РУБАНЦЕВ ВАЛЕРИЙ

PASCALABC.NET

Программирование графики на примерах

Валерий Рубанцев

PascalABC.NET

Программирование графики на примерах

Бесплатное издание детскиекниги.рф

От автора

Лучше один раз увидеть,
чем сто раз услышать.
Программистская мудрость

Обучение должно быть **наглядным**

Наглядность – один из самых известных дидактических принципов - требует, чтобы при обучении использовались модели, изображения или чертежи, которые можно *увидеть*.

Действительно, красивая картинка, созданная своими руками (и головой!) на экране компьютера, воздействует на человеческий ум куда сильнее, чем решение квадратного уравнения или рассказ о том, как Галилей сбрасывал ядра с Пизанской башни.

Гораздо интереснее и легче учиться считать на пальцах или на счётах, чем в уме или даже в тетради (но в ней-то, по крайней мере, можно нарисовать палочки, кружочки, яблоки или мешки с картошкой).

Картошку использовал и легендарный начдив Василий Иванович Чапаев для наглядного объяснения подчинённым плана предстоящего наступления (Рис. 1).

Рис. 1. Кадр из фильма *Чапаев*

Обучение должно быть **систематическим** и **последовательным**

При этом не следует забывать, что наглядность – не самоцель, а лишь одно из средств обучения. Все языки программирования, делающие ставку на графические приложения, имеют огромное число принципиальных недостатков, которые неизбежно создадут проблемы при переходе на более сложные языки программирования.

Обучение должно быть **прочным**

Большинство книг по программированию, а тем более для начинающих ограничиваются 160-250 страницами. Такое поверхностное обучение методом «галопом по Европам», без обстоятельных объяснений и многочисленных примеров годится только для бездумного копирования материала этих самоучителей, но никак не способствует самостоятельной и сознательной деятельности.

Обучение должно быть **доступным** и **посильным**

Приступать к разработке графических приложений следует только после усвоения основ программирования: переменных, констант, типов данных, операторов, методов, циклов и так далее.

Всем известно, что программирование графики, будь то *GDI+*, *OpenGL* или *DirectX* – непростая задача. Поэтому **цель** этой книги: показать начинающим программистам, что разрабатывать современные графические программы им вполне по силам!

Для этого мы будем использовать графическую библиотеку из программы *Смолл Бейсик* **SmallBasicLibrary**, а также дополнительные библиотеки, расширяющие её возможности. И при этом мы будем писать приложения на настоящем *паскале!*

В книге вас ждут многочисленные увлекательные и познавательные проекты. Вы **научитесь**:

- Создавать и настраивать *Графическое окно* приложения
- Рисовать пикселями
- Чертить простые геометрические фигуры: прямые линии, треугольники, прямоугольники, эллипсы
- Создавать элементы управления: метки, кнопки, текстовые поля
- Обрабатывать нажатия на клавиши и кнопки и перемещения мышки

- «Фильтровать» картинки
- Окрашивать поверхности и линии в разные цвета
- Дрессировать *Черепашку*, чтобы она вычерчивала замысловатые кривые, а затем и *КиберЧерепашку*, которая, повинуясь инстинктам, нарисует настоящие фракталы
- Разрабатывать проекты по школьным предметам: физике и астрономии, психологии и биологии
- Организовывать тараканьи бега по методу Монте-Карло и гадать не хуже осьминога Пауля

Я надеюсь, что компьютерная графика покажет вам программирование с новой, интересной стороны!

Валерий Рубанцев

Условные обозначения, принятые в книге:

Дополнение или замечание

Ненавязчивое требование или указание

```
Исходный  код
```

Задание для самостоятельного решения

Папка с исходным кодом **программы**.

Исходные коды всех проектов находятся в папке **_Projects**

Оглавление

Глава 1. Графический привет Миру

Запустите программу **PascalABC.NET** (Рис. 1.1).

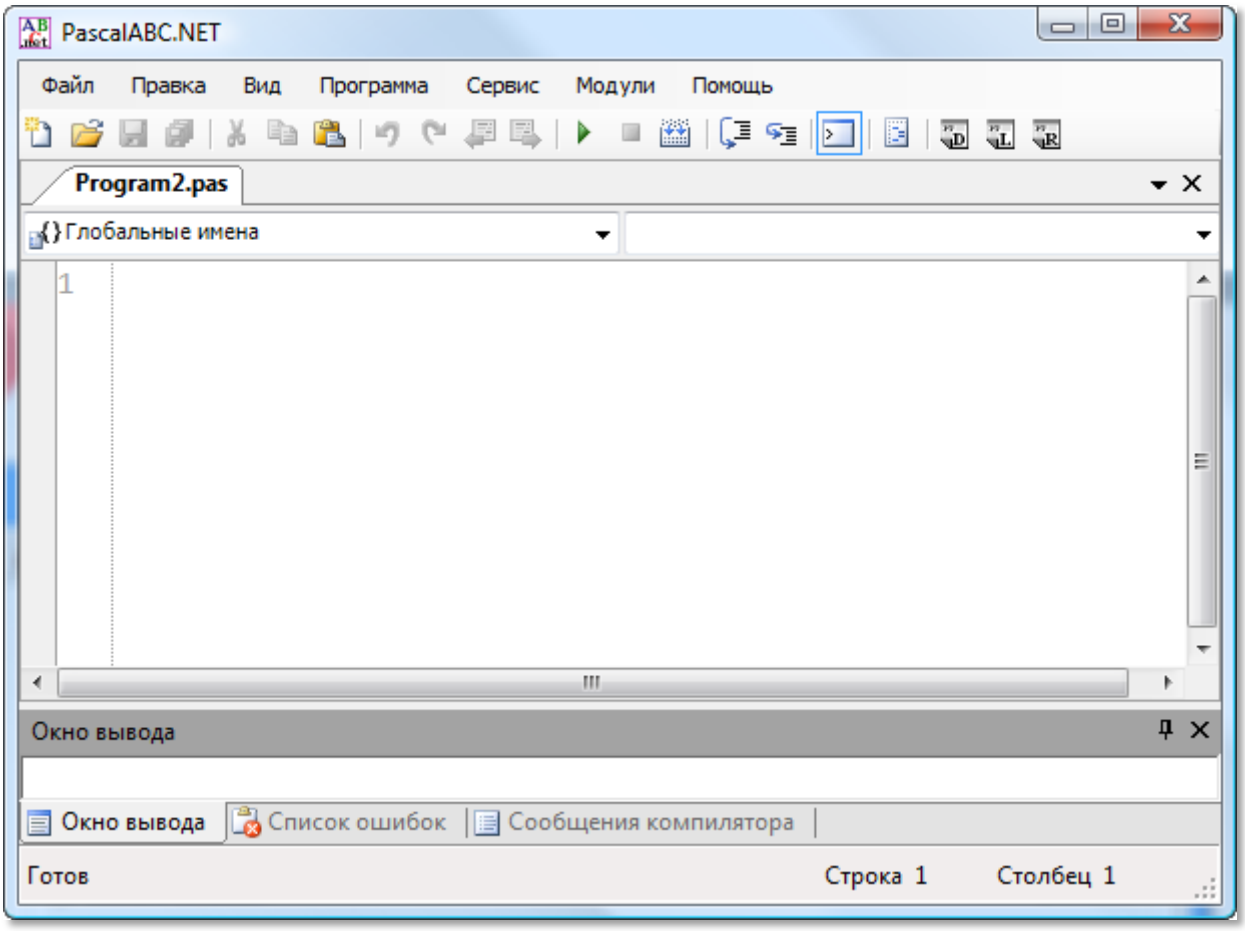


Рис. 1.1. Среда разработки

Начните новый проект, выполнив команду меню **Файл > Новый проект** (Рис. 1.2).

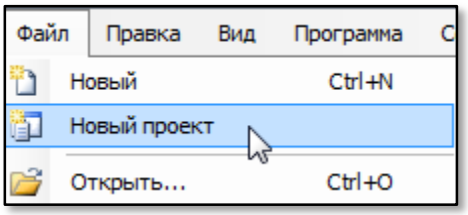


Рис. 1.2. Начинаем новый проект

В диалоговом окне *Новый проект* выберите тип проекта **Консольное приложение**, щёлкнув мышкой по его значку (Рис. 1.3).

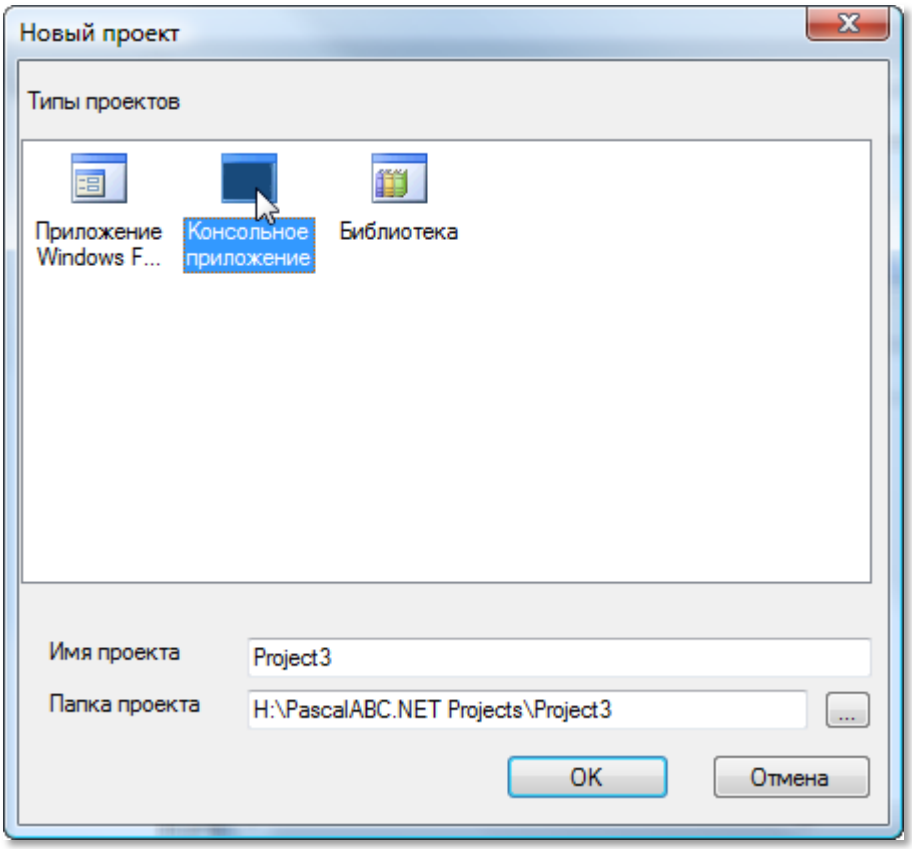


Рис. 1.3. Консольное приложение

В текстовом поле **Имя проекта** наберите имя проекта (Рис. 1.4).

В текстовом поле **Папка проекта** вы увидите полный путь к папке проекта по умолчанию. Обратите внимание, что папка называется так же, как и сам проект.

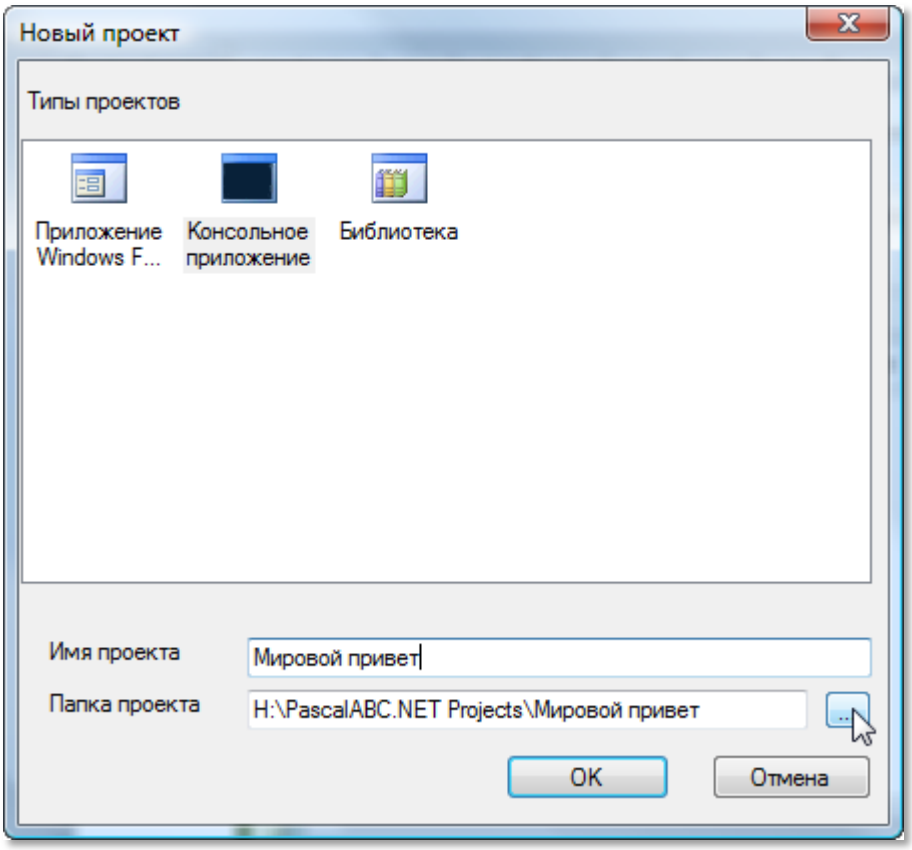


Рис. 1.4. Название проекта

Папку с проектами по умолчанию вы задали при установке программы *PascflABC.NET*. Если вы хотите сохранить проект в **другой** папке, то нажмите кнопку с тремя точками и в диалоговом окне **Обзор папок** перейдите в нужную папку (Рис. 1.5).

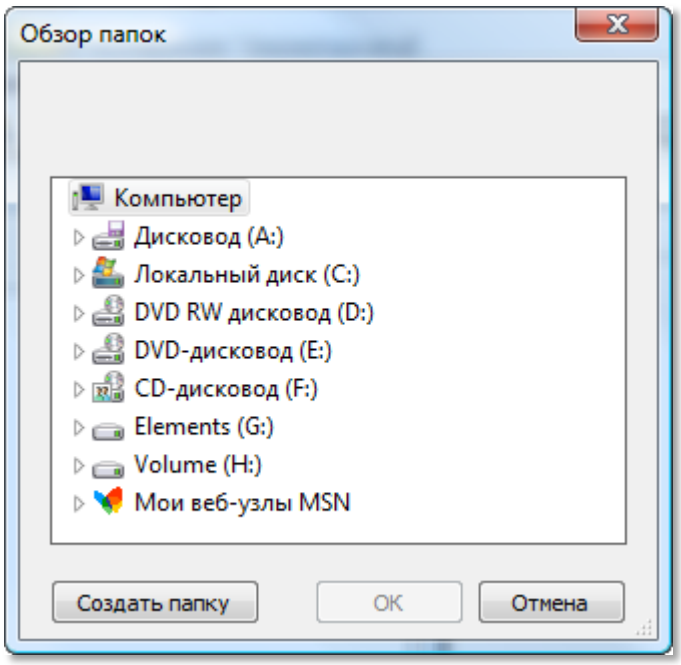


Рис. 1.5. Выбираем папку для проекта

После нажатия на кнопку **ОК** будет создана папка проекта, и в ней появится несколько файлов (Рис. 1.6).

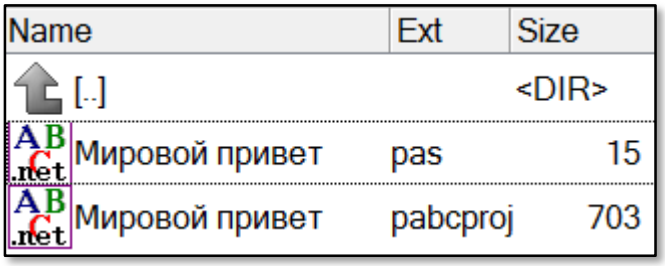


Рис. 1.6. Первые файлы проекта

Файл с расширением **pabcproj** – это файл проекта с информацией, которая необходима компилятору для создания приложения, а файл с расширением **pas** – это главный файл программы.

В нём сейчас записано то, что вы видите в *Редакторе кода* (Рис. 1.7), то есть операторные скобки главной программы.

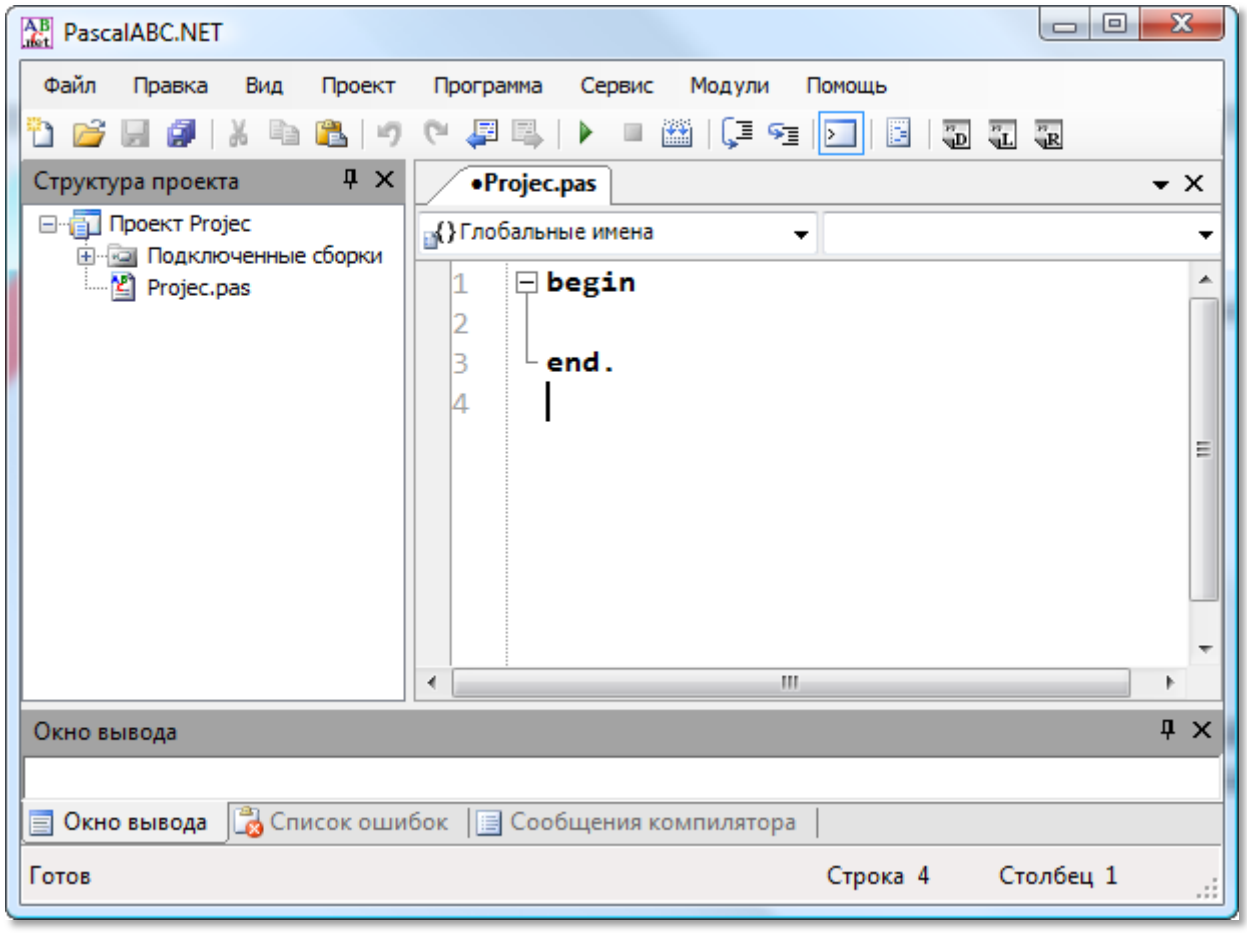


Рис. 1.7. Заготовка кода для проекта

В папку с программой нужно скопировать файл **SmallBasicLibrary.dll** с необходимой нам динамической библиотекой. Если вы скачали и установили *Смолл Бейсик*, то сможете найти этот файл в папке с этой программой и самостоятельно скопировать его в папку, а можете воспользоваться той библиотекой, что уже лежит во всех папках с проектами.

Разверните пункт **Подключенные сборки** в окне *Структура проекта*, нажав мышкой на узелок с плюсиком. Как вы видите, сейчас к проекту подключена только динамическая библиотека (сборка) *System* (Рис. 1.8).

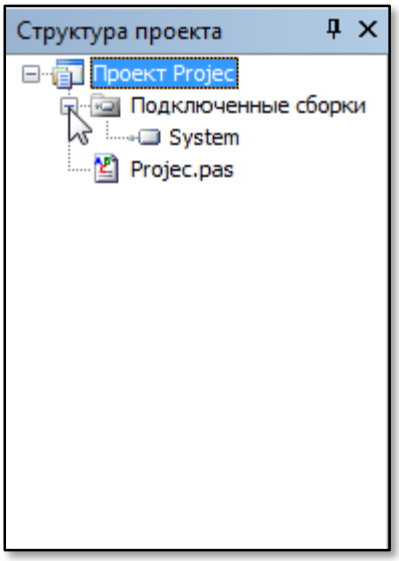


Рис. 1.8. Подключенные сборки

Она нужна во всех проектах, поэтому добавляется автоматически.

Теперь нам нужно подключить библиотеку *SmallBasicLibrary.dll*. Для этого нажмите правую кнопку мышки на строке *Подключенные сборки*, чтобы вызвать контекстное меню с единственным пунктом **Добавить сборку** (Рис. 1.9).

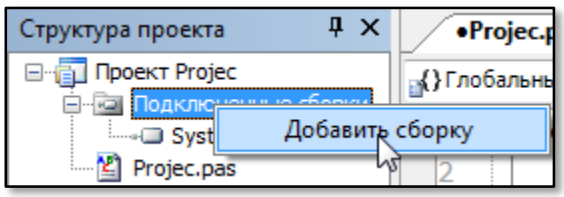


Рис. 1.9. Добавляем библиотеку

Щёлкните по нему, чтобы открыть диалоговое окно **Подключение сборок** (Рис. 1.10).

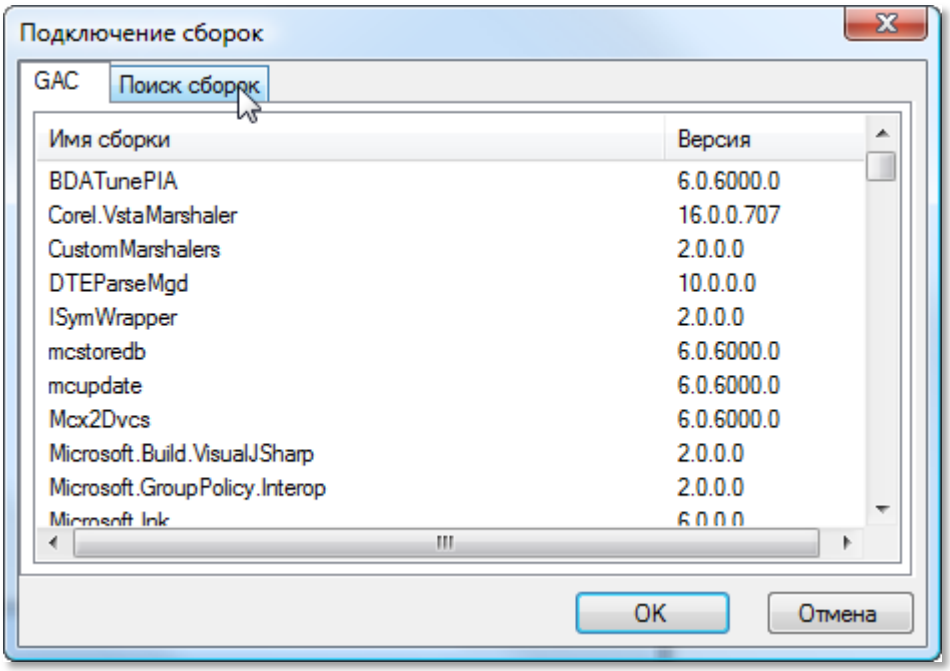


Рис. 1.10. Список доступных сборок

Вы можете вызвать это окно и командой меню **Проект > Подключить** сборку (Рис. 1.11).

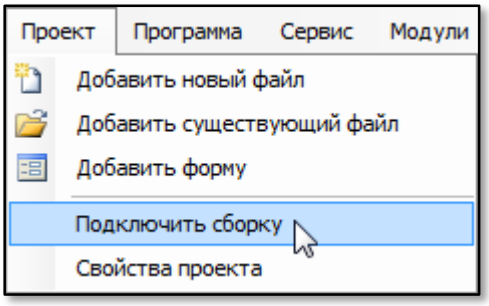


Рис. 1.11. Второй способ

А также командой контекстного меню проекта **Добавить сборку** (Рис. 1.12).

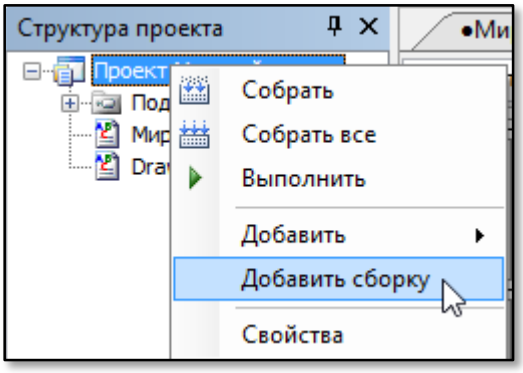


Рис. 1.12. Третий способ

В диалоговом окне *Подключение сборок* нажмите вкладку *Поиск сборок*, а затем кнопку **Поиск...** (Рис.).

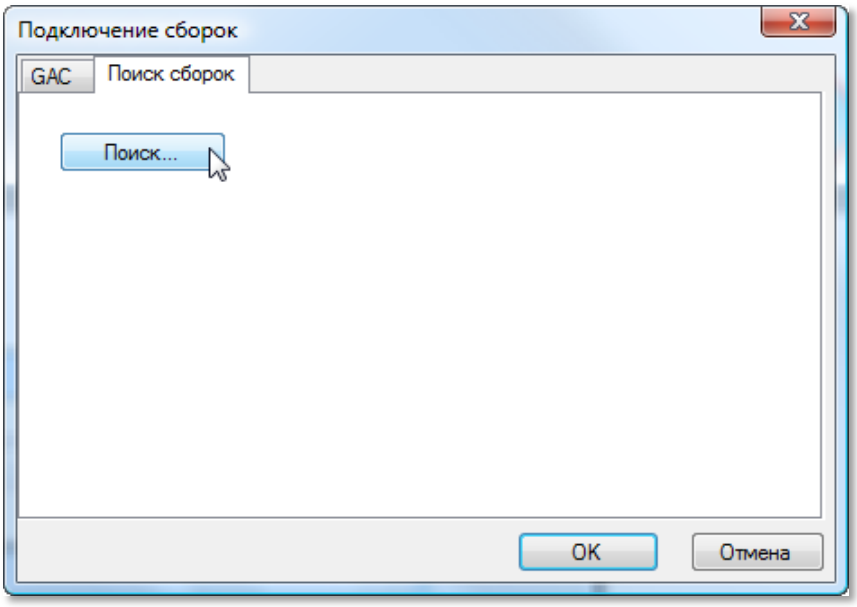


Рис. 1.13. Ищем сборку

Среда разработки откроет папку с проектом, в котором вы увидите скопированную ранее библиотеку. Выделите её мышкой и нажмите кнопку Открыть или просто дважды щёлкните по названию файла (Рис. 1.14).

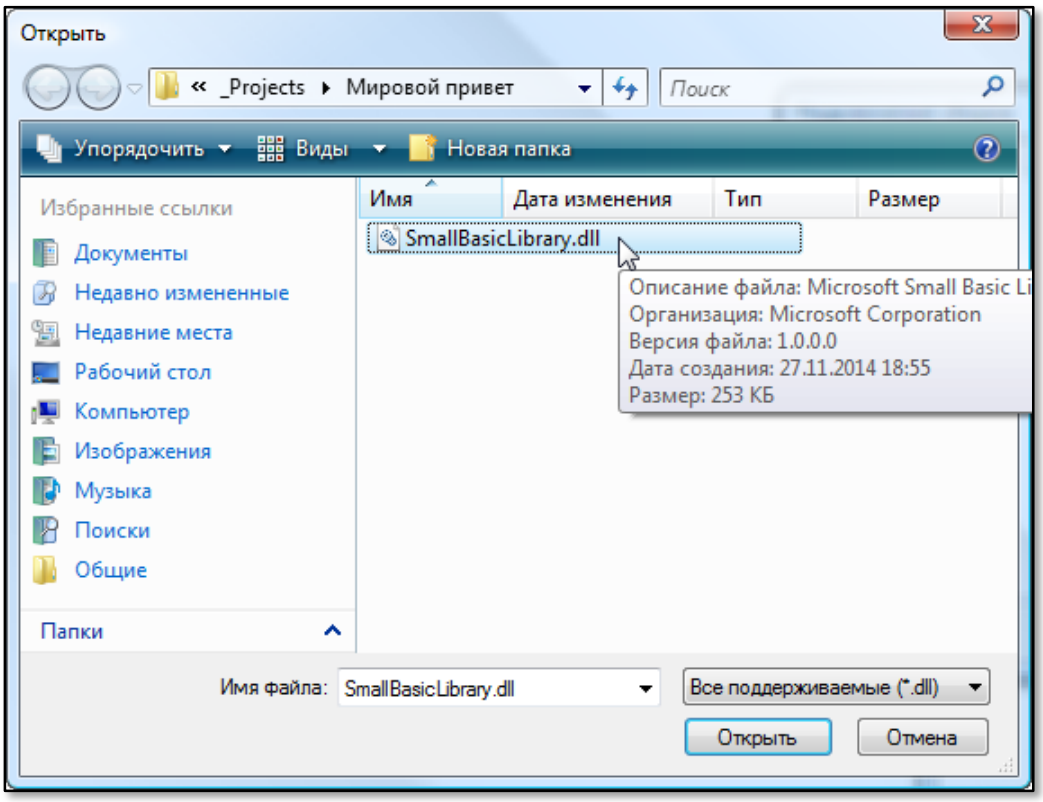


Рис. 1.14. Добрались до библиотеки

Убедитесь, что в окне *Структура проекта* появилась новая сборка (Рис. 1.15).

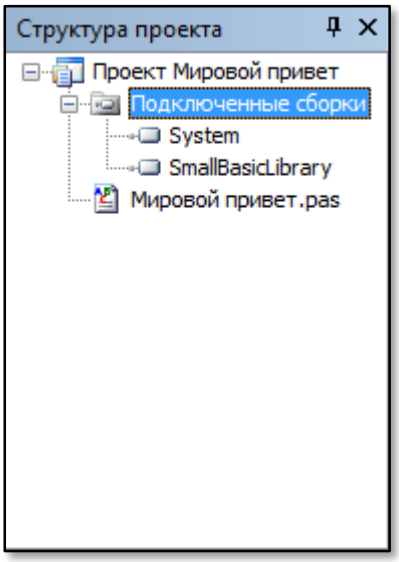


Рис. 1.15. Подключили!

Небольшие проекты вполне можно писать в процедурном стиле, но для нас графические программы важны не только как средство изучения компьютерной графики, но и как ещё одна возможность укрепить навыки **объектно-ориентированного программирования**. Впрочем, любые программы нужно писать правильно и грамотно – порядок есть порядок.

Поэтому в каждом приложении практически весь код будет сосредоточен в отдельном **классе**. А для каждого класса необходим новый **модуль**.

Чтобы добавить модуль к проекту, нажмите правую кнопку мышки на названии проекта в окне *Структуры проекта* и в контекстном меню выполните команду **Добавить > Новый файл** (Рис. 1.16).

