inttostr(i+1)+#10#13+
              'и столбце ' + inttostr(n+1) +'!';
              application.MessageBox(PChar(s),NAME_PROG, MB_OK);
              Result:= FALSE;
              exit
            end
            else if masPole[n,i]=NUM_GRAY then//эта клетка серая?
            begin
              masPole[n,i]:= NUM_BLACK;
             inc(ReadyCells)
            end;
          end;
      //эта строка решена:
      masRowsNum[-1,i].StatusGroup:= stGreen;
      end;
    InvalidateGrids;
  end; // TestingFullLine
```

Судьбина у этой функции тоже горькая – как пример перевода правил с русского языка на дельфийский она ещё ничего себе, но программе совершенно не нужна.

3. По *Правилу 3*, мы отыскиваем ряды с *полными суммами* и закрашиваем клетки всех групп *чёрным*, а между ними по одной клетке – *белым*. При этом мы проверяем, не «накладываются» ли чёрные клетки на уже известные белые, и наоборот. Так как некоторые ряды уже могли быть решены раньше (тогда их *StatusGroup = stGreen*), то их мы пропускаем. Не забываем подсчитывать решённые клетки (их количество равно сумме *серых* клеток, которые были в разгаданном ряду).

Для удобства введём две *функции*. Объявите их в разделе *private*:

```
function GetNumGroup(Location: TPoleLocation; nLine: integer): in-
teger;
function GetLenGroup(Location: TPoleLocation; nLine, nGroup: inte-
ger): integer;
```

Для них пригодится новый *тип*:

```
//положение числового поля: слева, сверху
type TPoleLocation= (plLEFT, plTOP);
```

Первая функция возвращает число групп в заданном ряду верхнего или левого полей, *вторая* – длину заданной группы:

```
//ПОЛУЧИТЬ КОЛИЧЕСТВО ГРУПП В РЯДУ nLine левого (LEFT) или
//верхнего (TOP) числового поля
function TForm1.GetNumGroup(Location: TPoleLocation; nLine: integer): in-
teger;
begin
  case Location of
    plLEFT: Result:= masRowsNum[-1, nLine].Num;
    plTOP:  Result:= masColsNum[nLine, -1].Num;
    else Result:= -1;
  end;
end;
//ПОЛУЧИТЬ ДЛИНУ ГРУППЫ nGroup В РЯДУ nLine левого (LEFT) или
//верхнего (TOP) числового поля
function TForm1.GetLenGroup(Location: TPoleLocation; nLine, nGroup: inte-
ger)
         : integer;
begin
  case Location of
    plLEFT: Result:= masRowsNum[nGroup, nLine].Num;
    plTOP:  Result:= masColsNum[nLine, nGroup].Num;
    else Result:= -1;
  end;
end;
```

Так как это правило значительно сложнее предыдущих, то и код получится *подлиннееееее*:

```
  //отмечаем строки с полными суммами
  function TestingFullSum(): Boolean;
  var
    i, j, n, k, sum: integer;
  begin
    Result:= TRUE;
    //проверяем верхнее числовое поле:
    for i:= 0 to POLE_WIDTH-1 do
      if masColsNum[i,-1].StatusGroup<> stGreen then //- столбец ещё не решён
      begin
        sum:= 0;
        //проверяем каждый столбец
        //находим сумму для каждого столбца:
        for j:= 0 to masColsNum[i,-1].Num-1 do
```

```
        sum:= sum + masColsNum[i,j].Num;
      sum:= sum + masColsNum[i,-1].Num-1;
      if sum= POLE_HEIGHT then //- нашли
      begin
        {s:='Есть полная сумма в столбце '+ inttostr(i+1)+'!';
        application.MessageBox(PChar(s),NAME_PROG, MB_OK);}
        //этот столбец решён:
        masColsNum[i,-1].StatusGroup:= stGreen;
        //закрашиваем столбец поля:
        k:= 0;
        for n:= 0 to masColsNum[i,-1].Num-1 do //-по всем группам
        begin
          //закрасить группу чёрным:
          for j:= k to k+ masColsNum[i,n].Num-1 do
          begin
            if masPole[i,j]= NUM_WHITE then //эта клетка белая?
            begin
              s:='Неверные числа в столбце '+ inttostr(i+1)+#10#13+
              'и строке ' + inttostr(j+1) +'!';
              application.MessageBox(PChar(s),NAME_PROG, MB_OK);
              Result:= FALSE;
              exit
            end
            else if masPole[i,j]=NUM_GRAY then//- эта клетка серая?
            begin
              masPole[i,j]:= NUM_BLACK;
              inc(ReadyCells)
            end;
          end; // for j
          //поставить белую клетку между группами:
          if n < GetNumGroup(plTOP, i)-1 then
          begin
            k:= k+ masColsNum[i,n].Num;
            if masPole[i,k]= NUM_BLACK then //эта клетка чёрная?
            begin
              s:='Неверные числа в столбце '+ inttostr(i+1)+#10#13+
              'и строке ' + inttostr(k+1) +'!';
              application.MessageBox(PChar(s),NAME_PROG, MB_OK);
              Result:= FALSE;
              exit
            end
            else if masPole[i,k]=NUM_GRAY then//- эта клетка серая?
            begin
              masPole[i,k]:= NUM_WHITE;
              inc(ReadyCells);
            end;
            inc(k);
          end; //n < GetNumGroup(plTOP, i)-1
        end; // for
      end; //if sum= POLE_HEIGHT
    end; //if masColsNum[i,-1].StatusGroup<> stGreen

  //проверяем левое числовое поле:
  for i:= 0 to POLE_HEIGHT-1 do
    if masRowsNum[-1,i].StatusGroup<> stGreen then //- строка ещё не решена
    begin
      sum:= 0;
      //проверяем каждый строку
      //находим сумму для каждой строки:
      for j:= 0 to masRowsNum[-1,i].Num-1 do
        sum:= sum + masRowsNum[j,i].Num;
      sum:= sum + masRowsNum[-1,i].Num-1;
      if sum= POLE_WIDTH then //нашли
```

```
        begin
          {s:='Есть полная сумма в строке '+ inttostr(i+1)+'!';
          application.MessageBox(PChar(s),NAME_PROG, MB_OK);}
          //эта строка решена:
          masRowsNum[-1,i].StatusGroup:= stGreen;
          //закрашиваем строку поля:
          k:= 0;
          for n:= 0 to masRowsNum[-1,i].Num-1 do //- по всем группам
          begin
            //закрасить группу чёрным:
            for j:= k to k+ masRowsNum[n,i].Num-1 do
            begin
              if masPole[j,i]= NUM_WHITE then //- эта клетка белая?
              begin
                s:='Неверные числа в строке '+ inttostr(i+1)+#10#13+
                'и столбце ' + inttostr(j+1) +'!';
                application.MessageBox(PChar(s),NAME_PROG, MB_OK);
                Result:= FALSE;
                exit
              end
              else if masPole[j,i]=NUM_GRAY then//- эта клетка серая?
              begin
                masPole[j,i]:= NUM_BLACK;
                inc(ReadyCells)
              end;
            end; // for j
            //поставить белую клетку между группами:
            if n< GetNumGroup(plLEFT, i)-1 then
            begin
              k:= k+ masRowsNum[n,i].Num;
              if masPole[k,i]= NUM_BLACK then //- эта клетка чёрная?
              begin
                s:='Неверные числа в строке '+ inttostr(i+1)+#10#13+
                'и  столбце' + inttostr(k+1) +'!';
                application.MessageBox(PChar(s),NAME_PROG, MB_OK);
                Result:= FALSE;
                exit
              end
              else if masPole[k,i]=NUM_GRAY then//- эта клетка серая?
              begin
                masPole[k,i]:= NUM_WHITE;
                inc(ReadyCells);
              end;
              inc(k);
            end; //n< GetNumGroup(plLEFT, i)-1
          end; // for
        end; //if sum= POLE_HEIGHT
      end; //if masRowsNum[i,-1].StatusGroup<> stGreen
      InvalidateGrids;
  end; //TestingFullSum
```

Эту функцию желательно оставить в программе, она может пригодиться в начале решения задачи. Но программа справится с любым кроссвордом и без неё.

4. По *Правилу 4*, если единственная группа в ряду длиннее половины ряда, то часть ряда можно закрасить *чёрным*. Функция *TestingLongLine* последовательно перебирает все ряды, и, если находит неразгаданный ряд и в нём *единственное* число, большее половины длины соответствующего ряда, то закрашивает часть клеток *чёрным* цветом. Как определить эти клетки, мы уже подробно рассматривали раньше.

```
//отмечаем длинные строки
function TestingLongLine(): Boolean;
var
  i, j, sum: integer;
begin
  Result:= TRUE;
  //проверяем верхнее числовое поле:
  for i:= 0 to POLE_WIDTH-1 do
    if masColsNum[i,-1].StatusGroup<> stGreen then //- столбец ещё не решён
    begin
      //в ряду должна быть одна группа:
      if GetNumGroup(plTOP, i)=1 then
      begin
        sum:= GetLenGroup(plTOP, i, 0)*2;
        if sum - POLE_HEIGHT >0 then //нашли
        begin
          {s:='Длинный столбец '+ inttostr(i+1)+'!';
          s:= s+'beg= '+inttostr((POLE_HEIGHT-GetLenGroup(plTOP, i, 0)));
          application.MessageBox(PChar(s),NAME_PROG, MB_OK);}
          //закрашиваем часть столбца поля чёрным:
          for j:= (POLE_HEIGHT- GetLenGroup(plTOP, i, 0))
                                    to GetLenGroup(plTOP, i, 0)-1 do
          begin
            if masPole[i,j]= NUM_WHITE then //- эта клетка белая?
            begin
              s:='Неверные числа в столбце '+ inttostr(i+1)+#10#13+
              'и строке ' + inttostr(j+1) +'!';
              application.MessageBox(PChar(s),NAME_PROG, MB_OK);
              Result:= FALSE;
              exit
            end
            else if masPole[i,j]=NUM_GRAY then//- эта клетка серая?
            begin
              masPole[i,j]:= NUM_BLACK;
              inc(ReadyCells)
            end;
          end; // for j
        end; // sum - POLE_HEIGHT >0
      end; // if GetNumGroup(TOP, i)=1
    end; //if masColsNum[i,-1].StatusGroup<> stGreen
  //проверяем левое числовое поле:
  for i:= 0 to POLE_HEIGHT-1 do
    if masRowsNum[-1,i].StatusGroup<> stGreen then //- столбец ещё не решён
    begin
      //в ряду должна быть одна группа:
      if GetNumGroup(plLEFT, i)=1 then
      begin
        sum:= GetLenGroup(plLEFT, i, 0)*2;
        if sum - POLE_WIDTH >0 then //нашли
        begin
          {s:='Длинная строка '+ inttostr(i+1)+'!';
          s:= s+'beg= '+inttostr((POLE_WIDTH-GetLenGroup(plLEFT, i, 0)));
          application.MessageBox(PChar(s),NAME_PROG, MB_OK);}
          //закрашиваем часть строки поля чёрным:
          for j:= (POLE_WIDTH- GetLenGroup(plLEFT, i, 0))
                                    to GetLenGroup(plLEFT, i, 0)-1 do
          begin
            if masPole[j,i]= NUM_WHITE then //эта клетка белая?
            begin
              s:='Неверные числа в строке '+ inttostr(i+1)+#10#13+
              'и столбце ' + inttostr(j+1) +'!';
              application.MessageBox(PChar(s),NAME_PROG, MB_OK);
```

```
                Result:= FALSE;
                exit
              end
              else if masPole[j,i]=NUM_GRAY then//- эта клетка серая?
              begin
                masPole[j,i]:= NUM_BLACK;
                inc(ReadyCells)
              end;
            end; // for j
          end; // sum - POLE_WIDTH >0
        end; // if GetNumGroup(TOP, i)=1
      end; //if masRowsNum[i,-1].StatusGroup<> stGreen
    InvalidateGrids;
  end; // TestingLongLine
```

Безусловно, эта функция самая «прогрессивная» из всей четвёрки, но и она – только шаг к главному алгоритму. Теперь мы знаем, как перевести «человеческие» правила на компьютерный язык. Можно было бы продолжить наши упражнения дальше и научить компьютер пользоваться *Железными правилами*, но мы этого делать не станем, а лучше подумаем, что общего у всех *Правил* и как этим воспользоваться.

Мандрагора, или Зри в корень!

Возьмите чистую посуду…

Из анекдота про мужика, опрометчиво взявшегося за стряпню

Энэ бэнэ рес!
Квинтер финтер жес!
Энэ бэнэ ряба,
Квинтер финтер жаба…

Незнайкина считалочка

…Это первое. Второе…

Ельциновская риторика

Главную идею алгоритма решения японских кроссвордов можно вывести непосредственно из *Правил*: *если при всех возможных расстановках групп в ряду какие-либо клетки* ***всегда*** *оказываются чёрными (или белыми), то они таковыми и являются на самом деле.* Например, *Правила 1-3* потому и верны, что клетки можно расставить *единственным* способом. *Правило 4* легко проверить, если выписать все расстановки групп в ряду. Но для одной группы всё настолько очевидно, что достаточно посмотреть положение группы в крайних положениях (об этом было рассказано достаточно подробно). Случай с двумя группами в ряду также обсуждался, но он уже не столь очевиден. А в реальных задачах число групп доходит до десятка и более в одном ряду. Проверить все расстановки вручную очень трудно, по-

этому приходится «изобретать» и другие правила, которыми проще пользоваться при *ручном* решении японских головоломок. Компьютер же, напротив, «любит» не думать, а перебирать варианты расстановок. Причём делает он это настолько быстро, что ему вообще не нужны никакие правила.

Итак, первое, чему мы должны научить компьютер, - это находить все расстановки групп в ряду. Этот комбинаторный *алгоритм* очень прост:

1. Размещаем все группы *чёрных* клеток в их крайнем левом положении, а все остальные клетки ряда окрашиваем в *белый* цвет (в программе это делается иначе: сначала все клетки красим белым, а потом ставим чёрные группы; результат, естественно, от *перемены мест слагаемых* не изменится). Так мы получаем *первую* расстановку.

2. Пока это возможно, сдвигаем *последнюю группу* вправо на одну клетку. Каждый сдвиг даёт новую расстановку.

3. Если это возможно, сдвигаем *предыдущую группу* на одну клетку вправо, а все следующие выставляем через одну белую клетку за ней. Получаем ещё один вариант расстановки и переходим к *пункту 2*. Иначе - повторяем это действие.

Проиллюстрируем действие алгоритма на *примере*. Пусть длина ряда равняется 10 клеткам. В нём содержится 3 группы чёрных клеток, длиной 3, 2 и 1 клетка, соответственно. Тогда все расстановки групп легко получить, следуя алгоритму (Рис. Ф4.35).

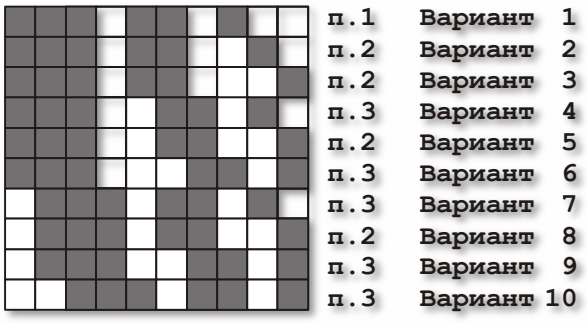


Рис. Ф4.35. Черно-белая комбинаторика

Легко заметить, что одна клетка ряда (третья слева) при *всех* расстановках групп *чёрная*, поэтому мы можем однозначно утверждать, что она именно чёрная. Цвет остальных клеток пока определить нельзя.

Так как не разгаданные клетки мы обозначаем *серым* цветом, то итоговый (или результирующий) ряд должен быть таким (Рис. Ф4.36).



Рис. Ф4.36. Одна клетка ряда решена

То есть одну клетку ряда мы смело можем (и должны!) выкрасить в *чёрный* цвет.

В процедуре *TestingOneLine*, реализующей описанный алгоритм, чтобы не запоминать *все* расстановки групп (их может оказаться очень много), сначала за итоговый ряд принимается *первая* расстановка (вариант), а затем каждая следующая (текущая) сравнивается с ней: если цвет клеток в одинаковых позициях у итоговой и текущей расстановок *не совпадают*, то клетка не может быть разгадана, то есть в итоговом ряду она *серая*, иначе цвет клетки в итоговом ряду не изменяется. Проверьте – результат окажется в точности такой же, что и в примере.

Недостаток нашего алгоритма в том, что он не учитывает уже разгаданные клетки в ряду. Допустим, что в разобранном примере разгадана начальная клетка – она *белая*. Тогда первые 6 расстановок групп можно проигнорировать, ведь в них начальная клетка *чёрная*. Отсюда следует *вывод*: для каждой расстановки нужно проверять, совместима ли она с уже известными клетками ряда. Если *чёрная* клетка текущей расстановки «накладывается» на *белую* клетку исходного ряда (того, что находится на игровом поле) или, наоборот, *белая* клетка накладывается на *чёрную*, то этот вариант расстановки не годится.

То же, но в картинках (Рис. Ф4.37).

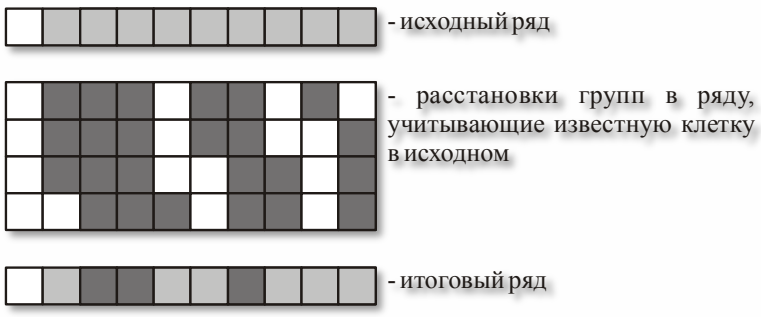


Рис. Ф4.37. Избегайте накладок!

Из этих рисунков видно, что даже одна известная клетка в исходном ряду позволяет отгадать значительно больше клеток, чем в совершенно «нетронутом», а число вариантов размещения групп сократилось до четырёх.

Если окажется, что разместить группы в ряду можно *единственным* способом (*nVar=1*), то найдено решение для *всего* ряда. Если же *ни одна* из расстановок групп (*nVar= 0*) не совместима с уже разгаданными клетками, то этот ряд не может быть решён. Такое положение возникает, когда в условии задачи имеется ошибка или ранее мы сделали неверное предположение о цвете одной из клеток (об этом мы поговорим немного погодя).

Функция TestingOneLine как раз и возвращает число возможных вариантов расстановки групп в ряду:

```
//------ ищем подходящие комбинации клеток в ряду --------
function TestingOneLine(nGroup: integer;
                        LenGroup: array of integer;
                        LenLine: integer;
                        masSource: array of integer;
                        var masResult: array of integer): integer;
//функция возвращает количество возможных расстановок групп в заданном ряду
//nGroup: integer;                                  // число (0..) групп в ряду
//LenGroup: array[0..MAX_CELLNUM-1] of integer; //длина группы
//LenLine: integer;                                 //длина ряда
//masSource: array[0..100] of integer;          //исходная полоска
//masResult: array[0..100] of integer;          //итоговая полоска
label MoveLastGroup, NextMove, PredMove;
var
  ptr: integer; // номер (0..nGroup-1) сдвигаемой группы
  pos: array[0..MAX_CELLNUM-1] of integer; //позиция первой клетки группы
  nVar: integer;                    //число найденых вариантов полосок
  masVar: array[0..100] of integer; //очередная полоска
  i: integer;
  //проверить, может ли группа разместиться в ряду
  function ExaminePos(n: integer): Boolean;
  //n - номер (0..) группы
  begin
    Result:= FALSE;
    if pos[n]+LenGroup[n]<=LenLine then Result:= TRUE;
  end;
  //записать расстановку:
  function WriteVariant(): Boolean;
  var i, j: integer;
  begin
    Result:= TRUE;
    //выставить белые клетки по длине ряда:
    for i:= 0 to LenLine-1 do masVar[i]:= NUM_WHITE;
    //расставить все группы чёрных клеток:
    for i:= 0 to nGroup-1 do
      for j:= pos[i] to pos[i]+LenGroup[i]-1 do
        masVar[j]:= NUM_BLACK;

    //скопировать первую расстановку в итоговую полоску:
    if nVar= 1 then
     for i:= 0 to LenLine-1 do masResult[i]:= masVar[i];
    //проверить, ложится ли полоска на уже имеющуюся на поле -
```

```
      //чёрная клетка текущей полоски не должна накладываться на
      //белую клетку поля, а белая - на чёрную:
      for i:= 0 to LenLine-1 do
        if (masVar[i]<> masSource[i]) and (masSource[i]<> NUM_GRAY) then
        begin
          Result:= False; exit;//- не ложится!
        end;
      //скорректировать итоговую полоску - если цвета клеток итоговой
      //и текущей полосок в одинаковых позициях разные, то
      //цвет клетки серый, иначе - без изменений
      //сравниваем полоски по всей длине:
      for i:= 0 to LenLine-1 do
        if (masVar[i]<> masResult[i]) then masResult[i]:= NUM_GRAY;
    end; // WriteVariant

  begin
    //номера начальных клеток полосок записываем
    //в массив pos:
    pos[0]:= 0;
    For i:= 1 to nGroup-1 do pos[i]:= pos[i-1]+ LenGroup[i-1]+1;
    //начальная расстановка групп:
    nVar:=1;
    if WriteVariant= FALSE then nVar:= 0;
  //начинаем сдвигать последнюю группу -
  MoveLastGroup:
    ptr:= nGroup-1;
  //передвигаем группу в след. клетку -
  NextMove:
    Inc(pos[ptr]);
    if ExaminePos(ptr) then //можно сдвигать
    begin
      inc(nVar);
      //записать расстановку:
      if WriteVariant= FALSE then dec(nVar);
      goto NextMove; //- сдвигаем до края
    end;
    //сдвигать нельзя -->
//переходим к предыдущей группе:
PredMove:
    dec (ptr);
    if ptr<0 then //- все группы сдвинуты до края
    begin
      {s:='Все варианты: '+ inttostr(nVar);
      application.MessageBox(PChar(s),NAME_PROG, MB_OK);}
      Result:= nVar;
      exit
    end;
    //ищем дальше -->
    //перемещаем начало текущей группы:
    inc(pos[ptr]);
    //проверяем:
    if ExaminePos(ptr)= FALSE then
    begin
      {s:='Все варианты: '+ inttostr(nVar);
      application.MessageBox(PChar(s),NAME_PROG, MB_OK);}
      Result:= nVar;
      exit
    end;
    //расставляем следующие группы сразу после текущей:
    For i:= ptr+1 to nGroup-1 do
    begin
      pos[i]:= pos[i-1]+LenGroup[i-1]+1;
      //проверить:
```

```
      if ExaminePos(i)= FALSE then //- для след. групп не хватает места
        goto PredMove;
    end;
    //всё нормально - записать новую расстановку:
      inc(nVar);
      if WriteVariant= FALSE then dec(nVar);
    //ищем след. растановку:
    Goto MoveLastGroup;
  end; // function TestingOneLine()
```

Эта функция обрабатывает один ряд поля (его копия передаётся в массиве *masSource*). Рассмотрим работу ещё одной функции – *TestingBrain*. Для каждого ещё не разгаданного ряда она вызывает *TestingOneLine*, а затем анализирует её значение. Если оно равно *0*, то *TestingBrain* возвращает значение *False* (не разгадана ни одна клетка), а *NoVar* устанавливает в *True* (найден нерешаемый ряд). Если значение равно *1*, то *TestingBrain* возвращает значение *True*, записывает разгаданные клетки ряда в массив *masPole* и подсчитывает число известных клеток:

```
  //проверить, можно ли ещё поставить белые и чёрные клетки в ряд
  function TestingBrain(): Boolean;
  //= TRUE, если разгадана хотя бы одна клетка
  var
    i, j, n: integer;
    ng: integer;
    LG: array[0..MAX_CELLNUM-1] of integer; //длина групп
    masSrc: array[0..100] of integer;       //исходная полоска
    masResult: array[0..100] of integer;    //итоговая полоска
  begin
    Result:= FALSE;

    //проверяем верхнее числовое поле:
    for i:= 0 to POLE_WIDTH-1 do //- по ширине поля
      //если столбец ещё не решён -->
      if masColsNum[i,-1].StatusGroup<> stGreen then
      begin
        //число групп в ряду:
        ng:= GetNumGroup(plTOP, i);
        //длина групп в ряду:
        for j:= 0 to ng-1 do lg[j]:= masColsNum[i,j].num;
        //скопировать ряд поля -> исходная полоска:
        for j:= 0 to POLE_HEIGHT-1 do masSrc[j]:= masPole[i,j];
        //получить новую полоску (ряд) на поле:
        n:= TestingOneLine(ng, lg, POLE_HEIGHT, masSrc, masResult);
        //s:='n= ' + inttostr(n)+ ' (Столбец: '+ inttostr(i);
        //application.MessageBox(PChar(s),NAME_PROG, MB_OK);
        //если нет ни одной подходящей растановки групп в ряду -->
        if n=0 then begin
          Result:= FALSE;
          NoVar:= TRUE;
          exit
        end;
        //если имеется единственная расстановка групп в ряду -->
        //ряд решён:
        if n= 1 then masColsNum[i,-1].StatusGroup:= stGreen;
        //вывести итоговую полоску на поле:
        for j:= 0 to POLE_HEIGHT-1 do
        begin
```

```
          if (masPole[i,j]= NUM_GRAY) and (masResult[j]<> NUM_GRAY) then
          begin
            Result:= TRUE;
            //ещё 1 клетка разгадана:
            inc(ReadyCells);
            masPole[i,j]:= masResult[j];
          end
        end;
        InvalidateGrids;
      end; // if masColsNum[i,-1].StatusGroup<> stGreen

    //проверяем левое числовое поле:
    for i:= 0 to POLE_HEIGHT-1 do
      //- столбец ещё не решён ->
      if masRowsNum[-1,i].StatusGroup<> stGreen then
      begin
        //число групп в ряду:
        ng:= GetNumGroup(plLEFT, i);
        for j:= 0 to ng-1 do lg[j]:= masRowsNum[j,i].num;
        for j:= 0 to POLE_WIDTH-1 do masSrc[j]:= masPole[j,i];
        n:= TestingOneLine(ng, lg, POLE_WIDTH, masSrc, masResult);
        //если нет ни одной подходящей растановки групп в ряду -->
        if n=0 then begin
          Result:= FALSE;
          NoVar:= TRUE;
          exit
        end;
        //если имеется единственная расстановка групп в ряду -->
        //ряд решён:
        if n= 1 then masRowsNum[-1,i].StatusGroup:= stGreen;
        //вывести итоговую полоску на поле:
        for j:= 0 to POLE_WIDTH-1 do
        begin
          if (masPole[j,i]= NUM_GRAY) and (masResult[j]<> NUM_GRAY) then
            begin
              Result:= TRUE;
              //ещё 1 клетка разгадана:
              inc(ReadyCells);
              masPole[j,i]:= masResult[j];
            end
        end;
        InvalidateGrids;
      end; // if masRowsNum[-1,i].StatusGroup<> stGreen
  end; //function TestingBrain()
```

Если отвлечься от частностей, то алгоритм решения японских кроссвордов получился очень простым. Конечно, для ручного гадания он непригоден, так как придётся перебирать большое число расстановок групп, а вот для машинного – *то что нужно*! Так как здесь проводится *полный* перебор размещений групп, то любая задача будет неизбежно решена (или будет установлено, что она решений не имеет). «Слабым местом» алгоритма является генерация *всех* размещений групп в ряду, даже если известно точное положение некоторых из них (разгаданные группы). Это может привести к тому, что задачи, имеющие длинные ряды и большое количество очень коротких групп (1-2 клетки), будут решаться несколько минут (обычно на это уходят секунды, причём и это время можно сильно сократить, если убрать перерисовку полей во время решения задачи, правда, то-

гда вам уже не удастся насладиться самим процессом решения). Впрочем, это не столько недостаток алгоритма, сколько изъян в самой задаче: хорошо сработанный кроссворд, как правило, имеет в рядах не более 5-7 групп, поэтому ответ будет найден очень быстро. В откровенно слабых заданиях встречаются ряды с десятком и более одноклеточных групп. Сделаны они наверняка не для компьютера, и нетрудно себе представить адские муки разгадчика, часами отыскивающего места для этих групп, чтобы в итоге получить банального ёжика, раскрашенного, как шахматная доска. Если вы планируете самостоятельно составлять японские кроссворды, *будьте добрее к людям, не мучайте их понапрасну*!

Как я уже отмечал, этот алгоритм с лихвой перекрывает все написанные и не написанные ещё правила решения кроссвордов, поэтому в программе можно оставить только его, а функции, реализующие *Правила*, просто удалить. И хотя в этом нет никакой необходимости, но вы можете попытаться *улучшить* предложенный алгоритм. Например, если число *серых* клеток в итоговом ряду сравняется с числом таких клеток в исходном ряду, то нет смысла дальше перебирать варианты размещения групп – ни одну клетку ряда в этот раз разгадать не удастся.

И нам осталось рассмотреть автоматическое решение задач в целом, то есть узнать, что происходит в программе после нажатия на кнопку *sbtStart*.

Магическая кнопка, или Первые аплодисменты

Кто ищет, тот всегда найдёт.

Из оптимистической песни

В первую очередь, нужно изменить режим работы программы - *SetStatus('ПОИСК');* Теперь все кнопки будут «знать», что программа занята важным делом, и не смогут ей помешать. Для удобства мы напишем небольшую *процедуру*:

```
//УСТАНОВИТЬ СТАТУС ПРОГРАММЫ
procedure TForm1.SetStatus (s: string);
begin
  status:= s;
  StatusBar1.Panels[5].text := s;
  application.ProcessMessages;
end;
```

Не забудьте объявить её в разделе *private*:

```
procedure SetStatus (s: string);
```

Затем мы проверяем правильность данных задачи и подсчитываем число уже разгаданных клеток. Поначалу кажется, что этого и делать не нужно, ведь задачу мы ещё не решали, но не торопитесь – вы могли начать решение вручную, а потом, отчаявшись, предоставили свою судьбу машине.

Часть клеток можно разгадать, воспользовавшись *Правилами*, реализованными в рассмотренных ранее функциях.

На этом лирическая часть решения завершается, и начинаются трудовые будни функции *TestingBrain*. Если она «опознала» хотя бы одну клетку, значит, ситуация на поле изменилась, и мы можем попытаться продвинуться дальше в своём решении, опять же вызывая *TestingBrain*. Так как решение может и затянуться (или вы просто передумаете решать кроссворд), то следует позаботиться о выходе из этой круговерти событий. Самый правильный способ – нажать кнопку *sbtStop*, которая установит флаг выхода из процедуры. Ообъявите *переменную* так же, как и раньше:

```
flgExit: boolean= False;
```

А уж если вы очень торопитесь, то просто закройте программу.

```
//ОСТАНОВИТЬ РЕШЕНИЕ ЗАДАЧИ
procedure TForm1.sbtStopClick(Sender: TObject);
begin
  if status<>'ПОИСК' then exit;
  flgExit:=true
end;

//ЗАКРЫТЬ ПРОГРАММУ
procedure TForm1.FormClose(Sender: TObject; var Action: TCloseAc-
tion);
begin
  flgExit:= true;
end;
```

Гоняя функцию *TestingBrain* по кругу, мы должны на каждом «витке» проверять, не разгадала ли она часом уже *все* клетки. Рано или поздно так и случится, и вот тут-то вы и увидите, что часть рядов «не решена», то есть не закрашена **зелёным** цветом. Это свидетельствует о том, что некоторые числа вообще не нужны для решения задачи. Но раз уж они есть, мы должны убедиться, что они «правильные». Решая чужую задачу, вы сможете позлорадствовать над горе-составителем, если найдёте ошибки, а, проверяя свою собственную, избежите участи предыдущего товарища.

И вот он – миг победы: всё сошлось, и задача покорно пала к вашим ногам. Если вы успели нажать кнопку *sbtSound*, то услышите в свою честь аплодисменты (не смущайтесь – вы их заслужили!).

Этим приятным звуковым эффектом заканчивается решение всех задач (конечно, «непорочных»), да не всегда так скоро. Я уже предупреждал вас о коварных составителях неудобоваримых задач, решение которых не обходится без русского авось, или – по-научному – тыка. Если *TestingBrain* вернётся ни с чем (найдёт коса на камень, и ни одна клетка не будет разгадана), то нам ничего другого не останется, как задать цвет какой-нибудь клетки насильно, а затем продолжить решение. Если ситуация повторится, придётся сделать ещё одно предположение и так далее (разумно ограничить число предположений, чтобы не тратить время на неудачные задачи. В этой программе предельное число предположений равно 20. Объявите константу *MAX_LEVEL*:

```
MAX_LEVEL= 20; //макс. сложность задачи (число предположений))
```

Текущий *уровень сложности* задачи хранится в переменной *Level*:

```
Level: Integer=0; //уровень сложности задачи (число предположений)
```

Но как выбрать сомнительную клетку, и в какой цвет её закрасить? – Мы поступим просто – первую попавшуюся от начала поля *серую* клетку сделаем *чёрной* (если предположение окажется неверным, то перекрасим её в *белый* цвет – других вариантов нет). Любители сложных задач могут поискать и более эффективный алгоритм подобных предположений. Другой вопрос: как мы сможем найти сомнительную клетку, если с её цветом не угадали, и как восстановить позицию на поле в тот судьбоносный момент? – Нужно запомнить всю необходимую информацию в массиве *SaveLevel* (объявите его там же, где и все переменные):

```
  //массив, в котором хранятся данные предыдущих уровней:
  SaveLevel: array[0..MAX_LEVEL] of TSaveLevel;
```

А хранить в нём мы будем вот что:

```
//данные уровня
type TSaveLevel = Record
  //статус рядов верхнего числового поля:
```

```
  TopStatus: array[0..MAX_POLE_WIDTH-1]of TStatusGroup;
  //статус рядов левого числового поля:
  LeftStatus: array[0..MAX_POLE_HEIGHT-1]of TStatusGroup;
  //копия поля:
  masPole: TPole;
  //коорд. "сомнительной" клетки
  XCell: integer;
  YCell: integer;
  //число разгаданных клеток:
  ReadyCells: integer;
end;
```

Этого вполне достаточно, чтобы безболезненно вернуться в случае надобности к нашим ошибкам и взяться за новые.

Но, даже решив задачу, в которой делались предположения, мы не можем быть уверены, что решили её правильно. То есть решение, безусловно, будет удовлетворять условиям задачи, но картинка получится неверной. Чтобы никого не обидеть, в качестве примера дуалей приведу тестовую задачу (Рис. Ф4.38).

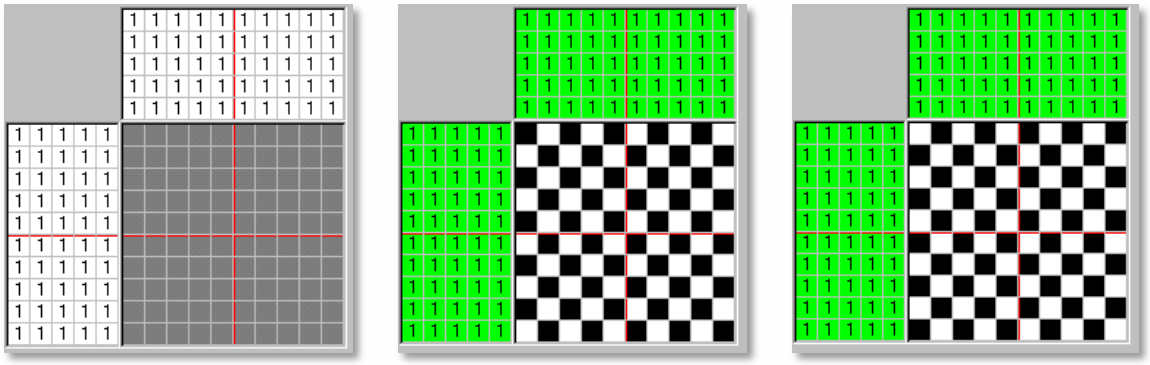


Рис. Ф4.38. И так, и сяк, и в клеточку!

Вы легко можете убедиться, что оба решения *верные*, хотя автор мог иметь в виду только одно из них. Мы не должны винить в этом нашу программу – она всё сделала правильно, а вся ответственность ложится исключительно на составителей. Задачи, имеющие побочные решения, это плохие задачи, и характеризуют автора не с лучшей стороны, поэтому обязательно проверяйте свои кроссворды, чтобы избежать досадных ляпсусов.

В процедуре, обрабатывающей нажатие кнопки *sbtStart*, текст некоторых функций заменён многоточием, так как он уже приводился полностью раньше. Что касается работы процедуры, то мы разобрали её достаточно подробно, чтобы у вас не возникало сомнений.

```
//================================================
//============== РЕШИТЬ ЗАДАЧУ ==================
//================================================
procedure TForm1.sbtStartClick(Sender: TObject);
label again;
var
  i, j: integer;
  s: string;
  NoVar: boolean;
  numVar: integer;   //число найденных вариантов решения задачи
  maxLevel: integer; //сложность решения задачи

  //проверить данные задачи
  function Testing(): Boolean ;
  . . .
  end; // Testing

  //проверить, не все ли клетки разгаданы:
  function IsReady(): Boolean;
  begin
    lblReady.Caption:= inttostr(ReadyCells);
    Result:= False;
    if ReadyCells= AllCells then Result:= TRUE; //готово!
  end;

  //отмечаем пустые строки
  procedure TestingZero;
  . . .
  end; // TestingZero

  {//отмечаем полные строки
  function TestingFullLine(): Boolean;
  . . .
  end; // TestingFullLine}

  //отмечаем строки с полными суммами
  function TestingFullSum(): Boolean;
  . . .
  end; //TestingFullSum

  //отмечаем длинные строки
  function TestingLongLine(): Boolean;
  . . .
  end; // TestingLongLine

  //------ ищем подходящие комбинации клеток в ряду --------
  function TestingOneLine(nGroup: integer;
                          LenGroup: array of integer;
                          LenLine: integer;
                          masSource: array of integer;
                          var masResult: array of integer): integer;
  . . .
  end; //function TestingBrain()

  //сохранить данные заданного уровня
  procedure SaveDataLevel(n: integer);
  //n - номер уровня (0..)
  var i, j: integer;
  begin
    //сохранить статус рядов верхнего числового поля:
    for i:= 0 to POLE_WIDTH-1 do
      SaveLevel[n].TopStatus[i]:= masColsNum[i,-1].StatusGroup;
    //сохранить статус рядов левого числового поля:
```

```
    for i:= 0 to POLE_WIDTH-1 do
      SaveLevel[n].LeftStatus[i]:= masRowsNum[-1,i].StatusGroup;
    //сохранить копию поля:
    SaveLevel[n].masPole:= masPole;
    //заменить первую серую клетку на чёрную:
    inc(ReadyCells);
    //сохранить число разгаданных клеток:
    SaveLevel[n].ReadyCells:= ReadyCells;
    for j:= 0 to POLE_HEIGHT-1 do
    for i:= 0 to POLE_WIDTH-1 do
      if masPole[i,j]= NUM_GRAY then begin
        masPole[i,j]:= NUM_BLACK;
        //сохранить координаты "сомнительной" клетки:
        SaveLevel[n].XCell:= i;
        SaveLevel[n].YCell:= j;
        exit
      end;
  end;  //SaveDataLevel

  //загрузить данные заданного уровня
  procedure LoadDataLevel(n: integer);
  //n - номер уровня (0..)
  var i: integer;
  begin
    //загрузить статус рядов верхнего числового поля:
    for i:= 0 to POLE_WIDTH-1 do
      masColsNum[i,-1].StatusGroup:= SaveLevel[n].TopStatus[i];
    //загрузить статус рядов левого числового поля:
    for i:= 0 to POLE_WIDTH-1 do
      masRowsNum[-1,i].StatusGroup:= SaveLevel[n].LeftStatus[i];
    //загрузить копию поля:
    masPole:= SaveLevel[n].masPole;
    //заменить чёрную клетку на белую:
    masPole[SaveLevel[n].XCell,SaveLevel[n].YCell]:= NUM_WHITE;
    //загрузит число разгаданных клеток:
    ReadyCells:= SaveLevel[n].ReadyCells;
  end;  //LoadDataLevel

//
//==================== Р Е Ш А Е М  З А Д А Ч У ======================
//
begin
  if status='ПОИСК' then exit; //- задача уже решается
  SetStatus('ПОИСК');
  //выключить режимы перемещения и рисования при решении задачи:
  sbtDraw.Down:= FALSE;
  sbtMove.Down:= FALSE;

  numVar:=0;
  maxLevel:= 0;

  //проверить данные фигуры:
  if Testing=FALSE then begin SetStatus('ОЖИДАНИЕ'); exit end;// - ошибка!

  //ни одна клетка пока не разгадана:
  ReadyCells:= 0;
  //подсчитать число уже разгаданных клеток:
  for j:= 0 to POLE_HEIGHT-1 do
    for i:= 0 to POLE_WIDTH-1 do
      if masPole[i,j]<>NUM_GRAY then inc(ReadyCells);
  lblReady.Caption:= inttostr(ReadyCells);

  //начинаем решение задачи с нулевого уровня:
```

```
  Level:= 0; lblLevel.Caption:= inttostr(Level);

  //ряды, в которых есть нулевые группы, можно сразу закрасить белым:
  TestingZero;

  //ряды, в которых есть число, равное размеру поля, можно сразу
  //закрасить чёрным:
  //if TestingFullLine=FALSE then begin SetStatus('ОЖИДАНИЕ'); exit end;

  //если (сумма клеток в группах) + (сумма клеток в группах-1) =
  //длине ряда, то этот ряд разгадан:
  if TestingFullSum=FALSE then begin SetStatus('ОЖИДАНИЕ'); exit end;

  //если единственная группа в ряду длиннее половины ряда,
  //то часть ряда можно закрасить чёрным:
  if TestingLongLine=FALSE then begin SetStatus('ОЖИДАНИЕ'); exit end;

again:
  NoVar:= FALSE;
  //пока в TestingBrain будет закрашена хотя бы одна клетка,
  //продолжаем решение задачи:
  while TestingBrain do begin
    application.ProcessMessages;
    if flgExit=true then begin
      flgExit:=false;
      SetStatus('ОЖИДАНИЕ'); exit
    end;
    if IsReady then //- все клетки закрашены!
    begin
      //проверить, все ли ряды решены
      //если не все - продолжить:
      for i:= 0 to POLE_WIDTH-1 do
        if masColsNum[i,-1].StatusGroup<> stGreen then goto again;
        break; // - все ряды решены
    end;
  end;

  if IsReady and (NoVar=FALSE) then //- готово!
  begin
    inc(numVar);
    WavPobeda;
    s:='Задача решена!';
    if numVar> 1 then s:= s+ #10#13+ ' Вариант - ' + inttostr(numVar);
    //сложность решения задачи:
    s:= s+ #10#13+ ' Сложность = ' + inttostr(maxLevel+1);
    application.MessageBox(PChar(s),NAME_PROG,MB_OK );
    if Level = 0 then begin
      SetStatus('ОЖИДАНИЕ'); exit
    end;
    //могут быть варианты:
    if application.MessageBox('Ищем варианты?',NAME_PROG, MB_YESNO)= ID_NO
    then begin
      SetStatus('ОЖИДАНИЕ'); exit
    end
    else NoVar:= TRUE
  end;
  //не удалось решить задачу на этом уровне
  if NoVar= TRUE then begin
    dec(Level);
    if Level< 0 then begin
      if numVar= 0 then
        s:='Задача решений не имеет!'
      else
```

```
        s:='Найдены все варианты решения задачи - '+ inttostr(numVar);
      application.MessageBox(PChar(s),NAME_PROG,MB_OK );
      SetStatus('ОЖИДАНИЕ'); exit
    end;
    //уровень сложности:
    lblLevel.Caption:= inttostr(Level);
    //загружаем данные предыдущего уровня:
    LoadDataLevel(Level);
    lblReady.Caption:= inttostr(ReadyCells);
    goto again
  end;

  //переходим на следующий уровень -
  //запомнить данные текущего уровня и
  //заменить первую серую клетку на чёрную:
  SaveDataLevel(Level);

  inc(Level);
  if Level > MAX_LEVEL then begin
    s:='Слишком сложная задача!';
    application.MessageBox(PChar(s),NAME_PROG,MB_OK );
    SetStatus('ОЖИДАНИЕ');
    exit
  end;
  lblLevel.Caption:= inttostr(Level);
  if Level > maxLevel then maxLevel:= Level;
  goto again
end; // sbtStartClick
```

Чтобы защитить программу от шаловливых ручек (из-за них и самолёты падают!), следует запретить нажимать кнопки *sbtMove* и *sbtDraw* во время решения задачи:

```
//НЕ ВКЛЮЧАТЬ РЕЖИМ ПЕРЕМЕЩЕНИЯ ПРИ РЕШЕНИИ ЗАДАЧИ
procedure TForm1.sbtMoveClick(Sender: TObject);
begin
  if status='ПОИСК' then sbtMove.Down:= FALSE;
end;

//ПЕРЕКЛЮЧИТЬ РЕЖИМ: ВВОД ЗАДАЧИ - РЕШЕНИЕ ЗАДАЧИ
procedure TForm1.sbtDrawClick(Sender: TObject);
begin
  if status='ПОИСК' then sbtDraw.Down:= FALSE;
end;
```

Мастерская Самоделкина, или Наши весёлые картинки

Берите в руки карандаш…

Кавээновская песенка

Когда вы перерешаете сотню-другую японских кроссвордов и у вас возникнет стойкое отвращение к этим головоломкам, попробуйте продлить очарованье и подобно киношному Ивану Васильевичу смените «профессию»: вместо того, чтобы решать чужие задачи, *попридумывайте* свои.

В программе есть всё необходимое для осуществления ваших творческих порывов, всё, за исключением, пожалуй, главного – рисовать картинки придётся вам самим, тут вам никакой компьютер не поможет. Зато с технической стороны нет никаких препятствий. Живописать можно аналогично тому, как вы вручную решали задачи. Только теперь над вами не довлеет «групповщина» чужих клеток, и вы целиком можете предаться воплощению своих самых смелых фантазий.

Нажмите кнопку *sbtNewFig*  и установите размеры поля побольше (их всегда можно безнаказанно изменить в процессе работы, так что не впадайте в истерику, если монументальность вашего полотна затмит разум), очистите его от всякой дряни (рекламное словечко, удачно ввернул!) кнопкой *sbtWhiteGrid* . Включите режим рисования кнопкой *sbtDraw*  и приступайте к своим художествам. Чёрные точки и линии наносятся на «холст» *левой* кнопкой мышки, а роль ластика играет *правая* кнопка того же грызуна (ежели у вас толчковая – правая и вы изменили настройки, то тогда, сами понимаете, всё наоборот).

Если вы неудачно разместили картинку на поле, то вдавите поглубже кнопку *sbtMove*  и, нажав *левую* кнопку мыши на игровом поле, двигайте её в нужном направлении.

Удовлетворившись содеянным, подгоните размеры поля под картинку так, чтобы не осталось пустых рядов, и сохраните свою работу на диске. Для этого вы можете оцифровать рисунок кнопкой *sbtNumbers*  и записать только числа (условие задачи) кнопкой *sbtSaveFig* . Но есть и альтернативный вариант – сохранить именно *картинку*. Если вы были внимательны, то помните, что кнопка *sbtSavePic*  до сих пор ничем себя не проявила. А всё потому, что она ждала своего часа, который, наконец, пробил.

По тем же причинам, что и раньше, мы не станем изобретать собственного формата для сохранения готового рисунка, а воспользуемся форматом *JCW* из программы Гантверга *Japan2*, тогда ваши задачи можно будет решать и в ней. Но имейте в виду, что файл *JCW* содержит не условие задачи, а только ответ (копию картинки с игрового поля) – для проверки решения (такие файлы используются при ручном разгадывании). Наша же программа при загрузке файлов *JCW* одновременно формирует и условие задачи, которое всегда можно записать в формате *JCP*, то есть из рисунка сделать головоломку. Тем, кто интересуется, как всё это шевелится, советую посмотреть процедуру *LoadJcwFile*, которая находится в обработчике нажатия на кнопку *sbtLoadFig*  - *procedure TForm1.sbtLoadFigClick(Sender: TObject);*.

После всех катавасий и круговертей вам не составит особого труда разобраться в технологических процесах, происходящих при поглаживании кнопки *sbtSavePic*:

```
//ЗАПИСАТЬ КАРТИНКУ
procedure TForm1.sbtSavePicClick(Sender: TObject);
var
  F: file of byte;
  fn,s: string;
  i,j: integer;
  b: byte;
begin
  if status='ПОИСК' then exit;
  //расширение файлов рисунков:
  savedialog1.DefaultExt:='jcw';
  savedialog1.Filter:='Japan puzzle (*.jcw)|*.JCW';
  //записываем в каталог 'FIGURE':
  s:=extractfilepath(application.exename)+'FIGURE\';
  savedialog1.InitialDir:= s;
  savedialog1.Title:='Запишите рисунок на диск';
  savedialog1.filename:= NameFig;
  if not savedialog1.Execute then exit;
  //имя конечного файла:
  fn:= savedialog1.filename;
  //изменить его, если при записи было выбрано другое имя:
  fn:=ChangeFileExt(fn, '.jcw');
  NameFig:=fn;
  assignfile(f,fn);
  rewrite(f);
  //записать фигуру -
  //высота и ширина фигуры:
  write (f, POLE_WIDTH);
  write (f, POLE_HEIGHT);
  for i:= 0 to POLE_WIDTH-1 do
    for j:= 0 to POLE_HEIGHT-1 do
    begin
      if masPole[i,j]= NUM_BLACK then b:= 2 else b:= 0;
```

```
      write (f, b);
    end;
  //закрыть файл:
  closefile(f);
  form1.caption:= NAME_PROG + '  [' + NameFig + ']';
  form1.Refresh ;
  messagebeep(0)
end; // sbtSavePicClick
```

Чтобы полнее использовать возможности программы по конвертированию рисунков в задачи, в ней предусмотрена загрузка растровых картинок *BMP* и значков *ICO* в игровое поле. Одновременно изображение появится и на компоненте *Image1* (*frmMemo*), чтобы оригинал всегда был у вас перед глазами.

Предупреждения.

1. Избегайте картинок больших размеров, всё равно вы не сможете их загрузить. В данной версии программы максимальные размеры растров не должны превышать 70 x 50 пикселей (вы, конечно, можете и «раздвинуть границы», если пожелаете).

2. Если картинки не монохромные, то все цвета за исключением белого и чёрного будут заменены *серым*. Вы должны перекрасить их в чёрный и белый цвет так, чтобы получился рисунок. Поручить это компьютеру нельзя, он скорее маляр, чем художник, так что результаты почти наверняка окажутся отвратительными.

А теперь полюбуйтесь, как в программе совершается таинство преобразования глупого растра в полную глубокого смысла картинку на поле:

```
//ЗАГРУЗИТЬ РАСТРОВУЮ КАРТИНКУ ИЛИ ЗНАЧОК
procedure TForm1.sbtOpenPictureClick(Sender: TObject);
var
  w, h: integer;
  i, j: integer;
  bmp: TBITMAP;
begin
  frmMemo.Image1.Visible:= FALSE;
  OpenPictureDialog1.DefaultExt:= 'BMP';
  OpenPictureDialog1.InitialDir:= extract-
filepath(application.exename);
  OpenPictureDialog1.Title:='Загрузите картинку';
  OpenPictureDialog1.Filter:= GraphicFil-
ter(TBitmap)+'|'+GraphicFilter(TIcon);
  if not OpenPictureDialog1.Execute then exit;
```

```
  //имя файла:
  NameFig:= OpenPictureDialog1.FileName;
  //загрузить картинку:
  frmMemo.Image1.Picture.LoadFromFile(NameFig);
  if OpenPictureDialog1.FilterIndex= 2 then begin //- значок
    bmp:= TBitmap.Create;
    bmp.Width:= frmMemo.Image1.Picture.icon.Width;
    bmp.Height:= frmMemo.Image1.Picture.icon.Height;
    bmp.Canvas.Draw(0,0,frmMemo.Image1.Picture.icon);
    frmMemo.Image1.Picture.Bitmap.Assign(bmp);
    bmp.Free;
  end;
  w:= frmMemo.Image1.Picture.Width;
  h:= frmMemo.Image1.Picture.Height;
  //проверить размеры картинки:
  if (h > MAX_POLE_HEIGHT) or (w > MAX_POLE_WIDTH)then begin
    application.MessageBox('Слишком большая картинка!',NAME_PROG,
MB_OK);
    exit
  end;
  frmMemo.Image1.Visible:= TRUE;
  POLE_WIDTH:= w; POLE_HEIGHT:= h;
  TOP_POLE_HEIGHT:= 2; LEFT_POLE_WIDTH:= 2;
  Prepare(POLE_WIDTH,POLE_HEIGHT, TOP_POLE_HEIGHT,
LEFT_POLE_WIDTH);
  //очистить цифровые поля:
  Clear_masColsNum;
  Clear_masRowsNum;
  for j:= 0 to POLE_HEIGHT-1 do
    for i:= 0 to POLE_WIDTH - 1 do begin
      case frmMemo.Image1.Picture.Bitmap.Canvas.Pixels[i,j] of
        clBlack: masPole[i,j]:= NUM_BLACK;
        clWhite: masPole[i,j]:= NUM_WHITE;
        else masPole[i,j]:= NUM_GRAY;
      end;
  end;
  //вывести в заголовке формы имя загруженного файла:
  form1.caption:= NAME_PROG + '  [' + NameFig + ']';
  dgPole.Invalidate
end; // sbtOpenPictureClick
```

Раз уж вы можете загружать в программу растры и значки, то лучше создавать картинки-задания в каком-нибудь растровом графическом редакторе или в редакторе иконок (их великое множество, выбирайте на свой вкус!). Они значительно удобнее и мощнее, чем наша скромная в этом отношении программа (всё-таки не царское это дело, рисование; вот и Остап Ибрагимович рисовал как курица лапой, а ведь был милейшим человеком).

А подготовив рисунок, просто загрузите его кнопкой *sbtOpenPicture* , а затем сохраните как картинку *JCW* или как задачу *JCP* для последующего применения.

Вы легко найдёте массу небольших картинок на своём диске или в Интернете, которые вполне можно, приложив руки и сердце, переделать в японские кроссворды. На первых порах это невинное занятие поможет вам приобрести необходимый для создания будущих шедевров опыт.

А вот наглядный пример хорошего отношения к людям. Слева показана форма, в которую загружен значок с изображением колобка. Справа – он же, но раскрашенный и оцифрованный (Рис. Ф4.39).

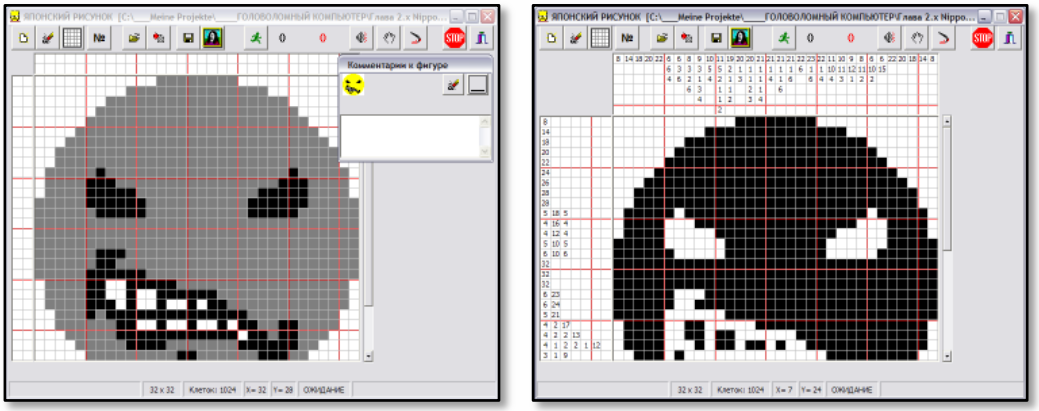


Рис. Ф4.39. Оцифровали!

Лёгким движением руки значок превратился - в японский кроссворд (знали бы японцы, что мы сделаем с их творением!). Тот, кто способен на большее, чем примитивный «плагиат», может изувечить оригинал до полной неузнаваемости. А те и другие сообща имеют полное право наклонировать теперь бешеное количество задач, не имеющих практически никаких достоинств.

«Шлите апельсины бочками», или «Ждите ответа!»

Ищите – и обрящете!

Библейское

Не очень-то увлекательно составлять задачи для самого себя, лучше позабавить ими кого-нибудь другого. Благо, существует немало изданий (смотри выше), отдающих должное японским кроссвордам. Вы вполне можете послать свои работы в редакцию и, если вам удалось в грубых квадратах выразить свои изящные замыслы, у вас есть последняя надежда увидеть свои кроссворды на страницах какой-нибудь газеты. Это поднимет ваш боевой дух и толкнёт вас на новые опрометчивые поступки в области сочинения головоломок.

Если же вас утомит сам процесс разработки задания, то просто распечатайте готовый рисунок на принтере, скопировав его предварительно в любой графический редактор. Этого вполне достаточно, чтобы послать задание в газету. А можете и красиво оформить его. Например, чудесно для этого подходит векторный графический редактор *CorelDraw*.

Засим позвольте откланяться – наш художественный салон закрывается.

«А напоследок я скажу…»

На этом можно и закончить изучение программы для составления и решения японских кроссвордов – теперь вы знаете о ней всё. Осталось - на всякий случай – привести начало программы, где объявляются константы, переменные, типы, процедуры и функции, так как это делалось на протяжении многих страниц, и вы могли что-нибудь упустить в пылу борьбы за урожай(ность).

```
const
  NAME_PROG = 'ЯПОНСКИЙ РИСУНОК';
  MAX_POLE_WIDTH = 70;   //макс. ширина поля в клетках
  MAX_POLE_HEIGHT = 50;  //макс. высота поля в клетках
  NUM_WHITE = 2    ;     //пустая клетка в masPole
  NUM_BLACK = 1;         //чёрная клетка в masPole
  NUM_GRAY= 0;           //клетка серая в masPole (фон)
  COLOR_WHITE: TColor= $FFFFFF;  //белая клетка на поле
  COLOR_BLACK: TColor= $0;       //чёрная клетка на поле
  COLOR_GRAY: TColor= $7F7F7F;   //серая клетка на поле
  MAX_CELLNUM = 14;         //макс. ширина/высота числовых полей
10
  MAX_ALL_CELLS_HEIGHT = 30;//макс. высота игрового и числового
полей
                            //(видимая часть)
  MAX_ALL_CELLS_WIDTH = 46; //макс. ширина игрового и числового
полей
  //курсоры:
  crHand :integer= 4;
  crMove :integer= 5;
  crKistj :integer= 7;
  MAX_LEVEL= 20;       //макс. сложность задачи (число предположе-
ний)

//данные игрового поля
type TPole = array[0..MAX_POLE_WIDTH-1, 0..MAX_POLE_HEIGHT-1]of
Integer;
//статус групп в числовых полях:
//stWhite - группа не решена,
//stYellow - группа решена,
//stGreen - ряд решён
```

```
type TStatusGroup=(stWhite, stYellow, stGreen);
//данные в каждой клетке числовых полей:
type TCell = Record
  sNum: String; //строковое представление числа в клетке
   Num: integer;//число в клетке
   StatusGroup: TStatusGroup; //статус группы
end;
//положение числового поля: слева, сверху
type TPoleLocation= (plLEFT, plTOP);
//данные уровня
type TSaveLevel = Record
  //статус рядов верхнего числового поля:
  TopStatus: array[0..MAX_POLE_WIDTH-1]of TStatusGroup;
  //статус рядов левого числового поля:
  LeftStatus: array[0..MAX_POLE_HEIGHT-1]of TStatusGroup;
  //копия поля:
  masPole: TPole;
  //коорд. "сомнительной" клетки
  XCell: integer;
  YCell: integer;
  //число разгаданных клеток:
  ReadyCells: integer;
end;

var
  //массив, в котором хранится копия поля
  masPole : TPole;
  POLE_WIDTH: integer = 28;     //ширина поля в клетках
  POLE_HEIGHT: integer = 28;    //высота поля в клетках
  //массив, в котором хранятся верхние числа:
  masColsNum : array[0..MAX_POLE_WIDTH-1, -1..MAX_CELLNUM-1]of
TCell;
  //массив, в котором хранятся левые числа:
  masRowsNum : array[-1..MAX_CELLNUM-1, 0..MAX_POLE_HEIGHT-1]of
TCell;
  LEFT_POLE_WIDTH: integer = 2; //ширина левого числового поля в
клетках
  TOP_POLE_HEIGHT: integer = 2; //высота верхнего числового поля в
клетках
  status: string='';
  NameFig: string= 'temp';
  //координаты клетки с курсором:
  cellMouse: TPoint;     //игровое поле
  cellMouseTop: TPoint;  //верхнее числовое поле
  cellMouseLeft: TPoint; //левое числовое поле
  AllCells: Integer=0;   //всего клеток на поле
  ReadyCells: Integer=0; //разгадано клеток
  flgExit: boolean= False;
  Level: Integer=0;      //уровень сложности задачи (число предпо-
ложений)
  //массив, в котором хранятся данные предыдущих уровней:
  SaveLevel: array[0..MAX_LEVEL] of TSaveLevel;
```

```
type
  TForm1 = class(TForm)
     . . .
  private
    { Private declarations }
    procedure Clear_masColsNum;
    procedure Clear_masRowsNum;
    procedure InvalidateGrids;
    procedure MoveNums;
    procedure WavError;
    procedure WavReadyLine;
    procedure WavPobeda;
    function GetNumGroup(Location: TPoleLocation; nLine: integer):
integer;
    function GetLenGroup(Location: TPoleLocation; nLine, nGroup:
integer)
              : integer;
    procedure SetStatus (s: string);
  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation

uses MemoUnit, NewFigUnit;