В диалоговом окне *Новый файл* выделен шаблон **Модуль**, поскольку других шаблонов нет (Рис. 1.17).

В текстовом поле *Имя файла* введите имя модуля – **DrawUnit -** и нажмите кнопку **ОК**.

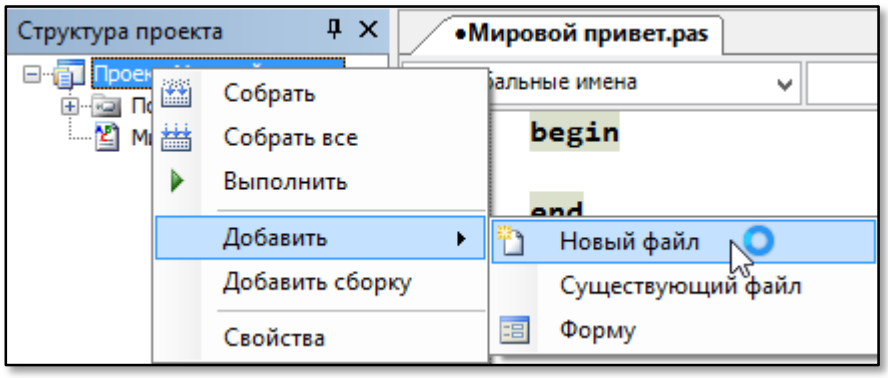


Рис. 1.16. Добавляем файл к проекту

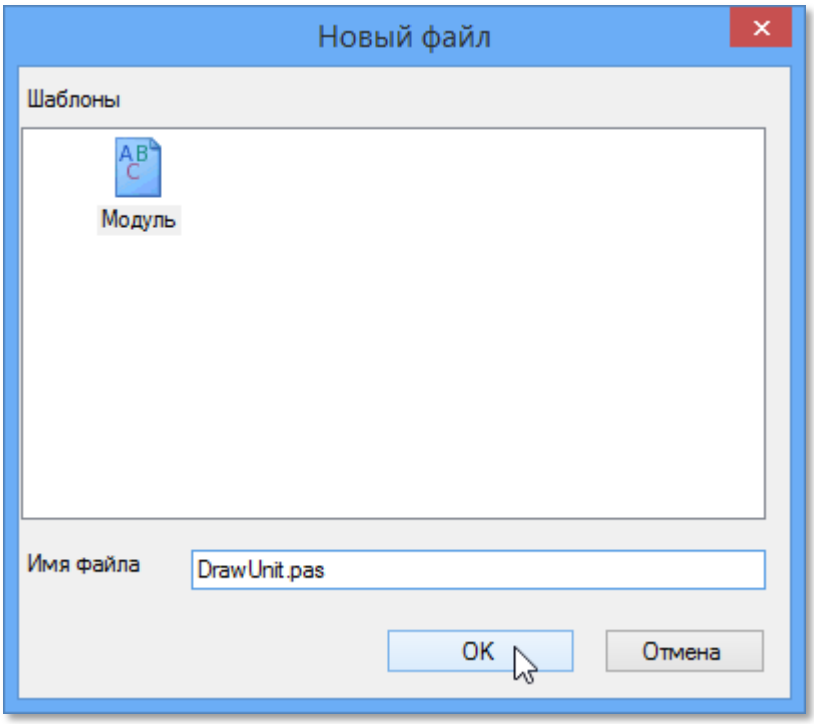


Рис. 1.17. «Выбираем» модуль

Так как наши программы *графические*, то все модули мы будем называть именно так, чтобы не ломать каждый раз голову.

Новый файл появится в окне *Структуры проекта* и откроется в окне *Редактора кода* (Рис. 1.18).

Добавить новый файл можно и командой меню **Проект > Добавить новый файл** (Рис. 1.19).

Чтобы использовать новый модуль в главном файле программы, нужно объявить его директивой **uses** (Рис. 1.20).

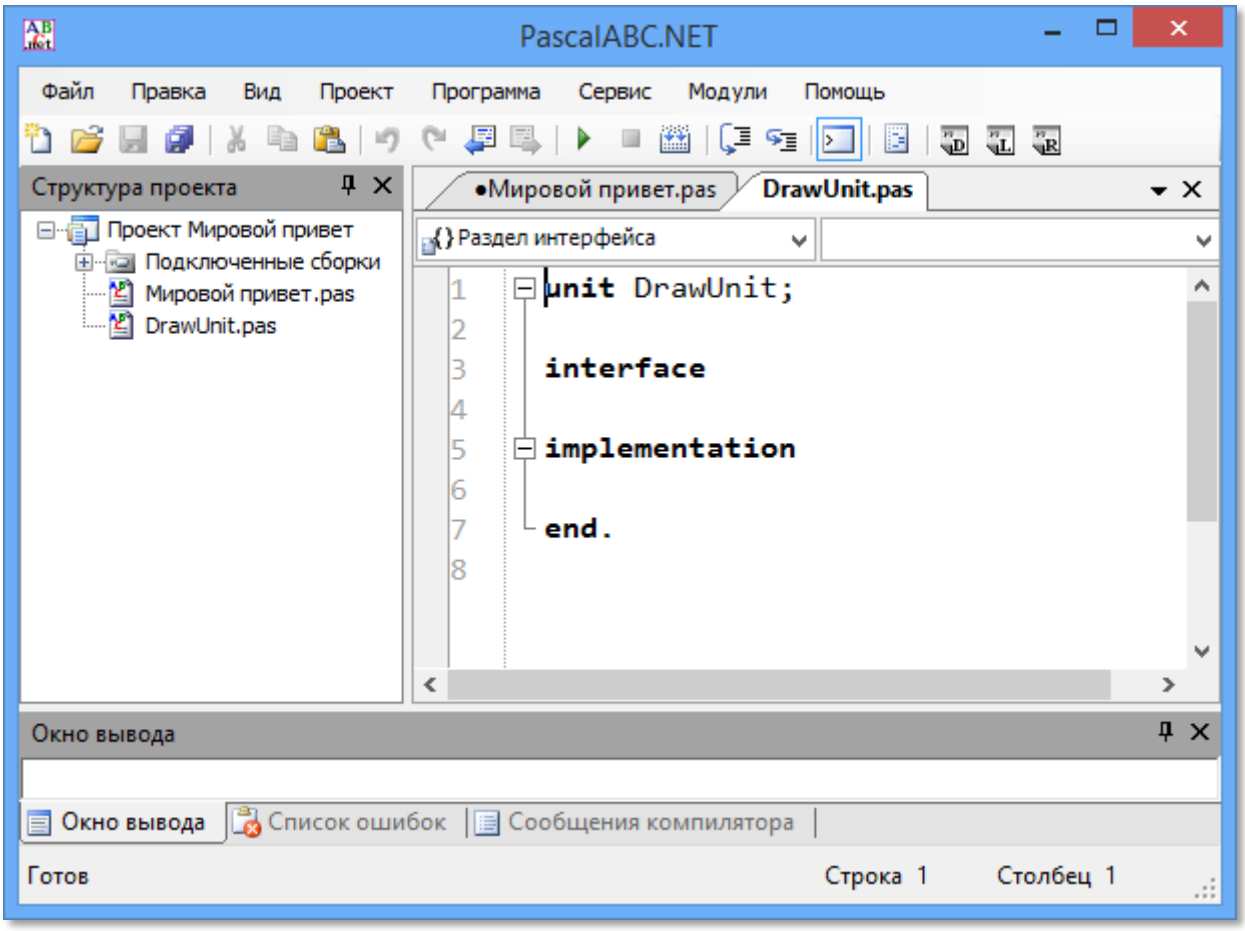


Рис. 1.18. Заготовка модуля

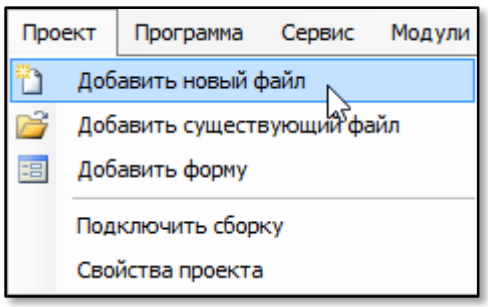


Рис. 1.19. Альтернативный способ

Вернитесь в модуль **DrawUnit** и сотрите строчки *interface* и *implementation*, а затем добавьте в разделе **uses** пространства имён *System* и *Microsoft.SmallBasic.Library*:

```
unit DrawUnit;

uses
   Microsoft.SmallBasic.Library, System;
```

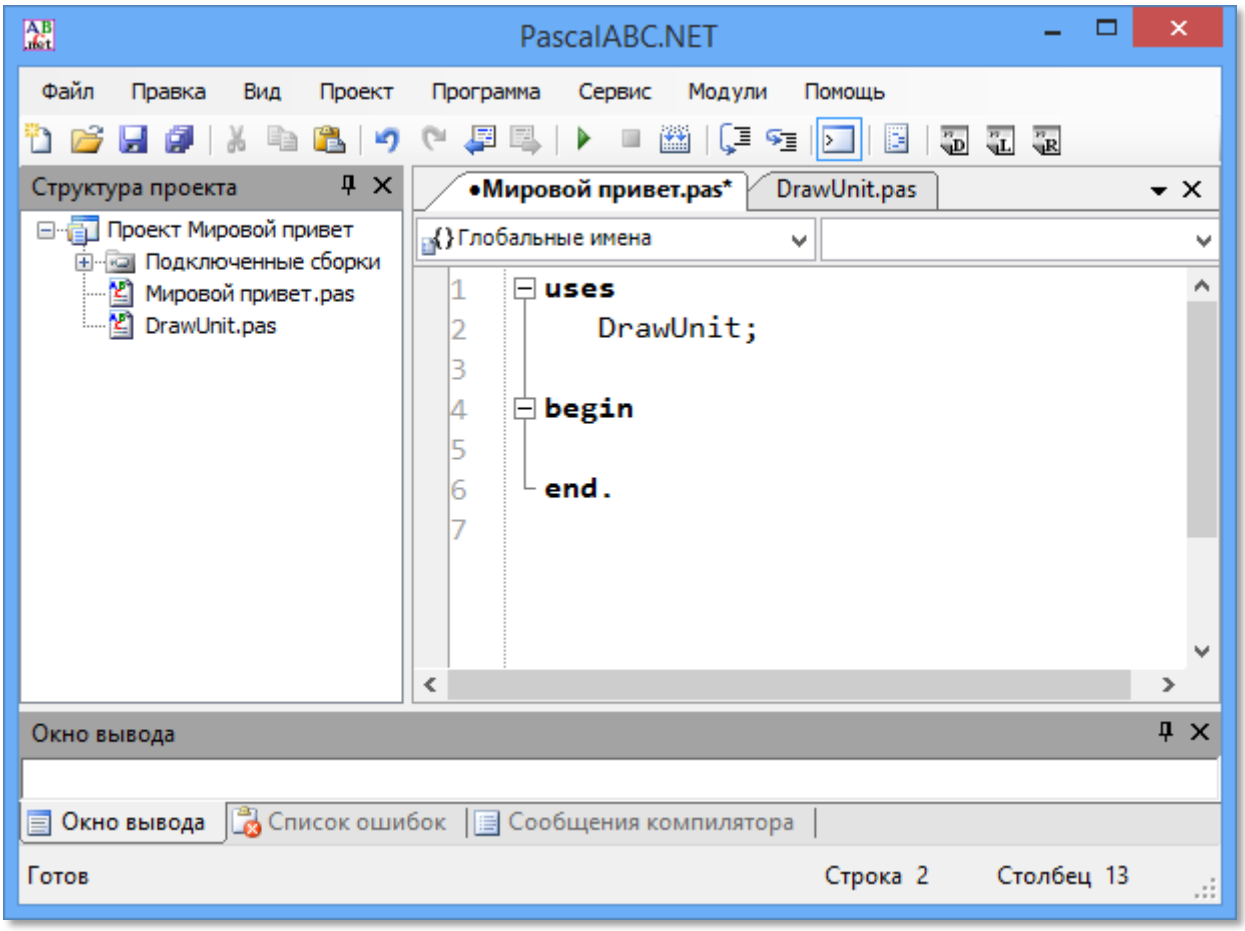


Рис. 1.20. Добавляем ссылку на модуль

Классы во всех проектах будут называться одинаково – **Draw**. В большинстве случаем нам будет достаточно двух открытых методов – *Prepare* и *Draw*:

```
type
   Draw = class

      public class procedure Prepare();
      begin

      end;

      public class procedure Draw();
      begin

      end;
   end; //end of class

end.
```

Первый метод подготовит окно приложения, а второй – начертит в нём различные графические примитивы.

В данном случае это методы **класса**, но чаще мы будем использовать их как методы объекта и тогда ключевое слово *class* после модификатора *public* необходимо стереть.

> Все классы имеют модификатор доступа **public**, поэтому доступны везде.
>
> По умолчанию все члены класса получают модификатор доступа **internal**, то есть будут видны во всех файлах проекта (сборки).
>
> В *паскале* действует такое правило: модификатор доступа действует на все члены класса, расположенные ниже по коду и не имеющие явного модификатора. Поэтому будьте внимательны, а лучше всего – каждый член класса предваряйте модификатором доступа, тогда вы наверняка избежите возможных ошибок.
>
> Всегда используйте **минимально** необходимый модификатор доступа!

Эту заготовку кода вы можете сохранить в папке с проектами и в дальнейшем использовать как **шаблон** модуля для сокращения подготовительной работы.

Нашу программу уже можно запустить, но она, естественно, ничего не сделает.

Давайте для начала создадим **окно приложения**.

Все методы для работы с окном приложения собраны в классе *GraphicsWindow*, а метод **Show** показывает окно на экране при запуске приложения.

Вызываем этот метод в методе рисования **Draw**:

```
public class procedure Draw();
begin
   GraphicsWindow.Show();
end;
```

Но, как вы помните, при запуске программы выполняется **основная программа**, поэтому вернитесь в главный файл и добавьте вызов метода *Draw* одноимённого класса:

```
uses
   DrawUnit;

// ПЕРВАЯ ГРАФИЧЕСКАЯ ПРОГРАММА

begin
   Draw.Draw();
end.
```

Для вызова методов класса создавать экземпляр класса не нужно, так что весь код укладывается в одну строку.

Запускаем программу – и вот уже *Графическое окно* перед нами во всей своей красе (Рис. 1.21)!

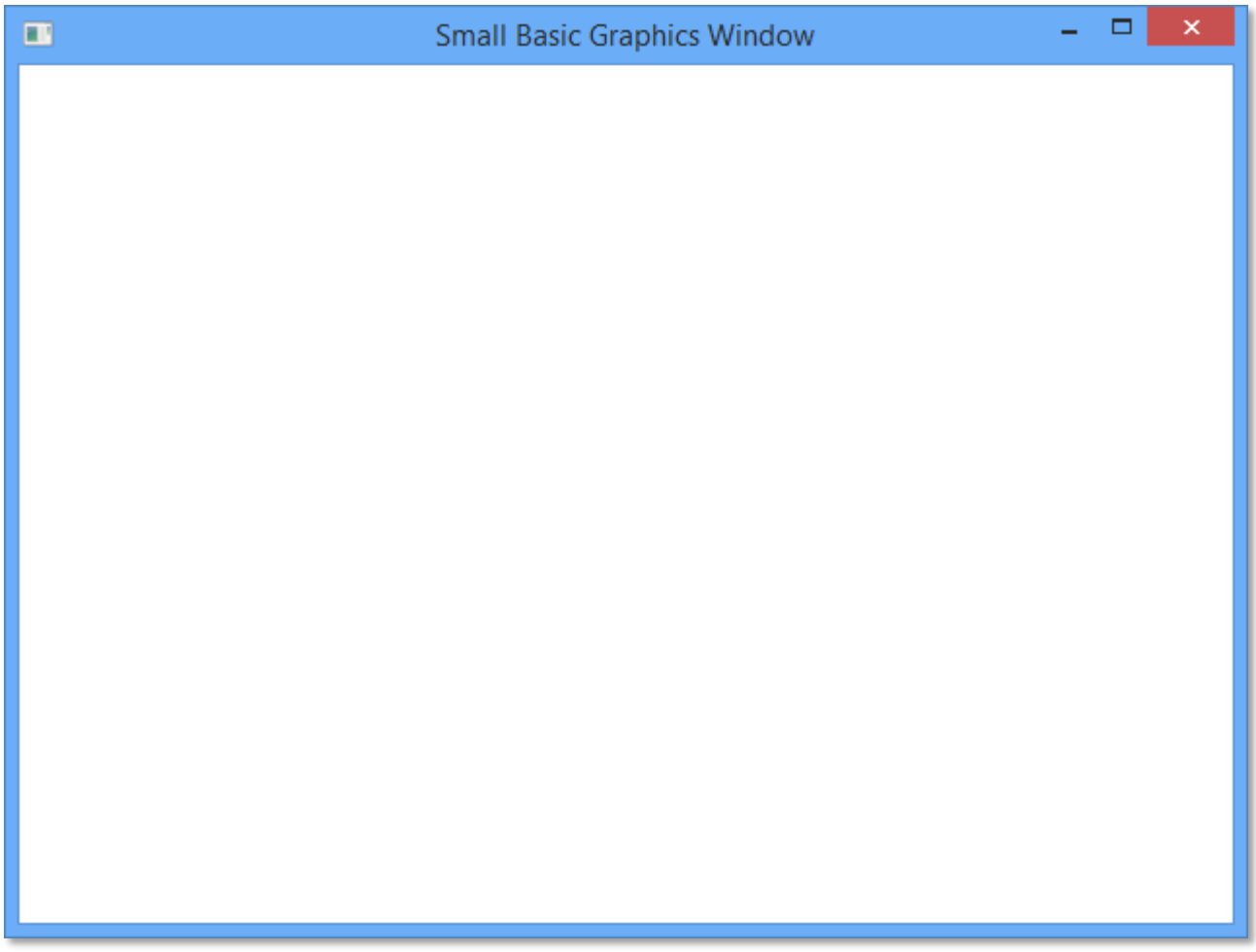


Рис. 1.21. Графическое окно создано!

Оно имеет размеры и заголовок по умолчанию, но скоро вы научитесь изменять свойства окна по своему желанию.

Закройте окно, нажав красную кнопку с крестиком в правом верхнем углу.

Хочу заметить, что если вы используете в программе любые методы класса *GraphicsWindow*, то окно будет создано автоматически, поэтому дополнительно вызывать метод *Show* не нужно.

Проект *Мировой привет*

Исходный код программы находится в папке **Мировой привет**.

Первую строчку в методе **Draw** вы можете закомментировать или стереть. Теперь добавьте две строчки кода, которые напечатают в *Графическом окне* приветствие Миру:

```
public class procedure Draw();
begin
   //GraphicsWindow.Show();
   GraphicsWindow.DrawText(10, 10, 'Hello, World!');
   GraphicsWindow.DrawText(10, 30, 'Здравствуй, Мир!');
end;
```

На Рис. 1.22 вы видите, что графические приветы удались нам на славу!

С помощью метода **DrawText** вы можете напечатать любые текстовые сообщения в любом месте *Графического окна*. Вы также можете выбрать любой шрифт, изменить его размер и цвет – но об этом мы более подробно поговорим дальше.

Запишите все файлы проекта, нажав кнопку **Сохранить все** (Рис. 1.23).

При наборе кода очень помогает команда меню **Сервис > Форматировать** код (Рис. 1.24).

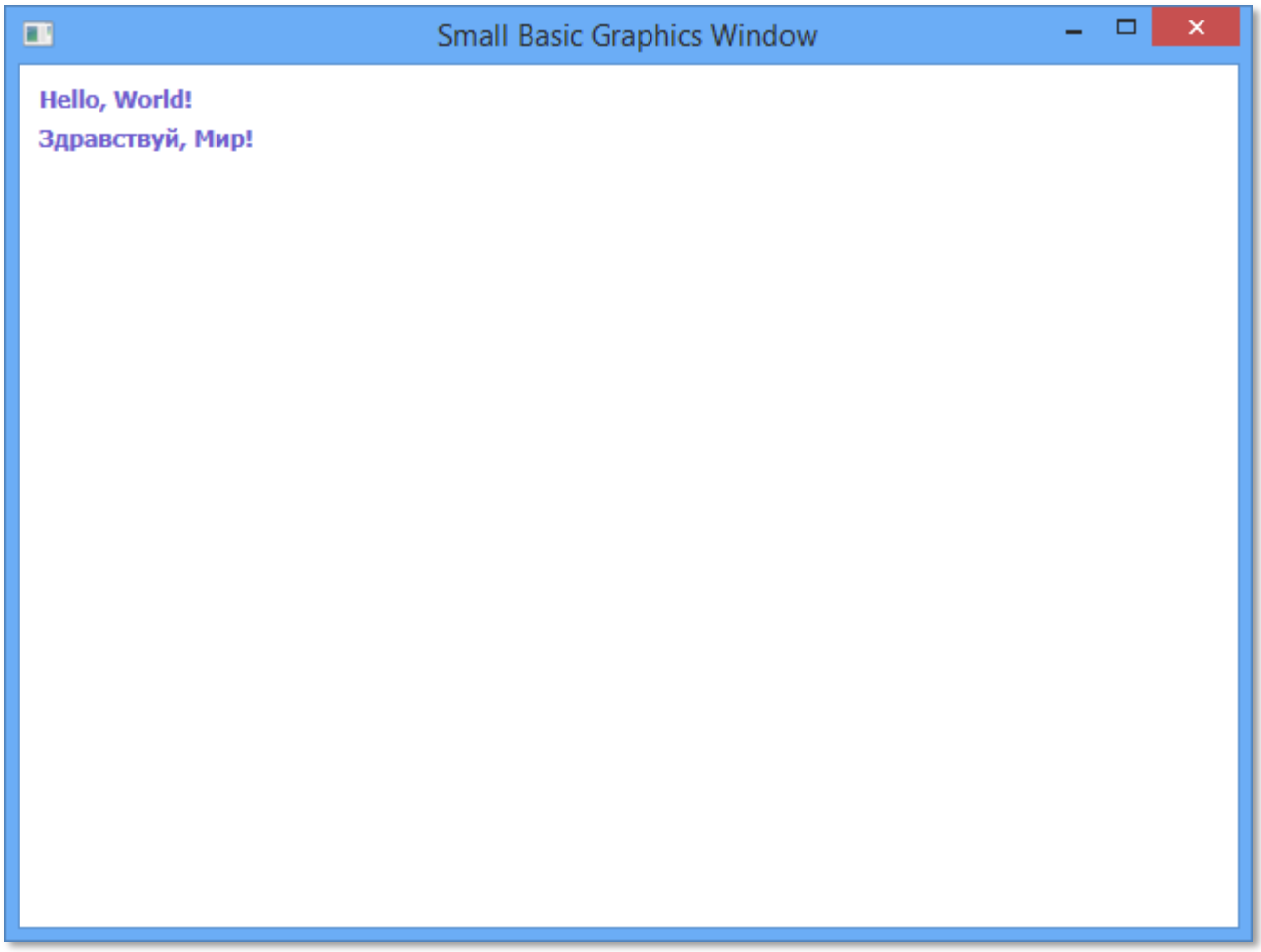


Рис. 1.22. С первым приветом!

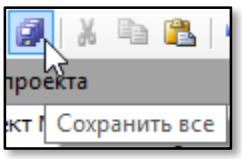


Рис. 1.23. Не забывайте сохраняться!

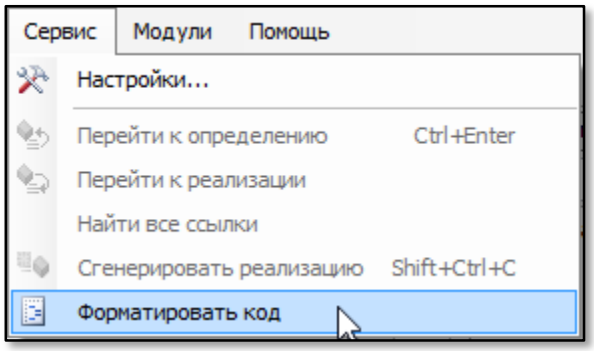


Рис. 1.24. Форматируем

Ещё удобнее пользоваться кнопкой **Форматировать код** на *Панели инструментов* (Рис. 1.25).

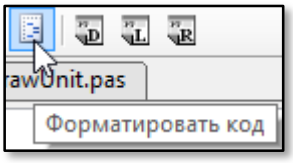


Рис. 1. 25. Кнопочный способ

Эта команда правильно расставит все отступы в исходном коде, если вы сами об этом не позаботились. Однако форматирование прекратится, как только обнаружатся синтаксические ошибки в программе. Их следует исправить, а затем повторить форматирование.

> Команда форматирования не всегда корректно обрабатывает закомментированные строки, так что имейте это в виду.
>
> Если до форматирования была выделена часть кода, то весь код будет повторён ещё раз. Поэтому снимайте выделение кода перед каждым форматированием!

До сих пор мы запускали программу кнопкой **Выполнить** на *Панели инструментов* (Рис. 1.26).

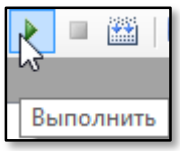


Рис. 1.26. Кнопка запуска программы

Однако запуск программы командой меню **Программа > Выполнить без связи с оболочкой** (Рис. 1.27) иногда завершается с ошибкой.

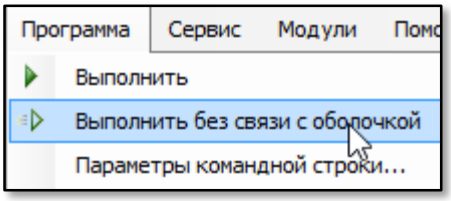


Рис. 1.27. Автономная программа

Поэтому добавляйте в начало главного файла программы директиву компилятора:

```
{$apptype windows}

uses
   DrawUnit;

// ПЕРВАЯ ГРАФИЧЕСКАЯ ПРОГРАММА

begin
   Draw.Draw();
end.
```

Проект *Текстовое окно*

Исходный код программы находится в папке **Текстовое окно**.

В библиотеке *SmallBasicLibrary* имеется и **Текстовое окно** (Рис. 1.28), которое вы можете использовать в своих программах.

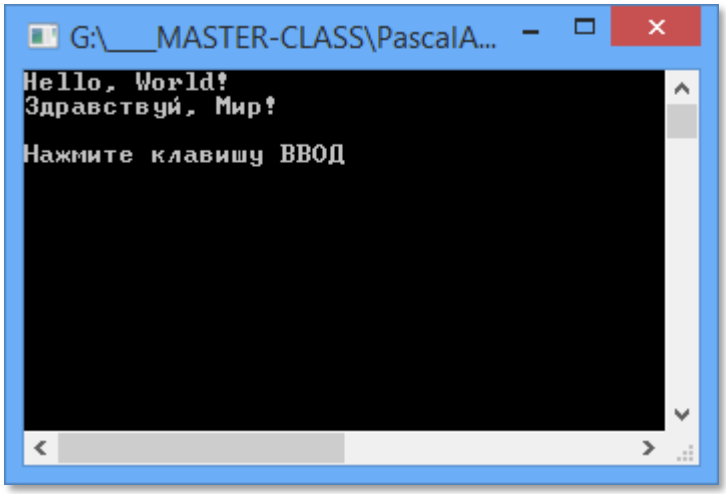


Рис. 1.28 *Текстовое окно*

Начните новый проект **Текстовое окно** и сохраните в папке с программами.

Откройте папку и скопируйте в неё библиотеку **SmallBasicLibrary.dll** и модуль **DrawUnit.pas** из предыдущего проекта.

В главный файл программы также копируйте код из предыдущей программы:

```
{$apptype windows}

uses
   DrawUnit;

begin
   Draw.Draw();
end.
```

Теперь добавьте ссылку на библиотеку, как это было описано выше (Рис. 1.29).

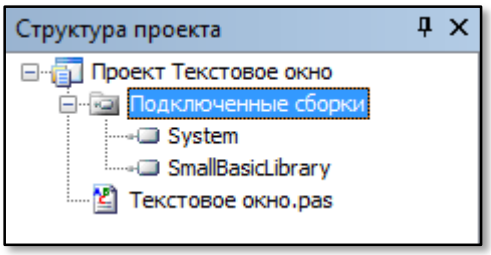


Рис. 1.29. Не забывайте про библиотеку!

Поскольку мы используем уже *готовый* модуль, то его нужно добавить к проекту командой контекстного меню **Добавить > Существующий файл** (Рис. 1.30) или командой меню **Проект > Добавить существующий файл** (Рис. 1.31).

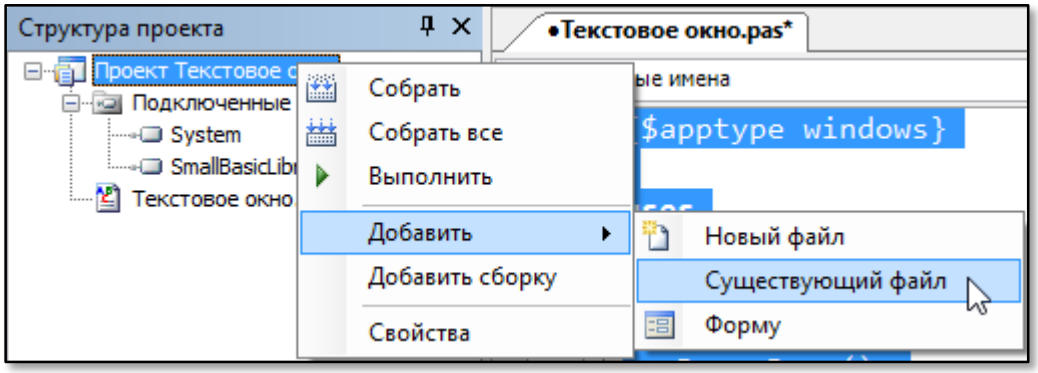


Рис. 1.30. Добавляем готовый модуль

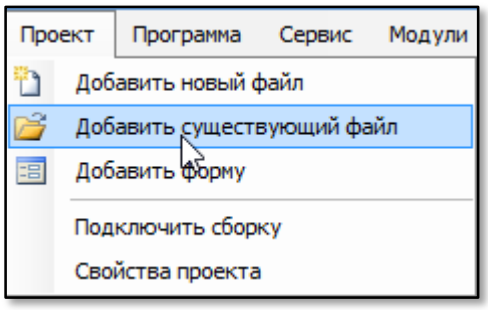


Рис. 1.31. Аналогично

В открывшемся диалоговом окне выделите файл **DrawUnit** и нажмите кнопку **Открыть** (Рис. 1.32) или дважды щёлкните мышкой по названию файла.

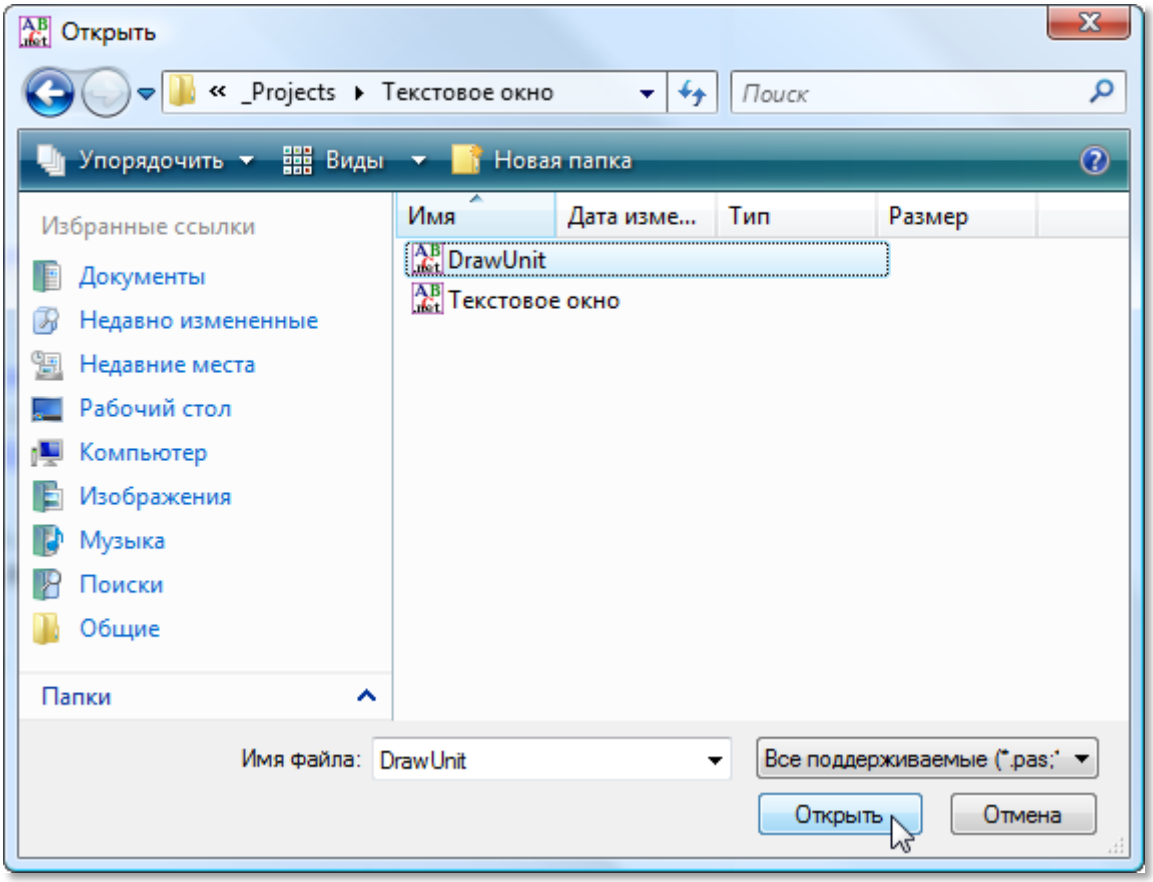


Рис. 1.32. Добавляем готовый модуль

Он появится в окне *Структуры проекта* (Рис. 1.33).

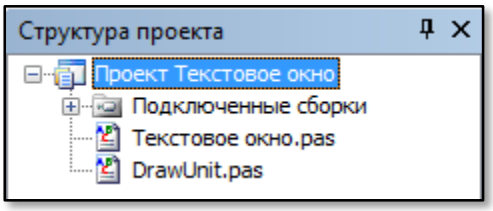


Рис. 1.33. Модуль добавлен

Чтобы его открыть в окне *Редактора кода*, дважды щёлкните по названию или выполните команду контекстного меню **Показать код** (Рис. 1.34).

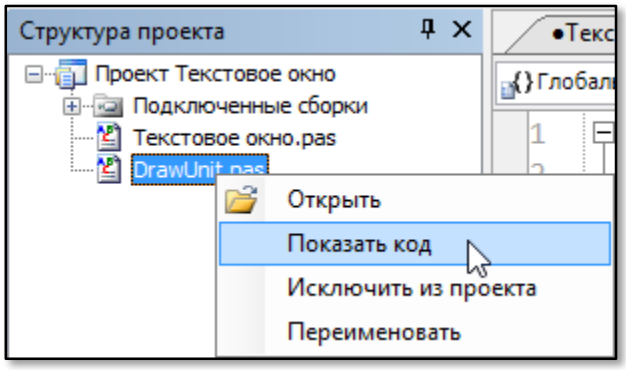


Рис. 1.34. Открываем модуль

Добавьте в метод **Draw** строчки для печати сообщений в *Текстовом окне*:

```
public class procedure Draw();
begin
   GraphicsWindow.DrawText(10, 10, 'Hello, World!');
   GraphicsWindow.DrawText(10, 30, 'Здравствуй, Мир!');
   TextWindow.WriteLine('Hello, World!');
   TextWindow.WriteLine('Здравствуй, Мир!');
   TextWindow.WriteLine('');
   TextWindow.Write('Нажмите клавишу ВВОД');
   TextWindow.Read();
end;
```

При запуске приложения кнопкой **Выполнить** на *Панели инструментов* будет создано *Графическое окно*, а сообщения для **Текстового окна** будут напечатаны в *Окне вывода* (Рис. 1.35).

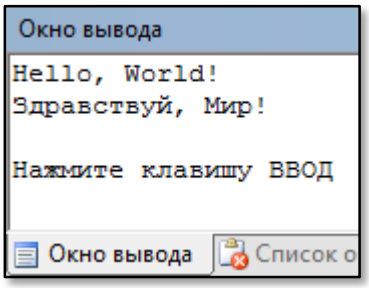


Рис. 1.35. Печатаем в *Окне вывода*

При запуске программы в режиме **Выполнить без связи с оболочкой** будут созданы 2 окна – *Графическое* и *Консольное* (Рис. 1.36).

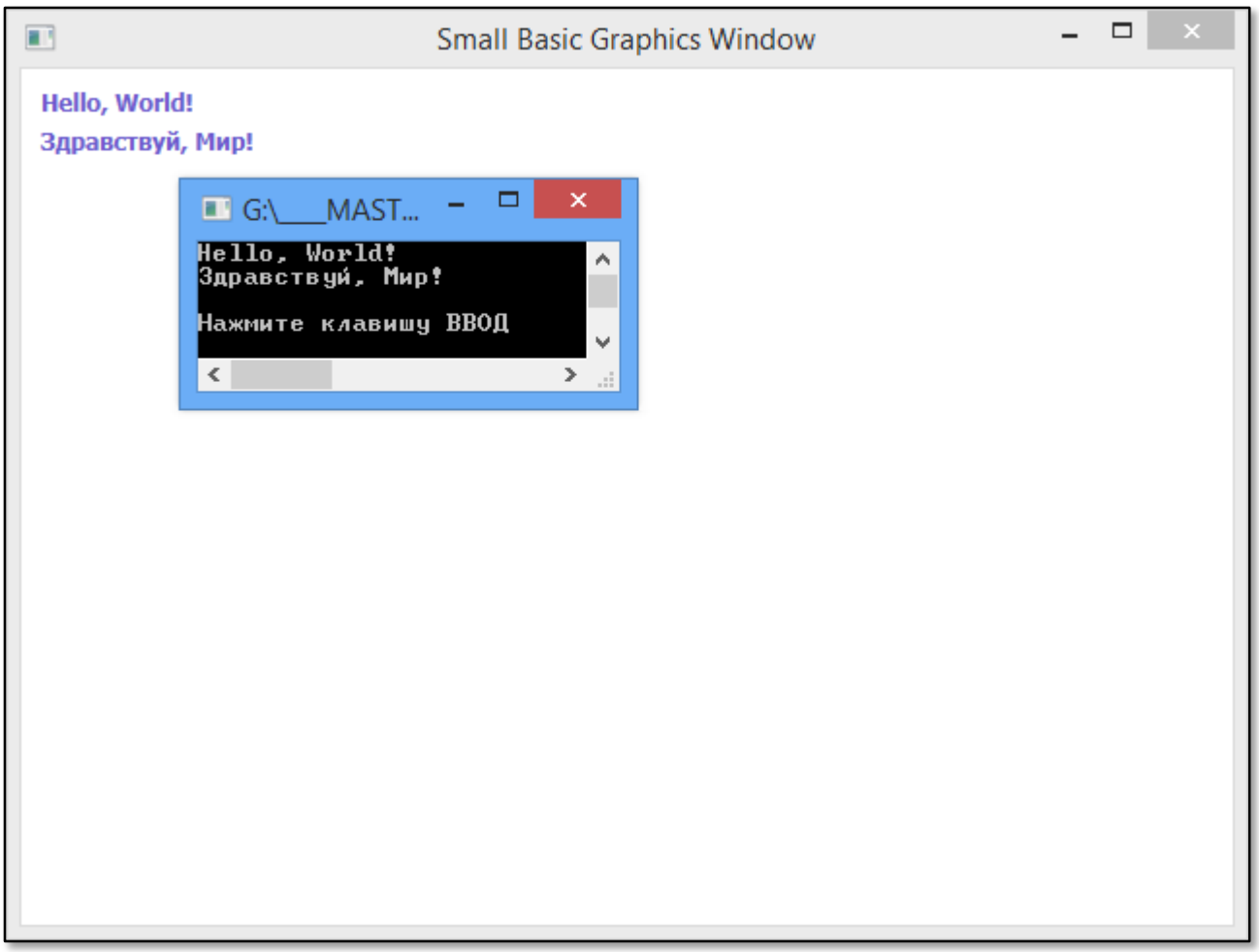


Рис. 1.36. Два приветливых окна

Если вы закомментируете строчки для *Графического окна*, то программу нужно запускать во втором режиме. Тогда будет создано только *Консольное окно* (Рис. 1.37).

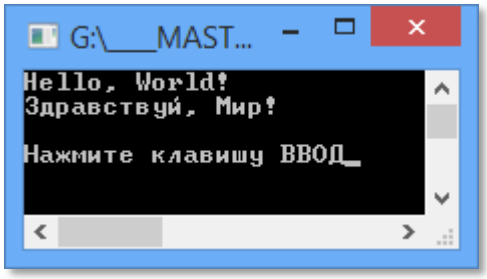


Рис. 1.37. Вывод сообщений в *Консольное окно*

Естественно, мы не будем использовать *Консольное окно* в дальнейших проектах, хотя иногда оно может пригодиться для вывода отладочной информации.

По умолчанию **выполняемый файл** проекта стирается, поэтому вы не сможете запустить скомпилированную программу с диска. При отладке приложения выполняемый файл и не нужен, но для готовой программы совершенно необходим.

Чтобы сохранить выполняемый файл и при **отладке**, выполните команду контекстного меню проекта **Свойства** или меню **Проект > Свойства проекта** (Рис. 1.38).

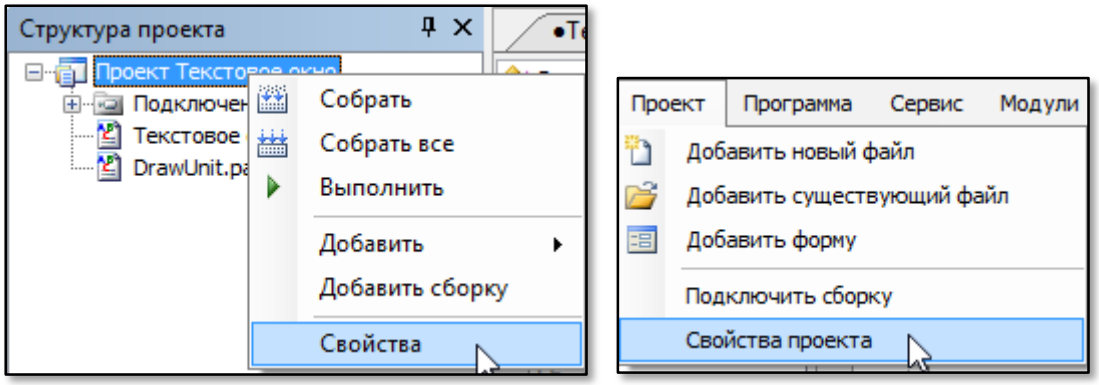


Рис. 1.38. Свойства проекта

В диалоговом окне *Свойства проекта* снимите галочку **Удалять EXE-файл после выполнения** (Рис. 1.39).

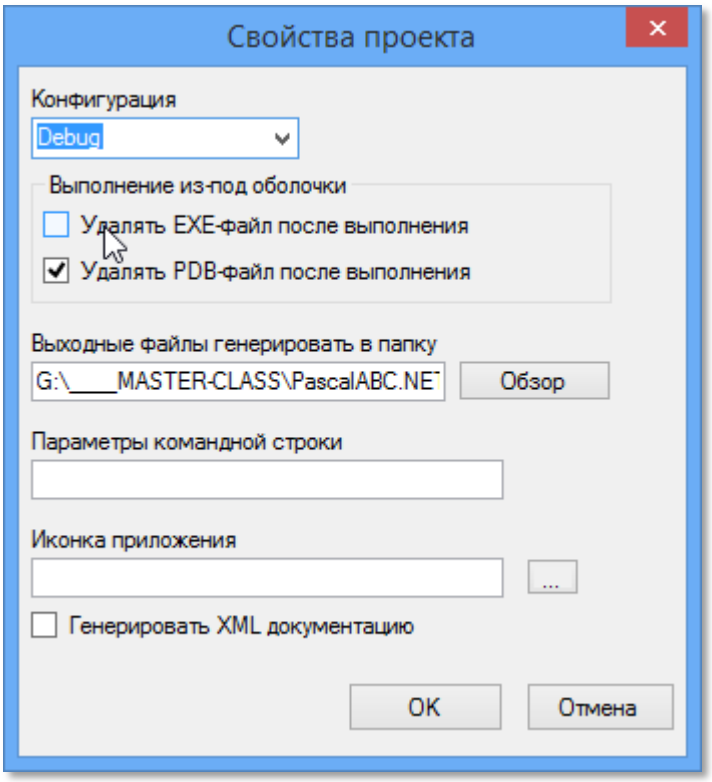


Рис. 1.39. Убираем галочку

Теперь выполняемые файлы будут сохраняться на диске.

Флажок **Удалять PDB-файл после выполнения** можно оставить, так как этот файл хранит только отладочную информацию.

После полной и окончательной отладки приложения включите конфигурацию **Release** (Рис. 1.40).

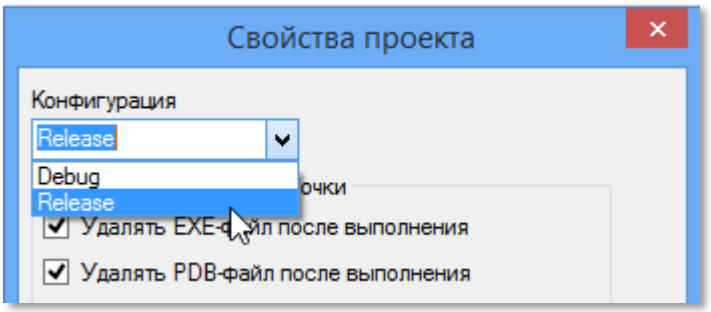


Рис. 1.40. Режим полной готовности

В этом режиме выполняемый файл всегда остаётся на диске.

Чтобы создать выполняемый файл без запуска приложения, выполните команду меню **Программа > Компилировать** или **Перекомпилировать все** (Рис. 1.41).

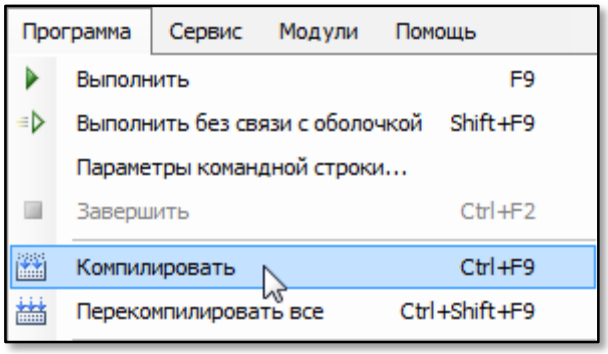


Рис. 1.41. Компиляция без запуска

Перекомпилировать программу можно и **кнопкой** на *Панели инструментов* (Рис. 1.42).

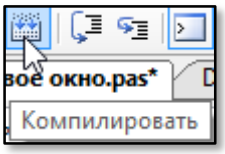


Рис. 1.42. Компилирующая кнопка

Сохраните проект на диске и закройте среду разработки.

Глава 2. Класс *GraphicsWindow* (Графическое окно)

С появлением операционной системы *Windows* основным интерфейсом пользователя стал *графический*, а текстовый вывод на консоль применяется только для быстрого получения результатов вычислений.

Вся информация – и текстовая, и графическая - при этой разновидности интерфейсов выводится в *Графическое окно*, которое представлено в библиотеке *SmallBasicLibrary* классом **GraphicsWindow**.

Всякая программа с графическим интерфейсом начинается с создания и настройки параметров окна приложения (Рис. 2.1). *Графическое окно* создаётся **автоматически** - достаточно хотя бы раз упомянуть его в программе. Например, так:

```
{$apptype windows}

// Класс GraphicsWindow (Графическое окно)

uses
   DrawUnit;


begin
   Draw.Draw();
end.

 public class procedure Draw();
 begin
    GraphicsWindow.Show();
    Thread.Sleep(TimeSpan.FromMilliseconds(5000));
    GraphicsWindow.Hide();
    Thread.Sleep(TimeSpan.FromMilliseconds(1000));
    GraphicsWindow.Show();
 end;
```

В библиотеке *SmallBasicLibrary* имеется удобный метод *Program.Delay* для создания пауз в работе программы, но компилятор *паскаля* использует

идентификатор *Program* для класса программы и делает указанный метод недоступным.

Можно добавить к идентификатору класса *Program* пространство имён *Microsoft.SmallBasic.Library*, в котором он находится, но компилятор не находит это пространство имён, хотя и подставляет при компиляции программы.

Поэтому метод *Program.Delay* нужно заменить более длинным, но более универсальным:

```
Thread.Sleep(TimeSpan.FromMilliseconds(5000));
```

Также нужно добавлять пространство имён для класса **Thread**:

```
System.Threading;
```

Метод **Show** **показывает** *Графическое окно* при запуске программы, а метод **Hide** делает его **невидимым** (но приложение продолжает работать).

Как видите, *Графическое окно* – это обычное окно *Windows*; оно имеет 4 стандартные кнопки (Рис. 2.2).

С их помощью можно проделывать обычные операции: сворачивать окно, разворачивать его, перемещать и, наконец, закрыть.

Окно имеет **заголовок** *Small Basic Graphics Window*, который легко заменить названием программы, присвоив свойству окна **Title** нужное значение (Рис. 2.3):

```
public class procedure Draw();
begin
   GraphicsWindow.Title := 'Моя программа';
end;
```

Если вы забудете, что написано, в заголовке окна, то легко освежите память, считав значение этого свойства в строковую переменную (Рис. 2.4):

```
var s := GraphicsWindow.Title.ToString();
Console.WriteLine(s);
```

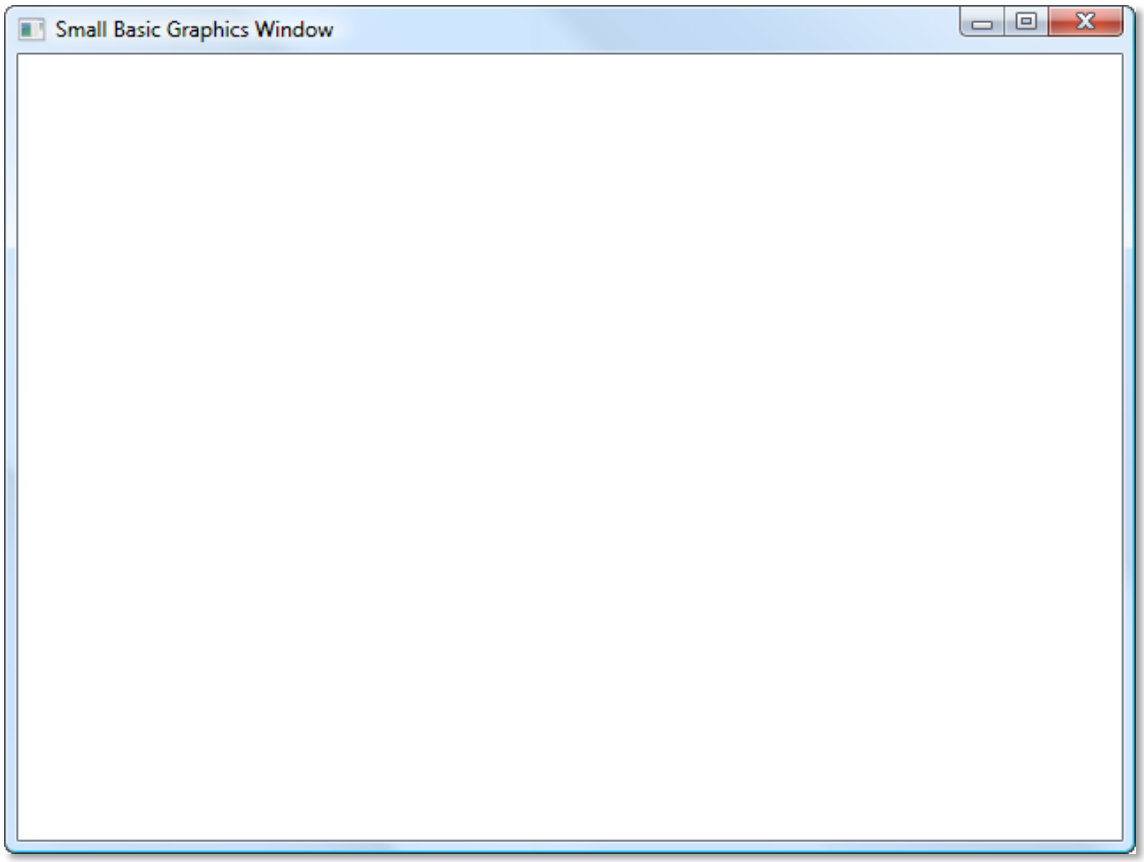


Рис. 2.1. *Графическое окно* создано!

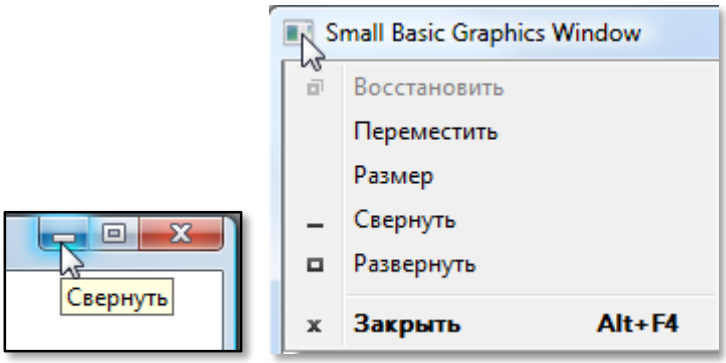


Рис. 2.2. Кнопки стандартного окна

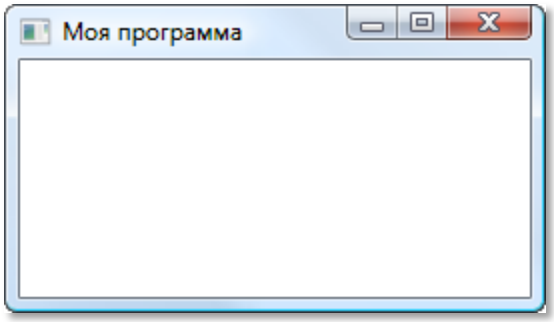


Рис. 2.3. Так информативнее!

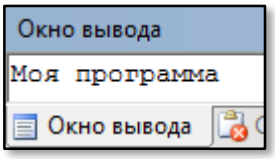


Рис. 2.4. Не напрягайте память!

Заголовок окна можно использовать и *нестандартно*, например, выводить в него число набранных в игре очков, время и другую полезную информацию (Рис. 2.5):

```
begin
   var draw := new Draw();
   draw.Prepare();
   draw.Draw();
end.
```

```
unit DrawUnit;

uses
   Microsoft.SmallBasic.Library, System, System.Threading;

type
   Draw = class

      // время в секундах:
      private time := 0;

      //ТАЙМЕР
      private procedure OnTimer();
      begin
         time += 1;
         GraphicsWindow.Title := time;
      end;

      public procedure Prepare();
      begin
         time := 0;
         Timer.Interval := 1000;
         Timer.Tick += OnTimer;
      end;

      public procedure Draw();
      begin
         GraphicsWindow.Title := 'Моя программа';
         var s := GraphicsWindow.Title.ToString();
         Console.WriteLine(s);
      end;
   end; //end of class
```

```
end.
```

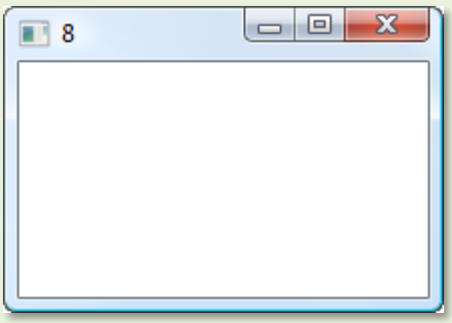


Рис. 2.5. Таймерное окно

Здесь нам пришлось отказаться от методов класса, потому что процедура **OnTimer** должна принадлежать **объекту**. Все остальные методы тоже не могут быть классовыми (или в этом уже нет необходимости).

По умолчанию пользователю разрешается не только перетаскивать окно по экрану, но также **изменять его размеры**, потянув за края или углы. Иногда это приводит к тому, что графические элементы управления перемещаются, и внешний вид программы становится неряшливым. Поэтому вы можете пресечь подобное «вольнодумство» пользователя и запретить ему изменять размеры окна, присвоив свойству **CanResize** значение **false**:

```
GraphicsWindow.CanResize := false;
```

Тогда у окна останется справа только одна кнопка - *Закрыть*, а в системном меню исчезнут некоторые команды (Рис. 2.6).

Впрочем, вы можете сменить гнев на милость и снова вернуть пользователю свободу действий:

```
GraphicsWindow.CanResize := true;
```

Свойство *CanResize* по умолчанию имеет значение *true*, поэтому, если вы не защитите программу, то пользователь безнаказанно сможет изменять размеры её окна.

Если вы не позаботитесь сами, то окно будет иметь стандартные **размеры** 640 на 480 пикселей и при запуске приложения **появится** у левого верхнего угла экрана.

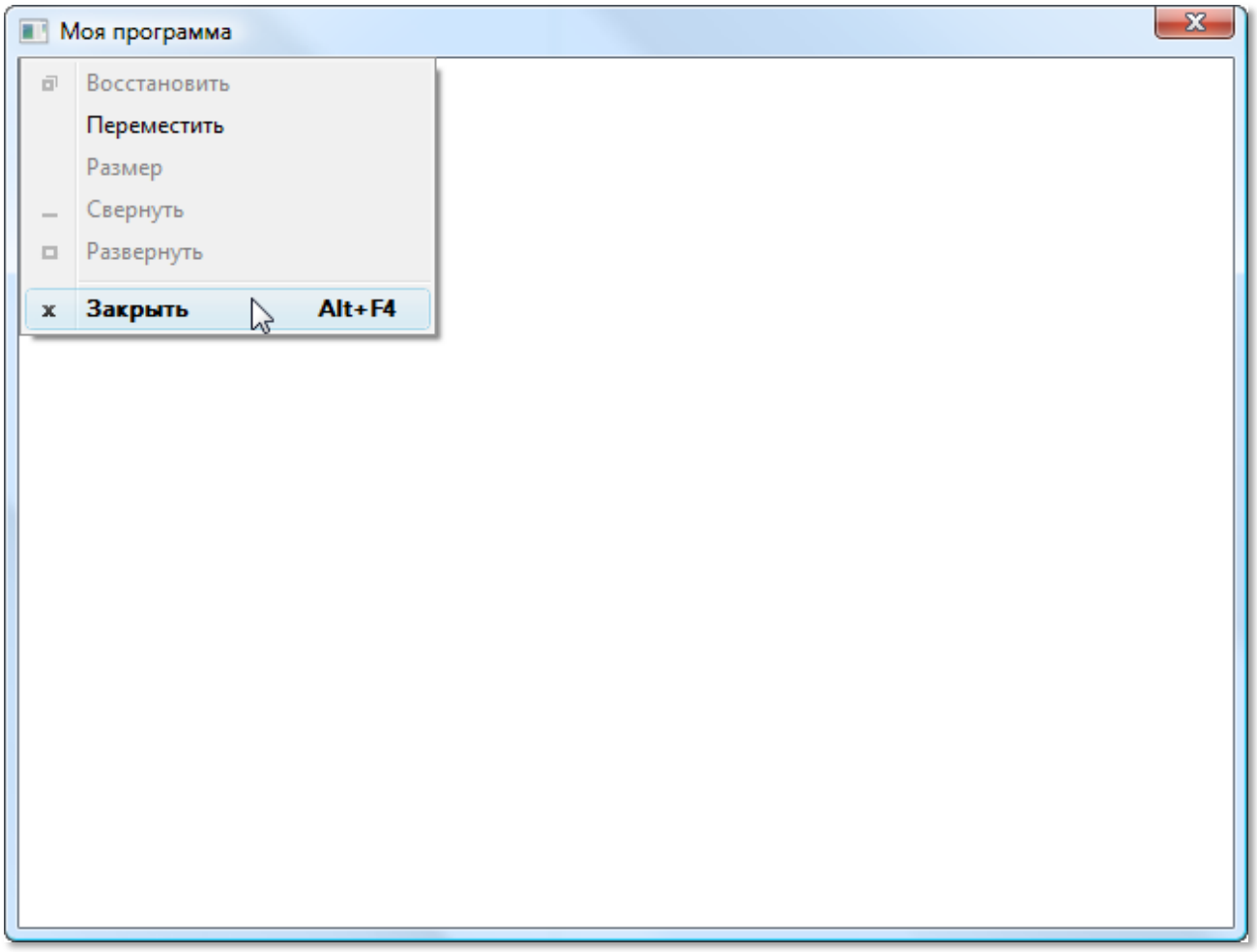


Рис. 2.6. Приложение строгих форм!

За **размеры окна** отвечают свойства с понятными названиями **Width** и **Height**. Присвоив им нужные значения:

```
GraphicsWindow.Width := 400;
GraphicsWindow.Height := 200;
```

вы получите окно размером поменьше (Рис. 2.7).

Можно сделать и «форточку» (Рис. 2.8):

```
GraphicsWindow.Width := 0;
GraphicsWindow.Height := 0;
```

Однако даже нулевые значения этих свойств не уменьшают окно до размеров точки!

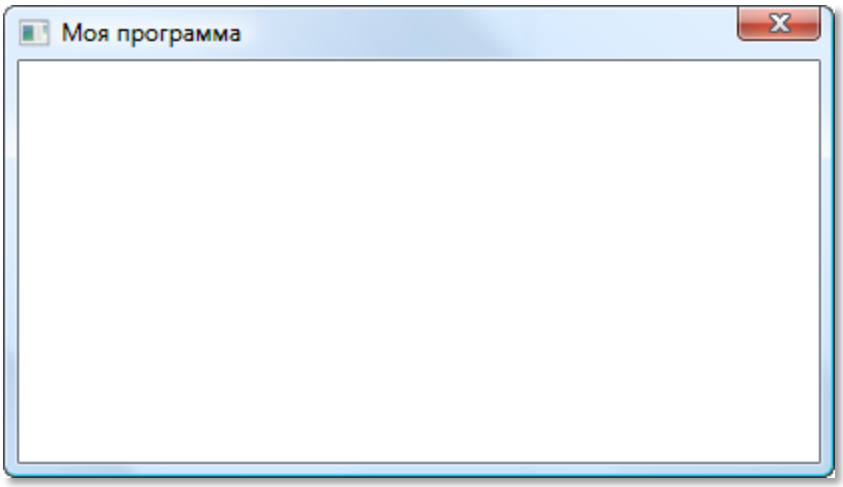


Рис. 2.7. Из окна мы сделали окошечко!

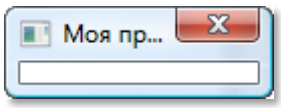


Рис. 2.8. Из окошечка мы сделали форточку!

Так же легко вы можете **установить окно** в любом месте экрана (Рис. 2.9), задав нужные координаты его левого верхнего угла **Left** и **Top**:

```
GraphicsWindow.Left := 400;
GraphicsWindow.Top := 400;
```

Как установить окно приложения в центре *Рабочего стола*, читайте дальше.