{$R *.DFM}
. . .
```



Исходный код программы находится в папке **NIPPON**.

Генеалогическое древо, или Японская родня



Гора родила мышь.

Нашему забору двоюродный плетень.

Из мудрых мыслей и афоризмов

Как и всякая другая известная головоломка, японский кроссворд имеет бедных родственников, к счастью, не столь многочисленных. Пока все дома, давайте познакомимся с несколькими из них.

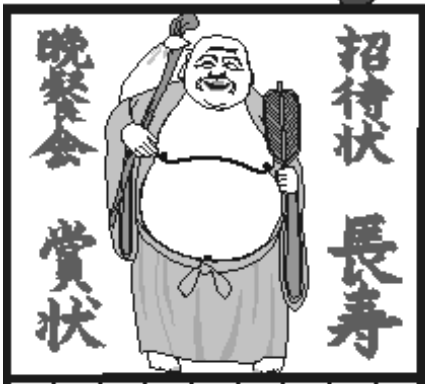
В последние годы японский кроссворд мутировал, породив химеру – *цветной* рисунок, в котором клетки могут быть не только чёрными и белыми, но и вообще любых мыслимых (и немыслимых) цветов. Второе отличие от классического рисунка состоит в том, что между группами цветных клеток прослойки в виде белых клеток не обязательны.

Решается подобное чудо природы практически так же, как и обычная задача, хотя забава куда как веселее, ведь для раскрашивания клеток придётся держать в руках целый пучок карандашей!

Так как цветные рисунки не имеют отношения к нашим досугам, то писать программу для их решения мы не станем. Если же вам неймётся, не терпится и не хочется ждать милостей от мерзкой природы – милости прошу, займитесь-ка этим сами. Чтобы не изобретать велосипед (или того хуже – самокат), попробуйте «уговорить» нашу программу на «цветное» решение. Но будьте готовы к большой работе!

Вторая, более светлая мысль авторов привела к *гибридизации*, то есть к скрещиванию японского кроссворда с обычным (который, если вы не знаете, является для нас важнейшим из всех искусств). Метис получился презанятным, но бесплодным. Посему, не выдержав напора эволюции, сгинул, как и все прочие динозавры. Так как автор этих строк и междустрочий тоже принимал непосредственное участие в выведении несуразных монстров, то – исключительно в кунсткамерных целях – решился на демонстрацию своей напечатанной в одной из московских газет работы (Рис. Ф4.40).

"ЯПОНСКИЙ" КРОССВОРД



8	1	7	1	7	1	1	1	5	1	1	6	1	1	5	1	5	1	5	1	9
8	1	1	1	1	1	9	1	1	6	1	4	1	1	1	1	9	4	4	1	8
	1	1	6	8	1	1	1	1	1	1	1	1	7	1	1	1	1	7	1	
	1	6	1		1	1	1	1	1	9	1	1	1	5	1	1	1		1	
	1		1		1		6	1	1			1			1		1		1	
	1		1		1			1	1			6			1		1			
	1														1		1			
	1																			

8	10							
1	1	1	1	1	1	1	1	1
6	7	1	1	1				
1	1	1	1	1	1	1	1	1
1	1	1	4	1	7			
1	1	1	1	5	1	1		
7	1	5	1					
1	1	8	5					
7	1	1	1	1	1	1		
1	1	1	1	1	1	1		
4	1	9						
1	8	1	1	1				
1	1	1	1	5				
5	6	1	1	1	1			
1	1	1	1	1	1	5	1	
1	1	7	1	1	1	1		
5	1	1	7	1				
1	1	1	1	1	1	1	1	1
4	7	6						

По правилам головоломки "Японский рисунок" восстановите сетку кроссворда, а затем заполните её словами, определения которых даны в произвольном порядке.

4-буквенные слова: 1. "Пуля - дура, ... - молодец" (Суворовское). 2. Необъяснимое явление. 3. Английский парный танец. 4. "Воздухоплаватель", оторвавшийся от коллектива. 5. Резкое снижение самолёта. 6. Жёлоб.

5-буквенные слова: 7. Съедобный гриб "хитрого" цвета. 8. Начало реки. 9. Схватка за урожай. 10. Руководитель факультета. 11. Аркан. 12. Тара для переноски котов. 13. Осёл и катапульта. 14. Вересковый кустарник. 15. Резвый, игривый ребёнок. 16. Землечерпалка. 17. Обезжиренное молоко. 18. Инструмент для прополки.

6-буквенные слова: 19. Заболевание суставов. 20. Детёныш яка. 21. Гигант на глиняных ногах. 22. Радуга как набор цветов. 23. Бегун-дальнобойщик. 24. Поэма Лермонтова о предыдущем пункте. 25. Сигаретный "бычок". 26. Угнетённое состояние. 27. Скульптура.

7-буквенные слова: 28. Вирусная болезнь глаз. 29. Участник корриды с копьём. 30. Наука для проверки гармонии. 31. Осколочная ... 32. Неявка на работу. 33. Невезение, провал. 34. Маленькая ... до старости щенок (посл.). 35. Охотничья плеть. 36. Приучение лошади к езде. 37. Писатель, который не попадает в рифму. 38. "Поцелуй же меня, кума-..." (песенное).

8-буквенные слова: 39. Праздник в Рио. 40. Один из тех, что съели Кука. 41. На все ноги мастер. 42. Комнатный беспорядок. 43. Чиполлино как ботанический объект. 44. Прослойка между бедняком и кулаком. 45. Каждый из неуловимых.

9-буквенные слова: 46. Визави сталагмита. 47. Жительница чёрного континента. 48. Невосприимчивость к заразе. 49. Жительница одной из стран Юго-Восточной Азии. 50. Периодическое издание.

Ответ.
1.Штык. 2.Чудо. 3.Шейк. 4.Икар. 5.Пике. 6.Сток. 7.Рыжик. 8.Исток. 9.Битва. 10.Декан. 11.Лассо. 12.Мешок. 13.Онагр. 14.Ерика. 15.Игрун. 16.Драга. 17.Обрат. 18.Тяпка. 19.Артрит. 20.Ячёнок. 21.Колосс. 22.Спектр. 23.Стайер. 24."Беглец". 25.Окурок. 26.Надлом. 27.Статуя. 28.Трахома. 29.Пикадор. 30.Алгебра. 31.Граната. 32.Невыход. 33.Неудача. 34.Собачка. 35.Арапник. 36.Наездка. 37.Прозаик. 38.Душечка. 39. Карнавал. 40.Абориген. 41.Чулочник. 42.Кавардак. 43.Луковица. 44.Середняк. 45.Мститель. 46.Сталактит. 47.Африканка. 48.Иммунитет. 49.Вьетнамка. 50.Временник. 51.Сердцеедка.

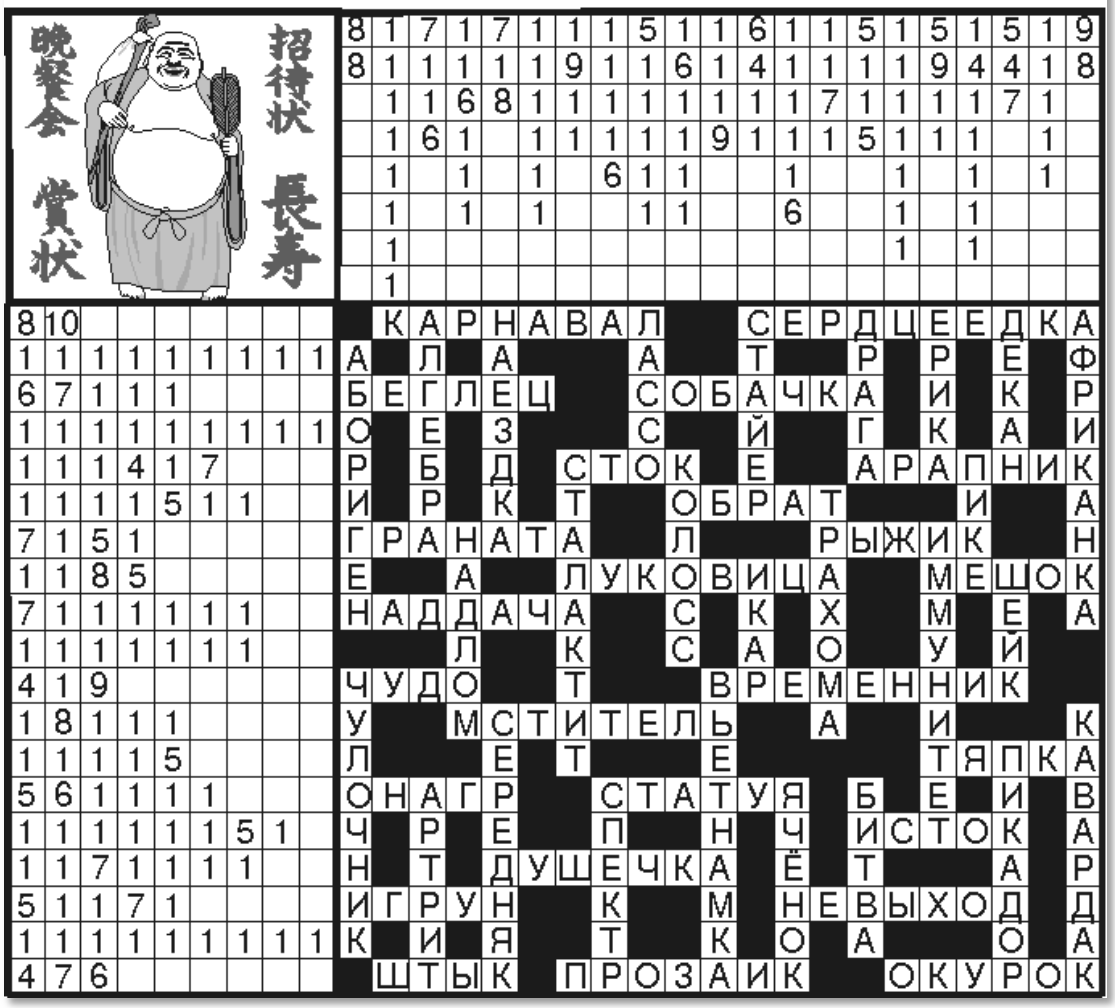


Рис. Ф4.40. А что, неплохо получилось!



Наша программа напичкана всем, что нужно для плодотворной работы. Но вы можете на досуге подумать вот над чем. Почти во всех процедурах и функциях программы одни и те же действия проводились отдельно для верхнего и левого числовых полей. Сделано это исключительно для простоты понимания, но теперь это вам уже ни к чему и вы спокойно можете переделать программу так, чтобы не писать дважды один и тот же код. Пример этого вы найдёте в самой программе. Например, функция *TestingOneLine* обрабатывает и вертикальные, и горизонтальные ряды. О других усовершенствованиях программы уже говорилось раньше. Кроме того, вы можете проявить разумную инициативу и даже отчаянную смелость на этом поприще.

Факультатив 5. Игра-головоломка *Logos*

Logos - это прекрасная, оригинальная компьютерная головоломка! Она требует от игрока минимальных знаний, поэтому в неё могут играть даже маленькие дети. И в то же время игра *Logos* прекрасно развивает логическое и комбинаторное мышление.

Правила игры

очень просты и немногочисленны:

- На клетки прямоугольного поля поочерёдно выставляются фишки, которые можно представить в виде игральных кубиков. На верхней, видимой грани «свежего» кубика находится *одно* очко. На передней – *два*, на нижней – *три* и, наконец, на задней – *четыре*. Боковые грани в игре не участвуют.

- Если вновь выставленная фишка соприкасается своими гранями с уже имеющимися на поле, то последние поворачиваются так, чтобы на их верхней грани оказалось на единицу большее число точек (очков). После четырёх очков, естественно, на верхнюю грань снова выйдет одно очко.

Например, на доске (игровом поле) стоит фишка (Рис. Ф5.1).

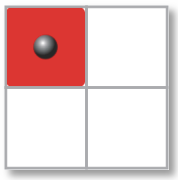


Рис. Ф5.1. Занимаем исходное положение!

- Если следующие фишки выставляются так, что они не имеют *общих* граней с фишками на поле, то эти фишки не изменяют своих значений (Рис. Ф5.2).

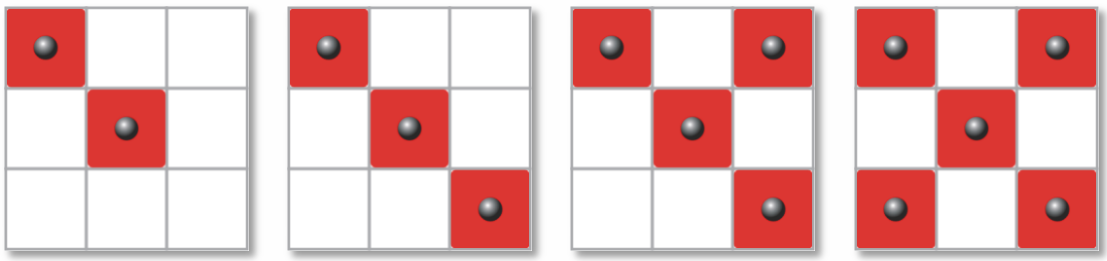


Рис. Ф5.2. Не тревожьте соседей!

- Если же новая фишка *соприкасается* с ранее выставленными фишками, то они поворачиваются на следующую грань (Рис. Ф5.3).

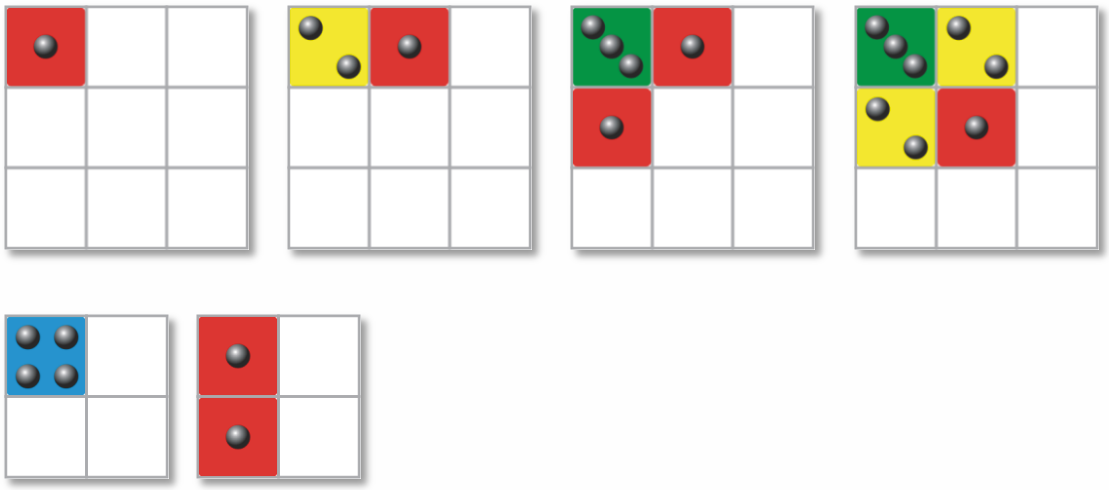


Рис. Ф5.3. Поворачиваемся с боку на бок!

Цель игры: выставляя на поле фишки с соблюдением правил, построить заданную фигуру.

Далее мы подробно разберём алгоритм игры, позволяющий достаточно легко решить *любую* задачу, а пока попробуйте самостоятельно справиться с несколькими самыми простыми заданиями.

Составьте следующие фигуры, последовательно выставляя на первоначально пустое поле фишки.

Задача 1. (Рис. Ф5.4).

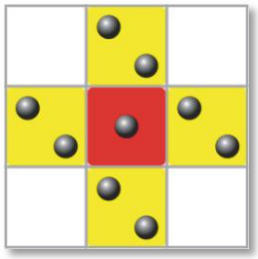


Рис. Ф5.4. Вот и первая задача!

Задача 2. (Рис. Ф5.5).

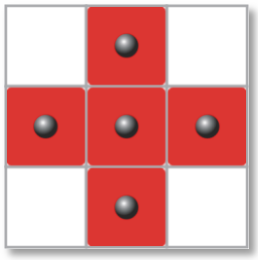


Рис. Ф5.5. Задача не приходит одна!

Задача 3. (Рис. Ф5.6).

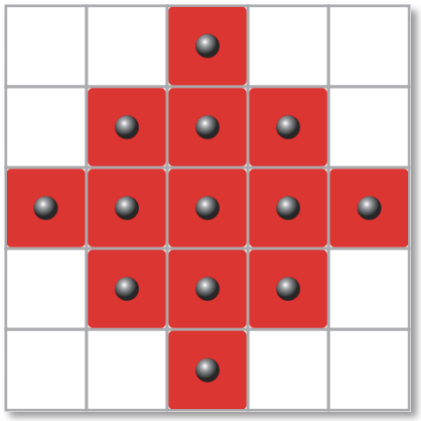
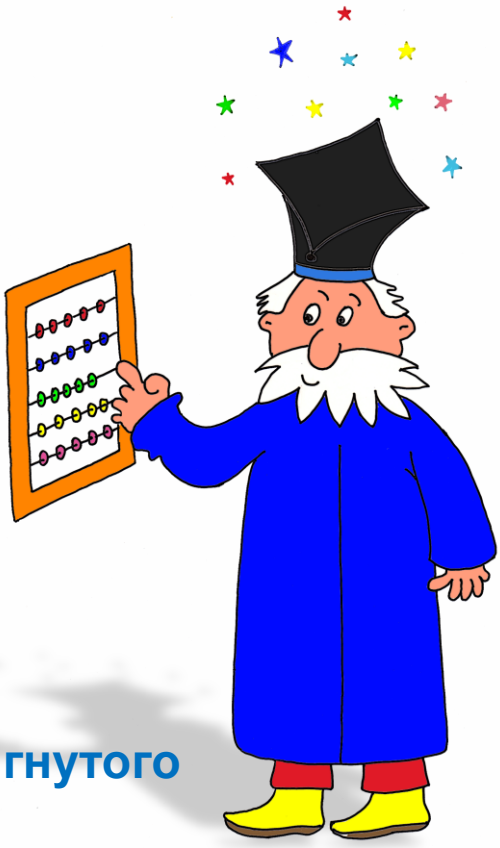


Рис. Ф5.6. Задачи ходят по-трое!



Программируем от достигнутого

Так как правила игры *Logos* не очень сильно отличаются от правил игры *ShortGame*, то мы возьмем за основу интерфейс этой игры, только исправим и дополним его новыми элементами. Поэтому начните работу над новым проектом с того, что загрузите в *Delphi*, а затем сохраните в новой папке все файлы игры *ShortGame*.

Правила игры *Logos* никак не ограничивают размеры игрового поля, но опыт показывает, что 13 клеток по горизонтали и столько же по вертикали вполне достаточно для любых разумных заданий. Это немного больше, чем в игре *ShortGame*, поэтому и форму, и поля придётся увеличить, а размеры ячеек сеток - уменьшить. Кроме того, форму необходимо «адаптировать» к новой игре – заменить *название* и *значок* в заголовке:

```
BorderIcons = [biSystemMenu, biMinimize]
```

```
  BorderStyle = bsSingle
  Caption = '  Logos'
  ClientHeight = 515
  ClientWidth = 726
  Color = clSkyBlue
  PopupMenu = PopupMenu1
  Position = poDesktopCenter

  OnCreate = FormCreate
  OnMouseMove = FormMouseMove
  OnShow = FormShow
```

Совсем немного придётся поработать и над обеими *сетками*.

Сетка dgPole:

```
    Left = 4
    Top = 44
    Width = 465
    Height = 465
    Cursor = crHandPoint
    ColCount = 14
    RowCount = 14 (не забывайте об оцифровке!)
    DefaultColWidth = 32
    DefaultRowHeight = 32
    DefaultDrawing = False
    FixedCols = 0
    FixedRows = 0
    Options = [goVertLine, goHorzLine]
    ScrollBars = ssNone

    OnDrawCell = dgPoleDrawCell
    OnMouseDown = dgPoleMouseDown
    OnMouseMove = FormMouseMove
```

Сетка dgPole2:

```
    Left = 484
    Top = 44
    Width = 241
    Height = 241
    Cursor = crHandPoint
    ColCount = 14
    RowCount = 14
    DefaultColWidth = 16
    DefaultRowHeight = 16 (размер клеток на правом поле устанавли-
ваем в два раза меньше, чем на игровом)
  DefaultDrawing = False
    Options = [goVertLine, goHorzLine]
    ScrollBars = ssNone

    OnDrawCell = dgPole2DrawCell
```

```
    OnMouseDown = dgPole2MouseDown
    OnMouseMove = FormMouseMove
```

Соответствующим образом изменяем и все глобальные *константы*:

```
const
  //макс. размеры поля:
  MAX_POLE_WIDTH= 14;
  MAX_POLE_HEIGHT= 14;
  NAME_PROG='Logos';
  letter: string= 'ABCDEFGHIJKLM';
```

Практически без изменений останутся объявления *типов*:

```
type
  TPole=   array[0..MAX_POLE_WIDTH-1,   0..MAX_POLE_HEIGHT-1]   of
Shortint;
  TMemory= Record
    Hod: integer;
    //позиция на полях:
    PosL, PosR: TPole;
  end;
  //состояние программы:
  TGameState= (gsWait, gsSolution);
  //ходы:
  TMoves= Record
    //колонка:
    Col: integer;
    //строка:
    Row: integer;
    //позиция на полях:
    PosL, PosR: TPole;
  end;
```

Для большинства глобальных *переменных* игры *ShortGame* найдётся работа и в нашей программы, а остальные мы будем добавлять по мере необходимости:

```
var
  //поля:
  masPole, masPole2: TPole;
  Memory: TMemory;
  GameState: TGameState= gsWait;
  //запись ходов:
  Moves: array[0..169] of TMoves;
  //номер хода:
  Hod: integer= 0;
  Hod_max: integer= 0;
  step: integer;
  rep: integer= 0;
```

Поскольку размеры формы и полей в этой программе *постоянны*, то подготовка к новой игре намного облегчается. Достаточно обнулить ходы и очистить оба поля:

```
//СОЗДАТЬ ФОРМУ
procedure TForm1.FormCreate(Sender: TObject);
begin
  //подготовить новую игру:
  NewPlay;
end; //FormCreate

//ПОДГОТОВКА К НОВОЙ ИГРЕ
procedure TForm1.NewPlay;
var
 i, j: integer;
begin
  //обнулить число ходов:
  Hod:= 0;   Hod_max:= 0;
  lblHod.Caption:=  'Ход - 0';

  //задача не имеет названия:
  form1.caption:= NAME_PROG + '   []';

  //очистить левое поле:
  for j:= 1 to dgPole.RowCount-1 do
    for i:= 1 to dgPole.ColCount-1 do
      masPole[i,j]:= 0;
  dgPole.Invalidate;

  //очистить правое поле:
  masPole2:= masPole;
  dgPole2.Invalidate;
  //запомнить начальную позицию:
  MemoryIn;
end; // NewPlay
```

В массивах *masPole* и *masPole2 нулём* мы будем обозначать *пустую* клетку поля, а числами *1, 2, 3, 4* – количество очков на верхней грани фишки. Всё настолько понятно, что даже не нужно вводить констант для содержимого клеток полей.

Изменившиеся размеры ячеек и игрового поля заставляют нас нарисовать новые *картинки* для оцифровки поля (первая картинка в программе не используется) (Рис. Ф5.7)



Рис. Ф5.7. Оцифровка поля

Как и раньше, все эти картинки нужно загрузить в компонент *imlNota* типа *TImageList*, предварительно установив их размер:

```
Height = 32
Width = 32
```

Рисунки фишек и клеток поля (Рис. Ф5.8) мы разместим в компонентах *TImage* на форме (если вам больше нравятся списки *TImageList*, то вы легко можете подправить код программы).



Рис. Ф5.8. Фишки-малышки

Для удобства решения (и «красивости») грани фишек окрашены в *разные* цвета, а клетки поля имеют разную интенсивность окраски. Те клетки игрового поля, которые соответствуют клеткам *правого* поля с фишками, мы будем окрашивать *светлее*, чтобы выделить «игровые» клетки, в которые следует ходить.

Верхний левый угол поля мы по-прежнему используем для анимации, только сделаем её более «изощрённой» - там будет пульсировать кубик с четырьмя точками. Три картинки (Рис. Ф5.9) для анимации мы загрузим в компонент *TImageList imlAnima* (не забудьте установить размер картинок 32 на 32 пикселя!).



Рис. Ф5.9. Четырёхточечный «пульсар»

Номер текущей (видимой) картинки хранится в глобальной *переменной*:

```
flgFlash: integer= 0;
```

Она последовательно принимает значение 0, 1, 2, 0, 1, 2, … , в соответствии с которым и выводится в ячейку картинка из списка. Частота пульсации определяется свойством *Interval* таймера. Неплохие результаты получаются, если установить *Interval = 100*.

```
//ПУЛЬСИРОВАНИЕ УГЛОВОГО КВАДРАТИКА
procedure TForm1.Timer2Timer(Sender: TObject);
var
  rect: TRect;
  n: integer;
begin
```

```
  //угловой квадратик поля:
  rect:= dgPole.CellRect(0,0);
  //номер картинки:
  case flgFlash of
    0,1: inc(flgFlash);
    2: inc(flgFlash);
    3: flgFlash:= 0;
  end;
  if flgFlash= 3 then n:= 0 else n:= flgFlash;
  imlAnima.Draw(dgPole.Canvas, rect.left, rect.top, n);
end;  //Timer2Timer
```

Других премудростей в процедуре рисования ячейки игрового поля нет:

```
//ОТРИСОВАТЬ ЯЧЕЙКУ ИГРОВОГО ПОЛЯ
procedure TForm1.dgPoleDrawCell(Sender: TObject; ACol, ARow: Inte-
ger;
  Rect: TRect; State: TGridDrawState);
var
  n, m: integer;
  r, dr: TRECT;
begin
  //какая фишка в ячейке:
  n:= masPole[ACol, ARow];
  //размеры картинок:
  r:= Bounds(0, 0, 32, 32);
  dr:= Bounds(Rect.Left, Rect.Top, 32, 32);
  m:= 100 * ARow + ACol;
  case m of
    0..13: //- верхняя строка
      imlNota.Draw(dgPole.Canvas, dr.left, dr.top, m);
    100,    200,    300,    400,    500,    600,    700,    800,
900,1000,1100,1200,1300: //- первая колонка
      imlNota.Draw(dgPole.Canvas, dr.left, dr.top, m div 100+13);
    else begin
      case n of
        0:    //- пустая клетка
          if masPole2[ACol, ARow]<> 0 then
            dgPole.canvas.CopyRect(dR, img00.Canvas, R)
          else
            dgPole.canvas.CopyRect(dR, img0.Canvas, R);
        1:   //- фишка 1
          dgPole.canvas.CopyRect(dR, img1.Canvas, R);
        2:   //- фишка 2
          dgPole.canvas.CopyRect(dR, img2.Canvas, R);
        3:   //- фишка 3
          dgPole.canvas.CopyRect(dR, img3.Canvas, R);
        4:   //- фишка 4
          dgPole.canvas.CopyRect(dR, img4.Canvas, R);
      end
    end
  end;
```

```
end;  //dgPoleDrawCell
```

Из-за различия в размерах ячеек *правого* поля и рисунков с оцифровкой и фишками нам придётся картинки «ужимать». С картинками-фишками всё достаточно просто, так как мы можем применить к ним для вывода в ячейки удобный метод канвы *StretchDraw* (поэтому, для сравнения, эти картинки и размещены в компонентах *TImage*, а не в *TImageList*). Но компонент *TImageList* такого метода не имеет и выводит картинки только того размера, который задан его свойствами *Height* и *Width* (то есть 32 на 32 пикселя), а размеры ячеек *правого* поля в два раза меньше (16 на 16 пикселей) – тут нам никак не избежать дополнительного растра *bmp* в памяти компьютера. Сначала мы скопируем картинку из списка его методом *Draw* в растр, а уже оттуда – в нужную ячейку сетки с масштабированием, используя метод канвы *StretchDraw*. Слегка замысловато, но зато не нужно заготавливать два комплекта рисунков разного размера.

```
//ОТРИСОВАТЬ ЯЧЕЙКУ ПОЛЯ-ОБРАЗЦА
procedure TForm1.dgPole2DrawCell(Sender: TObject; ACol, ARow: In-
teger;
  Rect: TRect; State: TGridDrawState);
var
  n, m: integer;
  r, dr: TRECT;
  bmp: TBitmap;
begin
  //фишка в ячейке:
  n:= masPole2[ACol, ARow];
  //размеры картинок:
  r:= Bounds(0, 0, 16, 16);
  dr:=        Bounds(Rect.Left,        Rect.Top,        Rect.Right-
Rect.Left,Rect.Bottom-Rect.Top);

  m:= 100 * ARow + ACol;

  bmp:= TBitmap.Create;
  bmp.Width:= 32;
  bmp.Height:= 32;

  case m of
    0..13: //- верхняя строка
      begin
        imlNota.Draw(bmp.Canvas, 0, 0, m);
        dgPole2.canvas.StretchDraw(dR, bmp);
      end;
    100,    200,    300,    400,    500,    600,    700,    800,
900,1000,1100,1200,1300: //- первая колонка
      begin
        imlNota.Draw(bmp.Canvas, 0, 0, m div 100 + 9);
```

```
        dgPole2.canvas.StretchDraw(dR, bmp);
      end;
    else begin
      case n of
        0:    //- пустая клетка
          dgPole2.canvas.StretchDraw(dR, img0.Picture.Graphic);
        1:   //- фишка 1
          dgPole2.canvas.StretchDraw(dR, img1.Picture.Graphic);
        2:   //- фишка 2
          dgPole2.canvas.StretchDraw(dR, img2.Picture.Graphic);
        3:   //- фишка 3
          dgPole2.canvas.StretchDraw(dR, img3.Picture.Graphic);
        4:   //- фишка 4
          dgPole2.canvas.StretchDraw(dR, img4.Picture.Graphic);
      end
    end
  end;
  bmp.Free;
  dgPole.Invalidate;
end;  //dgPole2DrawCell
```

Запустив программу, мы увидим *пустые* поля, ведь никто не побеспокоился загрузить задание. Что ж, позаботимся о себе сами. Проще всего расставлять фишки, нажимая *левую* кнопку мыши на нужной клетке поля. Сколько раз кликнул, такая фишка и появилась (место фишки с четырьмя очками займёт пустая клетка, затем с одним очком и так далее). На *левом*, игровом поле будет синхронно появляться бледный образ будущей задачи.

```
//ПОСТАВИТЬ ФИШКУ НА ПОЛЕ-ОБРАЗЦЕ
procedure TForm1.dgPole2MouseDown(Sender: TObject; Button: TMouse-
Button;
  Shift: TShiftState; X, Y: Integer);
var
  ACol,ARow: integer;
begin
  //координаты мыши:
  dgPole2.MouseToCell(x,y,ACol,ARow);
  //нажата левая кнопка мыши - ставим след. фишку:
  if (ssLeft in shift) then begin
    inc(masPole2[ACol,ARow]);
    if masPole2[ACol,ARow]>4 then masPole2[ACol,ARow]:= 0;
    dgPole2.Invalidate;
  end
end;  //dgPole2MouseDown
```

В качестве примера вы можете ввести задачи для самостоятельного решения.

И вот задача ждёт своего решателя. А он (то есть мы) уже тут как тут – нажимаем левую кнопку мыши в нужной клетке - и фишка с одним глазком сейчас же появляется в ней. Со вторым ходом сложнее – нужно изменить все соседние фишки. Эту операцию мы поручим *функции DoMove*, которая вернёт нам поле со всеми изменениями:

```
 //ВЫПОЛНИТЬ ХОД
function TForm1.DoMove(arr: TPole; x, y: integer): TPole;
var i: integer;
begin
  result:= arr;
  //нельзя ходить в занятую клетку:
  if arr[x,y]<> 0 then begin dec(hod);exit; end;
  //изменить соседние фишки:
  for i:= Max(y-1,1) to Min(y+1,dgPole.RowCount-1) do //- верти-
кальный ряд
  begin
    if arr[x,i] <> 0 then inc(arr[x,i]);
    if arr[x,i]>4 then arr[x,i]:= 1;
  end;
  for i:= Max(x-1,1) to Min(x+1,dgPole.ColCount-1) do //- горизон-
тальный ряд
  begin
    if arr[i,y] <> 0 then inc(arr[i,y]);
    if arr[i,y]>4 then arr[i,y]:= 1;
  end;
  //поставить в заданную клетку фишку 1:
  arr[x,y]:= 1;
  result:= arr;
end; //DoMove
```

Эта функция понадобится нам ещё и в другом модуле, поэтому её объявление следует перенести в раздел *public* типа формы:

```
  public
    { Public declarations }
    function DoMove(arr: TPole; x, y: integer): TPole;
    function IsReady: Boolean;
    procedure Ready;
```

Очень удобно иметь перед глазами запись всех сделанных ходов, как это принято в шахматах. У нас уже имеется форма *frmProtokol* для ведения протокола, так что нам осталось усовершенствовать её так, чтобы она всегда была на экране.

```
  Left = 600
  Top = 332
  BorderIcons = []
  BorderStyle = bsSingle
  Caption = '  Протокол'
```

```
  ClientHeight = 215
  ClientWidth = 191
  Font.Name = 'MS Sans Serif'
  FormStyle = fsStayOnTop  (располагаем форму поверх других окон приложения)
```

Модуль формы протокола нужно объявить в модуле *LogosUnit*, чтобы сделать его доступным в главном модуле.

```
implementation

uses ProtokolUnit, ReadyUnit;
```

Чтобы форма с протоколом не перекрывала элементы управления на главной форме, выводим её в правом нижнем углу:

```
//ПОКАЗАТЬ ФОРМУ
procedure TForm1.FormShow(Sender: TObject);
begin
  open('001.lgs');
  frmProtokol.show;
end; //FormShow

procedure TForm1.FormActivate(Sender: TObject);
begin
  frmProtokol.Left:= form1.Left + lblHod.Left;
  frmProtokol.Top:= form1.Top + lblHod.Top+lblHod.Height+35;
end; //FormActivate
```

Обратите внимание, что при каждом ходе записывается его номер и координаты клетки – в том виде, как это делается в настольных играх (а иначе зачем мы делали оцифровку полей?). И напоследок нужно проверить, а не решена ли задача. Для этого после каждого хода мы вызываем *функцию IsReady*, которая сравнивает оба поля и, если они полностью совпадают, возвращает значение *True*:

```
//ПРОВЕРИТЬ, НЕ РЕШЕНА ЛИ ЗАДАЧА
function TForm1.IsReady: Boolean;
var
  i, j: integer;
begin
  Result:= True;
  for j:= 1 to dgPole.RowCount-1 do
    for i:= 1 to dgPole.ColCount-1 do
      if masPole[i,j]<> masPole2[i,j] then
        begin Result:= False; exit end
end; //IsReady
```

Если вам повезёт, то программа поздравит вас с успехом надлежащей надписью. Хотелось бы миг победы сделать ещё более приятным, поэтому давайте *художественно* изобразим поздравление на отдельной форме *frmReady* (Рис. Ф5.10).

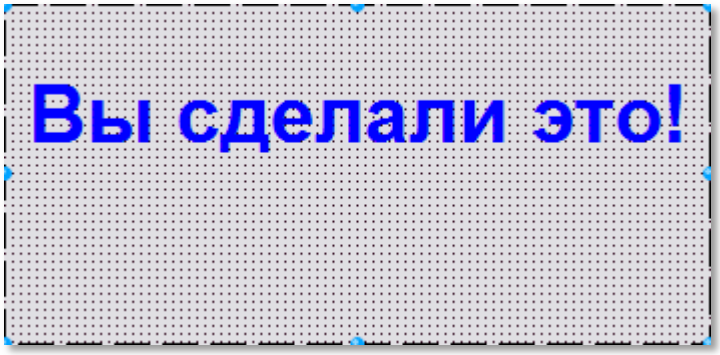


Рис. Ф5.10. Ай, молодца!

```
AutoSize = True
BorderStyle = bsNone
Position = poDesktopCenter
```

Всю клиентскую часть занимает компонент *Image1: TImage*:

```
Left = 0
Top = 0
Width = 353
Height = 170
```

На нём установите *метку Label1*:

```
Left = 12
Top = 32
Width = 329
Height = 46
Caption = 'Вы сделали это!'
Font.Charset = RUSSIAN_CHARSET
Font.Color = clBlue
Font.Height = -40
Font.Name = 'Arial'
Font.Style = [fsBold]
Transparent = True
```

В *процедуре Ready* метка возвращается в исходное положение (скоро вы узнаете почему), форма с надписью показывается на экране и запускается таймер *Timer1*:

```
//ЗАДАЧА РЕШЕНА!
procedure TForm1.Ready;
begin
```

```
  rep:= 0;
  ReadyUnit.frmReady.label1.Top:= 32;
  frmReady.Show;
  timer1.Enabled:= true;
end; //Ready
```

Таймер (**Enabled** = *False,* **Interval** = *100*) периодически вызывает *процедуру OutReady,* где фон заполняется случайными точками серого и белого цветов:

```
//СОЗДАТЬ ФОН ДЛЯ НАДПИСИ
procedure OutReady;
var
  bmp: TBitmap;
  i, j: integer;
begin
  //создать временный растр:
  bmp:=TBitmap.Create;
  bmp.Height:= ReadyUnit.frmReady.Image1.Height;
  bmp.Width:= ReadyUnit.frmReady.Image1.Width;
  //заполнить его точками серого и белого цвета:
  for j:= 0 to bmp.Height do
    for i:= 0 to bmp.Width do
      if random(2) = 0 then bmp.Canvas.Pixels[i,j]:= clGray
      else bmp.Canvas.Pixels[i,j]:= clWhite;
  //обвести рамкой:
  bmp.Canvas.Brush.Style:= bsClear;
  bmp.Canvas.Pen.Width:= 5;
  bmp.Canvas.Pen.Color:= clYellow;
  bmp.Canvas.Rectangle(0,0,bmp.Width,bmp.Height);
  //скопировать в объект Image1:
  ReadyUnit.frmReady.Image1.Picture.Bitmap.Assign(bmp);
  //уничтожить растр:
  bmp.FreeImage;
end;  //OutReady;
```

На этот фон накладывается надпись с поздравлением, которая медленно перемещается вниз и время от времени меняет цвет (Рис. Ф5.11). После исчерпания заданного числа срабатывания таймер выключается, а форма убирается с экрана:

```
//ПОЗДРАВИТЬ
procedure TForm1.Timer1Timer(Sender: TObject);
begin
  OutReady;
  inc(rep);
  if rep mod 10 > 5  then ReadyUnit.frmReady.label1.Font.Color:=
clBlue
  else ReadyUnit.frmReady.label1.Font.Color:= clRed;
  ReadyUnit.frmReady.label1.Top:= ReadyUnit.frmReady.label1.Top+1;
```

```
  if rep> 60 then begin
    timer1.Enabled:= false;
    frmReady.Hide;
  end;
end;  //Timer1Timer
```



Рис. Ф5.11. Художественная вещь!

Если вы захотите выставить позицию на игровом поле, то нажимайте *левую* кнопку мыши, удерживая клавишу *Shift*:

```
//ВЫСТАВИТЬ ФИШКУ ИЛИ СДЕЛАТЬ ХОД
procedure TForm1.dgPoleMouseDown(Sender: TObject; Button: TMouse-
Button;
  Shift: TShiftState; X, Y: Integer);
var
  ACol,ARow: integer;
begin
  //координаты мыши:
  dgPole.MouseToCell(x,y,ACol,ARow);
  if ACol * ARow = 0 then exit;
  //нажата левая кнопка мыши и клавиша Shift -
  //ставим след. фишку:
  if (ssLeft in shift) and (ssShift in shift ) then begin
    inc(masPole[ACol,ARow]);
    if masPole[ACol,ARow]>4 then masPole[ACol,ARow]:= 0;
    dgPole.Invalidate;
  end
  else if (ssLeft in shift) and (ssAlt in shift )then begin
    //не нужно ходить в пустую клетку:
    if masPole[ACol, ARow]= 0 then exit;
    //ход назад:
    masPole:=BackMove(masPole, ACol, ARow);
    inc(Hod); lblHod.Caption:= 'Ход - ' + inttostr(Hod);
    dgPole.Invalidate;
    //запомнить ход:
    Moves[Hod].Col:= ACol;
    Moves[Hod].Row:= ARow;
    Moves[Hod].PosL:= masPole;
    Hod_max:= Hod;
```

```
    //занести ход в протокол:
    frmProtokol.ListBox1.items.add(format('%3d',[Hod]) + '. ' +
                          letter[ACol]+ ' ' +inttostr(ARow));
  end
  //нажата левая кнопка мыши без клавиши Shift и Alt -
  //делаем ход:
  else if ssLeft in shift then begin
    if masPole[ACol, ARow]<> 0 then exit;
    masPole:=DoMove(masPole, ACol, ARow);
    inc(Hod); lblHod.Caption:= 'Ход - ' + inttostr(Hod);
    dgPole.Invalidate;
    //запомнить ход:
    Moves[Hod].Col:= ACol;
    Moves[Hod].Row:= ARow;
    Moves[Hod].PosL:= masPole;
    Hod_max:= Hod;
    //занести ход в протокол:
    frmProtokol.ListBox1.items.add(format('%3d',[Hod]) + '. ' +
                          letter[ACol]+ ' ' +inttostr(ARow));
    //проверить, не решена ли задача:
    if IsReady then Ready;
  end
end;  //dgPoleMouseDown
```

Дабы обеспечить себе возможность *возврата* к прежним позициям, мы запоминаем очередной ход и позицию на обоих полях в массиве *Moves*.

А о том, для чего нужно удерживать клавишу *Alt*, читайте дальше.

Всплывающее меню

Всплывающее меню в этой программе существенно уменьшилось в размерах по сравнению с игрой *ShortGame* (Рис. Ф5.12).

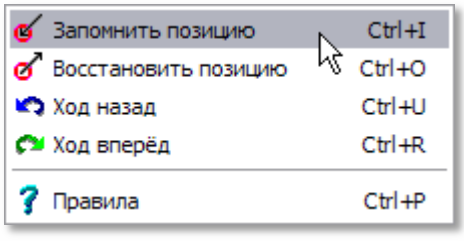


Рис. Ф5.12. Всплывающее меню

Вы даже можете совсем удалить его из проекта, заменив все оставшиеся пункты меню *кнопками* на форме, благо места для них теперь более чем достаточно.

Выбрав *первый* пункт меню, вы запомните игровую ситуацию

```
//ЗАПОМНИТЬ ПОЗИЦИЮ
procedure TForm1.MemoryIn;
begin
  Memory.hod:= Hod;
  Memory.PosL:= masPole;
  Memory.PosR:= masPole2;
end; //MemoryIn
procedure TForm1.miMemoryInClick(Sender: TObject);
begin
  MemoryIn;
end; //miMemoryInClick
```

А затем в любое время сможете вернуться к ней, щёлкнув на *втором* пункте:

```
//ВОССТАНОВИТЬ ПОЗИЦИЮ НА ПОЛЕ
procedure TForm1.miMemoryOutClick(Sender: TObject);
var i: integer;
begin
  with Memory do begin
    masPole:= PosL;
    masPole2:= PosR;
  end;
  dgPole.Invalidate;
  dgPole2.Invalidate;
  Hod:= Memory.hod;
  lblHod.Caption:= 'Ход - ' + inttostr(hod);
  //обновить протокол:
  frmProtokol.ListBox1.items.Clear;
  for i:= 1 to hod do
    frmProtokol.ListBox1.items.add(format('%3d',[i]) + '. ' +
                          letter[Moves[i].Col]+ ' '
+inttostr(Moves[i].Row));
end; //miMemoryOutClick
```

Следующий пункт меню вернёт вас к предыдущему ходу. А чтобы не возникало временны***х*** коллизий, откат ограничен запомненной позицией:

```
//ВЕРНУТЬ ХОД
procedure TForm1.miUndoClick(Sender: TObject);
var i: integer;
begin
  if hod= Memory.Hod then exit;
  dec(hod);
  lblHod.Caption:=  'Ход - ' + inttostr(hod);
  masPole:= Moves[hod].PosL;
  dgPole.Invalidate;
  //обновить протокол:
  frmProtokol.ListBox1.items.Clear;
  for i:= 1 to hod do
```

```
    frmProtokol.ListBox1.items.add(format('%3d',[i]) + '. ' +
                          letter[Moves[i].Col]+          '          '
+inttostr(Moves[i].Row));
end; //miUndoClick
```

С той же лёгкостью вы сможете повторить отвергнутый ход:

```
//ОТМЕНИТЬ ПОСЛЕДНИЙ ВОЗВРАТ ХОДА
procedure TForm1.miRedoClick(Sender: TObject);
var i: integer;
begin
  if Hod< Hod_max then inc(hod)
  else exit;
  //восстановить позицию на игровом поле:
  masPole:=Moves[Hod].PosL ;
  dgPole.Invalidate;
  lblHod.Caption:= 'Ход - ' + inttostr(hod);
  //обновить протокол:
  frmProtokol.ListBox1.items.Clear;
  for i:= 1 to hod do
    frmProtokol.ListBox1.items.add(format('%3d',[i]) + '. ' +
                          letter[Moves[i].Col]+ ' '
+inttostr(Moves[i].Row));
end;  //miRedoClick
```

Последний пункт меню, показывающий справочный файл, мы рассмотрим после того, как закончим конструирование интерфейса программы.

Весьма удобной может оказаться возможность *обнулить* ходы. Для этого достаточно просто щёлкнуть на метке, показывающей число сделанных ходов:

```
//ОБНУЛИТЬ ХОДЫ
procedure TForm1.lblHodClick(Sender: TObject);
begin
  Hod:= 0;   Hod_max:= 0;
  lblHod.Caption:=  'Ход - 0';
  frmProtokol.listbox1.items.add('');
end;  //lblHodClick
```

«Отзывчивые» кнопки

Вы, должно быть, много раз видели в различных программах кнопки, которые изменяют свой вид, когда курсор мыши заходит на них, и возвращаются в прежнее состояние, когда он их покидает. Ничего не скажешь, эффектное зрелище! А между тем такие визуальные превращения нетрудно встроить и в свою собственную программу. Можно даже использовать кнопки из программы *ShortGame*, если их чуточку

доработать. Так мы и сделаем, а также добавим ещё несколько кнопок, необходимых для работы данной программы.

Все заранее подготовленные картинки в формате *BMP* загрузите в компоненты *TImage* на форме (Рис. Ф5.13).

 - Image9  - Image10

 - Image18  - Image19

 - Image3  - Image4

 - Image1  - Image2

 - Image7  - Image8

 - Image13  - Image14

 - Image21  - Image22

 - Image11  - Image12

 - Image5  - Image6

 - Image20  - Image17

Рис. Ф5.13. Кнопочки программные!

Обратите внимание на то, что *розовые* кнопки немного больше по размеру, чем *синие* (величина всех картинок одинакова – 36 х 36 пикселей, чтобы центры кнопок в обоих положениях совпадали), поэтому кнопка под мышкой будет не только изменять цвет, но и «раздуваться».

Вот так выглядит форма с компонентами в окне *Конструктора формы* (Рис. Ф5.14).

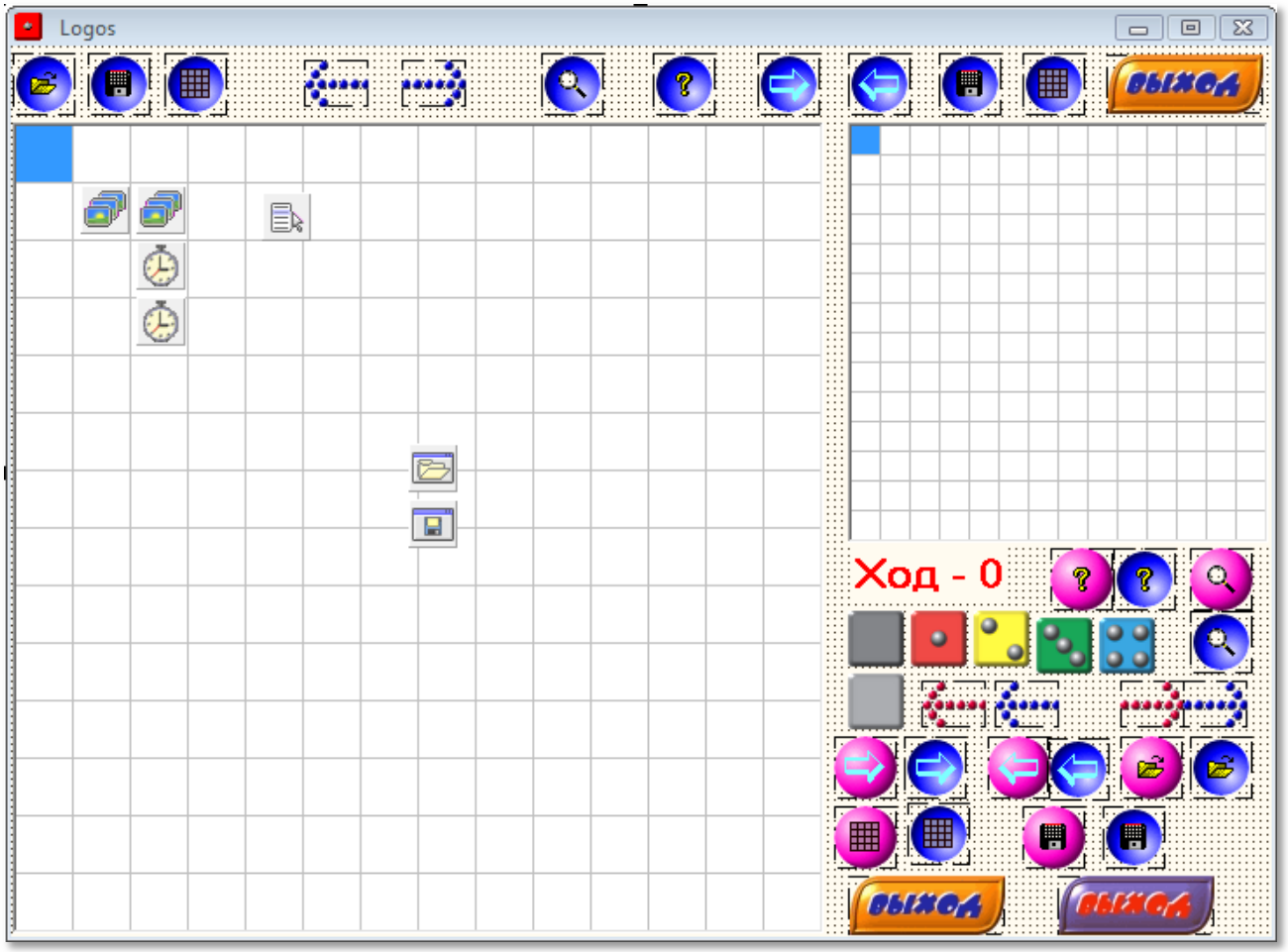


Рис. Ф5.14. Кнопочный боекомплект!

Как и в игре *ShortGame*, для каждой кнопки нужно установить на форме ещё один компонент *TImage* и загрузить в него ту картинку, которая соответствует неактивной (отжатой) кнопке. Большинство свойств у этих компонентов одинаковы (у «некруглых кнопок» другие размеры и верхняя координата):

```
Width = 36
Height = 36
Top = 4
AutoSize = True
Cursor = crHandPoint
ShowHint = True
Transparent = True
```

Различаются у них свойства *Hint* (подсказка), *Left, Top* (положение на форме) и *Tag* (по нему мы и будем определять, над какой кнопкой находится курсор мыши).

Рассмотрим действие самой простой кнопки – *Выход*. Она сделана в ином стиле, чем все остальные кнопки, чтобы по ошибке не «выпасть» из программы. Её изображение находится в компоненте *imgExit* (Рис. Ф5.15).



Рис. Ф5.15. Выход есть всегда!

```
Left = 628
Top = 5
Height = 34
Width = 89
Hint = 'Выход из программы'
Tag = 111

OnMouseDown = imgExitMouseDown
OnMouseMove = FormMouseMove
OnMouseUp = imgExitMouseUp
```

При нажатии на эту кнопку возникает событие, которое обрабатывается *процедурой imgExitMouseDown*. В ней просто одна картинка *заменяется* другой:

```
//ЗАКРЫТЬ ПРОГРАММУ
procedure TForm1.imgExitMouseDown(Sender: TObject; Button: TMouse-
Button;
  Shift: TShiftState; X, Y: Integer);
begin
  imgExit.Picture.Assign(Image20.Picture);
end; //imgExitMouseDown
```

При отпускании кнопки возвращается прежняя картинка и программа закрывается:

```
procedure TForm1.imgExitMouseUp(Sender: TObject; Button: TMouse-
Button;
  Shift: TShiftState; X, Y: Integer);
begin
  imgExit.Picture.Assign(Image17.Picture);
  close
end; //imgExitMouseUp
```

Но самые интересные превращения кнопки происходят тогда, когда на неё «наезжает» курсор. Вообще-то мы не можем определить момент захода курсора на кнопку и схода с неё, так как у компонента *TImage* нет соответствующего события (это верно для «старых» версий *Delphi*, в *XE3* эти события имеются, но для совместимости мы поступим не самым

лучшим образом!). Поэтому придётся воспользоваться другим свойством - *MouseMove*, которое возникает при движении мыши по компоненту.

Так как кнопок у нас довольно много, то нужно добавить в проект более десятка похожих процедур. Хуже того – подобные процедуры потребуются и для других компонентов на форме и самой формы, иначе нам не удастся зафиксировать момент схода курсора с кнопки – ведь при этом не возникает никакого события, просто курсор начнёт перемещаться по другому компоненту. Правильнее будет всю обработку перемещений курсора перенести в одну процедуру, например *FormMouseMove*, в которой и определять, где находится курсор.

По умолчанию значение свойства *Tag* у всех компонентов (и у формы в том числе) равно *нулю*, поэтому, присвоив кнопкам другие, индивидуальные значения, мы сможем установить, какая кнопка (*Sender*) вызвала этот обработчик.

Но, в первую очередь, необходимо обеспечить прямое попадание в процедуру *FormMouseMove* при движении курсора на других компонентах. Поэтому для каждого компонента (кнопок и всех близлежащих) в *Инспекторе объектов*, на вкладке *Events* следует присвоить свойству *OnMouseMove* значение *FormMouseMove* (достаточно выбрать его из выпадающего списка). Вот теперь при заходе курсора на кнопки и сходе с них будет вызываться процедура *FormMouseMove*. По значению свойства *Tag* мы определяем, какой компонент находится под курсором. Но тут есть одна тонкость. Если курсор только что зашёл на кнопку, то картинка на ней изменилась и при дальнейшем движении её больше не нужно выводить на кнопку, иначе она будет слегка мигать.

Чтобы избежать этого, в глобальной *переменной*

```
LastComp: integer= 0;
```

мы будем хранить значение тега того компонента, который *перед этим* вызывал процедуру *FormMouseMove*. Тогда мы легко проверим, сменился компонент или нет. Если это тот же самый компонент, то мы просто завершаем процедуры, без всяких последствий для кнопки.

Ну а дальше всё просто. Если курсор заходит на «новую» кнопку, то картинка на ней изменяется, а значение тега запоминается в переменной *LastComp*. Если до этого курсор был на *другой* кнопке, то прежде её следует вернуть в неактивное состояние:

```
//МЫШКА ПЕРЕМЕЩАЕТСЯ ПО ФОРМЕ И КОМПОНЕНТАМ НА НЕЙ
```

```
procedure TForm1.FormMouseMove(Sender: TObject; Shift:
TShiftState; X,
  Y: Integer);
var n: integer;
begin
  //на каком компоненте находится курсор:
  n:= (Sender as TComponent).tag;
  //тот же компонент - ничего не делать:
  if n= LastComp then exit;

  //выключить прежнюю кнопку:
  case LastComp of
    101: imgSavePosL.Picture.Assign(Image19.Picture);
    102: imgClearL.Picture.Assign(Image4.Picture);
    103: imgSavePosR.Picture.Assign(Image19.Picture);
    104: imgOpen.Picture.Assign(Image10.Picture);
    105: imgClearR.Picture.Assign(Image4.Picture);
    106: imgSolution.Picture.Assign(Image14.Picture);
    107: imgCopyLR.Picture.Assign(Image12.Picture);
    108: imgCopyRL.Picture.Assign(Image6.Picture);
    109: imgUndo.Picture.Assign(Image2.Picture);
    110: imgRedo.Picture.Assign(Image8.Picture);
    111: imgExit.Picture.Assign(Image20.Picture);
    112: imgHelp.Picture.Assign(Image22.Picture);
  end;

  //зашли на новую кнопку - включить:
  LastComp:= (Sender as TComponent).tag;
  case LastComp of
      0: exit;  //- не кнопка
    101: imgSavePosL.Picture.Assign(Image18.Picture);
    102: imgClearL.Picture.Assign(Image3.Picture);
    103: imgSavePosR.Picture.Assign(Image18.Picture);
    104: imgOpen.Picture.Assign(Image9.Picture);
    105: imgClearR.Picture.Assign(Image3.Picture);
    106: imgSolution.Picture.Assign(Image13.Picture);
    107: imgCopyLR.Picture.Assign(Image11.Picture);
    108: imgCopyRL.Picture.Assign(Image5.Picture);
    109: imgUndo.Picture.Assign(Image1.Picture);
    110: imgRedo.Picture.Assign(Image7.Picture);
    111: imgExit.Picture.Assign(Image17.Picture);
    112: imgHelp.Picture.Assign(Image21.Picture);
  end;
end;  //FormMouseMove
```

Вы можете добавлять сколько угодно своих кнопок, не забывая при этом дописывать нужные строки в процедуру *FormMouseMove*. Так как изображения кнопок в активном и пассивном положениях могут быть любыми, то вам доступны всякие визуальные эффекты. Вот так простыми средствами можно получить вполне профессиональный интерфейс.

Но продолжим разбор кнопок.

Очень просты в программировании кнопки, которые дублируют пункты меню. Например, кнопки *imgUndo* и *imgRedo* вынесены на форму для удобства пользования, так как постоянно открывать меню – занятие довольно нервное. Они попросту вызывают те же процедуры, что и соответствующие пункты меню.



```
Left = 168
Top = 8
Height = 27
Width = 37
Hint = 'Ход назад'
Tag = 109
OnMouseDown = imgUndoMouseDown
OnMouseMove = FormMouseMove
OnMouseUp = imgUndoMouseUp
```



```
Left = 224
Hint = 'Ход вперёд'
Tag = 110
OnMouseDown = imgRedoMouseDown
OnMouseMove = FormMouseMove
OnMouseUp = imgRedoMouseUp
```

```
//ВОЗВРАТ ХОДА
procedure TForm1.imgUndoMouseDown(Sender: TObject; Button: TMouse-
Button;
  Shift: TShiftState; X, Y: Integer);
begin
  imgUndo.Picture.Assign(Image2.Picture);
end; //imgUndoMouseDown
procedure TForm1.imgUndoMouseUp(Sender: TObject; Button: TMouse-
Button;
  Shift: TShiftState; X, Y: Integer);
begin
  miUndoClick(Self);
  imgUndo.Picture.Assign(Image1.Picture);
end; //imgUndoMouseUp

//ХОД ВПЕРЁД
procedure TForm1.imgRedoMouseDown(Sender: TObject; Button: TMouse-
Button;
  Shift: TShiftState; X, Y: Integer);
begin
  imgRedo.Picture.Assign(Image8.Picture);
```

```
end; //imgRedoMouseDown
procedure TForm1.imgRedoMouseUp(Sender: TObject; Button: TMouse-
Button;
  Shift: TShiftState; X, Y: Integer);
begin
  miRedoClick(Self);
  imgRedo.Picture.Assign(Image7.Picture);
end; //imgRedoMouseUp
```

Несложно и убрать «мусор» с левого и правого полей. В *первом* случае достаточно занести в массив *masPole* нули, а затем обновить поле. Во *втором* нужно обновить и правое, и левое поле, чтобы «синхронизировать» их содержимое.



imgClearL:

```
Left = 88
Top = 4
Hint = 'Очистить левое поле'
Tag = 102
OnMouseDown = imgClearLMouseDown
OnMouseMove = FormMouseMove
OnMouseUp = imgClearLMouseUp
```

imgClearR:

```
Left = 580
Hint = 'Очистить правое поле'
Tag = 105
OnMouseDown = imgClearRMouseDown
OnMouseMove = FormMouseMove
OnMouseUp = imgClearRMouseUp
```

```
//ОЧИСТИТЬ ЛЕВОЕ ПОЛЕ
procedure TForm1.imgClearLMouseDown(Sender: TObject; Button:
TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  imgClearL.Picture.Assign(Image4.Picture);
end; //imgClearLMouseDown
procedure TForm1.imgClearLMouseUp(Sender: TObject; Button: TMouse-
Button;
  Shift: TShiftState; X, Y: Integer);
var
  i, j: integer;
begin
  for j:= 1 to dgPole.RowCount-1 do
    for i:= 1 to dgPole.ColCount-1 do
```

```
      masPole[i,j]:= 0;
  dgPole.Invalidate;
  MemoryIn;
  imgClearL.Picture.Assign(Image3.Picture);
end; //imgClearLMouseUp

//ОЧИСТИТЬ ПРАВОЕ ПОЛЕ
procedure TForm1.imgClearRMouseDown(Sender: TObject;
  Button: TMouseButton; Shift: TShiftState; X, Y: Integer);
begin
  imgClearR.Picture.Assign(Image4.Picture);
end; //imgClearRMouseDown
procedure TForm1.imgClearRMouseUp(Sender: TObject; Button: TMouse-
Button;
  Shift: TShiftState; X, Y: Integer);
var
  i, j: integer;
begin
  for j:= 1 to dgPole2.RowCount-1 do
    for i:= 1 to dgPole2.ColCount-1 do
      masPole2[i,j]:= 0;
  dgPole2.Invalidate;
  masPole:= masPole2;
  dgPole.Invalidate;
  MemoryIn;
  frmProtokol.listbox1.Clear;
  imgClearR.Picture.Assign(Image3.Picture);
end; //imgClearRMouseUp
```

Очень полезными при разработке собственных заданий могут оказаться следующие кнопки.

imgCopyLR:



```
Left = 428
Hint = 'Копировать в правое поле'
Tag = 107
OnMouseDown = imgCopyLRMouseDown
OnMouseMove = FormMouseMove
OnMouseUp = imgCopyLRMouseUp
```

imgCopyRL:



```
Left = 480
Hint = 'Копировать в левое поле'
Tag = 108
```

```
OnMouseDown = imgCopyRLMouseDown
OnMouseMove = FormMouseMove
OnMouseUp = imgCopyRLMouseUp
```

Нажатие на *первую* из них приводит к копированию фишек с левого поля в правое:

```
//КОПИРОВАТЬ ЛЕВОЕ ПОЛЕ В ПРАВОЕ
procedure TForm1.imgCopyLRMouseDown(Sender: TObject; Button:
TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  //заменить картинку:
  imgCopyLR.Picture.Assign(Image12.Picture);
  masPole2:= masPole;
  dgPole2.Invalidate;
end; //imgCopyLRMouseDown
procedure TForm1.imgCopyLRMouseUp(Sender: TObject; Button: TMouse-
Button;
  Shift: TShiftState; X, Y: Integer);
begin
  imgCopyLR.Picture.Assign(Image11.Picture);
end; //imgCopyLRMouseUp
```

Нажатие на *вторую* приведёт к прямо противоположному результату:

```
//КОПИРОВАТЬ ПРАВОЕ ПОЛЕ В ЛЕВОЕ
procedure TForm1.imgCopyRLMouseDown(Sender: TObject;
  Button: TMouseButton; Shift: TShiftState; X, Y: Integer);
begin
  imgCopyRL.Picture.Assign(Image6.Picture);
  masPole:= masPole2;
  dgPole.Invalidate;
end; //imgCopyRLMouseDown
procedure TForm1.imgCopyRLMouseUp(Sender: TObject; Button: TMouse-
Button;
  Shift: TShiftState; X, Y: Integer);
begin
  imgCopyRL.Picture.Assign(Image5.Picture);
end; //imgCopyRLMouseUp
```

Решив предложенные задачи, вы наверняка захотите составить и свои. Тогда незамысловато выставляйте нужные фишки на одном из полей или делайте ходы на левом поле, а затем полученную позицию сохраните на диске. Таким образом вы сможете пополнить библиотеку заданий для этой игры.

Чтобы сохранить игровое поле, нажмите левую кнопку.

imgSavePosL:



```
Left = 44
Hint = 'Записать текущую позицию'
Tag = 101
OnMouseDown = imgSavePosLMouseDown
OnMouseMove = FormMouseMove
OnMouseUp = imgSavePosLMouseUp
```

Чтобы сохранить поле-образец, такую же кнопку, но справа.

imgSavePosR:



```
Left = 532
Hint = 'Записать текущую позицию'
Tag = 103
OnMouseDown = imgSavePosLMouseDown
OnMouseMove = FormMouseMove
OnMouseUp = imgSavePosLMouseUp
```

Так как и кнопки, и операции сохранения практически одинаковы для обоих полей, то давайте объединим их. Нажатие и отпускание кнопок осуществляется в *процедурах*

```
//ЗАПИСАТЬ ПОЗИЦИЮ НА ЛЕВОМ ИЛИ ПРАВОМ ПОЛЕ
procedure TForm1.imgSavePosLMouseDown(Sender: TObject; Button:
TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  imgSavePosL.Picture.Assign(Image19.Picture)
end; //imgSavePosLMouseDown
procedure TForm1.imgSavePosLMouseUp(Sender: TObject; Button:
TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  imgSavePosL.Picture.Assign(Image18.Picture);
  if (Sender as TComponent).tag= 101 then begin // - левое поле
    SavePos(masPole) end
  else begin                                    // - правое поле
    SavePos(masPole2) end;
end; //imgSavePosLMouseUp
```

А непосредственное сохранение заданного поля происходит в процедуре *SavePos*. Все задачи записываются в папку *Figures* и имеют расширение *LGS*.

Причём сохраняется всё поле целиком, что существенно упрощает процедуры записи и загрузки заданий. И, наконец, в заголовке формы выводится имя файла (задачи), заданное вами при его сохранении. Одновременно это имя присваивается глобальной *переменной*

```
  NameFig: string= '';
```

Благодаря этому вы всегда легко определите название решаемой задачи. Если захотите, то можете расставить задачи по уровню сложности и после того, как пользователь вашей программы справится с одним заданием, предложить ему задание более сложное.

```
//ЗАПИСАТЬ ПОЗИЦИЮ НА ЗАДАННОМ ПОЛЕ
procedure TForm1.SavePos(arr: TPole);
var
  F: textfile;
  fn,s: string;
  i,j: integer;
begin
  savedialog1.DefaultExt:='txt';
  savedialog1.Filter:='Задачи (*.lgs)|*.LGS';
  savedialog1.FilterIndex:=1;
  s:=extractfilepath(application.exename)+'Figures\';
  savedialog1.InitialDir:= s;
  savedialog1.Title:='Запишите позицию на диск';
  if NameFig<>'' then savedialog1.filename:= NameFig
  else savedialog1.filename:='temp.lgs';
  if not savedialog1.Execute then exit;
  //имя конечного файла:
  fn:= savedialog1.filename;
  NameFig:=fn;
  assignfile(f,fn);
  rewrite(f);
  //записать позицию:
  for j:= 1 to dgPole.RowCount-1 do begin
    s:='';
    for i:= 1 to dgPole.ColCount-1 do s:=s+ inttostr(arr[i][j]);
    writeln (f,s);
  end;
  closefile(f);
  //вывести в заголовок формы имя файла:
  form1.caption:= NAME_PROG + '    [' + NameFig + ']';
  messagebeep(0)
end; //SavePos
```

Вполне естественно, что задачу на диске нужно уметь *загружать* для решения в правое поле. За это будет отвечать кнопка

imgOpen:



```
Left = 1
Hint = 'Загрузить задачу с диска'
Tag = 104
OnMouseDown = imgOpenMouseDown
OnMouseMove = FormMouseMove
```

```
//ЗАГРУЗИТЬ ЗАДАЧУ
procedure TForm1.imgOpenMouseDown(Sender: TObject; Button: TMouse-
Button;
  Shift: TShiftState; X, Y: Integer);
begin
  imgOpen.Picture.Assign(Image10.Picture);
  Open('');
end; //imgOpenMouseDown
```

При нажатии этой кнопки вызывается процедура *Open* с *пустой* строкой в качестве параметра. В этом случае появляется диалоговое окно открытия файла, и вы можете выбрать любой из имеющихся на диске. Чтобы загрузить *определённую* задачу (например, при переходе на следующий уровень сложности), следует передать в процедуру *имя файла*. Осталось добавить, что если вы не планируете набирать условия задач в текстовом редакторе (а это очень легко сделать), то можно ещё более упростить процедуру, убрав из неё проверки правильности данных на диске.

```
//ЗАГРУЗИТЬ ЗАДАЧУ С ЗАДАННЫМ ИМЕНЕМ
procedure TForm1.Open (fn: string);
var
  s: string;
  F: TextFile;
  nLines,Len: integer;
  i,j:integer;
  ss: array[1..MAX_POLE_HEIGHT] of string;
begin
  if fn='' then begin  //- имя файла не задано
    form1.opendialog1.DefaultExt:='lgs';
    form1.opendialog1.Filter:='Logos files (*.lgs)|*.LGS';
    s:=extractfilepath(application.exename)+'FIGURES\';
    form1.opendialog1.InitialDir:= s;
    form1.opendialog1.Title:='Загрузите новую задачу';
    if not form1.opendialog1.Execute then exit;
    s:= form1.opendialog1.filename
    end
  else s:=extractfilepath(application.exename)+'FIGURES\'+ fn;

  //название задачи:
  NameFig:=s;

  {$i-}
```

```
  AssignFile(F,NameFig);
  Reset(F);
  {$i+}
  if IOResult<>0 then begin  //- ошибка при загрузке файла
    application.MessageBox ('Такой задачи нет!',NAME_PROG, MB_OK);
    exit
  end;

  nLines:=0;
  Len:=0;
  while not eof(f) do begin
    inc(nLines);
    //считать строку из файла:
    Readln(F, S);
    ss[nLines]:=s;
    If (Length(s) <> Len) and (Len<>0) Then begin
      application.MessageBox ('Неверная длина строки!',NAME_PROG,
MB_OK);
      exit
      end
    else
      Len:= Length(s);
  end;
  //закрыть файл:
  CloseFile(F);
  //проверить данные:
  if (Len> MAX_POLE_WIDTH) or (nLines> MAX_POLE_HEIGHT) then begin
    application.MessageBox('Неверные данные!',NAME_PROG, MB_OK);
    exit
  end;
  //вывести в заголовок имя файла:
  form1.caption:= NAME_PROG + '    [' + NameFig + ']';

  //очистить массивы полей:
  for j:= 1 to dgPole.RowCount-1 do
    for i:= 1 to dgPole.ColCount-1 do begin
      masPole[i,j]:= 0;
    end;
  //и протокол:
  frmProtokol.listbox1.items.Clear;
  //заполнить массив новыми данными:
  for j:=1 to nLines do
    for i:=1 to Len do
       masPole2[i][j]:= strtoint(ss[j,i]);
  //запомнить начальную позицию:
  MemoryIn;
  //обновить поля:
  dgPole.Invalidate;
  dgPole2.Invalidate;
  //обнулить ходы:
  Hod:= 0;   Hod_max:= Hod;
  lblHod.Caption:= 'Ход - ' + inttostr(Hod);
```

```
end; //Open
```

Очень удобно с помощью этой процедуры при открытии программы загружать задание первого уровня сложности. Это можно сделать в уже рассмотренной нами процедуре *FormShow:*

```
//ПОКАЗАТЬ ФОРМУ
procedure TForm1.FormShow(Sender: TObject);
begin
  open('001.lgs');
  frmProtokol.show;
end; //FormShow
```

В результате сразу же после старта программы в правое поле будет загружена задача, и пользователь сможет приступить к её решению.

Следует также позаботиться о сохранении найденного *решения* задачи. Оно находится в списке *ListBox1* на форме протокола. При щелчке на кнопке *Записать*, формируется название файла решения. Если задача уже имеет название (оно находится в заголовке формы, как вы знаете), то к нему добавляются буквы *otv*, чтобы задача и её решение имели схожие имена. Если задача *безымянная*, то ответ будет записан в файл *otv_temp.txt* (файл ответа (решения) имеет расширение *txt*, так как это обычный текстовый файл):

```
//ЗАПИСАТЬ ОТВЕТ
procedure TfrmProtokol.sbtSaveOtvClick(Sender: TObject);
var
  s,ss,fn: string;
  i: integer;
  f: textfile;
begin
  savedialog1.DefaultExt:='txt';
  savedialog1.Filter:='Решения (*.txt)|*.TXT';
  savedialog1.FilterIndex:=1;
  s:=extractfilepath(application.exename)+'Figures\';
  savedialog1.InitialDir:= s;
  savedialog1.Title:='Запишите решение на диск';

  //сформировать имя файла:
  if NameFig<>'' then begin
    s:=extractFileName(NameFig);
    ss:='';
    i:=1;
    while (s[i]<>'.') and (i<=length(s)) do begin
      ss:=ss+s[i];
      inc(i)
    end;
    ss:=ss+'otv.txt';
```

```
    savedialog1.filename:=ss
   end
   else
     savedialog1.filename:='otv_temp.txt';

   if not savedialog1.Execute then exit;
   //имя конечного файла:
   fn:= savedialog1.filename;
   assignfile(f,fn);
   rewrite(f);
   //записать протокол:
   for i:=0 to (ListBox1.Items.Count - 1) do
     writeln (f,ListBox1.Items.Strings[i]);
   Closefile(f);
   Messagebeep(0)
end; //sbtSaveOtvClick
```

Очистить протокол вы сможете, нажав кнопку *Стереть*:

```
//ОЧИСТИТЬ ПРОТОКОЛ
procedure TfrmProtokol.SpeedButton1Click(Sender: TObject);
begin
  ListBox1.Items.Clear;
end; //SpeedButton1Click
```

Веб спешит на помощь

Теперь, когда разработка интерфейса программы почти завершена (автоматическое решение задачи будет рассмотрено в следующем разделе), мы можем написать «справочное пособие» по игре *Logos*. Как вы знаете, это можно сделать по-разному, но в этой игре мы освоим новый способ – оформим файл справки в виде веб-страницы. *Internet Explorer* уже интегрирован в операционную систему *Windows*, так что проблем с просмотром таких страниц ни у кого не будет. А, кроме новизны, мы получим современную справку, которую легко разработать хотя бы в текстовом редакторе *MS Word* (вы также легко найдёте множество программ, специально предназначенных для создания веб-страниц и веб-дизайна). Простую веб-страницу сделать ничуть не сложнее, а, пожалуй, даже и легче, чем справку в формате *HLP*.

Итак, кнопка *imgHelp*



```
Left = 368
```

```
Hint = 'Помощь'
Tag = 112
OnMouseDown = imgHelpMouseDown
OnMouseMove = FormMouseMove
OnMouseUp = imgHelpMouseUp
```

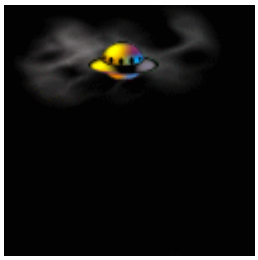
и пункт меню *miHelp* должны запустить *Internet Explorer* и открыть в нём наш файл справки. Для этого мы воспользуемся функцией *ShellExecute* из *Windows API* с соответствующими параметрами. Но прежде следует добавить модуль *ShellApi* в раздел *uses* (он находится в начале модуля главной формы *LogosUnit*).

Вызов справки совершенно одинаков в обоих случаях, поэтому вы можете объединить две следующие процедуры в одну.