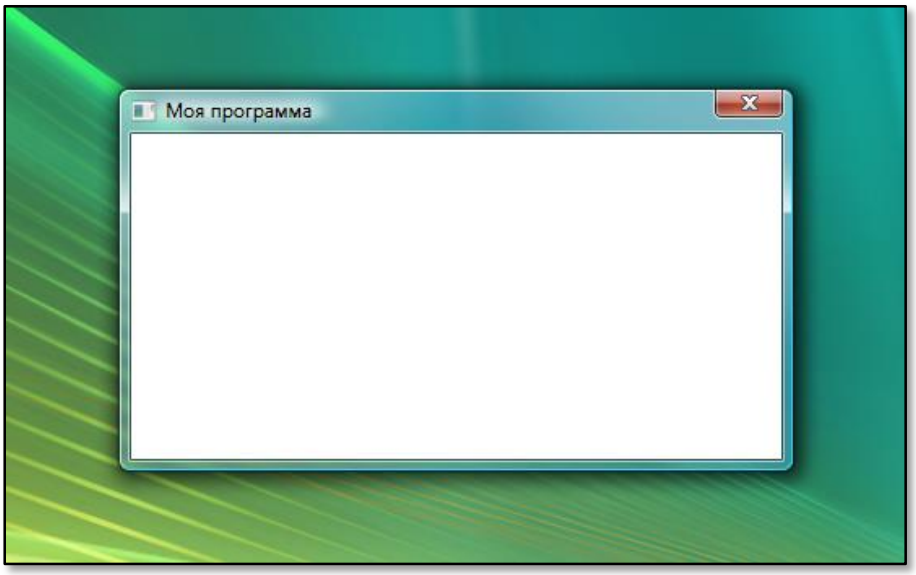


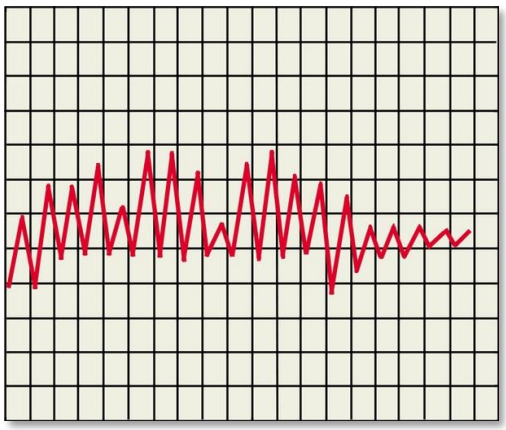
Рис. 2.9. В самое яблочко!

Элементы стандартного окна

- В верхней части окна находится полоска, которую обычно называют **заголовком**. Заголовок нужен для перемещения окна мышкой, а также на нём расположены:
  - **Значок формы** – это стандартный значок программы по умолчанию  .
  - **Заголовок формы** – там выводится значение свойства *Title Графического окна*.
  - **Кнопка сворачивания окна** (минимизации)  – окно превращается в кнопку 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 и работа приложения завершается.
- **Системное меню окна** – появляется, когда вы нажимаете правую кнопку мыши на заголовке окна. Оно дублирует уже рассмотренные нами команды управления окном.
- **Граница окна** – 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.
- **Клиентская область** – всё остальное пространство окна.

Проект «Форменная» лихорадка

Исходный код программы находится в папке **«Форменная» лихорадка**.



Давайте воспользуемся свойствами окна **Width** и **Height** и заставим его беспрерывно «ёрзать» по экрану. Сделать это проще простого. Для стимулирования окна нам потребуется **Таймер** (*Timer*). Задайте его свойству *Interval* значение 200-400 миллисекунд и напишите метод для обработки «тиканья» таймера:

```
{$apptype windows}

//«Форменная» лихорадка

uses
   DrawUnit;


begin
   var draw := new Draw();
   draw.Prepare();

end.

unit DrawUnit;

uses
   Microsoft.SmallBasic.Library, System, System.Threading;

type
   Draw = class

      // переключатель:
      private rele := true;

      public procedure Prepare();
      begin
         GraphicsWindow.Title := '«Форменная» лихорадка';
         Timer.Interval := 200;
         Timer.Tick += OnTimer;
```

```
            GraphicsWindow.Width := 400;
            GraphicsWindow.Height := 200;
        end;

    end; //end of class

end.
```

При срабатывании таймера, то есть через каждые 0,2-0,4 секунды будет вызываться метод **OnTimer**:

```
// ТАЙМЕР
private procedure OnTimer();
begin
    if (rele) then
    begin
        GraphicsWindow.Left += (Primitive)(5);
        GraphicsWindow.Top += (Primitive)(5);
    end
    else
    begin
        GraphicsWindow.Left -= (Primitive)(5);
        GraphicsWindow.Top -= (Primitive)(5);
    end;
    // переключаем направление перемещения формы:
    rele := not rele;
 end;
```

В зависимости от значения переключателя **rele** окно немного сдвигается вниз-вправо, а потом возвращается на место. На статичном рисунке этого не покажешь, а в жизни бегающее окно выглядит очень забавно (Рис. 2.10).

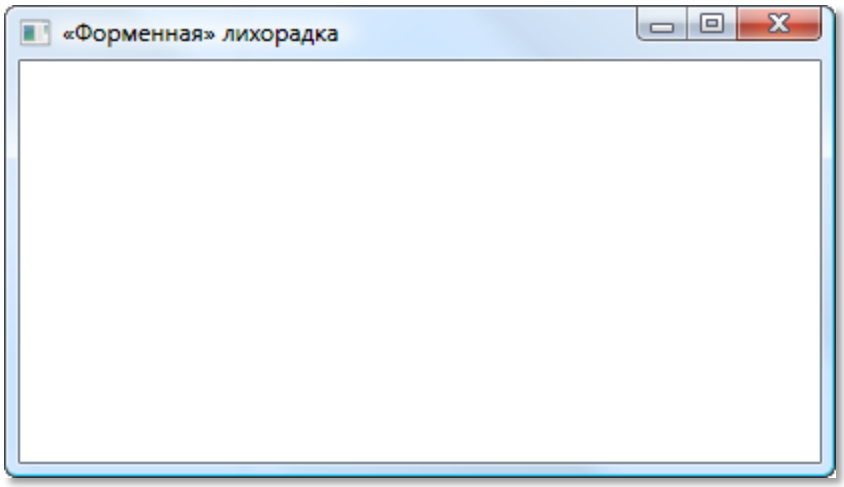
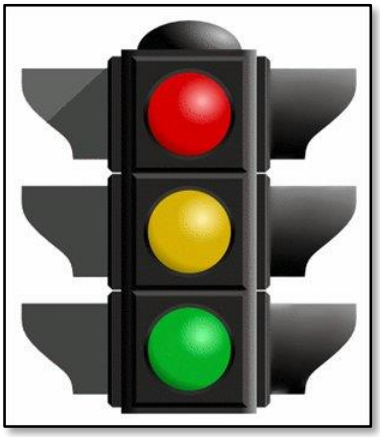


Рис. 2.10. Моментальный снимок лихорадочного окна

Проект *Светоформа*

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



Вы должны были обратить внимание, что клиентская область окна окрашена в *белый* цвет, но некоторые любят и другие цвета. Что ж, очень просто удовлетворить желания самых изощрённых любителей прекрасного. Для этого свойству **BackgroundColor** следует задать нужный цвет (Рис. 2.11):

```
GraphicsWindow.BackgroundColor := 'Green';
```

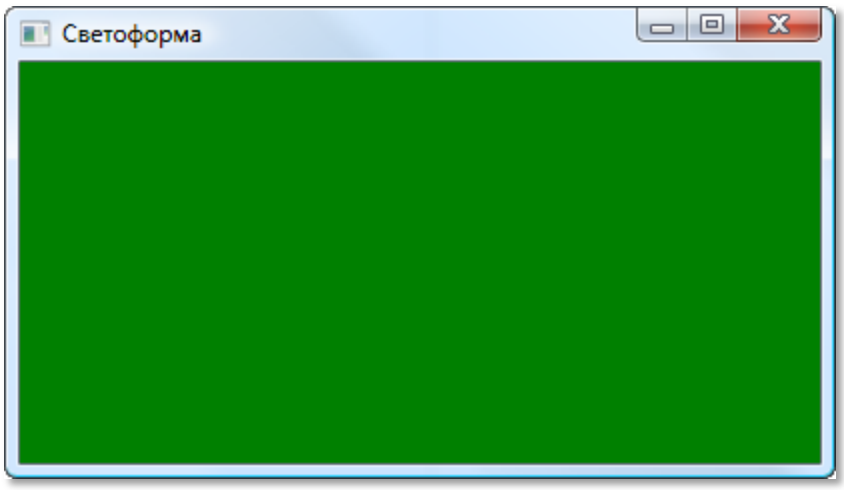


Рис. 2.11. Окно **позеленело**!

А теперь давайте превратим окно в **светофор** - чтобы оно переливалось всеми цветами радуги. Достаточно немного изменить метод из предыдущей программы, обрабатывающий «тиканье» таймера:

```
// ТАЙМЕР
private procedure OnTimer();
begin
   GraphicsWindow.BackgroundColor :=
     GraphicsWindow.GetRandomColor();
   if (rele) then
      GraphicsWindow.Show()
   else
```

```
      GraphicsWindow.Hide();

    // переключаем реле:
   rele := not rele;
end;
```

После старта программы окно будет то исчезать, то вновь появляться на экране – и каждый раз в другой «одёжке».

Обратите внимание на новый метод *Графического окна* **GetRandomColor**, который возвращает «случайный» цвет. Если вам безразлична последовательность смены цветов, то это самый быстрый и простой способ получить произвольный цвет (Рис. 2.12).

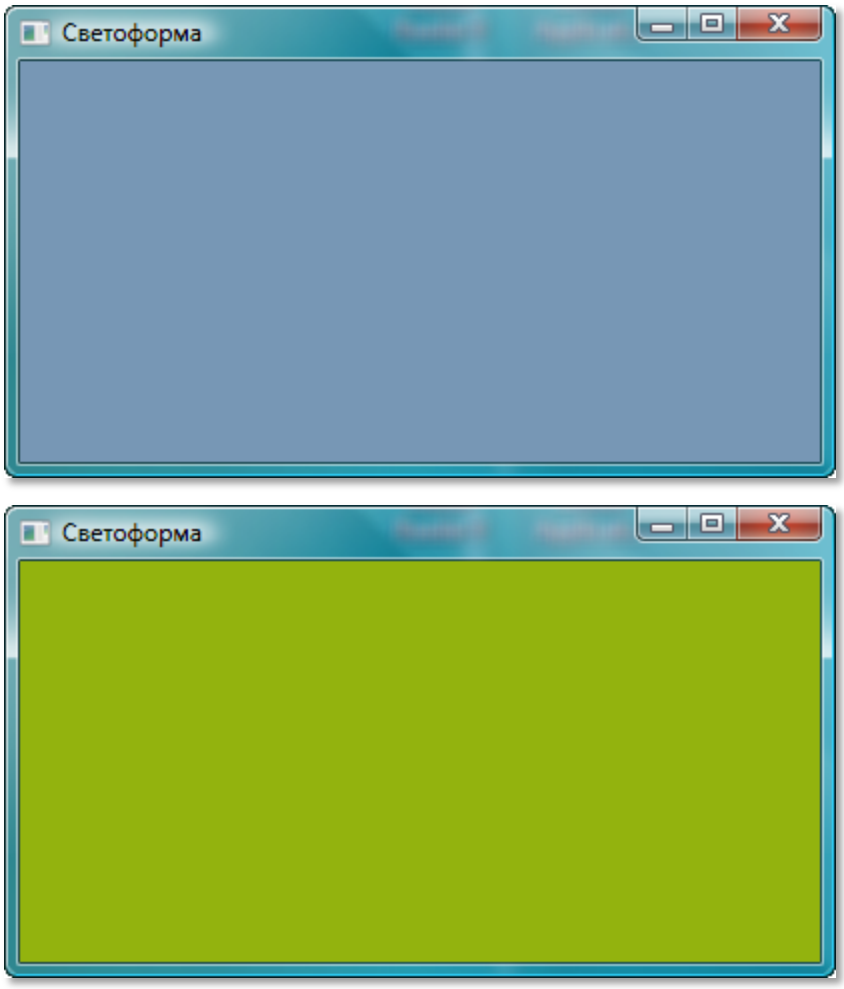


Рис. 2.12. Разноцветные окна

Проект *Светоформа 2*

Исходный код программы находится в папке **Светоформа 2**.

Ещё немного изменим программу, и окно будет не только перекрашиваться, но и **изменять свои размеры**:

```
{$apptype windows}

// Светоформа 2

uses
   DrawUnit;


begin
   var draw := new Draw();
   draw.Prepare();
end.

unit DrawUnit;

uses
   Microsoft.SmallBasic.Library, System;

type
   Draw = class

      // переключатель:
      private rele := 1;
      private n := 1;

      // ТАЙМЕР
      private procedure OnTimer();
      begin

         // изменяем цвет и размеры окна:
         if (n = 100) then
         begin
            rele *= -1;
            n := 0;
            GraphicsWindow.BackgroundColor :=
                    GraphicsWindow.GetRandomColor();
         end
         else
            n += 1;
```

```
        GraphicsWindow.Width += (Primitive)(rele);
        GraphicsWindow.Height += (Primitive)(rele);
        GraphicsWindow.Left := (Desktop.Width - GraphicsWindow.Width) / 2;
        GraphicsWindow.Top := (Desktop.Height - GraphicsWindow.Height) / 2;
      end;

      public procedure Prepare();
      begin
        GraphicsWindow.Hide();
        GraphicsWindow.Title := 'Светоформа';
        GraphicsWindow.Width := 400;
        GraphicsWindow.Height := 200;
        GraphicsWindow.BackgroundColor := 'Green';
        GraphicsWindow.Show();

        Timer.Interval := 10;
        Timer.Tick += OnTimer;
      end;

   end; //end of class
end.
```

Проект *Светоформа 3*

Исходный код программы находится в папке **Светоформа 3**.

Для задания цвета окна можно использовать и метод **GetColorFromRGB**, который преобразует составляющие цвета (красную - **Red**, зелёную - **Green** и синюю - **Blue**) в новый цвет:

```
unit DrawUnit;

uses
   Microsoft.SmallBasic.Library, System;

type
   Draw = class

      // переключатель:
      private rele := true;
      private rand := new Random();

      // ТАЙМЕР
      private procedure OnTimer();
      begin
         var red := rand.Next(256);
         var green := rand.Next(256);
         var blue := rand.Next(256);
         GraphicsWindow.BackgroundColor :=
              GraphicsWindow.GetColorFromRGB(red, green, blue);
         if (rele) then
            GraphicsWindow.Show()
         else
            GraphicsWindow.Hide();

         // переключаем реле:
         rele := not rele;
      end;

      public procedure Prepare();
      begin
         GraphicsWindow.Hide();
         GraphicsWindow.Title := 'Светоформа';
         GraphicsWindow.Width := 400;
         GraphicsWindow.Height := 200;
         GraphicsWindow.BackgroundColor := 'Green';
         GraphicsWindow.Show();

         Timer.Interval := 1000;
```

```
        Timer.Tick += OnTimer;
      end;
  end; //end of class
end.
```

Последняя клавиша

Свойство **LastKey** *Графического окна* содержит название последней нажатой на клавиатуре клавиши (Рис. 2.13):

```
while(true ) do
    GraphicsWindow.Title := GraphicsWindow.LastKey;
```

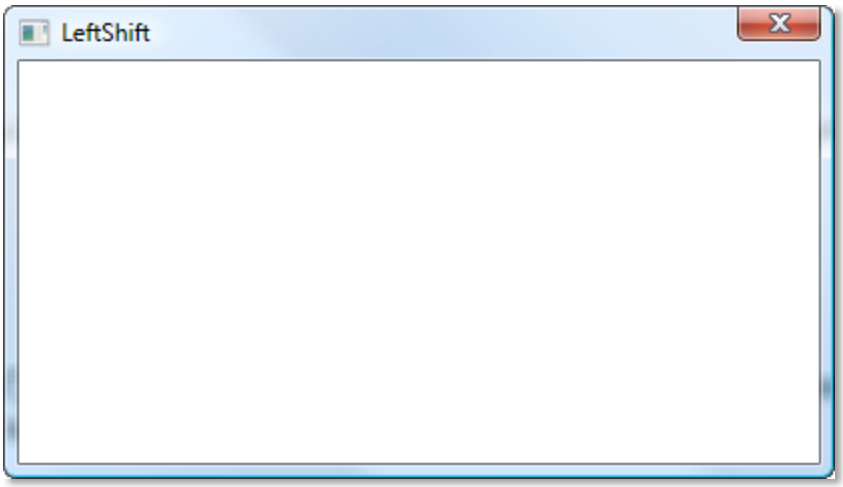


Рис. 2.13. Угадайте с трёх раз, какая клавиша была нажата?

Исходный код программы находится в папке **Графическое окно**.

Так как класс *Графического окна* имеет очень много методов и свойств, то его изучение мы продолжим в следующих главах.

Класс *Desktop* (*Рабочий стол*)

Поскольку окно приложения «лежит» на *Рабочем столе*, то давайте познакомимся с классом **Desktop**, который как раз и описывает свойства *Рабочего стола*.

Класс *Desktop* - это один из самых «бедных» классов – у него всего два свойства и один метод.

Если вы взглянете на *Рабочий стол*, то сразу поймёте, что его самые главные свойства – это *ширина, высота* (конечно, при этом имеются в виду не сантиметры, что важно для нас, а *пиксели*, с которыми компьютер дружит больше) и *фоновая картинка* (почему-то называемая обоями!).

Ширина *Рабочего стола* хранится в свойстве **Width**:

```
Desktop.Width : integer;
```

а **высота** - в свойстве **Height**:

```
Desktop. Height : integer;
```

Оба свойства предназначены **только для чтения**, это значит, что вы *можете узнать* размеры стола, но *не можете их изменить*. Казалось бы, размеры *Рабочего стола* вообще изменить нельзя, поскольку он существует только на экране монитора, а тот «не резиновый». На самом деле это не так: мы измеряем экран в *пикселях*, но пиксели можно сделать и больше и меньше – в зависимости от *разрешения* экрана, а его изменить очень даже можно.

Метод **SetWallPaper**

```
Desktop. SetWallPaper (fileName : string);
```

заменит картинку на *Рабочем столе* той, что вы указали в скобках. Файл с картинкой может находиться на диске вашего компьютера или на любом сайте (в этом случае нужно указать его полный адрес).

Поскольку обои меняют нечасто, то лучше это сделать более естественным способом!

Часто требуется **установить окно приложения по центру *Рабочего стола***. Если вы знаете размеры *Рабочего стола*, то свойствам **Left** и **Top** *Графического окна* следует задать такие значения:

```
GraphicsWindow.Left := (Desktop.Width - GraphicsWindow.Width) / 2;
GraphicsWindow.Top := (Desktop.Height - GraphicsWindow.Height) / 2;
```

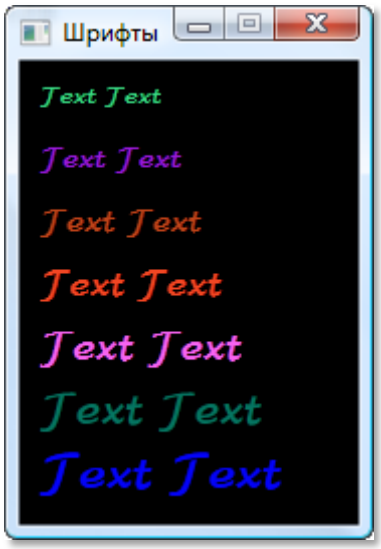
Глава 3. Текст в *Графическом окне*

Выводить информацию в заголовок окна было весьма забавно, но куда красивее можно напечатать текст в самом окне!

Вы можете печатать текст *любым* шрифтом, установленным на вашем компьютере. Достаточно присвоить свойству **FontName** название шрифта (Рис. 3.1):

```
GraphicsWindow.FontName := 'Arial';
```

Не пользуйтесь редкими шрифтами, если планируете передавать программу другим пользователям! Если у них такого шрифта не окажется, он будет заменён стандартным (Рис. 3.2).



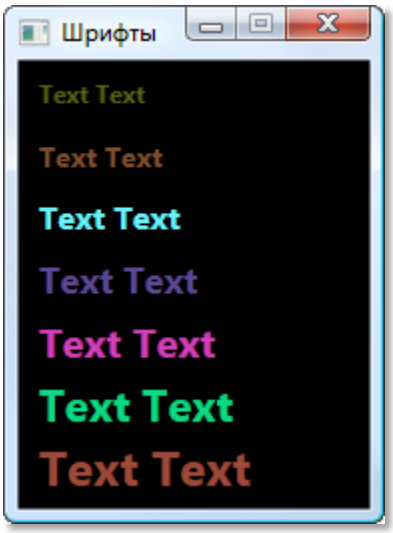


Рис. 3.1. Надпись шрифтом *Arbat*

Рис. 3.2. Шрифт *Arbat* не установлен

Практически на всех компьютерах стоят такие шрифты (Рис. 3.3).

Вот **программа**, которая напечатала все эти надписи:

```
public procedure Prepare();
begin
   GraphicsWindow.Hide();
   GraphicsWindow.Title := 'Шрифты';
   GraphicsWindow.BackgroundColor := 'Black';
```

```
   GraphicsWindow.Show();
end;


public procedure Draw();
begin
   //GraphicsWindow.FontName := 'Arial';
   //GraphicsWindow.FontName := 'Arbat';
   //GraphicsWindow.FontName := 'Courier New';
   GraphicsWindow.FontName := 'Times New Roman';
   //GraphicsWindow.FontName := 'Consolas';

   for var i := 0 to 6 do
   begin
      GraphicsWindow.FontSize := 12 + i * 2;
      GraphicsWindow.BrushColor :=
            GraphicsWindow.GetRandomColor();
      GraphicsWindow.DrawText(10, 10 + i * 30, 'Text Text');
   end;
end;
```

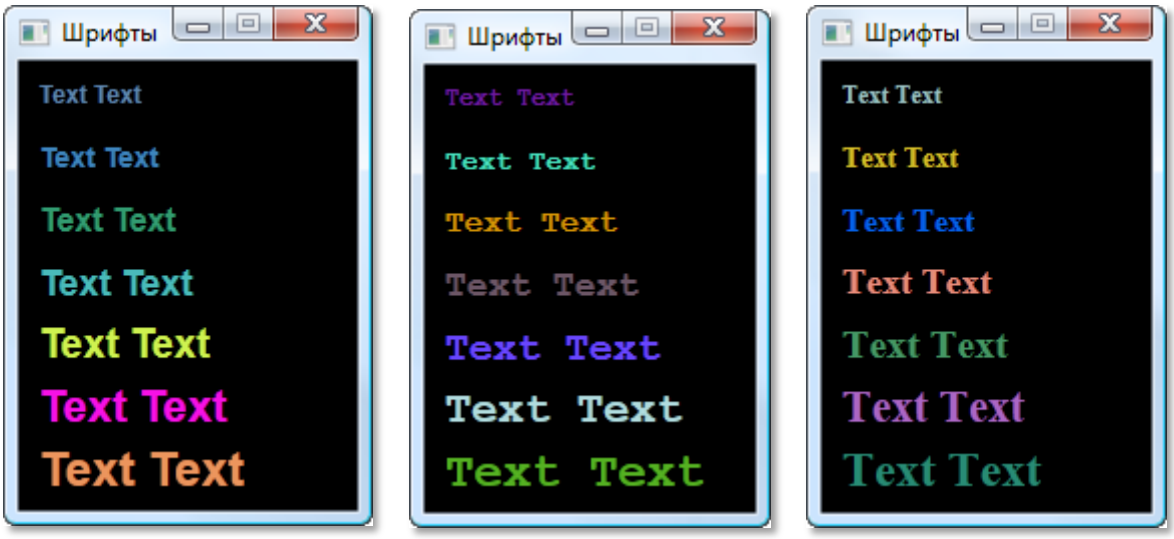


Рис. 3.3. *Arial* *Courier* New *Times New Roman*

Как видите, **размер** шрифта определяется свойством **FontSize**, а его **цвет** – свойством **BrushColor**.

Сам же текст выводится методом:

DrawText(**x** : double; **y** : double; **строка** : string);

- *Первое число* в скобках (*x*) – координата начала строки в пикселях, которая отсчитывается от левого края клиентской области окна.
- *Второе число* (*y*) – координата верхней границы строки, которая отсчитывается от верхнего края клиентской области.
- Затем следует указать **строку** для вывода на экран.

По умолчанию все надписи выполняются **жирным** шрифтом. Если вам больше по душе тонкие буквы, измените свойство **FontBold** (Рис. 3.4) :

```
GraphicsWindow.FontBold :=  false;
```

Буквы снова станут **жирными**, если вы вернёте этому свойству прежнее значение:

```
GraphicsWindow.FontBold :=  true;
```

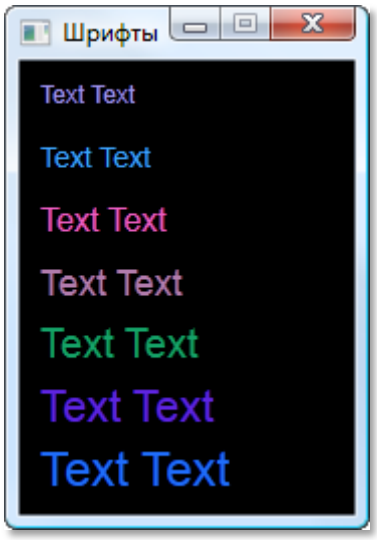


Рис. 3.4. Похудевший *Arial*

А такая строка заставит *склониться* все буквы в почтении (Рис. 3.5):

```
GraphicsWindow.FontItalic := true;
```

Но вы легко вернёте им самоуважение:

```
GraphicsWindow.FontItalic := false;
```

Для печати текста «в рамку» предназначен метод

DrawBoundText(**x** : double; **y** : double; **width** : double; **строка** : string);

который отличается от рассмотренного нами метода *DrawText* только тем, что он имеет дополнительный параметр *width*, который определяет максимальную ширину текста в окне.

Рис. 3.5. *Курсивный* шрифт

Попробуем напечатать строку цифр, ограничив её длину:

```
public procedure Draw2();
begin
   GraphicsWindow.FontName := 'Arial';
   GraphicsWindow.FontSize := 24;
   GraphicsWindow.BrushColor := 'Yellow';
   GraphicsWindow.FontBold := false;
   GraphicsWindow.BrushColor := GraphicsWindow.GetRandomColor();
   GraphicsWindow.DrawBoundText(10, 10, 150,
                                '12345678901234567890');
end;
```

Запустив программу, вы увидите только первые девять цифр и многоточие, которое сигнализирует о том, что часть строки напечатать не удалось (Рис. 3.6).

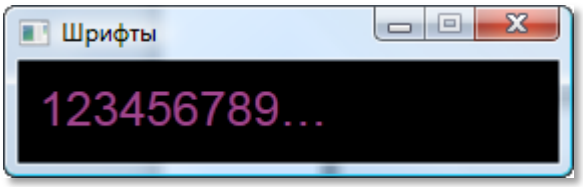


Рис. 3.6. Текст вышел за рамку!

Раздвинем границы рамки (Рис. 3.7):

```
GraphicsWindow.DrawBoundText(10, 10, 350,
                             '12345678901234567890');
```

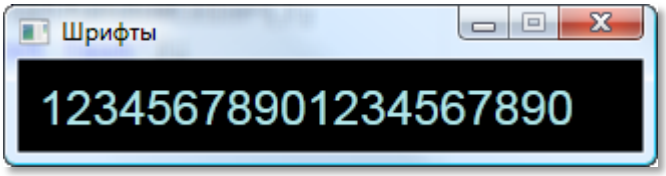


Рис. 3.7. А вот теперь всё нормально!

Проект *События Графического окна*

Исходный код программы находится в папке **События Графического окна**.

Операционная система *Windows* управляет всеми приложениями с помощью **сообщений**, которые она им посылает. Эти сообщения принимает окно приложения и реагирует на них. Например, если пользователь нажмёт какую-нибудь клавишу, система *Windows* посылает активному окну сообщение *WM_KEYDOWN*. Если это наше *Графическое окно*, то в нём возникает событие *KeyDown*. Если нас это событие интересует, то мы можем написать **метод-обработчик** для этого события и указать *Графическому окну* его название:

```
{$apptype windows}

// События Графического окна

uses
   DrawUnit;


begin
   var draw := new Draw();
   draw.Prepare();
end.

 public procedure Prepare();
 begin
    GraphicsWindow.Hide();
    GraphicsWindow.Title := 'События Графического окна';
    GraphicsWindow.Width := 400;
    GraphicsWindow.Height := 200;
    GraphicsWindow.CanResize := false;
    GraphicsWindow.BackgroundColor := 'Black';
    GraphicsWindow.Show();
    GraphicsWindow.KeyDown += OnKeyDown;
 end;


 // МЕТОД-ОБРАБОТЧИК НАЖАТИЯ КЛАВИШИ
 private procedure OnKeyDown();
 begin
    GraphicsWindow.ShowMessage('Вы нажали клавишу', 'СОБЫТИЯ');
 end;
```

Запустите программу, и как только вы нажмёте клавишу, получите диалоговое окно с приятным известием (Рис. 3.8).

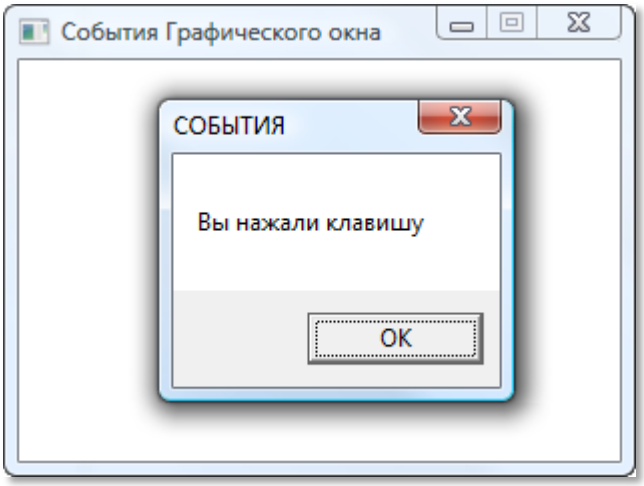


Рис. 3.8. Вам «письмо»!

Вы легко узнаете, какая именно клавиша была нажата, прочитав свойство **LastKey** (Рис. 3.9):

```
private procedure OnKeyDown();
begin
   //GraphicsWindow.ShowMessage('Вы нажали клавишу', 'СОБЫТИЯ');
   var s := 'Вы нажали клавишу ' + GraphicsWindow.LastKey.ToString();
   GraphicsWindow.ShowMessage(s, 'СОБЫТИЯ');
end;
```

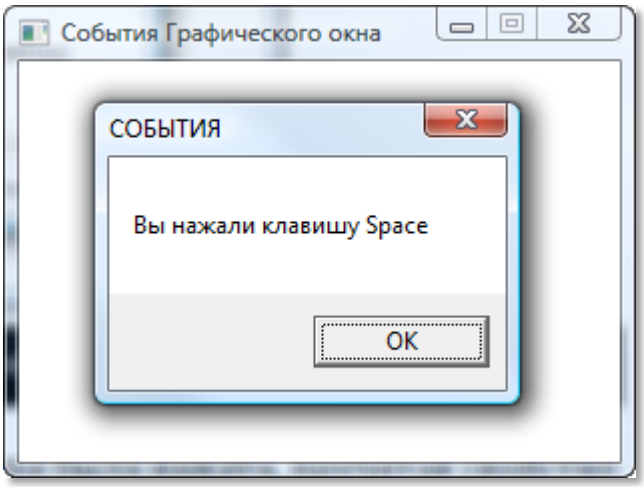


Рис. 3.9. Система знает всё!

При отпускании клавиши возникает событие **KeyUp**, которое вы также можете обработать с помощью метода **OnKeyUp** (Рис. 3.10):

```
public procedure Prepare();
begin
   . . .
   //GraphicsWindow.KeyDown += OnKeyDown;
   GraphicsWindow.KeyUp += OnKeyUp;
end;

// МЕТОД-ОБРАБОТЧИК ОТПУСКАНИЯ КЛАВИШИ
private procedure OnKeyUp();
begin
   var s := 'Вы отпустили клавишу ' +
            GraphicsWindow.LastKey.ToString();
   GraphicsWindow.ShowMessage(s, 'СОБЫТИЯ');
end;
```

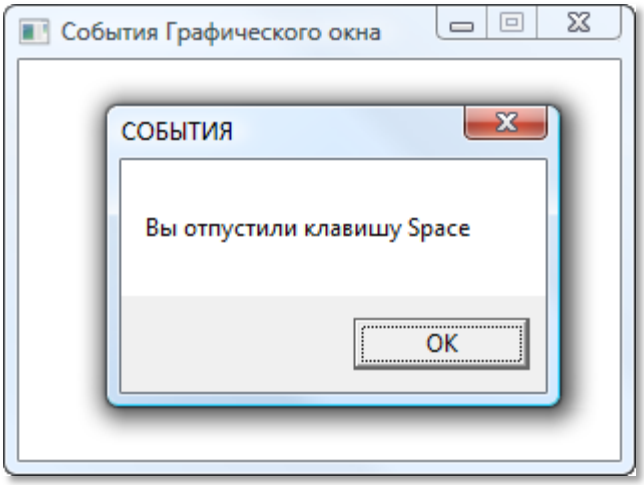


Рис. 3.10. Она и это знает!

Попутно вы познакомились с методом *Графического окна*

ShowMessage(**строка** : string; **заголовок** : string);

который очень часто используется для вывода информации, когда нужно приостановить выполнение программы, например, при отладке.

- *Первый* параметр этого метода – строка с сообщением
- *Второй* – заголовок окна, который обычно говорит о его назначении.

Проект *Розыгрыш*

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

Обогатившись знаниями о событиях, вы можете написать небольшую программу, которая будет самым живым образом реагировать на действия доверчивого пользователя:

```
{$apptype windows}

// Розыгрыш

uses
   DrawUnit;

begin
   var draw := new Draw();
   draw.Prepare();
end.

unit DrawUnit;

uses
   Microsoft.SmallBasic.Library, System;

type
   Draw = class

      public procedure Prepare();
      begin
         GraphicsWindow.Hide();
         GraphicsWindow.Title := ' Нажмите клавишу!';
         GraphicsWindow.Width := 490;
         GraphicsWindow.Height := 60;
         GraphicsWindow.CanResize := false;
         GraphicsWindow.BackgroundColor := 'Black';
         GraphicsWindow.FontSize := 24;
         //GraphicsWindow.Show();
         GraphicsWindow.Left := (Desktop.Width –
                                 GraphicsWindow.Width) / 2;
         GraphicsWindow.Top := (Desktop.Height –
                                GraphicsWindow.Height) / 2;
         GraphicsWindow.Show();
         GraphicsWindow.KeyDown += OnKeyDown;
```

```
        GraphicsWindow.KeyUp += OnKeyUp;
    end;

    // МЕТОД-ОБРАБОТЧИК НАЖАТИЯ КЛАВИШИ
    private procedure OnKeyDown();
    begin
        GraphicsWindow.Clear();
        GraphicsWindow.BrushColor := 'Red';
        GraphicsWindow.DrawText(10, 10,
                                ' Не нажимайте клавишу, мне больно!');
    end;

    // МЕТОД-ОБРАБОТЧИК ОТПУСКАНИЯ КЛАВИШИ
    private procedure OnKeyUp();
    begin
        GraphicsWindow.Clear();
        GraphicsWindow.BrushColor := 'Green';
        GraphicsWindow.DrawText(10, 10,
                                ' Спасибо, что вы отпустили клавишу!');
    end;

  end; //end of class
end.
```

Когда пользователь нажмёт клавишу, в окне появится жалобное сообщение (Рис. 3.11).

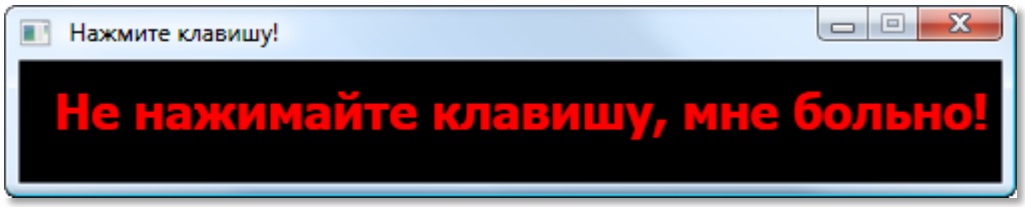


Рис. 3.11. А, может, это правда?!

Гуманный пользователь тут же отпустит клавишу, за что получит благодарность (Рис. 3.12).

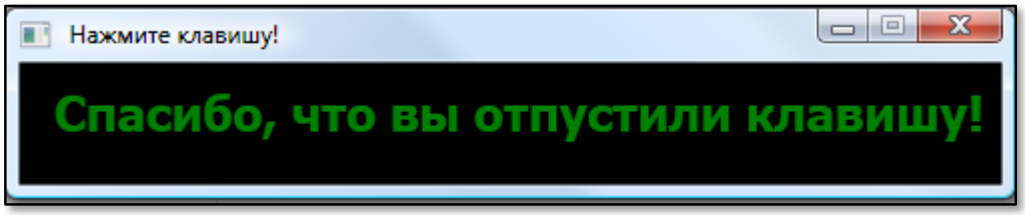


Рис. 3.12. Никогда не делайте больно!

Обратите внимание на метод **Clear**, который очищает окно от любых надписей и рисунков, закрашивая клиентскую область в цвет *BackgroundColor*.

Как видите, ничего нового в программе нет, а получилось смешно!

Проект *Розыгрыш и продолжаются!*

Исходный код программы находится в папке **Розыгрыш и продолжаются!**

Поскольку кроме клавиатуры, у пользователя всегда под рукой **мышка**, то вы вполне можете предположить, что *Графическое окно* реагирует и на мышиную «возню». И вы не ошиблись в своих ожиданиях: реагирует, да ещё как!

Мы не можем и не должны останавливаться на констатации сего приятного факта, а наоборот, сразу же примемся использовать мышиные события для продолжения розыгрышей.

Снова будем выводить в окно смешные надписи. При нажатии и отпускании кнопки мыши одна надпись в окне приложения будет сменяться другой. На этот раз мы возьмём замечательные выражения из «негритянских» рассказов юмориста Лукинского:

```
{$apptype windows}

// Розыгрыши продолжаются!

uses
   DrawUnit;


begin
   var draw := new Draw();
   draw.Prepare();
```

```
end.

unit DrawUnit;

uses
   Microsoft.SmallBasic.Library, System;

type
   Draw = class

      // МЕТОД-ОБРАБОТЧИК НАЖАТИЯ КНОПКИ МЫШКИ
      private procedure OnMouseDown();
      begin
         GraphicsWindow.Title := 'Отпусти кнопку мышки!';
         GraphicsWindow.Clear();
         GraphicsWindow.BrushColor := 'Blue';
         GraphicsWindow.DrawText(10, 10, ' С Новым годом!');
      end;

      // МЕТОД-ОБРАБОТЧИК ОТПУСКАНИЯ КНОПКИ МЫШКИ
      private procedure OnMouseUp();
      begin
         GraphicsWindow.Title := 'Нажми кнопку мышки!';
         GraphicsWindow.Clear();
         GraphicsWindow.BrushColor := 'Green';
         GraphicsWindow.DrawText(10, 10, ' Пошёл на фиг!');
      end;

      public procedure Prepare();
      begin
         GraphicsWindow.Hide();
         GraphicsWindow.Title := ' Нажми кнопку мышки!';
         GraphicsWindow.Width := 300;
         GraphicsWindow.Height := 60;
         GraphicsWindow.CanResize := false;
         GraphicsWindow.BackgroundColor := 'White';
         GraphicsWindow.FontSize := 32;
         GraphicsWindow.Left := (Desktop.Width -
                                 GraphicsWindow.Width) / 2;
         GraphicsWindow.Top := (Desktop.Height -
                                GraphicsWindow.Height) / 2;
         GraphicsWindow.Show();
         GraphicsWindow.MouseDown += OnMouseDown;
         GraphicsWindow.MouseUp += OnMouseUp;
      end;

   end; //end of class
end.
```

Отозвавшись на просьбу окна, выраженную в его заголовке, доверчивый пользователь нажмёт кнопку мышки и получит поздравление с праздником (Рис. 3.13).

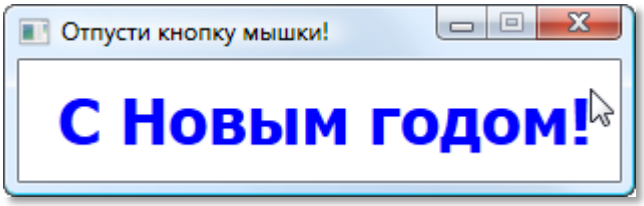


Рис. 3.13. Спасибо!

Теперь окно попросит его отпустить кнопку мышки и недобро пошлёт его - в другое приложение (Рис. 3.14).

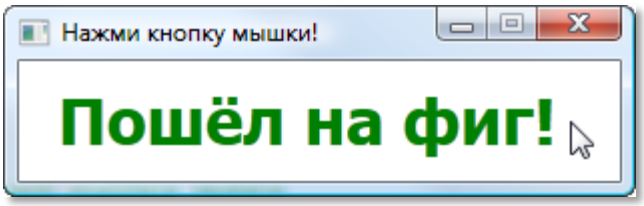


Рис. 3.14. Ну, Лукинский даёт!

Согласитесь, жить стало веселей. А вы можете повеселиться ещё больше, если заготовите множество таких фраз и подсунете их своим товарищам, близким и родственникам. Они непременно будут вам благодарны за доставленное удовольствие.

Проект *Прикольное окно*

Исходный код программы находится в папке **Прикольное окно**.

Попробуем сделать окно «прикольным». Прикол будет незатейливым: при попытке пользователя переместить окно, оно будет упрямо возвращаться в середину экрана (Рис. 3.15).

```
{$apptype windows}

// Прикольное окно

uses
   DrawUnit;


begin
   var draw := new Draw();
   draw.Prepare();
end.

unit DrawUnit;

uses
   Microsoft.SmallBasic.Library, System;

type
   Draw = class

      // ТАЙМЕР
      private procedure OnTimer();
      begin
         GraphicsWindow.Left := (Desktop.Width –
                                 GraphicsWindow.Width) / 2;
         GraphicsWindow.Top := (Desktop.Height –
                                GraphicsWindow.Height) / 2;
      end;

      public procedure Prepare();
      begin
         GraphicsWindow.Hide();
         GraphicsWindow.Title := ' Прикол';
```

```
        GraphicsWindow.Width := 420;
        GraphicsWindow.Height := 60;
        GraphicsWindow.CanResize := false;
        GraphicsWindow.BackgroundColor := 'Black';
        GraphicsWindow.BrushColor := 'Blue';
        GraphicsWindow.FontSize := 32;
        GraphicsWindow.Left := (Desktop.Width -
                                GraphicsWindow.Width) / 2;
        GraphicsWindow.Top := (Desktop.Height -
                               GraphicsWindow.Height) / 2;
        GraphicsWindow.Show();
        GraphicsWindow.DrawText(10, 10, 'Передвинь меня в угол!');

        Timer.Interval := 10;
        Timer.Tick += OnTimer;
      end;

  end; //end of class
end.
```

Рис. 3.15. Окно на приколе

Действует программа очень просто. Каждые 10 миллисекунд срабатывает таймер, и метод-обработчик **OnTimer** возвращает окно на место.

Проект *Ввод текста в Графическом окне*

Исходный код программы находится в папке **Ввод текста в Графическом окне**.

Графическое окно реагирует ещё на одно событие, которое может пригодиться вам, например, для ввода информации с клавиатуры.

Когда пользователь нажимает клавишу, возникает событие *TextInput*, а свойство **LastText** содержит символ, соответствующий этой клавише, - букву, цифру, знак препинания и т.д. В методе-обработчике вы можете составить из этих символов строку (Рис. 3.16), а затем вывести на экран или использовать в программе как-то иначе.

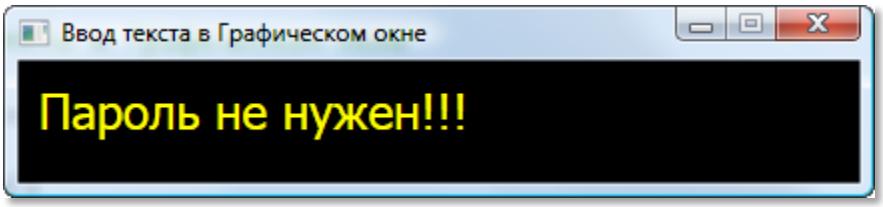


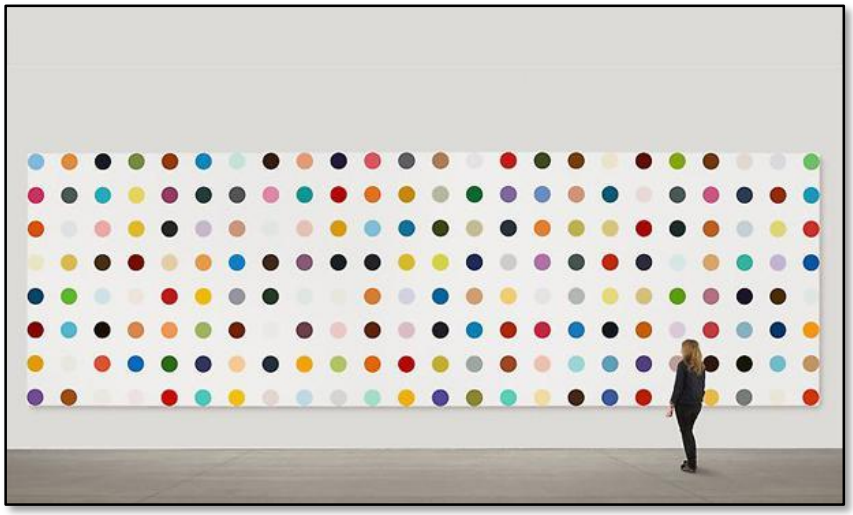
Рис. 3.16. Строка, собранная по буквам!

```
{$apptype windows}

// Ввод текста в Графическом окне

uses
   DrawUnit;


begin
   var draw := new Draw();
   draw.Prepare();
end.


unit DrawUnit;
```

```
uses
   Microsoft.SmallBasic.Library, System;

type
   Draw = class

      private s := String.Empty;

      // ТАЙМЕР
      private procedure OnText();
      begin
         GraphicsWindow.Clear();
         s += GraphicsWindow.LastText;
         GraphicsWindow.DrawText(10, 10, s);
      end;

      public procedure Prepare();
      begin
         GraphicsWindow.Hide();
         GraphicsWindow.Title := 'Ввод текста в Графическом окне';
         GraphicsWindow.Width := 420;
         GraphicsWindow.Height := 60;
         GraphicsWindow.CanResize := false;
         GraphicsWindow.BackgroundColor := 'Black';
         GraphicsWindow.BrushColor := 'Yellow';
         GraphicsWindow.FontSize := 24;
         GraphicsWindow.FontBold := false;
         GraphicsWindow.Left := (Desktop.Width –
                                 GraphicsWindow.Width) / 2;
         GraphicsWindow.Top := (Desktop.Height –
                                GraphicsWindow.Height) / 2;
         GraphicsWindow.Show();

         GraphicsWindow.TextInput += OnText;
      end;

   end; //end of class
end.
```

В данном примере после ввода каждой буквы приходится стирать всё окно, иначе надписи накладываются друг на друга, что выглядит неаккуратно. Скоро вы познакомитесь с геометрическими фигурами, которые можно рисовать в *Графическом окне*, и они помогут вам стирать любую часть клиентской области, не причиняя вреда окружающей среде.

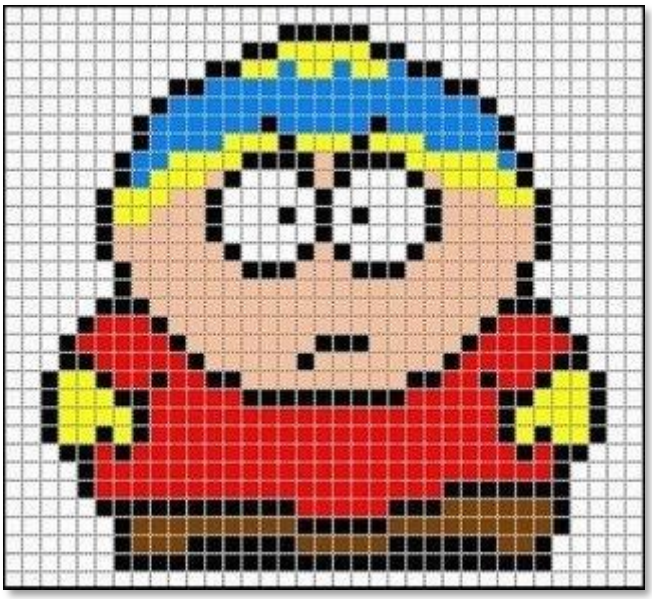
Глава 4. В каждой строчке только …



Из простых геометрических фигур с помощью компьютера можно создавать впечатляющие композиции. Клиентская область *Графического окна* выполняет ту же роль в *компьютерной графике*, что и холст в настоящей живописи. Нам уже довелось писать на ней разные словечки, а теперь мы возьмёмся за настоящее геометрическое творчество.

Проект *Пуантилизм, или Ставим точки*

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



Мостовая пусть качнётся, как очнётся!
Пусть начнётся, что ещё не началось!
Вы рисуйте, вы рисуйте,
вам зачтётся...
Что гадать нам:
удалось - не удалось?

Булат Окуджава. *Живописцы*

Самым простым геометрическим объектом является **точка**. Она, как известно из геометрии, размеров не имеет. По той же причине она не имеет ни цвета, ни запаха. Таким образом, настоящую точку увидеть нельзя, а вот компьютерную - можно, хотя размером она примерно в четверть миллиметра. Называется такая точка **пикселем**. Его можно окрасить в миллионы цветов, а из множества пикселей мы сумеем нарисовать на экране монитора любую картину.

Метод

```
SetPixel(x : integer; y : integer; color : string);
```

Графического окна закрашивает пиксель клиентской области окна (для краткости мы будем называть её *холстом* или *канвой*) с указанными координатами **(x, y)** в цвет **color**, который можно задавать любым способом из тех, что мы уже рассмотрели. А вот с координатами не всё так просто.

Расположение координатных осей на канве может показаться странным: ось *Y* (ордината) направлена **вниз**, а не вверх, поэтому, чем *больше* значение *Y*, тем *ниже* будет точка на экране. Положительное направление оси *X* совпадает с нашими представлениями, а вот начало координат находится в **левом верхнем** углу канвы, а вовсе не в её центре, как мы могли бы того ожидать (Рис. 4.1).

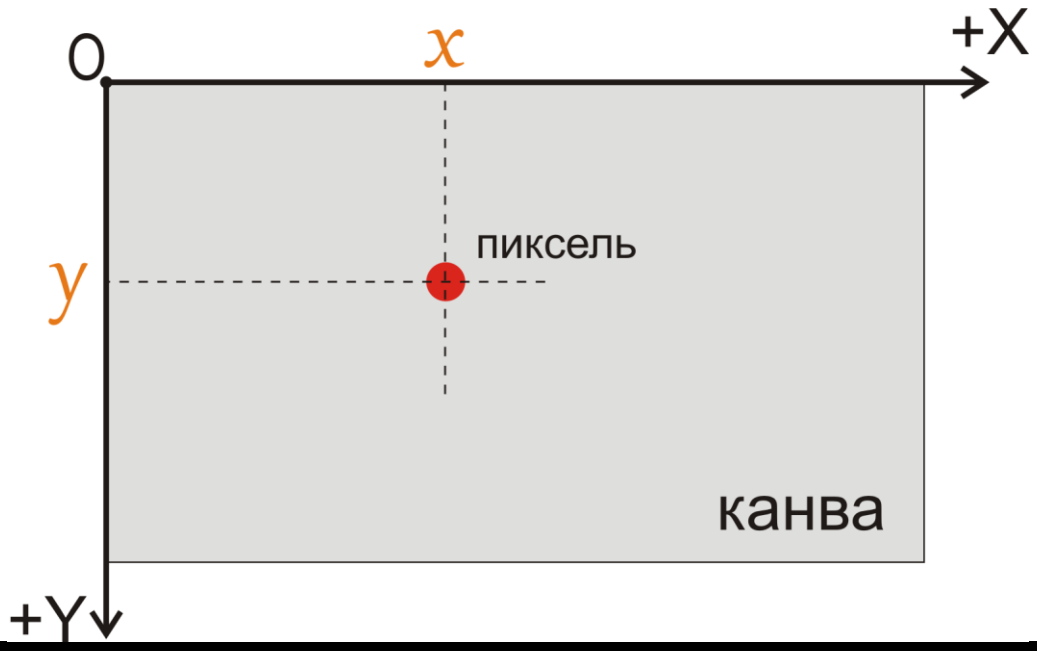


Рис. 4.1. Координатная система клиентской области окна приложения

Конечно, рисовать точками лучше в каком-нибудь графическом редакторе, чем в программе на *паскале*, но поскольку все фигуры построены из точек, то давайте напишем простенькую программу, которая будет самостоятельно выводить на канву точки случайно выбранного цвета. Эстетического удовольствия вы не получите никакого, но зато хорошенько познакомитесь с методом *SetPixel*. Программа **Пиксели** просто окрашивает точки канвы в случайные цвета, поэтому картинка получается совершенно хаотическая (Рис. 4.2).

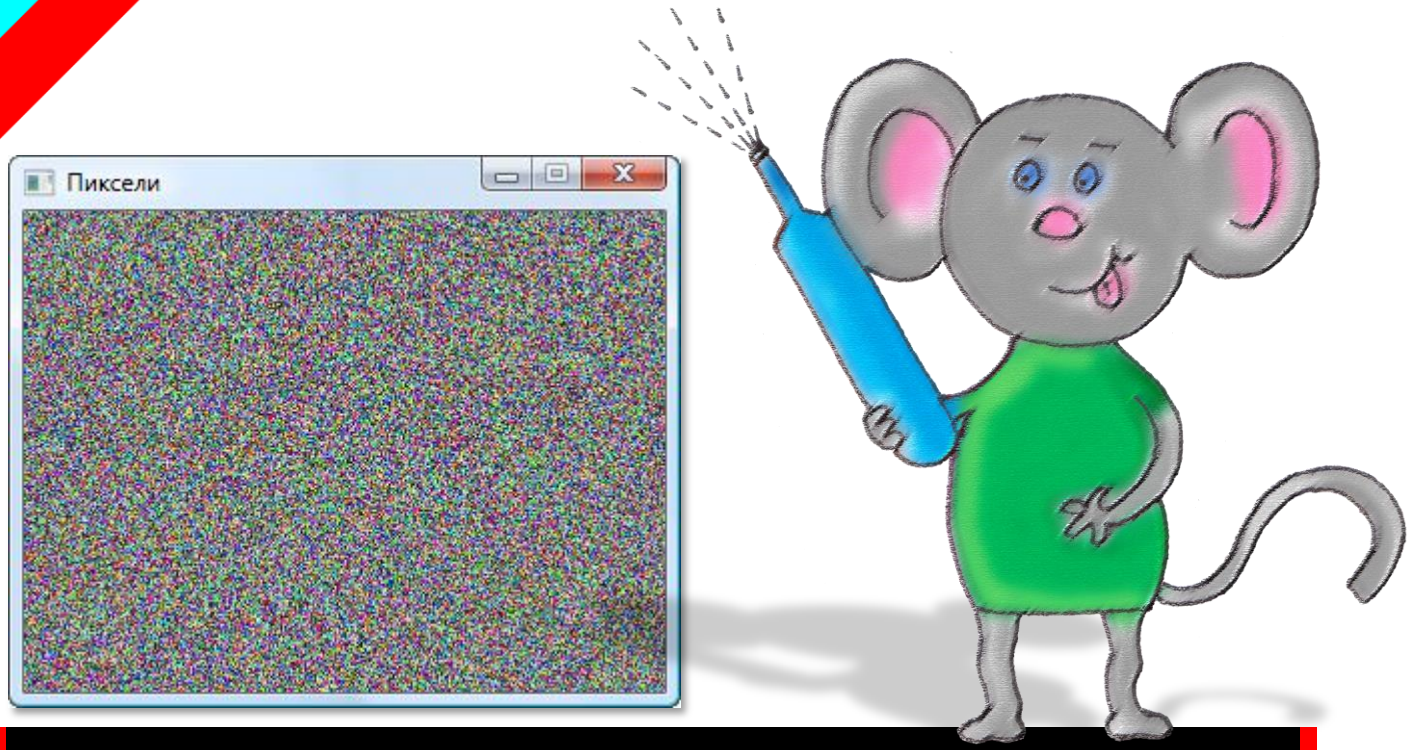


Рис. 4.2. Пёстро!

Поскольку рисование точек на канве процесс довольно неспешный, то не задавайте большие размеры окна. Результат будет ничем не лучше, а времени вы потратите больше!

Низкая скорость рисования объясняется просто: на самом деле программа не закрашивает пиксели канвы, а рисует малюсенькие **квадратики** со сторонами в 1 пиксель. Понятно, что на вычерчивание десятков тысяч квадратиков потребуется немало времени!

```
{$apptype windows}


uses
   DrawUnit;

begin
   var draw := new Draw();
   draw.Prepare();
   draw.Draw();
end.

     public procedure Prepare();
      begin
         GraphicsWindow.Hide();
         GraphicsWindow.Title := 'Пиксели';
         GraphicsWindow.Width := 320;
         GraphicsWindow.Height := 240;
         GraphicsWindow.CanResize := false;
         GraphicsWindow.BackgroundColor := 'Black';
         GraphicsWindow.Left := (Desktop.Width –
```

```
                                GraphicsWindow.Width) / 2;
        GraphicsWindow.Top := (Desktop.Height -
                                GraphicsWindow.Height) / 2;
        GraphicsWindow.Show();
    end;
```

В бесконечном цикле *while* программа последовательно перебирает все точки канвы и окрашивает их с помощью метода *SetPixel*:

```
    public procedure Draw();
    begin
      // Окрашиваем пиксели канвы в разные цвета:
      var height: integer := GraphicsWindow.Height;
      var width: integer := GraphicsWindow.Width;
      while(true) do
      begin
        for var y := 0 to height - 1 do
        begin
          for var x := 0 to width - 1 do
          begin
            // выбираем случайный цвет для пикселя:
            var clr := GraphicsWindow.GetRandomColor();
            // и окрашиваем его:
            GraphicsWindow.SetPixel(x, y, clr);
          end
        end
      end
    end;
```

Добавим к нашему проекту несколько строчек кода, чтобы получить интересный визуальный эффект: надпись **Пиксели** будет печататься *над* цветными точками (Рис. 4.3).

```
public procedure Draw();
begin
  GraphicsWindow.FontSize := 56;
  GraphicsWindow.BrushColor := 'Red';
        . . .
          // и окрашиваем его:
          GraphicsWindow.SetPixel(x, y, clr);
        end;
        GraphicsWindow.DrawText(40, height / 2 - 40, 'Пиксели');
    end
  end
end;
```

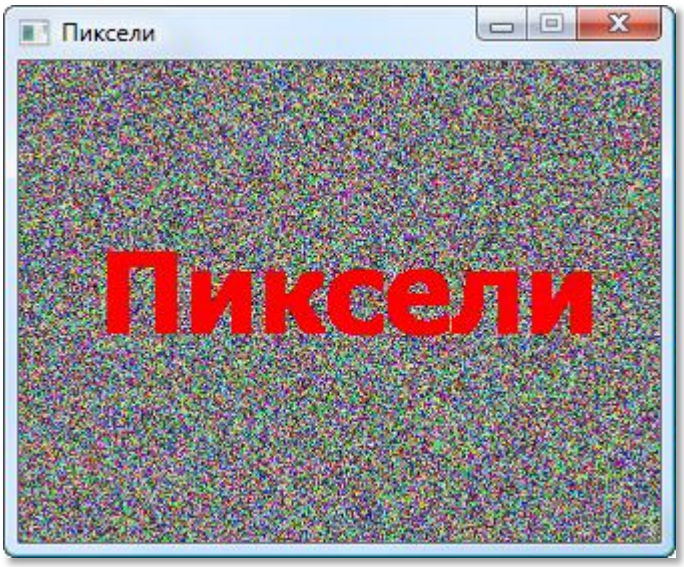


Рис. 4.3. Нестираемая надпись

Проект *Проявляющаяся надпись*

Исходный код программы находится в папке **Проявляющаяся надпись**.

Вместо нестираемой надписи мы запросто получим надпись, **проявляющуюся** при пикселизации канвы.

Для этого окрасим буквы в цвет канвы, и тогда надпись сначала будет совершенно не видна:

```
{$apptype windows}

// Проявляющаяся надпись

uses
   DrawUnit;


begin
   var draw := new Draw();
   draw.Prepare();
   draw.Draw();
end.
```

```
      public procedure Prepare();
      begin
         GraphicsWindow.Hide();
         GraphicsWindow.Title := 'Пиксели 2';
         GraphicsWindow.Width := 320;
         GraphicsWindow.Height := 240;
         GraphicsWindow.CanResize := false;
         GraphicsWindow.BackgroundColor := 'Black';
         GraphicsWindow.Left := (Desktop.Width –
                                  GraphicsWindow.Width) / 2;
         GraphicsWindow.Top := (Desktop.Height –
                                 GraphicsWindow.Height) / 2;
         GraphicsWindow.Show();

         GraphicsWindow.FontSize := 56;
         GraphicsWindow.BrushColor := 'Black'
      end;
```

А дальше, в бесконечном цикле *while* мы окрашиваем случайный пиксель канвы в произвольный цвет:

```
      public procedure Draw();
      begin
         var rand := new Random();
         // Окрашиваем пиксели канвы в разные цвета:
         var height: integer := GraphicsWindow.Height;
         var width: integer := GraphicsWindow.Width;
         while(true) do
         begin
            // случайные координаты пикселя:
            var x := rand.Next(width);
            var y := rand.Next(height);
            // выбираем случайный цвет для пикселя:
            var clr := GraphicsWindow.GetRandomColor();
                  //и окрашиваем его:
            GraphicsWindow.SetPixel(x, y, clr);
```

И печатаем **надпись**:

```
      if (rand.Next(101) > 99) then
         // печатаем надпись цветом фона:
         GraphicsWindow.DrawText(40, height / 2 - 40, 'Пиксели');
   end;
end;
```

Теперь надпись будет как бы проявляться при окрашивании пикселей канвы в разные цвета (Рис. 4.4).