```
//ПОКАЗАТЬ ФАЙЛ СПРАВКИ
procedure TForm1.miHelpClick(Sender: TObject);
var s: string;
begin
  s:= extractfilepath(application.exename)+'LogosHelp.htm';
  ShellExecute(handle, pChar('open'),pChar('iexplore'),pChar(s),
  '', SW_SHOWMAXIMIZED);
end;  //miHelpClick
procedure TForm1.imgHelpMouseDown(Sender: TObject; Button: TMouse-
Button;
  Shift: TShiftState; X, Y: Integer);
begin
  imgHelp.Picture.Assign(Image22.Picture);
end;  //imgHelpMouseDown
procedure TForm1.imgHelpMouseUp(Sender: TObject; Button: TMouse-
Button;
  Shift: TShiftState; X, Y: Integer);
var s: string;
begin
  imgHelp.Picture.Assign(Image21.Picture);
  s:= extractfilepath(application.exename)+'LogosHelp.htm';
  ShellExecute(handle, 'open','iexplore',pChar(s),
  '', SW_SHOWMAXIMIZED);
end;  //imgHelpMouseDown
```

Дело за малым – написать файл справки в редакторе *MS Word* и сохранить его как веб-страницу (вы также можете подобрать *готовую* веб-страницу и изменить надлежащим образом текст и рисунки на ней).

Файл справки может быть таким:

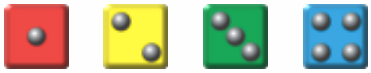


LOGOS

Отличная игра-головоломка, суть которой состоит в том, чтобы составить на игровом поле заданную фигуру из фишек.

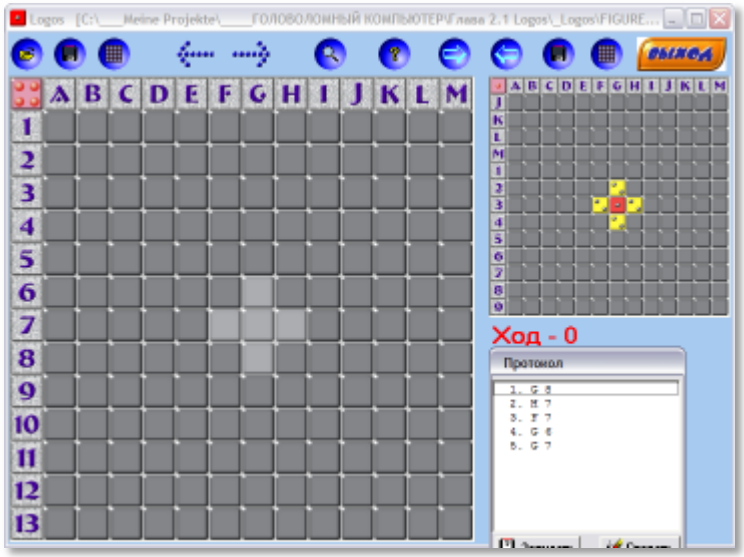
Правила игры

Каждая фишка имеет на верхней стороне 1, 2, 3 или 4 очка:



Фишки последовательно выставляются на **свободные клетки** игрового поля. У новой фишки на верхней стороне всегда находится одно очко. А вот все соседние с ней по горизонтали (слева и справа) и вертикали (сверху и снизу) фишки увеличивают своё значение на единицу, то есть фишка с одним очком превращается в фишку с двумя очками, фишка с двумя очками – в фишку с тремя очками, фишка с тремя очками – в фишку с четырьмя очками. Так как фишек с пятью очками нет, то фишка с четырьмя очками вновь становится «одноочковой».

Искомая конфигурация фишек приведена на маленьком поле справа. На игровом поле выделены более светлым оттенком клетки для выполнения ходов.



Чтобы сделать *ход*, достаточно просто щёлкнуть мышкой в нужной клетке игрового поля. Например, для решения этой задачи необходимо сначала, в любой последовательности, щёлкнуть на внешних клетках G6, F7, H7, G8, а затем в центральной – G7. Ход «партии» вы можете видеть в плавающем окне с заголовком «Протокол».

Управление игрой

Управлять игрой можно с помощью ***кнопок*** в верхней части окна приложения или ***всплывающего меню***, которое вызывается нажатием правой кнопки мыши.

Управляющие кнопки

 - открыть список заданий на диске; вы можете выбрать и загрузить любое из них.

 - сохранить позицию на игровом поле (левая кнопка) или поле с заданием (правая кнопка). Позволяет разрабатывать и записывать на диск свои собственные задачи.

 - стереть все фишки с игрового поля (левая кнопка) или с поля-образца (правая кнопка).

 - отменить последний ход.

 - вернуть отменённый ход.

 - найти решение задачи.

 - показать справку помощи.

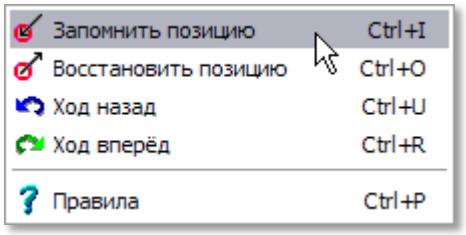
 - копировать левое поле в правое.

 - копировать правое поле в левое.



- закрыть программу.

Всплывающее меню



Последние три пункта меню просто дублируют соответствующие управляющие кнопки.

Выбрав первый пункт, вы запомните позицию на игровом поле, а, выбрав второй, восстановите её. Эти пункты меню помогут вам вернуться к тому положению на поле, решение которого вы уже нашли.

Как вы видите (Рис. Ф5.16), можно использовать различные шрифты и вставлять картинки, например, анимированные *GIF*-файлы; один из них находится в начале текста справки (летающая тарелка; естественно, в текстовом редакторе виден только один «кадр» анимации, так что зрелище, конечно, не то).

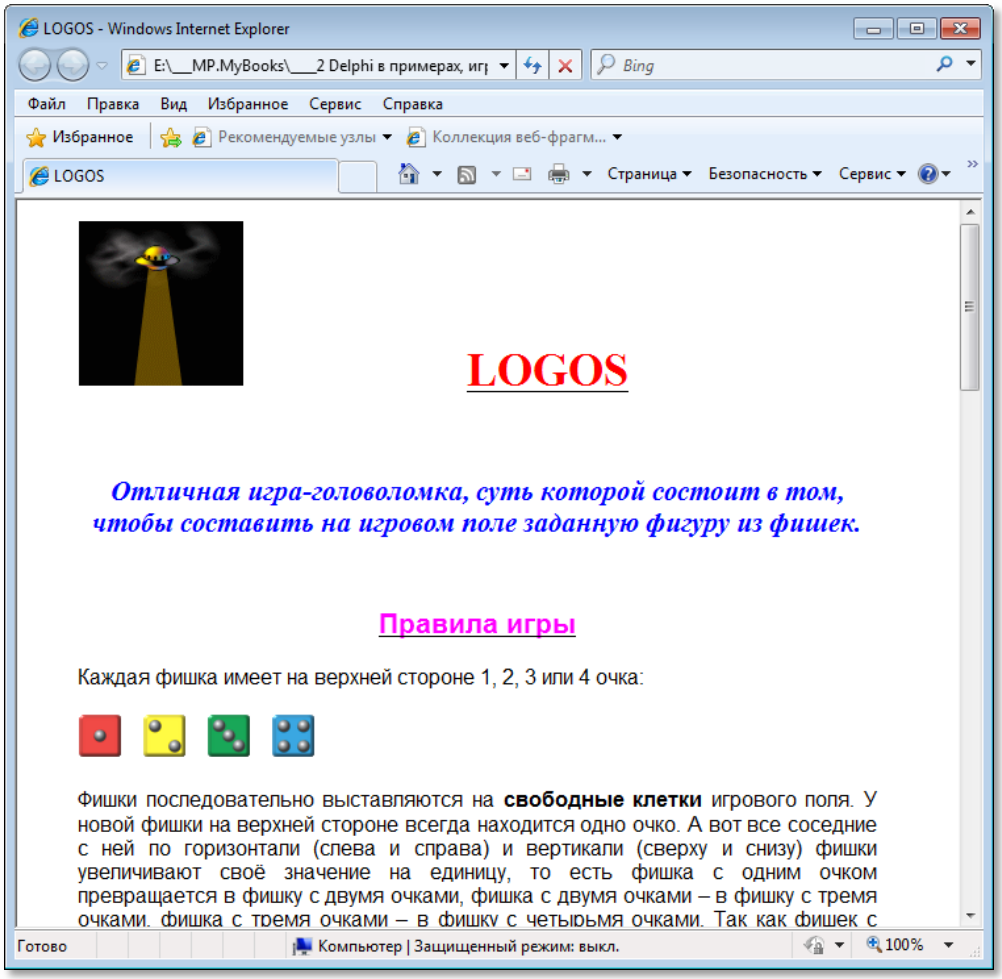


Рис. Ф5.16. Справка открыта в браузере *Internet Explorer*

Для разработки статичных картинок годится любой графический редактор, а для создания анимации существует много специализированных *GIF-аниматоров*, которые позволяют легко добавлять к вашим картинкам разнообразные визуальные спецэффекты.

После того как вы набрали текст помощи и вставили рисунки, добавьте переходы в нужные места файла. Для этого выделите текст, при щелчке на котором будет осуществляться переход, и вызовите контекстное меню, нажав *правую* кнопку мыши. В нём выберите пункт *Гиперссылка*, после чего откроется диалоговое окно *Вставка гиперссылки*. Так как место перехода находится в том же файле, то выберите слева пункт *Связать с местом в документе* и в списке укажите заголовок в файле помощи (у нас они выделены сиреневым цветом и подчёркиванием – не забудьте при наборе заголовка в списке стилей выбрать *Заголовок 1 (2, 3), начало документа* или *закладку!* Более подробно о создании гиперссылок вы можете прочитать в *Справке по Microsoft Office Word*.

И последнее – сохраните свою работу как веб-страницу в папке с игрой под именем *LogosHelp*. Для проверки загрузите справку в *Internet Explorer*, а если обнаружите недостатки, то прямо из него перешлите для доработки в *Word*. О том, как работать в редакторе, вы уже знаете.

Естественно, ту же самую справку вы увидите, если попросите помощи непосредственно из программы *Logos*. Я надеюсь, результаты вас полностью удовлетворят. Впрочем, вы можете сколько угодно улучшать свою «веб-страницу», украшая её новыми элементами. Но и в «простом» варианте помощь выглядит отлично!

Некоторые особенности программы не отражены в предлагаемом файле справки. Если вы считаете нужным упомянуть о них, то сделайте это!

Шиворот-навыворот и задом наперёд

Конец – делу венец.

Русская пословица

Если вы пробовали решать предложенные в начале главы задачи, то, конечно, ощутили на себе, как непросто решить даже эти элементарные «примеры». Что уж говорить о «навороченных»!

А всё потому, что большинство фишек в ходе игры претерпевает несколько превращений, за которыми трудно уследить. Те, кто играл в реверси, прекрасно знакомы с подобной проблемой.

Безусловно, по конечной позиции сложно вычислить, с какой фишки началась игра и какие фишки пристраивались к ней потом. Зато с *последней* фишкой всё гораздо проще – из правил игры неотвратимо следует, что последняя фишка обязательно имеет *одно* очко на верхней грани, ведь после неё фишки больше не выставлялись, а, значит, она не могла изменить своего значения. Таким образом, вся «хитрость» задач игры *Logos* заключается в том, что их нужно решать с *конца*.

С ретроспективным анализом мы уже встречались при разработке игры *XorGame*, так что эта новость не должна застигнуть вас врасплох. Поэтому давайте решим с помощью этой методы задачи-примеры.

Задача 1.

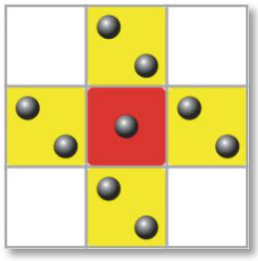


Рис. Ф5.17. Если не знаешь, с чего начать, начни с конца!

В конечной позиции только одна фишка – центральная – имеет одно очко на верхней грани (Рис. Ф5.17). Из вышесказанного совершенно очевидно, что она и была выставлена на поле последней.

С ней граничат 4 фишки, которые в результате последнего хода *увеличили* своё значение на единицу. То есть, если убрать центральную фишку, то все остальные фишки должны *уменьшить* своё значение на единицу. Стало быть, перед последним ходом ситуация на поле выглядела так (Рис. Ф5.18).

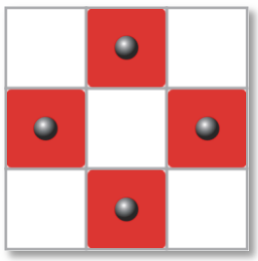


Рис. Ф5.18. Ход назад!

У этих фишек нет соседей, поэтому в ходе игры они не могли изменить своего значения, следовательно, их можно выставлять на поле в *любом* порядке, например, так (Рис. Ф5.19).

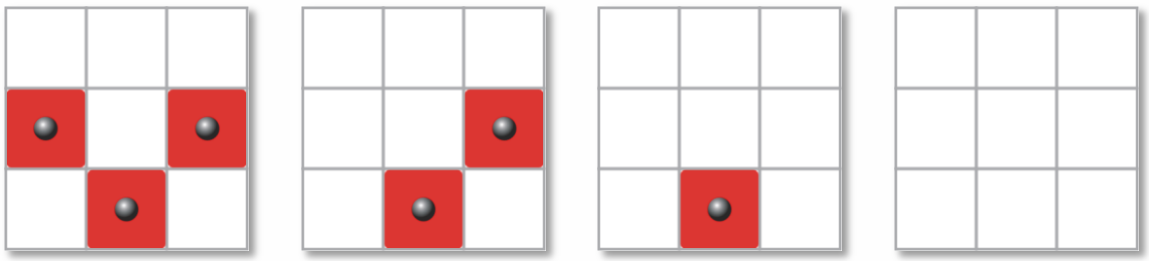


Рис. Ф5.19. Пятимся назад!

Таким образом, пользуясь ретроспективным анализом, мы пришли от конечной позиции к *начальной*. А в игре нужно, наоборот, перевести начальную позицию в конечную, поэтому следует *обратить* ходы - выставить сначала все периферийные фишки, а уже затем центральную.

Решим более сложную *задачу 2* (Рис. Ф5.20).

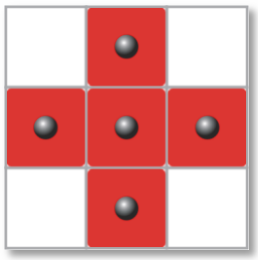


Рис. Ф5.20. Вторая задача

В конечной позиции *все* фишки имеют одно очко на верхней грани - так какая же из них была выставлена последней?

Легко понять, что фишка с одним очком либо вообще не изменяла своего значения в игре, либо изменяла его четырежды, чтобы совершить полное превращение: 1 → 2 → 3 → 4 → 1. Во втором случае у неё должно быть ровно 4 соседа. Отсюда следует важный вывод: *все фишки с одним очком, имеющие не более трёх соседних, никогда не изменяли своего значения*, то есть после них на поле не выставлялись фишки, граничащие с ними по длине стороны.

Поскольку у всех периферийных фишек только *один* сосед, то их можно смело удалять с поля в любой последовательности (Рис. Ф5.21).

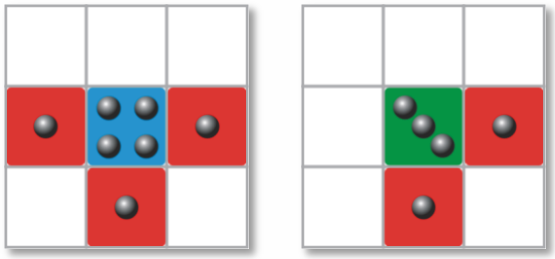


Рис. Ф5.21. Удаляем беспощадно

Дальнейшее решение не требует иллюстраций. Без сомнения, мы придём к центральной фишке с одним очком, которой и должен быть сделан первый ход.

В задаче 3 (Рис. Ф5.22) легко отыскать фишки с одним очком, имеющие менее четырёх соседей, - это все периферийные фишки.

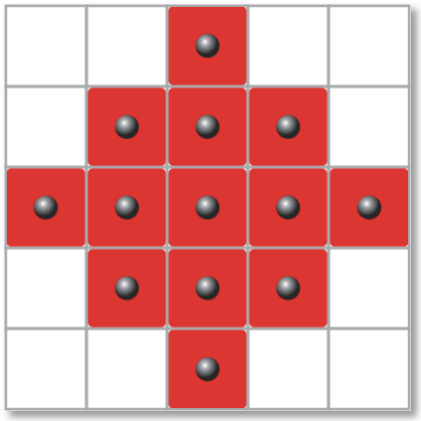


Рис. Ф5.22. Однако, ромбик!

Убирая их в любой последовательности (внимательно следите за превращениями остальных фишек!), мы неизбежно получим такую фигуру (Рис. Ф5.23).

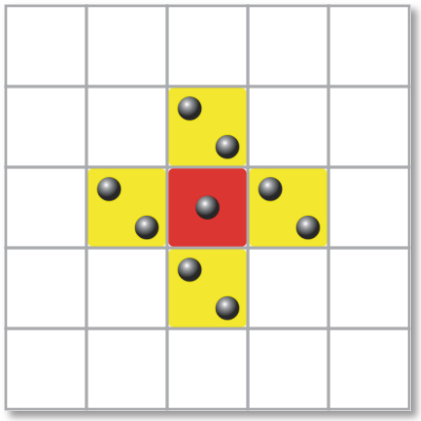


Рис. Ф5.23. Знакомая позиция!

Нетрудно узнать в ней первую задачу, которую мы уже благополучно решили.

Нечасто, но бывают такие ситуации на поле, когда *все* фишки с одним очком имеют по четыре соседа, поэтому найденное нами правило о трёх соседях здесь не действует. Например, в *задаче 25* после удаления четырёх фишек на границе фигуры возникнет такая «неразрешимая» позиция (Рис. Ф5.24).

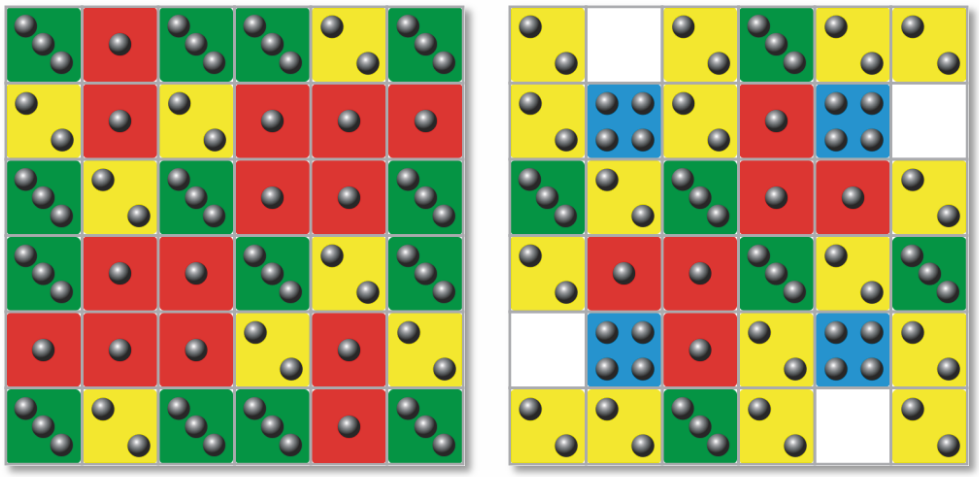


Рис. Ф5.24. Куда пойти?

Политически неверно было бы удалить здесь те фишки с одним очком, у которых нет соседних фишек с двумя очками. В результате таких ходов вы загоните себя в тупик, так как на поле вообще не останется ни одной фишки с одним очком и ходить будет нечем. Отсюда вытекает *второе* важное правило: *если на поле остались только такие фишки с одним очком, которые граничат с четырьмя другими, то удалять следует те из них, у которых есть соседняя фишка с двумя очками*.

Этих двух правил вполне достаточно, чтобы решить задачу любой сложности. Одна беда – при решении задачи мы будем двигаться в обратную сторону. Практически это выглядит так. Загружаем (или набираем вручную) в правое поле задачу, копируем её в левое поле кнопкой  и последовательно удаляем фишки, пользуясь нашими правилами. Если вы нигде не собьётесь с верного пути, то полностью очистите игровое поле от фишек. Теперь нужно сделать то же самое, но в обратном порядке. Ходы запомнили? Шутка – у нас все ходы записаны в протоколе, поэтому просто ходите в нужные клетки, начиная с конца списка ходов. Небось, надоело высчитывать клетки по их координатам? Тогда щёлкните в списке на нужной строке – и фишка появится в нужном месте в нужное время. Щёлкнули, а она не появилась? И это верно: прежде

чем щёлкать, нужно подумать о последствиях, всё-таки мышка – это вам не волшебная палочка!

В модуле *ProtokolUnit* формы протокола нужно добавить *процедуру*, обрабатывающую щелчок на одной из строк списка. Чтобы не прокручивать длинный список ходов вручную,мы также поручим это нашей процедуре. Далее в ней «расшифровывается» строка с записью хода, после чего выполняется ход в нужную клетку игрового поля.

```
procedure TfrmProtokol.ListBox1Click(Sender: TObject);
var
  s,h: string;
  n: integer;
  x, y: integer;
begin
  //прокрутить список так, чтобы сверху была выделенная строка:
  ListBox1.TopIndex:= ListBox1.ItemIndex;
  //выделенная строка из списка:
  s:= ListBox1.Items.Strings[ListBox1.ItemIndex];
  //найти точку:
  n:= pos('.',s);

  if n=0 then exit; //- это не запись хода!

  //выделить номер хода:
  h:= copy(s,1,n-1);
  Hod:= strtoint(h);
  //вывести его на экран:
  LogosUnit.Form1.lblHod.Caption:= 'Ход - ' + inttostr(Hod);
  //выделить координаты клетки:
  x:= pos(s[n+2],letter);
  s:= copy(s,n+4,2);
  y:= strtoint (s);
  //выполнить ход:
  masPole:= LogosUnit.Form1.DoMove(masPole, x, y);
  Form1.dgPole.invalidate;
  //проверить, не решена ли задача:
  if LogosUnit.Form1.IsReady then LogosUnit.Form1.Ready;
end; //ListBox1Click
```

При этом решение задачи всё равно придётся начинать с *последних* ходов и подниматься в списке наверх. Было бы гораздо удобнее выполнять ходы «нормально», то есть сверху вниз. Для осуществления своих народных чаяний установим на форме протокола внизу большую *кнопку sbtInv*:

```
Left = 4    Top = 192
Width = 185    Height = 22
Hint = 'Обратить ходы'
Caption = 'Обратить ходы'
Font.Style = [fsBold]
```

```
OnClick = sbtInvClick
```

и наделим её способностью всё ставить с головы на ноги:

```
//ОБРАТИТЬ СПИСОК ХОДОВ
procedure TfrmProtokol.sbtInvClick(Sender: TObject);
var
  s: array [0..169] of string;
  str: string;
  i,j: integer;
  n,m: integer;
begin
  n:= ListBox1.Items.Count-1;

  j:=-1;
  for i:= 0 to n do
    if ListBox1.Items.Strings[i] <> '' then begin
      inc(j);
      s[j]:= ListBox1.Items.Strings[i];
  end;
  ListBox1.Clear;

  for i:= j downto 0 do begin
    m:= pos('.',s[i]);
    str:= copy(s[i], m+2, length(s[i])-m);
    str:= format('%3d',[j-i+1]) + '. ' + str;
    ListBox1.Items.Add(str);
  end;
  ListBox1.Items.Add('');
end; //sbtInvClick
```

После этих «ухищрений» вы сначала решаете задачу «ретроспективно», а затем обращаете последовательность ходов, щёлкаете на строках протокола и наблюдаете любопытные картины из жизни фишек на игровом поле.

Если вы внимательный читатель, то должны были заметить, что на самом деле вы не можете делать ходы на игровом поле в обратном направлении (убирать фишки), так как программа умеет только *выставлять* фишки на поле. Правда, можно вдавить поглубже клавишу *Shitf* и просто изменить нужные фишки так, чтобы создавалась иллюзия обратного хода, но всё иллюзорное так ненадёжно! Поэтому в процедуре *dgPoleMouseDown* (об этом говорилось выше) предусмотрена возможность делать ходы назад. Для этого надо щёлкать на нужной фишке, удерживая клавишу *Alt*. Тогда вместо функции *DoMove* вызывается функция *BackMove*, которая корректно снимает фишку с поля.

Разобраться в действии этой функции несложно, особенно если сравнить её с *DoMove*:

```
//ХОД НАЗАД
function TForm1.BackMove(arr: TPole; x, y: integer): TPole;
var i: integer;
begin
  result:= arr;
  //изменить соседние фишки:
  for i:= Max(y-1,1) to Min(y+1,dgPole.RowCount-1) do //- верти-
кальный
  begin
    if (i<> y) and (arr[x,i] <> 0) then begin
      dec(arr[x,i]);
      if arr[x,i]=0 then arr[x,i]:= 4; end;
  end;
  for i:= Max(x-1,1) to Min(x+1,dgPole.ColCount-1) do //- горизон-
тальный
  begin
    if (i<> x) and (arr[i,y] <> 0) then begin
      dec(arr[i,y]);
      if arr[i,y]=0 then arr[i,y]:= 4; end;
  end;
  //уменьшить на единицу значение заданной фишки:
  dec(arr[x,y]);
  result:= arr;
end; //BackMove
```

Учимся лениться

Не появилось ли у вас «естественного» желания вообще всю работу по решению задачи свести к одному щелчку на желанной кнопке ? Вопрос, конечно, риторический - но прежде чем предаться лени и неге, нужно взвалить всю нетворческую работу на плечи компьютера.

Нажать на кнопку мы, правда, всегда сумеем:

```
//НАЙТИ РЕШЕНИЕ ЗАДАЧИ
procedure TForm1.imgSolutionMouseDown(Sender: TObject;
  Button: TMouseButton; Shift: TShiftState; X, Y: Integer);
begin
  imgSolution.Picture.Assign(Image14.Picture);
end; //imgSolutionMouseDown
procedure TForm1.imgSolutionMouseUp(Sender: TObject; Button:
TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  imgSolution.Picture.Assign(Image13.Picture);
```

```
  if GameState= gsSolution then exit;
  //перевести программу в режим поиска:
  GameState:= gsSolution;
  //искать решение:
  solution(masPole2);
  //перевести программу в режим ожидания ввода команд:
  GameState:= gsWait;
end; //imgSolutionMouseUp
```

А вот с процедурой поиска решения, увы, так просто не разделаться. Хотя самое главное мы уже знаем – это наши правила решения задач, остальное, как говорится, дело техники.

Так, мы легко можем определить необходимое число ходов, ведь оно в точности равно числу фишек в конечной позиции. Затем отыскиваем в текущей позиции, пока можно, фишки с одним очком на верхней грани, имеющие не более трёх соседей (*Правило 1*, функция *Sosedi*). Когда такие фишки закончатся, переходим к поиску фишек с одним очком, имеющим четырёх соседей, в том числе с двумя очками на верхней грани. И так далее, и так далее, и так далее… В конце концов, либо задача будет решена, либо мы определим, что она неразрешима.

Естественно, задача и в этом случае решается в обратном порядке, но в процедуре *SavePos* ходы записываются как следует, так что вы можете пощёлкать на них в списке и убедиться в правильности решения задачи, а также и сохранить его на диске.

```
//АВТОМАТИЧЕСКОЕ РЕШЕНИЕ ЗАДАЧИ
procedure TForm1.Solution(arr: TPole);
var
  Hod: integer;
  w,h: integer;
  x, y: byte;
  flgHod: Boolean;
  arr1, arr2: TPole;

  //Подсчитать число ходов (фишек)
  function Steps: integer;
  var
    i, j: integer;
  begin
    Result:= 0;
    for j:= 1 to h do
      for i:= 1 to w do
        if arr2[i,j]<> 0 then inc(Result)
  end;

  //Подсчитать число соседей у единицы
  function Sosedi: integer;
```

```
var
  i, j: integer;
  i1: integer;
begin
  //искать по всему полю:
  for j:= 1 to h do
    for i:= 1 to w do
      if arr2[i,j]=1 then begin  //- нашли единицу
        Result:= 0;
        //сколько соседей?
        for i1:= Max(j-1,1) to Min(j+1,h) do //- по вертикали
          if (i1<> j) and (arr2[i,i1] <> 0) then begin //- есть
            inc(Result); end;
        for i1:= Max(i-1,1) to Min(i+1,w) do //- по горизонтали
          if (i1<> i) and (arr2[i1,j] <> 0) then begin //- есть
            inc(Result); end;
        if Result< 4 then begin
          x:= i; y:= j;
          //дальше не искать:
          exit;
        end;
      end;
  //не найдены:
  Result:= -1;
end;

//Найти единицу с 4-мя соседями (с двойкой)
function Sosedi2: integer;
var
  i, j: integer;
  i1: integer;
  flg: Boolean;
begin
  //искать по всему полю:
  for j:= 1 to h do
    for i:= 1 to w do
      if arr2[i,j]=1 then begin  //- нашли единицу
        Result:= 0;
        flg:= False;
        //сколько соседей?
        for i1:= Max(j-1,1) to Min(j+1,h) do //- по вертикали
          if (i1<> j) and (arr2[i,i1] <> 0) then begin //- есть
            inc(Result);
            if arr2[i,i1] =2 then flg:= True; end;
        for i1:= Max(i-1,1) to Min(i+1,w) do //- по горизонтали
          if (i1<> i) and (arr2[i1,j] <> 0) then begin //- есть
            inc(Result);
            if arr2[i1,j]= 2 then flg:= True; end;

        if flg and (Result=4) then begin  //- нашли
          x:= i; y:= j;
          //дальше не искать:
          exit; end;
```

```
        end;
    //не найдены:
    Result:= -1;
  end;

  //записать ходы в протокол:
  procedure SavePos;
  var
    n: integer;
    x, y: integer;
    s: string;
  begin
    frmProtokol.listbox1.items.Clear;
    for n:= Hod  downto 1 do begin
      s:='';
      x:= Move[n].x; y:= Move[n].y;
      frmProtokol.ListBox1.items.add(format('%3d',[Hod-n+1]) + '.
' +
                          letter[x]+ ' ' +inttostr(y));
    end;
    frmProtokol.listbox1.items.add('');
  end;

begin
  //очистить протокол:
  frmProtokol.listbox1.Clear;

  //скопировать массивы полей:
  arr2:= masPole2;
  arr1:= masPole;
  //размеры полей:
  w:= dgPole.ColCount-1; h:= dgPole.RowCount-1;

  //подсчитать число ходов:
  step:= steps;
  if step= 0 then begin
    application.MessageBox ('Вы не загрузили задачу!',NAME_PROG,
MB_OK);
    exit
  end;

  Hod:= 0;

  repeat
    //ходим в клетки с 1, имеющей не более 3 соседей:
    while Sosedi<> -1 do
    begin
      //делаем ход в эту клетку:
      inc(Hod);
      Move[Hod].x:= x;
      Move[Hod].y:= y;
      arr2:= BackMove(arr2, x, y);
      masPole:= arr2; dgPole.Invalidate;
```

```
      //делаем задержку, чтобы увидеть процесс решения задачи:
      sleep(100);
      application.ProcessMessages;
      if hod = step then begin
         savepos;
         frmProtokol.Hide;
         application.MessageBox ('Задача решена!',NAME_PROG,
MB_OK);
         frmProtokol.Show;
         exit
      end
    end;
    flgHod:= False;
    if Sosedi2<> -1 then begin
      //делаем ход в эту клетку:
      flgHod:= True;
      inc(Hod);
      Move[Hod].x:= x;
      Move[Hod].y:= y;
      arr2:= BackMove(arr2, x, y);
      masPole:= arr2; dgPole.Invalidate;
      sleep(100);
      application.ProcessMessages;
      if hod = step then begin
         savepos;
         frmProtokol.Hide;
         application.MessageBox ('Задача решена!',NAME_PROG,
MB_OK);
         frmProtokol.Show;
         exit
      end;
    end;
  until flgHod= False;
  //не удалось решить задачу:
  application.MessageBox ('Задача решений не имеет!',NAME_PROG,
MB_OK);
  exit;
end;  //Solution
```

Хотел бы предостеречь вас от бездумного использования автоматического решения задач, ведь в таком случае программа теряет всякий смысл! А вот при разработке и проверке собственных заданий кнопка  вам пригодится для проверки своих творений. Не говоря уже о том, что найденные нами алгоритмы могут пригодиться вам при создании других программ!



Исходный код программы находится в папке **Logos**.



«С любимыми не расставайтесь…»

Если вы ещё не наигрались в *Logos*, то попробуйте приумножить достоинства этой игры новыми возможностями.

1. Если вы произвольно выставите фишки на поле, то, скорее всего, получите неразрешимую задачу. Тогда можно постараться решить её с *наименьшим* числом штрафных ходов. Штрафным назовём такой ход, при котором разрешается перевернуть любую фишку на следующую грань (или на предыдущую, или на единицу), не затрагивая соседних фишек.

С такими послаблениями всякая задача будет разрешима («правильные» задачи решаются без штрафных очков, остальные – с тем или иным количеством штрафных очков), но вот насколько легко будет найти *лучшее* решение?

2. Ни одна фишка на поле не выставляется, а конечная позиция является начальной. *Цель игры*: перевернуть все фишки единицей вверх. Ход выполняется так же, как в игре *ShortGame*. Различие только в том, что теперь у фишек не две стороны, а четыре. Усложнит ли это решение задачи?

3. Придумайте другие варианты игры *Logos*. Самый простой путь – искать их на стыке с другими играми («гибридизация» игр).

Факультатив 6. Цветные линии, или Критическая масса мыслей

Не каждое лыко в строку.

Русская пословица

Прежде чем мы приступим к разработке собственной программы, давайте познакомимся с мировым опытом в области собирания шариков.

Мир *Цветных линий*

Цветные линии (или *Color Lines*, на аглицкий манер) - одна из самых известных компьютерных игрушек (Рис. Ф6.1), была создана в недрах фирмы *Геймос* усилиями Олега Дёмина ещё в те времена, когда компьютерами управляла операционная система *MS DOS*. С тех пор она породила, пожалуй, не меньше вариантов, чем знаменитый *тетрис*.



Рис. Ф6.1. Мировая вещь!

И это не удивительно: правила игры очень просты и легко поддаются программированию, а сама игра настолько интересна, что от неё сначала трудно оторваться, а потом хочется вновь к ней возвращаться. Наиграться в неё невозможно!

В классическом варианте, представленном здесь программами *Color Lines* (версия для *DOS*, автор Олег Дёмин, 1992 год) и *LinesWin* той же фирмы *Геймос*, но уже для *Windows* (авторы Геннадий Денисов, Игорь Ивкин, Константин Миронович, 1995 год) игра ведётся на квадратном поле размерами 9 на 9 клеток (Рис. Ф6.2).

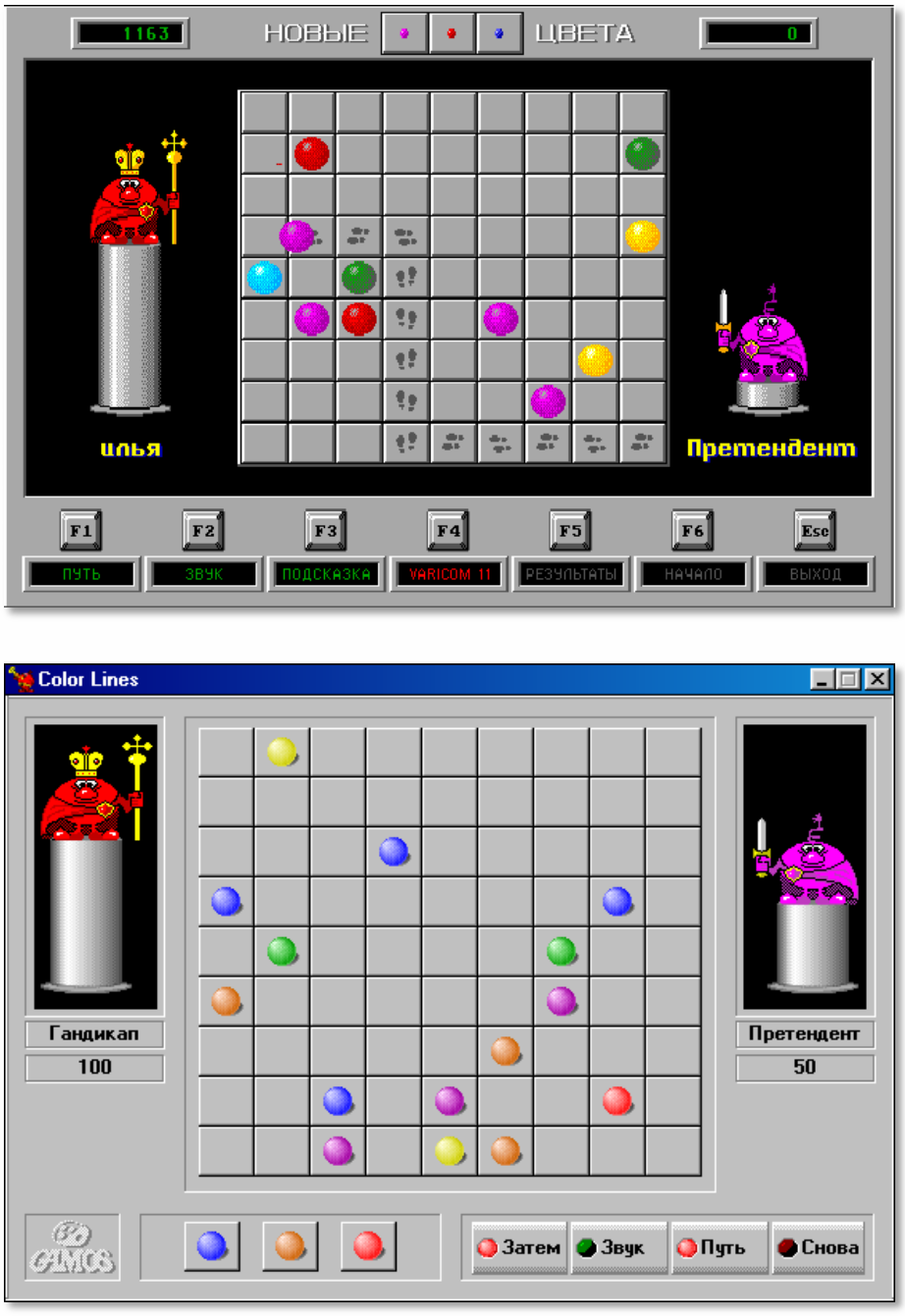


Рис. Ф6.2. «Исходники»!

При старте игры на поле появляются 5 разноцветных шариков (всего имеется 7 разных окрасок). Игрок на каждом ходу перемещает любой шарик в другую клетку, если он сможет добраться до неё ходом шахматной ладьи. Когда удастся выстроить непрерывную цепь шариков одного цвета длиной не менее пяти штук по горизонтали, вертикали или

диагонали, она (само)уничтожается, освобождая поле, а игрок получает по очку за каждый шарик, плюс возможность сделать ещё один ход. Если на поле одновременно возникает несколько цепочек, то все они будут «стёрты с лица земли».

Если в результате хода выстроить такой цепочки не удалось, то на поле в произвольных местах возникают 3 новых шарика. Вы можете заранее узнать цвет следующей порции шариков, если включите режим подсказки, но тогда призовых очков получите меньше.

Игра заканчивается, когда на поле не останется места для вывода очередных шариков.

Цель игры: набрать как можно больше очков и попасть в список рекордсменов. Легко понять, что никакого логического выигрыша в игре нет, так как отсутствуют выигрышные ситуации (если не учитывать почти невероятный случай, когда игроку удастся уничтожить *все* шарики на поле!). Игрок в любом случае проигрывает, разница только в том, что кто-то сможет продержаться дольше и набрать больше очков. Подобную ситуацию мы наблюдаем и в тетрисе, где также победить нельзя. Но вы можете разбить игру (свой вариант!) на несколько уровней с переходом на следующий после набора определённого количества очков.

Если не учитывать различий в качестве графики, то обе версии игры совершенно одинаковы. Существуют только два технических отличия – шарики по-разному перемещаются по клеткам, и в *ДОС*-версии шарик оставляет за собой следы, что довольно забавно.

Недалеко ушла от оригинала игра *Цветные линии* (Антон Мелёхин, версия 1.1, 1997 год) (Рис. Ф6.3).



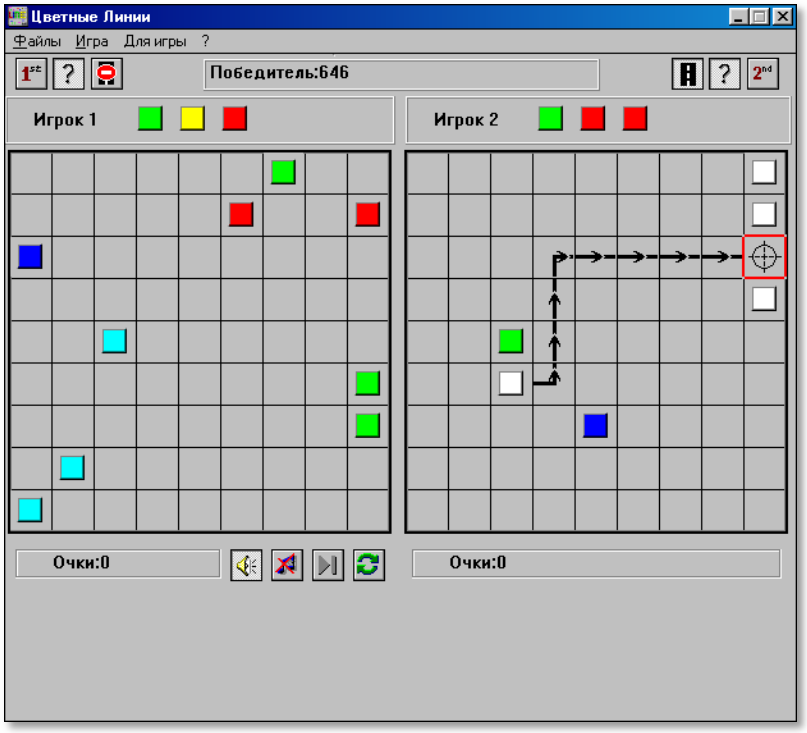


Рис. Ф6.3. Цветные линии Антона Мелёхина

Написана она ещё на первой версии нашего родного *Delphi*. Игра ведётся по классическим правилам, только в начале игры появляются 3 шарика, что явно недостаточно, так как практически невозможно сделать ни одного осмысленного хода. За каждую уничтоженную цепочку начисляется удвоенное количество очков.

Предусмотрена возможность игры вдвоём, но каждый игрок ведёт совершенно самостоятельную партию (один управляет мышкой, второй - клавиатурой), так что никакого резона в таком усовершенствовании игры нет. Чтобы внести элемент соревновательности, следовало бы играть параллельно, но тогда один игрок может просто повторять ходы другого. Правильнее было бы соревноваться с компьютером, то есть второго игрока должна заменить программа.Для этого потребуется наделить её каким бы то ни было интеллектом, чтобы она выбирала хорошие ходы (или лучшие!) из нескольких возможных.

По ходу игры вы можете слушать музыку. Сейчас это стало модным атрибутом игровых программ, хотя совершенно необязательно интеллектуальные игры портить музыкальным сопровождением. Тем более что каждый меломан может запустить, например, в *Универсальном проигрывателе*, любимую музыку в фоновом режиме.

К достоинствам программы можно отнести динамическое выделение предполагаемого пути шарика стрелками. Очень удобно и наглядно!

Две следующие программы - *Lines'98* (Рис. Ф6.4)

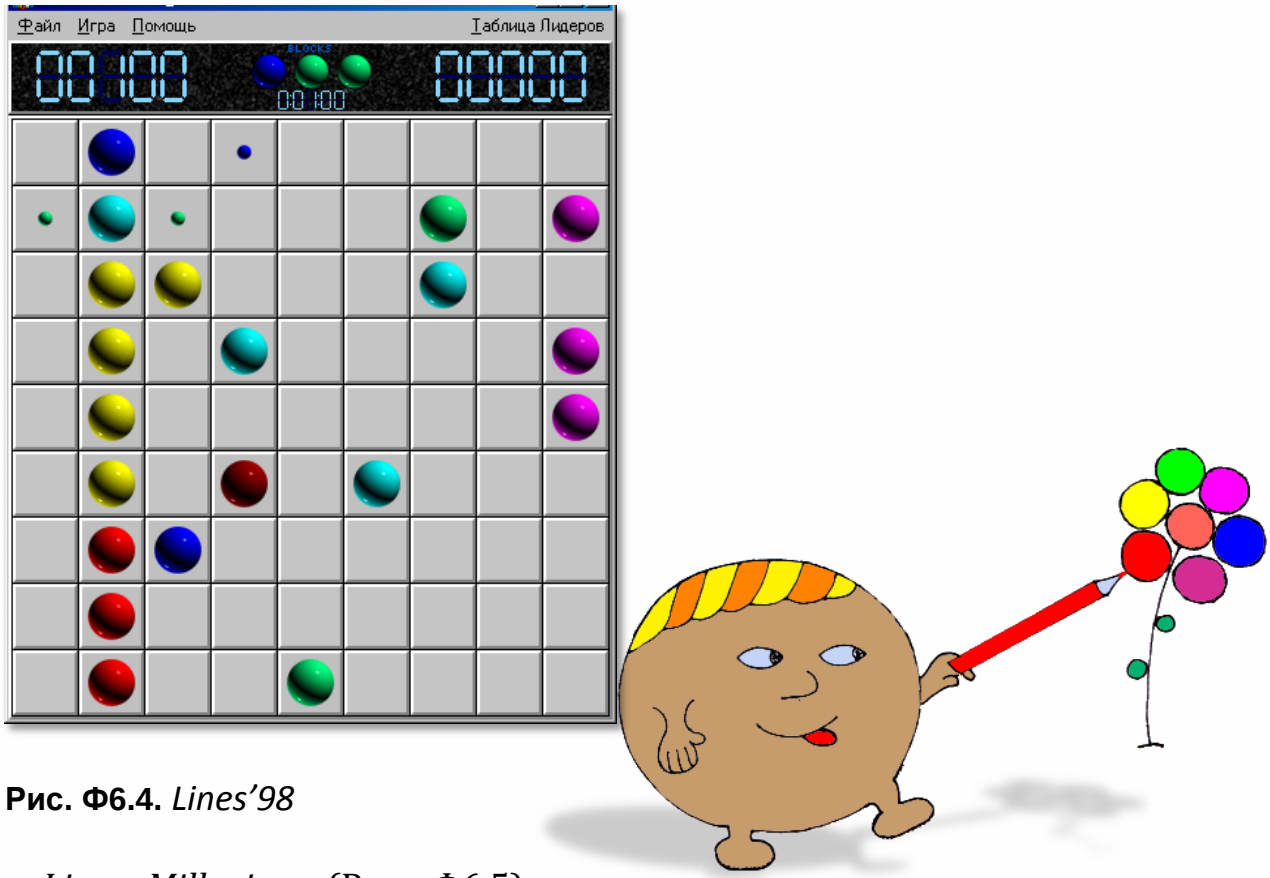


Рис. Ф6.4. *Lines'98*

и *Lines Millenium* (Рис. Ф6.5) – принадлежат «перу» одного автора – Дмитрия Кивилёва. Это всё та же классическая игра, но в новом обличии, точнее – во множестве обличий, так как вы можете изменить её внешний вид, выбирая один из нескольких возможных *скинов* (*skins*) – модная нынче технология «пластической операции» интерфейса.

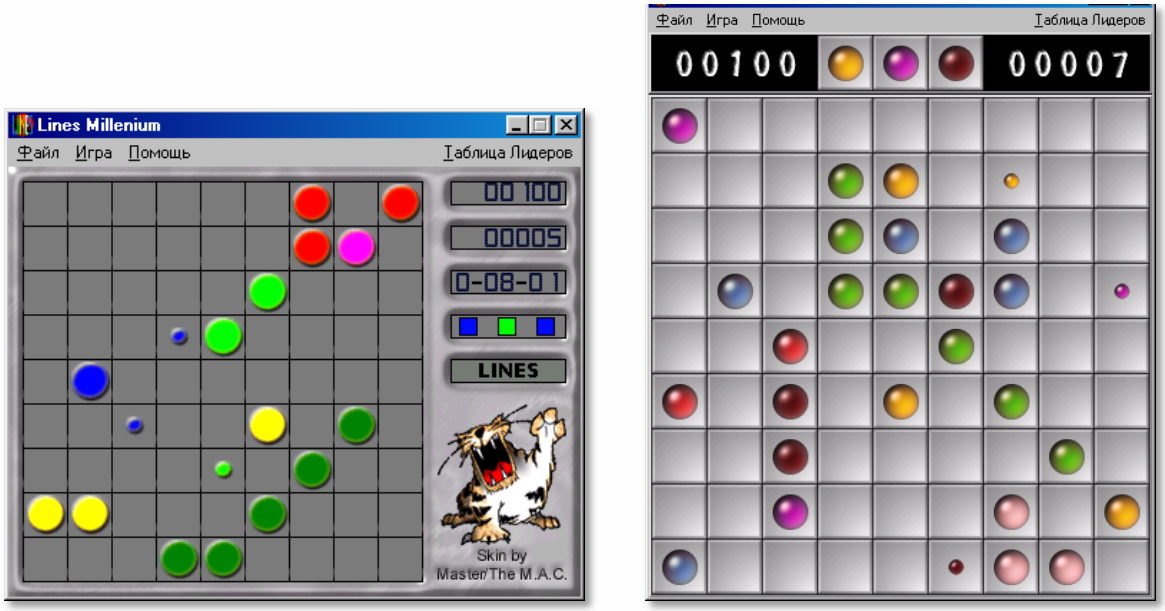


Рис. Ф6.5. *Lines Millenium*

Кроме стандартной игры, в которой требуется уничтожать линии (*Lines*), имеется ещё две её разновидности - *Squares* и *Blocks*. В первой нужно собирать прямоугольники (не менее четырёх шариков), во второй - связные области (не менее семи шариков).

Другие нововведения: прогрессивное начисление очков в зависимости от длины цепочки, выбор и настройка звуковых эффектов для основных событий, сохранение и загрузка игры, откат на один ход, изменение количества вновь появляющихся шариков (3-5), следующие шарики могут отображаться не только на табло, но и на поле (шарики меньшего размера).

В целом игра оставляет весьма приятное впечатление.

Несомненно, порадует вас обилием скинов и игра *Cool Lines* (автор Настя Рассохина, 2000 год) (Рис. Ф6.6), тем более что их легко изготовить самостоятельно.

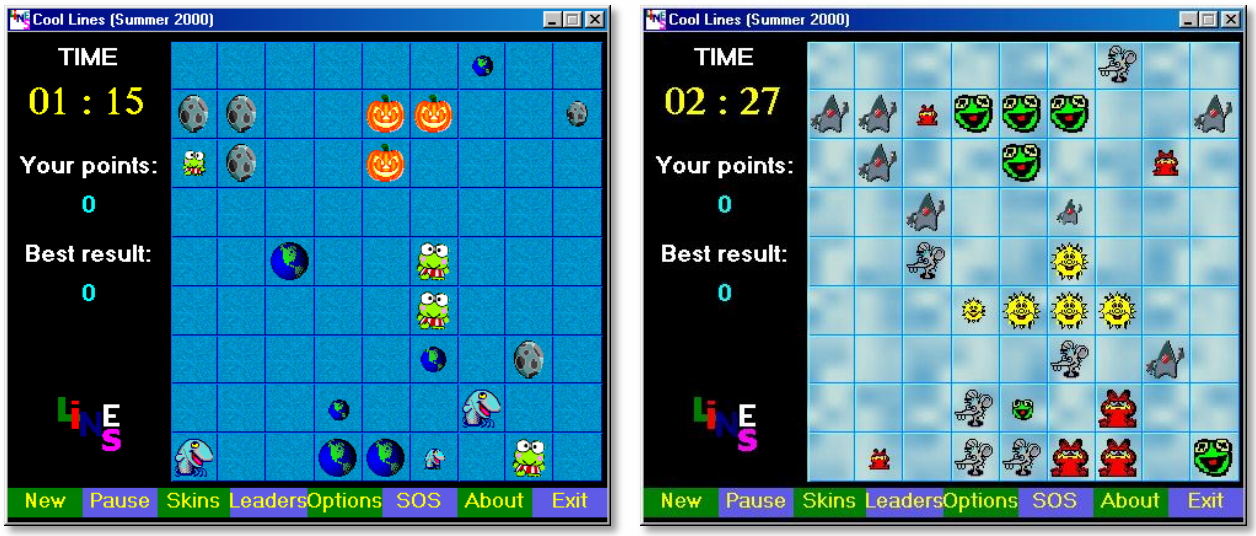


Рис. Ф6.6. *Cool Lines*

В остальном эта игра лишь незначительно отличается от оригинала:

- можно уничтожать не только линии, но и квадраты из четырёх «шариков»;

- отсчитывается время игры;

- количество вновь появляющихся фигурок непостоянно (5-9) в ходе одной игры;

- имеется 3 уровня сложности, которые и определяют набор фигурок.

В игре *Picture Lines 1.0* (автор Вячеслав Витер, 2000 год) тоже поддерживаются скины (Рис. Ф6.7).



Рис. Ф6.7. *Picture Lines 1.0*

Игра проходит на поле стандартных размеров, но имеется возможность выбирать количество вновь появляющихся шариков (3-5) и их цвета (5-9). Уничтожаются не только прямые линии, но и прямоугольники, прямые и косые кресты и блоки (связные области). Выводится время игры, процент занятой шариками территории, путь шарика. Текущую игру можно сохранить на диске, а затем вновь загрузить.

Из новинок можно отметить появляющуюся по мере уничтожения шариков картинку, что, впрочем, ничуть не улучшает игру, так как б***о***льшая часть поля всё равно занята шариками. Было бы лучше вынести проявляющуюся картинку на отдельную панель или форму.

Всё бы ничего, но назойливая реклама над игровым полем не способствует интересу к этой игре, тем более что ничего принципиально нового в ней нет.

В игре *Цветные линии 1.01* (Cheensoft, 1997 год), можно отметить только указатель заполнения поля шариками на компоненте *ProgressBar* (Рис. Ф6.8).



Рис. Ф6.8. *Цветные линии 1.01*

В игре *Alhademic Lines 1.20* (Alhademic Group, 2000 год) (Рис. Ф6.9) вы найдёте красивую графику (выбор из нескольких скинов и наборов фишек), приятную музыку и удобный интерфейс.

Собирать можно не только линии, но также прямоугольники и блоки. Имеется возврат последнего хода, три уровня сложности (6-8 цветов шариков), динамический указатель пути, подсказки на табло и на поле). Если вы не ищете экзотики, то эта игра для вас.



Рис. Ф6.9. *Alhademic Lines*

Игры *Тёмные линии* (*DarkLines 2000 Elite 2.1*) (Рис. Ф6.10) и *DarkLines 3: ERA* (2001 год) (Рис. Ф6.11) имеют игровое поле 10 на 10 клеток, позволяют изменять скины и вид фишек.



Рис. Ф6.10. *Тёмные линии*

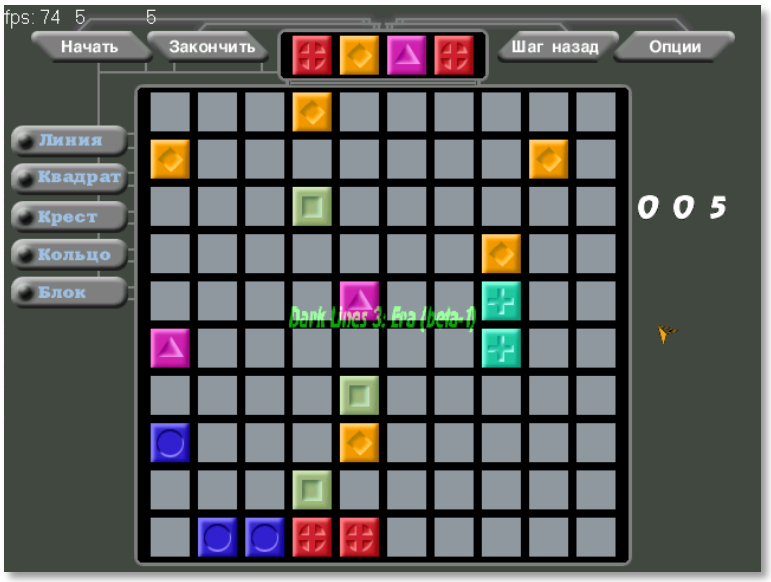


Рис. Ф6.11. *Тёмные DarkLines 3: ERA*

Из дополнительных возможностей можно назвать: откат на 1 ход назад, несколько уровней сложности игры (в усложнённом варианте часть клеток поля на время блокируется – с них и на них нельзя ходить), появляется не 3, а 4 новых фишки, составлять можно не только линии, но и прямоугольники, кольца, кресты, углы или блоки.

5 Star Free Lines 2.1 (2001 год) (Рис. Ф6.12).

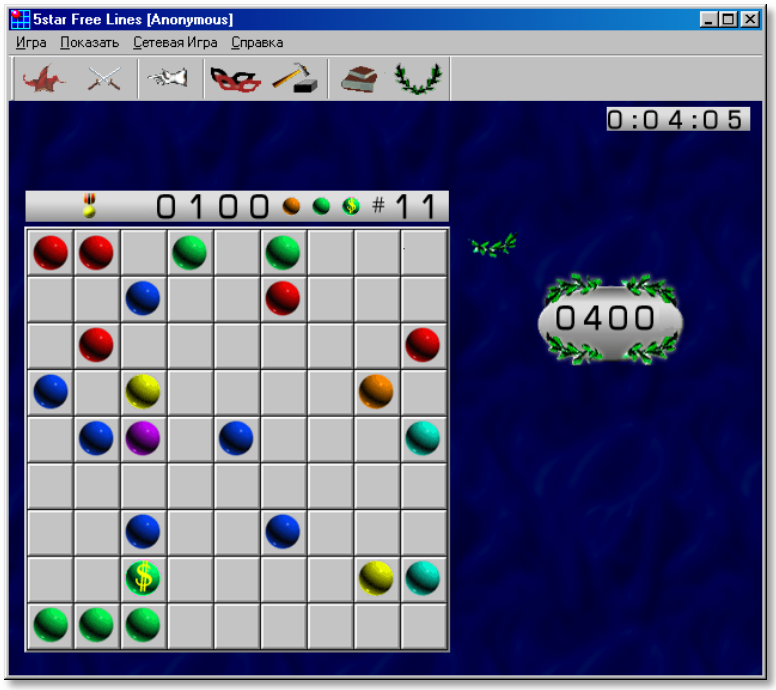


Рис. Ф6.12. *5 Star Free Lines 2.1*

Игра ведётся по классическим правилам, но имеется особый шарик – *джокер*, который может заменить шарик любого цвета. Очки начисляются прогрессивно - в зависимости от длины цепочки.

В игре *Color Lines 99* (Рис. Ф6.13) Максима Иванова вас порадуют: несколько видов оформления поля, интересная заставка и – классические правила, осложнённые хитрым начислением набранных очков.



Рис. Ф6.13. *Color Lines 99*

В игре *Magic Lines* (версия 3.6, 2006 год, А.Сергеев, П.Тарасов) (Рис. Ф6.14) следует в первую очередь отметить интересную графику: вращающиеся объёмные шарики, освещение которых меняется в зависимости от положения курсора мыши. От классического варианта эта игра отличается только тем, что сразу появляются только три шарика, а следующие показываются не только на табло, но и на поле.

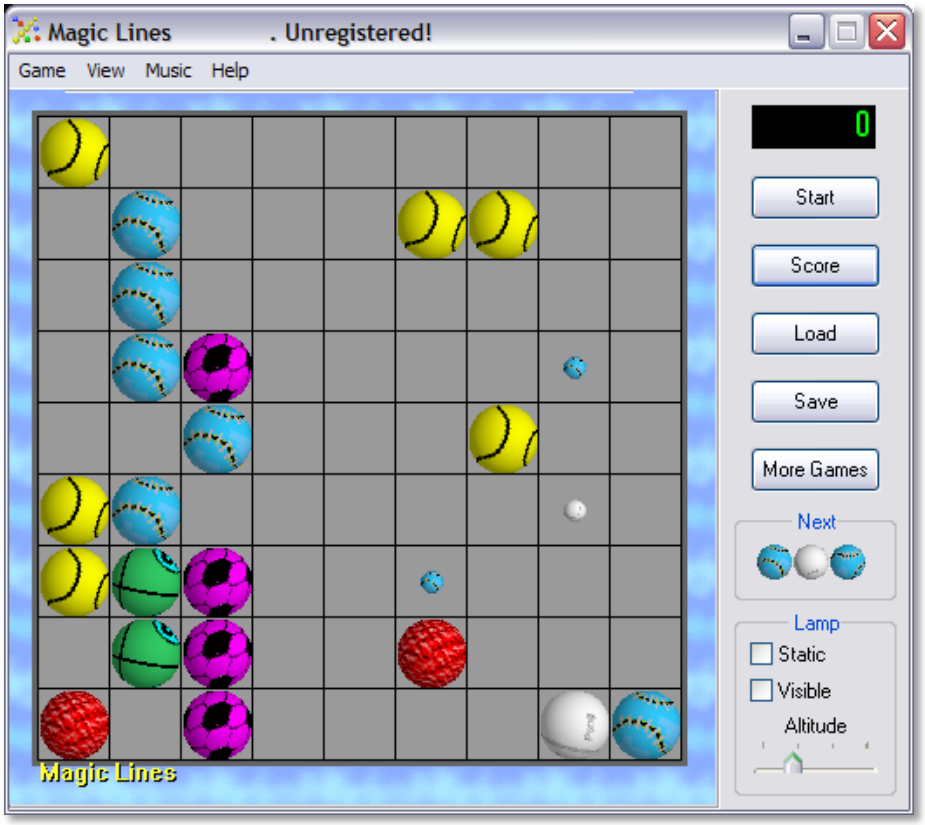


Рис. Ф6.14. *Magic Lines*

3D Lines 1.2 (автор Александр Шелемехов, 2001 год) - настоящая *трёхмерная* игра (Рис. Ф6.15).

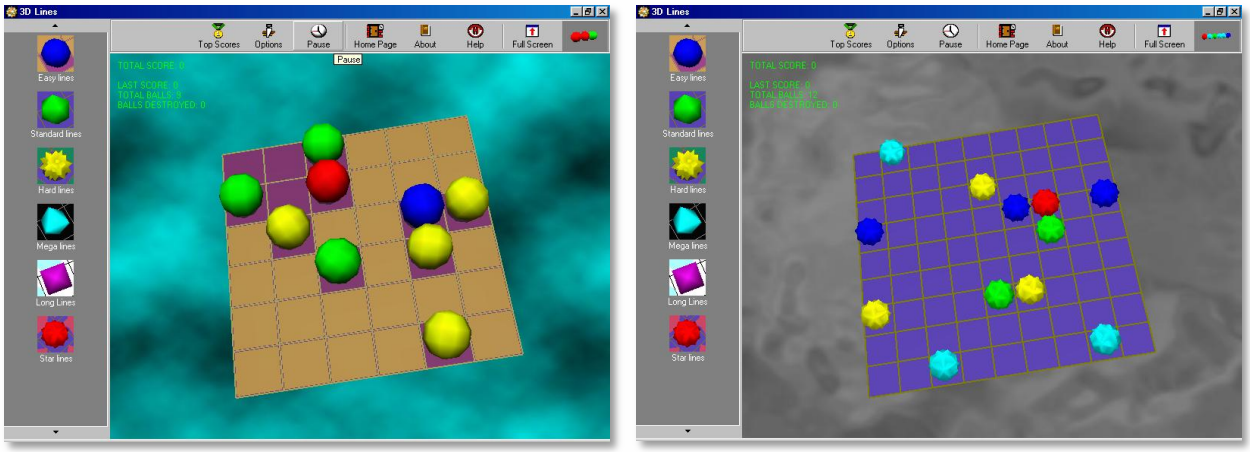


Рис. Ф6.15. *3D Lines 1.2*

В *Honey Lines* (Nikita, 1995 год) игра происходит на *гексагональном* поле (Рис. Ф6.16), что не только радует глаз, но и облегчает задачу игрока, ведь теперь линии можно выстраивать не по четырём, а по *шести* направлениям. Более того, разрешается не только перемещать фишки на свободные клетки, но и менять местами две соседние и даже пропустить очередной ход, хотя пользы от этого не много.



Рис. Ф6.16. *Honey Lines*

Очень любопытный вариант игры, правда, слегка омрачённый плохо подобранной музыкой и звуковыми эффектами. Ну, и ложка дёгтя в бочку мёда: так как задача игрока - не наполнение сотов, а их опустошение, то, скорее, он не пчёлка, а трутень. Но тут как в сказке: добро всегда побеждает зло, а соты всё рано будут наполнены мёдом, сколько бы вы его ни уничтожали!

Нить Ариадны, или Как выбраться из лабиринта

На первый взгляд, в игре *Цветные линии* нет никакого *лабиринта*. А всё-таки он есть!

Вот вы выбрали шарик и указали клетку, на которую он должен перейти. Тут и возникают вопросы: *возможен* ли такой переход и, если возможен, то *как* найти *самый короткий* путь из одной клетки в другую.



В данной игре это необязательное условие, но и гонять шарик по всему полю тоже негуманно; во многих других играх придётся искать именно *кратчайший* путь, так что лучше подготовиться к этому заблаговременно.

Поскольку целый класс задач посвящён выбору правильного пути, то мы рассмотрим эту проблему более подробно. Оставим на время нашу программу и сыграем этюд на тему *лабиринта*.

Итак, откройте новый проект и установите на форме сетку *dgPole* размером 12 x 12 клеток. Для хранения информации о лабиринте заведите массив *masPole:*

```
masPole: array[0..11, 0..11] of integer;
```

Как известно, в лабиринте есть *проходимые* и *непроходимые* клетки (в реальных лабиринтах, разумеется, клеток нет, а вот в головоломных – чаще всего имеются). Обозначим первые константой *GO*, вторые – *STOP*. Значения им можно присвоить любые *отрицательные*, чтобы уметь отличать их от номеров, которые мы будем записывать в клетки:

```
const
  GO= -1;
  STOP= -2;
```

Прежде всего, научимся из пустой сетки делать лабиринт. Вообще-то над хорошим лабиринтом нужно изрядно поработать (вспомните Дедала, знаменитого строителя первого лабиринта на острове Крит), но для нас качество лабиринта никакого значения не имеет, поэтому мы будем просто закрашивать клетки *белым* (проходимые) и *чёрным* (непроходимые) цветом.

Сложность лабиринта определяется, в том числе, количеством непроходимых клеток на поле, поэтому вы можете создавать лабиринты любой сложности, изменяя значение переменной *Dif*. А дальше всё просто:

случайным образом мы расставляем чёрные и белые клетки на поле – и лабиринт готов:

```
//СОЗДАТЬ ЛАБИРИНТ
procedure TForm1.CreateLab;
var
  i, j: integer;
begin

  for j:= 0 to 11 do
    for i:= 0 to 11 do
      if Random(101) >= Dif then
        masPole[i,j]:= GO
      else
        masPole[i,j]:= STOP;
  dgPole.Invalidate;
end;
```

Он будет появляться при старте программы:

```
//СТАРТ ПРОГРАММЫ
procedure TForm1.FormCreate(Sender: TObject);
begin
  Randomize;
  CreateLab;
end;
```

А если он вам не понравится, вы в любой момент можете его заменить другим, нажав кнопку:

```
//НОВЫЙ ЛАБИРИНТ
procedure TForm1.Button1Click(Sender: TObject);
begin
  CreateLab;
end;
```

Для изменения «сложности» лабиринта (по сути, числа препятствий), установим на поле *скроллер sbDif*:

```
//УСТАНОВИТЬ СЛОЖНОСТЬ ЛАБИРИНТА 0..100%
procedure TForm1.sbDifChange(Sender: TObject);
begin
  dif:= sbDif.Position;
  lblDif.Caption:= inttostr(Dif);
  CreateLab;
end;
```

Теперь нужно выделить в сетке *начальную* и *конечную* клетки. Так как это только набросок для будущих программ, то мы не будем тратить много времени на этот процесс. Поступим просто: нажав *левую* кнопку мыши, мы

зададим *начальную* клетку, *правую – конечную*. Под координаты этих клеток отведём *переменные*:

```
BeginCell, EndCell: TPoint;
```

А числа в этих клетках обозначим *константами*

```
  BEG_CELL= 0;
  END_CELL= -3;
```

END_CELL может быть любым *отрицательным* (или большим положительным) числом, а вот *BEG_CELL* непременно должна быть равна *нулю*, так как именно с этой клетки и начнётся поиск короткого пути. Подробности смотрите в описании алгоритма поиска.

```
//ВЫДЕЛИТЬ КЛЕТКИ
procedure TForm1.dgPoleMouseDown(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
var
  ACol,ARow: integer;
begin
  //координаты мыши:
  dgPole.MouseToCell(x,y,ACol,ARow);

  //если нажата левая кнопка мыши, то отмечаем начальную клетку:
  if ssLeft in shift then begin

     //проходимая клетка:
     if ssCtrl in shift then
       masPole[ACol,ARow]:= GO
     else if ssAlt in shift then
       masPole[ACol,ARow]:= STOP
     //непроходимая клетка:
     else begin
       BeginCell:= Point(ACol,ARow);
       masPole[ACol,ARow]:= BEG_CELL
     end

  end
  //отметить конечную точку:
  else begin
    EndCell:= Point(ACol,ARow);
    masPole[ACol,ARow]:= END_CELL;
  end;

  dgPole.Invalidate
end;
```



Не всякий лабиринт придётся вам по вкусу, поэтому у вас должна быть возможность самостоятельно конструировать лабиринт. Сделать это очень просто: щёлкая *левой* кнопкой мыши, удерживайте клавишу *CTRL*,

чтобы поставить *проходимую* клетку, или *ALT*, чтобы запретить проход через данную клетку.

Для запуска процедуры поиска пути мы установим на форме ещё одну кнопку.

Вот так примерно может выглядеть интерфейс этой программы (Рис. Ф6.17, слева).

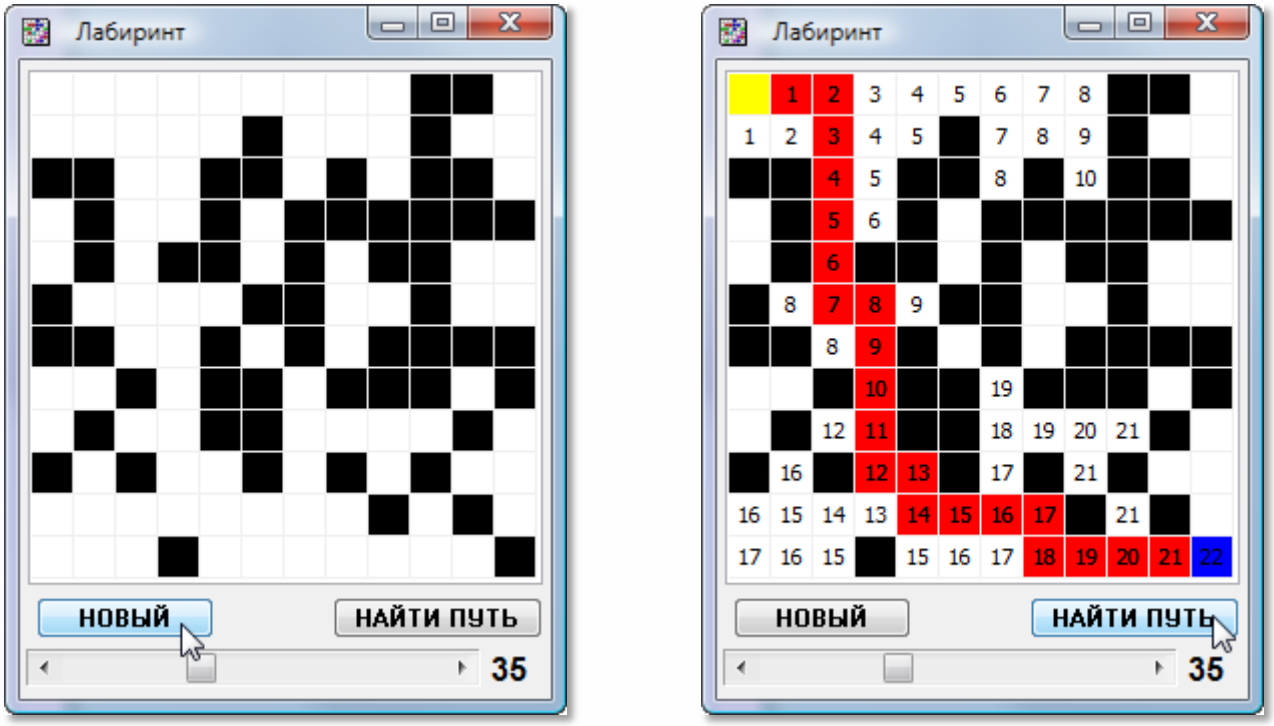


Рис. Ф6.17. Интерфейс программы Путь в лабиринте найден!

Итак, отмечаем клетки и нажимаем кнопку *Найти путь*. Если путь существует, он будет показан на экране (Рис. Ф6.17, справа), в противном случае появится только соответствующая надпись:

```
//ИСКАТЬ ПУТЬ
procedure TForm1.Button2Click(Sender: TObject);
begin
  if FindPath then
    ShowPath
  else
    showmessage('Пути нет')
end;
```

Есть несколько *алгоритмов* поиска пути в лабиринте, мы воспользуемся наиболее эффективным из них, который называется *поиском в ширину* (в

применении к поиску пути в лабиринте он называется также *методом волновой трассировки*, или *волновым алгоритмом*).

Заведём массив *CoordList*, в который будем записывать координаты тех клеток, которые следует рассмотреть. Сначала мы помещаем в него координаты начальной точки *BeginCell*. Переменная *ptrRead* всегда указывает на очередную анализируемую клетку в этом списке. Так как пока в списке только одна клетка, *ptrRead* указывает на неё. На каждом шаге мы извлекаем из списка координаты очередной клетки и проверяем всех её соседей. В лабиринте разрешается двигаться только влево-вправо и вниз-вверх и только по *проходимым* клеткам (в них записано число *GO* или *END_CELL* – для *конечной* клетки). Найдя подходящую клетку, мы дописываем её координаты в конец списка *CoordList*. Индекс для записи хранится в переменной *ptrWrite*. В начале поиска мы присваиваем ей значение *2*, так как это первый свободный индекс в массиве *CoordList*. После того как записаны координаты новой клетки, указатель *ptrWrite* смещается на следующий индекс. Кроме того, в саму клетку мы записываем число, на единицу большее, чем в текущей клетке. Оно, как легко понять, показывает *минимальное* число шагов, необходимое для перехода в эту клетку из начальной. Именно поэтому в начальную клетку мы и записали *0*. Во все соседние проходимые клетки будет записана *единица*. Во все проходимые (и ещё не отмеченные!) клетки, соседствующие с ними, - *двойка*, и так далее.
Этот процесс хорошо виден на следущих двух картинках (Рис. Ф6.18).

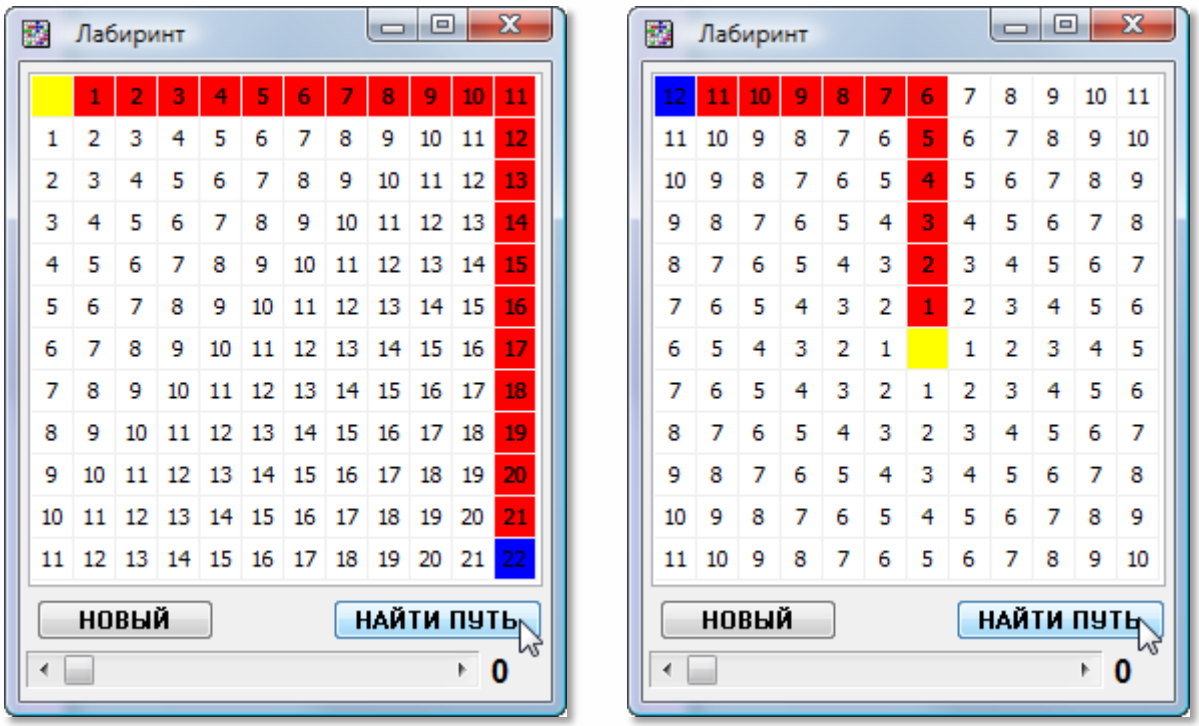


Рис. Ф6.18. Погнали волну по лабиринту!

Внимательно проследите, как программа нумерует клетки! Это полезное занятие приведёт вас к ответу, почему алгоритм – весьма образно! – называется *волновым*.

Поиск заканчивается либо при достижении *конечной* клетки (тогда функция *FindPath* возвращает *True*), либо при исчерпании списка (путь не найден, функция возвращает *False*).