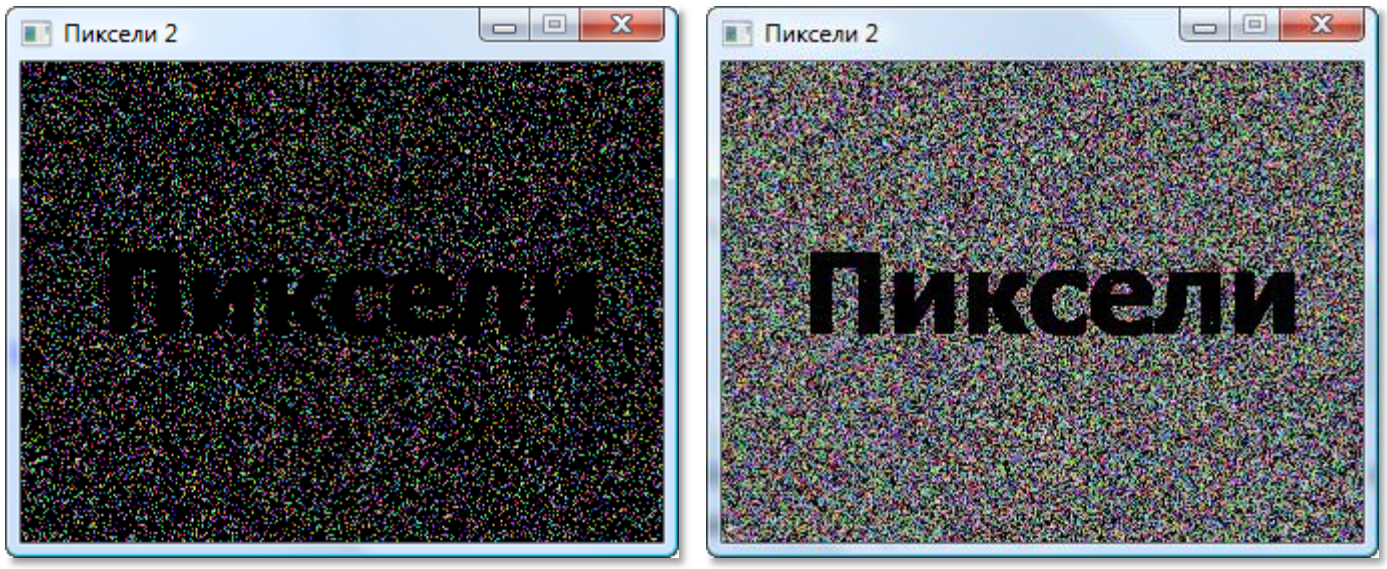


Рис. 4.4. Проявляющаяся надпись

Ещё проще закрасить фон каким-то одним цветом:

```
// выбираем случайный цвет для пикселя:
//var clr := GraphicsWindow.GetRandomColor();
var clr := 'Blue';
```

Эффект также получается интересный (Рис. 4.5)!

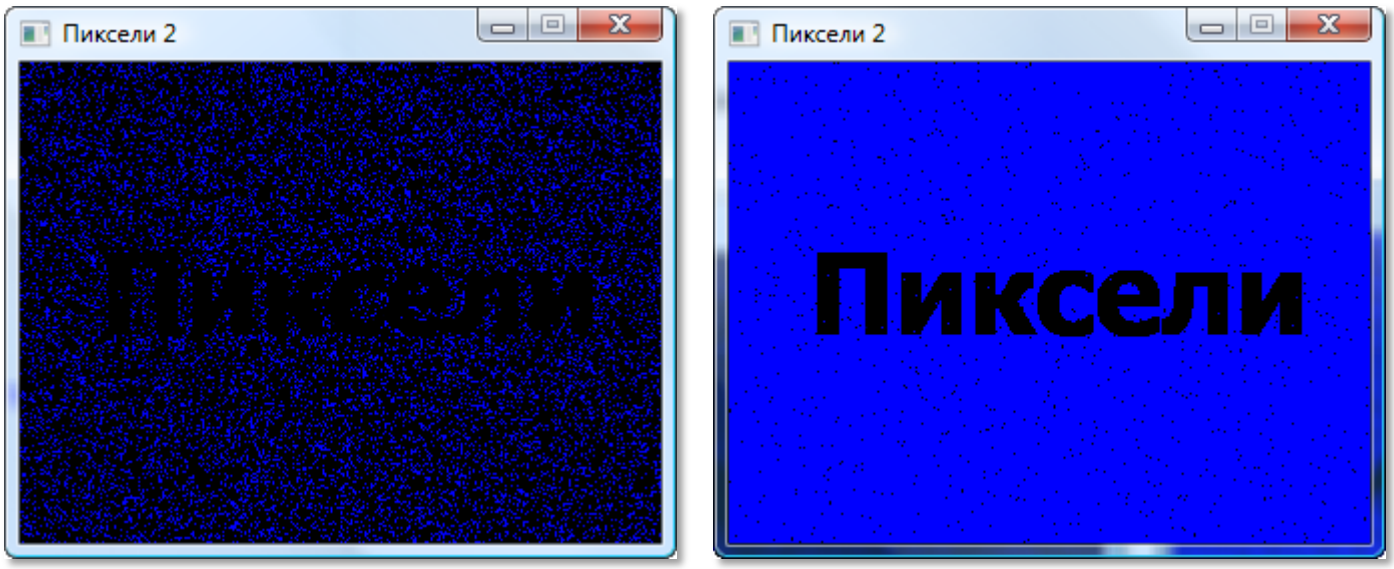


Рис. 4.5. Надпись выступает из фона

Проект *Пипетка*

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

Мы можем представить канву в виде двухмерного **массива** пикселей, каждый из которых характеризуется *координатами* (индексами массива) и *цветом*. Чтобы узнать, какой цвет имеет тот или иной пиксель канвы, достаточно обратиться к свойству *Графического окна*

```
GetPixel(x : integer; y: integer): string;
```

Начните новый проект и сохраните его в папке **Пипетка**.

Пипеткой в графических редакторах называют инструмент, который помогает узнать цвет пикселя под курсором. В этом режиме работы он как бы превращается в пипетку (Рис. 4.6)

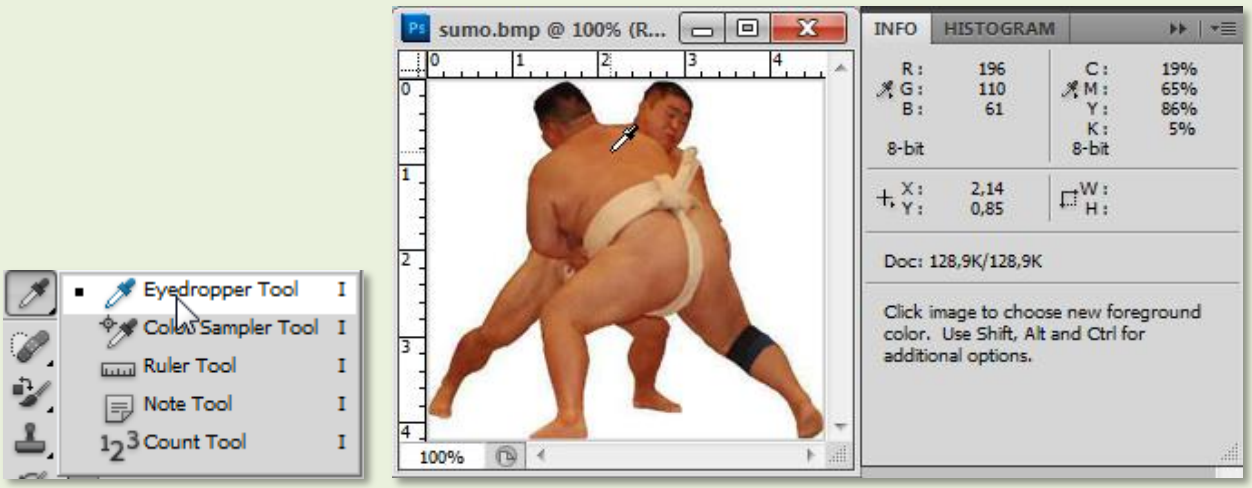


Рис. 4.6. Пипетка в *Фотошопе*

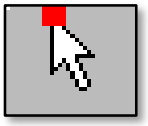
```
{$apptype windows}

// Пипетка

uses
   DrawUnit;
```

```
begin
   Draw.Prepare();
   Draw.Draw();
end.

unit DrawUnit;

uses
   Microsoft.SmallBasic.Library, System, System.Threading;

type
   Draw = class

      const GWWIDTH = 320;
      const GWHEIGHT = 240;

      class procedure Prepare();
      begin
         GraphicsWindow.Hide();
         GraphicsWindow.Title := 'Цвет пикселя';
         GraphicsWindow.Width := GWWIDTH;
         GraphicsWindow.Height := GWHEIGHT;
         GraphicsWindow.Show();
         GraphicsWindow.Left := (Desktop.Width –
                                 GraphicsWindow.Width) / 2;
         GraphicsWindow.Top := (Desktop.Height –
                                GraphicsWindow.Height) / 2;
         GraphicsWindow.CanResize := false;
      end;
```

Сначала мы окрашиваем все пиксели канвы в разные цвета, а затем перемещаем мышку по канве и в строке заголовка читаем координаты курсора и цвет пикселя под ним.

```
      class procedure Draw();
      begin
         var width := GWWIDTH + 10;
         var height := GWHEIGHT + 10;

         // Окрашиваем пиксели канвы в случайные цвета:
         for var y := 0 to height - 1 do
            for var x := 0 to width - 1 do
            begin
               // выбираем случайный цвет для пикселя:
               var clr := GraphicsWindow.GetRandomColor();
               // и окрашиваем его:
               GraphicsWindow.SetPixel(x, y, clr);
            end;
```

Координаты курсора (точнее – его «горячей точки», которая обычно находится в верхнем левом углу) можно легко узнать по значению свойств **MouseX** и **MouseY** *Графического окна*.



Вся интрига этой программы заключается вот в этих строках:

```
  while (true) do
   begin
      var x := GraphicsWindow.MouseX;
      var y := GraphicsWindow.MouseY;
      var clr := GraphicsWindow.GetPixel(x, y);
      var str := 'Цвет пикселя (' + x.ToString() + ','  +
                 y.ToString() + ') = ' + (string)(clr);
      GraphicsWindow.Title := str;
   end;
end;
```

Цвет пикселя под *горячей точкой* курсора (её координаты можно прочитать в заголовке окна) в 16-ричном виде указан там же (Рис. 4.7).

К сожалению, пиксели такие маленькие, что трудно определить их цвет на глаз, поэтому добавим в программу небольшое «окошко», которое будем закрашивать цветом текущего пикселя (Рис. 4.8).

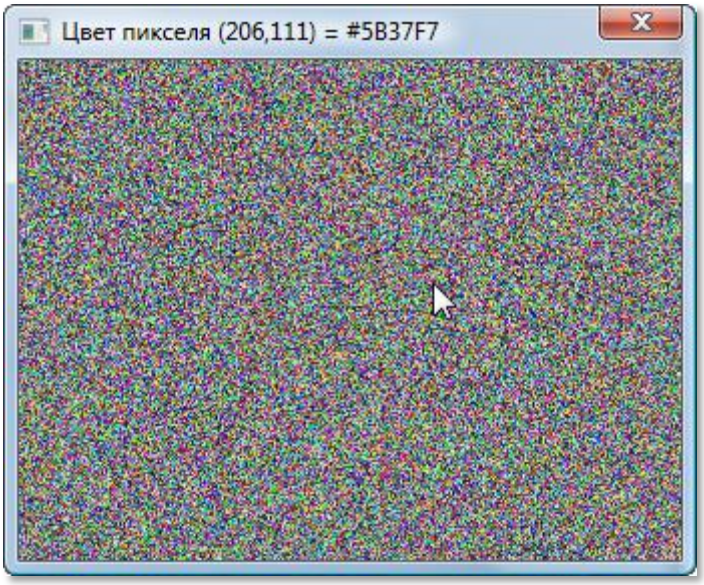


Рис. 4.7. Цветная пипетка в действии!

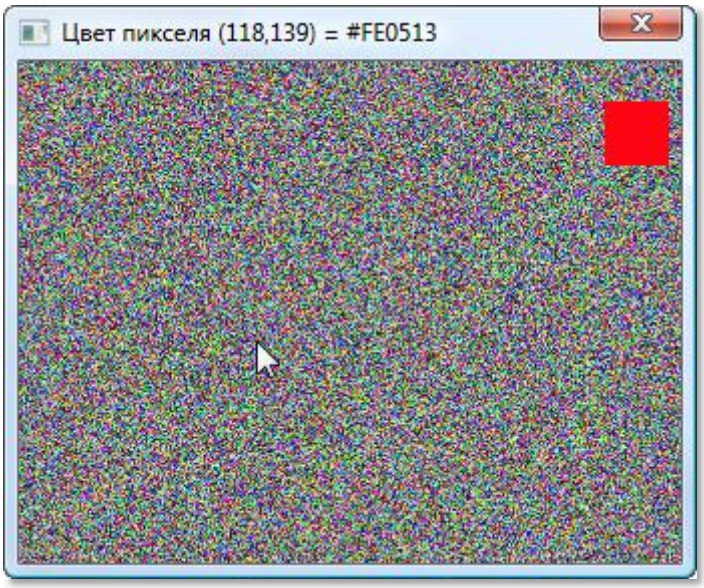


Рис. 4.8. А вот теперь каждый пиксель виден как на ладони!

```
while (true) do
begin
   var x := GraphicsWindow.MouseX;
   var y := GraphicsWindow.MouseY;
   var clr := GraphicsWindow.GetPixel(x, y);
   var str := 'Цвет пикселя (' + x.ToString() + ','  +
              y.ToString() + ') = ' + (string)(clr);
   GraphicsWindow.Title := str;
   // окошко:
   GraphicsWindow.BrushColor := clr;
   GraphicsWindow.FillRectangle(width - 32 - 6, 20, 32, 32);

end;
```

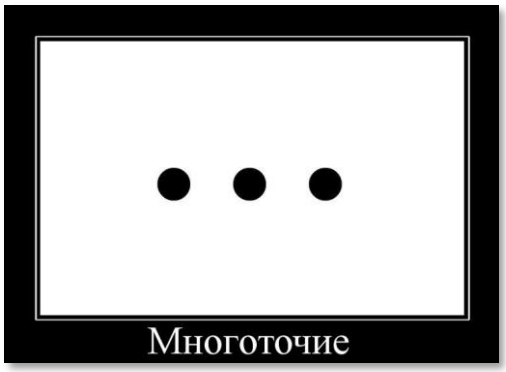
Если вам понравится какой-нибудь цвет, запомните его код и применяйте в программах. Например, так:

```
GraphicsWindow.BrushColor := '#46f9fe';
```

Проект *Многоточие*

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

Раз уж компьютерные точки всё равно имеют размер, то мы сделаем их КРУПНЕЕ, чтобы лучше разглядеть. В этом деле нам потребуется не микроскоп и даже не увеличительное стекло, а новая программа **Многоточие**, которая умеет рисовать огромные точки, так что после её работы экран будет усыпан ими, как новогодний пол – конфетти (Рис. 4.9).



И в этом случае цвет и место точек задаются случайным образом, но теперь картина получается более «художественная».

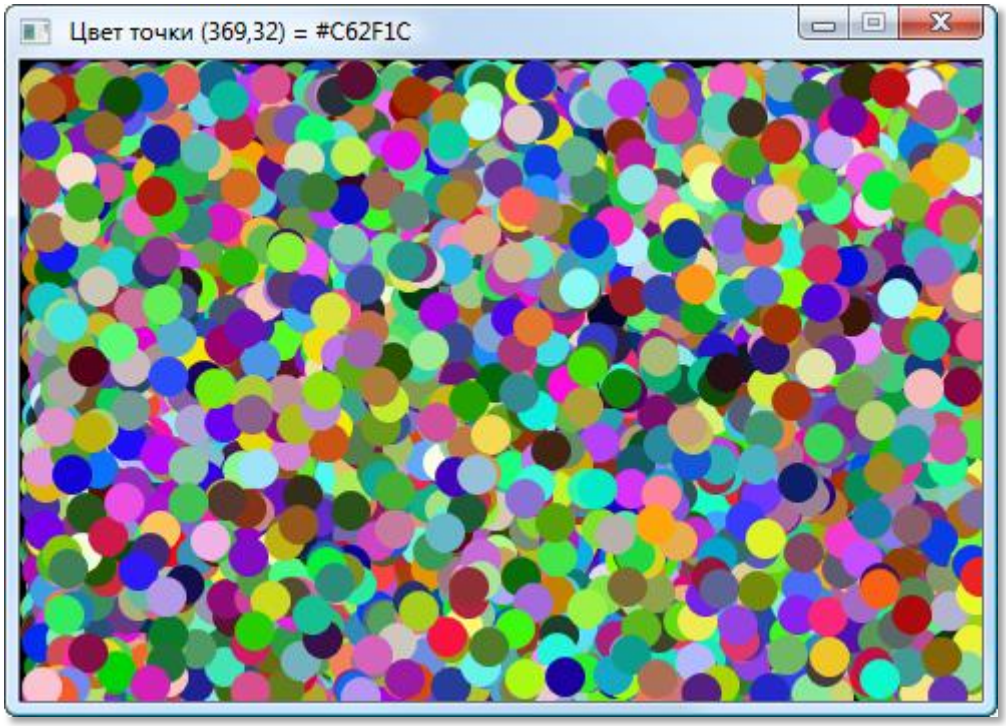


Рис. 4.9. Супер-пиксели!

Начало программы мы просто копируем из предыдущих проектов:

```
{$apptype windows}

// Многоточие
```

```
uses
   DrawUnit;

begin
   var draw := new Draw();
   draw.Prepare();
   draw.Draw();
end.

unit DrawUnit;

uses
   Microsoft.SmallBasic.Library, System;

type
   Draw = class
      private rand := new Random();

      public procedure Prepare();
      begin
         GraphicsWindow.Hide();
         GraphicsWindow.Title := 'Многоточие';
         GraphicsWindow.Width := 480;
         GraphicsWindow.Height := 320;
         GraphicsWindow.CanResize := false;
         GraphicsWindow.BackgroundColor := 'Black';
         GraphicsWindow.Left := (Desktop.Width –
                                  GraphicsWindow.Width) / 2;
         GraphicsWindow.Top := (Desktop.Height –
                                 GraphicsWindow.Height) / 2;
         GraphicsWindow.Show();

         GraphicsWindow.FontSize := 56;
         GraphicsWindow.BrushColor := 'Black'
      end;
```

Но теперь вместо отдельных пикселей мы рисуем «точки»:

```
      public procedure Draw();
      begin
         var height: integer := GraphicsWindow.Height;
         var width: integer := GraphicsWindow.Width;

          // радиус 'точки':
         var radius := 10;
         for var i := 0 to 10000 do
         begin
               // выбираем случайные координаты  'точки':
```

```
            var x := rand.Next(width - radius);
            var y := rand.Next(height - radius);
            // выбираем случайный цвет для  'точки':
            var clr := GraphicsWindow.GetRandomColor();
            GraphicsWindow.BrushColor := clr;
            GraphicsWindow.Title := ' Цвет точки (' + x.ToString() +
            ','  + y.ToString() + ') = ' + clr.ToString();
            // рисуем цветную 'точку':
            GraphicsWindow.FillEllipse(x, y, 2 * radius, 2 * radius);
        end
    end;

  end; //end of class
end.
```

На самом деле мы, конечно, схитрили и нарисовали не точки, а **окрашенные эллипсы** (а точнее - кружочки). Для этого нам понабился метод:

FillEllipse(**x** : double; **y** : double; **width** : double; **height** : double);

- Первая пара параметров – это **координаты** верхнего левого угла описанного прямоугольника.
- Вторая пара – **ширина** и **высота** *эллипса* (Рис. 4.10).
- **Цвет** эллипса определяется свойством *BrushColor*.

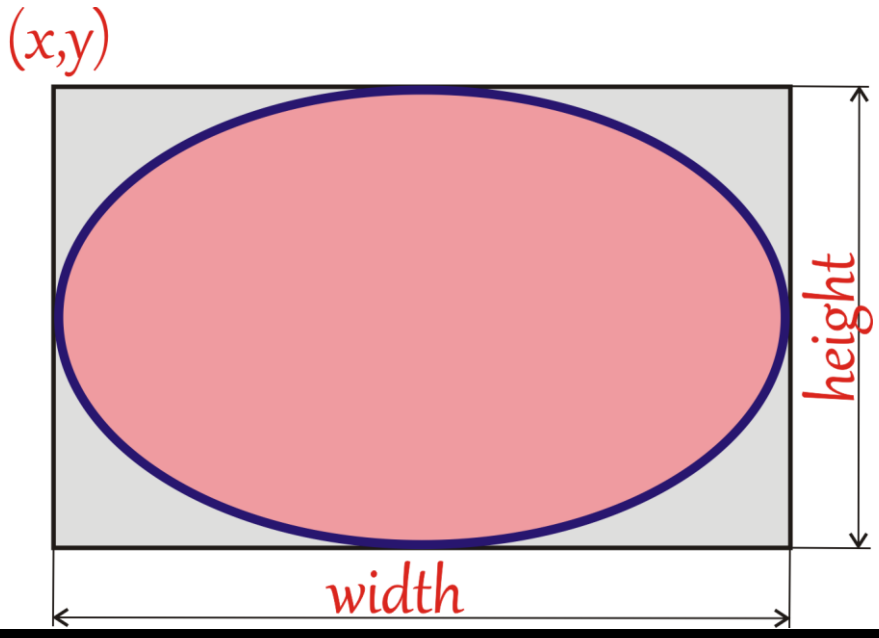


Рис. 4.10. Закрашенный эллипс

Для рисования **незакрашенного эллипса** нам пригодится метод:

DrawEllipse(**x** : double; **y** : double; **width** : double; **height** : double);

Так как он внутри пустой, то есть имеет цвет фона, то необходимо выделить *контур* эллипса, иначе мы его вообще не увидим. Цвет контура определяется значением свойства **PenColor**. Немного изменим код:

```
// рисуем цветную 'точку':
//GraphicsWindow.FillEllipse(x, y, 2 * radius, 2 * radius);
GraphicsWindow.PenColor := clr;
GraphicsWindow.DrawEllipse(x, y, 2 * radius, 2 * radius);
```

И точки-кружочки превращаются в элегантные **окружности** (Рис. 4.11).

Поскольку точки одного и того же размера вгоняют в тоску, то давайте усеем канву точками всевозможного калибра (Рис. 4.12). Сделать это проще простого. Достаточно радиус кружков задавать *случайным* образом:

```
for var i := 0 to 10000 do
begin
   radius := rand.Next(10)+4;
```

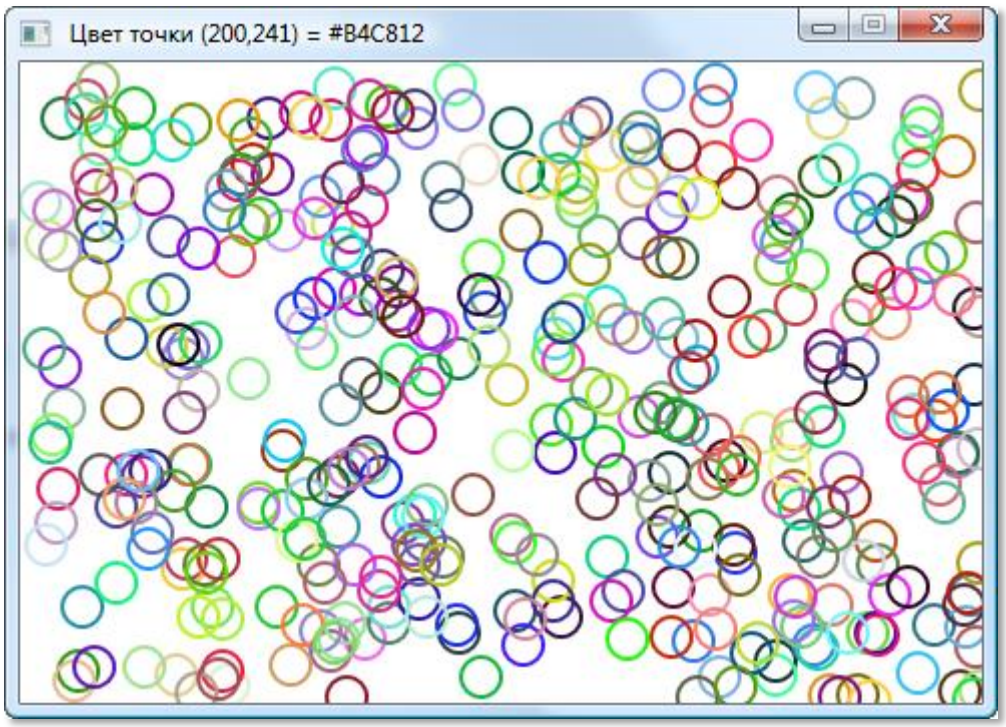


Рис. 4.11. Цветные колечки

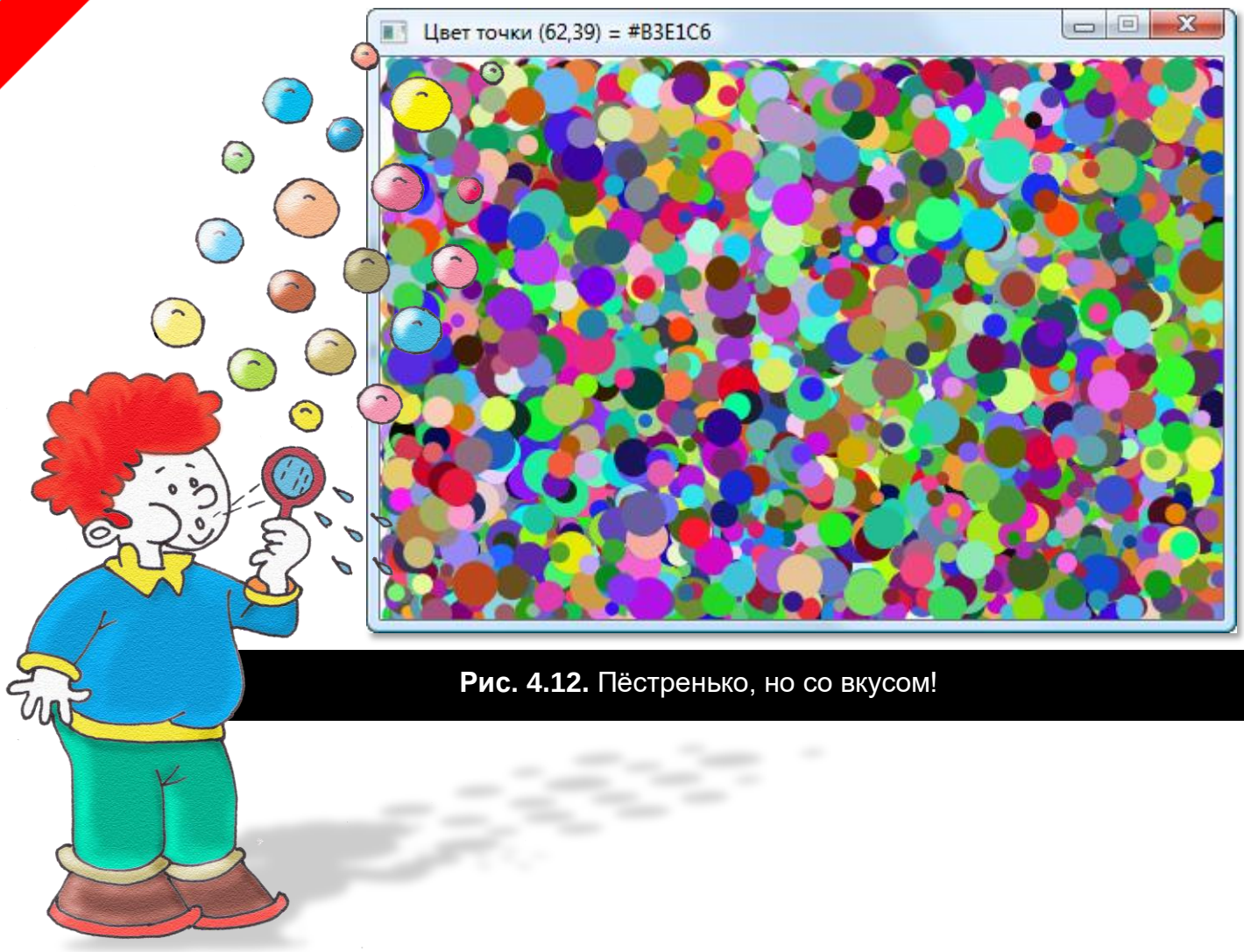


Рис. 4.12. Пёстренько, но со вкусом!

Глава 5. Занимательные игры с пикселями

Случайные точки, которыми мы «баловались» в предыдущей главе, создают **хаотичный** «узор», поэтому теперь мы будем окрашивать пиксели по **строгим** математическим формулам. Они могут быть и довольно простыми, но узоры при этом давать великолепные!

Проект *Синусоидные полоски*

Исходный код программы находится в папке **Синусоидные полоски**.

Начните новый проект и сохраните его в папке **Синусоидные полоски**. Рисунок здесь будут создавать горизонтальные строки одинаково окрашенных пикселей (проще говоря, линии, которые мы рисуем отдельными точками). Цвет пикселей изменяется по высоте клиентской области окна сверху вниз по совсем простой **формуле**:

```
var clrpx := 255 * (1 + System.Math.Sin(y / wl)) / 2;          (1)
```

Поскольку в формуле присутствует **синус** угла, то проект так и называется - *Синусоидные полоски*. Частота горизонтальных волн зависит от длины волны **wl**, так что вы легко получите разные картинки, изменяя значение этого параметра. И вам осталось узнать только о назначении множителя 255 в формуле (1). Дело в том, что выражение в скобках изменяется в диапазоне 0..2, а делённое на два – в диапазоне 0..1. Цветные составляющие пикселя должны иметь значение от 0 до 255, откуда и вытекает необходимость введения в формулу этого множителя. Цвет очередного пикселя печатается в заголовке окна. Если вам эта информация не нужна, закомментируйте строчку:

```
GraphicsWindow.Title := 'Цвет пикселя (' + x.ToString() + ',' +
                        y.ToString() + ') = ' + clr.ToString();
```

Остальная часть кода не должна вызвать у вас никаких проблем:

```
{$apptype windows}

// Синусоидные полоски

uses
   DrawUnit;

begin
   Draw.Prepare();
   Draw.Draw();
end.

unit DrawUnit;

uses
   Microsoft.SmallBasic.Library, System;

type
   Draw = class
      class procedure Prepare();
      begin
         GraphicsWindow.Title := 'Синусоидные полоски';
         GraphicsWindow.Width := 320;
         GraphicsWindow.Height := 240;
         GraphicsWindow.Left := (Desktop.Width –
                                  GraphicsWindow.Width) / 2;
         GraphicsWindow.Top := (Desktop.Height –
                                 GraphicsWindow.Height) / 2;
         GraphicsWindow.CanResize := false;
      end;

      class procedure Draw();
      begin
         var height: integer := GraphicsWindow.Height;
         var width: integer := GraphicsWindow.Width;
         // длина волны синусоиды:
         var wl := 10.0;
         for  var y := 0 to height - 1 do
         begin
            // цвет очередного пикселя:
            var clrpx := 255 * (1 + System.Math.Sin(y / wl)) / 2;
            var clr := GraphicsWindow.GetColorFromRGB(clrpx,
                                                      clrpx, clrpx);
            for  var x := 0 to width - 1 do
            begin
               GraphicsWindow.Title := 'Цвет пикселя (' + x.ToString() +
                           ',' + y.ToString() + ') = ' + clr.ToString();
               // окрашиваем его:
               GraphicsWindow.SetPixel(x, y, clr);
            end;
         end;
```

```
      end;
   end; //end of class
end.
```

Картинка получилась славная, но **чёрно**-**белая** (Рис. 5.1).

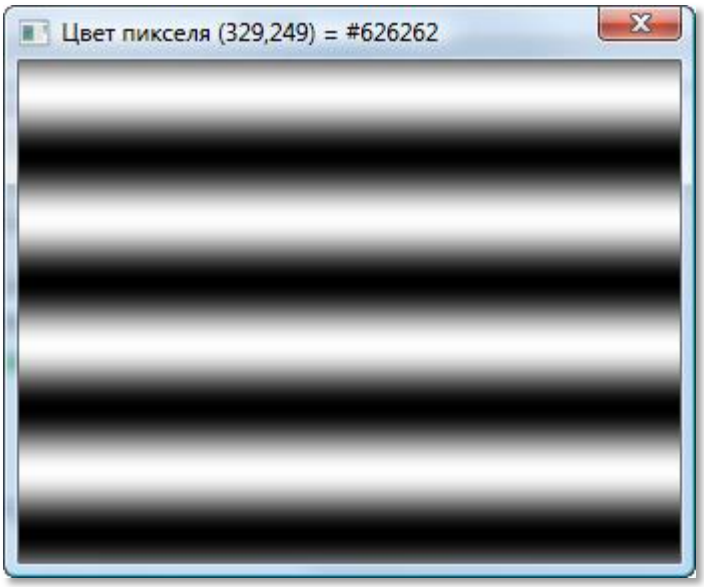


Рис. 5.1. Синусоидные полоски

Но вы без труда окрасите волны в нужный цвет, слегка изменив параметры в методе *GetColorFromRGB* (Рис. 5.2 и 5.3).