```
//НАЙТИ ПУТЬ МЕЖДУ ДВУМЯ ТОЧКАМИ
function TForm1.FindPath: boolean;
var
  //список координат:
  CoordList: array[1..143] of TPoint;
  //указатели в списке:
  ptrWrite, ptrRead: integer;
  p, q: integer;
  i, j: integer;

  //проверить координаты
  function TestCoord(x,y: integer): boolean;
  begin
    Result:= true;
    if (x<0) or (x>11) or (y<0) or (y> 11)or
     ((masPole[x,y]<> GO) and (masPole[x,y]<> END_CELL)) then
      Result:= false;
  end;

begin
  //если BeginCell = EndCell, то начальная клетка совпадает с конечной,
  //и путь искать не нужно!
  //заносим в список координаты начальной клетки:
  CoordList[1]:= BeginCell;
  //устанавливаем указатель для считывания координат на начало списка:
  ptrRead:= 1;
  //устанавливаем указатель для записи новых координат на следующий ин-
декс:
  ptrWrite:= 2;
  //в начальной клетке в массиве masPole находится BEG_CELL= 0

  //двигаемся от начала списка к его концу, пока он не кончится:
  while ptrRead < ptrWrite do begin
    //координаты текущей клетки:
    p:= CoordList[ptrRead].x; q:= CoordList[ptrRead].y;
    //проверяем соседние с ней клетки:
    for i:= p - 1 to p + 1 do
      for j:= q - 1 to q + 1 do
        //если нашли соседнюю проходимую клетку,
        if ((i=p) or (j=q)) and TestCoord(i,j) then
        begin
          //то записываем в неё число, на единицу большее,
          //чем в текущей клетке:
          masPole[i,j]:= masPole[p,q] + 1;
          //если дошли до конечной клетки,
          if (i= EndCell.x) and (j= EndCell.y) then begin
            //то путь найден:
```

```
            Result:= True;
            exit;
          end
          else begin
            //записываем координаты соседней клетки в конец списка:
            CoordList[ptrWrite]:= Point(i,j);
            //перемещаем указатель:
            inc(ptrWrite);
            dgPole.Invalidate;
            //showmessage(inttostr(masPole[i,j]) + ' x='+inttostr(i)+ '
y='+inttostr(j));
          end;
        end;
      //переходим к следующей клетке в списке:
      inc(ptrRead);
  end;
  //путь не найден:
  Result:= False;
end;
```

За работой алгоритма удобно наблюдать в *пошаговом* режиме. Раскомментируйте эту строку, и вы сможете отслеживать все изменения в сетке:

```
showmessage(inttostr(masPole[i,j]) + ' x='+inttostr(i)+ ' y='+inttostr(j));
```

Легко заметить, что этот же алгоритм можно использовать и для *закрашивания* области поля в какой-либо цвет. Для этого достаточно в качестве начальной клетки (или точки) выбрать любую клетку внутри области, а координаты конечной точки задать *отрицательными*. В этом случае путь найти невозможно, поэтому вся область будет заполнена числами или закрашена выбранным цветом.

Все изменения в массиве *masPole* нужно отразить на экране:

```
//ОТРИСОВАТЬ СЕТКУ
procedure TForm1.dgPoleDrawCell(Sender: TObject; ACol, ARow: Inte-
ger;
  Rect: TRect; State: TGridDrawState);
var
  s: string;
begin
  //закрасить клетку своим цветом:
  case masPole[ACol, ARow] of
    GO: dgPole.Canvas.Brush.Color:= clWhite;
    STOP: dgPole.Canvas.Brush.Color:= clBlack;
    BEG_CELL: dgPole.Canvas.Brush.Color:= clYellow;
    END_CELL: dgPole.Canvas.Brush.Color:= clBlue;
    //номер хода:
    else begin
       with Rect, dgPole.Canvas do begin
```

```
        Brush.Style:= bsClear;
        s:= inttostr(masPole[ACol, ARow]);
        textrect(Rect, left+(right-left-textwidth(s)) div 2,
             top+(bottom-top-textheight(s)) div 2, s);
      end;
    end;
  end;
  dgPole.Canvas.FillRect(Rect);
end;
```

Теперь мы знаем, существует ли путь между двумя точками или нет, но как его найти? А очень просто! Нужно двигаться от *конечной* точки к *начальной* по тем числам, которые мы записали в клетки. Очевидно, что в конечной клетке находится самое большое число, равное длине всего пути. Попасть в неё можно только из клетки с числом на единицу меньше. И так далее, пока мы не достигнем точки с нулём, которая служит *начальной*. По пути выделяем на экране все клетки пути *красным* цветом.

```
//ПОКАЗАТЬ  ПУТЬ
procedure TForm1.ShowPath;
var
  n, LenPath: integer;
  i, j, p, q: integer;
  path: array[0..144] of TPoint;
  Rect: TRECT;
  s: string;

  //проверить координаты:
  function TestCoord(x,y: integer): boolean;
  begin
    Result:= true;
    if (x<0) or (x>11) or (y<0) or (y> 11)or (masPole[x,y]<> n-1) then
      Result:= false;
  end;

begin
  //длина пути равна числу в конечной клетке:
  LenPath:= masPole[EndCell.x, EndCell.y];
  n:= LenPath;
  //конечная клетка пути:
  path[n]:= EndCell;
  //двигаемся от неё к начальной клетке:
  repeat
    //найти соседнюю клетку с числом n-1:
    p:= path[n].x;  q:= path[n].y;
    //проверяем соседние клетки:
    for i:= p - 1 to p + 1 do
      for j:= q - 1 to q + 1 do
        //нашли подходящую клетку:
        if ((i=p) or (j=q)) and TestCoord(i,j) then
        begin
          //записываем её координаты:
          path[n-1]:= Point(i,j);
          break;
```

```
        end;
    //ищем клетку с предыдущим номером:
    dec (n);
  until n<0;
  //показать путь в сетке:
  for i:= 1 to LenPath-1 do begin
    {ListBox1.Items.Add(inttostr(i)+ ' ' + inttostr(path[i].x)+ ' '+
                                  inttostr(path[i].y));}
    Rect:= dgPole.CellRect(path[i].x,path[i].y);
    //выделить красным цветом:
    dgPole.Canvas.Brush.Color:= clRed;
    dgPole.Canvas.FillRect(Rect);
     with Rect, dgPole.Canvas do begin
        s:= inttostr(i);
        textrect(Rect, left+(right-left-textwidth(s)) div 2,
             top+(bottom-top-textheight(s)) div 2, s);
     end;
  end;
end;
```

Действуя таким образом, мы найдём только *один* из возможных (и самых коротких!) путей. Нам пока большего и не надо, но, если вам придётся проверять *единственность* решения какой-либо задачи, то имейте это в виду!



Исходный код программы находится в папке **lab**.

Большая стройка, или «Линейное» программирование

Подковавшись практически и теоретически, мы вполне созрели до собственноручного построения программы!

Не забудьте установить компонент с нашей кнопкой! Впрочем, вы легко можете избавиться от нововведений, если замените кнопки *TEasyButton* стандартными. Для этого удалите новые кнопки, установите те кнопки, которые вам нравятся, и настройте их свойства и обработчики событий.

Начнём с *формы*. Она будет простой, прямоугольной, но без строки заголовка (*BorderStyle=bsNone*), которая в данном случае только портила бы вид окна приложения. Заголовок мы заменим картинкой в компоненте *Image1* (картинки находятся в папке *Res/button*). Перемещать окно приложения придётся также за картинку:

```
  private
```

```
    { Private declarations }
    rgn: HRGN;
    procedure WMNCHitTest( var Msg: TWMNCHitTest); message
WM_NCHITTEST;
```

```
//ПЕРЕМЕЩЕНИЕ ОКНА
procedure TForm1.WMNCHitTest( var Msg: TWMNCHitTest);
begin
  Inherited;
  WITH Msg DO
    WITH ScreenToClient(Point(XPos,YPos)) DO
      if PtInRegion(rgn,X, Y) then
         Result:= HTCAPTION
      else Result:= HTNOWHERE
end;
```

Когда мы нажимаем кнопку мыши на картинке, эта процедура указывает приложению, что мышка будто бы находится на заголовке окна. Но сначала мы должны сформировать *область* по форме картинки *Image1*. Это удобно сделать при запуске приложения:

```
////////////////////////////////////////////////////////
////////////////////// ФОРМА ///////////////////////////
////////////////////////////////////////////////////////

//СОЗДАТЬ ФОРМУ
procedure TForm1.FormCreate(Sender: TObject);
begin
  //цвет фона:
  fon:= rgb(120,144,120);
  Color:= fon;
  Panel1.Color:= fon;
  Panel2.Color:= fon;
  dgNextBalls.Color:= fon;
  //создать поле:
  Pole:= tPole.Create;
  //очки за игру:
  Score:= tScore.Create;

  //сформировать "заголовок окна":
  with image1 do
    rgn:= CreateRectRgn(Left,Top,Left+Width,top+ Height);

  //размеры поля по умолчанию:
  Pole.Width:= 9;
  Pole.Height:= 9;
  //шарики:
  NumBalls.nSrazu:= 5;
  NumBalls.nPotom:= 3;
  NumBalls.nColor:= 7;

  TypeFig:=tfLines;

  //размеры клетки сетки:
  wCell:= dgPole.DefaultColWidth;
  hCell:= dgPole.DefaultRowHeight;
end;
```

Здесь же мы окрашиваем окно приложения и компоненты в какой-нибудь забавный цвет и производим настройки игры, речь о которых пойдёт ниже. Непосредственная подготовка к новой игре будет происходить при активации формы - так мы сможем уже после старта программы приступить к игре, не нажимая никаких кнопок:

```
//ПОДГОТОВИТЬСЯ К ИГРЕ
procedure TForm1.FormActivate(Sender: TObject);
begin
  Prepare;
  delay(1000);
  Pole.StartGame;
end;
```

Конечно, при этом игра стартует со свойствами по умолчанию:

- размеры поля 9 x 9 клеток;

- сразу появятся 5 шариков, а дальше каждый раз будут появляться 3 шарика;

- всего в игре используются шарики 7 разных цветов;

- собирать из шариков нужно линии.

Все эти настройки вы можете видеть в процедуре *FormCreate*.

Чтобы края формы получились более красивыми установите свойство формы *Ctrl3D* в *False*, а в процедуре-обработчике формы *FormPaint* напишите:

```
//ОТРИСОВАТЬ ФОРМУ
procedure TForm1.FormPaint (Sender: TObject);
var
  RECT: TRect;
begin
  //придать объёмный вид краям формы:
  RECT:= ClientRect;
  DrawEdge(Canvas.Handle, Rect, EDGE_RAISED, BF_RECT)
end;
```

Ну и чтобы закончить с формой, забежим немного вперёд и напишем обработчик события *OnFormDestroy*, которое возникает при закрытии приложения:

```
//УНИЧТОЖИТЬ ФОРМУ
procedure TForm1.FormDestroy(Sender: TObject);
begin
 Pole.Free;
```

```
 //DeleteObject(rgn);
 Timer1.Enabled:= False;
 Timer2.Enabled:= False;
 Score.Free;
 Tonus.Free;
 hod.Free;
end;
```

Сущность этих «мероприятий» станет понятна вам немного позже.

Теперь мы украсим форму *кнопками TEasyButton* (или другого типа, если вы передумали). Достаточно шести штук:

 **Name**=btnNewGame **Hint**=Новая игра **ButtonBorde**r= brNone

 **Name**=btnWH **Hint**=Размеры поля

 **Name**=btnOptions **Hint**=Настройки

 **Name**=btnTypeFig **Hint**=Вид собираемых фигур

 **Name**=btnHelp **Hint**=Справка

 **Name**= btnExit **Hint**=Выход из программы

Размеры всех кнопок 40 х 40 пикселей. Картинки для них находятся в папке *Res/button*.

Назначение всех кнопок ясно следует из подсказок.

С некоторыми из них мы можем «разобраться» уже сейчас. Что может быть проще, чем закрыть программу? – Нажимаем кнопку *btnExit* – и всё:

```
//ЗАКРЫТЬ ПРОГРАММУ
procedure TForm1.btnExitClick(Sender: TObject);
begin
  close
end;
```

Как вы помните, при этом возникает событие *OnFormDestroy* и вызывается обработчик этого события.

Немного сложнее вывести справку по программе – кнопка *btnHelp*:

```
//ВЫЗОВ СПРАВКИ
procedure TForm1.btnHelpClick(Sender: TObject);
var s: string;
begin
  s:= extractfilepath(application.exename)+'LinesHelp.htm';
  ShellExecute(handle, 'open','iexplore',pChar(s),
  '', SW_SHOWMAXIMIZED);
end;
```

В самой процедуре, конечно, ничего нового для нас нет, но придётся написать файл справки. Впрочем, в начале главы для этого имеется вся необходимая информация!

Перейдём к следующей кнопке – *btnTypeFig*. В классическом варианте из шариков следует строить *линии*, но мы будем собирать и *блоки*, под которыми мы будем понимать, любые фигуры (в том числе и линии, за исключением «диагональных»), имеющие «площадь» не менее 6 шариков:

```
  //тип собираемых фигурок:
  TypeFig:(tfLines, tfBlocks);
```

Выбирать тип собираемых фигур можно и до, и во время игры, нажав кнопку *btnTypeFig*. Под кнопкой появится меню - дальше следует действовать, как обычно (Рис. Ф6.19).

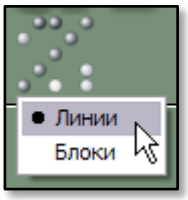


Рис. Ф6.19. Выбираем тип сборки!

```
//ВЫБРАТЬ ТИП СОБИРАЕМЫХ ФИГУР
procedure TForm1.btnTypeFigClick(Sender: TObject);
var
  p: Tpoint;
begin
  //вывести всплывающее меню:
  p:= ClientToScreen(point(btnTypeFig.Left, btnTypeFig.Top+
btnTypeFig.Height));
  popupmenu2.Popup(p.x, p.y);
  //сбросить рекорд:
  score.Rec:= 0;
end;
```

Установим на форме всплывающее меню *PopupMenu2* и наберём две строки (Рис. Ф6.20).

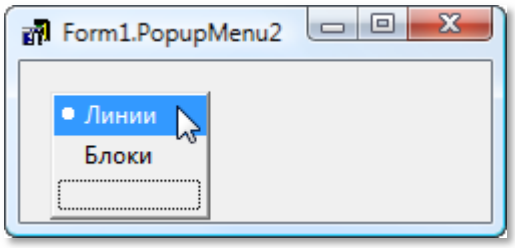


Рис. Ф6.20. Выбираем тип сборки!

По умолчанию играется *классический* вариант - с *линиями* (*первая* строка, **Name**=*miLines*, **tag**=*1*). После выбора пункта меню программа направляет нас в обработчик события *OnmiLinesClick*:

```
procedure TForm1.miLinesClick(Sender: TObject);
begin
  //показать пункт меню как выбранный:
  (Sender as TMenuItem).Checked:= True;
  if (Sender as TMenuItem).tag=1 then TypeFig:= tfLines
  else TypeFig:= tfBlocks;
  //начать новую игру:
  //NewGame;
end;
```

Если вы хотите, чтобы игра велась без смены фигур, то раскомментируйте строку *//NewGame;*.
Другие настройки игры вызываются кнопкой *btnOptions*. Их довольно много, поэтому мы добавим к проекту ещё одну *форму* – *frmOptions*, на которой и разместим все элементы управления: *табулятор TabControl1* и *кнопку ГОТОВО* (Рис. Ф6.21).

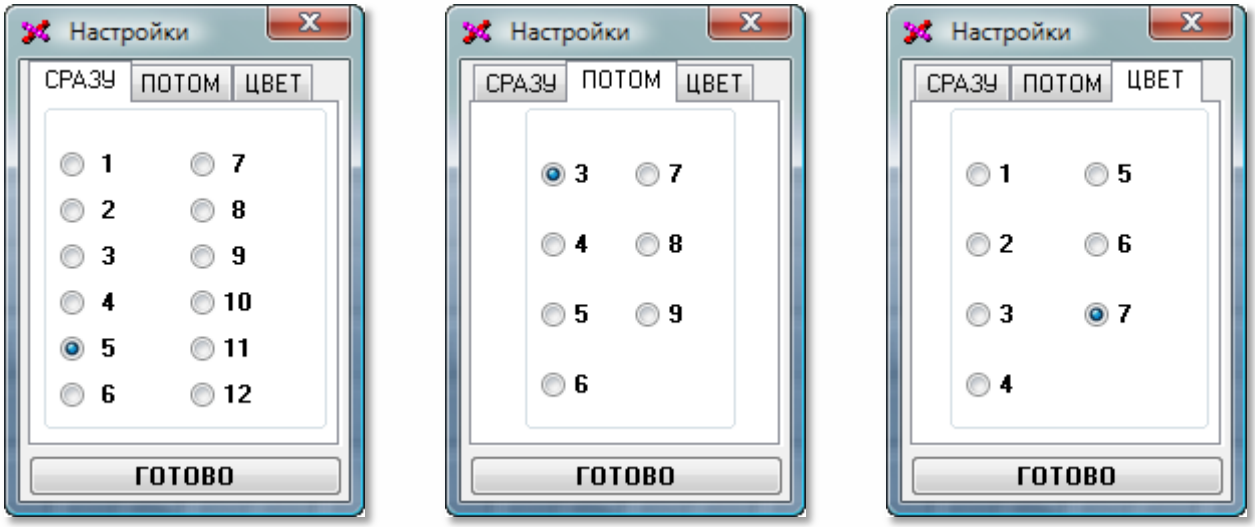


Рис. Ф6.21. Настройки – по настроению!

Форма будет появляться под кнопкой после нажатия на неё:

```
//НАСТРОЙКА ПАРАМЕТРОВ ИГРЫ
procedure TForm1.btnOptionsClick(Sender: TObject);
begin
  //вывести форму для настройки параметров игры:
  frmOptions.Left:=left+btnWH.Left+btnWH.width;
  frmOptions.Top:= top+btnWH.Top+ btnWH.Height;

  frmOptions.ShowModal;

  //установить новые параметры -

  //число шариков, появляющихся при старте новой игры:
  NumBalls.nSrazu:= frmOptions.RadioGroup1.ItemIndex+ 1;
  //число шариков, появляющихся после очередного хода:
  if NumBalls.nPotom<> frmOptions.RadioGroup2.ItemIndex+ 3 then
begin
    NumBalls.nPotom:= frmOptions.RadioGroup2.ItemIndex+ 3;
    //обновить шарики в табло:
    Pole.InitNextBalls;
    Pole.ShowNextBalls;
    dgNextBalls.Invalidate;
  end;

  //число шариков разного цвета:
  NumBalls.nColor:= frmOptions.RadioGroup3.ItemIndex+ 1;

  //так как параметры изменились, то обнулить рекорд:
  score.Rec:= 0;
end;
```

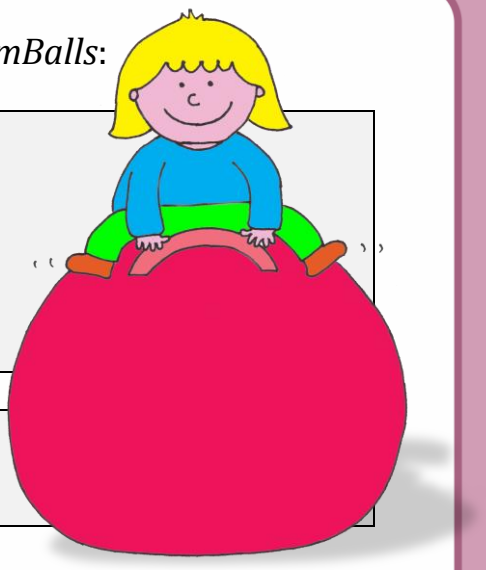
Табулятор имеет 3 страницы с компонентом *TRadioGroup* на каждой. Их нужно заполнить данными так, как показано выше.

Таким образом, по умолчанию сразу после начала игры появляются 5 шариков, но потом мы можем задать от 1 до 12. Чем больше шариков, тем энергичнее начинается игра!

После каждого хода на поле возникают ещё 3 шарика – по умолчанию, как в классичеком варианте. Если их количество уменьшить, то игра будет протекать очень вяло, но зато можно добавлять и *больше* шариков – до 9 за один раз. Ясно, что большое количество шариков сильно усложняет игру, и наоборот.

По умолчанию на поле появляются шарики 7 разных цветов. Ещё более увеличивать «пестроту» на поле смысла нет, поэтому число цветов можно только уменьшить – до одного. Так у игрока появятся новые возможности по составлению фигур из шариков!

Объединим всю информацию о шариках в *записи TNumBalls*:



```
type
  //количество шариков:
  TNumBalls= Record
    nSrazu: integer;
    nPotom: integer;
    nColor: integer;
  end;
```

```
var
  //число шариков в игре:
  NumBalls: TNumBalls;
```

Например, переменная *NumBalls.nSrazu* хранит число шариков, которые *сразу* появляются на поле.

При запуске приложения в процедуре *TForm1.FormCreate* как раз и устанавливаются значения по умолчанию:

```
  //шарики:
  NumBalls.nSrazu:= 5;
  NumBalls.nPotom:= 3;
  NumBalls.nColor:= 7;
```

Текст модуля *OptionsUnit:*

```
unit OptionsUnit;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
Dialogs,
  StdCtrls, ExtCtrls, ComCtrls;

type
  TfrmOptions = class(TForm)
    TabControl1: TTabControl;
    RadioGroup1: TRadioGroup;
    RadioGroup2: TRadioGroup;
    RadioGroup3: TRadioGroup;
    Button1: TButton;
    procedure TabControl1Change(Sender: TObject);
    procedure RadioGroup3Click(Sender: TObject);
    procedure Button1Click(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
```

```
  end;

var
  frmOptions: TfrmOptions;

implementation

{$R *.DFM}

procedure TfrmOptions.TabControl1Change(Sender: TObject);
begin
  case TabControl1.TabIndex of
    0: begin
         RadioGroup1.Visible:= true;
         RadioGroup2.Visible:= False;
         RadioGroup3.Visible:= False;
       end;
    1: begin
         RadioGroup1.Visible:= False;
         RadioGroup2.Visible:= true;
         RadioGroup3.Visible:= False;
       end;
    2: begin
         RadioGroup1.Visible:= False;
         RadioGroup2.Visible:= False;
         RadioGroup3.Visible:= true;
       end;
  end;
end;

procedure TfrmOptions.RadioGroup3Click(Sender: TObject);
begin
  close
end;

procedure TfrmOptions.Button1Click(Sender: TObject);
begin
  close
end;

end.
```

Тут единственная «хитрость» состоит в том, чтобы показывать нужный набор переключателей, а остальные скрывать, чтобы они не мешали. Конечно, можно просто увеличить размеры формы *frmOptions* и разместить все переключатели (или движки, скроллеры и т.п.) так, чтобы они не перекрывали друг друга.

Ну и последнее, что можно изменять в игре, – это *размеры* игрового поля. Если вы помните, в классическом варианте размер поля постоянен – 9 x 9 клеток. Мы раздвинем границы – пусть в нашей программе размеры поля

изменяются от 6 x 6 до 12 x 12 клеток. После нажатия на кнопку *btnWH* под ней появится меню, в котором можно выбрать нужные размеры игрового поля (Рис. Ф6.22).

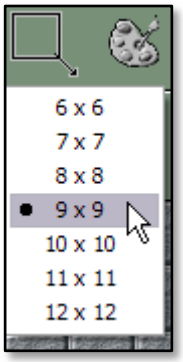


Рис. Ф6.22. Универсальное поле

Код процедуры аналогичен коду процедуры *TForm1.btnTypeFigClick*:

```
//ИЗМЕНИТЬ РАЗМЕРЫ ПОЛЯ
procedure TForm1.btnWHClick(Sender: TObject);
var
  p: Tpoint;
begin
  //вывести всплывающее меню:
  p:= ClientToScreen(point(btnWH.Left, btnWH.Top+ btnWH.Height));
  popupmenu1.Popup(p.x, p.y);
  //сбросить рекорд:
  score.Rec:= 0;
end;
```

```
//ИЗМЕНИТЬ РАЗМЕРЫ ПОЛЯ В МЕНЮ
procedure TForm1.wh6Click(Sender: TObject);
var
  n: integer;
begin
  //показать пункт меню как выбранный:
  (Sender as TMenuItem).Checked:= True;
  //размеры поля:
  n:= (Sender as TMenuItem).Tag;
  //размеры поля:
  Pole.Width:= n;
  Pole.Height:= n;
  //изменить сетку:
  Prepare;
  dgPole.Invalidate;
  application.ProcessMessages;

  //начать новую игру:
  NewGame;
```

```
end;
```

Поэтому устанавливаем на форме компонент *TPopupMenu* и набираем строки (Рис. Ф6.23).

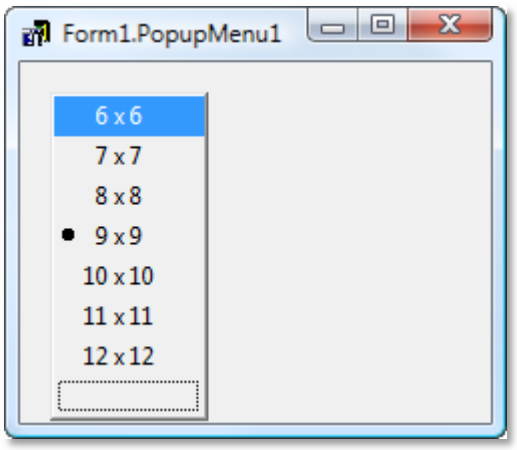


Рис. Ф6.23. Выбираем любимый размер

Затем задаём значения свойства **tag**=*6, 7,...,12.*

И последняя кнопка - *btnNewGame* - просто начинает новую игру по первому нашему требованию:

```
//НАЧАТЬ НОВУЮ ИГРУ
procedure TForm1.btnNewGameClick(Sender: TObject);
begin
  NewGame
end;
```

```
//НОВАЯ ИГРА
procedure TForm1.NewGame;
begin
  //подготовить сетку:
  Prepare;
  //начать игру:
  Pole.StartGame;
end;
```

Теперь мы умеем делать всё, кроме главного, - у нас нет игрового поля! Поэтому на компоненте *Panel1* мы установим сетку *dgPole*, размеры клеток которой - 32 x 32 пикселя:

```
 Var
 //размеры клеток сетки:
  wCell: integer = 32;
  hCell: integer = 32;
```

Этого вполне достаточно, чтобы картинки в них хорошо смотрелись на экране.

При старте каждой новой игры вызывается *процедура TForm1.Prepare*, в которой вычисляются размеры поля (то есть компонента *dgPole*) и оно устанавливается в центре панели:

```
//ПОДГОТОВИТЬ ИГРОВОЕ ПОЛЕ
procedure TForm1.Prepare;
var
  w,h, lw: integer;
begin
  //размер клетки в пикселях:
  w:= wCell;
  h:= wCell;
  //толщина линий:
  lw:= dgPole.GridLineWidth;
  //размеры игрового поля в клетках:
  dgPole.ColCount:= Pole.Width;
  dgPole.RowCount:= Pole.Height;
  //размеры в пикселях видимой части игрового поля:
  dgPole.Width:= 3 + (w + lw)* dgPole.ColCount-1;
  dgPole.Height:= 3 + (h + lw)* dgPole.RowCount-1;
  //отцентрировать поле:
  dgPole.Left:= panel1.Left+ (panel1.Width-dgPole.Width) div 2+1;
  dgPole.Top:= panel1.Top+ (panel1.Height -dgPole.Height) div 2+1;
end;
```

Так выглядит интерфейс программы в окне *Конструктора формы* (Рис. Ф6.24).

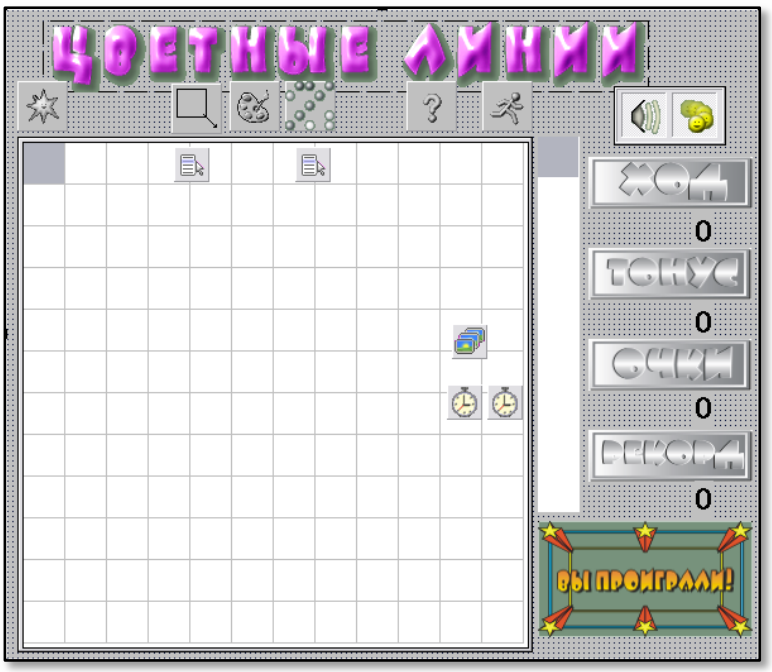


Рис. Ф6.24. Все в сборе!

А так – после запуска приложения (Рис. Ф6.25).

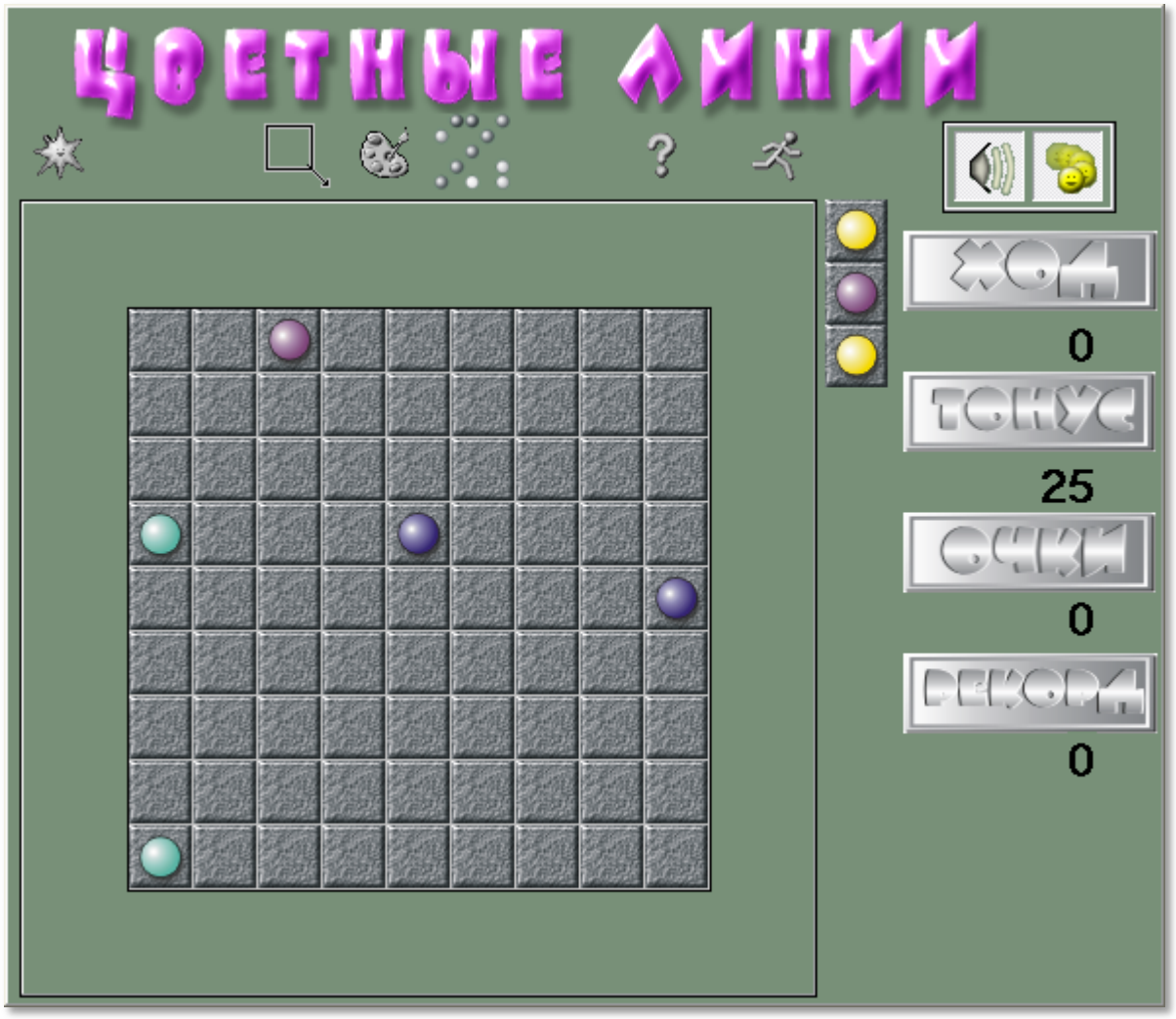


Рис. Ф6.25. Попрошу к шарам!

Как видите, есть несколько компонентов, ещё не охваченных нашим вниманием.

Но сейчас у нас есть дела поважнее – мы напишем класс *поля TPole*, для которого добавим к проекту новый *модуль* – *PoleUnit*. Это разумно, так как код достаточно длинный и лучше отделить его от основного текста программы. Тем более что этот модуль с небольшими изменениями можно будет использовать и в других проектах.

Введём *константы*:

```
const
  //макс. размеры поля:
  MAX_WIDTH=12;
  MAX_HEIGHT=12;
  //константы лабиринта:
```

```
  GO= -1;       // - проходимая клетка
  STOP= -2;     // - непроходимая клетка
  BEG_CELL= 0; // - начальная клетка пути
```

И новые *типы* данных:

```
type
  //цвет шариков:
  TBallColor= (bcNone, bcRed, bcGreen, bcWater, bcBrown, bcBlue,
bcLila, bcYellow);
  //цвет клетки поля:
  TCellColor= (ccGray, ccGreen, ccRed);
  //клетка поля:
  TCell= Record
    //цвет шарика в клетке:
    BallColor: TBallColor;
    //цвет клетки:
    CellColor: TCellColor;
    //новый шарик:
    New: boolean;
  end;
  //массив поля:
  TField= array[0..MAX_WIDTH, 0..MAX_HEIGHT] of integer;
```

Здесь всё настолько прозрачно, что любые комментарии излишни!

Сам класс поля мы разберём подробно:

```
  //Поле:
  TPole= class
    //ширина поля в клетках:
    Width: integer;
    //высота поля в клетках:
    Height: integer;
    //координаты выделенного шарика:
    SelBall: TPoint;
    //массив клеток:
    Cell: array[0..MAX_WIDTH, 0..MAX_HEIGHT] of TCell;
    //массив для поиска пути:
    masPath: TField;
    Path: array[0..MAX_WIDTH*MAX_HEIGHT] of TPoint;
    //следующие шарики:
    NextBalls: array[1..12] of TBallColor;

    //создать поле:
    constructor Create;
    //инициализация поля:
    procedure Init;
    //очистить поле:
    procedure Clear;
    //щелчок на поле:
```

```
    procedure Click(X, Y: integer);
    //инициализация первых шариков:
    procedure InitFirstBalls;
    //снять выделение с шарика:
    procedure DeActivateBall;
    procedure Animation;
    //начать новую игру:
    procedure StartGame;
    //появление шариков на поле:
    procedure CreateBalls;
    //загадать следующие шарики:
    procedure InitNextBalls;
    //показать следующие шарики в табло:
    procedure ShowNextBalls;

  protected

  private
    //минимальная длина цепочки:
    LenLines: integer;
    //минимальный размер блока:
    LenBlocks: integer;
    //задержка при уничтожении цепочки:
    DelayDestroy: integer;
    //задержка при возникновении шарика:
    DelayCreate: integer;
    //задержка при движении шарика:
    DelayMove: integer;

    //переместить шарик:
    procedure DoHod(X, Y: Integer);
    procedure DoMoveBall(X, Y: Integer);
    //выделить шарик:
    procedure ActivateBall(X, Y: Integer);
    //найти путь:
    function FindPath(EndCell: TPoint): boolean;
    //проложить путь:
    procedure CreatePath(EndCell: TPoint);
    //уничтожить фигуру:
    function DeleteFigs: integer;
    //подпрыгивающий шарик:
    procedure JumpBall;
    //игра закончена:
    procedure GameOver;
    procedure GameWin;
  end;
```

Прежде всего, нужно создать поле; это делается при старте программы в процедуре *TForm1.FormCreate*:

```
  //создать поле:
  Pole:= tPole.Create;
```

При создании экземпляра класса мы задаём параметры игры по *умолчанию*:

```
////////////////////////////////////////////////////////
////////////////////// ПОЛЕ ////////////////////////////
////////////////////////////////////////////////////////

constructor tPole.Create;
begin
  inherited create;
  //установить значения по умолчанию:
  LenLines:= 5;
  LenBlocks:= 6;
  DelayDestroy:= 150;
  DelayCreate:= 80;
  DelayMove:= 40;
end;
```

А именно: необходимая длина линий – 5 шариков, минимальное число шариков в блоке – 6; остальные переменные используются при анимации. После создания поля в процедуре *TForm1.FormActivate* запускается новая игра:

```
//ПОДГОТОВИТЬСЯ К ИГРЕ
procedure TForm1.FormActivate(Sender: TObject);
begin
  Prepare;
  delay(1000);
  Pole.StartGame;
end;
```

То же самое происходит и при нажатии на кнопку *Новая игра*:

```
//НОВАЯ ИГРА
procedure TForm1.NewGame;
begin
  //подготовить сетку:
  Prepare;
  //начать игру:
  Pole.StartGame;
end;
```

Старт новой игры – событие нетривиальное: здесь нужно всё *всесторонне обдумать*:

```
//ВЫВЕСТИ ПЕРВЫЕ ШАРИКИ
procedure TPole.StartGame;
begin
  //очистить поле:
  Clear;
```

```
  //показать следующие шарики в табло:
  InitNextBalls;
  ShowNextBalls;

  //разместить первые шарики в массиве:
  InitFirstBalls;
  //показать шарики на поле:
  CreateBalls;

  //оставшиеся ходы:
  tonus:= tTonus.Create;
  tonus.GetValue;
  tonus.ShowValue;

  //число ходов:
  Hod:= tHod.Create;
  Hod.ShowValue;

  //очки:
  Score.init;
  Score.ShowValue;

  //инициализация поля:
  Init;
  //снять выделение:
  DeActivateBall;

  sound('newgame');
  sleep(1500);
  application.ProcessMessages;
end;
```

Поскольку игровое поле представлено у нас и сеткой *dgPole* на форме, и массивом клеток *Cell: array[0..MAX_WIDTH, 0..MAX_HEIGHT] of TCell;*, то в массиве нужно убрать все шарики с поля, присвоив переменным *Pole.Cell[].BallColor* значение *bcNone*, что и означает отсутствие шариков в соответствующих клетках, а во все клетки сетки мы должны вывести картинку, изображающую *пустую* клетку:

```
//ОЧИСТИТЬ ПОЛЕ
procedure TPole.Clear;
var
  i, j: integer;
  dr: TRECT;
begin
  with form1, dgPole do
    for j:= 0 to RowCount do
      for i:= 0 to ColCount do begin
        Pole.Cell[i, j].BallColor:= bcNone;
        dr:= CellRect(i, j);
        imlPic.Draw(Canvas, dr.left, dr.top, 0);
```

```
      end;
end;
```

Ну, если с массивом всё ясно, то манипуляции с полем ещё требуют пояснений.

Картинки с клетками поля во всех состояниях хранятся в компоненте *imlPic: TImageList* (Рис. Ф6.26). Их огромное количество, что объясняется анимацией.

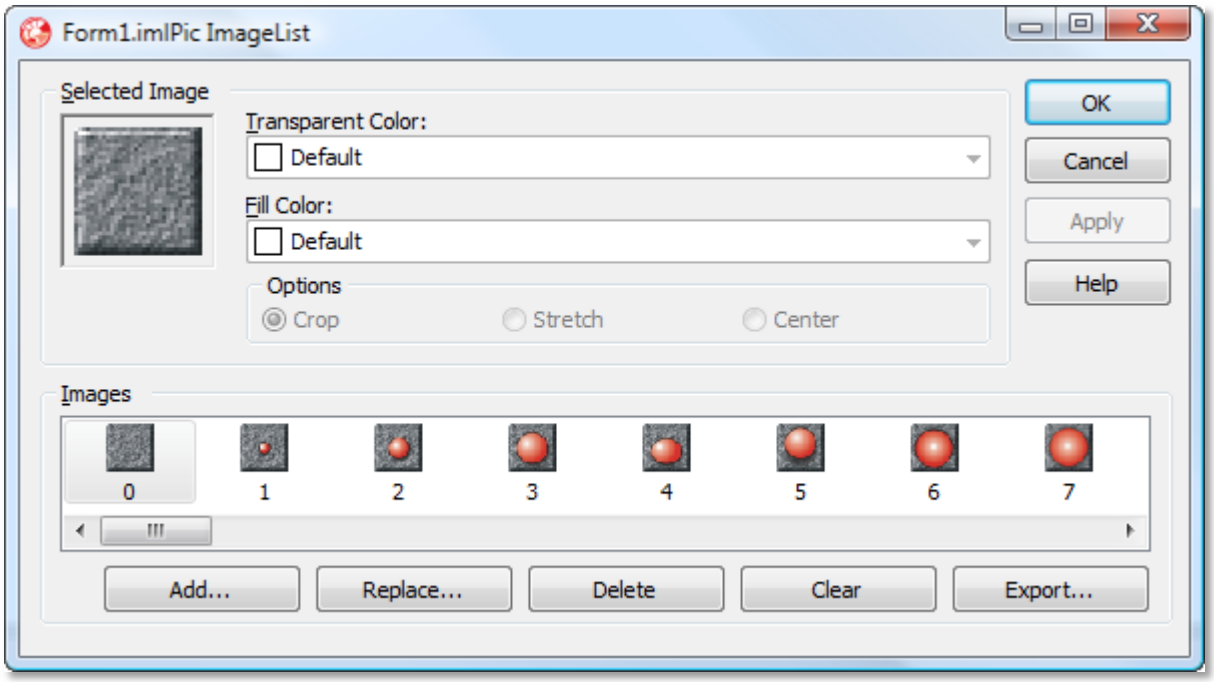


Рис. Ф6.26. Шарики в списке картинок

Все они имеют размеры 32 x 32 пикселя, чтобы точно соответствовать клеткам поля. Конечно, требуется некоторая графическая сноровка, чтобы их нарисовать (Рис. Ф6.27).

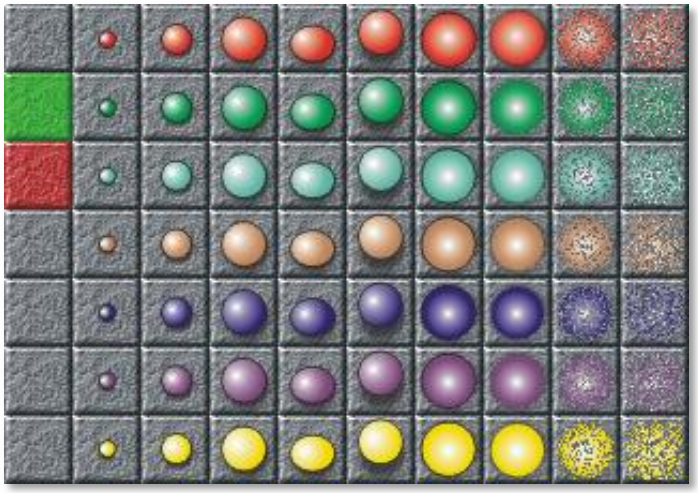


Рис. Ф6.27. Кадры анимации шариков

Лично я пользовался векторным редактором *CorelDRAW*, чего, как говорится, и вам желаю.

Когда картинки уже загружены в *imlPic*, то вы вести их в нужную клетку поля не представляет ни малейшего труда:

```
dr:= CellRect(i, j);
imlPic.Draw(Canvas, dr.left, dr.top, 0);
```

Так как в компоненте *TDrawGrid* картинки «не держатся», то их необходимо постоянно перерисовывать при изменении ситуации на поле:

```
//ОТРИСОВАТЬ СЕТКУ
procedure TForm1.dgPoleDrawCell(Sender: TObject; ACol, ARow:
Integer;
  Rect: TRect; State: TGridDrawState);
var
  r, dr: TRECT;
  index: integer;
begin
  //размеры картинок:
  dr:= Bounds(Rect.Left, Rect.Top, wCell, hCell);
  //картинка с шариком в ячейке:
  index:= (ord(Pole.Cell[ACol, ARow].BallColor));
  if index> 0 then //- в клетке есть шарик
    index:= (index-1)* period + 3
  else              //- пустая клетка
    index:= (ord(Pole.Cell[ACol, ARow].CellColor))* period;
  //скопировать картинку из списка в ячейку:
  imlPic.Draw(dgPole.Canvas, dr.left, dr.top, index);
end;
```

Константа *PERIOD* равна числу картинок при анимации шарика:

```
const
  //число картинок на шарик одного цвета:
  PERIOD= 10;
```

При инициализации поля в *процедуре tPole.Init* все клетки поля (не шарики!) окрашиваются в *серый* цвет, который они сохраняют, пока ни один шарик на поле не выделен:

```
//ИНИЦИАЛИЗАЦИЯ
procedure tPole.Init;
var i, j: integer;
begin
  //все клетки - серые:
  for j:= 0 to Height do
    for i:= 0 to Width do
```

```
      Cell[i,j].CellColor:= ccGray;

  //нет выделенного шарика:
  SelBall:= Point(-1,-1);
end;
```

В переменной *SelBall* хранятся координаты того шарика, на котором игрок кликнул. Так как координаты поля начинаются с *нуля*, то *отрицательные* значения говорят о том, что выделенного шарика на поле нет.

Затем в процедуре *StartGame* заполняется массив *NextBalls*, в котором хранятся шарики, которые появятся на поле после первого хода игрока:

```
//ИНИЦИАЛИЗАЦИЯ СЛЕДУЮЩИХ ШАРИКОВ
procedure TPole.InitNextBalls;
var
  i: integer;
begin
  Randomize;
  for i:= 1 to NumBalls.nPotom do begin
    NextBalls[i]:= TBallColor(ord(random(NumBalls.nColor)+1));
  end
end;
```

Также их нужно показать в сетке *dgNextBalls*, которая расположена справа от игрового поля:

```
//ПОКАЗАТЬ СЛЕДУЮЩИЕ ШАРИКИ В ТАБЛО
procedure TPole.ShowNextBalls;
var
  i: integer;
  dr: TRECT;
  index: integer;
begin
  for i:= 1 to NumBalls.nPotom do begin
    index:= (ord(NextBalls[i])-1) * period;
    dr:= form1.dgNextBalls.CellRect(0, i-1);
    form1.imlPic.Draw(form1.dgNextBalls.Canvas, dr.left, dr.top,
index+3);
  end;
end;
```

Процедура отрисовки клеток этой сетки:

```
////////////////////////////////////////////////////////
////////////////////// СЕТКА-ТАБЛО ////////////////////////
////////////////////////////////////////////////////////

procedure TForm1.dgNextBallsDrawCell(Sender: TObject; ACol, ARow:
Integer;
```

```
  Rect: TRect; State: TGridDrawState);
var
  dr: TRECT;
  index: integer;
begin
  dr:= dgNextBalls.CellRect(0,ARow);
  //стереть ячейки без шариков:
  if ARow+1 > NumBalls.nPotom then begin
    dgNextBalls.Canvas.Brush.Color:= fon;
    dgNextBalls.Canvas.FillRect(dgNextBalls.CellRect(ACol, ARow))
  end
  else begin
  //вывести шарик в ячейке:
  index:= (ord(Pole.NextBalls[ARow+1])-1) * period;
  imlPic.Draw(dgNextBalls.Canvas, dr.left, dr.top, index+3);
  end;
end;
```

Каждый программист должен предусмотреть в программе *уловки*, помогающие в игре. Мы сделаем так:

```
//ХИТРЫЙ ПРИЁМ
procedure TForm1.dgNextBallsMouseDown(Sender: TObject;
  Button: TMouseButton; Shift: TShiftState; X, Y: Integer);
var
  ACol, ARow: integer;
begin
  //координаты клетки:
  dgNextBalls.MouseToCell(X, Y, ACol, ARow);
  //если нажать кнопку мыши на первом шарике,
  //то все след. шарики заменятся на новые:
  if ARow= 0 then begin
    //показать следующие шарики в табло:
    Pole.InitNextBalls;
    Pole.ShowNextBalls;
  end
  //если нажать кнопку мыши на втором шарике,
  //то поле очистится:
  else if ARow= 1 then begin
    Pole.Clear;
    //снять выделение:
    Pole.DeActivateBall;
    //показать следующие шарики в табло:
    Pole.InitNextBalls;
    Pole.ShowNextBalls;

    //разместить первые шарики в массиве:
    Pole.InitFirstBalls;
    //появление шариков на поле с бульканием:
    Pole.CreateBalls;

    //осталось ходов:
    tonus:= tTonus.Create;
```

```
    tonus.GetValue;
    tonus.ShowValue;
  end;
end;
```

То есть: нажимая *первый* (верхний) шарик в сетке-табло, мы получаем *новый* набор следующих шариков и можем выбрать нужную нам комбинацию. Следующий приём ещё более радикальный. Нажав *второй* шарик, мы полностью очистим поле, сохранив при этом все свои честно заработанные очки!

Немного нелогично, но после *следующих* шариков, мы заполняем массив *Cell текущих* шариков (тех, что *сразу* появляются на поле):

```
//ИНИЦИАЛИЗАЦИЯ ПЕРВЫХ ШАРИКОВ
procedure TPole.InitFirstBalls;
var
  i, j, n: integer;
begin
  //разместить первые шарики в массиве:
  n:= 0;
  repeat
    i:= random(Width);
    j:= random(Height);
    if Cell[i, j].BallColor= bcNone then begin
      //поместить в пустую клетку шарик случайного цвета:
      Cell[i, j].BallColor:=
TBallColor(ord(random(NumBalls.nColor)+1));
      Cell[i, j].new:= true;
      inc(n)
    end;
  until n= NumBalls.nSrazu;
end;
```

Для этого мы выбираем *случайную* клетку поля и, если она *пустая*, то помещаем в неё шарик опять же *случайного* цвета (из возможных), а также указываем, что шарик «новый»:

```
Cell[i, j].new:= true;
```

В следующей процедуре:

```
//ПОЯВЛЕНИЕ ШАРИКОВ:
procedure TPole.CreateBalls;
var
  i, j, index: integer;
  dr: TRECT;
begin
  for j:= 0 to Height-1 do
```

```
    for i:= 0 to Width-1 do begin
      dr:= form1.dgPole.CellRect(i, j);
      index:= (ord(Pole.Cell[i, j].BallColor)-1) * period;
      if Cell[i, j].new= true then begin
        Cell[i, j].new:= false;
        form1.imlPic.Draw(form1.dgPole.Canvas, dr.left, dr.top,
index+1);
        Delay(DelayCreate);
        form1.imlPic.Draw(form1.dgPole.Canvas, dr.left, dr.top,
index+2);
        Delay(DelayCreate);
        Sound('buljk');
        form1.imlPic.Draw(form1.dgPole.Canvas, dr.left, dr.top,
index+ 3);
      end;
    end;
end;
```

мы просматриваем всё поле и, найдя «новый» шарик, выводим его на поле с анимацией и звуковым эффектом.

Здесь нам протребуются две простенькие процедуры. Первая издаёт звуки (файлы хранятся в *папке WAV*):

```
//ВОСПРОИЗВЕСТИ ЗВУК
procedure Sound(name: PChar);
var s: PChar;
begin
  if form1.sbtSound.down= true then begin
    //сформировать имя файла:
    s:= PChar('wav\'+ name + '.wav');
    sndPlaySound(s, SND_ASYNC or SND_FILENAME);
  end;
end;
```

Тут следует заметить, что булькающие звуки, сопровождающие появление шариков на свет божий, могут и раздражать игрока, поэтому добавим на форму *кнопку sbtSound*  (**Hint**=Звук вкл-откл **AllowAllUp**=True **Down**=True **GroupIndex**=1). Она действует как *переключатель*. Когда кнопка находится в *нажатом* положении (по умолчанию это так и есть, то есть мы надеемся, что игроку приятен наш выбор звуков), звук воспроизводится, когда в *отжатом* – нет. Всё демократично…

Так как шарики возникают в сетке излишне резво, то все фазы их рождения разделены паузой:

```
//ЗАДЕРЖКА
procedure Delay(MSecs: Longint);
```

```
var
  FirstTickCount, Now: Longint;
begin
  FirstTickCount := GetTickCount;
  repeat
    Application.ProcessMessages;
    Now := GetTickCount;
  until (Now - FirstTickCount >= MSecs) or (Now < FirstTickCount);
end;
```

При создании поля устанавливается такая *задержка*:

```
DelayCreate:= 80;
```

Вы можете изменить её в полном соответствии со своим темпераментом…

Идём дальше. В каждой игре важна *статистика*, без которой играть неинтересно! В нашей игре - это количество оставшихся у нас ходов:

```
  //оставшиеся ходы:
  tonus:= tTonus.Create;
  tonus.GetValue;
  tonus.ShowValue;
```

Число уже сделанных ходов:

```
  //число ходов:
  Hod:= tHod.Create;
  Hod.ShowValue;
```

Набранные очки:

```
  //очки:
  Score.init;
  Score.ShowValue;
```

Заведём для статистики ещё три *класса*: *tTonus*, *tHod* и *tScore* в главном модуле программы. Затем объявим соответствующие *переменные*:

```
var
  //игровой тонус:
  Tonus: tTonus;
  //число ходов:
  Hod: tHod;
  //набранные очки:
  Score: tScore;
```

Рассмотрим все классы по очереди. *Игровой «тонус»* :

```
  //игровой тонус:
  type tTonus= class
    //число оставшихся ходов:
    value: integer;
    //число пустых клеток на поле:
    EmptyCell: integer;
    //есть ли на поле шарики?
    EmptyPole: boolean;
    //есть ли на поле хотя бы одна свободная клетка?
    FullPole: boolean;
    constructor Create;
    //показать:
    procedure ShowValue;
    //получить значение тонуса:
    procedure GetValue;

  end;
```

Здесь нас в, первую очередь, может интересовать, сколько ходов мы сможем ещё продержаться – *value*. Определить их число можно так: подсчитаем оставшиеся пустые клетки *EmptyCell* и разделим их на число шариков, появляющихся после каждого хода - *NumBalls.nPotom*. Значение «тонуса» выводим для всеобщего обозрения в метку *lblTonus* (Рис. Ф6.28).



Рис. Ф6.28. Тонус отсутствует!

В компонент *TImage* мы загружаем картинку с подходящей надписью (все картинки находятся в *папке image*).

```
////////////////////////////////////////////////////////
//////////////////  ИГРОВОЙ ТОНУС ///////////////////////
////////////////////////////////////////////////////////

constructor tTonus.Create;
begin
  inherited;
  value:= 0;
end;

//ВЫЧИСЛИТЬ КОЛИЧЕСТВО ОСТАВШИХСЯ ХОДОВ
procedure tTonus.GetValue;
var
  i, j: integer;
begin
```

```
  //подсчитать число пустых клеток на поле:
  EmptyCell:= 0;
  for j:= 0 to Pole.Height-1 do
    for i:= 0 to Pole.Width-1 do
      if Pole.Cell[i, j].BallColor= bcNone then inc(EmptyCell);
  //число оставшихся ходов:
  value:= EmptyCell div NumBalls.nPotom;
  //проверить, не пусто ли поле:
  if EmptyCell = Pole.Height * Pole.Width then
    EmptyPole:= true
  else
    EmptyPole:= false;
  //проверить, есть ли на поле свободные клетки:
  if EmptyCell = 0 then
    FullPole:= true
  else
    FullPole:= false;
end;

//ВЫВЕСТИ ЗНАЧЕНИЕ ТОНУСА НА ЭКРАН
procedure tTonus.ShowValue;
begin
  form1.lblTonus.Caption:= inttostr(value);
  form1.lblTonus.Invalidate;
  application.ProcessMessages
end;
```

Во всех играх важно знать *число* сделанных *ходов*:

```
  //ходы:
  type tHod= class
    //число сделанных ходов:
    value: integer;
    constructor Create;
    //показать:
    procedure ShowValue;
  end;
```

Здесь всё просто:

```
////////////////////////////////////////////////////////
////////////////////// ХОДЫ ////////////////////////////
////////////////////////////////////////////////////////

constructor tHod.Create;
begin
  inherited;
  value:= 0;
end;

//ВЫВЕСТИ ЗНАЧЕНИЕ НА ЭКРАН
procedure tHod.ShowValue;
```

```
begin
  form1.lblHod.Caption:= inttostr(value);
  form1.lblHod.Invalidate;
  application.ProcessMessages
end;
```

Число ходов мы показываем в метке *lblHod* (Рис. Ф6.29).



Рис. Ф6.29. Пора ходить!

Ещё более важно знать *набранные очки*, чтобы установить *рекорд*:

```
  //очки:
  type tScore= class
    //набранные очки:
    value: integer;
    //рекорд:
    Rec: integer;
    constructor Create;
    procedure Init;
    //показать:
    procedure ShowValue;
    //установить новое значение:
    procedure SetValue(n: integer);
    //запомнить новый рекорд:
    procedure SetRec;
  end;
```

```
////////////////////////////////////////////////////////
////////////////////// ОЧКИ ////////////////////////////
////////////////////////////////////////////////////////

constructor tScore.Create;
begin
  inherited;
end;

//ИНИЦИАЛИЗАЦИЯ
procedure tScore.Init;
begin
  value:= 0;
end;

//ПОДСЧИТАТЬ НАБРАННЫЕ ОЧКИ
procedure tScore.SetValue(n: integer);
```

```
begin
  if n> 0 then
    if TypeFig= tfLines then
      value:= value + 5*n - 20
    else
      value:= value + 6*n - 26
end;

//ВЫВЕСТИ ОЧКИ НА ЭКРАН
procedure tScore.ShowValue;
begin
  with form1 do begin
    lblScore.Caption:= inttostr(value);
    lblScore.Invalidate;
    lblRecord.Caption:= inttostr(rec);
    lblRecord.Invalidate;
  end;
  application.ProcessMessages
end;

//РЕКОРДНЫЕ ОЧКИ
procedure tScore.SetRec;
begin
  if value> rec then
    rec:= value;
  showValue;
end;
```

Набранные очки и рекорд мы печатаем в метках *lblScore* и *lblRecord,* соответственно (Рис. Ф6.30).

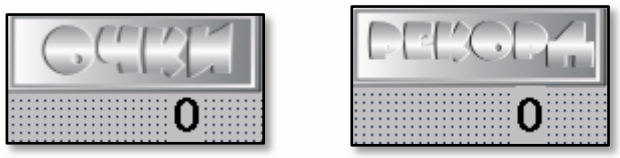


Рис. Ф6.30. Текущие очки и рекорд

В игре, конечно, очень важен подсчёт набранных очков. Я остановился на таких формулах:

```
//ПОДСЧИТАТЬ НАБРАННЫЕ ОЧКИ
procedure tScore.SetValue(n: integer);
begin
  if n> 0 then
    if TypeFig= tfLines then
      value:= value + 5*n - 20
    else
      value:= value + 6*n - 26
end;
```

Вы можете выбрать формулы по своему вкусу!

И - нам осталось рассмотреть самый конец процедуры *TPole.StartGame*:

```
  //инициализация поля:
  Init;
  //снять выделение:
  DeActivateBall;

  sound('newgame');
  sleep(1500);
  application.ProcessMessages;
```

Процедура *Init* нам уже известна, а *процедура DeActivateBall* просто снимает выделение с шарика:

```
//СНЯТЬ ВЫДЕЛЕНИЕ С ШАРИКА
procedure tPole.DeActivateBall;
var
  i, j: integer;
begin
  SelBall:= Point(-1,-1);
  //остановить таймер:
  form1.timer2.Interval:= 1;
  form1.timer2.Enabled:= false;

  //все клетки - серые:
  for j:= 0 to Height-1 do
    for i:= 0 to Width-1 do
      Cell[i,j].CellColor:= ccGray;
  form1.dgPole.Invalidate;
  application.ProcessMessages;
end;
```

Сейчас мы проясним ситуацию с шариками на поле до полной прозрачности.

Мы находимся в положении, когда всё готово для новой игры: имеется поле заданных размеров, на нём находится некоторое количество шариков, нам известны следующие шарики, мы имеем всю информацию о ходе игры. Что должен сделать игрок дальше? – Конечно, выбрать какой-нибудь шарик на поле и указать место, на которое он должен переместиться!

Итак, нажимаем кнопку мышки на клетке поля:

```
//НАЖАТЬ КНОПКУ МЫШИ НА ПОЛЕ
procedure TForm1.dgPoleMouseDown(Sender: TObject; Button: TMouseButton;
```

```
  Shift: TShiftState; X, Y: Integer);
var
  ACol, ARow: integer;
begin
  //координаты клетки:
  dgPole.MouseToCell(X, Y, ACol, ARow);
  Pole.Click(ACol, ARow);
end;
```

Обработка этого события происходит в *процедуре tPole.Click:*

```
//ЩЕЛЧОК НА ПОЛЕ
procedure tPole.Click(X, Y: integer);
//X, Y - координаты клетки
begin
  sound('click');
  //если кнопка нажата в клетке с шариком, выделить его:
  if Cell[X, Y].BallColor <> bcNone then
    ActivateBall(X, Y)
  //если кнопка нажата в пустой клетке, то сделать ход:
  else
    DoHod(X, Y)
end;
```

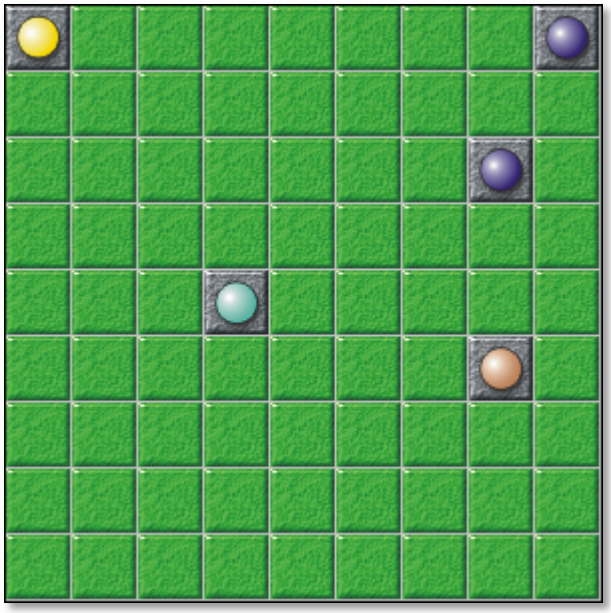
Здесь нужно проверить, есть ли шарик в этой клетке. Если есть, мы «активируем» (выделяем) его. Если же клетка *пустая*, то нужно выполнить ход.

Рассмотрим первую процедуру - *ActivateBall(X, Y):*

```
//ВЫДЕЛИТЬ ШАРИК
procedure tPole.ActivateBall(X, Y: Integer);
//X, Y - координаты клетки с шариком
var
  i,j: integer;
begin
  //если шарик повторно выделен:
  if  (X= SelBall.x) and (Y= SelBall.y) then
    DeActivateBall
  //если выделен новый шарик -
  else begin
    //запомнить координаты шарика:
    SelBall:= Point(X, Y);
    //запустить прыжки шарика:
    JumpBall;
    //выделить доступные клетки:
    FindPath(Point(-1,-1));
      for j:= 0 to height-1 do
        for i:= 0 to width-1 do
          //если в клетке число >0, то клетка доступна:
          if masPath[i,j]> 0 then
            Cell[i,j].CellColor:= ccGreen
          //иначе - клетка недоступна:
          else
```

```
            Cell[i,j].CellColor:= ccRed;
    form1.dgPole.Invalidate;
  end;
end;
```

Поскольку на поле нужно кликать в *двух* клетках, то мы должны как-то отмечать кликнутый шарик, чтобы не потерять его из виду, делая второй клик. Сделать это можно по-разному. Я выбрал простую анимацию выделенного шарика – он *прыгает* (Рис. Ф6.31) в своей клетке (к сожалению, на статичной картинке этого не покажешь).



Выделен шарик в третьей сверху строке

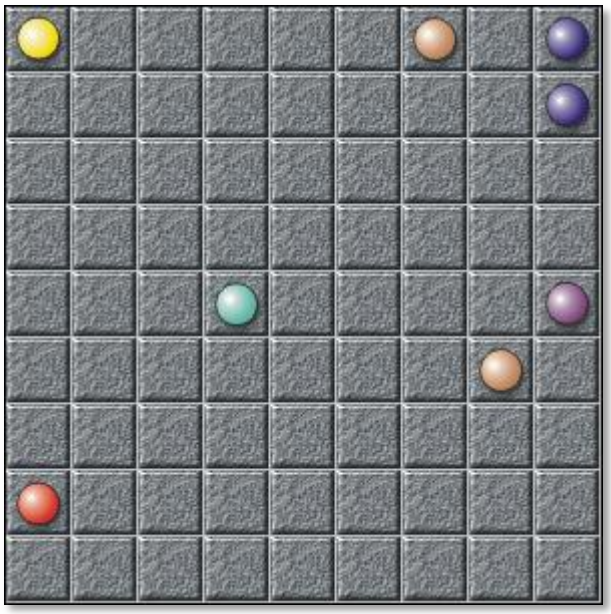


Рис. Ф6.31. Ход сделан – выделенный шарик переместился в новую клетку

В игре бывает всякое: вы могли по ошибке кликнуть не тот шарик. Тогда кликните на нём ещё раз – и он станет *обычным* шариком:

```
  //если шарик повторно выделен:
  if  (X= SelBall.x) and (Y= SelBall.y) then
    DeActivateBall
```

Если клик удачный, то мы запоминаем координаты клетки и заставляем шарик прыгать:

```
    //запомнить координаты шарика:
    SelBall:= Point(X, Y);

    //запустить прыжки шарика:
    JumpBall;
```

Так как анимация - процесс весьма сложный, то заведём новый тип данных *TAnimate*:

```
//анимация шариков:
  TAnimate= Record
   //интервал:
   Interval: cardinal;
   //координаты клетки поля:
   x, y: integer;
   //список индексов в списке:
   index: array[1..100] of integer;
   //текущий индекс в массиве:
   ptrIndex: integer;
   //число картинок в анимации:
   Frame: integer;
   //имя файла звука:
   sound: PChar;
   //индекс для воспроизведения звука:
   SoundIndex: integer;
   //число повторений:
   Rep: integer;
   //выполнено:
   Exec: integer;
  end;
```

И соответствующую *переменную*:

```
var
  //анимированная ячейка:
  AnimateCell: TAnimate;
```

После клика на шарике мы задаём параметры анимации:

```
//ВЫДЕЛИТЬ ШАРИК
```

```
procedure tPole.JumpBall;
var i: integer;
begin
    //запустить прыжки шарика:
    AnimateCell.Interval:= 120;
    AnimateCell.x:= SelBall.x;
    AnimateCell.y:= SelBall.y;
    i:= (ord(Cell[SelBall.x, SelBall.y].BallColor)-1) * period;
    AnimateCell.index[1]:=i+4;
    AnimateCell.index[2]:=i+3;
    AnimateCell.index[3]:=i+5;
    AnimateCell.index[4]:=i+3;
    AnimateCell.ptrIndex:= 1;
    AnimateCell.Frame:= 4;
    AnimateCell.sound:= 'jump1';
    AnimateCell.SoundIndex:= 4;
    AnimateCell.Rep:= 0;

end;
```

Анимация шарика реализована так. Через заданный интервал времени *Interval* в заданную клетку *(x, y)* последовательно выводятся картинки из компонента с картинками *imlPic*, который мы уже рассматривали. Наиболее естественно менять картинки с помощью компонента *таймер Timer2*  (**Interval**=1 **Enabled**=False)

Как только он будет запущен (*form1.timer2.Enabled:= true;),* вызывается процедура

```
//ТАЙМЕР-АНИМАТОР:
procedure TForm1.Timer2Timer(Sender: TObject);
begin
  //запустить подпрыгивающий шарик:
  Pole.Animation;
end;
```

А она, в свою очередь, через заданный промежуток времени вызывает процедуру *TPole.Animation:*

```
//ПОДПРЫГИВАЮЩИЙ ШАРИК
procedure TPole.Animation;
//x. y - координаты клетки поля;
//index - номер картинки;
var
  dr: Trect;
  x,y, index: integer;
begin
  //все повторения выполнены:
```

```
  if (AnimateCell.Rep<> 0) and (AnimateCell.Exec= AnimateCell.Rep)
then
  begin
    form1.timer2.Enabled:= false;
    exit;
  end;
  form1.timer2.Interval:= AnimateCell.Interval;
  x:= AnimateCell.x;
  y:= AnimateCell.y;
  //квадрат клетки:
  dr:= form1.dgPole.CellRect(x,y);
  //номер текущей картинки:
  index:= AnimateCell.Index[AnimateCell.ptrIndex];
  //вывести текущую картинку:
  form1.imlPic.Draw(form1.dgPole.Canvas, dr.left, dr.top, Index);

  if (AnimateCell.sound<> '') and
    (AnimateCell.SoundIndex= AnimateCell.ptrIndex)
  then
    Sound(AnimateCell.sound);

  inc(AnimateCell.ptrIndex);
  if AnimateCell.ptrIndex> AnimateCell.Frame then
    AnimateCell.ptrIndex:= 1;

  if AnimateCell.rep<> 0 then
    inc(AnimateCell.exec);
end;
```

Тут всё несколько заковыристо, но это свойство всех анимаций!

Этим можно было бы и ограничиться, но давайте поможем игроку, ведь ему будет значительно легче ориентироваться на поле, если мы *отметим* клетки, в которые он *может* сделать ход (Рис. Ф6.32).