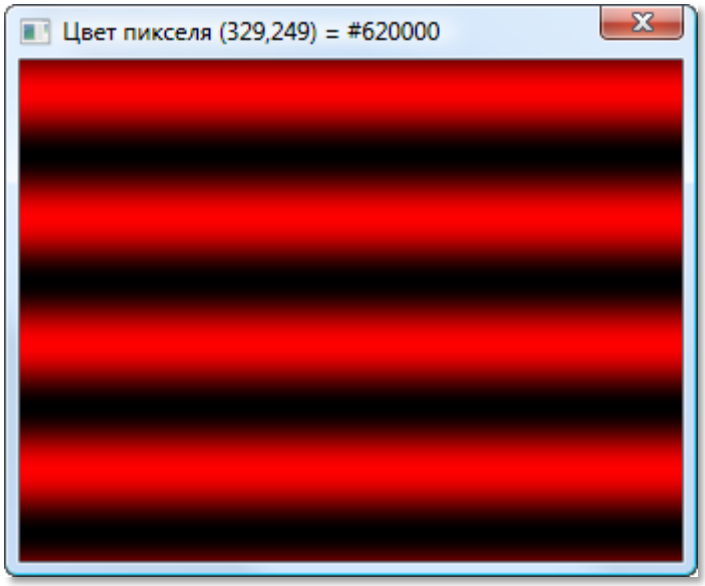


Рис. 5.2. GetColorFromRGB(clrpx,0,0)

Рис. 5.3. GetColorFromRGB(0,0,clrpx)

Цветные волны ещё лучше!

Проект *Двойные волны*

Исходный код программы находится в папке **Двойные волны**.

А теперь давайте пустим **две** волны – вертикальную и горизонтальную:

```
//длина горизонтальной волны:
var wX := 20.0;
//длина вертикальной волны:
var wY := 20.0;
```

Для этого в формулу для вычисления цвета пикселя добавим ещё один синус:

```
var clrpx := 255 * (1 + System.Math.Sin(x / wX) *
                        System.Math.Sin(y / wY)) / 2;
```

Остальная часть программы почти не отличается от предыдущей:

```
{$apptype windows}

// Двойная волна

uses
   DrawUnit;

begin
   Draw.Prepare();
   Draw.Draw();
end.

unit DrawUnit;
uses
   Microsoft.SmallBasic.Library, System;

type
   Draw = class
      class procedure Prepare();
      begin
         GraphicsWindow.Title := 'Двойная волна';
         GraphicsWindow.Width := 320;
         GraphicsWindow.Height := 240;
         GraphicsWindow.Left := (Desktop.Width -
                                 GraphicsWindow.Width) / 2;
```

```
        GraphicsWindow.Top := (Desktop.Height –
                               GraphicsWindow.Height) / 2;
        GraphicsWindow.CanResize := false;
    end;

    class procedure Draw();
    begin
        var height: integer := GraphicsWindow.Height;
        var width: integer := GraphicsWindow.Width;

        // длина горизонтальной волны:
        var wX := 20.0;
       // длина вертикальной волны:
        var wY := 20.0;
        for  var y := 0 to height - 1 do
            for  var x := 0 to width - 1 do
            begin
                // цвет очередного пикселя:
                var clrpx := 255 * (1 + System.Math.Sin(x / wX) *
                                    System.Math.Sin(y / wY)) / 2;
                var clr := GraphicsWindow.GetColorFromRGB(0, clrpx, 100);
                GraphicsWindow.Title := 'Цвет пикселя (' +
                          x.ToString() + ',' + y.ToString() + ') = ' +
                          clr.ToString();
                // окрашиваем его:
                GraphicsWindow.SetPixel(x, y, clr);
            end;
    end;
  end; //end of class
end.
```

Получилась интересная *сетчатая* структура (Рис. 5.4).

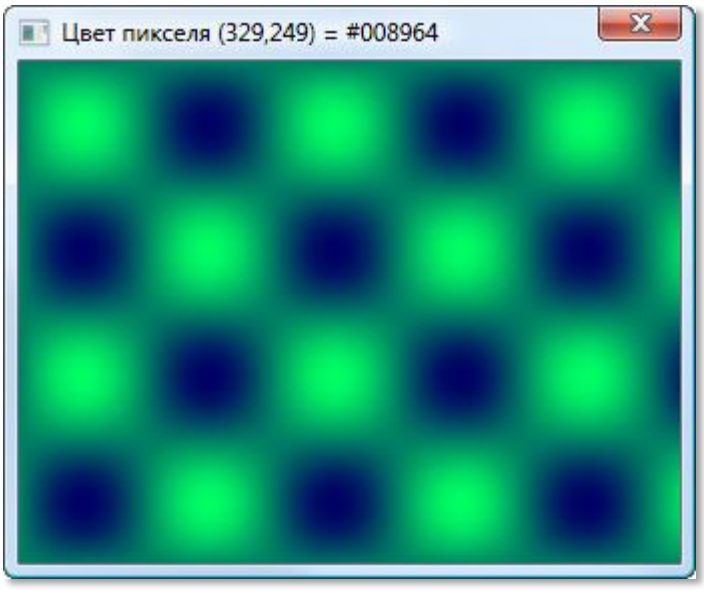


Рис. 5.4. Двойные волны

Проект *Лунки*

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

В следующем проекте мы так изменим формулу для вычисления цвета пикселей, чтобы в ней участвовала **абсолютная величина** синусов:

```
var clrpx := 255 * (1 + System.Math.Abs(System.Math.Sin(x / wX)) *
                    System.Math.Abs(System.Math.Sin(y / wY))) / 2;
```

Все лунки приобретут *одинаковый* цвет, в отличие от проекта *Двойные волны*, где лунки окрашивались в *разные* цвета (Рис. 5.5):

```
{$apptype windows}

// Лунки

uses
   DrawUnit;

begin
   Draw.Prepare();
   Draw.Draw();
end.

      . . .
               var clr := GraphicsWindow.GetColorFromRGB(clrpx, 0,
                                                         clrpx);
               GraphicsWindow.Title := 'Цвет пикселя (' +
                                       x.ToString() + ',' +
                                       y.ToString() + ') = ' +
                                       clr.ToString();
               // окрашиваем его:
               GraphicsWindow.SetPixel(x, y, clr);
            end;
      end;
   end; //end of class
```

```
end.
```

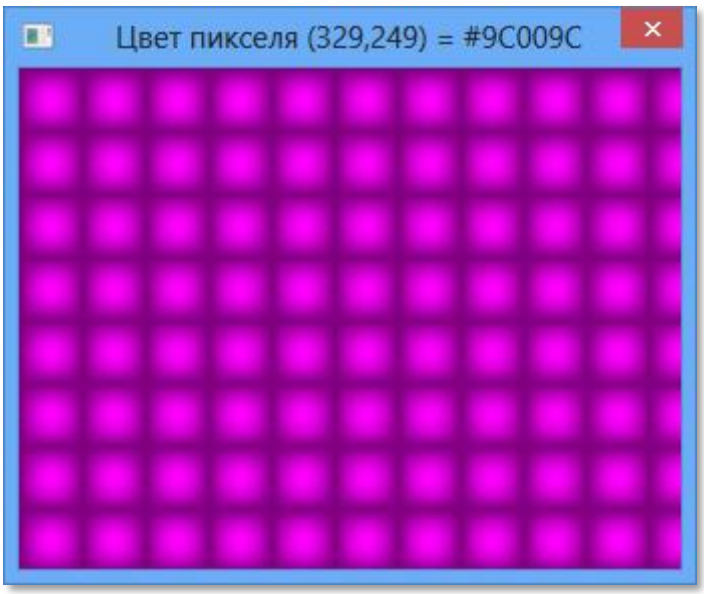


Рис. 5.5. Лунки

Проект *Радиальные волны*

Исходный код программы находится в папке **Радиальные волны**.

Бросая в воду камешки,
смотри на круги,
ими образуемые; иначе та-
кое бросание
будет пустою забавою.

Козьма Прутков

Если вы бросали камни в воду, то, конечно, обратили внимание на то, что волны, которые разбегаются от места падения камня, имеют вовсе не форму прямых линий или лунок - они совершенно *круглые*. Чтобы загнуть волны в дугу, нам придётся перейти от прямоугольных координат к **полярным**, что мы и сделаем с помощью формулы:

```
var rad := System.Math.Sqrt((x - CX) * (x - CX) +
                            (y - CY) * (y - CY)) / wl;
```

Сама же формула для цвета пикселя останется без изменений:

```
var clrpx := 255 * (1 + System.Math.Sin(rad)) / 2;
```

Исходный код программы очень похож на наши прежние проекты:

```
{$apptype windows}

// Радиальные волны

uses
   DrawUnit;

begin
   Draw.Prepare();
```

```
   Draw.Draw();
end.

unit DrawUnit;

uses
   Microsoft.SmallBasic.Library, System;

type
   Draw = class

      class procedure Prepare();
      begin
         GraphicsWindow.Title := 'Радиальные волны';
         GraphicsWindow.Width := 320;
         GraphicsWindow.Height := 320;
         GraphicsWindow.Left := (Desktop.Width –
                                  GraphicsWindow.Width) / 2;
         GraphicsWindow.Top := (Desktop.Height –
                                 GraphicsWindow.Height) / 2;
         GraphicsWindow.CanResize := false;
      end;

      class procedure Draw();
      begin
         var height: integer := GraphicsWindow.Height;
         var width: integer := GraphicsWindow.Width;
         // координаты центра волн:
         var CX := width div 2;
         var CY := height div 2;

         // длина волны:
         var wl := 6.0;

         for  var y := 0 to height - 1 do
            for  var x := 0 to width - 1 do
            begin
               // цвет очередного пикселя:
               var rad := System.Math.Sqrt((x - CX) * (x - CX) +
                                           (y - CY) * (y - CY)) / wl;
               var clrpx := 255 * (1 + System.Math.Sin(rad)) / 2;
               var clr := GraphicsWindow.GetColorFromRGB(0, 0, clrpx);

               GraphicsWindow.Title := 'Цвет пикселя (' + x.ToString() +
                           ',' + y.ToString() + ') = ' + clr.ToString();
               // окрашиваем его:
               GraphicsWindow.SetPixel(x, y, clr);
            end;

      end;
   end; //end of class
```

```
end.
```

Зато волны получились – как настоящие (Рис. 5.6)!

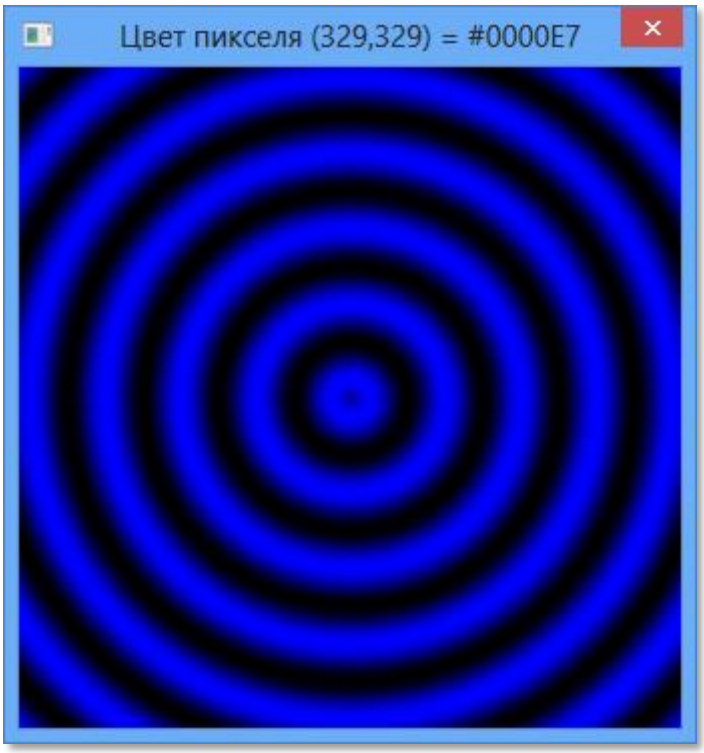


Рис. 5.6. «Полярные» волны!

Проект *Ромбы*

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

Применив для пикселестроения более хитроумную формулу, мы получим великолепный **ромбический узор**, от которого глаза трудно оторвать (Рис. 5.7).

```
{$apptype windows}

// Ромбы

uses
   DrawUnit;

begin
   Draw.Title();
   Draw.Draw();
end.

unit DrawUnit;

uses
   Microsoft.SmallBasic.Library, System;

type
```

```
   Draw = class

      class procedure Title();
      begin
         GraphicsWindow.Title := 'Ромбы';
         GraphicsWindow.Width := 320;
         GraphicsWindow.Height := 320;
         GraphicsWindow.Left := (Desktop.Width –
                                 GraphicsWindow.Width) / 2;
         GraphicsWindow.Top := (Desktop.Height –
                                GraphicsWindow.Height) / 2;
         GraphicsWindow.CanResize := false;
      end;

      class procedure Draw();
      begin
         var height: integer := GraphicsWindow.Height;
         var width: integer := GraphicsWindow.Width;
         // длина волны:
         var wl := 60.0;

         for  var y := 0 to height - 1 do
            for  var x := 0 to width - 1 do
            begin
               // цвет очередного пикселя:
               var clrpx := (System.Math.Abs(x mod
                            Convert.ToInt32(wl) - wl / 2) +
                            System.Math.Abs(y mod
                            Convert.ToInt32(wl) - wl / 2));
               clrpx *= 255 * 2 / wl;
               var clr := GraphicsWindow.GetColorFromRGB(clrpx, 0,
                                                         clrpx);
               GraphicsWindow.Title := 'Цвет пикселя (' +
                                       x.ToString() + ',' +
                                       y.ToString() + ') = ' +
                                       clr.ToString();
               // окрашиваем его:
               GraphicsWindow.SetPixel(x, y, clr);
            end;
      end;
   end; //end of class

end.
```

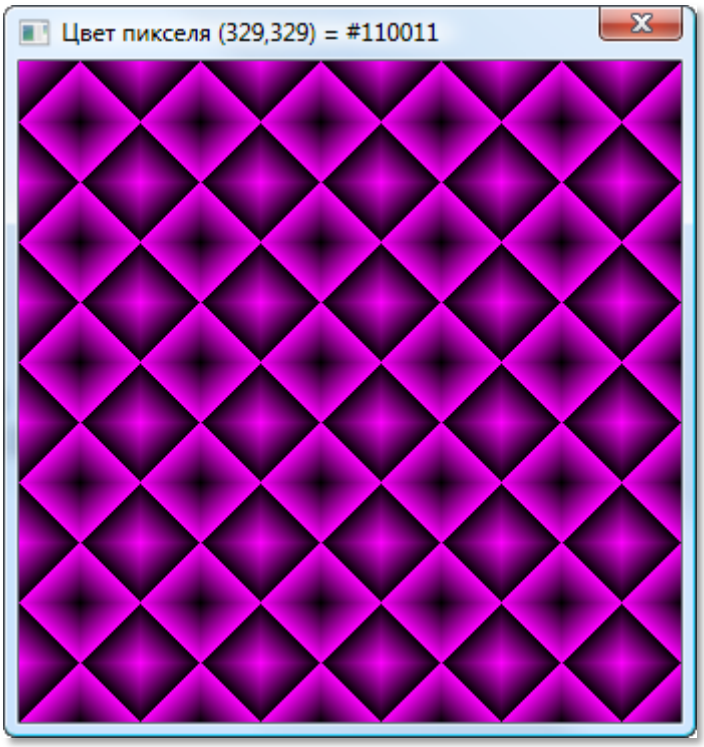


Рис. 5.7. Канва в ромбик

Проект *Синусоиды Винни-Пуха*

Исходный код программы находится в папке **Синусоиды Винни-Пуха**.

По никому неведомой причине неизвестный автор этих синусоид (Рис. 5.8) посвятил их нашему любимому Винни, прославившемуся своим пыхтеньем (следствие непомерного обжорства).

В этом примере 9 **параметров**, входящих в формулу для вычисления цвета пикселя, выбираются случайно в начале программы:

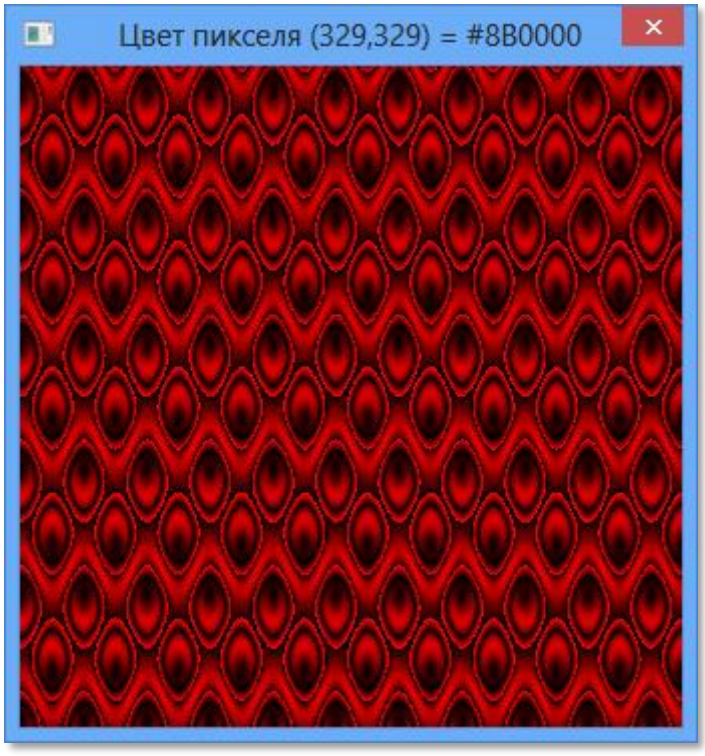
```
{$apptype windows}

// Синусоиды Винни-Пуха

uses
   DrawUnit;

begin
   Draw.Prepare();
   Draw.Draw();
end.

unit DrawUnit;

uses
   Microsoft.SmallBasic.Library, System;

type
   Draw = class

      class procedure Prepare();
      begin
         GraphicsWindow.Title := 'Синусоиды Винни-Пуха';
         GraphicsWindow.Width := 320;
         GraphicsWindow.Height := 320;
         GraphicsWindow.Left := (Desktop.Width –
                                 GraphicsWindow.Width) / 2;
         GraphicsWindow.Top := (Desktop.Height –
                                GraphicsWindow.Height) / 2;
         GraphicsWindow.CanResize := false;
      end;

      class procedure Draw();
      begin
         var height: integer := GraphicsWindow.Height;
         var width: integer := GraphicsWindow.Width;
         // длина волны:
         var wX := 10.0;
         var wY := 10.0;
         var rand := new Random();
         var a := new int32[10];
         for var i := 1 to 10 - 1 do
            a[i] := rand.Next(2) + 1;
```

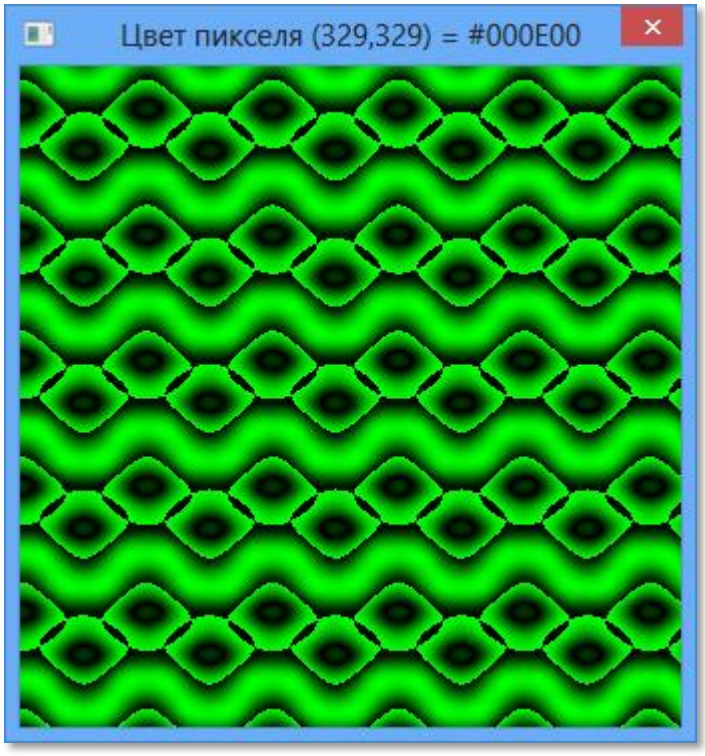
Это значит, что при каждом новом запуске программы вы будете получать **свежую** картинку. Вот где простор для поиска новых узоров!

С другой стороны, случайный поиск не очень эффективен. Может быть, ст**О**ит попробовать подбирать коэффициенты вручную?

Формула для вычисления цвета пикселей просто ужасная, но компьютер мы ею не напугаем:

```
      for  var y := 0 to height - 1 do
           for  var x := 0 to width - 1 do
           begin
              // цвет очередного пикселя:
              var clrpx := 256 * (1 +
                  (a[4] * System.Math.Sin(a[0] * System.Math.Sin(a[6] *
                   x / wX) + a[1] * System.Math.Cos(a[7] * y / wY)) +
                a[5] * System.Math.Cos(a[2] * System.Math.Cos(a[8] * x / wX)
                   + a[3] * System.Math.Sin(a[9] * y / wY)))) / 2;

              var clr := GraphicsWindow.GetColorFromRGB(0, 0, clrpx);
               GraphicsWindow.Title := 'Цвет пикселя (' + x.ToString() +
                        ',' + y.ToString() + ') = ' + clr.ToString();
              // окрашиваем его:
              GraphicsWindow.SetPixel(x, y, clr);
           end;
     end;
  end; //end of class
end.
```





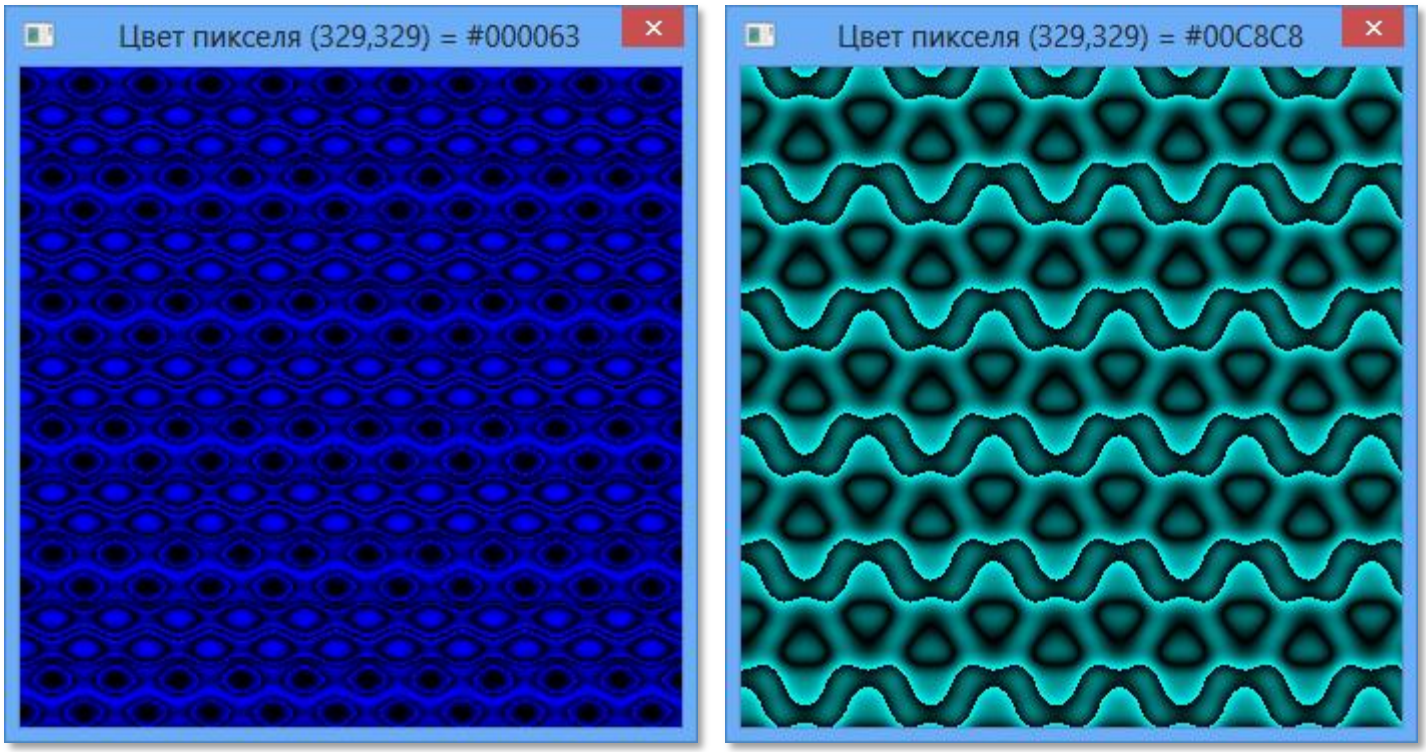


Рис. 5.8. Так как параметры для вычисления цвета пикселей выбираются случайно, то каждый раз вы будете получать другую картинку!

Проект *Туманность*

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

Аналогично, выбирая **случайные параметры** для формулы в этом примере, вы получите причудливые линии, похожие на космические туманности (Рис. 5.9):

```
{$apptype windows}

// Туманность

uses
   DrawUnit;

begin
   Draw.Prepare();
   Draw.Draw();
end.

unit DrawUnit;

uses
   Microsoft.SmallBasic.Library, System;

type
   Draw = class
```

```
      class procedure Prepare();
      begin
         GraphicsWindow.Title := 'Туманность';
         GraphicsWindow.Width := 320;
         GraphicsWindow.Height := 320;
         GraphicsWindow.Left := (Desktop.Width –
                                 GraphicsWindow.Width) / 2;
         GraphicsWindow.Top := (Desktop.Height –
                                 GraphicsWindow.Height) / 2;
         GraphicsWindow.CanResize := false;
         GraphicsWindow.BackgroundColor := 'Black';
      end;

      class procedure Draw();
      begin
         var height := (int32)(GraphicsWindow.Height);
         var width := int32(GraphicsWindow.Width);

         // координаты центра:
         var CX := width div 2;
         var CY := height div 2;

         var rand := new Random();
         var a := new double[7];
         for var i := 1 to 7 - 1 do
            a[i] := System.Math.PI * (1 - 2.0 * rand.Next(1000) / 1000);

         while (true) do
         begin
            var x := System.Math.Sin(a[1] * a[6]) -
                              System.Math.Cos(a[2] * a[5]);
            var y := System.Math.Sin(a[3] * a[5]) -
                              System.Math.Cos(a[4] * a[6]);
            var clr := GraphicsWindow.GetColorFromRGB(128 * (x + y),
                                                      128 * (x + y),
                                                      255);
            GraphicsWindow.SetPixel(CX*2 + x * 165, CY*2 + y * 165, clr);
            a[5] := x;
            a[6] := y;
         end;

      end;
   end; //end of class

end.
```

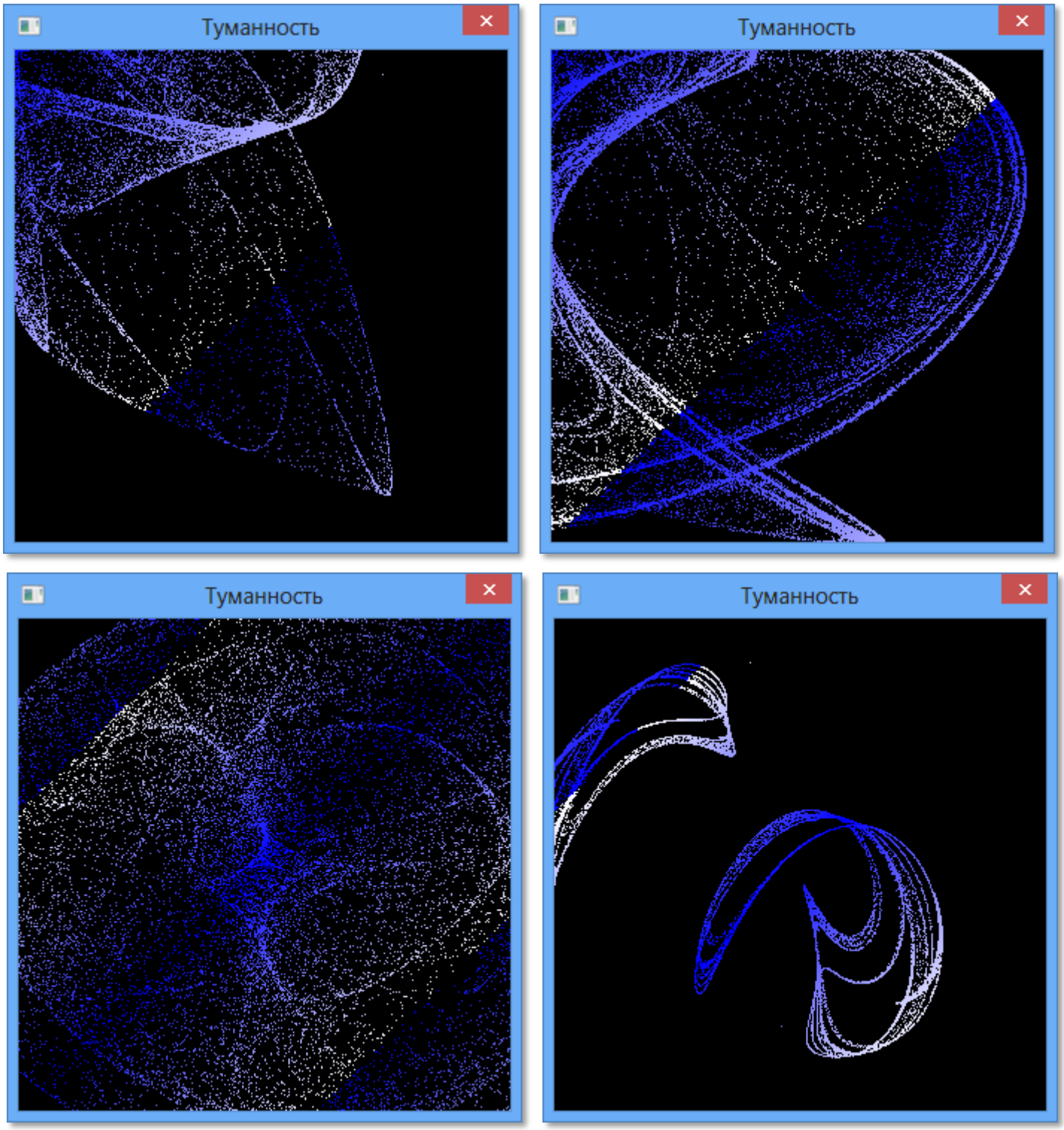


Рис. 5.9. Туманности

Глава 6. Элементы управления

Без **элементов управления** (*ЭУ*) очень трудно изменять параметры программы и её поведение. Конечно, у нас в руках – мышка, а под рукою - клавиатура, но кнопки всё равно удобнее! А метки или текстовые поля и вовсе никакими клавишами и кнопками не заменишь.

Все элементы управления создаются с помощью методов класса **Controls**. Например, любое приложение *Windows* должно иметь хотя бы одну *кнопку*, чтобы запускать и останавливать выполнение программы. Давайте создадим приложение с **кнопкой**.