```
    //выделить доступные клетки:
    FindPath(Point(-1,-1));
      for j:= 0 to height-1 do
        for i:= 0 to width-1 do
          //если в клетке число >0, то клетка доступна:
          if masPath[i,j]> 0 then
            Cell[i,j].CellColor:= ccGreen
          //иначе - клетка недоступна:
          else
            Cell[i,j].CellColor:= ccRed;
```

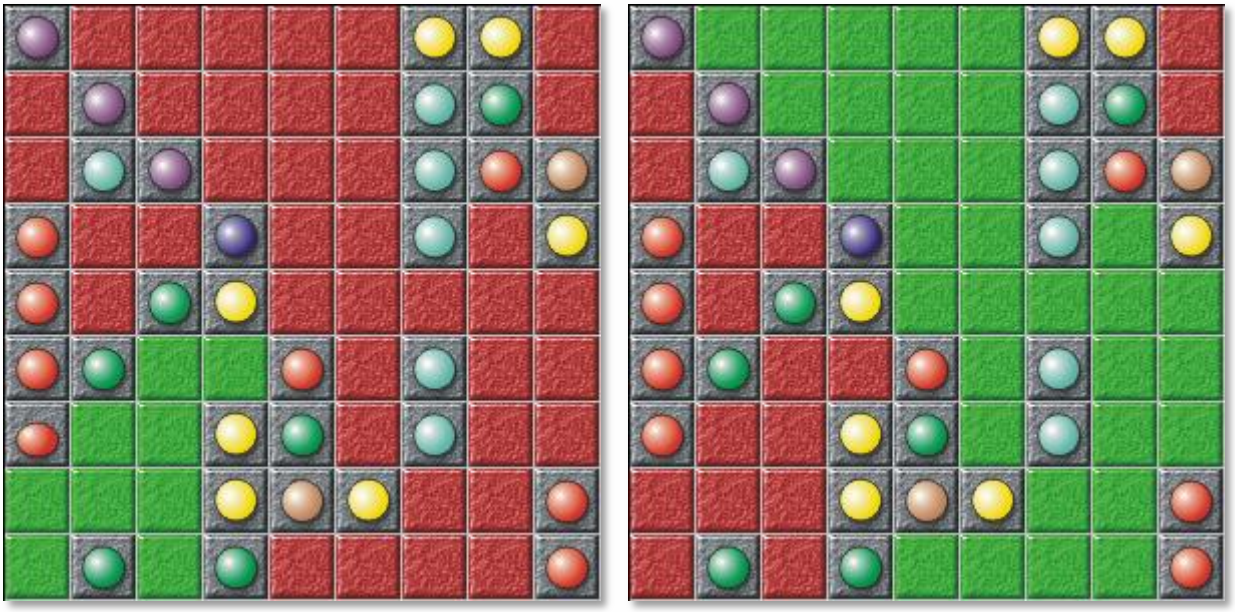


Рис. Ф6.32. Выделен *красный* шарик в 7-й строке Выделен *жёлтый* шарик в 4-й строке

Эти рисунки показывают, как изменяется цвет клеток на поле после клика на шарике. *Серые* в нормальном состоянии клетки окрашиваются в *зелёный* – они доступные – и в *красный* – недоступные – цвет. Согласитесь, теперь выбрать нужный ход значительно проще!

Как найти путь в лабиринте, мы уже разобрались в начале главы. Там же мы отмечали, что этот алгоритм позволяет и закрашивать замкнутые области. Для этого достаточно указать координаты конечной клетки так, чтобы она была заведомого за пределами поля:

```
FindPath(Point(-1,-1));
```

Путь, естественно, функция *TPole.FindPath* путь не найдёт, но зато она отметит все клетки, доступные из начальной (в данном случае это клетка с выделенным шариком). В массиве *masPath* им соответствуют *ненулевые* значения:

```
//НАЙТИ ПУТЬ МЕЖДУ ДВУМЯ КЛЕТКАМИ
function TPole.FindPath(EndCell: TPoint): boolean;
var
  //список координат:
  CoordList: array[1..MAX_WIDTH * MAX_HEIGHT] of TPoint;
  //указатели в списке:
  ptrWrite, ptrRead: integer;
  p, q: integer;
  i, j: integer;

  //проверить координаты
  function TestCoord(x,y: integer): boolean;
```

```
  begin
    Result:= true;
    if (x<0) or (x> Width-1) or (y<0) or (y> Height-1)or
     (masPath[x,y]<> GO) then Result:= false;
  end;

begin
  //формируем массив лабиринта:
  for j:= 0 to height-1 do
    for i:= 0 to width-1 do
      //если в клетке есть шарик, то клетка непроходима:
      if Cell[i,j].BallColor<> bcNone then
        masPath[i,j]:= STOP
      //иначе - клетка непроходима:
      else
        masPath[i,j]:= GO;

  //заносим в список координаты начальной клетки:
  CoordList[1]:= SelBall;
  //в начальную клетку в массиве CellPath ставим BEG_CELL= 0:
  masPath[SelBall.x,SelBall.y]:= BEG_CELL;
  //устанавливаем указатель для считывания координат на начало
списка:
  ptrRead:= 1;
  //устанавливаем указатель для записи новых координат на
следующий индекс:
  ptrWrite:= 2;

  //двигаемся от начала списка к его концу, пока он не кончится:
  while ptrRead < ptrWrite do begin
    //координаты текущей клетки:
    p:= CoordList[ptrRead].x; q:= CoordList[ptrRead].y;
    //проверяем соседние с ней клетки:
    for i:= p - 1 to p + 1 do
      for j:= q - 1 to q + 1 do
        //если нашли соседнюю проходимую клетку,
        if ((i=p) or (j=q)) and TestCoord(i,j) then
        begin
          //то записываем в неё число, на единицу большее,
          //чем в текущей клетке:
          masPath[i,j]:= masPath[p,q] + 1;

          //если дошли до конечной клетки,
          if (i= EndCell.x) and (j= EndCell.y) then begin
            //то путь найден:
            Result:= True;
            exit;
          end
          else begin
            //записываем координаты соседней клетки в конец
списка:
            CoordList[ptrWrite]:= Point(i,j);
            //перемещаем указатель:
```

```
            inc(ptrWrite);
          end;
        end;
      //переходим к следующей клетке в списке:
      inc(ptrRead);
  end;
  //путь не найден:
  Result:= False;
end;
```

После формирования массива *masPath* нам достаточно просмотреть все клетки поля и окрасить их в соответствии со значением *Cell[].CellColor*.

Итак, игрок выделил шарик, а затем указал клетку, в которую его следует переместить:

```
//ПЕРЕМЕСТИТЬ ШАРИК
procedure tPole.DoHod(X, Y: Integer);
//X, Y - координаты клетки
var
  n, i, j: integer;
begin
  //если шарик не был выделен, то перемещать нечего:
  if (SelBall.x= -1) and(SelBall.y= -1) then begin
    Sound('cantmove'); exit
  end;

  //найти путь в новую клетку -

  //если пути нет, то переместить шарик нельзя:
  if not FindPath(Point(x,y)) then begin // - пути нет
    Sound('cantmove'); exit
  end;

  //если путь найден - переместить шарик:
  DoMoveBall(x, y);

  //увеличить число сделанных ходов:
  inc(Hod.value);
  //и показать на экране:
  Hod.ShowValue;

  //если образовалась нужная фигура, уничтожить её:
  n:= DeleteFigs;

  //вычислить и показать набранные очки:
  Score.SetValue(n);
  Score.ShowValue;

  //вычислить и показать игровой тонус:
  Tonus.GetValue;
```

```
  Tonus.ShowValue;

  //если был сделан результативный ход и на поле есть шарики,
  //то ждать следующего хода:
  if (n> 0) and (Tonus.EmptyPole= false) then exit;

  //вывести следующие шарики NextBalls -

  //если для новых шариков не осталось места, то игра проиграна:
  if Tonus.EmptyCell < NumBalls.nPotom then begin
    GameOver;
    exit
  end;

  //если поле пустое - победа:
  if Tonus.EmptyPole= true then begin
    GameWin;
    exit
  end;

  //найти место для новых шариков:
  n:= 0;
  repeat
    i:= random(Width);
    j:= random(Height);
    if Cell[i, j].BallColor= bcNone then begin
      //поместить в пустую клетку «следующий» шарик:
      Cell[i, j].BallColor:= NextBalls[n+1];
      Cell[i, j].new:= true;
      inc(n)
    end;
  until n= NumBalls.nPotom;
  //появление шариков с булькaнием:
  CreateBalls;

  //если образовалась нужная фигура, уничтожить её:
  n:= DeleteFigs;

  //вычислить и показать набранные очки:
  Score.SetValue(n);
  Score.ShowValue;

  //вычислить и показать игровой тонус:
  Tonus.GetValue;
  Tonus.ShowValue;

  //показать следующие шарики:
  InitNextBalls;
  ShowNextBalls;

  //если поле пустое - победа:
  if Tonus.EmptyPole= true then begin
    GameWin;
```

```
    exit
  end;

  //если ходов нет - поражение:
  if Tonus.FullPole= true then begin
    GameOver;
    exit
  end;
end;
```

Сначала мы должны учесть «несчастный случай», когда игрок кликнул на *пустой* клетке поля, не выделив шарик. Понятно, что ничего хорошего в этом случае делать не нужно:

```
  //если шарик не был выделен, то перемещать нечего:
  if (SelBall.x= -1) and (SelBall.y= -1) then begin
    Sound('cantmove'); exit
  end;
```

Если игрок действует правильно, то мы ищем путь из начальной клетки в конечную:

```
  //если пути нет, то переместить шарик нельзя:
  if not FindPath(Point(x,y)) then begin // - пути нет
    Sound('cantmove'); exit
  end;
```

Хоть мы и позаботились о раскрашивании клеток поля в разные цвета, игрок может оказаться невнимательным и указать *недоступную* клетку. Если же путь найден, мы перемещаем шарик:

```
  //если путь найден - переместить шарик:
  DoMoveBall(x, y);
```

Занятие это не из простых:

```
//ПЕРЕМЕСТИТЬ ШАРИК
procedure tPole.DoMoveBall(X, Y: Integer);
//X, Y - координаты клетки
var
  dr: TRECT;
  index, i,j: integer;
  bc: TBallColor;
  p, q: integer;
begin
  //убрать выделение с клеток:
  DeactivateBall;
  //создать путь:
  CreatePath(Point(x,y));
```

```
//переместить шарик из начальной клетки в конечную:
if form1.sbtPath.down= true then
  for i:= 0 to masPath[x, y]-1 do
  begin
    //координаты очередной клетки:
    p:= path[i].x; q:= path[i].y;
    //переместить шарик из очередной клетки в следующую -
    //цвет шарика:
    bc:= Cell[p, q].BallColor;
    //очередная клетка становится пустой и серой:
    Cell[p, q].BallColor:= bcNone;

    //исчезновение шарика:
    dr:= form1.dgPole.CellRect(p, q);
    index:= (ord(bc)-1) * period;
    form1.imlPic.Draw(form1.dgPole.Canvas, dr.left, dr.top,
index+2);
    sleep(DelayMove);
    form1.imlPic.Draw(form1.dgPole.Canvas, dr.left, dr.top,
index+1);
    sleep(DelayMove);
    //пустая клетка:
    form1.imlPic.Draw(form1.dgPole.Canvas, dr.left, dr.top, 0);
    sleep(DelayMove);

    //в следующей появляется шарик:
    p:= path[i+1].x; q:= path[i+1].y;
    Cell[p, q].BallColor := bc;

    //появление шарика:
    dr:= form1.dgPole.CellRect(p, q);
    form1.imlPic.Draw(form1.dgPole.Canvas, dr.left, dr.top,
index+1);
    sleep(DelayMove);
    form1.imlPic.Draw(form1.dgPole.Canvas, dr.left, dr.top,
index+2);
    sleep(DelayMove);
    form1.imlPic.Draw(form1.dgPole.Canvas, dr.left, dr.top,
index+3);
    sleep(160);
    Sound('move1');
  end
  else begin
    //координаты первой клетки:
    i:= path[0].x; j:= path[0].y;
    //цвет шарика:
    bc:= Cell[i, j].BallColor;
    //первая клетка становится пустой и серой:
    Cell[i, j].BallColor:= bcNone;
    form1.dgPoleDrawCell(Self, i, j, form1.dgPole.CellRect(i,
j),[]);
    //координаты последней клетки:
```

```
    p:= path[masPath[x, y]].x; q:= path[masPath[x, y]].y;
    //в ней появляется шарик:
    Cell[p, q].BallColor := bc;
    form1.dgPoleDrawCell(Self, p, q, form1.dgPole.CellRect(p,
q),[]);
    Sound('move1');
    sleep(160);
  end;
end;
```

Сначала мы возвращаем клетки поля в нормальное состояние (серый цвет), затем формируем путь шарика в массиве *path*:

```
//СФОРМИРОВАТЬ  ПУТЬ
procedure TPole.CreatePath(EndCell: TPoint);
var
  n, LenPath: integer;
  i, j, p, q: integer;

  //проверить координаты:
  function TestCoord(x,y: integer): boolean;
  begin
    Result:= true;
    if (x<0) or (x> width-1) or (y<0) or (y> height-1)or
       (masPath[x,y]<> n-1) then
      Result:= false;
  end;

begin
  //длина пути равна числу в конечной клетке:
  LenPath:= masPath[EndCell.x, EndCell.y];
  n:= LenPath;
  //конечная клетка пути:
  path[n]:= EndCell;
  //двигаемся от неё к начальной клетке:
  repeat
    //найти соседнюю клетку с числом n-1:
    p:= path[n].x;  q:= path[n].y;
    //проверяем соседние клетки:
    for i:= p - 1 to p + 1 do
      for j:= q - 1 to q + 1 do
        //нашли подходящую клетку:
        if ((i=p) or (j=q)) and TestCoord(i,j) then
        begin
          //записываем её координаты:
          path[n-1]:= Point(i,j);
          break;
        end;
    //ищем клетку с предыдущим номером:
    dec (n);
  until n<0;
end;
```

Шарик анимированно исчезает в начальной клетке и появляется в соседней, и так - пока не допрыгает до *конечной* клетки пути. Процесс перемещения шарика, сопровождаемый бульканьем и анимацией, может быстро надоесть, поэтому в программе должна быть возможность отключить последовательное перемещение шарика и сразу «телепортировать» его в заданную точку игрового пространства. Для этого мы установим *кнопку sbtPath* (**Hint**=Путь вкл-откл **GroupInde**x=2), действующую подобно *sbtSound*. Когда она *нажата* - , - будет показан весь путь шарика по клеткам, если *отжата* - , - он *сразу* появится в конечной клетке.

Если в результате хода образовалась фигура (линия или блок) достаточного размера, то её нужно уничтожить:

```
  //увеличить число сделанных ходов:
  inc(Hod.value);
  //и показать на экране:
  Hod.ShowValue;

  //если образовалась нужная фигура, уничтожить её:
  n:= DeleteFigs;
```

Поиском и ликвидацией фигур у нас занимается *функция tPole.DeleteFigs:*

```
//УНИЧТОЖИТЬ ФИГУРУ
function tPole.DeleteFigs: integer;
//возвращает число уничтоженных шариков
var
  i,j,n, index: integer;
  bc: TBallColor;
  dr: TRect;

  //найти на поле блоки нужной величины
  function GetBlocks(x,y: integer; Select: boolean): boolean;
  //x,y - координаты первой клетки
  var
    masPath2: TField;
    //список координат:
    CoordList: array[1..MAX_WIDTH * MAX_HEIGHT] of TPoint;
    //указатели в списке:
    ptrWrite, ptrRead: integer;
    p, q: integer;
    i, j: integer;

    //проверить координаты
    function TestCoord(x,y: integer): boolean;
    begin
```

```
      Result:= true;
      if (x<0) or (x> Width-1) or (y<0) or (y> Height-1)or
       (masPath2[x,y]<> GO) then Result:= false;
    end;

  begin
    //сначала в блоке один шарик:
    n:= 1;
    //отметить в массиве первую клетку:
    if Select then masPath[x,y]:= 1;

    //формируем массив лабиринта:
    for j:= 0 to height-1 do
      for i:= 0 to width-1 do
        //если в клетке есть шарик, то клетка непроходима:
        if Pole.Cell[i,j].BallColor<> bc then
          masPath2[i,j]:= STOP
        //иначе - клетка непроходима:
        else
          masPath2[i,j]:= GO;

    //заносим в список координаты начальной клетки:
    CoordList[1]:= Point(x,y);
    //в начальную клетку в массиве CellPath ставим BEG_CELL= 0:
    masPath2[x,y]:= 0;
    //устанавливаем указатель для считывания координат на начало
списка:
    ptrRead:= 1;
    //устанавливаем указатель для записи новых координат на
следующий индекс:
    ptrWrite:= 2;

    //двигаемся от начала списка к его концу, пока он не кончится:
    while ptrRead < ptrWrite do begin
      //координаты текущей клетки:
      p:= CoordList[ptrRead].x; q:= CoordList[ptrRead].y;
      //проверяем соседние с ней клетки:
      for i:= p - 1 to p + 1 do
        for j:= q - 1 to q + 1 do
          //если нашли соседнюю проходимую клетку,
          if ((i=p) or (j=q)) and TestCoord(i,j) then
          begin
            inc(n);
            //то записываем в неё 1:
            masPath2[i,j]:= 1;
            if Select then masPath[i,j]:= 1;
            //записываем координаты соседней клетки в конец
списка:
            CoordList[ptrWrite]:= Point(i,j);
            //перемещаем указатель:
            inc(ptrWrite);
          end;
      //переходим к следующей клетке в списке:
```

```
      inc(ptrRead);
  end;

    //найдена ли цепочка нужной длины?
    if n>= LenBlocks then Result:= true
    else Result:= false;
  end;

  //уничтожить блок
  procedure DeleteBlocks;
  var
    i,j: integer;
  begin
  //проверить клетки поля:
  for j:= 0 to height do
    for i:= 0 to width do begin
      //цвет очередного шарика:
      bc:= Cell[i,j].BallColor;
      //пустая или проверенная клетка:
      if (bc= bcNone) or (masPath[i,j]= 1) then Continue;
      //искать вправо:
      if GetBlocks(i,j,false) then GetBlocks(i,j,true);
    end;
  end;

  //найти на поле цепочки нужной длины
  function GetLenChain(x,y,dx,dy: integer; Select: boolean):
boolean;
  begin
    //сначала в цепочке один шарик:
    n:= 1;
    //отметить в массиве первую клетку:
    if Select then masPath[x,y]:= 1;
    //координаты следующей клетки:
    x:= x+dx; y:= y+dy;
    while (x>=0) and (x< width) and (y>=0) and (y< height) and
          (Cell[x,y].BallColor= bc) do begin
      //цепочка удлинилась:
      inc(n);
      //отметить клетку:
      if Select then masPath[x,y]:= 1;
      //координаты следующей клетки:
      x:= x+dx; y:= y+dy;
    end;
    //найдена ли цепочка нужной длины?
    if n>= LenLines then Result:= true
    else Result:= false;
  end;

  //уничтожить линии
  procedure DeleteLines;
  var i, j: integer;
  begin
```

```
  //проверить клетки поля:
  for j:= 0 to height do
    for i:= 0 to width do begin
      //цвет очередного шарика:
      bc:= Cell[i,j].BallColor;
      if bc= bcNone then Continue; // - пустая клетка
      //искать вправо:
      if GetLenChain(i,j,1,0,false) then
GetLenChain(i,j,1,0,true);
      //искать вниз:
      if GetLenChain(i,j,0,1,false) then
GetLenChain(i,j,0,1,true);
      //искать вправо-вниз:
      if GetLenChain(i,j,1,1,false) then
GetLenChain(i,j,1,1,true);
      //искать влево-вниз:
      if GetLenChain(i,j,-1,1,false) then GetLenChain(i,j,-
1,1,true);
    end;
  end;


begin
  //обнулить массив masPath:
  for j:= 0 to height-1 do
    for i:= 0 to width-1 do
      masPath[i,j]:= 0;

     if TypeFig= tfBlocks then DeleteBlocks
     else Deletelines;

  n:= 0;
  //подсчитать число уничтожаемых шариков:
  for j:= 0 to height-1 do
    for i:= 0 to width-1 do
      if masPath[i,j]= 1 then inc(n);
  Result:= n;
  if n= 0 then exit;

  //уничтожить шарики:
  for n:= 1 to 5 do begin
    for j:= 0 to height-1 do
      for i:= 0 to width-1 do
        if masPath[i,j]= 1 then begin
          //исчезновение шарика:
          dr:= form1.dgPole.CellRect(i, j);
          bc:= Cell[i,j].BallColor;
          index:= (ord(bc)-1) * period+n+5;
          if n=5 then begin
            index:= 0; // - пустая клетка
            Cell[i,j].BallColor:= bcNone;
          end;
```

```
          form1.imlPic.Draw(form1.dgPole.Canvas, dr.left, dr.top,
index);
        end;
    sleep(DelayDestroy);
  end;
  Sound('destroy2');
end;
```

Так как линии и блоки отличаются по форме, то ими занимаются *разные* процедуры:

```
    if TypeFig= tfBlocks then DeleteBlocks
    else Deletelines;
```

Длину *линий* мы вычисляем так. Находим на поле шарик и считаем, сколько он имеет соседей справа, снизу, а также по диагоналям вниз-вправо и вниз-влево. Если удалось найти цепочки нужной длины, фукция *GetLenChain* возвращает *TRUE*, и вызывается вторично, чтобы отметить шарики в длинных линиях: *masPath[]:= 1*.

Число шариков в *блоке* мы подсчитываем по тому же алгоритму, по которому отыскивали путь в лабиринте и окрашивали клетки поля. После применения алгоритма в массиве *masPath* единицами будут отмечены клетки, образующие блоки.

Таким образом, независимо от типа собираемых фигур, по содержимому массива *masPath* мы легко определим, какие шарики нужно уничтожить и сколько их:

```
  n:= 0;
  //подсчитать число уничтожаемых шариков:
  for j:= 0 to height-1 do
    for i:= 0 to width-1 do
      if masPath[i,j]= 1 then inc(n);
  Result:= n;
  if n= 0 then exit;
```

Если ни одной фигуры не найдено (*n= 0*), то и уничтожать нечего. В противном случае «фигуристые» шарики уничтожаются – также анимированно (5 кадров).

Функция *DeleteFigs* возвращает число шариков в фигуре (оно может быть равно *нулю*, если в результате хода не возникло ни одной фигуры), и мы можем вывести новую *статистику* игры:

```
  //вычислить и показать набранные очки:
  Score.SetValue(n);
```

```
  Score.ShowValue;

  //вычислить и показать игровой тонус:
  Tonus.GetValue;
  Tonus.ShowValue;

  //если был сделан результативный ход и на поле есть шарики,
  //то ждать следующего хода:
  if (n> 0) and (Tonus.EmptyPole= false) then exit;
```

Также мы должны учесть, что по правилам игры после результативного хода (*n> 0*) игрок может сделать ещё один ход – если, конечно, на поле остались пустые клетки:

```
  //если для новых шариков не осталось места, то игра проиграна:
  if Tonus.EmptyCell < NumBalls.nPotom then begin
    GameOver;
    exit
  end;
```

```
  //если поле пустое - победа:
  if Tonus.EmptyPole= true then begin
    GameWin;
    exit
  end;
```

Если игру можно продолжить, то мы выводим на поле новую порцию шариков:

```
  //найти место для новых шариков:
  n:= 0;
  repeat
      i:= random(Width);
      j:= random(Height);
      if Cell[i, j].BallColor= bcNone then begin
        //поместить в пустую клетку «следующий» шарик:
        Cell[i, j].BallColor:= NextBalls[n+1];
        Cell[i, j].new:= true;
        inc(n)
      end;
  until n= NumBalls.nPotom;
  //появление шариков с бульканием:
  CreateBalls;
```

Если и эти шарики образовали с уже имеющимися на поле новые фигуры, их также следует уничтожить:

```
  //если образовалась нужная фигура, уничтожить её:
  n:= DeleteFigs;
```

```
  //вычислить и показать набранные очки:
  Score.SetValue(n);
  Score.ShowValue;

  //вычислить и показать игровой тонус:
  Tonus.GetValue;
  Tonus.ShowValue;
```

В сетке-табло показываем следующие шарики:

```
  //показать следующие шарики:
  InitNextBalls;
  ShowNextBalls;
```

После самоуничтожения фигур поле могло очиститься полностью:

```
  //если поле пустое - победа:
  if Tonus.EmptyPole= true then begin
    GameWin;
    exit
  end;
```

Но могла возникнуть и такая позиция, в которой ходов у игрока нет:

```
  //если ходов нет - поражение:
  if Tonus.FullPole= true then begin
    GameOver;
    exit
  end;
```

Обычно игра рано или поздно заканчивается печально:

```
//ИГРА ЗАКОНЧЕНА
procedure tPole.GameOver;
begin
  form1.imgPor.visible:= true;
  sleep(1000);
  application.ProcessMessages;
  sound('gameover');
  sleep(7500);
  form1.imgPor.visible:= false;
  sleep(1000);
  application.ProcessMessages;
  Score.SetRec;
  form1.NewGame;
end;
```

Это событие сопровождается противной музыкой и обидной надписью (Рис. Ф6.33) (картинка для неё находится в компоненте *imgPor*).



Рис. Ф6.33. Финита ла комедиа!

После чего программа определяет, не стали ли вы новым рекордсменом, и начинает новую игру.

Очень редко кому доведётся на своём веку пережить более счастливый финал (Рис. Ф6.34).

```
//ПОБЕДА!
procedure tPole.GameWin;
begin
  form1.imgPobeda.visible:= true;
  sleep(1000);
  application.ProcessMessages;
  sound('pobeda');
  sleep(7500);
  form1.imgPobeda. visible:= false;
  Score.SetRec;
  form1.NewGame;
end;
```



Рис. Ф6.34. Бывает всякое – это жизнь!

«Победная» картинка загружена в компонент *imgPobeda*.

Эти картинки хранятся в *папке result*.



Исходный код программы находится в папке **Линии**.

«Требую продолжения банкета!», или Где начало того конца, которым оканчивается начало?

Король умер. Да здравствует король!

Круговращение французких королей

Собирание шариков в линии и блоки – процесс сугубо абстрактный, весьма далёкий от насущных потребностей человека, поэтому мы попытаемся придать ему некий рациональный смысл. Например, уничтожение групп шариков по достижении ими «критической массы» можно легко интерпретировать как взаимодействие атомов. Собирание атомов в молекулы - это модные сейчас нано-технологии, то есть процессы, вполне достойные просвещённого ума! Эти примеры трансформирования игры можно продолжать сколь угодно долго, поэтому мы переходим от залихватских фантазий к вполне практическому программированию учебно-воспитательных приложений.

Молекулярный конструктор

Не робей – собирай скорей!

Антинаучный призыв

В *первой* игре мы будем собирать *молекулы* из атомов. Понятно, что между молекулами и блоками шариков принципиальной разницы нет (в смысле игры!), поэтому исправления исходной программы будут минимальны.

В первую очередь, наш конструктор предназначен для школьников, здесь важен сам по себе процесс сборки молекул из атомов, так что все лишние настройки программы вместе с кнопками мы безжалостно удалим, дабы они не отвлекали игрока от сущности бытия, и оставим только самое необходимое для имитации сборки молекул. Прочтите *справочный файл* к программе, из которого легко определить, что именно нам нужно изменить в программе *Линии*:

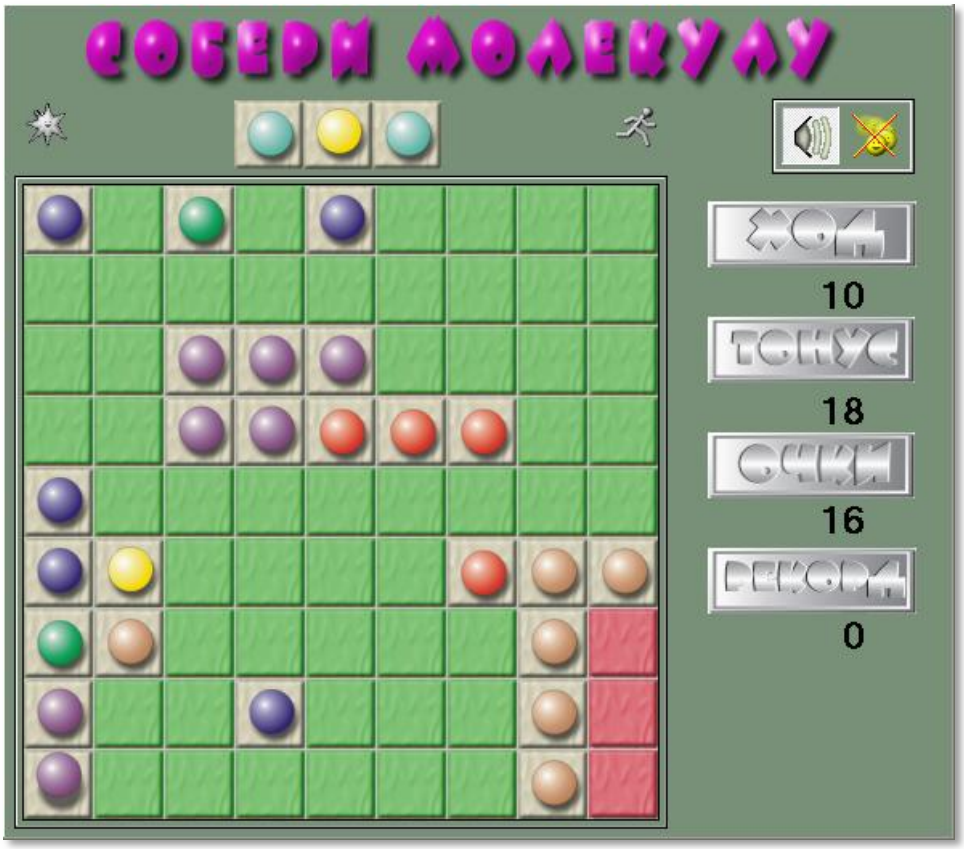
Ваша задача – собирать молекулы, состоящие не менее чем из *шести* атомов одного вида. Атомы в молекуле могут соединяться только по горизонтали и вертикали, но не по диагонали.

Всего имеется 7 разновидностей атомов, представленных в игре шариками разного цвета.

Как только вы соберёте молекулу нужной величины, в ней произойдёт цепная реакция и она исчезнет, а вы за этот научно-технический подвиг получите призовые очки и сможете продолжить сборку (сделать ещё один ход). Чем длиннее получится цепочка, тем больше очков вы за неё получите. В противном случае появятся ещё три новых атома.

Чтобы переместить атом с одного места на другое, щёлкните его мышкой. Он перейдёт в возбуждённое состояние и будет подпрыгивать. Затем щёлкните на той клетке игрового поля, куда бы вы хотели переместить атом. Но имейте в виду, что атом нельзя переместить на уже занятую клетку или на клетку *красного* цвета. Если вы выбрали какой-то атом, а затем передумали, то просто щёлкните на другом атоме – и он станет выбранным.

Число оставшихся у вас ходов (при условии, что вы не сможете больше собрать ни одну молекулу) указывается справа, под надписью *Тонус*. Там же вы найдёте и другую полезную информацию:



Молекулярный конструктор в действии

Игра заканчивается, если на поле не останется места для новых атомов (*проигрыш*) или все атомы будут уничтожены (*выигрыш*).

В любое время вы можете *возобновить* игру, нажав кнопку , а также *закончить* игру, нажав кнопку . Между этими кнопками показана следующая порция атомов.

Вы также можете отключить *звук* с помощью кнопки  и показ пути с помощью кнопки .

Таким образом, нужно исправить только 1 строчку кода:

```
//СОЗДАТЬ ФОРМУ
procedure TForm1.FormCreate(Sender: TObject);
begin
…
  TypeFig:=tfBlocks;
end;
```

Заменить картинки в компоненте *imlPic* (Рис. Ф6.35).

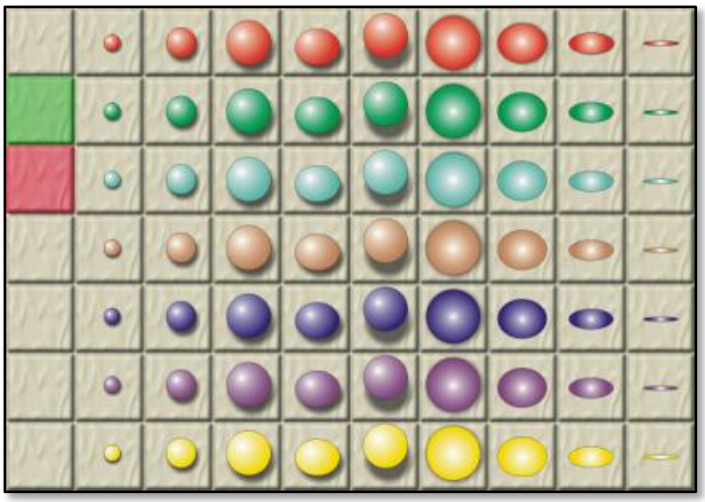


Рис. Ф6.35. Атомы во всей красе!

И, конечно, заголовок окна приложения (Рис. Ф6.36).



Рис. Ф6.36. Название игры



Исходный код программы находится в папке **MakeMol**.

Экологический субботник

Чистота – залог здоровья.

Такая пословица

Чисто не там, где убирают,
а там, где не сорят.

Тоже

Не надо мусорить, да?!

Гринписовский призыв

Если в первой игре мы воспользовались таким свойством игры *Линии* как *собирание* фигур, то во *второй* нам пригодится другое её свойство – мы будем *уничтожать* фигуры, освобождая поле от шариков - так мы будем наводить чистоту в родном городе.

В соответствии с новыми задачами изменим интерфейс программы. Заголовок окна (Рис. Ф6.37).



Рис. Ф6.37. Название игры

Таблички с надписями (Рис. Ф6.38).



Рис. Ф6.38. Статистические таблички

Вместо таблички *Тонус* появятся 2 новые (Рис. Ф6.39).



Рис. Ф6.39. Специфические таблички

Шарики заменит городской *мусор* – огрызок яблока, шляпа, опавшие листья и старый башмак (Рис. Ф6.40).

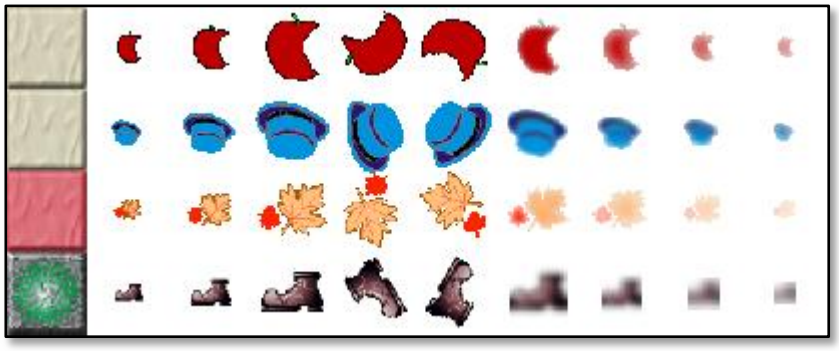


Рис. Ф6.40. Типичная картина!

В игре используются только *три* первых артефакта, иначе выиграть будет затруднительно!

Новые условия игры мы учитываем при создании формы. Сразу после запуска приложения в городе появляются 10 «штук» мусора, а затем – по 3. Разновидностей мусора, как мы только что выяснили, тоже 3. Собирать мусор следует в кучи, имеющие форму *блоков*:

```
//СОЗДАТЬ ФОРМУ
procedure TForm1.FormCreate(Sender: TObject);
begin
…
  //шарики:
  NumBalls.nSrazu:= 10;
  NumBalls.nPotom:= 3;

  //количество цветов:
  NumBalls.nColor:= 3;

  TypeFig:=tfBlocks;
…
end;
```

Поскольку величина поля не изменяется и всегда равна 9 х 9 клеток, то и размер клеток можно *увеличить*, чтобы картинка лучше «смотрелась»:

```
var
  //размеры клетки сетки:
  wCell: integer = 43;
  hCell: integer = 43;
```

Соответственно придётся исправить и *класс tTonus*:

```
  //игровой тонус:
  type tTonus= class
    //число оставшихся ходов:
    value: integer;
    //число пустых клеток на поле:
    EmptyCell: integer;
    //есть ли на поле шарики?
    EmptyPole: boolean;
    //есть ли на поле хотя бы одна свободная клетка?
    FullPole: boolean;
    //процент очищенной площади города:
    Chisto: integer;
    //число клеток с мусором:
    Musor: integer;
    constructor Create;
    procedure ShowValue;
    procedure GetValue;
  end;
```

```
////////////////////////////////////////////////////////
//////////////////  ИГРОВОЙ ТОНУС //////////////////////
////////////////////////////////////////////////////////

constructor tTonus.Create;
begin
  inherited;
  Chisto:= 0;
end;

//ВЫЧИСЛИТЬ ПЛОЩАДЬ
procedure tTonus.GetValue;
var
  i, j: integer;
begin
  //подсчитать число пустых клеток на поле:
  EmptyCell:= 0;
  for j:= 0 to Pole.Height-1 do
    for i:= 0 to Pole.Width-1 do
      if Pole.Cell[i, j].BallColor= bcNone then inc(EmptyCell);

  chisto:= 0;
  for j:= 0 to Pole.Height-1 do
    for i:= 0 to Pole.Width-1 do
      if Pole.masClear[i, j]= true then inc(Chisto);

  value:= Chisto*100 div (Pole.Height*Pole.Width);

  //мусор:
  musor:= Pole.Height * Pole.Width - EmptyCell;
  //проверить, не пусто ли поле:
  if EmptyCell = Pole.Height * Pole.Width then
    EmptyPole:= true
```

```
  else
    EmptyPole:= false;
  //проверить, есть ли на поле свободные клетки:
  if EmptyCell = 0 then
    FullPole:= true
  else
    FullPole:= false;
end;

//ВЫВЕСТИ ЗНАЧЕНИЕ ТОНУСА НА ЭКРАН
procedure tTonus.ShowValue;
begin
  form1.lblTonus.Caption:= inttostr(value)+'%';
  form1.lblTonus.Invalidate;
  form1.lblMusor.Caption:= inttostr(Musor);
  form1.lblMusor.Invalidate;
  application.ProcessMessages
end;
```

В *классе tScore* мы изменим подсчёт набранных очков:

```
////////////////////////////////////////////////////////
////////////////////// ОЧКИ ////////////////////////////
////////////////////////////////////////////////////////

…

//ПОДСЧИТАТЬ НАБРАННЫЕ ОЧКИ
procedure tScore.SetValue(n: integer);
begin
  if hod.value > 80 then dec(value)
  else
    if n> 0 then value:= value + 4*n - 9;

end;

…
```

Фоном для поля послужат 2 фотографии города – чёрно-белая (метафора *грязного* города) и цветная (*чистого*) (Рис. Ф6.41).

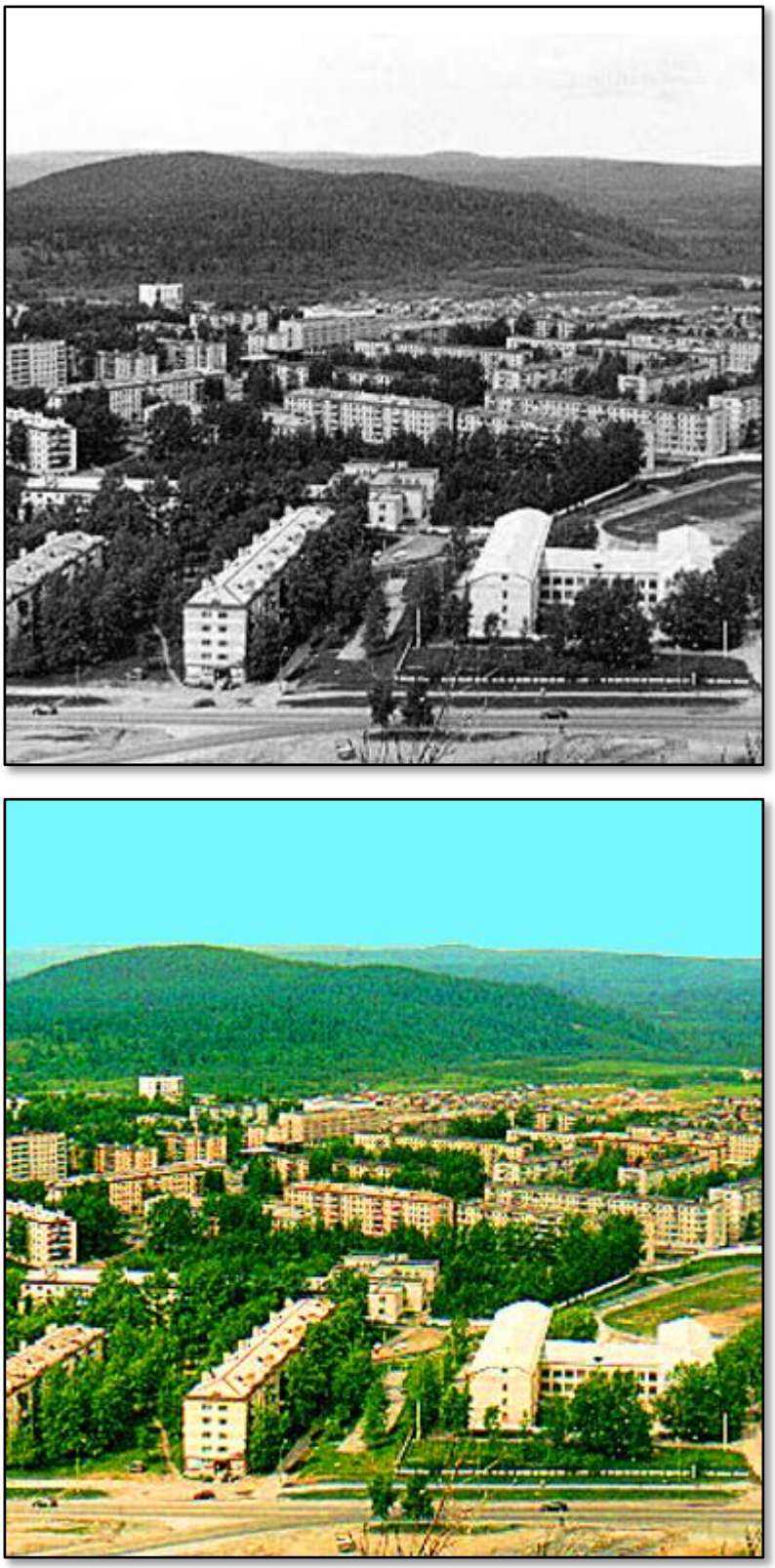


Рис. Ф6.41. Наглядный призыв к чистоте!

Их размеры для поля 9 х 9 клеток и размера клеток 43 х 43 пикселя должны быть 387 х 387 пикселей. Хранить картинки разумно в компоненте **imlGorod***: TImageList* (Ф6.42).

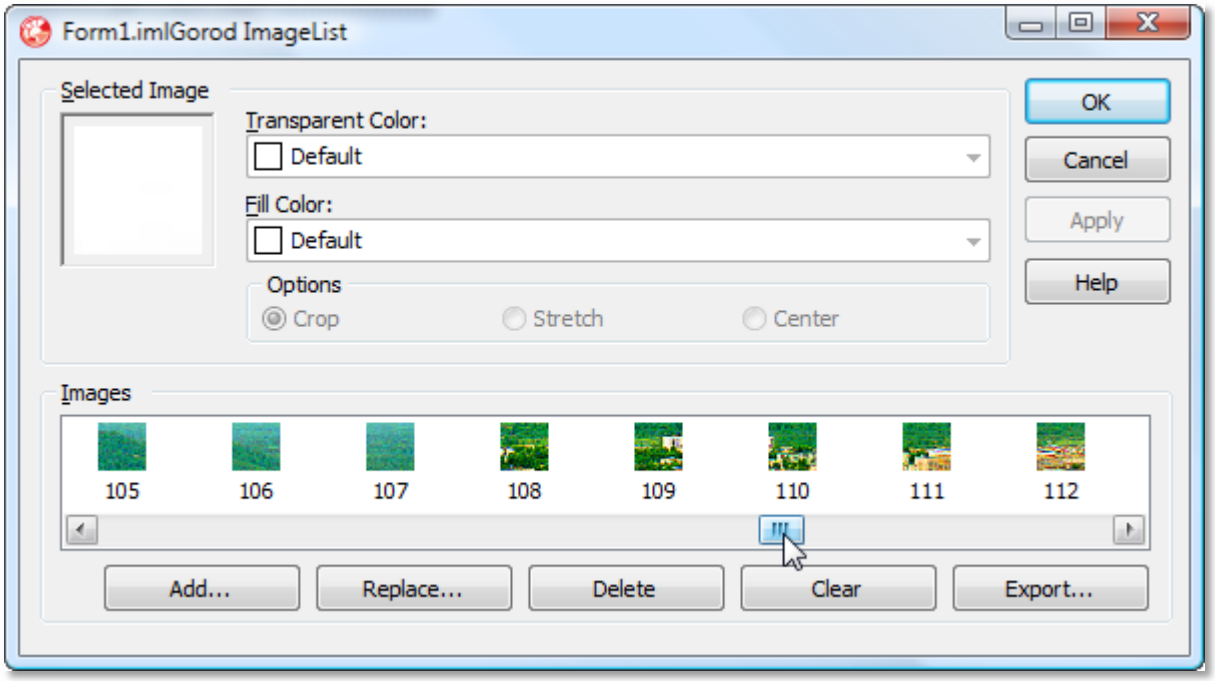


Рис. Ф6.42. Городской фотоальбом

Всего из двух фотографий получится 9 x 9 x 2 = 162 маленьких квадратика. Естественно, фотографии не нужно нарезать вручную – при загрузке их в компонент *TImageList* эта операция выполняется автоматически!

Так как теперь каждая клетка поля имеет *фон* (изменяющийся по ходу игры), на который накладываются картинки с мусором, то процедура отрисовки клеток поля становится более сложной:

```
//ОТРИСОВАТЬ СЕТКУ
procedure TForm1.dgPoleDrawCell(Sender: TObject; ACol, ARow:
Integer;
  Rect: TRect; State: TGridDrawState);
var
  dr: TRECT;
  index: integer;
  bmp: TBitmap;
begin
  //размеры картинок:
  dr:= Bounds(Rect.Left, Rect.Top, wCell, hCell);

  //создать промежуточный растр:
  bmp:= TBitmap.Create;
  bmp.Width:= wCell;
  bmp.Height:= hCell;

  //картинка с шариком в ячейке:
  index:= (ord(Pole.Cell[ACol, ARow].BallColor));

  if index> 0 then begin//- в клетке есть "шарик"
```

```
    index:= GetIndex(ACol, ARow);
    //скопировать картинку с городом в ячейку:
    imlGorod.Draw(bmp.Canvas, 0, 0, index);

    //затем - картинку с "шариком":
    index:= (ord(Pole.Cell[ACol, ARow].BallColor));
    index:= (index-1)* period + 3;
    imlPic.Draw(bmp.Canvas, 0, 0, index);
    dgPole.canvas.Draw(Rect.Left, Rect.Top, bmp);
    end
  else begin            //- пустая клетка
    index:= GetIndex(ACol, ARow);
    //скопировать картинку с городом в ячейку:
    imlGorod.Draw(dgPole.Canvas, dr.left, dr.top, index);
    //отметить доступные и недоступные клетки:
    if Pole.Cell[ACol, ARow].CellColor<> ccGray then begin
      //нарисовать контурный квадрат:
      InflateRect(dr,-3,-3);
      dgPole.Canvas.Brush.Style:= bsClear;
      dgPole.Canvas.Pen.Width:= 1;
      if Pole.Cell[ACol, ARow].CellColor= ccGreen then
        //цвет линий:
         dgPole.Canvas.Pen.Color:= rgb(0,255,0)
      else
         dgPole.Canvas.Pen.Color:= rgb(255,0,0);
      dgPole.Canvas.Rectangle(dr.left, dr.top, dr.right,
dr.bottom);
    end;
  end;
  bmp.Free;
end;
```

Сначала мы создаём в памяти компьютера промежуточный растр *bmp*, затем копируем в него картинку с городом. Тут нужно уметь различать грязные и чистые участки поля:

```
function TForm1.GetIndex(X, Y: integer): integer;
begin
  if not Pole.masClear[x,y] then  //gorod2
    result:= y*9+ x
  else
    result:= y*9+ x+81;
end;
```

Индексы картинок *грязного* поля в *imlGorod* равны 0..80, *чистого* - 81..161.

Если в клетке есть мусор, мы копируем соответствующую картинку поверх фона. Если клетка без мусора, то выводим в неё рамку *зелёного* (доступная клетка) или *красного* (недоступная) цвета (когда выделен какой-либо

«шарик» на поле, естественно!). Скомбинированную картинку с фоном и мусором рисуем в ячейке поля.

Теперь рассмотрим изменения в классе *tPole*. Прежде всего, нужно изменить значение переменной *LenBlocks*, так как минимальный размер блока (кучи) равен трём (иначе «победить» мусор будет очень трудно):

```
////////////////////////////////////////////////////////
////////////////////// ПОЛЕ ////////////////////////////
////////////////////////////////////////////////////////
constructor tPole.Create;
begin
  inherited create;
  LenBlocks:= 3;
  DelayDestroy:= 150;
  DelayCreate:= 80;
  DelayMove:= 40;
end;
```

Подпрыгивающий шарик в этой игре будет крутящимся в вихре мусором:

```
//ПОДПРЫГИВАЮЩИЙ ШАРИК
procedure TPole.Animation;
//x. y - координаты клетки поля;
//index - номер картинки;
var
  dr: Trect;
  x,y, index: integer;

begin
  //все повторения выполнены:
  if (AnimateCell.Rep<> 0) and (AnimateCell.Exec= AnimateCell.Rep)
then
  begin
    form1.timer2.Enabled:= false;
    exit;
  end;
  form1.timer2.Interval:= AnimateCell.Interval;
  x:= AnimateCell.x;
  y:= AnimateCell.y;
  //квадрат клетки:
  dr:= form1.dgPole.CellRect(x,y);
  //город:
  index:= form1.GetIndex(x, y);
  //скопировать картинку с городом в ячейку:
  form1.imlGorod.Draw(form1.dgPole.Canvas, dr.left, dr.top,
index);

  //номер текущей картинки:
  index:= AnimateCell.Index[AnimateCell.ptrIndex];
  //вывести текущую картинку:
```

```
  form1.imlPic.Draw(form1.dgPole.Canvas, dr.left, dr.top, Index);
  if (AnimateCell.sound<> '') and
    (AnimateCell.SoundIndex= AnimateCell.ptrIndex)
  then
    Sound(AnimateCell.sound);

  inc(AnimateCell.ptrIndex);
  if AnimateCell.ptrIndex> AnimateCell.Frame then
    AnimateCell.ptrIndex:= 1;

  if AnimateCell.rep<> 0 then
    inc(AnimateCell.exec);
end;
```

Изменения в этой и следующих процедурах обусловлены необходимостью подкладывать фон под «шарик»:

```
//ПЕРЕМЕСТИТЬ ШАРИК
procedure tPole.DoMoveBall(X, Y: Integer);
//X, Y - координаты клетки
var
  dr: TRECT;
  index, i,j: integer;
  bc: TBallColor;
  p, q: integer;
  ind: integer;
begin
  //убрать выделение с клеток:
  DeactivateBall;
  //создать путь:
  CreatePath(Point(x,y));
  //переместить шарик из начальной клетки в конечную:

if form1.sbtPath.down= true then  //показать путь
  for i:= 0 to masPath[x, y]-1 do
  begin
    //координаты очередной клетки:
    p:= path[i].x; q:= path[i].y;
    //переместить шарик из очередной клетки в следующую -
    //цвет шарика:
    bc:= Cell[p, q].BallColor;
    //очередная клетка становится пустой и серой:
    Cell[p, q].BallColor:= bcNone;

    //исчезновение шарика:
    dr:= form1.dgPole.CellRect(p, q);
    index:= (ord(bc)-1) * period;
    //город:
    //скопировать картинку с городом в ячейку:
    ind:= form1.GetIndex(p, q);
```

```
    form1.imlGorod.Draw(form1.dgPole.Canvas, dr.left, dr.top,
ind);
    //пустая клетка:
    form1.imlGorod.Draw(form1.dgPole.Canvas, dr.left, dr.top,
Ind);

    sleep(DelayMove);

    //в следующей клетке появляется шарик:
    p:= path[i+1].x; q:= path[i+1].y;
    Cell[p, q].BallColor := bc;

    //появление шарика:
    dr:= form1.dgPole.CellRect(p, q);
    //скопировать картинку с городом в ячейку:
    form1.imlGorod.Draw(form1.dgPole.Canvas, dr.left, dr.top,
    form1.GetIndex(p, q));
    form1.imlPic.Draw(form1.dgPole.Canvas, dr.left, dr.top,
index+3);
    sleep(DelayMove);
  end
  else begin
    //координаты первой клетки:
    i:= path[0].x; j:= path[0].y;
    //цвет шарика:
    bc:= Cell[i, j].BallColor;
    //первая клетка становится пустой и серой:
    Cell[i, j].BallColor:= bcNone;
    form1.dgPoleDrawCell(Self, i, j, form1.dgPole.CellRect(i,
j),[]);
    //координаты последней клетки:
    p:= path[masPath[x, y]].x; q:= path[masPath[x, y]].y;
    //в ней появляется шарик:
    Cell[p, q].BallColor:= bc;
    form1.dgPoleDrawCell(Self, p, q, form1.dgPole.CellRect(p,
q),[]);
    Sound('move1');
    sleep(160);
  end;
end;
```

```
//УНИЧТОЖИТЬ ФИГУРУ
function tPole.DeleteFigs: integer;
//возвращает число уничтоженных шариков
var
  i,j,n, index: integer;
  bc: TBallColor;
  dr: TRect;

  //найти на поле блоки нужной величины
  function GetBlocks(x,y: integer; Select: boolean): boolean;
  //x,y - координаты первой клетки
```

```
  var
    masPath2: TField;
    //список координат:
    CoordList: array[1..MAX_WIDTH * MAX_HEIGHT] of TPoint;
    //указатели в списке:
    ptrWrite, ptrRead: integer;
    p, q: integer;
    i, j: integer;

    //проверить координаты
    function TestCoord(x,y: integer): boolean;
    begin
      Result:= true;
      if (x<0) or (x> Width-1) or (y<0) or (y> Height-1)or
       (masPath2[x,y]<> GO) then Result:= false;
    end;

  begin
    //сначала в блоке один шарик:
    n:= 1;
    //отметить в массиве первую клетку:
    if Select then masPath[x,y]:= 1;

    //формируем массив лабиринта:
    for j:= 0 to height-1 do
      for i:= 0 to width-1 do
        //если в клетке есть шарик, то клетка непроходима:
        if Pole.Cell[i,j].BallColor<> bc then
          masPath2[i,j]:= STOP
        //иначе - клетка непроходима:
        else
          masPath2[i,j]:= GO;

    //заносим в список координаты начальной клетки:
    CoordList[1]:= Point(x,y);
    //в начальную клетку в массиве CellPath ставим BEG_CELL= 0:
    masPath2[x,y]:= 0;
    //устанавливаем указатель для считывания координат на начало
списка:
    ptrRead:= 1;
    //устанавливаем указатель для записи новых координат на
следующий индекс:
    ptrWrite:= 2;

    //двигаемся от начала списка к его концу, пока он не кончится:
    while ptrRead < ptrWrite do begin
      //координаты текущей клетки:
      p:= CoordList[ptrRead].x; q:= CoordList[ptrRead].y;
      //проверяем соседние с ней клетки:
      for i:= p - 1 to p + 1 do
        for j:= q - 1 to q + 1 do
          //если нашли соседнюю проходимую клетку,
          if ((i=p) or (j=q)) and TestCoord(i,j) then
```

```
          begin
            inc(n);
            //то записываем в неё 1:
            masPath2[i,j]:= 1;
            if Select then masPath[i,j]:= 1;
            //записываем координаты соседней клетки в конец
списка:
            CoordList[ptrWrite]:= Point(i,j);
            //перемещаем указатель:
            inc(ptrWrite);
          end;
      //переходим к следующей клетке в списке:
      inc(ptrRead);
  end;

    //найдена ли цепочка нужной длины?
    if n>= LenBlocks then Result:= true
    else Result:= false;
  end;

  //уничтожить блок
  procedure DeleteBlocks;
  var
    i,j: integer;
  begin
  //проверить клетки поля:
  for j:= 0 to height do
    for i:= 0 to width do begin
      //цвет очередного шарика:
      bc:= Cell[i,j].BallColor;
      //пустая или проверенная клетка:
      if (bc= bcNone) or (masPath[i,j]= 1) then Continue;
      //искать вправо:
      if GetBlocks(i,j,false) then GetBlocks(i,j,true);
    end;
  end;

begin
  //обнулить массив masPath:
  for j:= 0 to height-1 do
    for i:= 0 to width-1 do
      masPath[i,j]:= 0;

  DeleteBlocks;

  n:= 0;
  //подсчитать число уничтожаемых шариков:
  for j:= 0 to height-1 do
    for i:= 0 to width-1 do
      if masPath[i,j]= 1 then inc(n);
  Result:= n;
  if n= 0 then exit;
```

```
  //уничтожить шарики:
  for n:= 1 to 5 do begin
    for j:= 0 to height-1 do
      for i:= 0 to width-1 do
        if masPath[i,j]= 1 then begin
          masClear[i,j]:= true;
          //исчезновение шарика:
          dr:= form1.dgPole.CellRect(i, j);
          bc:= Cell[i,j].BallColor;
          index:= (ord(bc)-1) * period+n+5;
          if n=5 then begin
            index:= form1.GetIndex(i,j); // - пустая клетка
            form1.imlGorod.Draw(form1.dgPole.Canvas, dr.left,
dr.top, index);
            Cell[i,j].BallColor:= bcNone;
          end
          else
            //скопировать картинку с городом в ячейку:
            form1.imlGorod.Draw(form1.dgPole.Canvas, dr.left,
dr.top, form1.GetIndex(i, j));
            form1.imlPic.Draw(form1.dgPole.Canvas, dr.left,
dr.top, index);
        end;
    sleep(DelayDestroy);
  end;
  Sound('destroy2');
end;
```

```
//ПОЯВЛЕНИЕ ШАРИКОВ:
procedure TPole.CreateBalls;
var
  i, j, index, ind: integer;
  dr: TRECT;
begin
  //r:= form1.GetRect;
  for j:= 0 to Pole.Height-1 do
    for i:= 0 to Pole.Width-1 do begin
      dr:= form1.dgPole.CellRect(i, j);
      index:= (ord(Pole.Cell[i, j].BallColor)-1) * period;
      if Pole.Cell[i, j].new= true then begin
        Pole.Cell[i, j].new:= false;

        //скопировать картинку с городом в ячейку:
        ind:= form1.GetIndex(i, j);
        form1.imlGorod.Draw(form1.dgPole.Canvas, dr.left, dr.top,
ind);

        form1.imlPic.Draw(form1.dgPole.Canvas, dr.left, dr.top,
index+1);
        Delay(DelayCreate);
        form1.imlGorod.Draw(form1.dgPole.Canvas, dr.left, dr.top,
ind);
```

```
        form1.imlPic.Draw(form1.dgPole.Canvas, dr.left, dr.top,
index+2);
        Delay(DelayCreate);

        form1.imlGorod.Draw(form1.dgPole.Canvas, dr.left, dr.top,
ind);
        form1.imlPic.Draw(form1.dgPole.Canvas, dr.left, dr.top,
index+ 3);
        Sound('buljk');
      end;
    end;
end;
```

Необходимо исправить и *процедуру DoHod*, так как изменились правила игры:

```
//ПЕРЕМЕСТИТЬ ШАРИК
procedure tPole.DoHod(X, Y: Integer);
//X, Y - координаты клетки
var
  n, i, j: integer;
begin
…

  //если был сделан результативный ход и на поле есть шарики,
  //то ждать следующего хода:
  if (n> 0) and (Tonus.EmptyPole= false) then exit;

  //вывести следующие шарики NextBalls -

  //если для новых шариков не осталось места, то игра проиграна:
  if Tonus.EmptyCell < NumBalls.nPotom then begin
    GameOver;
    exit
  end;

  //если поле пустое - победа:
  if (Tonus.Chisto= Pole.width*Pole.height) and (Tonus.Musor= 0)
then begin
    GameWin;
    exit
  end;

  //найти место для новых шариков:
  n:= 0;
  repeat
    i:= random(Pole.Width);
    j:= random(Pole.Height);
    if Pole.Cell[i, j].BallColor= bcNone then begin
      //поместить в пустую клетку шарик случайного цвета:
      Pole.Cell[i, j].BallColor:= NextBalls[n+1];
      Pole.Cell[i, j].new:= true;
```

```
      inc(n)
    end;
  until n= NumBalls.nPotom;

…

  //если поле пустое - победа:
  if (Tonus.Chisto= Pole.width*Pole.height) and (Tonus.Musor= 0)
then begin
    GameWin;
    exit
  end;

  //если ходов нет - поражение:
  if Tonus.FullPole= true then begin
    GameOver;
    exit
  end;
end;
```

В этой процедуре новый мусор появляется в тех клетках, где его нет:

```
  //найти место для новых шариков:
  n:= 0;
  repeat
    i:= random(Pole.Width);
    j:= random(Pole.Height);
    if Pole.Cell[i, j].BallColor= bcNone then begin
      //поместить в пустую клетку шарик случайного цвета:
      Pole.Cell[i, j].BallColor:= NextBalls[n+1];
      Pole.Cell[i, j].new:= true;
      inc(n)
    end;
  until n= NumBalls.nPotom;
```

Это значит, что игроку придётся убирать мусор и с уже очищенных участков города. Если вам это кажется несправедливым, вы можете помещать новый мусор только в грязные клетки…

А наградой за победу в игре послужит новая картинка в компоненте *imgPobeda* (Рис. Ф6.43).



Рис. Ф6.43. Заработал – получи грамоту!

Справочный файл к программе:

Ваша задача в этой игре – очистить родной город от мусора. И тогда - сначала грязный и серый – он станет цветным и радостным!

Город разбит на 81 участок (это клеточки на игровом поле). На каждом из них может находиться мусор трёх разных видов, обозначенных огрызком яблока, рваной шляпой и опавшими листьями.

Мусор нужно собирать в кучки. Не менее трёх соседних по вертикали и горизонтали участков должны быть заняты мусором одного вида. Если вам удастся это сделать, то мусор исчезнет, город на этих участках станет чисто-цветным, а вы получите призовые очки. Чем больше размер кучки, тем больше очков за неё вы получите, так что не ленитесь, гребите шибче!

Чтобы переместить мусор с одного участка города на другой, щёлкните на участке с мусором, а затем на «целевом» участке. Но имейте в виду, что между этими двумя участками должен существовать путь, свободный от мусора. В игре все доступные участки выделяются *зелёной* рамкой, недоступные – *красной* (они появляются после выбора участка с мусором). Кроме того, вы не можете всё «валить в одну кучу» - на грязный участок мусор переносить нельзя. Если вы выбрали какой-то участок с мусором, а затем передумали, то просто щёлкните на другом участке – он станет выбранным.

Как только вы соберёте кучку мусора нужной величины, она исчезнет, а вы сможете продолжить свою работу (сделать ещё один ход). В противном случае «свежий» мусор появится ещё на трёх участках города.

Число грязных участков указывается справа, под надписью *Мусор*, процент убранной территории – под надписью *Чисто*:



Если вы не справитесь с задачей за 80 ходов, то дальше призовые очки вам начисляться не будут, а наоборот, вы будете терять по одному очку за каждый лишний ход.

Игра заканчивается, когда город будет полностью очищен от мусора (*Чисто* = 100%) и ни на одном участке не останется мусора (Мусор = 0).

В любое время вы можете начать новую игру, нажав кнопку , а также закончить игру, нажав кнопку . Между этими кнопками показан тот мусор, который благодарные горожане уже приготовили для вас. Он скоро появится, ждите!

Вы также можете отключить *звук* с помощью кнопки  и *показ пути* с помощью кнопки .

Город ждёт – все на экологический субботник!

Твори, выдумывай, пробуй!

Правильный лозунг

На мой взгляд, этот вариант игры не только более «воспитательный», но и гораздо интереснее классического: сыграйте несколько «партий» - вам понравится! Подобным же образом вы можете наполнить новым смыслом и прочие известные игры, а приём с фоном в сетке пригодится вам в других проектах.



Исходный код программы находится в папке **Gorod**.

Факультатив 7. «Что наша «Жизнь»? – Игра!», или Клеточный автомат Джона Конвея

Игру *Жизнь* - самый известный в мире *клеточный автомат* – придумал в 1970 году математик из Кембриджского университета Джон Конвей. Как и все прочие клеточные автоматы, она моделирует развитие некой начальной популяции клеток в соответствии с немногочисленными законами (правилами) их существования.

Законы *Жизни*

Таковы суровые законы жизни.
Или, короче выражаясь,
жизнь диктует нам свои суровые законы.

И. Ильф, Е. Петров,
Золотой теленок

Мир, или вселенная, в которой живут организмы, представляет собой *бесконечную* плоскую сетку, разделённую на квадратики – ячейки, клетки. Мы (или наша программа) в качестве Творца засеваем некоторые ячейки организмами, которые образуют *начальное* (нулевое) поколение. Через некоторое время в популяции происходят такие изменения:

1. Каждая клетка (организм) имеет 8 *соседних* клеток по всем направлениям (вселенная бесконечна!). Если пустая ячейка имеет *трёх* живых соседей, то в ней возникает новая жизнь. Обозначим это правило зарождения жизни так: *B(orn)=3*.
2. Клетка *погибает*, если у неё слишком мало живых соседей (1) или слишком много (4,5,6,7,8).
3. Отсюда следует, что *продолжает жить* только клетка, имеющая 2-х или 3-х соседей. Это правило продолжения жизни записывается так: *S(urvive)=2,3*.

Тут следует заметить, что время во вселенной изменяется дискретно и каждому моменту времени соответствует своя популяция клеток. При формировании следующего поколения учитываются все клетки, уже имеющиеся на поле (живые на этот момент), но не те, которые возникают вновь (рождающиеся в этот момент).

Как видите, правила жизни в клеточном мире очень просты, что отнюдь не мешает достаточно примитивным начальным популяциям образовывать

очень сложные конфигурации клеток. Легко догадаться, что развитие жизни во вселенной определяется только количеством и взаимным расположением клеток в *исходном* состоянии. Найдено множество любопытных начальных структур клеток, но это не должно сдерживать ваши благородные порывы в создании новых миров!

Кроме того, изменение начальной конфигурации клеток существенно зависит от законов существования организмов. Описанные выше правила относятся к *классическому* варианту игры, придуманному Конвеем. Но поскольку каждая клетка может иметь от 0 до 8 соседей, то значения *B* и *S* также изменяются в этом диапазоне, причём *B* и *S* могут иметь и *несколько* значений, что даёт множество сочетаний этих параметров. Конечно, не все они дают интересные результаты, но, возможно, вы найдёте ошеломляющие примеры разнообразия жизни в разных вселенных!

Ну, вот и всё об истории *Жизни*, пора и за дело браться!

Как построить вселенную, или Даём жизни!

Жизнь - это сложная штука,
но эта сложная штука открывается просто,
как ящик.
Надо только уметь его открыть.
Кто не может открыть, тот пропадает.

И. Ильф, Е. Петров,
Двенадцать стульев

Сама вселенная с успехом может быть представлена нашим любимым компонентом *TDrawGrid*. Единственное, что омрачает радость, - *ограниченные* размеры компонента, а, значит, и нашей реализации вселенной. К сожалению, бесконечной сделать её не удастся в любом случае, поэтому протяжённые популяции клеток будут искажаться на границах мира. Избежать этого нельзя, но у нас имеется 2 варианта реакции на краевые эффекты: оставить мир *плоским* и игнорировать все клетки, уходящие за пределы компонента, либо свернуть плоскость в *сферу* и сделать мир «бесконечным», хотя и небольшим. В *первом* случае мы не только потеряем часть популяции, но и рискуем существенно исказить оставшуюся, поскольку исчезнувшие клетки могли бы дать новые организмы в видимой части вселенной. Во *втором* случае клетки, пересекающие границы поля, будут появляться с *противоположной*

стороны и деформировать популяцию самым неожиданным образом. Как говорится, хрен редьки не слаще, но зато у игрока есть выбор…

Максимальные размеры поля в этой версии такие:

```
const
  NAME_PROG = 'ЖИЗНЬ';
  //размеры поля в клетках:
  POLE_WIDTH = 75; // ширина
  POLE_HEIGHT = 54;// высота
```

Вы, конечно, можете сделать поле и б***О***льших размеров, но тогда его придётся прокручивать, чтобы обозреть всё происходящее в мире.

Главное место на форме займёт *сетка-поле*:

```
Name = dgPole
Color = clBlack
ColCount = 75
RowCount = 54
DefaultColWidth = 8
DefaultRowHeight = 8
GridLineWidth = 0
Hint = |Левая кнопка - поставить, правая – убрать
```

Прокомментировать уместно только *цвет* вселенной. Я выбрал *чёрный* не только потому, что и наша Вселенная того же цвета, но – главным образом – потому, что на чёрном фоне другие цвета выглядят более контрастными.

В целом *интерфейс* программы может быть таким (Рис. Ф7.1).

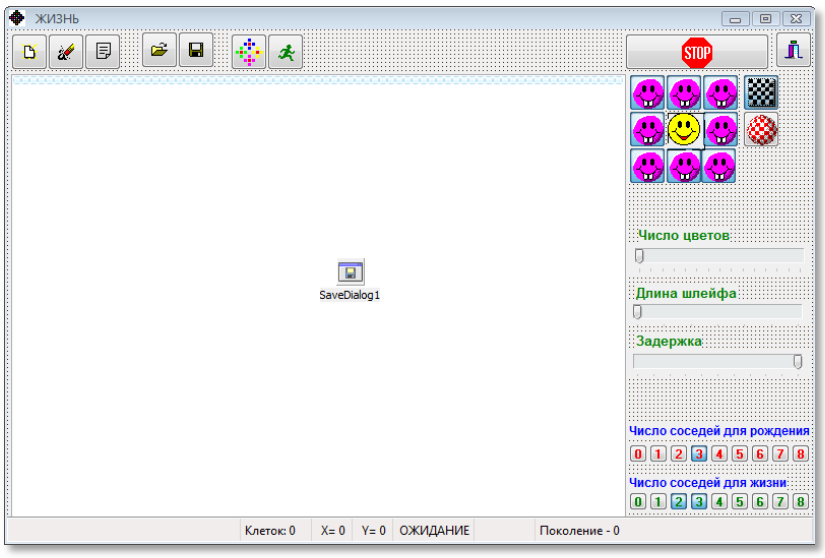


Рис. Ф7.1. Вселенная в начале *Жизни*

А это наше приложение в действии (Рис. Ф7.2)

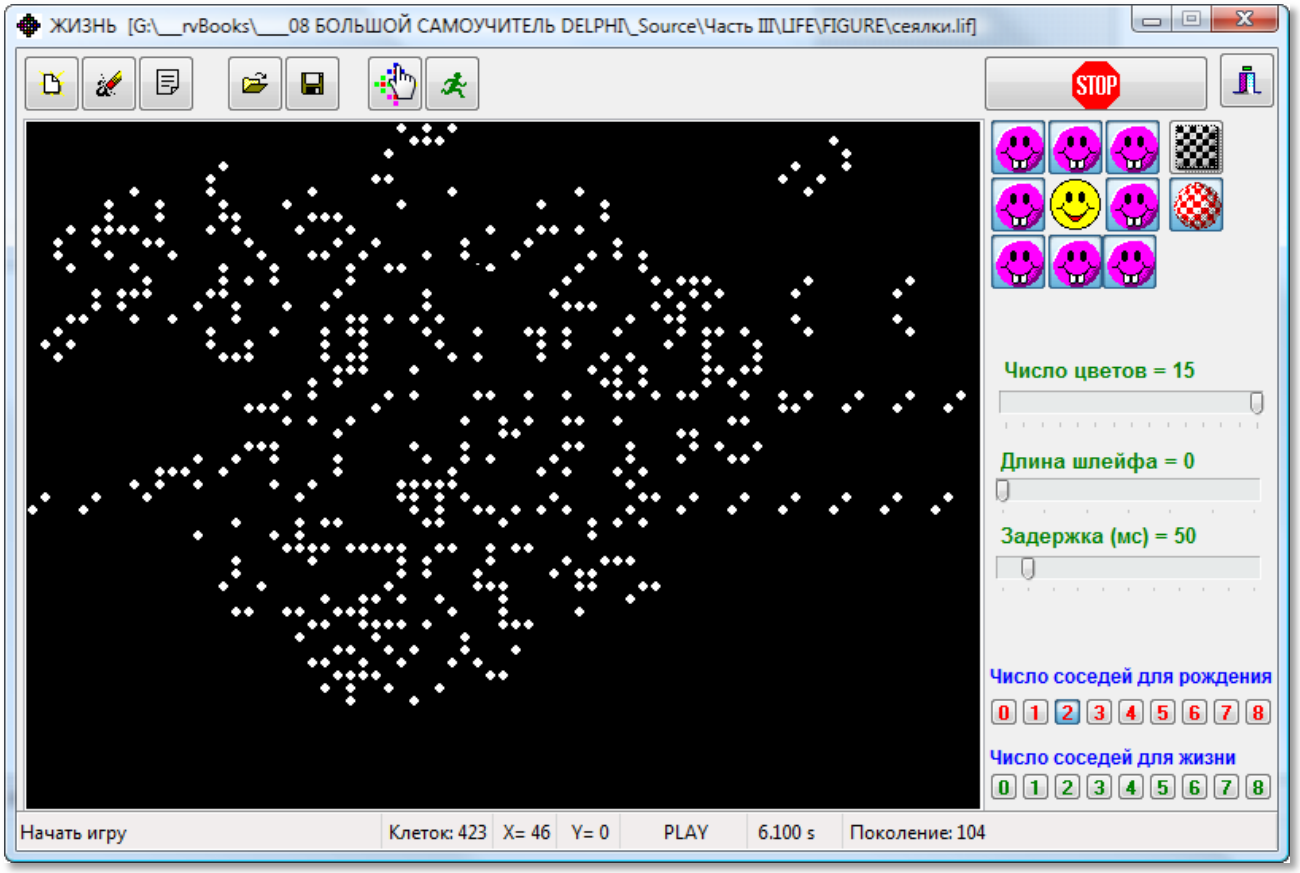


Рис. Ф7.2. Время пошло!

Наш мир возникает вместе с созданием формы приложения:

```
////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////
//                              ФОРМА                         //
////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////
//СОЗДАТЬ ФОРМУ
procedure TForm1.FormCreate(Sender: TObject);
var i: integer;
begin
  //очистить поле:
  ClearPole;
  //загрузить курсоры:
  Screen.Cursors[crKistj]:= LoadCursor(hinstance,'kistj');
  Screen.Cursors[crHand]:= LoadCursor(hinstance,'hand');
  dgPole.Cursor :=crHand;

  RandSeed:=6; //4
  //задать цвета клеток опр. возраста:
  AgeColor[0]:=COLOR_PUSTO;
  AgeColor[1]:=COLOR_KLETKA;
```

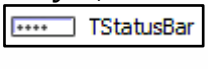
```
  for i:=2 to MAX_COLOR do AgeColor[i]:= RGB(Random(255),
Random(255), Random(255));
  //задать цвета шлейфа:
  tbShleifChange(Self);
  //задать скорость игры:
  tbDelayChange(Self);
  //максимальная продолжительность жизни клеток:
  MAX_AGE:=MAX_COLOR;
  tbColor.max:=MAX_COLOR;
  //задать число цветов клеток:
  tbColorChange(Self);
  //русская раскладка клавиатуры:
  LoadKeyboardLayout('00000419',KLF_ACTIVATE);
end;
```

Сначала он будет абсолютно чист:

```
//ОЧИСТИТЬ ПОЛЕ
procedure TForm1.ClearPole;
var
  i,j: integer;
  hc, wc:  integer;
begin
  if GameStatus='PLAY' then exit;
  nGen:=0;
  statusbar1.Panels[6].text:='Поколение: '+inttostr(nGen);
  hc:= dgPole.DefaultRowHeight;
  wc:= dgPole.DefaultColWidth;
  dgPole.Canvas.Brush.Color:=COLOR_PUSTO;
  For i:= 0 To POLE_WIDTH-1 do
    for j:= 0 to  POLE_HEIGHT - 1 do begin
      dgPole.Canvas.Ellipse (i*wc,j*hc,i*wc+wc,j*hc+hc);
      masPole[i,j]:= NUM_PUSTO;
   end;
   vsegoKl_inFig:=0;
   statusbar1.Panels[1].text:='Клеток: 0';
   cellmouse.x:=-1;
   cellmouse.y:=-1;
end;
```

В переменной *nGen* мы будем хранить номер текущего поколения. Для начальной колонии это значение равно *нулю*:

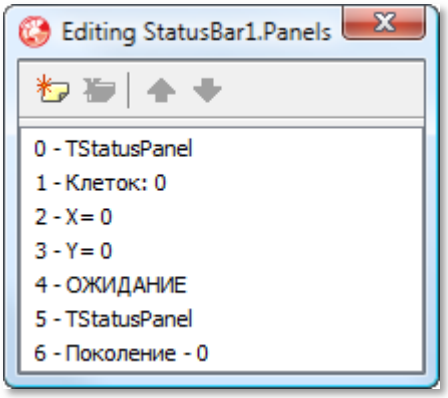
```
//номер поколения:
  nGen: integer=0;
```

Вся информация о текущем состоянии популяции отображается в компоненте *StatusBar1* 

:

AutoHint=True
Height=25

Panels:




```
                        Width=230
Text=Клеток: 0          Width=70
Text=X= 0               Width=40
Text=Y= 0               Width=40
Text=ОЖИДАНИЕ           Width=80
                        Width=60
Text=Поколение - 0      Width=50
```

ShowHint=True

Текстовую информацию очень удобно выводить в нужную *панель* этого компонента. Так как все панели образуют *массив*, то достаточно указать только индекс панели в этом массиве. Например, так указывается номер текущего поколения:

```
statusbar1.Panels[6].text:='Поколение: '+inttostr(nGen);
```

В *нулевой* панели выводятся *подсказки*, которые также появляются возле компонента, когда на нём останавливается курсор мыши. Если вы хотите, чтобы эти подсказки были разными, то напишите их с разделительной чертой: *Очистить*| *Очистить поле*. Подсказка *до* черты будет появляться у компонента, *после* черты – в строке состояния.

В *первой* панели указывается число клеток в текущей популяции. Во *второй* и *третьей* – координаты мышки на поле, что облегчает рисование новых колоний. В *четвёртой* – состояние игры. В *пятой* – время формирования нового поколения. И в последней, *шестой*, - номер текущего поколения.

После вывода информации мы очищаем игровое поле. Для этого в каждой клетке поля кистью цвета

```
dgPole.Canvas.Brush.Color:=COLOR_PUSTO;
```

мы рисуем маленькие *кружки*:

```
dgPole.Canvas.Ellipse (i*wc,j*hc,i*wc+wc,j*hc+hc);
```



Было бы проще *полностью* закрашивать клетки, но в этом случае отдельные «особи» сливаются в одну область и плохо различимы (конечно, их можно отделить друг от друга вертикальными и горизонтальными линиями сетки, но всё равно получится хуже…).

Параллельно мы заполняем массив константой *NUM_PUSTO*, которая соответствует *пустой* клетке поля:

```
masPole[i,j]:= NUM_PUSTO;
```

Объявим *константы*:

```
const
  //цвет пустой клетки на поле:
  COLOR_PUSTO: TColor= clBlack;
  //цвет новой клетки фигуры на поле:
  COLOR_KLETKA: TColor= clWhite;
  //пустая клетка в masPole:
  NUM_PUSTO = 0;
  //чёрная клетка в masPole:
  NUM_KLETKA = 1;
```

Таким образом, клетки без организмов будут *чёрными* (они просто сливаются с чёрным фоном), с организмами – *белые*.

Массив *поля*

```
type TPole = array[-1..POLE_WIDTH,-1..POLE_HEIGHT]of Integer;
//(добавляем по одной клетке по периметру поля)

var
//массивы, в кот. хранится копия поля:
  masPole, masPole2: TPole;
```

мы задаём так, чтобы за его пределами (по периметру) была ещё одна клетка. В игре эти дополнительные клетки не участвуют, но облегчают формирование нового поколения, так как избавляют нас от лишних проверок.

Последние две строки процедуры *TForm1.ClearPole*

```
    cellmouse.x:=-1;
    cellmouse.y:=-1;
```

говорят о том, что курсор мыши находится за пределами поля. Эти координаты используются при рисовании исходной популяции:

```
Var
  //координаты клетки с курсором:
  cellMouse: TPoint;
```

Возвращаемся к процедуре *TForm1.FormCreate*. Чтобы при рисовании курсор больше соответствовал своему назначению, будем изменять его *форму*:

```
  //загрузить курсоры:
  Screen.Cursors[crKistj]:= LoadCursor(hinstance,'kistj');
  Screen.Cursors[crHand]:= LoadCursor(hinstance,'hand');
  dgPole.Cursor:=crHand;
```

Форма курсора на игровом поле в *обычном* режиме (Рис. Ф7.3).



Рис. Ф7.3. Обычный курсор

В режиме *рисования* клеток на игровом поле курсор превращается в кисточку (Рис. Ф7.4).



Рис. Ф7.4. Рисовальный курсор

Нужно также позаботиться о *константах*:

```
const
  //курсоры:
  crHand :integer= 4;
  crStrelka :integer= 5;
  crUkaska: integer= 6;
  crKistj :integer= 7;
```

Все курсоры хранятся в файле ресурсов, который следует подключить к проекту:

```
uses MemoUnit, LoadUnit;

{$R *.DFM}
{$R MyCursors.res}
```

В этой программе используются не все имеющиеся курсоры, так что вы можете выбирать любые!

В классической игре клетки, как и мёд у Винни-Пуха, либо есть, либо нет, но если вам захочется разнообразия, то вы можете раскрашивать клетки в разные цвета:

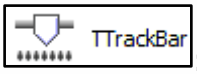
```
  //максимальное число цветов клеток:
  MAX_COLOR=15;
  MAX_SHLEIF= 6;
```

Первая константа задаёт максимальное число цветов живых клеток, *вторая* – мёртвых. В переменной *MAX_AGE*

```
  //максимальная продолжительность жизни клеток:
  MAX_AGE: integer;
```

хранится число цветов живых клеток. Вы только должны иметь в виду, что клетки могут жить вечно, поэтому никаких цветов не хватит, чтобы их раскрасить в разные цвета, и в программе по достижении клеткой возраста *MAX_AGE* она вновь становится «молодой» и окрашивается в цвет *NUM_KLETKA.* И так далее, по кругу.

```
  RandSeed:=6; //4
  //задать цвета клеток опр. возраста:
  AgeColor[0]:=COLOR_PUSTO;
  AgeColor[1]:=COLOR_KLETKA;
  for i:=2 to MAX_COLOR do AgeColor[i]:= RGB(Random(255),
Random(255), Random(255));
  //задать цвета шлейфа:
  tbShleifChange(Self);
  //задать скорость игры:
  tbDelayChange(Self);
  //максимальная продолжительность жизни клеток:
  MAX_AGE:=MAX_COLOR;
  tbColor.max:=MAX_COLOR;
  //задать число цветов клеток:
  tbColorChange(Self);
```

До игры и по ее ходу вы можете изменять количество цветов живых клеток *движком tbColor* 

:

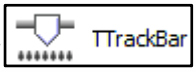
```
Hint=Количество цветов|Максимальный возраст особи
Max=15
Min=1
```

```
////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////
//                          ДВИЖКИ                        //
////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////

//ИЗМЕНИТЬ КОЛИЧЕСТВО ЦВЕТОВ КЛЕТОК
procedure TForm1.tbColorChange(Sender: TObject);
begin
  MAX_AGE:=tbColor.Position;
```

```
  lblColor.Caption:= 'Число цветов = '+inttostr(MAX_AGE);
end;
```

Значение переменной *MAX_AGE* выводится в *метке lblColor*.

Аналогично можно изменять и количество цветов умерших клеток (длину шлейфа) *движком tbShleif* 

:

```
Hint= Шлейф|Ушедшие поколения
Max=6
Min=0
```

```
//ЗАДАТЬ ЧИСЛО ЦВЕТОВ В ШЛЕЙФЕ
procedure TForm1.tbShleifChange(Sender: TObject);
var i,j: integer;
begin
  shleif:=tbShleif.Position;
  if shleif>0 then
    for i:= 1 to shleif do begin
      j:=255 div (Shleif+1) * i;
      AgeColor[-(shleif-i+1)]:= RGB(j,j,j)
    end;
  lblShleif.Caption:= 'Длина шлейфа = '+inttostr(shleif);
end;
```

И последний *движок* - *tbDelay* - управляет скоростью смены поколений на экране:

```
Hint= Задержка|Пауза между поколениями
Max=10
Min=0
```

```
//ЗАДАТЬ ЗАДЕРЖКУ ПРИ ВЫВОДЕ НОВОГО ПОКОЛЕНИЯ
procedure TForm1.tbDelayChange(Sender: TObject);
begin
  Delay:=tbDelay.Position * 50;
  lblDelay.Caption:= 'Задержка (мс) = '+inttostr(Delay)
end;
```

После создания поля на нём нет ни одной живой клетки. Такой мир развиваться не может, и мы должны произвести начальный посев. Сделать это можно двумя способами: *нарисовать* новую популяцию прямо на поле или *загрузить* готовую с диска (несколько заготовок вы найдёте в *папке Figure*).

Начнём с рисования. Нажмём кнопку мышки на поле:

```
//НАЖАТЬ КНОПКУ МЫШКИ НА ПОЛЕ
```

```
procedure TForm1.dgPoleMouseDown(Sender: TObject; Button:
TMouseButton; Shift: TShiftState; X, Y: Integer);
var
  ACol,ARow: integer;
  hc, wc:  integer;
begin
  if GameStatus='PLAY' then exit;
  //изменить форму курсора:
  screen.Cursor:=TCursor(crKistj);
  dgPole.Cursor :=TCursor(crKistj);
  screen.Cursor:=TCursor(crDefault);

  dgPole.MouseToCell(x,y,ACol,ARow);
  //размеры клеток поля:
  hc:= dgPole.DefaultRowHeight;
  wc:= dgPole.DefaultColWidth;
  //если нажата левая кнопка - белая клетка:
  if ssLeft in shift then
  begin
    if masPole[ACol, ARow]= NUM_PUSTO
    then inc(vsegoKl_inFig);
    //занести в массив цвет клетки:
    masPole[ACol, ARow]:=NUM_KLETKA;
    //закрасить клетку:
    dgPole.Canvas.Brush.Color:=COLOR_KLETKA;
    dgPole.Canvas.Ellipse (ACol*wc,ARow*hc,ACol*wc+wc,ARow*hc+hc);
  end;
  //если нажата правая кнопка - чёрная клетка:
  if ssRight in shift then
  begin
    if masPole[ACol, ARow]= NUM_KLETKA
    then dec(vsegoKl_inFig);
    //занести в массив цвет клетки:
    masPole[ACol, ARow]:= NUM_PUSTO;
    //закрасить клетку:
    dgPole.Canvas.Brush.Color:=COLOR_PUSTO;
    dgPole.Canvas.Ellipse (ACol*wc,ARow*hc,ACol*wc+wc,ARow*hc+hc);
  end;
  //теперь клеток в фигуре:
  statusbar1.Panels[1].text:='Клеток: '+inttostr(vsegoKl_inFig);
end;
```

Курсор мышки превратится в кисточку, напоминая о наших намерениях. В клетку *(ACol,ARow)* мы помещаем новый организм - если нажата *левая* кнопка мышки:

```
masPole[ACol, ARow]:=NUM_KLETKA;
```

или убираем организм из клетки, если нажата *правая* кнопка:

```
masPole[ACol, ARow]:= NUM_PUSTO;
```

Одновременно мы рисуем кружки белого или чёрного цвета и подсчитываем число клеток в популяции.

Так можно «высаживать» организмы «поштучно», но иногда требуется засеять большие площади. Для этого, не отпуская кнопку мышки, перемещаем её по полю:

```
//ПЕРЕМЕЩАТЬ МЫШКУ ПО ПОЛЮ - РИСОВАТЬ ФИГУРУ
procedure TForm1.dgPoleMouseMove(Sender: TObject; Shift:
TShiftState; X, Y: Integer);
var
  ACol,ARow: integer;
  hc, wc: integer;
begin
  if GameStatus='PLAY' then exit;
  //размеры клеток поля:
  hc:= dgPole.DefaultRowHeight;
  wc:= dgPole.DefaultColWidth;
  dgPole.MouseToCell(x,y,ACol,ARow);
  statusbar1.Panels[2].text:='X= '+inttostr(ACol);
  statusbar1.Panels[3].text:='Y= '+inttostr(ARow);
  //если нажата левая кнопка - белая клетка:
  if (ssLeft in shift) and ((cellmouse.x<>ACol) or
(cellmouse.y<>ARow))then
  begin
    if masPole[ACol, ARow]= NUM_PUSTO then inc(vsegoKl_inFig);
    //занести в массив цвет клетки:
    masPole[ACol, ARow]:=NUM_KLETKA;
    //закрасить клетку:
    dgPole.Canvas.Brush.Color:=COLOR_KLETKA;
    dgPole.Canvas.Ellipse (ACol*wc,ARow*hc,ACol*wc+wc,ARow*hc+hc);
  end;
  //если нажата правая кнопка - чёрная клетка:
  if (ssRight in shift) and ((cellmouse.x<>ACol) or
(cellmouse.y<>ARow))then
  begin
    if masPole[ACol, ARow]= NUM_KLETKA then dec(vsegoKl_inFig);
    //занести в массив цвет клетки:
    masPole[ACol, ARow]:= NUM_PUSTO;
    //закрасить клетку:
    dgPole.Canvas.Brush.Color:=COLOR_PUSTO;
    dgPole.Canvas.Ellipse (ACol*wc, ARow*hc, ACol*wc+wc,
ARow*hc+hc);
  end;
  //теперь клеток в фигуре:
  statusbar1.Panels[1].text:='Клеток: '+inttostr(vsegoKl_inFig);
end;
```

Процедура рисования живности практически та же самая, только дополнительно мы показываем в строке состояния координаты мышки на поле.

Осталось отпустить кнопку мышки и закончить рисование:

```
//ОТПУСТИТЬ КНОПКУ МЫШКИ
procedure TForm1.dgPoleMouseUp(Sender: TObject; Button:
TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  //изменить форму курсора:
  screen.Cursor:=TCursor(crHand);
  dgPole.Cursor :=TCursor(crHand);
  screen.Cursor:=TCursor(crDefault);
end;
```

Здесь всё понятно «без слов».

Если вам потребуется убрать всю популяцию разом, воспользуйтесь *кнопкой sbtClearPole*  (**Hint**= Очистить| Очистить поле):

```
//КНОПКА Очистить поле
procedure TForm1.sbtClearPoleClick(Sender: TObject);
begin
  ClearPole;
end;
```

Аналогично, только более радикально действует *кнопка sbtNewFig*  (**Hint**=Новая фигура| Новая фигура):

```
//КНОПКА Новая фигура
procedure TForm1.sbtNewFigClick(Sender: TObject);
begin
  if GameStatus='PLAY' then exit;
  //название новой фигуры:
  NameFig:= 'temp';
  caption:= NAME_PROG + '  [' + NameFig + ']';
  //очистить поле:
  ClearPole;
  //стереть комментарии:
  frmMemo.memo1.Clear;
  frmMemo.memo1.SetFocus;
  //номер поколения:
  nGen:=0;
  statusbar1.Panels[6].text:='Поколение: '+inttostr(nGen);
end;
```

Конечно, вам не терпится посмотреть, как же будет эволюционировать нарисованная вами популяция организмов. Что ж, это вполне законное желание. Сейчас мы и займёмся проблемами эволюции, но прежде мы должны позаботиться о настройке параметров окружающей среды, дабы получить именно то, что мы и предвкушаем увидеть.

По умолчанию мир развивается по *классическим* законам, то есть для рождения новой клетки необходимо иметь *трёх* и только трёх соседей, а для продолжения жизни – *двух* или *трёх*. Эти данные уместно хранить в двух массивах:

```
var
  //массив числа соседей,
  //необходимых для рождения клетки:
  Birth: array[0..8] of boolean;
  //массив числа соседей,
  //необходимых для продолжения жизни:
  Life: array[0..8] of boolean;
```

По правилам *Жизни*, *Birth[3]=True, Birth[0..2, 4..8]=False; Life[2,3]=True, Life[0..1, 4..8]=False*. Чтобы игрок мог изменять правила игры, установим на форме по 9 кнопок (Рис. Ф7.5).



Рис. Ф7.5. «Правильные» кнопки

Первый ряд кнопок служит для назначения числа соседей, необходимых для зарождения новых клеток, *второй* – для продолжения жизни уже имеющихся клеток. Нажатые клетки показывают условия эволюции:

sbtBirth0.. sbtBirth8
Hint=Рождение|Рождение новой клетки
AllowAllUp=True
GroupIndex=10..18
Down=False (**sbtBirth3 Down**=True)
Tag=0..8

```
//КНОПКИ Рождение новой клетки
//Число соседей, необходимых для рождения новой клетки
procedure TForm1.sbtBirth0Click(Sender: TObject);
begin
  if (sender as Tspeedbutton).down
  then Birth[(sender as Tspeedbutton).tag]:=true
```

```
  else Birth[(sender as Tspeedbutton).tag]:=false
end;
```

```
sbtLife0.. sbtLife8
Hint=Сохранение|Клетка остаётся живой
AllowAllUp=True
GroupIndex=20..28
Down=False (sbtLife2  sbtLife3 Down=True)
Tag=0..8
```

```
//КНОПКИ Сохранение - Клетка остаётся живой
//Число соседей, необходимых для продолжения жизни
procedure TForm1.sbtLife0Click(Sender: TObject);
begin
  if (sender as Tspeedbutton).down
  then Life[(sender as Tspeedbutton).tag]:=true
  else Life[(sender as Tspeedbutton).tag]:=false
end;
```

Обычно все восемь соседних клеток учитываются при подсчёте живых соседей, однако вы можете самостоятельно назначить соседей для клеток. Установим на форме еще 8 кнопок:

```
Name=sbtSosedLU AllowAllUp=True Down=True GroupIndex=1 Hint=Сосед
Name=sbtSosedU AllowAllUp=True Down=True GroupIndex=2 Hint=Сосед
Name=sbtSosedRU AllowAllUp=True Down=True GroupIndex=3 Hint=Сосед
Name=sbtSosedL AllowAllUp=True Down=True GroupIndex=4 Hint=Сосед
Name=sbtSosedR AllowAllUp=True Down=True GroupIndex=5 Hint=Сосед
Name=sbtSosedLD AllowAllUp=True Down=True GroupIndex=6 Hint=Сосед
Name=sbtSosedD AllowAllUp=True Down=True GroupIndex=7 Hint=Сосед
Name=sbtSosedRD AllowAllUp=True Down=True GroupIndex=8 Hint=Сосед
```

Как видите, первоначально все кнопки *нажаты*, что полностью соответствует классическим правилам игры (Рис. Ф7.6).

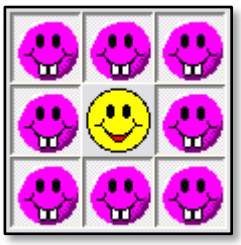


Рис. Ф7.6. «Соседские» кнопки

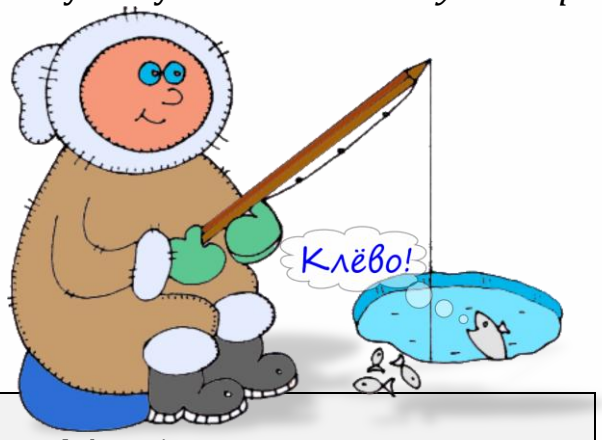
Теперь давайте вспомним об ограниченности мира. В начале главы мы уже говорили об этом и придумали 2 способа конструирования вселенной в игре. Для выбора нужной геометрии пространства нам потребуются две *кнопки-переключателя* (Рис. Ф7.7).

```
Name=sbtFlat Down=True GroupIndex=9 Hint=Плоский мир
Name=sbtSphere Down=False GroupIndex=9 Hint=Сферический мир
```



Рис. Ф7.7. Искажаем пространство!

Популяция развивается хоть и дискретно, но довольно быстро, так что бывает трудно уследить за сменой поколений. Конечно, движком *tbDelay* можно изменить скорость развития жизни, но всё равно за одним поколением сразу же следует другое. Поэтому мы установим *кнопку sbtStep* :




```
AllowAllUp=True
Down=False
GroupIndex=30
Hint=Один шаг|Пошаговая игра
```

Когда эта кнопка *нажата*

```
//КНОПКА Пошаговая игра
procedure TForm1.sbtStepClick(Sender: TObject);
begin
  step:= sbtStep.down
end;
```

переменная *step* равна *TRUE*, и на экран выводится только *одно* следующее поколение.

```
var
//режим вывода поколений:
  //TRUE - пошаговый
  //FALSE - непрерывный
  step: boolean= false;
```

А затем нужно снова нажать кнопку *sbtGo*  (**Hint**=Игра|Начать игру).

```
//КНОПКА Начать игру
procedure TForm1.sbtGoClick(Sender: TObject);
var
  i,j: integer;
  index: integer;
  h, w:  integer;
```

```
time,time0: single;
s: string;
hc, wc: integer;

//сформировать следующее поколение
procedure NewGen;
  begin
    //подсчитать число соседей:
    index:=0;
    //левый верхний сосед:
    if sbtSosedLU.Down and (masPole[i-1,j-1]>0)then inc(index);
    //верхний сосед:
    if sbtSosedU.Down and (masPole[i,j-1]>0)then inc(index);
    //правый верхний сосед:
    if sbtSosedRU.Down and (masPole[i+1,j-1]>0)then inc(index);
    //левый сосед:
    if sbtSosedL.Down and (masPole[i-1,j]>0)then inc(index);
    //правый сосед:
    if sbtSosedR.Down and (masPole[i+1,j]>0)then inc(index);
    //левый нижний сосед:
    if sbtSosedLD.Down and (masPole[i-1,j+1]>0)then inc(index);
    //нижний сосед:
    if sbtSosedD.Down and (masPole[i,j+1]>0) then inc(index);
    //правый нижний сосед:
    if sbtSosedRD.Down and (masPole[i+1,j+1]>0)then inc(index);
  end;

//скроллинг поля
procedure scroll;
var i_s,j_s: integer;
  begin
    //копируем TopRow --> PostLastRow:
    for i_s:=0 to POLE_WIDTH-1 do
      masPole[i_s,POLE_HEIGHT]:=masPole[i_s,0];
    //копируем LastRow --> PreTopRow:
    for i_s:=0 to POLE_WIDTH-1 do //
      masPole[i_s,-1]:=masPole[i_s,POLE_HEIGHT-1];
    //копируем LeftCol --> PostRightCol:
    for j_s:=0 to POLE_HEIGHT-1 do //
      masPole[POLE_WIDTH,j_s]:=masPole[0,j_s];
    //копируем Right --> PreLeftCol:
    for j_s:=0 to POLE_HEIGHT-1 do //
      masPole[-1,j_s]:=masPole[POLE_WIDTH-1,j_s];
    //копируем левую верхнюю угловую клетку:
    masPole[POLE_WIDTH,POLE_HEIGHT]:=masPole[0,0];
    //копируем правую верхнюю угловую клетку:
    masPole[-1,POLE_HEIGHT]:=masPole[POLE_WIDTH-1,0];
    //копируем левую нижнюю угловую клетку:
    masPole[POLE_WIDTH,-1]:=masPole[0,POLE_HEIGHT-1];
    //копируем правую нижнюю угловую клетку:
    masPole[0,0]:=masPole[POLE_WIDTH-1,POLE_HEIGHT-1];
  end;
```

```
begin
  if GameStatus= 'PLAY' then exit;
  //игра началась:
  Outstatus('PLAY');

  //число соседей, позволяющих продолжить жизнь:
  for i:=0 to 8 do
    Life[i]:=(FindComponent('sbtLife'+inttostr(i)) as
TSpeedButton).down;
  //число соседей, необходимых для рождения:
  for i:=0 to 8 do
    Birth[i]:=(FindComponent('sbtBirth'+inttostr(i)) as
TSpeedButton).down;
  //засечь начало игры:
  time0:=GetTickCount();
  //размеры поля:
  h:= POLE_HEIGHT - 1;
  w:= POLE_WIDTH-1;
  hc:= dgPole.DefaultRowHeight;
  wc:= dgPole.DefaultColWidth;

//следующее поколение:
repeat
  inc(nGen);
  statusbar1.Panels[6].text:='Поколение: '+inttostr(nGen);
  vsegoKl_inFig:=0;

  //если поле замкнутое, копируем крайние строки и столбцы:
  if sbtSphere.Down then scroll;

  //просматриваем поле и заполняем вспомогательный массив:
  For j := 0 To h do //- по ширине
    for i := 0 to w do begin //- по высоте
      NewGen;
      if masPole[i,j]>0 then //- живая клетка -> стареет или
умирает
      begin

      if Life[index] then //- продолжает жить
        begin
          inc(vsegoKl_inFig); //- ещё одна клетка на поле
          if masPole[i,j]>=MAX_AGE then
             masPole2[i,j]:= NUM_KLETKA //- постаревшая клетка
          else masPole2[i,j]:=masPole[i,j]+1;
        end
        else //- стареет или умирает -> пустая клетка
        if Shleif>0 then masPole2[i,j]:= -1
        else masPole2[i,j]:= NUM_PUSTO;
        //закрасить клетку соотв. цветом:
        dgPole.Canvas.Brush.Color:= AgeColor[masPole2[i,j]];
        dgPole.Canvas.Ellipse (i*wc,j*hc,(i+1)*wc,(j+1)*hc);
      end
```

```
      else //пустая клетка поля(или клетка шлейфа)
      begin
        if Birth[index] then //- рождается
        begin
          masPole2[i,j]:= NUM_KLETKA; //- новорождённая клетка
          inc(vsegoKl_inFig);
          //закрасить клетку соотв. цветом:
          dgPole.Canvas.Brush.Color:= AgeColor[masPole2[i,j]];
          dgPole.Canvas.Ellipse (i*wc,j*hc,i*wc+wc,j*hc+hc);
        end
        else begin //- стареет или умирает -> пустая клетка
          masPole2[i,j]:= NUM_PUSTO;
          if Shleif>0 then
            if masPole[i,j]<0 then begin
               masPole2[i,j]:= masPole[i,j]-1;
               if masPole2[i,j]<-SHLEIF
               then masPole2[i,j]:= NUM_PUSTO
               else;
               //закрасить клетку соотв. цветом:
               dgPole.Canvas.Brush.Color:=
AgeColor[masPole2[i,j]];
               dgPole.Canvas.Ellipse
(i*wc,j*hc,(i+1)*wc,(j+1)*hc);
            end
            else masPole2[i,j]:= NUM_PUSTO //- пустая клетка ->
цвет не меняет
          else begin  //Shleif=0
            masPole2[i,j]:= NUM_PUSTO;
            if masPole[i,j]<0 then begin  //- стереть следы
              dgPole.Canvas.Brush.Color:= COLOR_PUSTO;
              dgPole.Canvas.Ellipse (i*wc,j*hc,(i+1)*wc,(j+1)*hc)
            end;
          end;
        end;
      end;
  end;
  //копируем вспомогательный массив в основной:
  masPole:= masPole2;

  sleep(Delay);
  //теперь клеток в фигуре:
  statusbar1.Panels[1].text:='Клеток: '+inttostr(vsegoKl_inFig);
  //время игры:
  time:=(GetTickCount()-time0)/1000;
  str(time:6:3,s);
  statusbar1.Panels.Items[5].text:=s + ' s';

  //если была нажата кнопка STOP, то закончить
  //формирование новых поколений:
  application.ProcessMessages;
  if flgExit=true then begin
    flgExit:=false;
    Outstatus('ОЖИДАНИЕ');
```

```
    exit
  end;
until (vsegoKl_inFig=0) or step;
   Outstatus('ОЖИДАНИЕ');
end;
```

Эта процедура, безусловно, самая важная в программе – именно она производит все «манипуляции» с клетками текущей популяции.

Сначала нужно определить, по каким законам будет развиваться жизнь:

```
  //число соседей, позволяющих продолжить жизнь:
  for i:=0 to 8 do
    Life[i]:=(FindComponent('sbtLife'+inttostr(i)) as
TSpeedButton).down;
  //число соседей, необходимых для рождения:
  for i:=0 to 8 do
    Birth[i]:=(FindComponent('sbtBirth'+inttostr(i)) as
TSpeedButton).down;
. . .
  //если поле замкнутое, копируем крайние строки и столбцы:
  if sbtSphere.Down then scroll;
```

Затем в цикле

```
//следующее поколение:
repeat

. . .

until (vsegoKl_inFig=0) or step;

Outstatus('ОЖИДАНИЕ');
```

генерируются новые поколения клеток - до тех пор, пока колония не вымрет. Другая возможность окончания цикла – активный пошаговый режим; в этом случае процедура завершается и переходит в режим ввода команд уже после формирования первого поколения. Вы также можете «насильственно» закончить работу процедуры, если нажмёте *кнопку sbtStop* 

(**Hint** =Остановить|Остановить игру):

```
//КНОПКА Остановить игру
procedure TForm1.sbtStopClick(Sender: TObject);
begin
  if GameStatus<>'PLAY' then exit;
  flgExit:=true
end;
```

В этой процедуре-обработчике переменной *flgExit* присваивается значение *TRUE*:

```
 var
 //=TRUE - остановить игру:
  flgExit: boolean= False;
```

При формировании нового поколения в процедуре *TForm1.sbtGoClick* проверяется эта переменная, и как только она становится равной *TRUE*, процедура завершается:

```
  application.ProcessMessages;
  if flgExit=true then begin
    flgExit:=false;
    Outstatus('ОЖИДАНИЕ');
    exit
  end;
```

Обратите также внимание на переменную *GameStatus:*

```
var
  GameStatus: string='';
```

Она отражает *текущее* состояние программы, которое анализируется при нажатии кнопок. Если программа находится в состоянии ожидания ввода команд, то они выполняются. Если работает игровая процедура, то все попытки вмешательства в эволюцию игнорируются.

Для каждой клетки поля в процедуре *NewGen*

```
  //сформировать следующее поколение
  procedure NewGen;

  . . .
```

мы подсчитываем в переменной *index* количество соседей и поступаем в соответствии с законами жизни. Если *Life[index]=TRUE* и в клетке находится живая особь (*masPole[i,j]>0*), то она продолжает жить, но стареет (клетки от старости не умирают, а только окрашиваются в другой цвет, чтобы было удобнее наблюдать за развитием популяции). В противном случае особь умирает, оставляя после себя пустую, *чёрную* клетку (если игрок задал положительную длину шлейфа, то клетка будет «чернеть» постепенно, хотя умрёт мгновенно; причина здесь та же, что и при старении клеток):

```
  NUM_PUSTO;
  //закрасить клетку соотв. цветом:
  dgPole.Canvas.Brush.Color:= AgeColor[masPole2[i,j]];
  dgPole.Canvas.Ellipse (i*wc,j*hc,(i+1)*wc,(j+1)*hc);
```

Если *Birth[index] =TRUE* и клетка *пустая*, то в ней возникает новая жизнь:

```
  //закрасить клетку соотв. цветом:
  dgPole.Canvas.Brush.Color:= AgeColor[masPole2[i,j]];
  dgPole.Canvas.Ellipse (i*wc,j*hc,i*wc+wc,j*hc+hc);
```

Здесь важно учесть уже умершие клетки, которые остаются на экране в качестве шлейфа. С каждым новым поколением они уходят всё дальше и дальше в небытиё и по достижении установленной длины шлейфа исчезают навсегда, что по-своему очень печально…

И последнее замечание. Поскольку клетки нового поколения рождаются и умирают одновременно, то мы ничего не можем изменить в массиве *masPole[]* с текущим поколением, пока не просчитаем следующее. Поэтому новую популяцию следует формировать в *дополнительном* массиве *masPole2[],* а затем целиком скопировать в основной:

```
  //копируем вспомогательный массив в основной:
  masPole:= masPole2;
```

Как видите, всё достаточно просто, а было бы ещё проще, если бы не раскрашивание стареющих и умерших клеток в разные цвета…

Игровой статус отображается внизу, в строке состояния:

```
////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////
//                    СТРОКА СОСТОЯНИЯ                    //
////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////

procedure TForm1.OutStatus(s:string);
begin
  GameStatus:=s;
  statusbar1.Panels[4].text:= GameStatus
end;
```

Свои соображения по поводу новой фигуры (популяции) вы можете записать в форме *frmMemo* (Рис. Ф7.8).

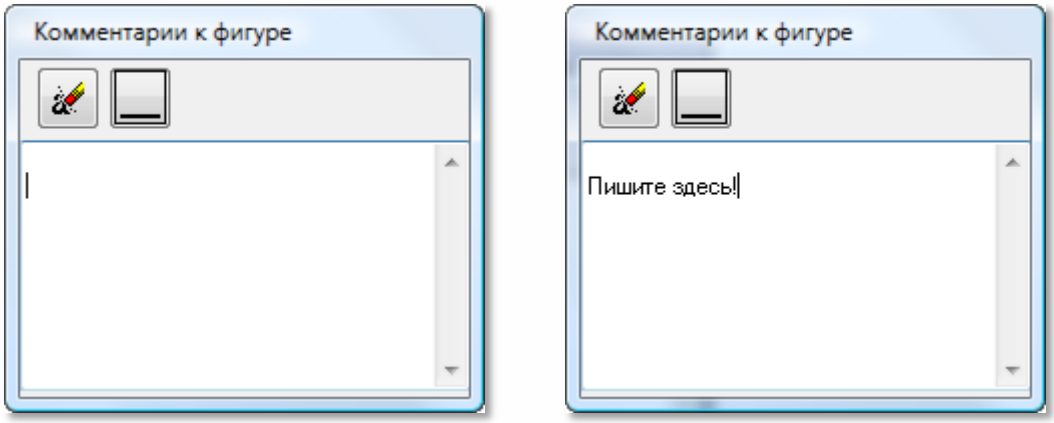


Рис. Ф7.8. «Замечательная» форма

Она появляется справа от главного окна приложения при старте программы. Предполагается, что ваши комментарии будут на русском языке:

```
  //русская раскладка клавиатуры:
  LoadKeyboardLayout('00000419',KLF_ACTIVATE);
```

Эту форму можно свернуть, нажав на ней *кнопку sbtMinimize* (**Hint**=Свернуть|Свернуть окно), а при необходимости вновь возвратить на экран *кнопкой sbtMemo* (**Hint**=Комментарий|Развернуть комментарий к фигуре), которая находится на главной форме приложения (можно и просто дважды щёлкнуть по заголовку свёрнутой формы):

```
//КНОПКА Развернуть комментарий к фигуре
procedure TForm1.sbtMemoClick(Sender: TObject);
begin
  frmMemo.WindowState:=wsNormal
end;
```

Ваши записи хранятся в *компоненте Memo1* TMemo (**Hint**=Комментарий|Комментарии к фигуре) и будут записаны в файл вместе с фигурой.

Кроме возможностей по редактированию текста, предоставляемых самим компонентом, можно добавить ещё и полное стирание текста, которое производится при нажатии на *кнопку sbtClear* (**Hint**=Очистить|Очистить):

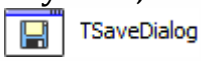
```
implementation

uses lifeunit;
```

```
{$R *.DFM}

procedure TfrmMemo.sbtClearClick(Sender: TObject);
begin
  frmMemo.memo1.Clear;
  frmMemo.memo1.SetFocus
end;

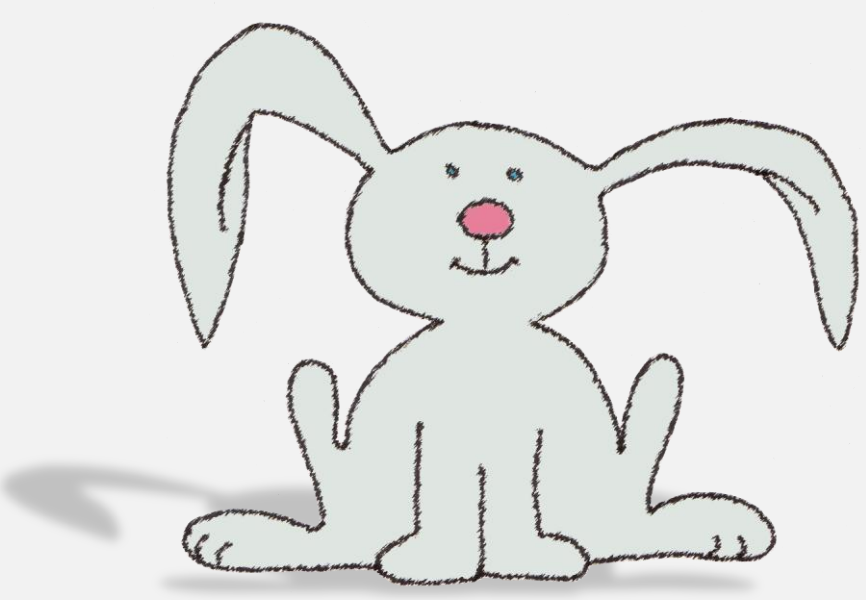
procedure TfrmMemo.sbtMinimizeClick(Sender: TObject);
begin
  frmMemo.WindowState:=wsMinimized
end;
```

Чтобы записать всю информацию о популяции, нужно нажать *кнопку sbtSaveFig*  (**Hint**= Записать|Записать фигуру). Как обычно в таких случаях, мы добавляем для этого на форму *компонент SaveDialog1* 

.

```
//КНОПКА Записать фигуру
procedure TForm1.sbtSaveFigClick(Sender: TObject);
var
  F: textfile;
  fn,s: string;
  i,j: integer;
begin
  if GameStatus='PLAY' then exit;
  savedialog1.DefaultExt:='txt';
  savedialog1.Filter:='Фигуры (*.' + ext + ')|*.' + ext;
  savedialog1.FilterIndex:=1;
  s:=extractfilepath(application.exename)+'FIGURE\';
  savedialog1.InitialDir:= s;
  savedialog1.Title:='Запишите фигуру на диск';
  savedialog1.filename:= NameFig;
  if not savedialog1.Execute then exit;
  fn:= savedialog1.filename; {имя конечного файла}
  NameFig:=fn;
  assignfile(f,fn);
  rewrite(f);
  //записать фигуру:
  writeln (f,vsegoKl_inFig);     //- число клеток в фигуре
  for j:=0 to POLE_HEIGHT-1 do  //- по всем строкам поля
    for i:=0 to POLE_WIDTH-1 do //- по длине строки
      if masPole[i][j]>0 then  begin //- живая клетка -> записать
координаты
        write (f,i); write (f,' '); writeln (f,j);
      end;
  //записать вид вселенной:
  if sbtFlat.down
  then i:=1
```

```
  else i:=0; write (f,i); write (f,' ');
  if sbtSphere.down
  then i:=1
  else i:=0;
  writeln (f,i);
  //записать соседей:
  if sbtSosedLU.down
  then i:=1
  else i:=0;
  write (f,i);
  write (f,' ');
  if sbtSosedU.down
  then i:=1
  else i:=0;
  write (f,i);
  write (f,' ');
  if sbtSosedRU.down
  then i:=1
  else i:=0;
  write (f,i);
  write (f,' ');
  if sbtSosedL.down
  then i:=1
  else i:=0;
  write (f,i);
  write (f,' ');
  if sbtSosedR.down
  then i:=1
  else i:=0;
  write (f,i);
  write (f,' ');
  if sbtSosedLD.down
  then i:=1
  else i:=0;
  write (f,i);
  write (f,' ');
  if sbtSosedD.down then i:=1
  else i:=0; write (f,i);
  write (f,' ');
  if sbtSosedRD.down
  then i:=1
  else i:=0;
  writeln (f,i);
  //записать правило рождения клеток - соседей, необходимых для
рождения:
  for i:=0 to 8 do begin
    if ((FindComponent ('sbtBirth'+inttostr(i)) as
TSpeedButton).down )
    then j:=1
    else j:=0;
    write (f,j);
    write (f,' ');
  end;
```



```
  writeln (f,'');
  //записать правило выживания клеток - соседей, позволяющих
продолжить жизнь
  for i:=0 to 8 do begin
    if ((FindComponent('sbtLife'+inttostr(i)) as
TSpeedButton).down)
    then j:=1
    else j:=0;
    write (f,j);
    write (f,' ');
  end;
  writeln (f,'');
  //записать макс. возраст - количество цветов, обозначающих
возраст:
  writeln (f,tbColor.position );
  //записать шлейф - количество цветов, обозначающих погибшие
поколения:
  writeln (f,tbShleif.position );
  //записать задержку:
  writeln (f,tbDelay.position );
  //записать комментарий к фигуре:
  s:=frmMemo.memo1.Lines.Text;
  if s<>'' then begin
    s:='//'+s;
    writeln (f,s)
  end;
  closefile(f);
  //showmessage(inttostr(i)+' '+inttostr(j));
  //вывести в заголовок название фигуры:
  caption:= NAME_PROG + '  [' + NameFig + ']';
  Refresh ;
  messagebeep(0)
end; //Save
```

По умолчанию данные записываются в *папку Figure* и имеют расширение *lif*:

```
Var
  //расширение файлов с фигурами:
  ext: string='lif';
```

Если текущая популяция ещё не сохранялась на диске и имя для неё не выбрано, то файл будет называться *temp*. Это обычный текстовый файл, поэтому конструировать и редактировать фигуры можно в любом текстовом редакторе. Например, для популяции (Рис. Ф7.9)



Рис. Ф7.9. Образцовая популяция

файл на диске будет таким:



```
70
40 16
. . .
31 35
1 0
1 1 1 1 1 1 1 1
0 0 0 1 0 0 0 0 0
0 0 1 1 0 0 0 0 0
1
0
10
//Тестовая популяция.
```

Сначала указывается общее число всех особей в популяции, а затем координаты всех клеток с живыми организмами. Сохраняется также информация обо всех настройках программы, о которых мы уже подробно говорили. Завершает файл комментарий из *Memo1*. Обратите внимание, что комментарий должен начинаться как и в *Delphi*, с двух слэшей.

Ничего нового и трудного в этой процедуре для вас нет, но вы должны иметь в виду, что на диске сохраняется текущая популяция из массива *masPole [].* Вероятно, не менее важно сохранить начальную популяцию и номер текущего поколения. Для этого следует завести ещё один массив того же типа, что и *masPole*:

```
type TPole = array[-1..POLE_WIDTH,-1..POLE_HEIGHT]of Integer;
```

В начале процедуры *TForm1.sbtGoClick* нужно скопировать массив с исходной популяцией в *новый* массив. Но здесь есть одна тонкость: в пошаговом режиме этот массив будет переписываться, так что, если вас интересует именно самая первая популяция, то заведите ещё и логическую переменную, которая будет сигнализировать о том, что массив с исходной популяцией уже сформирован.

Теперь разберём обратную процедуру – *загрузку* ранее сохранённой фигуры из файла. Для этого служит *кнопка sbtLoadFig*  (**Hint**= Загрузить | Загрузить фигуру):

```
//КНОПКА Загрузить фигуру
procedure TForm1.sbtLoadFigClick(Sender: TObject);
var
  s, s2: string;
  found: integer;
  i:integer;
  SearchRec: TSearchRec;
begin
  if GameStatus='PLAY' then exit;
  //сформировать список файлов в папке FIGURE:
  s:=extractfilepath(application.exename)+'FIGURE\*.'+ ext;
  found:= FindFirst (s,faAnyFile,SearchRec);
  frmLoad.listbox1.items.Clear;
  while found=0 do begin
    s:=SearchRec.name;
    s2:='';
    i:=1;
    while (s[i]<>'.') and (i<=length(s)) do begin
      s2:=s2+s[i];
      inc(i)
    end;
    frmLoad.listbox1.items.add(s2);
    found:= FindNext (SearchRec)
  end;
  FindClose (SearchRec);
  nameFig:='';
  //показать форму со списком фигур:
  frmLoad.showmodal;
  if nameFig='' then exit; //фигура не выбрана

  //показать в заголовке название выбранной фигуры:
  form1.caption:= NAME_PROG + '  [' + NameFig + ']';
  //в информационной строке вывести число клеток в фигуре:
  statusbar1.Panels[1].text:='Клеток: '+inttostr(vsegoKl_inFig);
  //скопировать комментарии к загруженной фигуре:
  frmMemo.memo1.Lines.Text:=frmLoad.memo1.Lines.Text;
  //поколение 0:
  nGen:=0;
  statusbar1.Panels[6].text:='Поколение: '+inttostr(nGen);
  OutPole;
end;//sbtLoadFigClick
```

Очень просто загрузить фигуру на поле, но как выбрать нужную из множества сохранённых на диске? Постараемся облегчить игроку

проблему выбора. Для этого нам потребуется новая *форма frmLoad* (Рис. Ф7.10).

```
BorderStyle=bsSingle
Caption=  Выберите фигуру
Position=poMainFormCenter
```

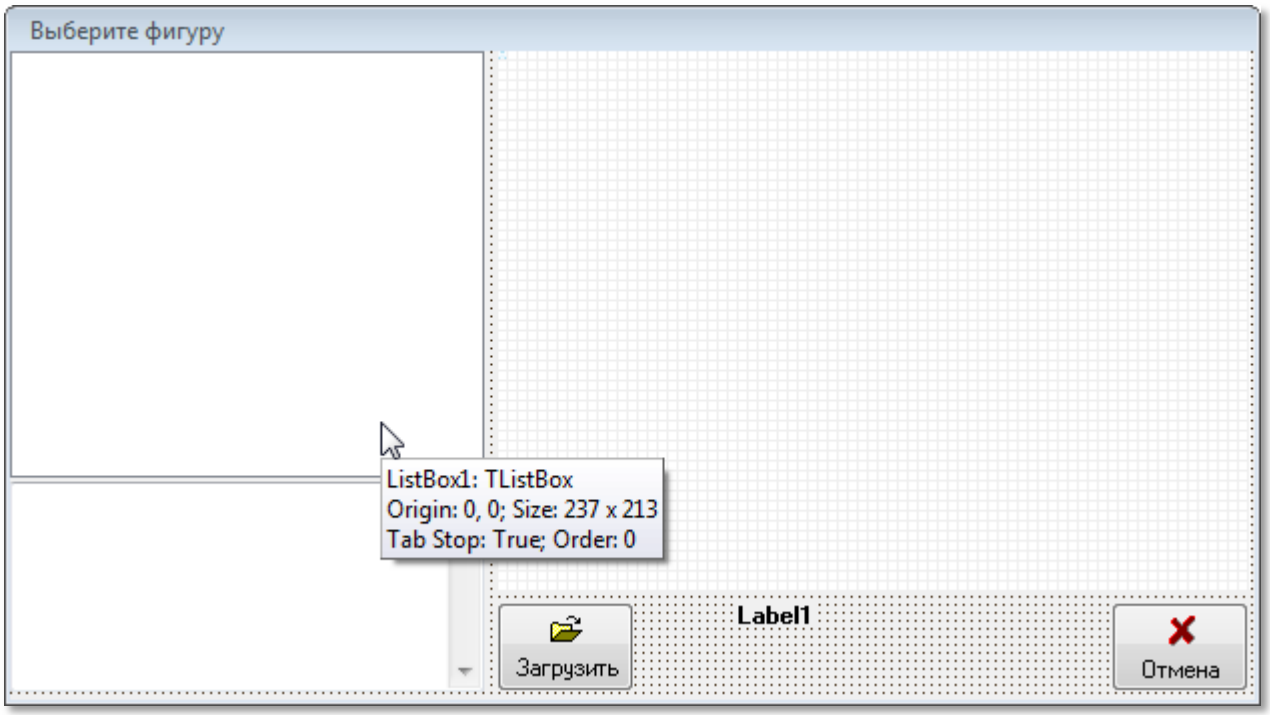


Рис. Ф7.10. Загрузочная форма

На ней мы расположим необходимые *элементы управления*.

В список *ListBox1* мы будем добавлять в алфавитном порядке (*Sorted=True*) названия всех файлов из *папки FIGURE*, имеющих заданное расширение.

В компоненте *Memo1* (*ScrollBars=ssVertical*) мы будем выводить *комментарий* к выбранной фигуре, подобно тому, как мы это делали в главном окне.

То же самое относится и к сетке *dgPole*, которая полностью аналогична сетке, отображающей игровое поле, только размер клеток у неё вдвое меньше:

```
DefaultColWidth=4
DefaultRowHeight=4
```

Метка *Label1* показывает число клеток в фигуре.
Кнопка *sbtLoad* (**Caption**=Загрузить) загружает выбранную фигуру.

Кнопка *sbtCancel* (**Caption**=Отмена) возвращает в вызывающую процедуру пустое имя файла – новая фигура не загружена!

```
//ОТМЕНИТЬ ЗАГРУЗКУ ФИГУРЫ
procedure TfrmLoad.sbtCancelClick(Sender: TObject);
begin
  nameFig:= '';
  close
end;
```

```
if nameFig='' then exit; //фигура не выбрана
```

После того как процедура *TForm1.sbtLoadFigClick* составит список всех файлов с фигурами:

```
frmLoad.listbox1.items.add(s2);
```

на экране появляется форма *frmLoad*:

```
  nameFig:='';
  //показать форму со списком фигур:
  frmLoad.showmodal;
  if nameFig='' then exit; //фигура не выбрана
```

В ней и нужно выбрать нужный файл из списка (Рис. Ф7.11).

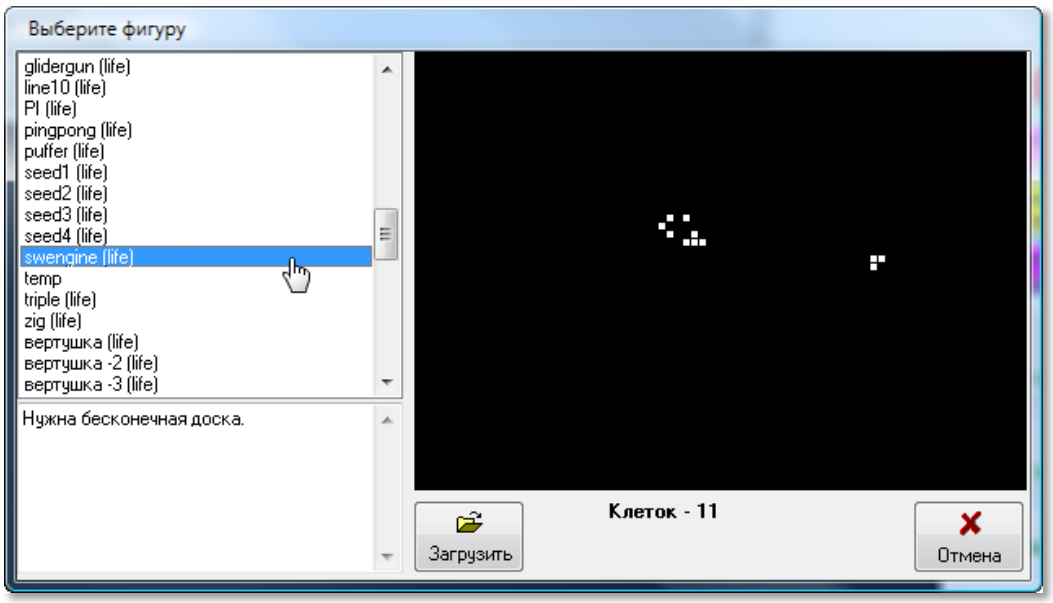


Рис. Ф7.11. Выбираем исходное поколение

Для этого, как обычно, нужно нажать *левую* кнопку мыши на названии файла:

```
//ВЫБРАТЬ ФИГУРУ ИЗ СПИСКА
```

```
procedure TfrmLoad.ListBox1MouseDown(Sender: TObject; Button:
TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  LifeUnit.nameFig:=
frmLoad.ListBox1.Items[listbox1.ItemIndex]+'.'+ ext;
  ViewFig;
end;
```

После чего новой фигуре присваивается имя выбранного файла, а сама фигура изображается на поле:

```
//ПОКАЗАТЬ ВЫБРАННУЮ ФИГУРУ
procedure ViewFig;
var
  s: string;
  F: TextFile;
  i: integer;
  x, y: integer;
  area: tRect;
begin
  s:=extractfilepath(application.exename)+'FIGURE\';
  NameFig:=s+nameFig; // - полный путь к фигуре
  {$i-}
  AssignFile(F,NameFig);
  Reset(F);
  {$i+}
  if IOResult<>0 then begin   {ошибка при загрузке файла}
    application.MessageBox ('Такой фигуры нет!',NAME_PROG, MB_OK);
    exit
  end;

  //очистить поле:
  ClearPole;
  //и комментарии:
  frmLoad.memo1.Clear;
  //считать количество клеток в фигуре:
  Readln(F, vsegoKl);
  //показать на экране:
  frmLoad.label1.Caption:= 'Клеток - ' + inttostr(vsegoKl);

  //считать фигуру:
  for i:=1 to vsegoKl do begin
    //считать координаты очередной клетки:
    Readln(F, x, y);
    //занести их в массив:
    masPole2[x,y]:=NUM_KLETKA;
    //и вывести на поле:
    frmLoad.dgPole.Canvas.Brush.Color:=COLOR_KLETKA;
    area:=frmLoad.dgPole.CellRect(x,y);
    frmLoad.dgPole.Canvas.FillRect(area);
  end;
```

```
  //считать вид вселенной:
   Readln(f,Flat,Sphere);
  //считать соседей:

Readln(f,SosedLU,SosedU,SosedRU,SosedL,SosedR,SosedLD,SosedD,Sosed
RD);
  //считать правило рождения клеток -
  //соседей, необходимых для рождения:
   for i:=0 to 8 do Read(f,Birth[i]);
  //считать правило выживания клеток -
  //соседей, позволяющих продолжить жизнь:
  for i:=0 to 8 do Read(f,Life[i]);
  //считать макс. возраст:
  Readln(f,Age);
  //считать шлейф - количество цветов,
  //обозначающих погибшие поколения:
  Readln(f,_shleif);
  //считать задержку:
  Readln(f,Delay);

  //считать комментарий к фигуре:
  s:='';
  while not eof(f) do begin
    Readln(F, S); //- считать строку из файла
    //комментарий к фигуре?
    if ((length(s)>1) and (s[1]='/') and (s[2]='/'))
    then begin //- комментарий
      s:=copy(s,3,length(s));
      frmLoad.memo1.Lines.Text:=s
    end
    else
     frmLoad.memo1.Lines.Text:=frmLoad.memo1.Lines.Text+#10+s
  end;
  //закрыть файл:
  closefile(f);
end;
```

Процедура очистки поля от предыдущей фигуры:

```
//ОЧИСТИТЬ ПОЛЕ
procedure ClearPole;
var
  i,j: integer;
begin
  frmLoad.dgPole.Canvas.Brush.Color:=COLOR_PUSTO;
  frmLoad.dgPole.Canvas.FillRect(frmLoad.dgPole.ClientRect );
  For i := 0 To POLE_WIDTH-1 do
    for j := 0 to  POLE_HEIGHT - 1 do
      masPole2[i,j]:=NUM_PUSTO;
  //клеток в фигуре:
  vsegoKl:=0;
```

```
  frmLoad.label1.Caption:= 'Клеток - 0';
end;
```

Мы подробно разобрали процесс сохранения файла, поэтому загрузка файла не должна вызвать у вас вопросов. Заметьте только, что клетки фигуры загружаются во вспомогательный массив *masPole2*:

```
var
  masPole2: TPole;
  vsegoKl: integer;
  Flat, Sphere: integer;//boolean;
  SosedLU,SosedU,SosedRU,SosedL,SosedR,SosedLD,SosedD,SosedRD:
integer;
  Birth: array[0..8] of integer;
  Life: array[0..8] of integer;
  Age: integer;
  Delay: integer;
  _shleif: integer;
```

Далее вы можете выбрать в списке любой другой файл, отменить загрузку новой фигуры или загрузить выбранную фигуру, если она вам понравилась:

```
//ЗАГРУЗИТЬ ФИГУРУ
procedure TfrmLoad.sbtLoadClick(Sender: TObject);
var i: integer;
begin
  //число клеток в фигуре:
  vsegoKl_inFig:=vsegoKl;
  //скопировать массив поля с выбранной фигурой:
  masPole:=masPole2;
  //установить вид вселенной:
  with Form1 do begin
    if Flat=1 then begin
       sbtFlat.down:= true;
       sbtSphere.down:= false
    end
    else begin
       sbtFlat.down:= false;
       sbtSphere.down:= true
    end;
    //установить соседей:
    if SosedLU=1
      then sbtSosedLU.down:= true else sbtSosedLU.down:= false;
    if SosedU=1
      then sbtSosedU.down:= true else sbtSosedU.down:= false;
    if SosedRU=1
      then sbtSosedRU.down:= true else sbtSosedRU.down:= false;
    if SosedL=1
      then sbtSosedL.down:= true else sbtSosedL.down:= false;
    if SosedR=1
```

```
      then sbtSosedR.down:= true else sbtSosedR.down:= false;
    if SosedLD=1
      then sbtSosedLD.down:= true else sbtSosedLD.down:= false;
    if SosedD=1
      then sbtSosedD.down:= true else sbtSosedD.down:= false;
    if SosedRD=1
      then sbtSosedRD.down:= true else sbtSosedRD.down:= false;
    //установить правило рождения клеток -
    //число соседей, необходимых для рождения:
    for i:=0 to 8 do
      if Birth[i]=1
        then (FindComponent ('sbtBirth'+inttostr(i)) as
TSpeedButton).down:= true
        else (FindComponent ('sbtBirth'+inttostr(i)) as
TSpeedButton).down:= false;
    //установить правило выживания клеток -
    //число соседей, позволяющих продолжить жизнь:
    for i:=0 to 8 do
      if Life[i]=1
        then (FindComponent ('sbtLife'+inttostr(i)) as
TSpeedButton).down:= true
        else (FindComponent ('sbtLife'+inttostr(i)) as
TSpeedButton).down:= false;
    //установить макс. возраст -
    //количество цветов, обозначающих возраст:
    tbColor.position:= Age;
    tbColorChange(Self);
    //установить шлейф - количество цветов,
    //обозначающих погибшие поколения:
    tbShleif.position:=_shleif;
    tbShleifChange(Self);
    //установить задержку:
    tbDelay.position:= Delay;
    tbDelayChange(Self);
  end;
  close
end;
```

В этой процедуре копируется игровое поле и значение переменных игры, а также устанавливаются в нужное положение элементы управления главного окна приложения в соответствии с их значениями в момент сохранения файла на диске.

Затем в заголовке окна выводится имя новой фигуры и другая информация:

```
  //показать в заголовке название выбранной фигуры:
  form1.caption:= NAME_PROG + '  [' + NameFig + ']';
  //в информационной строке вывести число клеток в фигуре:
  statusbar1.Panels[1].text:='Клеток: '+inttostr(vsegoKl_inFig);
  //скопировать комментарии к загруженной фигуре:
```

```
  frmMemo.memo1.Lines.Text:=frmLoad.memo1.Lines.Text;
  //поколение 0:
  nGen:=0;
  statusbar1.Panels[6].text:='Поколение: '+inttostr(nGen);
  OutPole;
```

Вы можете отредактировать фигуру, записать её на диск, изменить параметры игры и, наконец, запустить её эволюцию.

Есть ли жизнь на Марсе...
нет ли жизни на Марсе...
это науке неизвестно!

Комедия *Карнавальная ночь*

А теперь вернёмся в основную программу, чтобы нажать *кнопку sbtExit*  (**Hint**= Выход|Выход из программы),

```
//КНОПКА Выход из программы
procedure TForm1.sbtExitClick(Sender: TObject);
begin
  close
end;
```

чтобы закрыть её, а вместе с ней и последнюю страницу нашей книги о программировании игр и головоломок.



Исходный код программы находится в папке **LIFE**.

Факультатив 8. Бонусы, или Хитрая головоломка – хитори

Седьмого факультативного занятия не было в исходной рукописи, но *хитори* - это моя любимая интеллектуальная забава, о которой я даже написал книгу **[PB10]**. Надеюсь, она не разочарует и вас: хитори - одна из самых популярных японских головоломок в мире и одна из немногих, которую интересно решать и с карандашом в руке. По «решабельности» она значительно превосходит и пресловутое судоку, и какуро (*лично для меня*, как пел известный Перепелица).

Игровое поле в хитори – всегда квадратное и имеет размеры от 4 x 4 до 17 x 17 клеток (и даже больше). В каждой клетке записано число – от 1 до размерности поля (то есть при размерах поля 9 x 9 клеток это будут числа от 1 до 9, как в судоку). В некоторых строках и столбцах числа могут повторяться (Рис. Ф8.1). Таким образом, хитори – близкая родственница латинских квадратов, диаго и судоку.

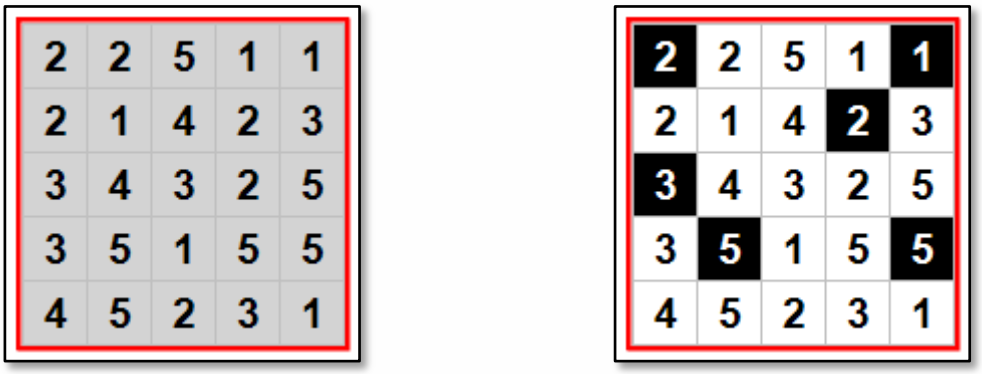


Рис. Ф8.1. Нерешённая задача Решённая задача

Цель решения хитори заключается в том, чтобы вычеркнуть *повторяющиеся* числа (или, иначе, закрасить клетку *черным* цветом) так, чтобы в каждой строке и каждом столбце все числа остались в одиночестве (то есть не повторялись). При этом в процессе решения задачи должны соблюдаться правила. А правил всего три:

Правило 1. Числа в *белых* клетках не должны повторяться ни в одной строке и ни в одном столбце. Если в *серых* клетках числа повторяются, то часть из них должна быть вычеркнута, то есть клетка должна стать *черной*.

Правило 2. Черные клетки не могут соприкасаться сторонами – только углами. На Рис. 7.1, справа это условие соблюдается.

Правило 3. Белые клетки должны образовать *связную* область. Это значит, что из любой белой клетки должен существовать путь в любую другую белую клетку, проходящий через *стороны* белых клеток.

Чтобы различать клетки разного смысла, мы будем окрашивать их в разные цвета. *Серые* клетки – неразгаданные. На Рис. 7.1, слева мы видим задачу в исходном состоянии, когда ни одна клетка не разгадана.

Разгаданные клетки могут быть *белыми* и *черными*. В черных клетках находятся *вычеркнутые* числа, в белых – *невычеркнутые*, сохраненные. На Рис. 7.1, справа показана полностью решенная задача – все клетки белого и черного цвета.

Программированию хитори я отвел целую главу в книге ***Как решать*** *головоломные задачи на компьютере*. В ней я использовал язык *Си-шарп*, а нас сейчас куда больше интересует *Delphi*. Но поскольку изначально программа была написана на *Delphi* и именно она хорошо показала себя при подготовке книги, то я хочу предложить вам две ее разновидности. *Первая - rvHitori-Solver* – хорошо умеет *решать* задачи, но и только (Рис. Ф8.2).

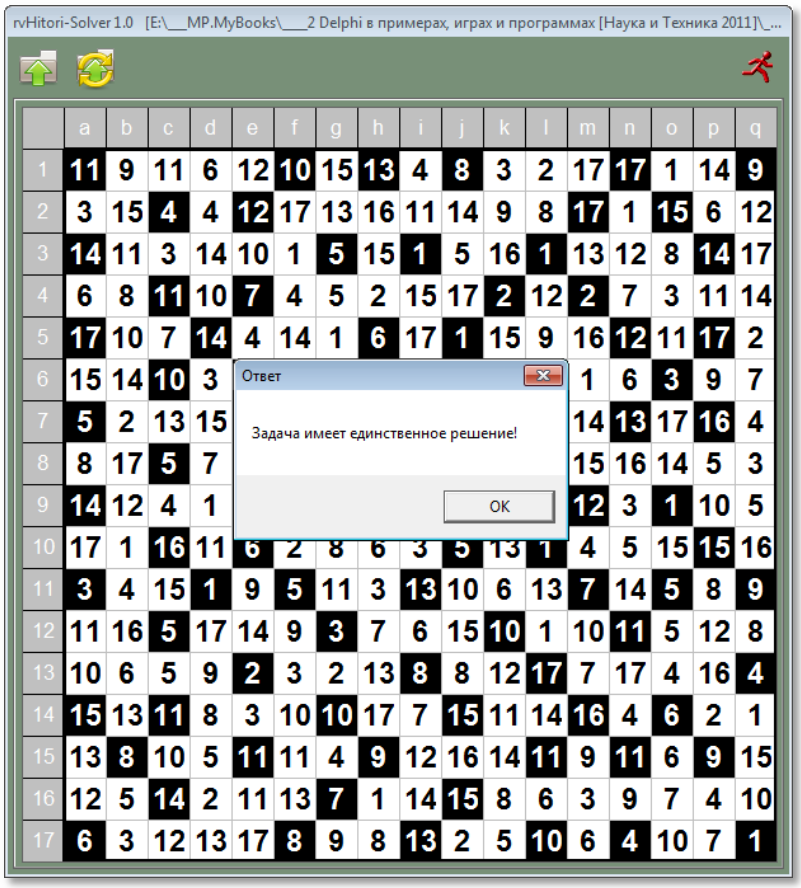


Рис. Ф8.2. *rvHitori-Solver* решает задачу

Вторая – *rvHitori* – не только решит предложенные ей задачи, но и поможет вам *составлять* свои собственные (Рис. Ф8.3)!

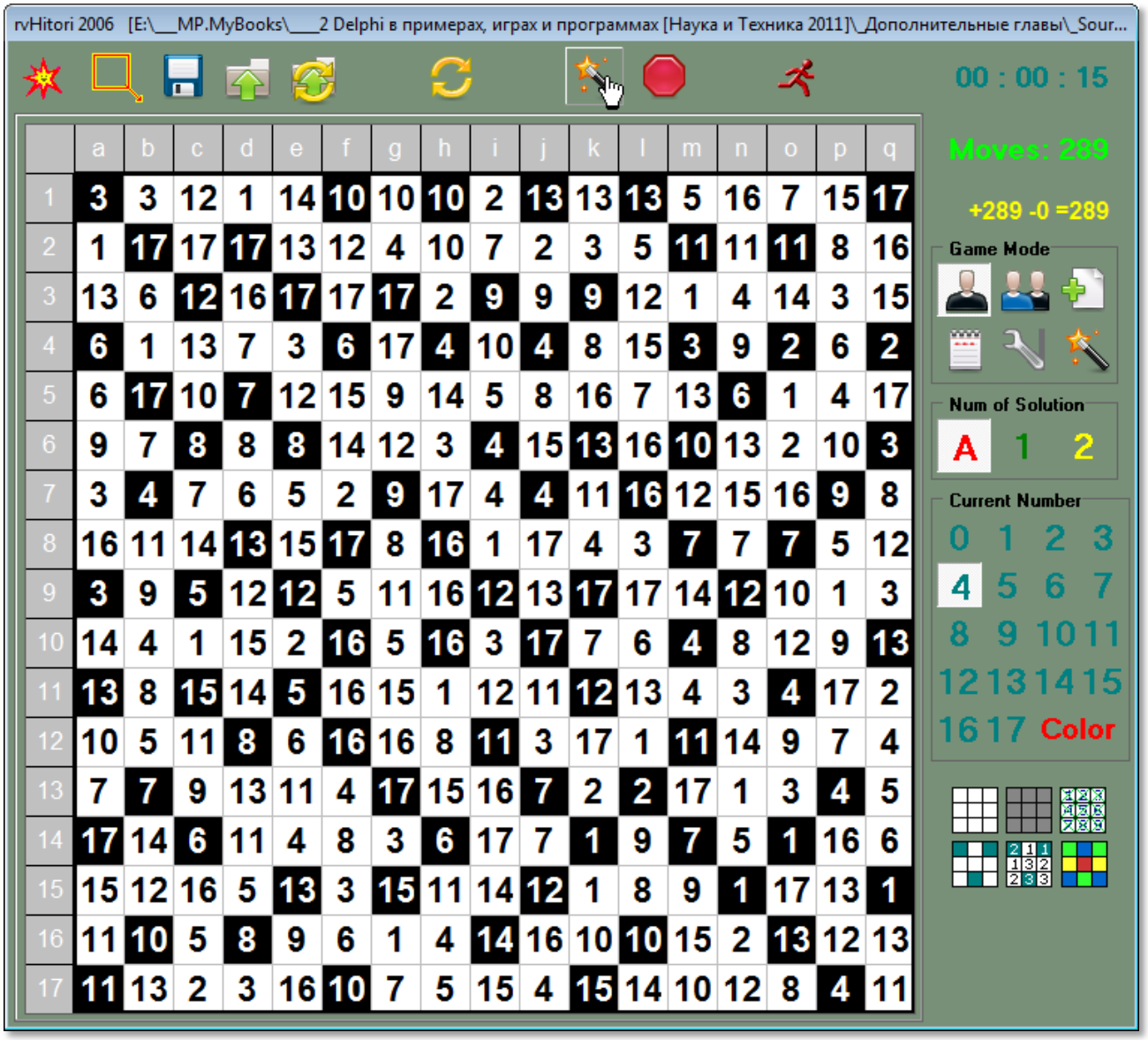


Рис. Ф8.3. *rvHitori* умеет всё!

Например, исключительно с помощью этой программы и были составлены все задачи для упомянутой выше добрым словом книги. А еще вы сможете решать задачи непосредственно в приложении, *не отходя от кассы*! А также составлять задачи *вручную*, пользоваться *подсказками* и прочими благами цивилизации.

В отличие от всех других «сольверов», программа *rvHitori* решает задачи не банальным перебором, а *логически*, то есть опираясь на статические и динамические приемы решения задач, которые ясно и подробно описаны в моей книге. На отдельной форме она ведет полный *протокол* решения за-

дачи и указывает приемы, которые были использованы для нахождения цвета той или иной клетки поля (Рис. Ф8.4).

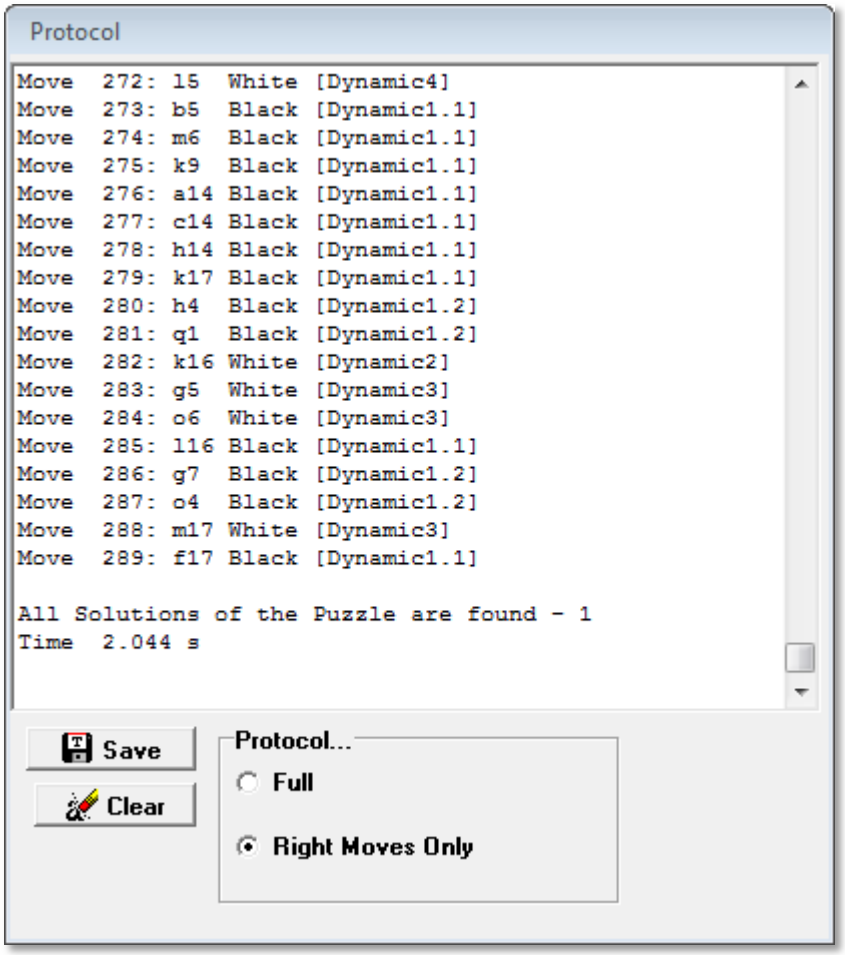


Рис. Ф8.4. Все ходы записаны!

Вы легко можете представить себе размер кода, который описывает почти два десятка приемов решения задач хитори. Поэтому я предоставляю вам в полное распоряжение исходный код обеих программ – с надеждой и верой, что вы *самостоятельно* в нем разберетесь, потому что самостоятельность - самый лучший и правильный способ изучения программирования.



Исходный код программ находится в папках **_rvHitori-SolverRU** и **_rvHitori-SolverPro**.

Факультатив 9. *FireMonkey*, или Стильный интерфейс

Вы, конечно же, обращали внимание на то, что элементы управления одной и той же программы, запущенной под разными версиями ОС *Windows*, немного отличаются друг от друга (Рис. Ф9.1).

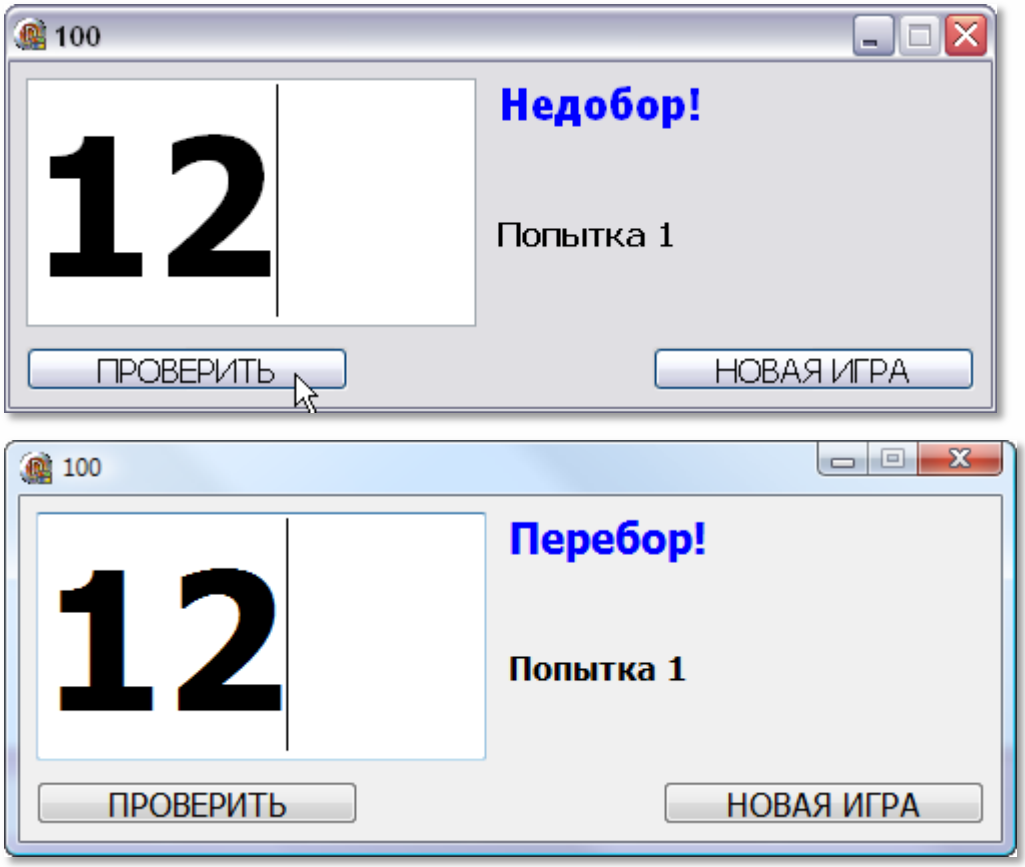


Рис. Ф9.1. Игра в сотню на *Windows XP* и *Windows Vista*

Это не очень хорошо, поскольку приходится проверять работу программ под разными версиями *Windows*. И это совсем нехорошо при разработке программ в *Delphi XE3*, потому что теперь вы можете использовать исходный код в нескольких операционных системах – *Windows*, *Mac OS X* и *iOS*.

Внешний вид самого окна определяется ОС (и вашими настройками), поэтому здесь ничего сделать нельзя. Что касается элементов управления, то их внешний вид также зависит от ОС. До сих пор мы использовали компоненты **VCL** (*Visual Component Library*) – библиотеки визуальных компонентов. Их внешний вид жёстко определяется установленной версией ОС *Windows*, поэтому в других ОС они в принципе работать не будут.

Последние версии *Delphi* ориентированы на создание *кроссплатформенных* приложений, которые должны одинаково работать на компьютерах и мобильных устройствах с разными ОС. Как мы уже знаем, сейчас поддерживаются три ОС, но в будущем их число будет расти.

Именно поэтому в *Delphi XE2* появилась **новая библиотека компонентов** (её также называют *платформой*) под причудливым названием **FireMonkey** (первая версия сокращённо называется *FMX*, а вторая – *FM²*). Её значок вы можете видеть на Рис. Ф9.2.



Рис. Ф9.2. Значок платформы *FireMonkey*

Если мы набросаем новых компонентов на форму и запустим приложение в разных версиях *Windows*, то никакой разницы во внешнем виде элементов управления мы не увидим (Рис. Ф9.3).



К сожалению, проверить это радостное событие на Маке я не могу, но твёрдо верю, что компания *Embarcadero* работает без обмана.

В основе платформы *FireMonkey* лежит библиотека *VG-Scene*, разработанная Евгением Крюковым из Улан-Удэ. В 2011 году права на эту библиотеку купила компания *Embarcadero* и впервые она появилась в *Delphi XE2* под новым названием.



Аналогами этот платформы можно считать *Adobe Flash* и *Microsoft WPF*, но программировать на *Delphi* с *FireMonkey* значительно проще, в чём вы скоро и убедитесь.

Для визуализации элементов управления используется *векторная* графика, что позволяет легко масштабировать ЭУ без ухудшения их качества, а для различных эффектов (например, размытия, теней, свечения) и анимации – *Pixel Shader Level 2.0*. Приложения *FM²* используют графический процессор компьютера, поэтому работают очень быстро. Они поддерживают *2D-* и *3D*-графику, градиенты и частичную прозрачность. *FireMonkey* имеет встроенные перья, кисти геометрические примитивы, которые можно трансформировать. Платформа поддерживает наиболее распространённые графические форматы – *JPEG, PNG, BMP, TIF* и *GIF*.

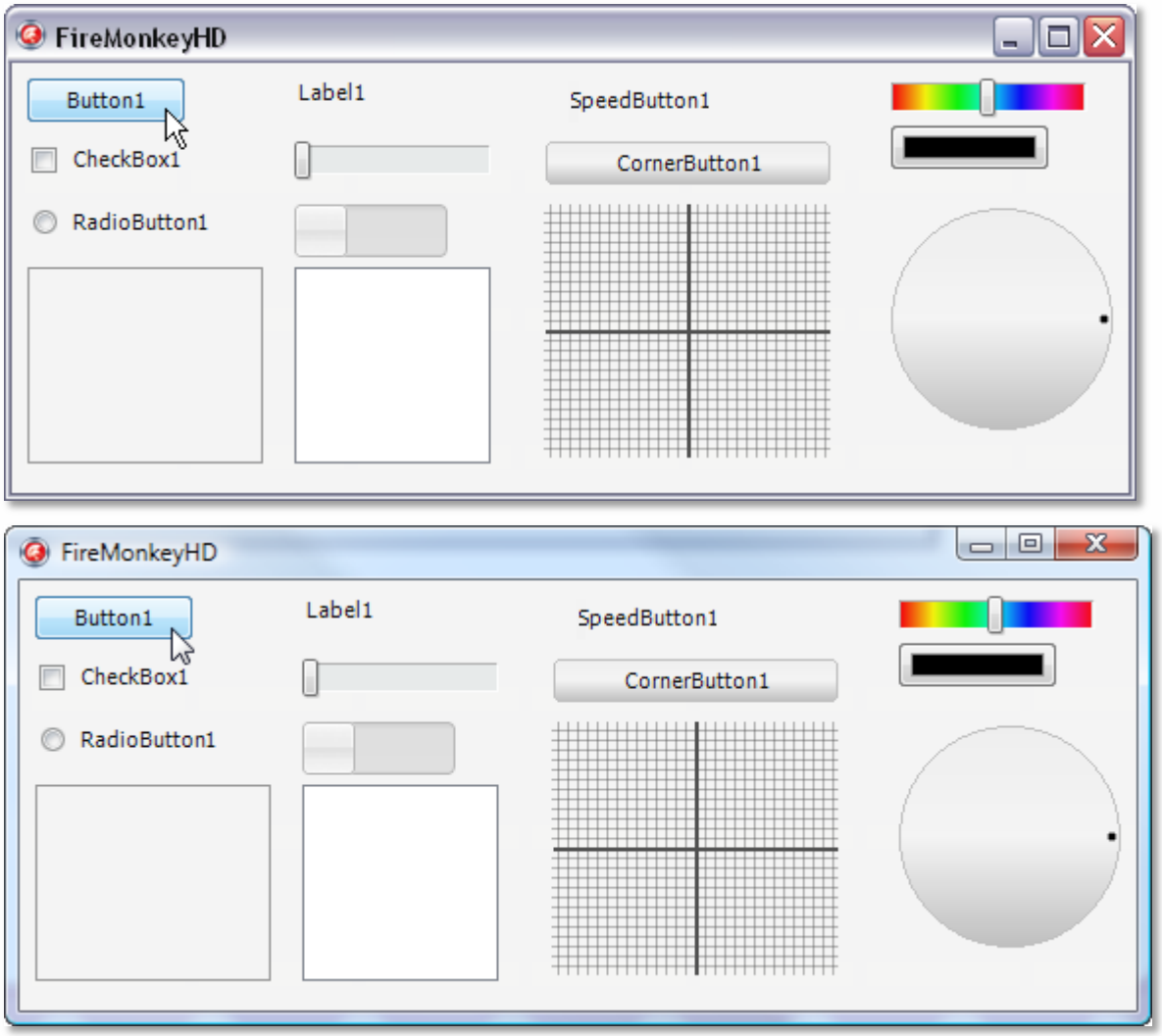


Рис. Ф9.3. Приложение *FireMonkey* на *Windows XP* и *Windows Vista*

Библиотека *FireMonkey* содердит все основные компоненты для проектирования визуального интерфейса пользователя, но с помощью встроенного редактора можно создавать *собственные* компоненты и редактировать стандартные с помощью скинов (стилей, тем).

Приложения *FireMonkey*

После таких дифирамбов вам, безусловно, трудно удержаться от проверки всех достоинств новой платформы на практике.

Хочу вас сразу предупредить: подробный рассказ о новой библиотеке потребует не меньше страниц, чем в этой книге, поэтому мы сможем только прикоснуться к тому богатству возможностей, которые таятся в *FireMonkey*.

Чтобы создать приложение *FireMonkey*, выполните команду *Главного меню* **File > New > FireMonkey Desktop Application – Delphi** (Рис. Ф9.4).

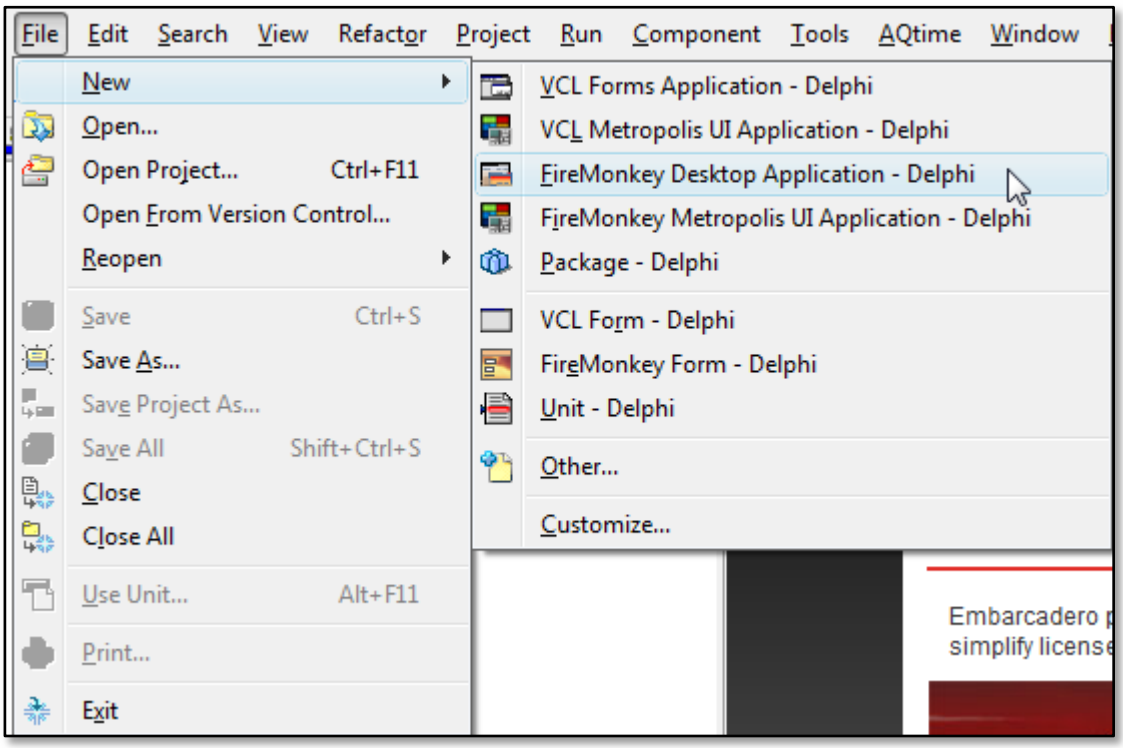


Рис. Ф9.4. Начинаем!

Затем в диалоговом окне **FireMonkey Desktop Application** выберите **HD FireMonkey Application** и нажмите кнопку **OK** (Рис. Ф9.5).

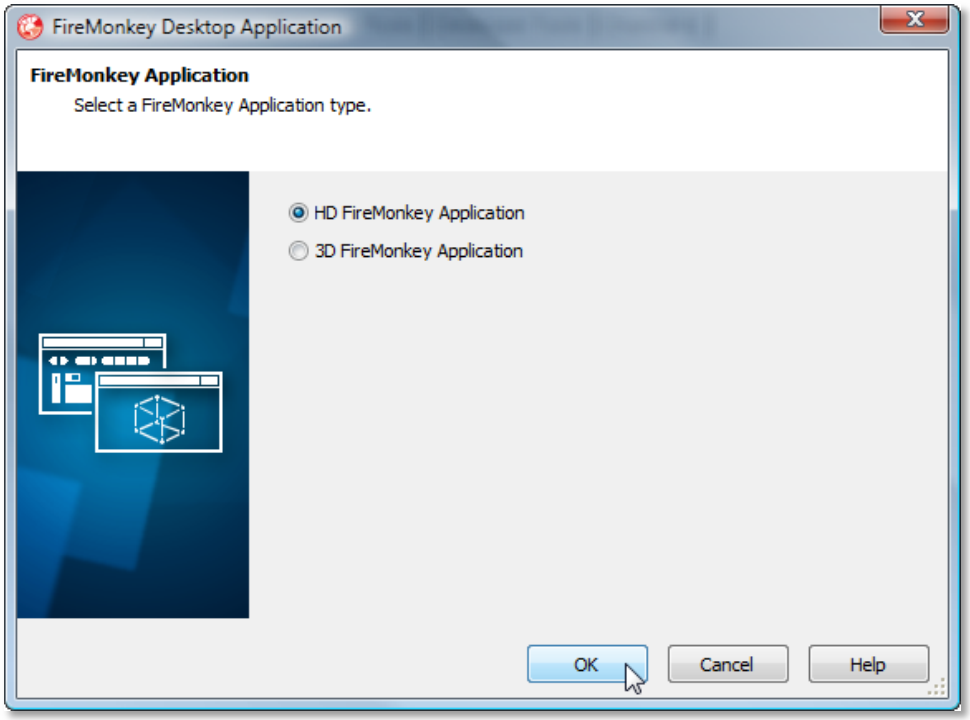


Рис. Ф9.5. Выбираем тип проекта

Приложения *FireMonkey* могут быть двух типов - **HD FireMonkey Application** (*High Density*, двумерная графика) и **3D FireMonkey Application** (трёхмерная графика).

В окне *Конструктора формы* появится **форма** *FireMonkey* (Рис. Ф9.6).

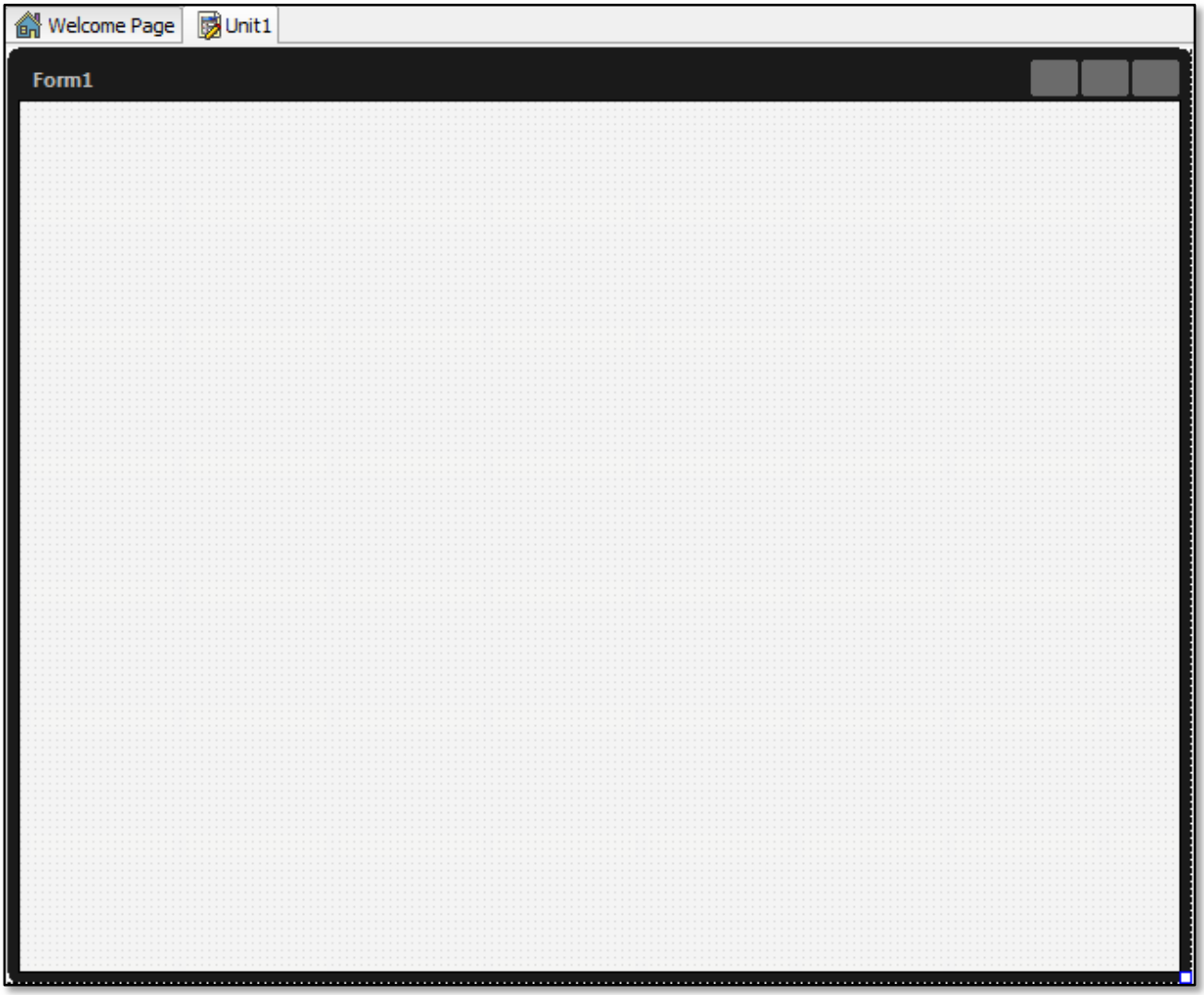


Рис. Ф9.6. Пустая форма

Вместо стандартных кнопок в заголовке формы вы увидите серые прямоугольники, но это не должно вас пугать – в работающем приложении они появятся на своих местах (Рис. Ф9.7).

Запускаются приложения *FireMonkey* точно так же, как и обычные, кнопками .

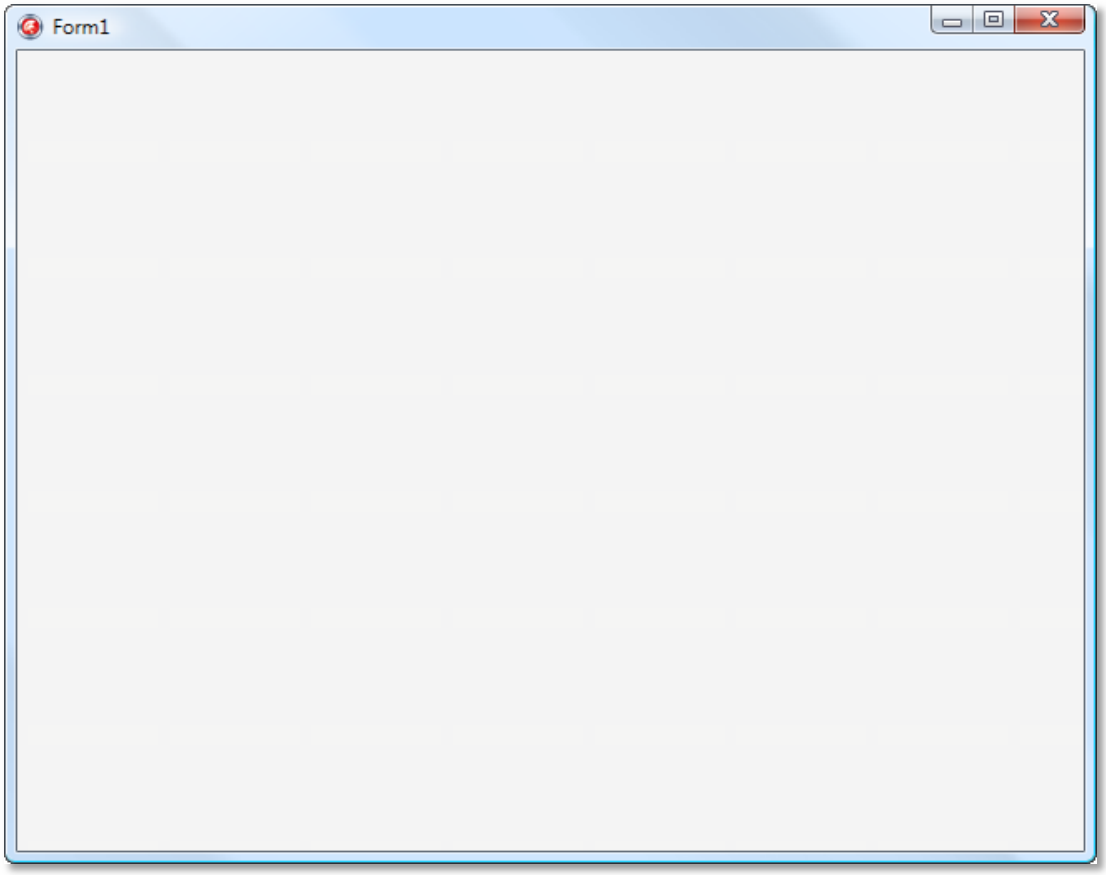


Рис. Ф9.7. Окно приложения

Одновременно в *Менеджере проекта* откроется новая папка с файлами проекта (Рис. Ф9.8).

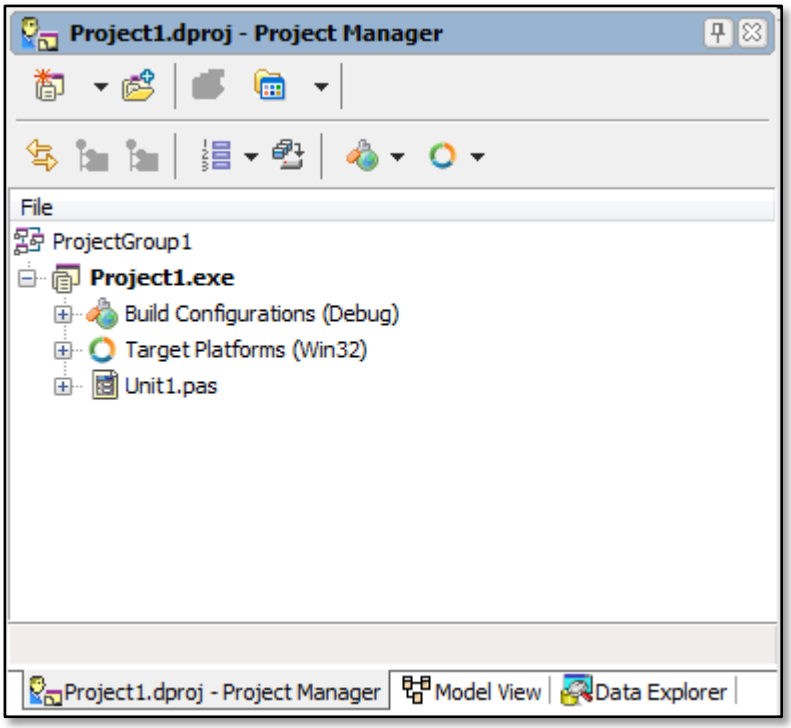
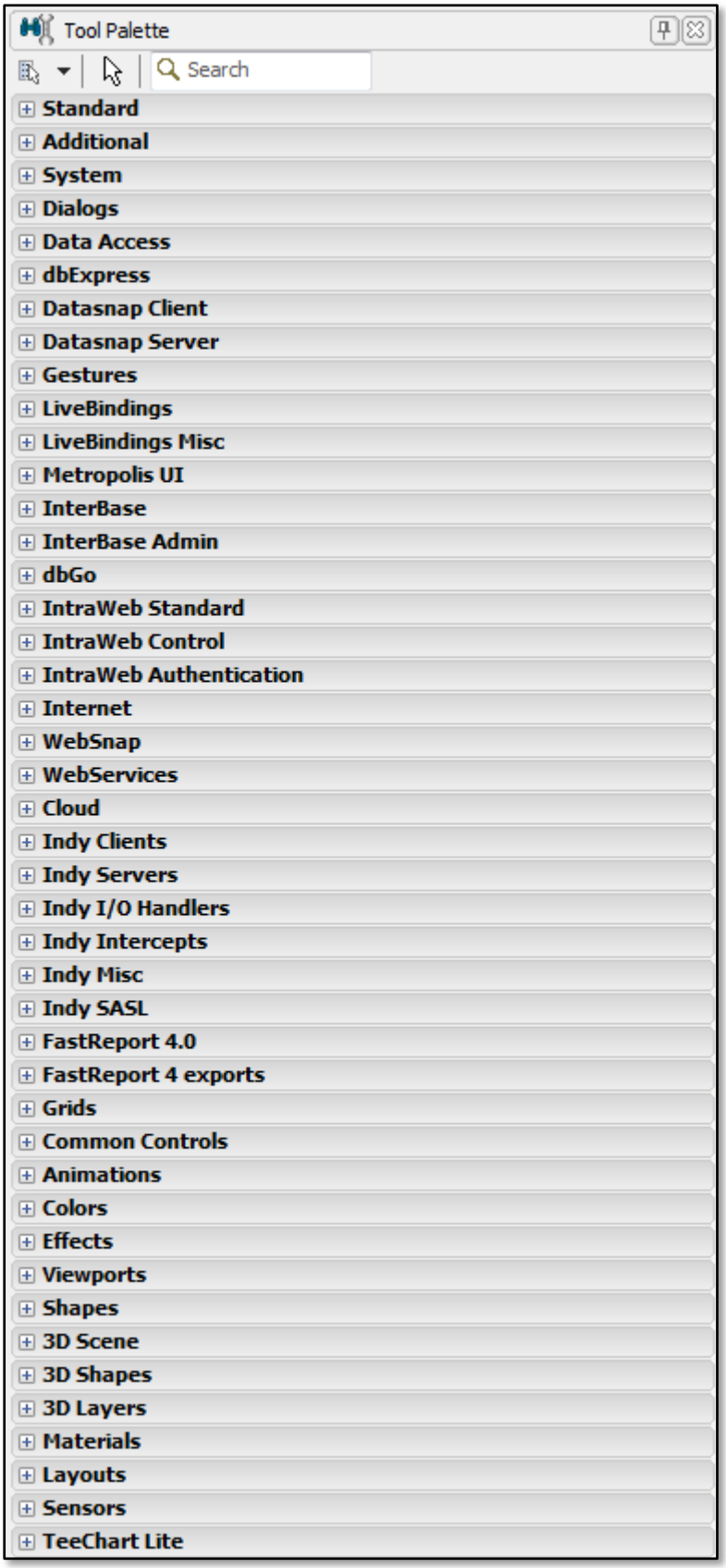


Рис. Ф9.8. Файлы приложения *FireMonkey*

Компоненты *FireMonkey*



Но нам не пристало любоваться пустым окном, тем более что оно ничем не отличается от любых других окон, поэтому давайте познакомимся с **компонентами**, которые мы можем использовать в своих приложениях. Для этого мы переходим в окно *Палитры инструментов* и видим ОГРОМНЫЙ список страниц с компонентами (Рис. Ф9.9)!



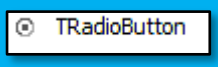
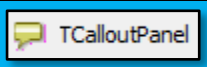
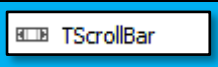
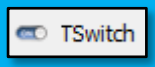
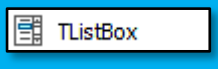
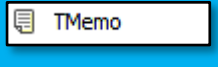
Все компоненты доступны в обоих типах приложений *FireMonkey*, но не все они будут работать!

Рис. Ф9.9. Полный список компонентов библиотеки *FireMonkey*

Мы раскроем только несколько страниц - с самыми нужными компонентами.

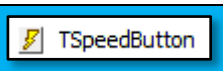
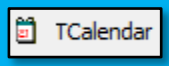
Standard

– **стандартные** компоненты, которые наиболее часто используются в приложениях.

	**главное меню** программы.
	невизуальный компонент, создаёт **список событий**.
	**контекстное меню**, появляющееся на экране при щелчке правой кнопкой мыши.
	стандартная **кнопка**.
	**флажок**; позволяет установить или отменить связанное с ним свойство. *В группе* может быть выбрано *любое* количество флажков.
	**переключатель**; одиночный действует, как флажок, но в *группе* может быть выбран только *один* переключатель.
	**контейнер для флажков** и **переключателей**.
	**панель**, служит контейнером для размещения на ней других компонентов. Может быть использована для создания панелей инструментов и строки состояния.
	**панель** со стрелкой-указателем.
	**метка** для вывода надписей на форму.
	контейнер для **графики**.
	**полоса прокрутки**, **скроллер**.
	**переключатель**.
	**список строк**.
	**комбинированный список** (текстовое поле и список).
	простой **многострочный текстовый редактор**.
	**текстовое поле**, однострочный текстовый редактор. Позволяет вводить данные в работающей программе.

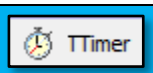
Additional

– **дополнительные** компоненты. Они также очень часто используются при разработке интерфейса приложения.

	**кнопка**, которая может выполнять функции флажка и переключателя, годится для создания панелей инструментов.
	**кнопка** со **скруглёнными** углами.
	невизуальный компонент, **проигрыватель медиафайлов**: **звуковых** – *wma, mp3, wav* – и **видео** – *avi, wmv*.
	**календарь.**
	основа для вывода на экран **графических изображений**.

System

На этой странице находится единственный, но очень нужный компонент.

	**таймер**.

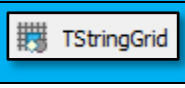
Dialogs

– невизуальные компоненты для вызова **диалоговых окон**.

	диалоговое окно **открытия файлов.**
	диалоговое окно **сохранения файлов.**

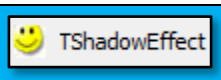
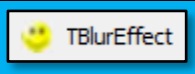
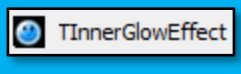
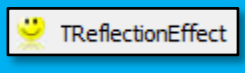
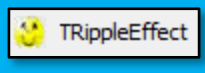
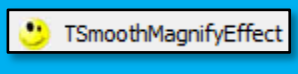
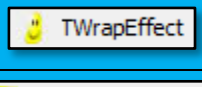
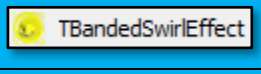
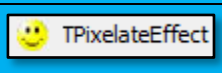
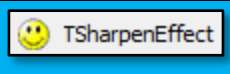
Grids

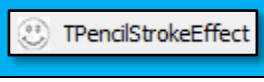
– **сетки** для табличного вывода информации.

	сетка для вывода **графики** и **текста**.
	сетка для вывода **строк**.

Effects

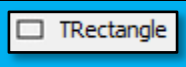
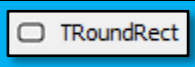
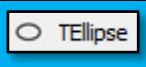
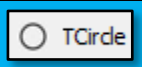
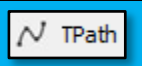
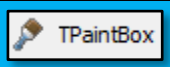
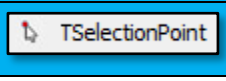
– **визуальные эффекты** для растровых изображений и элементов управления.

	**тень**.
	**размытие**.
	**свечение**.
	**внутреннее свечение**.
	**скосы**.
	**отражение** вниз.
	круговые **волны**.
	**закручивание**.
	круглая **лупа**.
	круглая **лупа** с **размытием**.
	вертикальные **полосы**.
	зигзагообразные **волны**.
	криволинейная **деформация**.
	спиральные **волны**.
	**втягивание** в круговую область.
	мозаичное **размножение картинки** по горизонтали и вертикали.
	**пикселизация**.
	**выдавливание**.
	повышение **резкости**.
	уменьшение числа **цветов**.
	окрашивание в цвет **сепии**.

	**набросок**.
	карандашный **рисунок**.

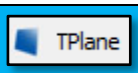
Shapes

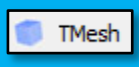
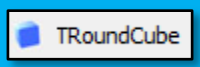
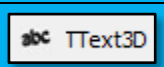
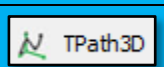
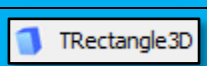
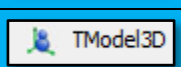
– **двумерные** геометрические и графические **объекты**.

	прямая **линия**.
	**прямоугольник**.
	**прямоугольник** со **скруглёнными** углами.
	**эллипс**.
	**окружность**.
	**дуга**.
	**сектор**.
	**текст**.
	**путь**.
	**картинка**.
	поверхность для **рисования**.
	**прямоугольник** со **стрелкой**.
	**прямоугольная** пунктирная **рамка**.
	**точка**.

3D Shapes

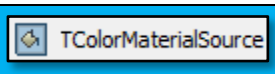
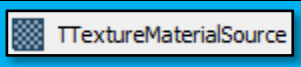
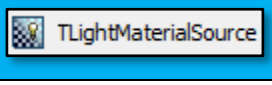
– **трёхмерные** геометрические **объекты**.

	**плоскость**.
	плоский **диск**.

	**куб**.
	**сетка**.
	**сфера**.
	**цилиндр**.
	**куб** со **скруглёнными** гранями.
	**конус**.
	плоская **сетка** для рисования.
	**каркасный куб**.
	**текст**.
	**путь**.
	прямоугольный **параллелепипед**.
	эллиптический **цилиндр**.
	**модель**.

Materials

– **материалы** трёхмерных объектов.

	**цветной** материал.
	**текстурный** материал.
	материал **освещения**.

Как мы видим, некоторые аналоги *VCL*-компонентов в библиотеке *Fire-Monkey* отсутствуют, но зато появилось много новых.

Привет, *FireMonkey*!



Остеровский мультик

Поскольку название платформы *FireMonkey* переводится как *Огненная Обезьяна*, то мы передадим наш **пламенный** привет не просто абстрактной платформе, а настоящей рыжей обезьяне – Мартышке из остеровского мультфильма.

Начните новый проект *HD FireMonkey Application*, задайте свойству формы **Caption** значение *Привет, FireMonkey!* и установите на форме компонент **imgMonkey** типа *TImage*.

В него мы загрузим картинку, оживляющую скупой интерфейс нашего приложения.

Перейдите в *Инспектор объектов*, щёлкните по строчке **Bitmap**, а затем по **кнопке** с тремя точками. Откроется список трёх желаний, из которого выберите строчку **Edit...** (Рис. Ф9.10).

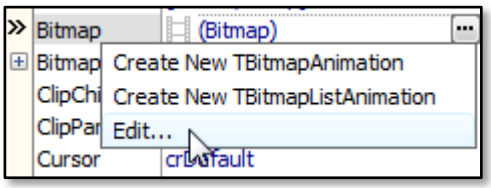


Рис. Ф9.10. Загружаем картинку

Когда распахнётся мощное окно **Редактора картинок** *Bitmap Editor*, нажмите кнопку **Load...** (Рис. Ф9.11),

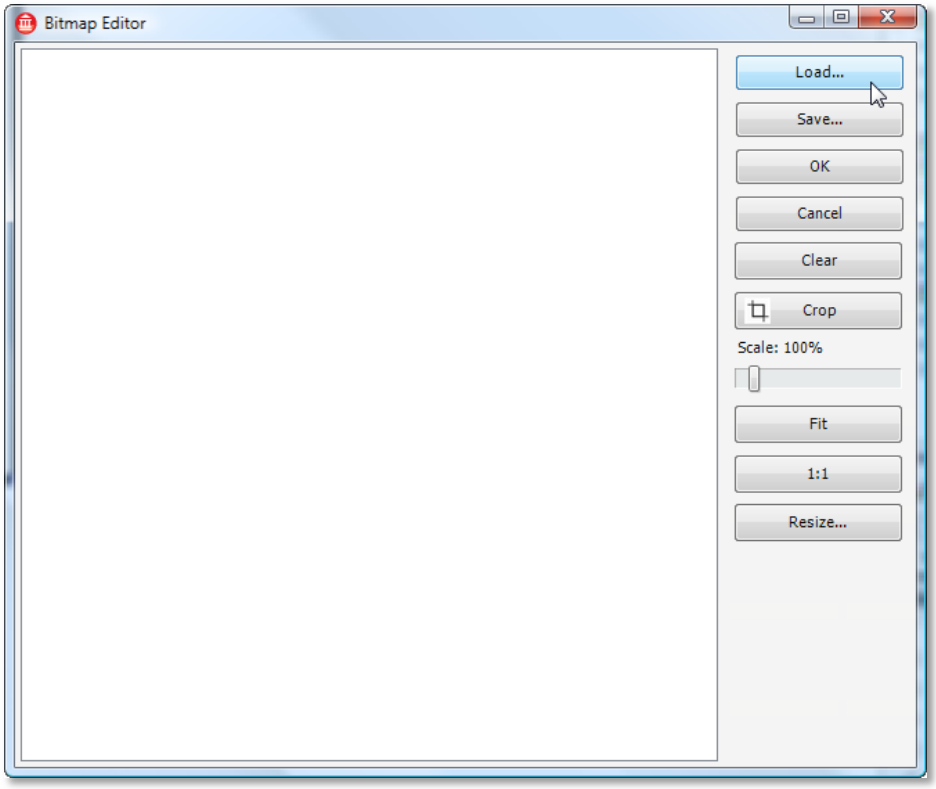


Рис. Ф9.11. Простой графический редактор

Перейдите в папку с картинкой, загрузите её в редактор и нажмите кнопку **ОК** (Рис. Ф9.12).

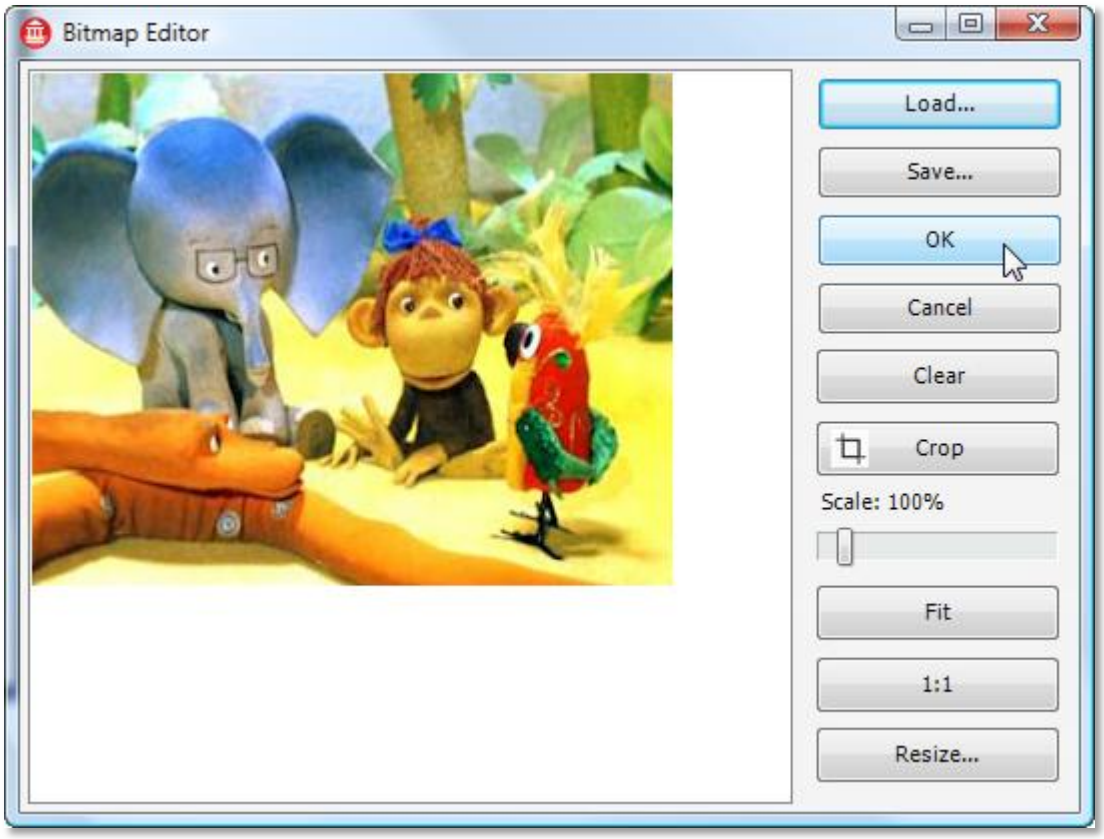


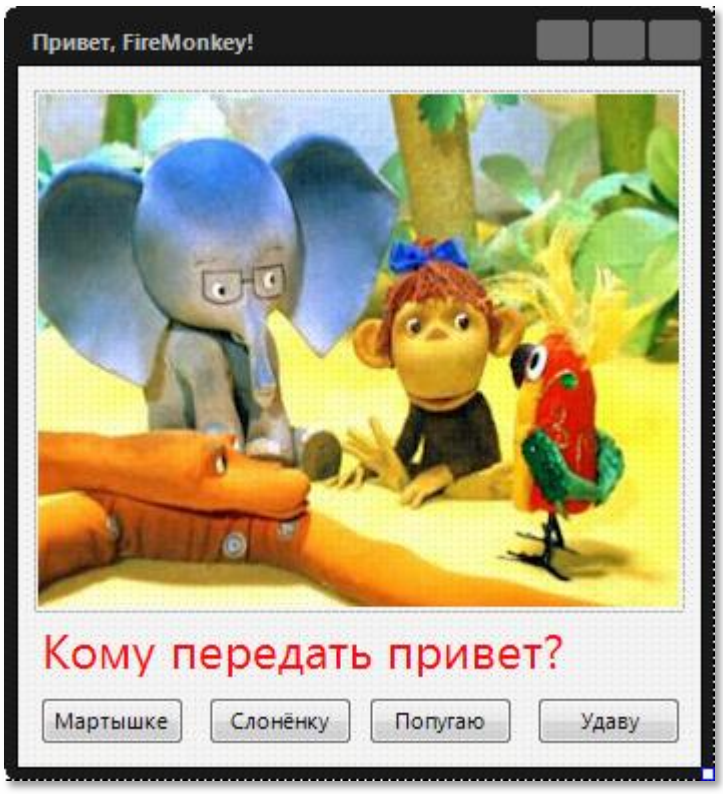
Рис. Ф9.12. Картинка загружена!

Я заранее позаботился о том, чтобы в папке лежала нужная картинка. И вы всегда думайте наперёд!



Картинка появится в компоненте *imgMonkey*, после чего вам нужно подогнать его размеры так, чтобы не ущемлять милых животных ни с какой стороны (Рис. Ф9.13).

Рис. Ф9.13. Мизансцена готова



Для вывода приветов нам понадобится *метка* **lblPrivet** с надписью *Кому передать привет?* и 4 *кнопки* – **btnM**, **btnS**, **btnP** и **btnU** – для выбора адресата (Рис. Ф9.14).

Рис. Ф9.14. Кнопки с приветами

Теперь щёлкайте по кнопкам и заполняйте кодом методы, обрабатывающие нажатие на кнопку:

```
procedure TfrmMain.btnMClick(Sender: TObject);
begin
   PrintPrivet('Мартышка');
end;
procedure TfrmMain.btnSClick(Sender: TObject);
begin
   PrintPrivet('Слонёнок');
end;
procedure TfrmMain.btnPClick(Sender: TObject);
begin
   PrintPrivet('Попугай');
end;
procedure TfrmMain.btnUClick(Sender: TObject);
begin
   PrintPrivet('Удав');
end;
```

А метод **PrintPrivet** передаёт *общий* привет:

```
procedure TfrmMain.PrintPrivet(str: string);
begin
   lblPrivet.Text:= 'Привет, ' + str + '!';
end;
```

Запускаем приложение, нажимаем кнопки – и все зверушки получают по привету (Рис. Ф9.15).

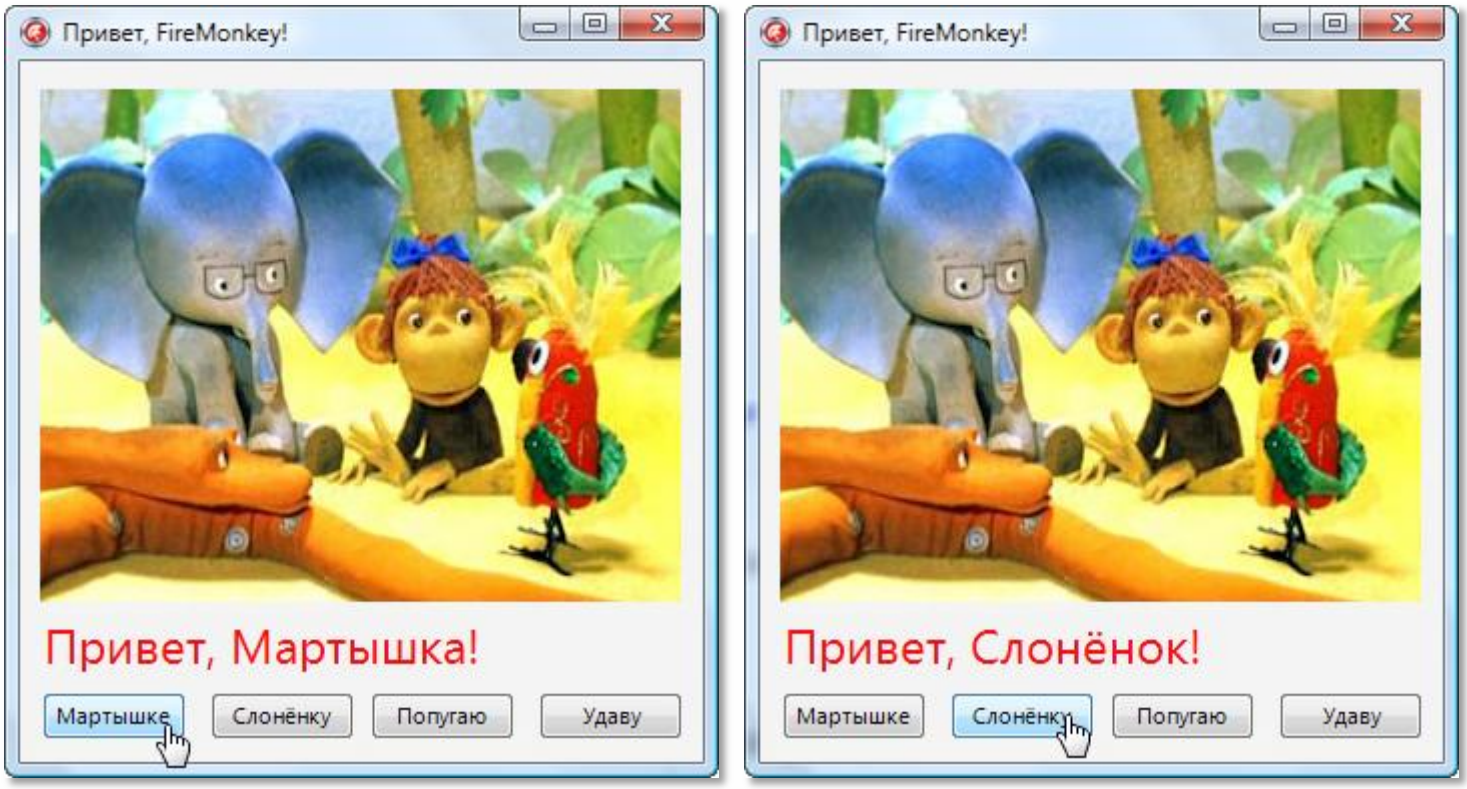


Рис. Ф9.15. Двое уже с приветом!



Исходный код программы находится в папке **FireMonkey > Privet**.

Тюнинг кнопок

Вы, конечно, заметили, что по внешнему виду наше приложение ничем не отличается от наших прежних творений. Казалось бы, не стоило и огород городить – но не спешите: сейчас мы придадим нашим кнопкам незабываемый облик!

Для начала задайте им размер 64 x 96 пикселей и сотрите с них надписи. Последняя операция необходима для того, чтобы они не портили картинки, которые мы поместим в верхней части кнопок. Так как надписи на кнопках всегда выводятся в *центре* кнопочного пространства, то они неизбежно наедут на картинки и испортят нам всю картину (ещё раз извините за тавтологию)!

Теперь набросайте на форму 4 компонента **TImage** и столько же **меток**.

В метках напишите ПРОПИСНЫМИ буквами имена персонажей, а в компоненты *TImage* загрузите их портреты.

В окне *Структуры проекта* должен получиться вот такой список компонентов (Рис. Ф9.16).

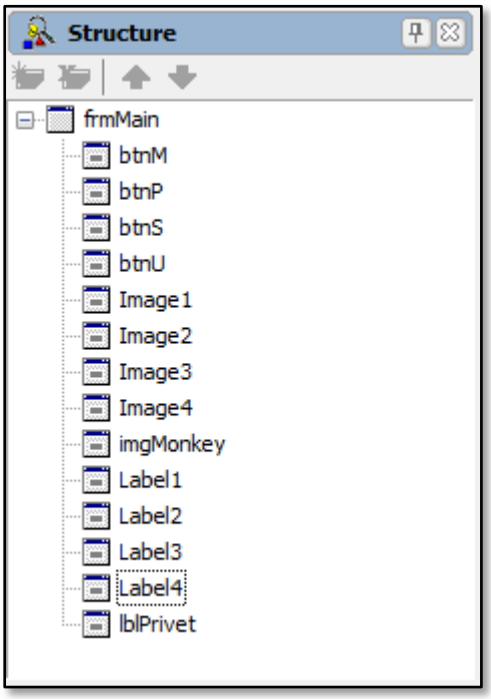


Рис. Ф9.16. Все компоненты приложения

А теперь я вам объясню, для чего нам понадобились картинки и метки. Дело в том, что стандартные кнопки *FireMonkey* не имеют свойства *Bitmap*, поэтому их нельзя украсить картинкой. Зато они могут играть роль **контейнера** для других компонентов. В данном случае – для картинок и меток.

Чтобы поместить какой-либо компонент в контейнер, подцепите его мышкой в окне *Структуры кода* и перетащите на нужную кнопку. Список компонентов приобретёт *иерархическую* структуру (Рис. Ф9.17).

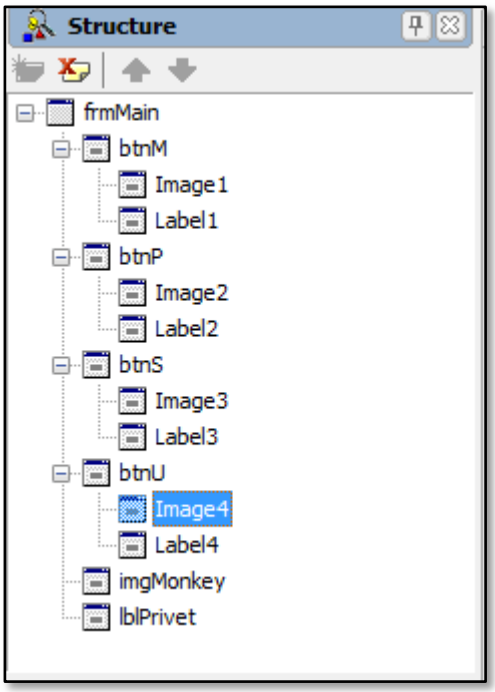


Рис. Ф9.17. Иерархия – мать порядка!

Поочерёдно выделяйте на кнопках картинки и метки (Рис. Ф9.18) и в окне *Инспектора объектов* задавайте им следующие значения свойства **Align** (выравнивания):

```
Align = alTop          - для картинок
Align = alMostBottom   - для меток
```

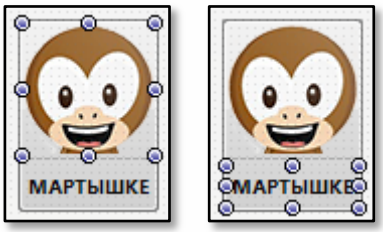


Рис. Ф9.18. Выравниваем!

Если вы постараетесь не хуже меня, то при запуске программы будете вознаграждены пёстрым видом интерфейса (Рис. Ф9.19).



Рис. Ф9.19. И эти с приветом!

Таким нехитрым способом вы можете создавать компоненты невиданной красоты и очарования!

Повышаем ЭФФЕКТивность кнопок!

Все визуальные элементы управления можно украсить эффектами со страницы **Effects**.

Не все они годятся для элементов управления, но некоторые эффекты вполне уместны и смотрятся неплохо. Например, тень придаёт элементу управления объём, и этот приём используется очень часто.

Так как у нас на форме имеется 6 компонентов, то перетащите столько же эффектов **TShadowEffect** на форму. Но они появятся не на форме, а в окне *Структуры проекта*. Перенесите эффекты на компоненты так, чтобы всем сестрам досталось по серьгам. В итоге должен получиться такой список (Рис. Ф9.20).

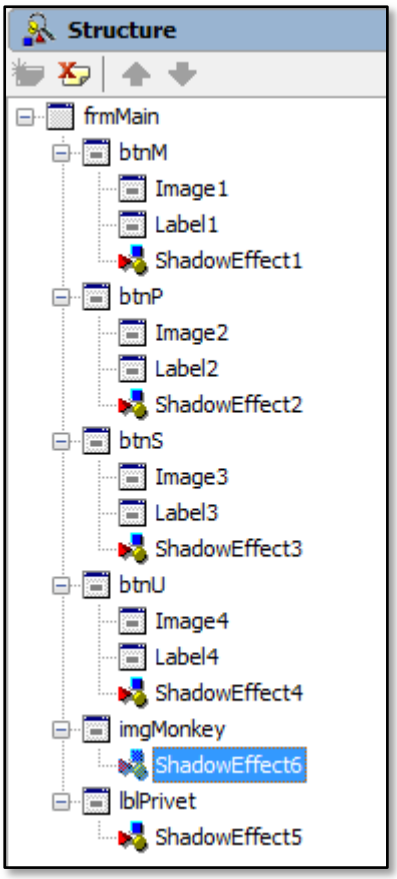


Рис. Ф9.20. Список растёт!

Действие эффектов хорошо видно уже и в *Конструкторе формы*, но для проверки мы запустим приложение и убедимся, что элементы управления «приподнялись» над формой и выглядят отлично (Рис. Ф9.21).



Рис. Ф9.21. Бросаем тень!

Если параметры тени по умолчанию вас не удовлетворяют, вы можете дополнительно настроить их в *Инспекторе объектов* (Рис. Ф9.22).

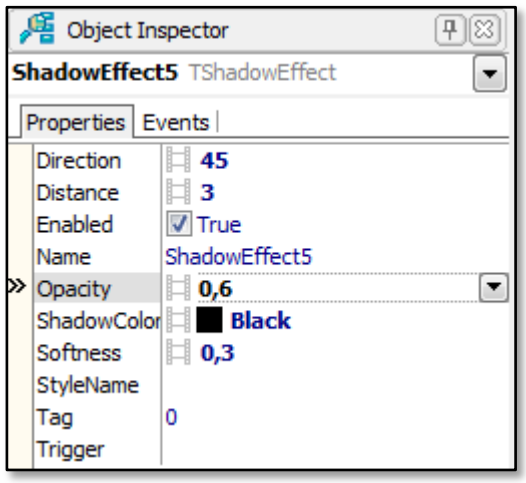


Рис. Ф9.22. Теневые параметры

Для *кнопок* годятся также эффекты **TGlowEffect** (*первая* кнопка), **TInnerGlowEffect** (*вторая* кнопка), **TIvertEffect** (*третья* кнопка), **TSepiaEffect** (*четвёртая* кнопка) (Рис. Ф9.23). Для картинок вы можете использовать любые эффекты, например, **TWaveEffect**, как на рисунке.

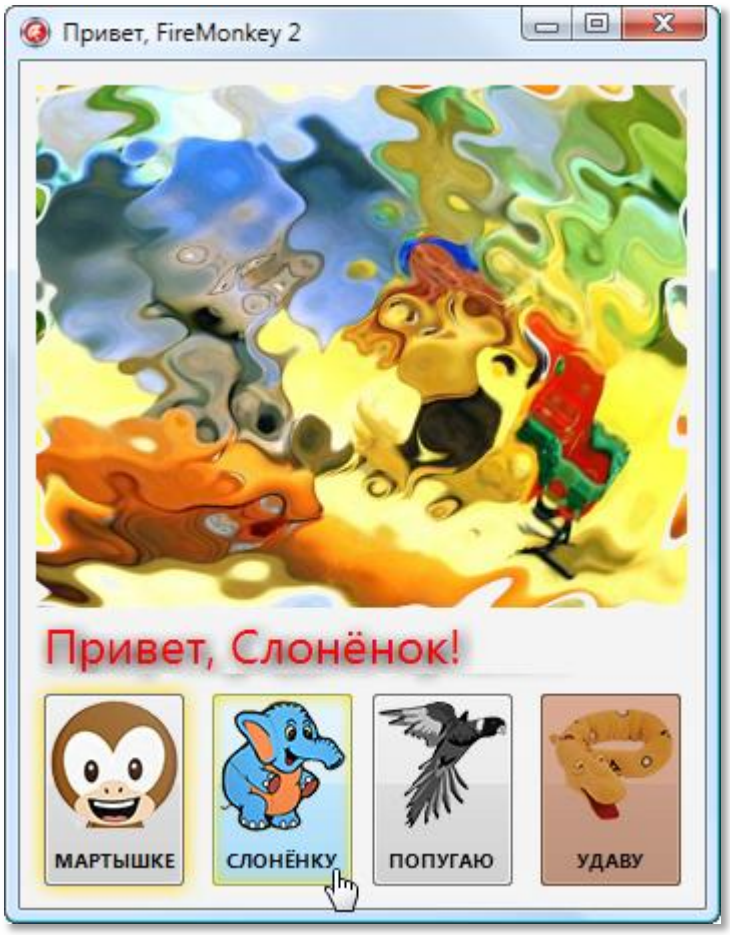


Рис. Ф9.23. Волнительная картинка!

На мой взгляд, для кнопок лучше выбирать эффект тени, а для картинок – эффект отражения (Рис. Ф9.24).



Рис. Ф9.24. Эффективный интерфейс готов!



Исходный код программы находится в папке **FireMonkey > Privet2**.

СТИЛИзация интерфейса

Внешний вид приложений, написанных на *Delphi XE3* с применением технологии *FireMonkey*, можно менять одним щелчком мышки. Для этого достаточно воспользоваться **стилями**, которые поставляются с дистрибутивом.

Немного увеличьте высоту формы и опустите все элементы управления так, чтобы сверху освободилось место для кнопки **sbtOpenStyle** типа *TSpeedButton*. Напишите на ней *Open Style* или *Открыть стиль* – как вам угодно. Также добавьте компонент **TOpenDialog** для открытия файлов.

Щёлкните по новой кнопке и в методе **sbtOpenStyleClick** напишите пару строк:

```
procedure TfrmMain.sbtOpenStyleClick(Sender: TObject);
begin
  if OpenDialog1.Execute then
    TStyleMana-
ger.SetStyle(TStyleManager.LoadFromFile(OpenDialog1.FileName));
end;
```



В раздел *uses* не забудьте добавить пространство имён **FMX.Styles**!

Теперь смело и решительно запускайте приложение, нажимайте кнопку **Open Style** и выбирайте любой стиль.



Чтобы вам не пришлось разыскивать готовые стили, я скопировал их в папку с исходным кодом программ *FireMonkey*.





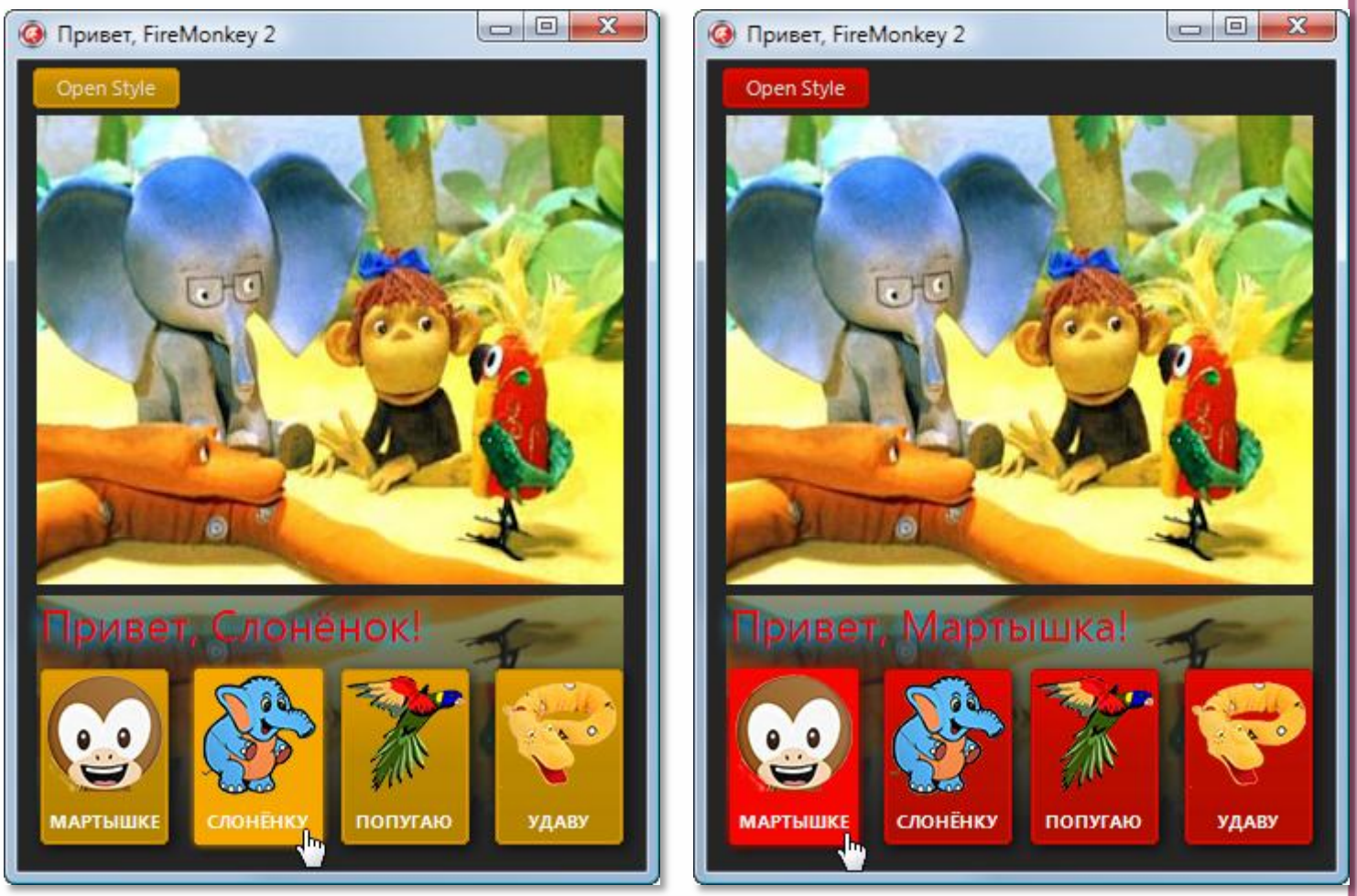


Рис. Ф9.25. Стильно!

Рис. Ф9.25 позволит вам легко оценить преимущества такого способа декорации интерфейса пользователя.



Исходный код программы находится в папке **FireMonkey > Privet2**.

Редактирование стилей

Если бы губы Никанора Ивановича
да приставить к носу Ивана Кузьмича...

Николай Васильевич Гоголь, пьеса *Женитьба*

Применение *готовых* стилей, безусловно, впечатляет, но всегда не хватает какой-то закавыки или малости, а без неё и жизнь не мила. Тогда можно открыть в любом текстовом редакторе файл нужного стиля с расширением **.style** и править его вручную:

```
object TLayout
  Align = alClient
  Height = 715.000000000000000000
  Position.Y = 33.000000000000000000
  Width = 1162.000000000000000000
  object TLayout
    StyleName = 'switchstyle'
    DesignVisible = False
    Height = 14.000000953674320000
    Position.X = 416.000000000000000000
    Position.Y = 514.000000000000000000
    Width = 130.000000000000000000
    object TBrushObject
      StyleName = 'On'
      Brush.Color = xFF65C8E0
    end
    object TBrushObject
      StyleName = 'Off'
      Brush.Color = xFF484848
    end
. . .
```

И хотя всё гениальное – просто, да не всё простое – гениально, поэтому лучше воспользоваться средствами *Delphi XE3* для редактирования стилей.

Установите на форму любой компонент (например, кнопку) и нажмите на нём правую кнопку мышки, чтобы вызвать контекстное меню. В нём вы увидите две строки – **Edit Custom Style...** и **Edit Default Style...** (Рис. Ф9.26).

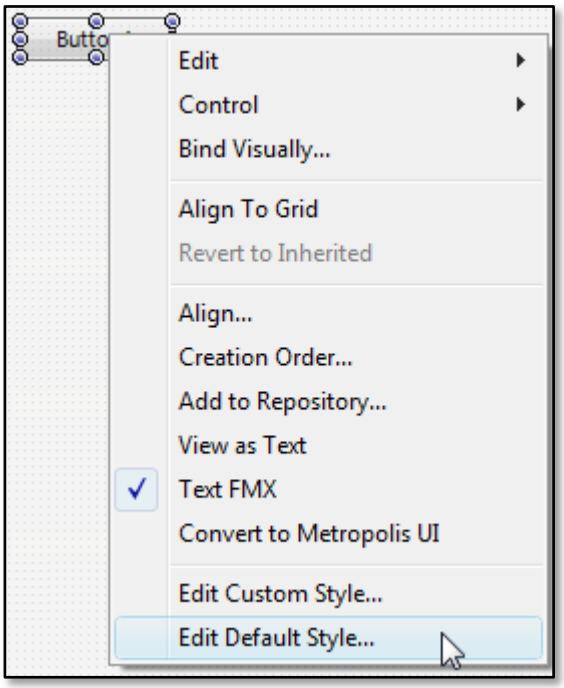


Рис. Ф9.26. Начинаем редактировать стили!

Все новые элементы управления создаются со стилем по умолчанию. Вряд ли его нужно изменять, но давайте посмотрим, как это можно сделать.

После выполнения команды **Edit Default Style...** на форме появится новый компонент **StyleBook1** (Рис. Ф9.27), который содержит стиль для кнопки по умолчанию.

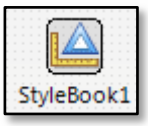


Рис. Ф9.27. Список стилей

Вы можете дважды щёлкнуть по нему, чтобы открыть окно *Редактора стилей* (Рис. Ф9.28).

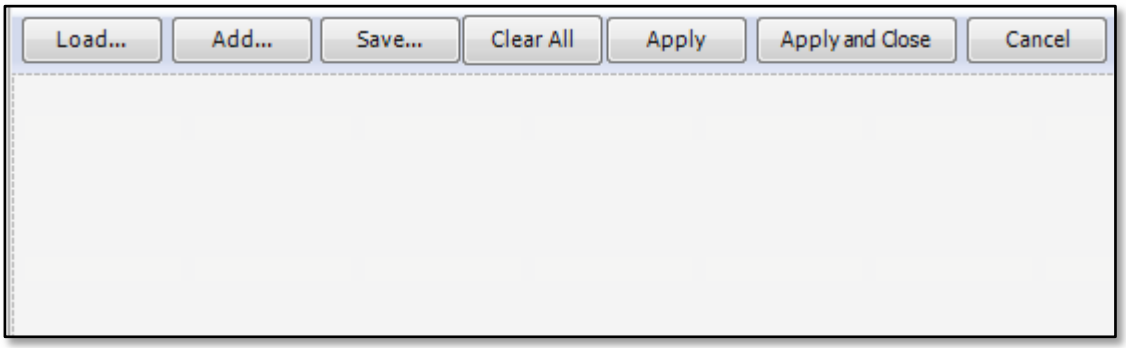


Рис. Ф9.28. *Редактор стилей*

Впрочем, если вы нажали строчку **Edit Default Style...**, то *Редактор* уже открылся.

В окне *Структуры проекта* появится список всех элементов, из которых состоит кнопка – фоновая прямоугольная картинка и текст (Рис. Ф9.29).

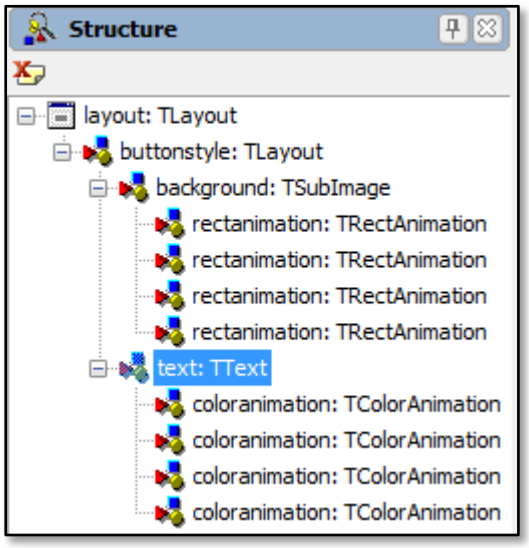


Рис. Ф9.29. Элементы стиля

Давайте изменим цвет надписи на кнопке. Для этого выделите строчку **text: TText** в окне *Структуры проекта*, а в *Инспекторе объектов* выберите любой цвет, отличный от чёрного (Рис. Ф9.30).



Рис. Ф9.30. Выбираем цвет надписей

Нажмите на кнопку **Apple and Close**, чтобы изменения вступили в силу.

Действительно, надпись на нашей кнопке перекрасилась. Если мы будем добавлять на форму *новые* кнопки, то все они будут украшены надписями заданного цвета (Рис. Ф9.31).



Рис. Ф9.31. Обновлённые кнопки

Если вы запустите приложение и наведёте курсор мышки на кнопку, то надпись снова станет чёрной, потому что мы не изменили конечный цвет анимации. Но мы этого делать и не будем. Более того, закончите редактирование стиля кнопки по умолчанию, вернув тексту исходный чёрный цвет.
Теперь выполните команду **Edit Custom Style...**, нажмите в *Редакторе стилей* кнопку **Load...** и загрузите какой-нибудь стиль (Рис. Ф9.32).

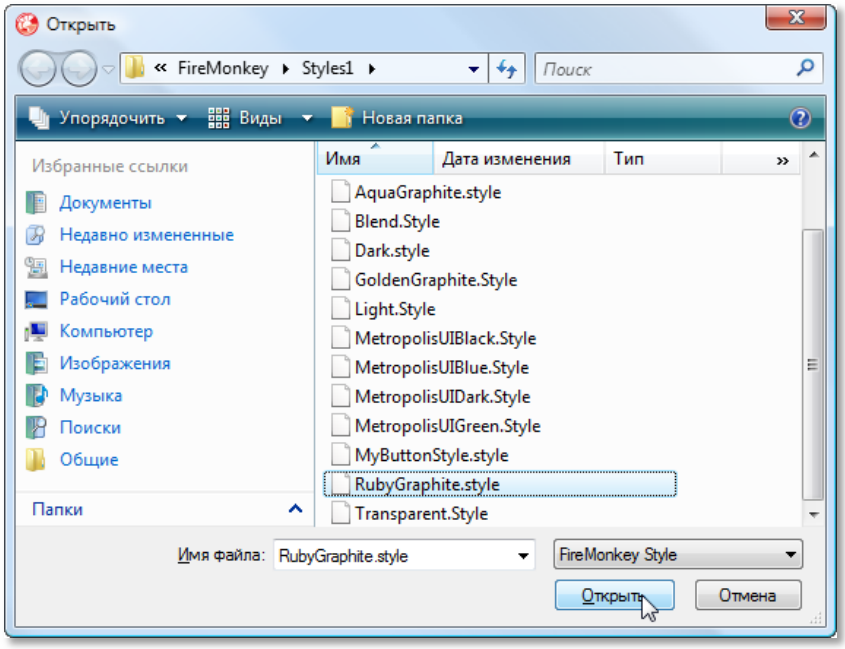


Рис. Ф9.32. Загружаем стиль в *Редактор*

Для примера мы изменим цвет кнопок (фона, а не надписей).

В окне *Структуры проекта* выделите строку **background: TRectangle** (Рис. Ф9.33).

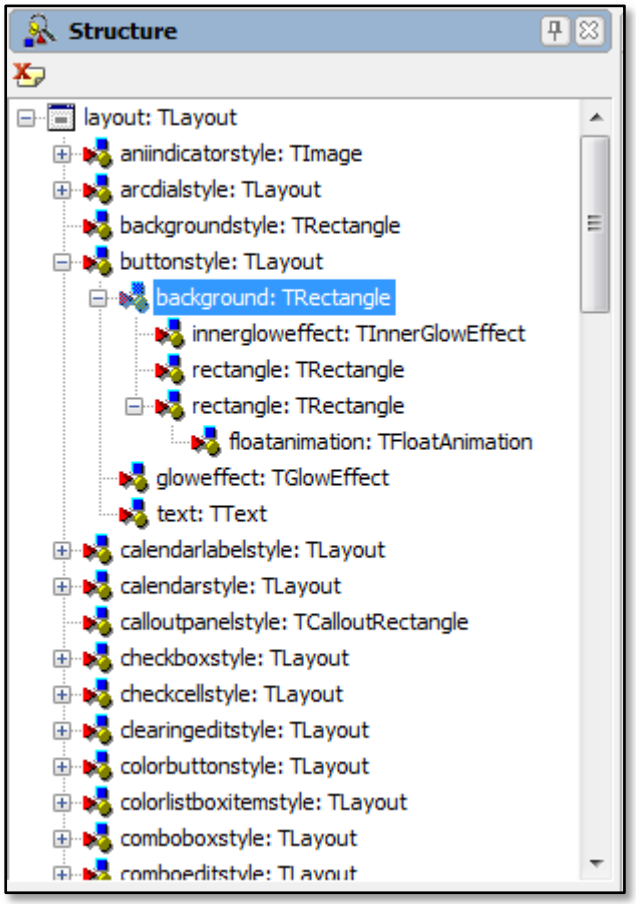


Рис. Ф9.33. Изменяем цвет фона кнопки

В *Инспекторе объектов* щёлкните по строчке **Fill**, а затем по кнопке с точками (Рис. Ф9.34).

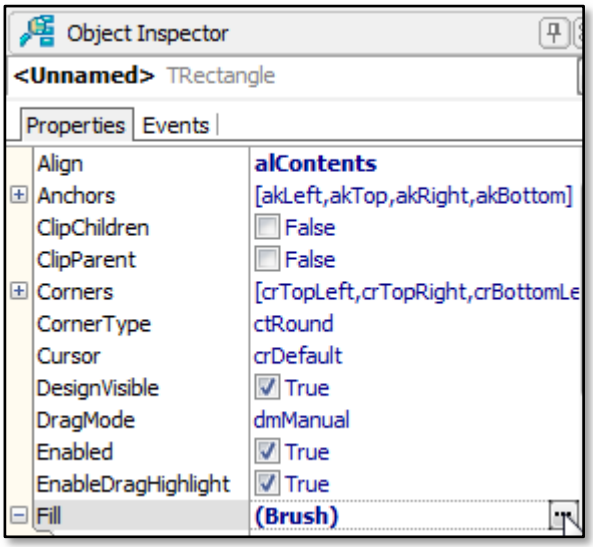


Рис. Ф9.34. Вызываем *Редактор кисти*

В открывшемся **Редакторе кисти** поочерёдно выделяйте квадратики под полоской градиентной заливки и выбирайте цвет в окне **Gradient** (Рис. Ф9.35).

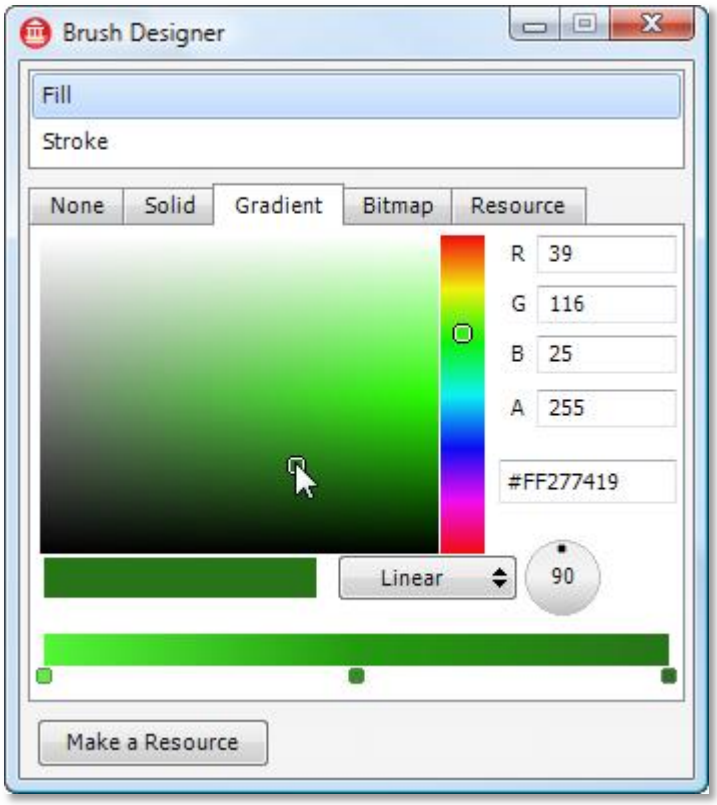


Рис. Ф9.35. Выбираем цвет заливки

Закройте *Редактор кисти* – фон кнопки изменился согласно нашим желаниям (Рис. Ф9.36).

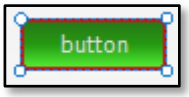


Рис. Ф9.36. Перекрашенная кнопка

Нажмите кнопку **Save...** в *Редакторе стилей* и сохраните новый стиль (Рис. Ф9.37).

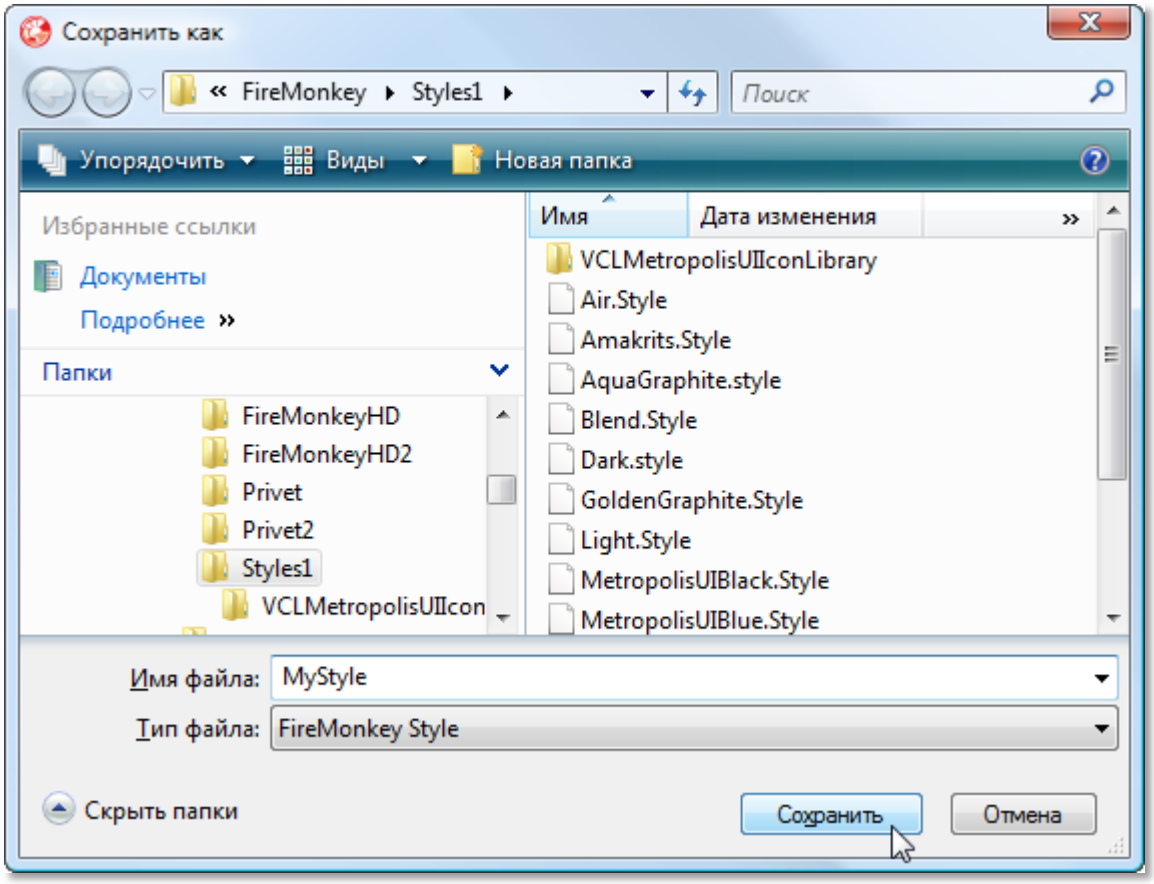


Рис. Ф9.37. Сохраняем стиль на диске

Нажмите кнопку **Apple and Close** – все кнопки тут же примут новый облик (Рис. Ф9.38).



Рис. Ф9.38. Новый стиль для кнопок

Но вы можете назначить *любой* кнопке *индивидуальный* стиль. Для этого выделите её на форме и в *Инспекторе объектов* нажмите строчку **StyleLookup**. Откроется список доступных стилей для кнопки (Рис. Ф9.39).

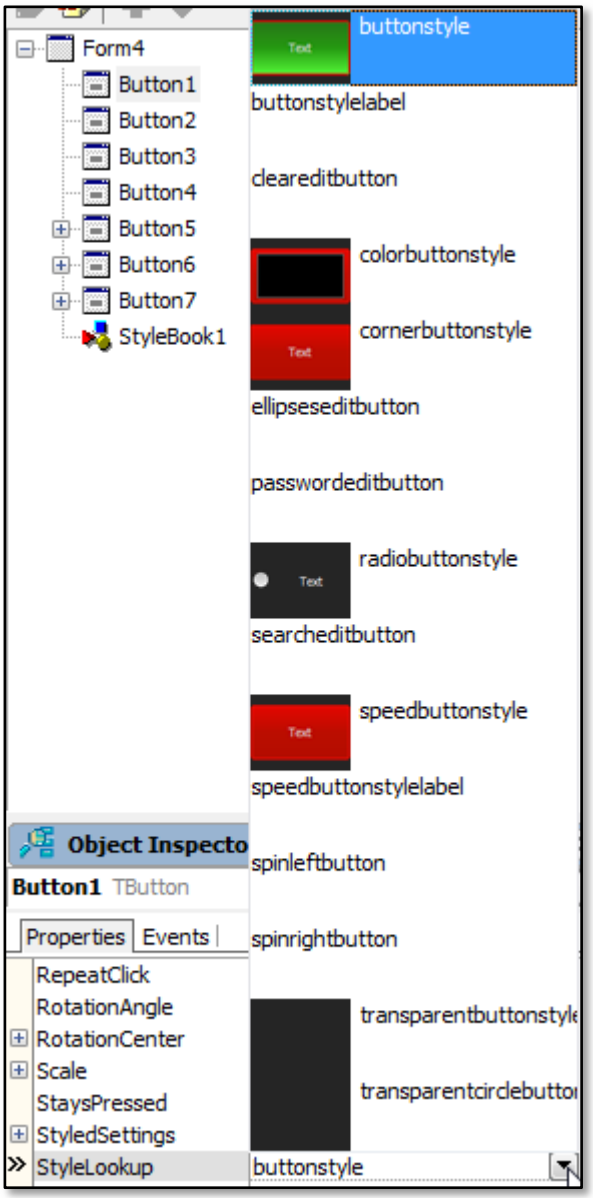


Рис. Ф9.39. Список кнопочных стилей

Сейчас все кнопки связаны со стилем *buttonstyle*, который мы с вами разработали. Выберите для кнопок другие стили (Рис. Ф9.40).

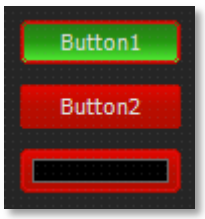


Рис. Ф9.40. Разностильные кнопки

Дальше вы можете самостоятельно дорабатывать стили всех элементов управления, а затем применять их в своих проектах.

Выходим в свободное пространство!

Крутится, вертится шар голубой,
Крутится, вертится над головой…

Песня из фильма *Юность Максима*

Термин **3D** (*трёхмерный*) сейчас не менее популярен, чем *нанотехнологии*, поэтому мы воздадим должное *основами* трёхмерной графики, которую можно создавать средствами *Delphi XE3* и *FireMonkey*.

Перво-наперво нам нужно создать новый проект **3D FireMonkey Application**. Как это сделать, вы уже прекрасно знаете. В окне *Конструктора формы* возникнет новая форма, и вы можете изменить её заголовок и цвет фона, например так, как показано на Рис. Ф9.41.



Рис. Ф9.41. Форма для трёхмерной графики

Перетащите на неё любые объекты со страницы **3D Shapes**. Но для первого знакомства лучше выбрать знакомые с детства предметы – *кубики* (компонент **TCube**) и *шарик* (компонент **TSphere**) (Рис. Ф9.42).

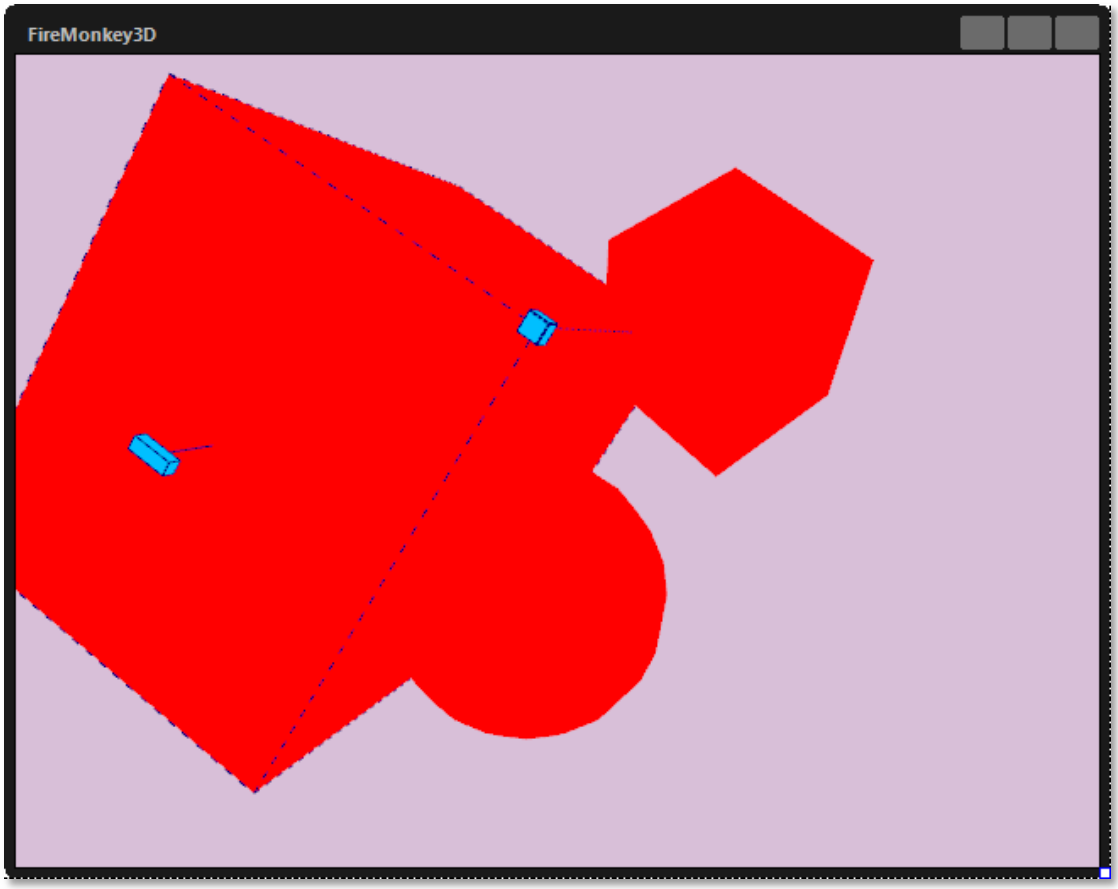


Рис. Ф9.42. Трёхмерные объекты выходят на сцену

Их свойства (размер, положение в пространстве, поворот относительно осей координат) можно изменить в *Инспекторе объектов* (Рис. Ф9.43).

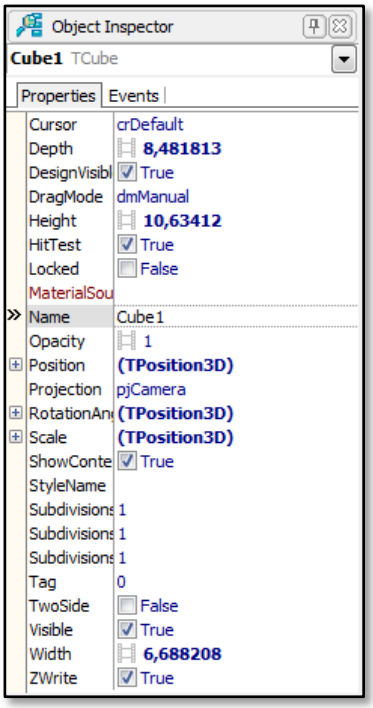


Рис. Ф9.43. Список свойств куба

Но гораздо интереснее изменять их свойства *интерактивно*, то есть непосредственно на форме. Щёлкните по какому-нибудь объекту, что выделить его. Вокруг активного объекта появятся пунктирные габаритные линии и синие узелки. Они бывают двух видов – *параллелепипеды* позволяют *вращать* объект вокруг одной из осей координат, а *кубик* – *масштабировать* его вдоль этих осей.

Для поворота объекта нажмите продолговатый узелок. Из синего он превратится в красный (Рис. Ф9.44).

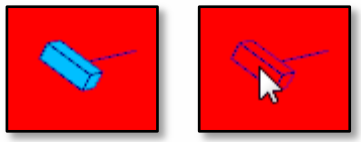


Рис. Ф9.44. Объект готов к повороту

«Вытянутость» узелка показывает *направление* вращения объекта. Двигайте мышку и наблюдайте за его реакцией.

Для масштабирования объекта установите мышку на одной из граней узелка-кубика. Грань кубика покраснеет, а грань самого объекта (или габаритного параллелепипеда), наоборот, станет синей (Рис. Ф9.45).

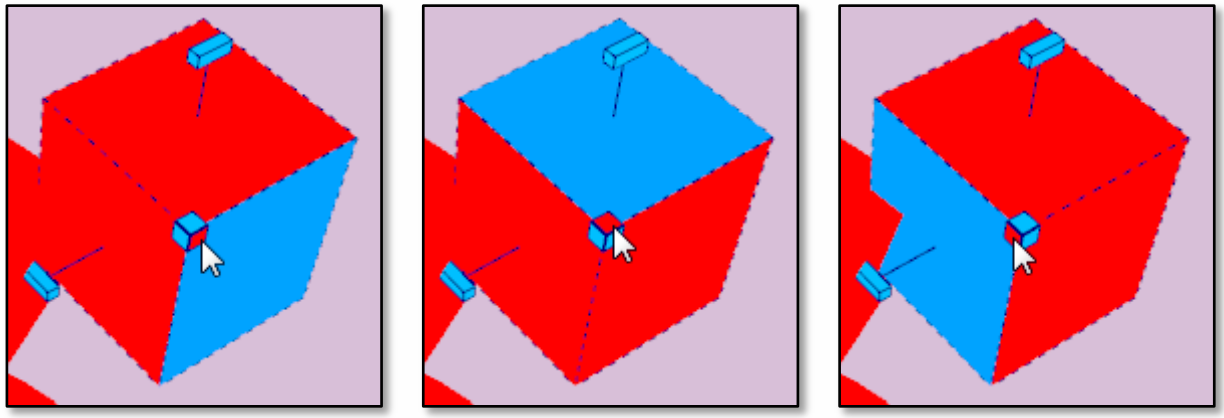


Рис. Ф9.45. Объект готов к масштабированию

Передвигая мышку в разные стороны, вы легко измените размеры объекта – ширину, высоту и глубину.

Позицию объекта на сцене изменить ещё проще – возьмитесь мышкой за его «тело» и перемещайте объект куда угодно.

Если вы сейчас запустите приложение, то трёхмерные объекты разочаруют вас своим невыразительным, плоским видом (Рис. Ф9.46).

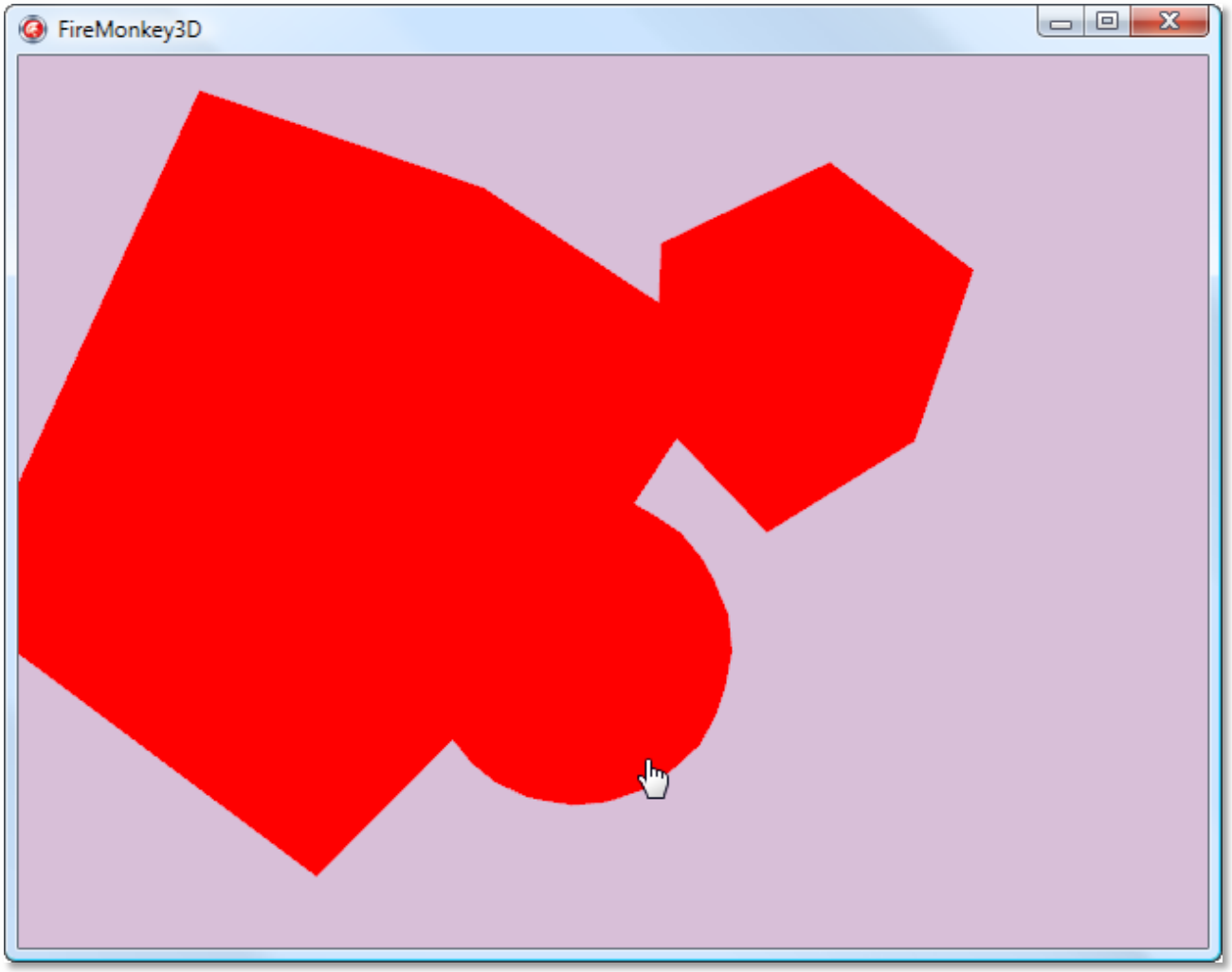


Рис. Ф9.46. Красно, но не прекрасно!

А всё потому, что материал объектов по умолчанию окрашен в ровный красный цвет, поэтому все предметы и кажутся плоскими.

Как мы уже знаем, на странице **Materials** имеется 3 вида «источников» материала для трёхмерных объектов. *Первый* материал - **TColorMaterialSource** - окрашивает поверхность объекта в заданный цвет. Это неинтересно, поэтому этот материал мы использовать не будем. *Второй* материал – **TTextureMaterialSource** – накладывает изображение на поверхность объекта. *Третий* материал - **TLightMaterialSource** – освещает объект текстурированным или окрашенным светом.

Чтобы изучить действие этих материалов, перетащите указанные компоненты на форму. Они будут добавлены в список компонентов в окне *Структуры проекта* (Рис Ф9.47).

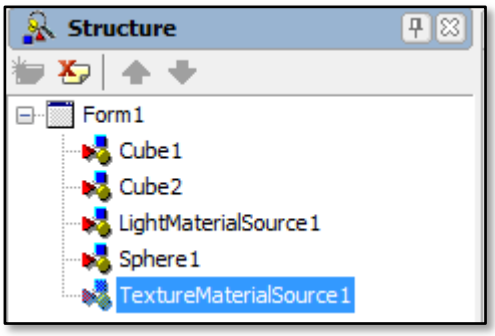


Рис. Ф9.47. Добавляем материализма!

Последовательно выделяйте эти компоненты в списке и в их свойство **Texture** загружайте какое-нибудь растровое изображение (Рис. Ф9.48).

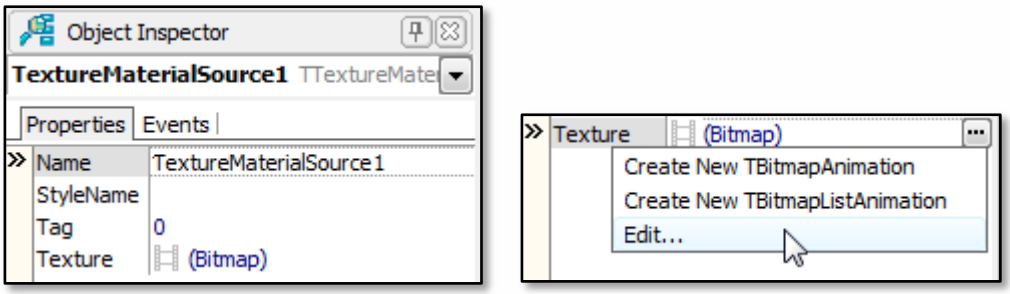


Рис. Ф9.48. Загружаем текстуру

Как это сделать, вы уже знаете.

Не удивляйтесь, что поверхность объектов не изменилась – сами по себе источники материала не действуют на объекты. Мы должны каждый объект *связать* с каким-либо источником материала.

Выделите объект на форме или в окне *Структуры проекта*, а в *Инспекторе объектов* щёлкните по свойству **MaterialSource**. Откроется список доступных материалов (Рис. Ф9.49).

Назначьте первому кубу *текстурный* материал, а второму кубу и шару – *световой*. Уже в режиме проектирования мы увидим, что на поверхности первого куба появилось изображение, а вот второй куб и шар стали чёрными (Рис. Ф9.50).

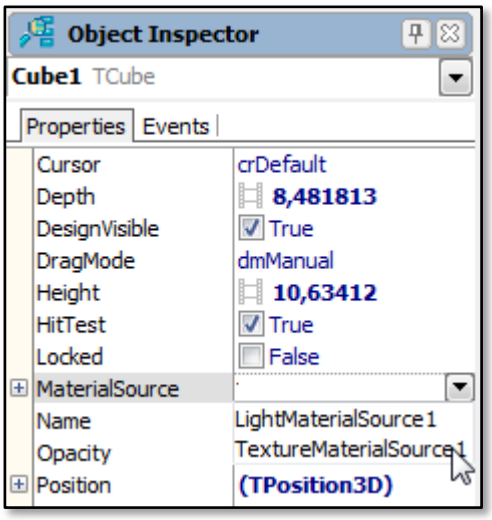


Рис. Ф9.49. Список материалов

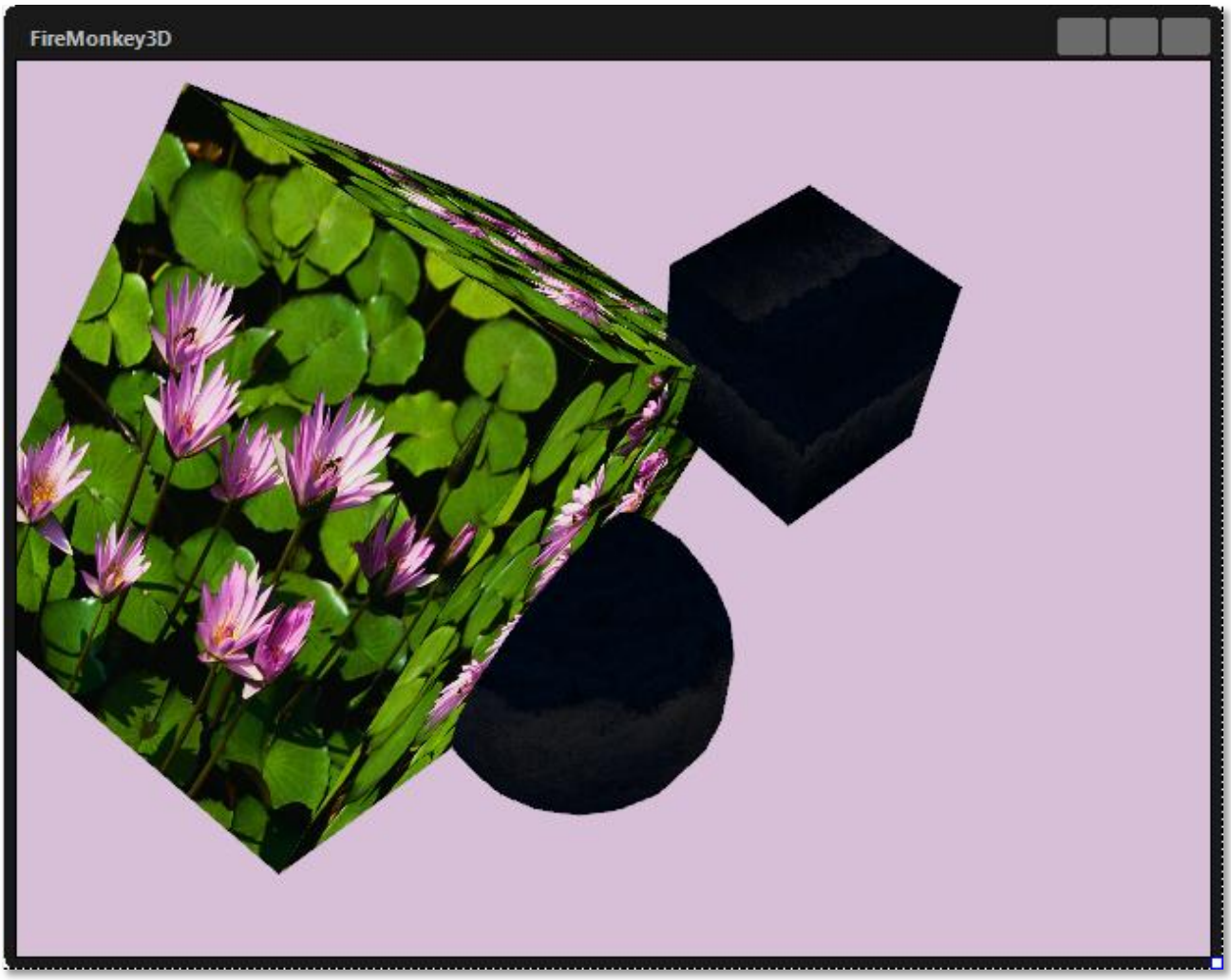


Рис. Ф9.50. Материализованные объекты

Чтобы «световой» материал проявился, необходимо добавить на форму *источник света* – компонент **TLight** (Рис. Ф9.51).

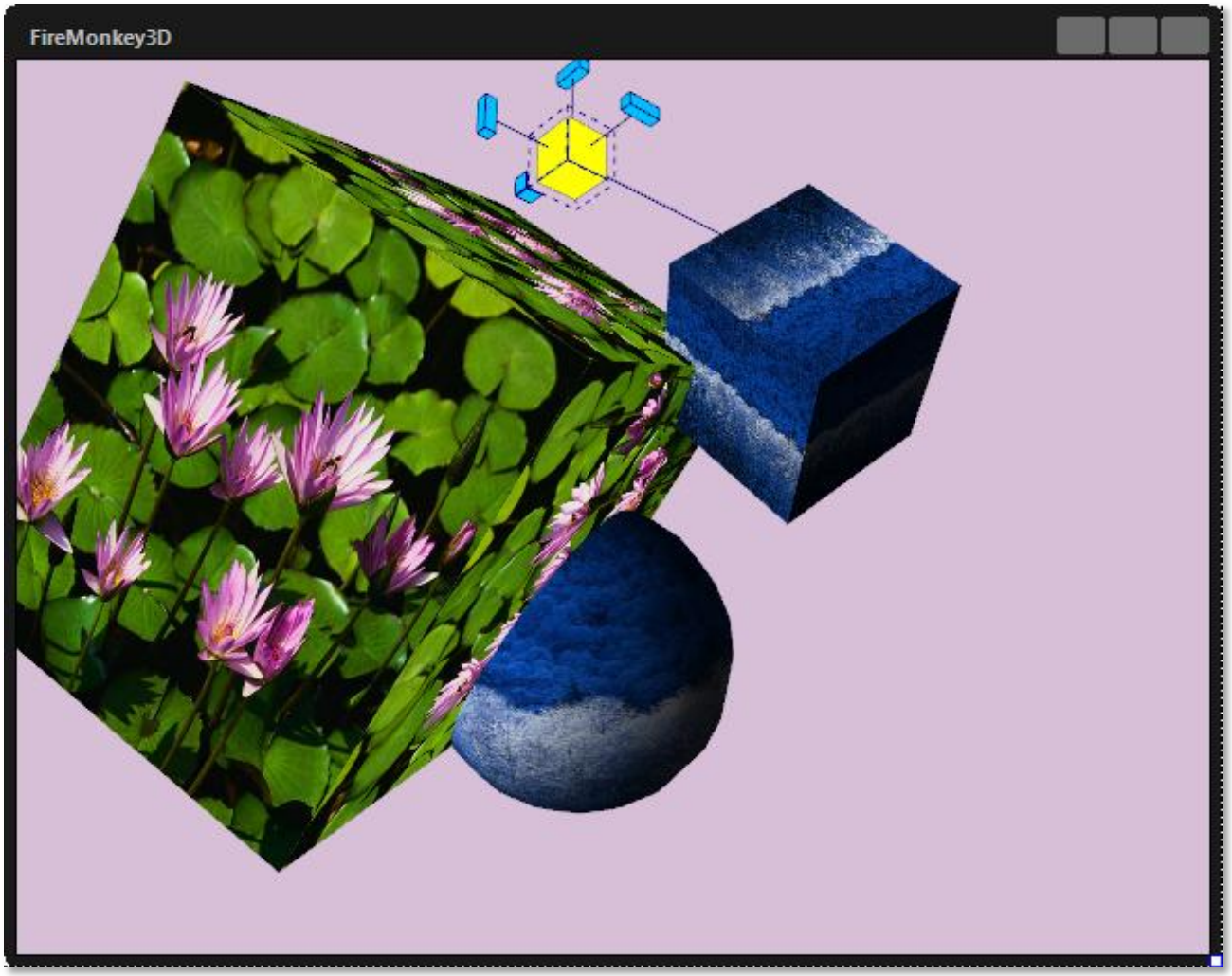
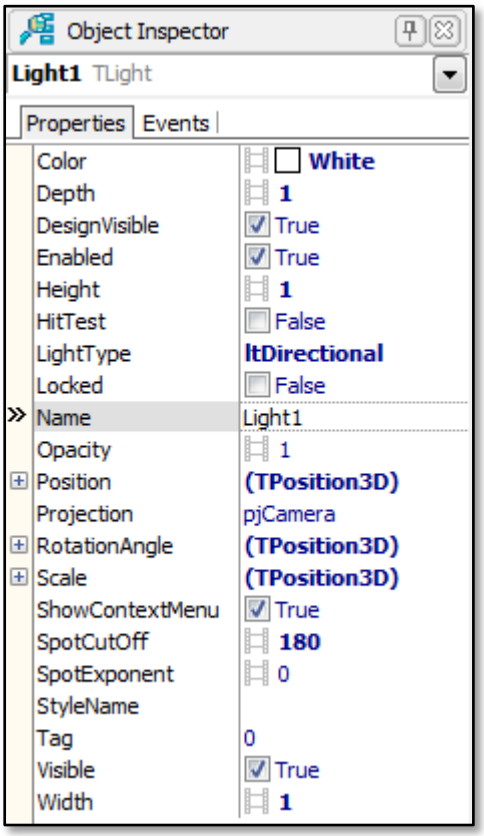


Рис. Ф9.51. Да будет свет!



Его также можно перемещать и поворачивать, как и любые другие объекты на сцене. При этом изменяется освещённость разных частей объектов со световыми материалами.

В *Инспекторе объектов* вы можете выбрать *тип* источника освещения и *цвет* его лучей (Рис. Ф9.52).

Рис. Ф9.52. Список свойств источника освещения

Запускаем приложение – трёхмерные объекты смотрятся замечательно (Рис. Ф9.53)!

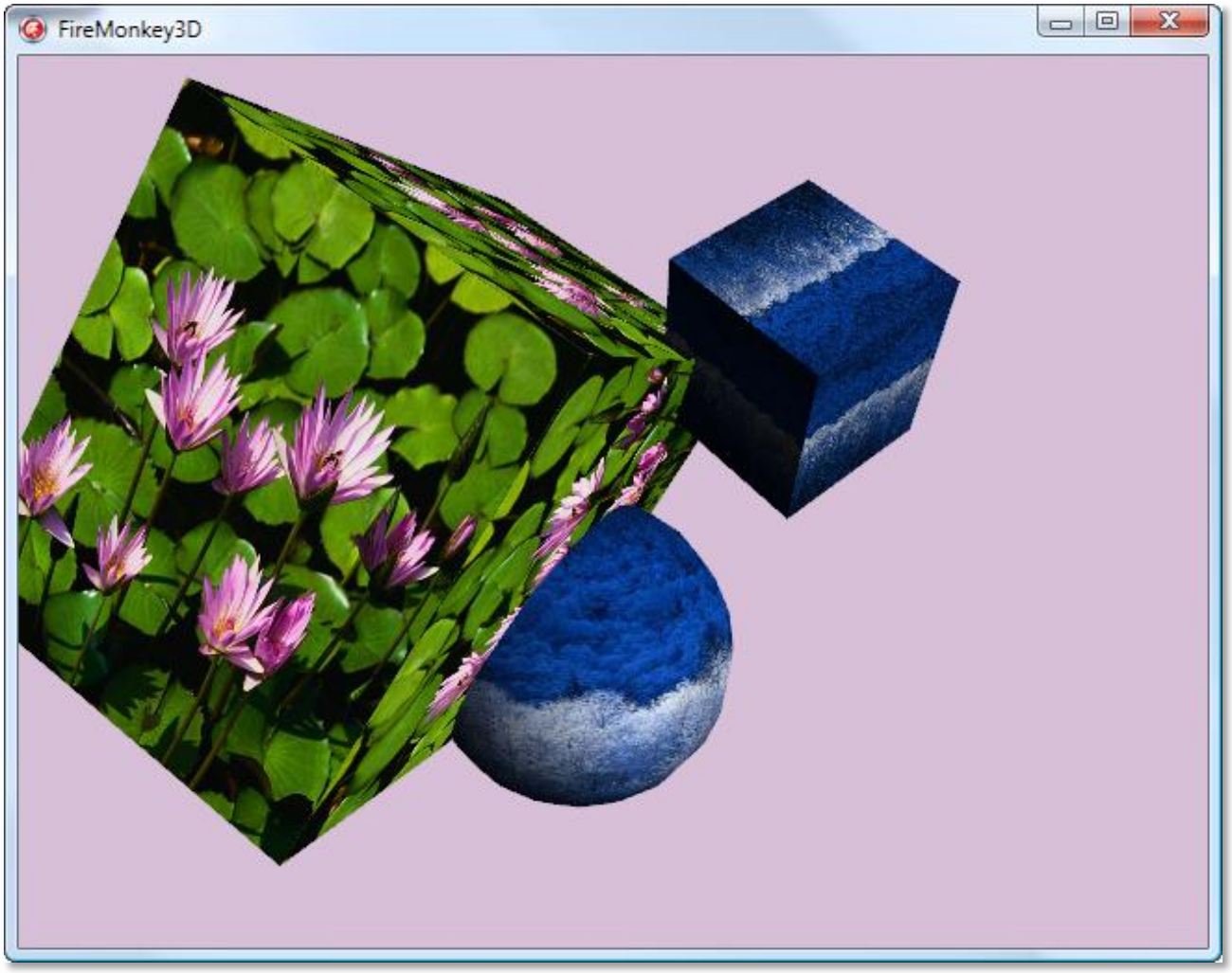


Рис. Ф9.53. Трёхмерно!

И нам остаётся разве что добавить им живинки, то есть закрутить их по всем осям. Это можно сделать и вручную, установив на форме движки, но мы поступим более хитроумно и делегируем наши вращательные обязанности **таймеру**.

Задайте его свойству **Interval** значение 100 и щёлкните пару раз по строчке **Timer1** в *Структуре проекта*, чтобы открылся метод **Timer1Timer**.

Здесь вы можете крутить объекты, как вам угодно, просто изменяя значения их свойств **RotationAngle.X, Y, Z**:

```
procedure TForm2.Timer1Timer(Sender: TObject);
begin
  Cube1.RotationAngle.X := Cube1.RotationAngle.x+3;
  Cube1.RotationAngle.Y := Cube1.RotationAngle.Y+2;
```

```
  Cube1.RotationAngle.Z := Cube1.RotationAngle.Z+5;
  Cube2.RotationAngle.Z := Cube2.RotationAngle.Z+6;
  Sphere1.RotationAngle.Y := Sphere1.RotationAngle.Y + 7;
end;
```

Скорость вращения объектов зависит от частоты срабатывания таймера и от приращения углов.

Я намеренно (то есть умышленно) пододвинул объекты друг к другу так, чтобы они *пересекались*. Запустите приложение – оно работает достаточно быстро и плавно, а объекты пересекаются по всем физическим и геометрическим законам природы (Рис. Ф9.54).

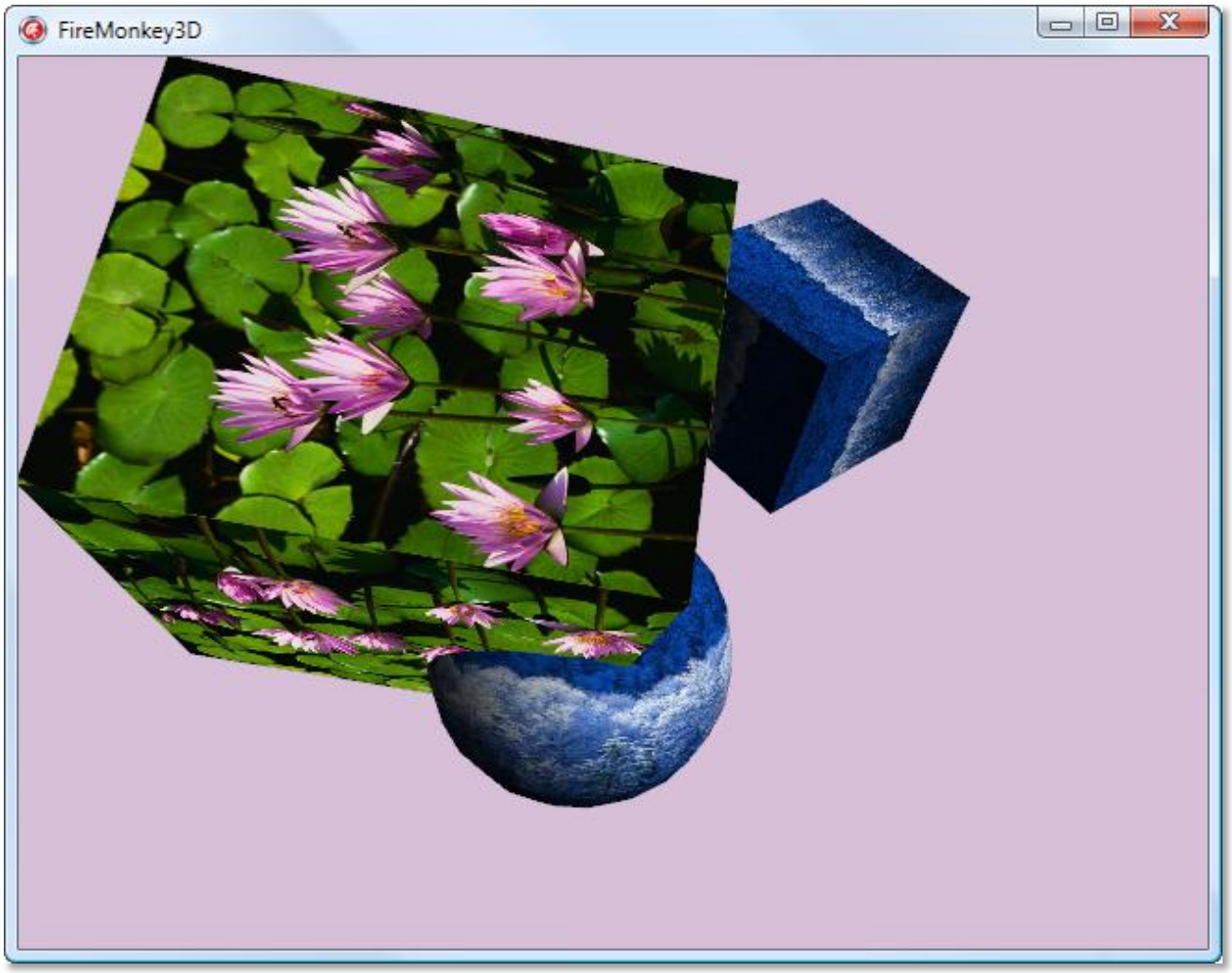


Рис. Ф9.54. Закрутилось!

При желании вы можете *объединить* несколько объектов в одну группу, перетаскивая их мышкой в окне *Структуры проекта* на «главный» объект (Рис. Ф9.55).

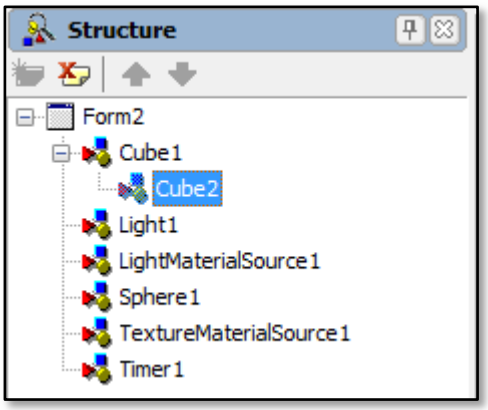


Рис. Ф9.55. Группируем объекты

Например, второй куб может превратиться в «спутника» первого куба и вращаться одновременно и с ним, и сам по себе. Зрелище весьма интересное и напоминает вращение Луны вокруг Земли (Рис. Ф9.56)!

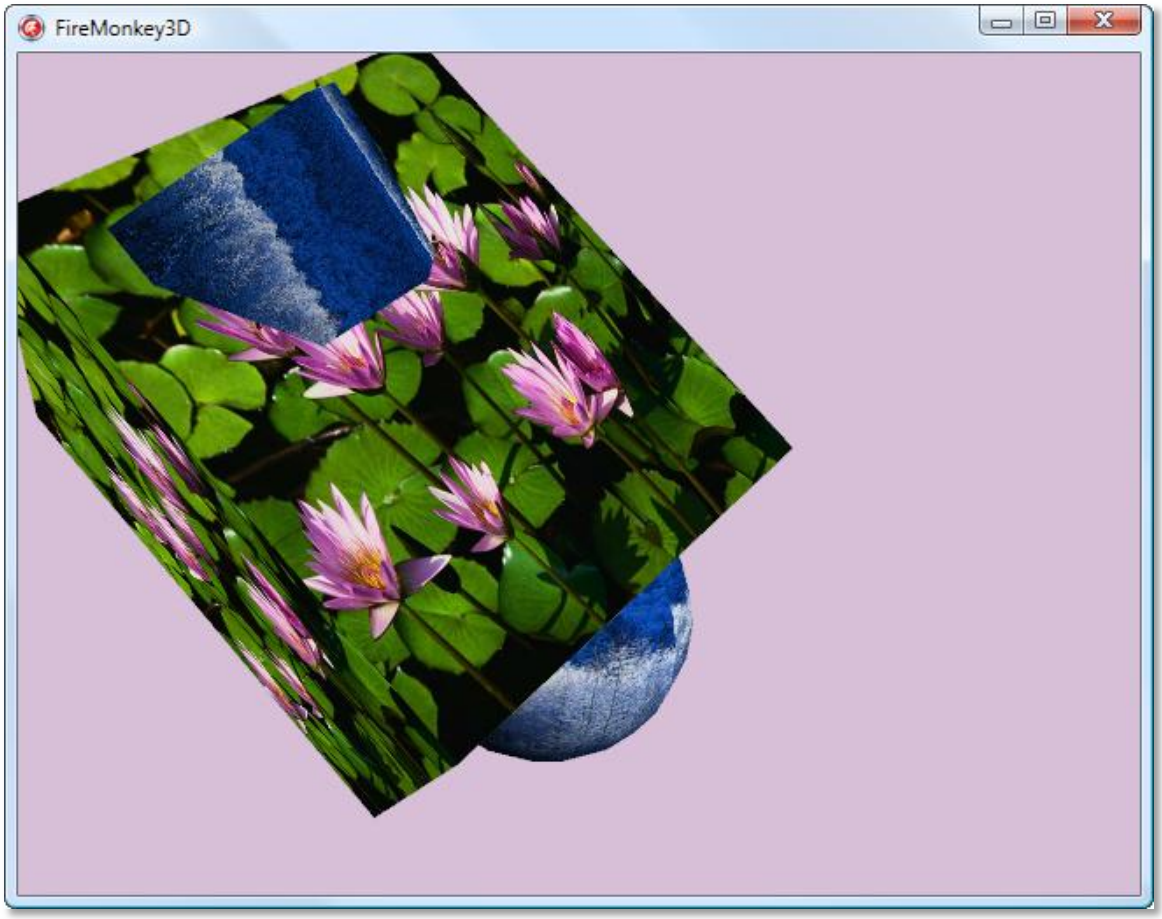


Рис. Ф9.56. Обоюдовращательные движения объектов

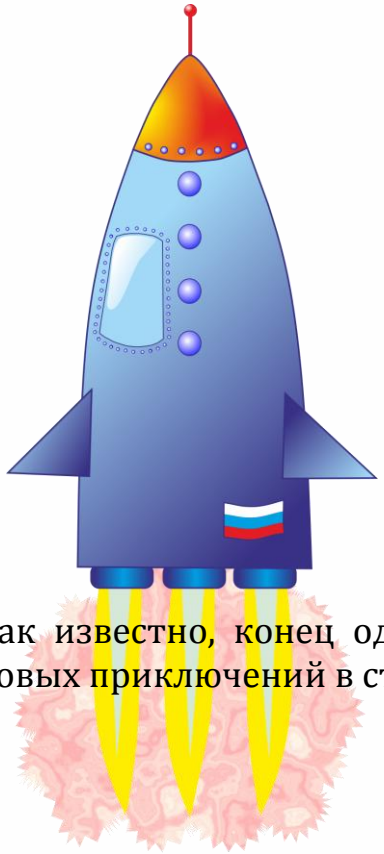


Исходный код программы находится в папке **FireMonkey > FireMonkey3D1**.

Вы также можете *анимировать* многие свойства объектов и материалов. Или заставить их взаимодействовать друг с другом и пользователем программы. Кроме геометрических примитивов, вы можете загружать и сложные модели, созданные в трёхмерных графических редакторах, например, в *3D Studio Max*. Но, согласитесь, даже краткий рассказ об этих возможностях *Delphi XE3* потребовал бы отдельной **толстой** книги, а *эта* книга, наконец-то,

Закончилась!

Happy END., или Карлсоны всегда возвращаются



- Шура, это конец!
- Это конец первой серии, студент!

Достойный ответ Остапа Бендера
меланхоликам и пессимистам

Сколько верёвочке ни виться,
а конец будет.

Уходя, гасите свет.

Житейское

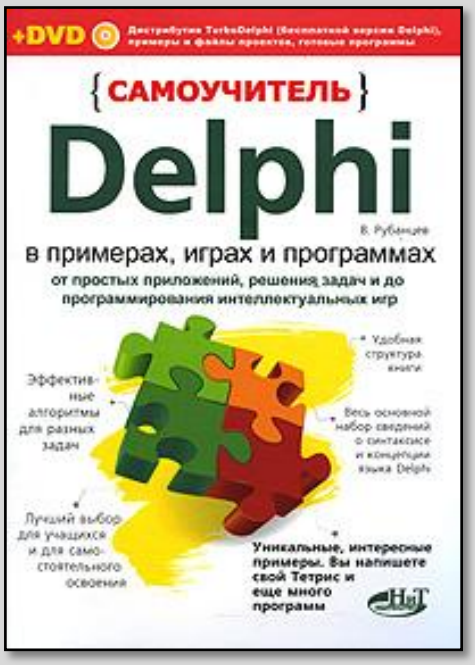
Как известно, конец одного путешествия – это начало следующего! До новых приключений в стране *Delphi*!

Вперёд – труба зовёт!

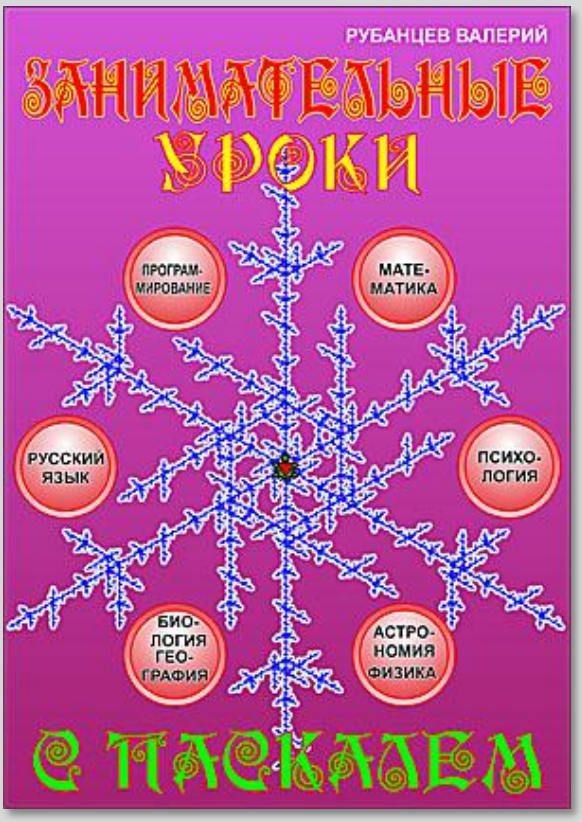


Он улетел, но он обещал вернуться!

Литература

[PB10]		*Рубанцев Валерий* **Хитори**: круче, чем судоку Эксмо-Пресс, 2010. – 288 с. ISBN: 978-5-699-39575-0 Серия: Зарядка для ума
[PB11]		*Рубанцев Валерий* **Delphi в примерах, играх и программах.** От простых приложений, решения задач и до программирования интеллектуальных игр Наука и Техника, 2011. – 672 с. ISBN: 978-5-94387-664-6 Серия: Самоучитель

В октябре 2012 года будут выпущены следующие *электронные* книги:



Рубанцев Валерий

Занимательные уроки с паскалем

Я + R, 2013. – 690 с.

Учитесь программировать с новой версией паскаля *PascalABC.NET 1.8*!

Книгу можно скачать с сайта **rvGames.de**.



Рубанцев Валерий

Занимательные уроки с компьютером, или Small Basic для начинающих

Я + R, 2013. – 570 с.

Учитесь программировать с первой официальной версией *Small Basic 1.0*!

Книгу можно скачать с сайта **rvGames.de**.

Готовятся к изданию и будут выпущены в 2012-13 годах следующие *электронные* книги:

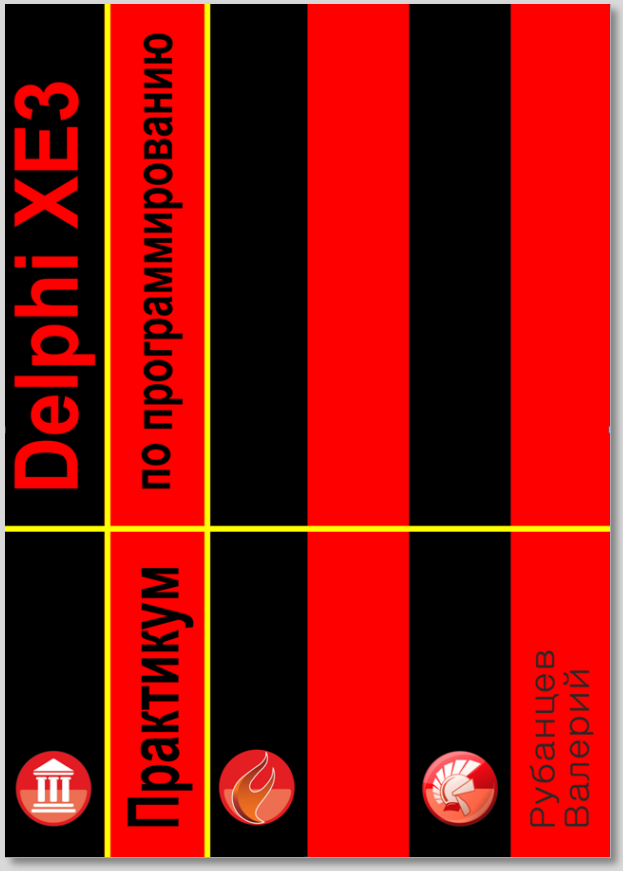


Рубанцев Валерий

Новейший самоучитель Delphi XE3

Я + R, 2013. – 400 с.

Главное о новинках версии *Delphi XE3*: платформа *FireMonkey*, *обобщения* (*Generics*), *стиль метро Windows 8* (*Metropolis UI).*



Рубанцев Валерий

Практикум по программированию на Delphi XE3 в занимательных проектах для начинающих

Я + R, 2013. – 300 с.

Практические занятия к книге *Новейший самоучитель Delphi XE3*: занимательные проекты *FireMonkey* - словесные, криптографические, числовые и комбинаторные.



Рубанцев Валерий

Занимательная графика на Си-шарпе. Самоучитель для начинающих

Я + R, 2013. – 746 с.

Современная графика на занимательных примерах и полный справочник по *GDI+*.



Рубанцев Валерий

Тотальный тренинг по Си-шарпу. Для начинающих

Я + R, 2013. – 613 с.

Закрепляем теоретические знания на практических примерах!

		*Рубанцев Валерий* **Как решать комбинаторные задачи на компьютере** Я + R, 2013. – 367 с. Жемчужины комбинаторики на Си-шарпе!
		*Рубанцев Валерий* **Как решать головоломные задачи на компьютере** Я + R, 2013. – 468 с. Решаем какуро, судоку, лабиринты, сквэрворды, кен-кен, фобидоши и другие замечательные головоломки на Си-шарпе!
	Обложка находится в работе	*Рубанцев Валерий* **Си-шарп для начинающих. Подробный самоучитель**

		Я + R, 2013. – Более 500 с.
	Обложка находится в работе	*Рубанцев Валерий* **Практикум по программированию на языке Си-шарп для начинающих** Я + R, 2013. – 373 с.
	Обложка находится в работе	*Рубанцев Валерий* **Большой самоучитель-справочник по Си-шарпу** Я + R, 2013. – Более 700 с.
	Обложка находится в работе	*Рубанцев Валерий* **Processing – Java для начинающих. Подробный самоучитель** Я + R, 2013. – Более 700 с.

	Обложка находится в работе	*Рубанцев Валерий* **Компьютерные игры, симуляторы и модели. Практика программирования на языке Java для начинающих** Я + R, 2013. – 438 с.

Подробную информацию о книгах и сроках их выпуска вы найдёте на сайте **rvGames.de**. О выходе *печатных* книг мы сообщим дополнительно.