Для этого мы воспользуемся методом:

```
Controls.AddButton(Caption : string; left integer; top : integer) : string;
```

Параметр **Caption** – это надпись на кнопке, а остальные два параметра задают положение кнопки в клиентской области окна.

Проект *Элементы управления*

Исходный код программы находится в папке **Элементы управления**.

Начнём нашу новую программу с того, что добавим **кнопку** (Рис. 6.1):

```
{$apptype windows}

// Элементы управления

uses
   DrawUnit;

begin
   var draw := new Draw();
   draw.Prepare();
   draw.Draw();
end.

unit DrawUnit;
```

```
uses
   Microsoft.SmallBasic.Library, System;

type
   Draw = class
      procedure Prepare();
      begin
         GraphicsWindow.Hide();
         GraphicsWindow.Title := 'Элементы управления';
         GraphicsWindow.Width := 320;
         GraphicsWindow.Height := 240;
         GraphicsWindow.CanResize := false;
         GraphicsWindow.BackgroundColor := 'White';
         GraphicsWindow.Show();
         GraphicsWindow.Left := (Desktop.Width –
                                 GraphicsWindow.Width) / 2;
         GraphicsWindow.Top := (Desktop.Height –
                                GraphicsWindow.Height) / 2;
      end;

      procedure Draw();
      begin
         var height := int32(GraphicsWindow.Height);
         var width := int32(GraphicsWindow.Width);

         var btnTimer := Controls.AddButton('Запустить таймер',
                                            10, height - 48);
      end;

   end; //end of class
end.
```

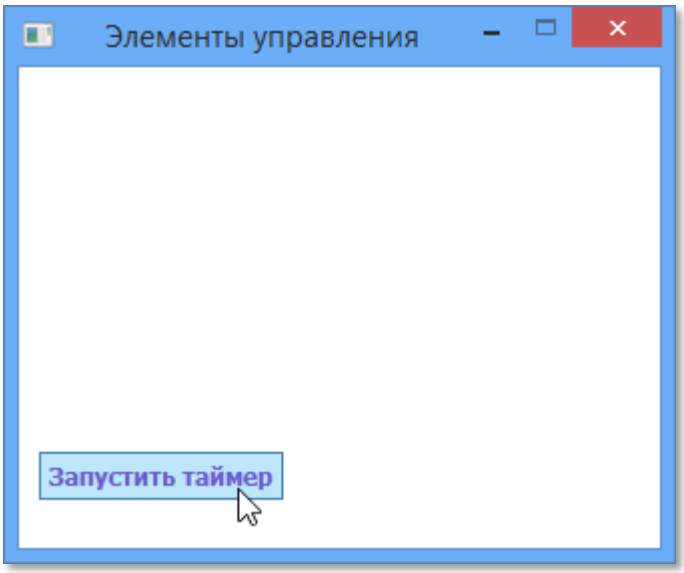


Рис. 6.1. Кнопка прикноплена!

Обратите внимание на то, что кнопке следует дать **имя**, по которому мы сможем потом к ней обращаться. Чтобы отличать названия кнопок от других идентификаторов программы, мы будем начинать их названия с префикса *btn*. Это сокращение от английского слова *button*, что и означает *кнопка*.

Размеры кнопки по умолчанию будут такими, чтобы надпись поместилась на ней целиком. Если же вы захотите изменить размеры кнопки или любого другого элемента управления, то укажите *имя* первым в методе:

Controls. SetSize (**control** : string; **width** : integer; **height** : integer);

А затем – *ширину* и *высоту* кнопки.

Добавим к программе одну строчку:

```
Controls.SetSize(btnTimer,164,32);
```

И наша кнопка станет более солидной (Рис. 6.2).

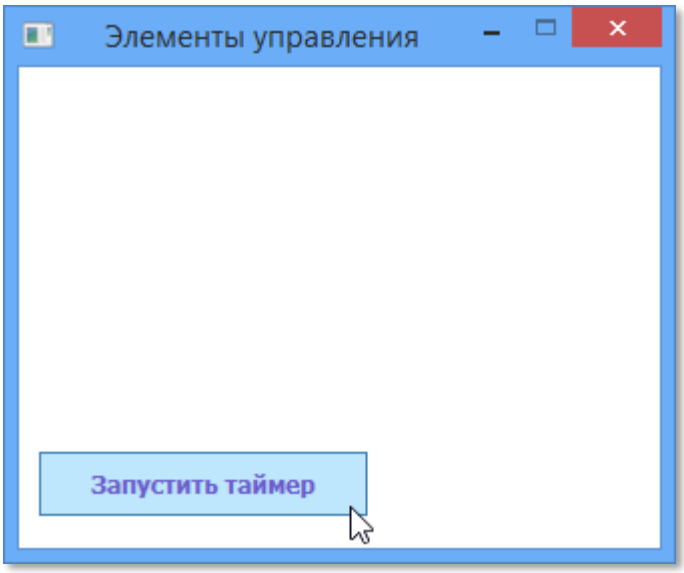


Рис. 6.2. Увеличенная кнопка

Иногда нужно **переместить** кнопку в другую позицию в окне приложения. Для этого имеется метод:

Controls. Move (**control** : string; **x** : integer; **y** : integer);

Первый параметр вам уже известен, второй и третий – новые *координаты* левого верхнего угла элемента управления.

Воспользуемся этим методом, чтобы опять **разыграть** пользователя. Заставим кнопку беспрерывно елозить по экрану (Рис. 6.3):

```
var offset := -1;
var left := 10;
var top := height - 48;

while(true ) do
begin
   for var i := 0 to 40 do
   begin
      left += offset;
      top += offset;
      Controls.Move(btnTimer, left, top);
      Thread.Sleep(TimeSpan.FromMilliseconds(10));
   end;
   offset *= -1;
end
```

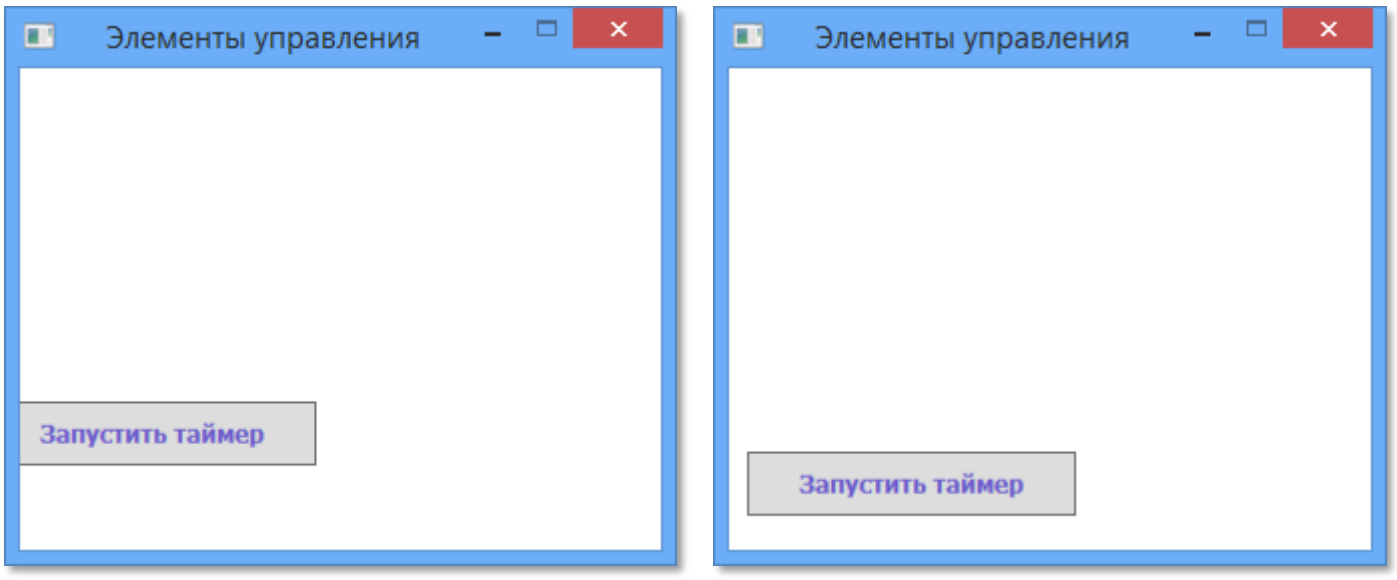


Рис. 6.3. Беспокойная кнопка

Вы уже, наверное, пробовали нажать кнопку и были раздосадованы, что таймер не запускается, несмотря на обещание, написанное на кнопке. Правильно, обещания нужно выполнять, поэтому добавим к программе метод-обработчик события *ButtonClicked*, которое возникает при нажатии на кнопку:

```
btnTimer := Controls.AddButton('Запустить таймер', 10, height - 48);
Controls.SetSize(btnTimer, 164, 32);
Controls.ButtonClicked += OnClick;
```

В этом случае название кнопки должно иметь строковый тип, а объявить кнопку нужно как поле класса!

```
private procedure OnClick();
begin
   Controls.SetButtonCaption(btnTimer, 'Остановить таймер');
end;
```

С помощью метода

Controls. SetButtonCaption(**buttonName** : string; **Caption** : string);

вы легко **измените надпись** на кнопке, чтобы она соответствовала новому состоянию программы (Рис. 6.4).

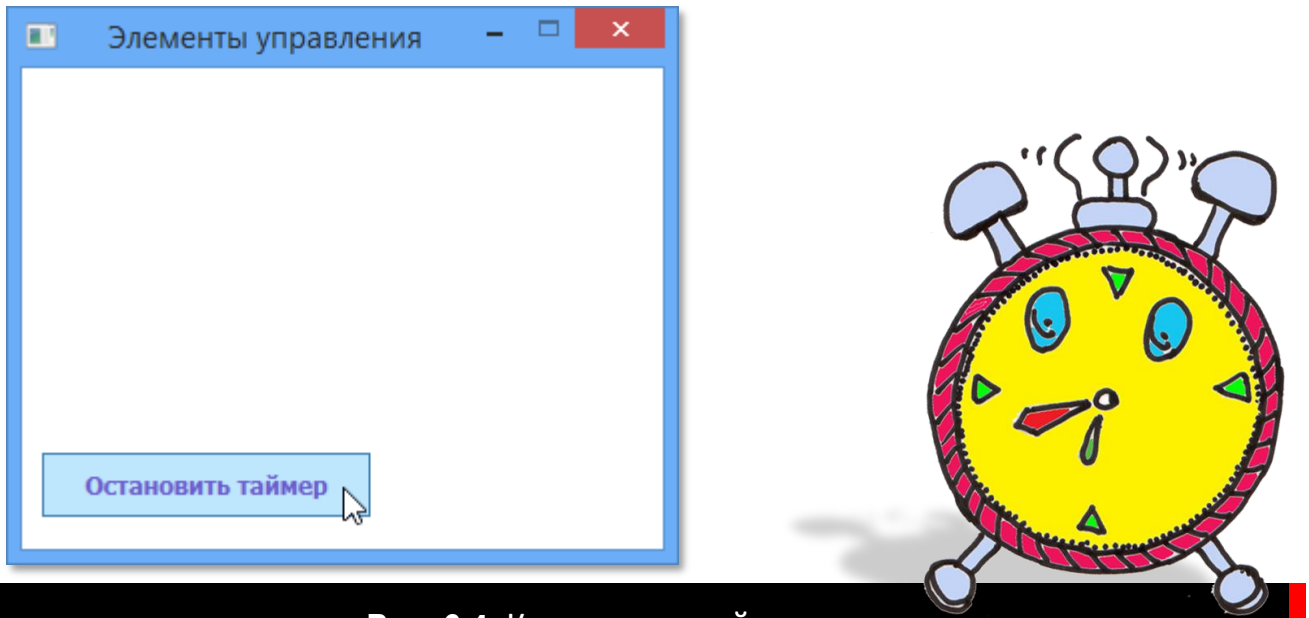


Рис. 6.4. Кнопка с новой надписью

Таким образом, нажав кнопку в первый раз, вы *запускаете* таймер, во второй раз – останавливаете его. Но вот незадача: после того как вы остановили таймер, должна вернуться первоначальная надпись, а этого не происходит. Исправляем оплошность:

```
private procedure OnClick();
begin
   if (Controls.GetButtonCaption(btnTimer).ToString() =
       'Запустить таймер') then
      Controls.SetButtonCaption(btnTimer, 'Остановить таймер')
   else
      Controls.SetButtonCaption(btnTimer, 'Запустить таймер');
end;
```

Теперь надписи на кнопке правильные. Осталось добавить **таймер**, который также является элементом управления, но *невизуальным (невидимым).*

```
Timer.Interval := 1000;
Timer.Pause();
Timer.Tick += OnTick;
time := 0;
```

Объявите переменную **time** как поле!

Свойство **Interval** отвечает за время срабатывания таймера. В данном случае мы установили 1000 миллисекунд, что равняется одной секунде. Это значит, что через каждую секунду будет происходить событие *Tick*, которое мы сможем обработать в методе *OnTick*.

Поскольку значение свойства *Interval* задаётся в *миллисекундах*, а нам нужны секунды, то придётся его поделить на 1000. Затем прошедшее после нажатия на кнопку время мы выводим в заголовке окна приложения:

```
private procedure OnTick();
begin
   time += Timer.Interval / 1000;
   GraphicsWindow.Title := ' Прошло: ' + time.ToString();
end;
```

Так как мы хотим, чтобы таймер начал отсчёт времени только после нажатия на кнопку, то сначала его нужно остановить методом **Pause**.

А вот метод-обработчик нажатия на кнопку придётся изменить:

```
private procedure OnClick();
begin
   if (Controls.GetButtonCaption(btnTimer).ToString() =
      'Запустить таймер') then
   begin
      Controls.SetButtonCaption(btnTimer, 'Остановить таймер');
      Timer.Resume();
   end
   else
   begin
      Controls.SetButtonCaption(btnTimer, 'Запустить таймер');
      Timer.Pause();
   end;
end;
```

Метод **Resume** запускает таймер после остановки его методом *Pause*. Вот теперь наш таймер работает как часы (Рис. 6.5)!

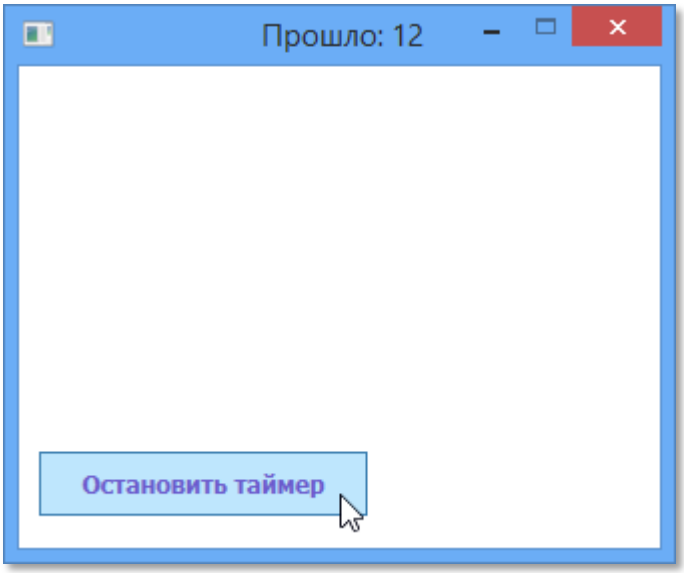


Рис. 6.5. Время пошло!

Проект *Ввод в текстовое поле*

Исходный код программы находится в папке **Ввод в текстовое поле**.

Выводить информацию, хотя бы и важную для нас, в заголовок окна вполне допустимо при отладке программы, но лучше использовать для этого специально для этого предназначенный элемент управления **текстовое поле** (или **метку**). Для того чтобы добавить *текстовое поле*, нужно вызвать метод:

Controls.AddTextBox(**left** : integer; **top** : integer) : **string**;

В скобках указывают координаты верхнего левого угла элемента управления:

```
txtTime := Controls.AddTextBox(10, 10);
```

Дополним таймерную процедуру одной строкой:

```
private procedure OnTick();
begin
   time += Timer.Interval / 1000;
   GraphicsWindow.Title := ' Прошло: ' + time.ToString();
  Controls.SetTextBoxText(txtTime,
                          ' Прошло: ' + time.ToString());
end;
```

Метод

Controls. SetTextBox(**textBoxName** : string; **text** : string);

выводит текст в указанное *текстовое поле* (Рис. 6.6).

Вы можете установить *размеры текстового поля* такими же, как и кнопки (Рис. 6.7):

```
Controls.SetSize(txtTime,164,32);
```

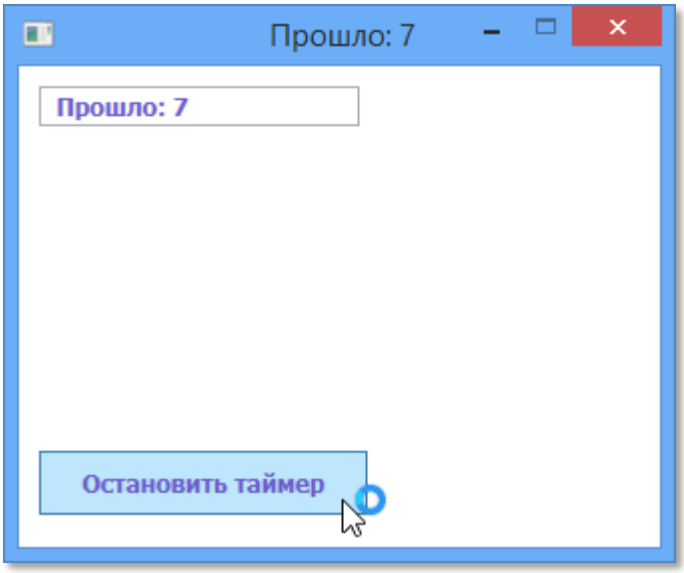


Рис. 6.6 Вот теперь совсем другое дело!

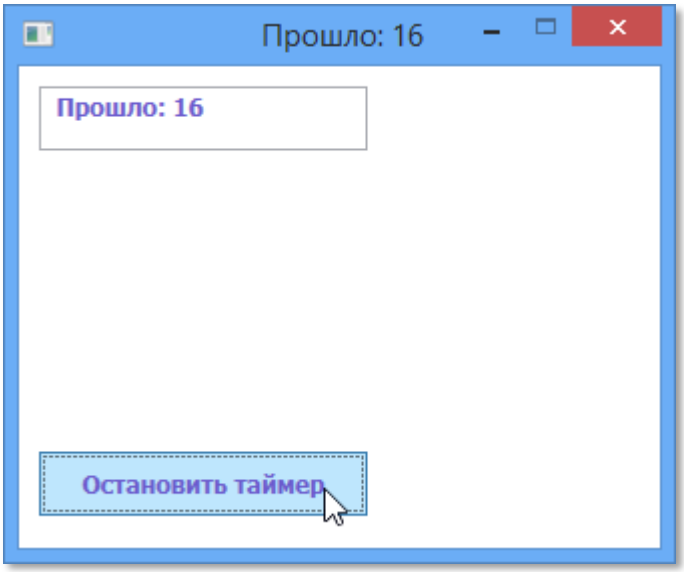


Рис. 6.7. Подровняли

В *текстовое поле* можно **вводить** *информацию* с клавиатуры (Рис. 6.8).

А это очень важно, если вы хотите **получить** от пользователя программы какой-нибудь ответ или пароль.

Сейчас мы напишем маленькую программу, показывающую, как это можно сделать.

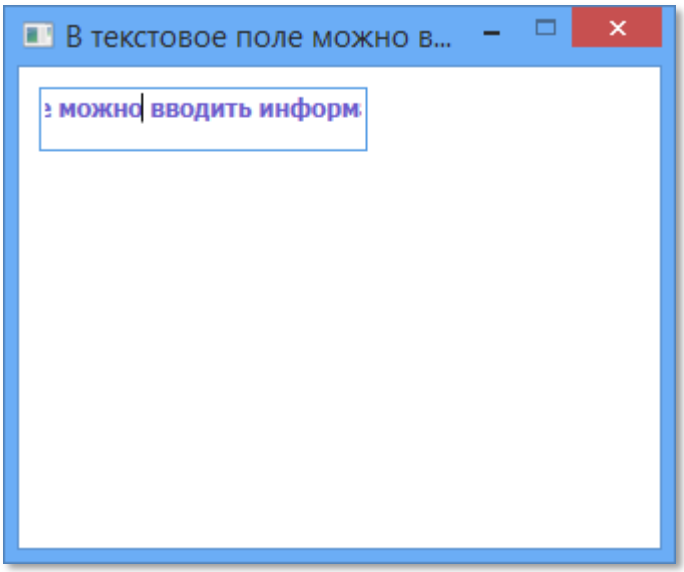


Рис. 6.8. Обратная связь с пользователем установлена!

```
{$apptype windows}
// Ввод в текстовое поле

uses
   DrawUnit;

begin
   var draw := new Draw();
   draw.Prepare();
   //draw.Draw();
   draw.Input();
end.
```

```
     private txtInfo: string;
     public procedure Prepare();
      begin
         GraphicsWindow.Hide();
         GraphicsWindow.Title := 'Ввод в текстовое поле';
         GraphicsWindow.Width := 320;
         GraphicsWindow.Height := 240;
         GraphicsWindow.CanResize := false;
         GraphicsWindow.BackgroundColor := 'White';
         GraphicsWindow.Show();
         GraphicsWindow.Left := (Desktop.Width –
                                 GraphicsWindow.Width) / 2;
         GraphicsWindow.Top := (Desktop.Height –
                                GraphicsWindow.Height) / 2;
         txtInfo := Controls.AddTextBox(10, 10);
         Controls.SetSize(txtInfo, 164, 32);
         Controls.TextTyped += OnText;
      end;
```

Чтобы ввести текст, кликните мышкой внутри элемента управления. Так вы сделаете его *активным*, то есть передадите ему *фокус ввода*. При наборе текста возникает событие *TextTyped*, которое мы обрабатываем в методе *OnText*:

```
private procedure OnText();
begin
   GraphicsWindow.Title := Controls.GetTextBoxText(txtInfo);
end;
```

Здесь с помощью метода **GetTextBoxText** мы получаем строку, записанную в *текстовом поле*, и выводим её в заголовок окна. Вы также можете присвоить это значение переменной

```
var s := Controls.GetTextBoxText(txtInfo);
```

и затем использовать, как вам заблагорассудится.

Текстовое поле довольно «умное»: вы можете выделять в нём текст, изменять его, копировать и вставлять в другое текстовое поле!

Вы, конечно, заметили, что в *текстовом поле* можно набрать строку любой длины, но при этом видна только её часть. Чтобы прочитать длинную строку, придётся использовать клавиши со стрелками, а это не очень удобно. Поэтому для этих целей лучше использовать **многострочное текстовое поле**.

Чтобы поместить этот элемент управления на форму, воспользуйтесь методом

Controls. AddMultiLineTextBox(**left** : integer; **top** : integer);

который имеет два **параметра** – координаты верхнего левого угла элемента управления:

```
txtInfo := Controls.AddMultiLineTextBox(10,10);
Controls.SetSize(txtInfo, 300, 220);
Controls.TextTyped += OnText;
```

А скопируем-ка мы в него что-нибудь из *Пушкина* (Рис. 6.9).

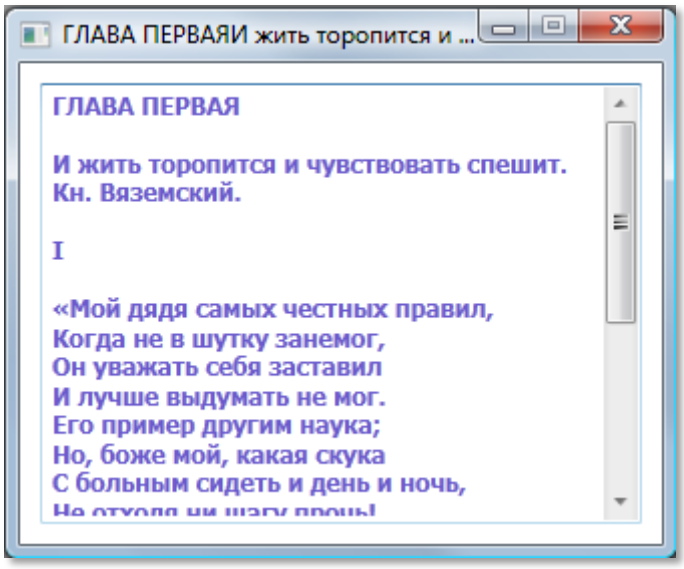


Рис. 6.9. Маловато будет!

Текст разбит на строки, но даже начало поэмы в *многострочное текстовое поле* не помещается. Зато теперь весь текст легко просмотреть, пользуясь полосами прокрутки, что гораздо удобнее, чем клавишами со стрелками (Рис. 6.10).

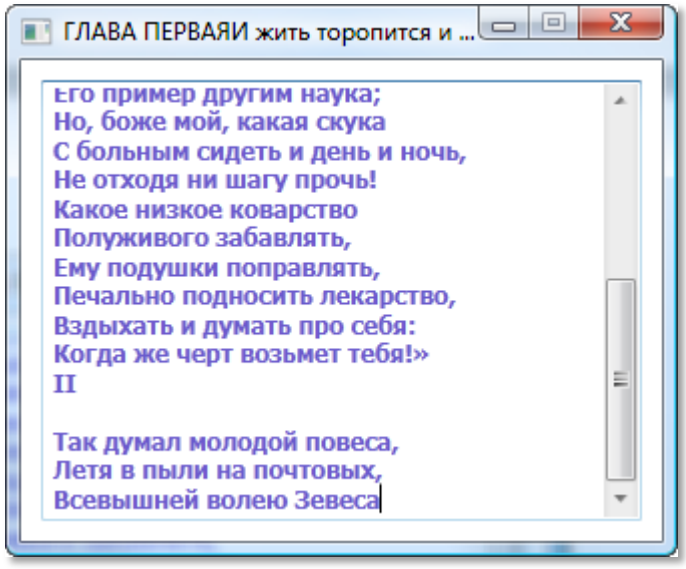


Рис. 6.10. Крути-верти

> Форматированный текст будет выведен, как обычный, поэтому забудьте о шрифтах, выравнивании и других изысках – здесь им места нет.

Проект *Дополнительные свойства и методы элементов управления*

Исходный код программы находится в папке **Дополнительные свойства**.

Заменим одну кнопку, которая отвечала за старт и остановку таймера, *двумя* (Рис. 6.11).

```
// Элементы управления

unit DrawUnit;

uses
   Microsoft.SmallBasic.Library, System, System.Threading;

type
   Draw = class

      private time: integer;
      private btnStart: string;
      private btnStop: string;
      private txtTime: string;

      public procedure Prepare();
      begin
         GraphicsWindow.Hide();
         GraphicsWindow.Title := 'Элементы управления';
         GraphicsWindow.Width := 320;
         GraphicsWindow.Height := 240;
         GraphicsWindow.CanResize := false;
         GraphicsWindow.BackgroundColor := 'White';
         GraphicsWindow.Show();
         GraphicsWindow.Left := (Desktop.Width –
                                 GraphicsWindow.Width) / 2;
         GraphicsWindow.Top := (Desktop.Height –
                                GraphicsWindow.Height) / 2;

         var height := Int32(GraphicsWindow.Height);
         var width := Int32(GraphicsWindow.Width);

         // Button1:
         btnStart := Controls.AddButton('Запустить таймер', 8,
                                        height - 32);
         // Button2:
         btnStop := Controls.AddButton('Остановить таймер', 180,
```

```
                                                height - 32);
        Controls.ButtonClicked += OnClick;

        Timer.Interval := 1000;
        Timer.Pause();
        Timer.Tick += OnTick;
        time := 0;

        txtTime := Controls.AddTextBox(10, 10);
        Controls.SetSize(txtTime, 164, 32);

    end;

  end; //end of class
end.
```

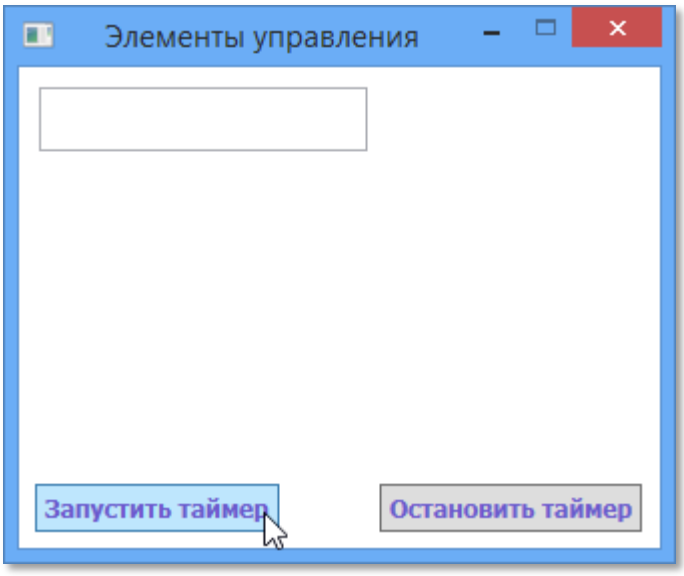


Рис. 6.11. Так удобнее?

Как говорится, одна кнопка хорошо, а две лучше. Однако правая кнопка сейчас лишняя, ведь таймер пока не запущен.

Дабы не искушать неискушенного пользователя, который обязательно нажмёт не ту кнопку, уберём правую кнопку с глаз долой (Рис. 6.12):

```
Controls.HideControl(btnStop);
```

Для этого мы использовали метод

Controls. HideControl(**controlName** : string);

которому передаётся название элемента управления.

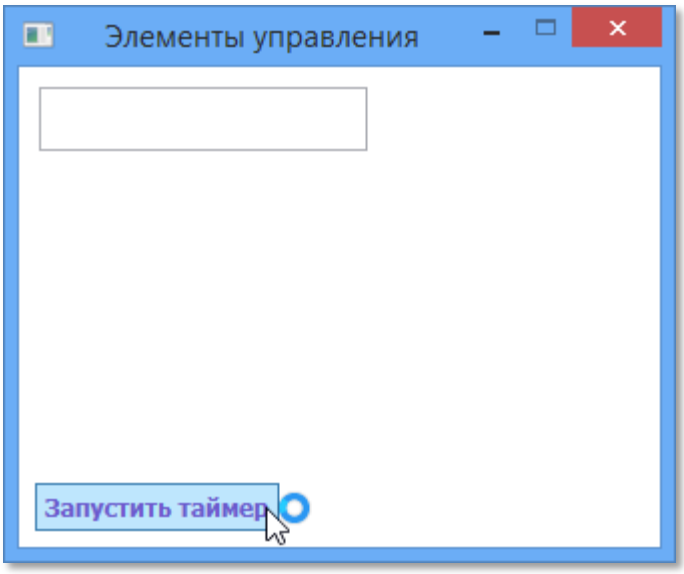


Рис. 6.12. Кнопка-невидимка

Из кода видно, что для всех кнопок создаётся один и тот же метод-обработчик:

```
Controls.ButtonClicked += OnClick;
```

Возникает вопрос: как же мы узнаем, какая из двух кнопок была нажата? – Так как на кнопках *разные* надписи, то с помощью метода **GetButtonCaption** мы можем установить, что написано на кнопке, и так распознать её. Но мы поступим иначе. Свойство **LastClickedButton** содержит **название** последней нажатой кнопки. Зная это, перепишем обработчик для таймера:

```
private procedure OnClick();
begin
   var btn := string(Controls.LastClickedButton);
   if (btn = btnStart) then
   begin
      Controls.HideControl(btnStart);
      Controls.ShowControl(btnStop);
      Timer.Resume();
   end
   else
   begin
      Controls.HideControl(btnStop);
      Controls.ShowControl(btnStart);
```

```
      Timer.Pause();
   end
end;
```

В нём не только включается и выключается таймер, но и скрывается ненужная кнопка (Рис. 6.13).

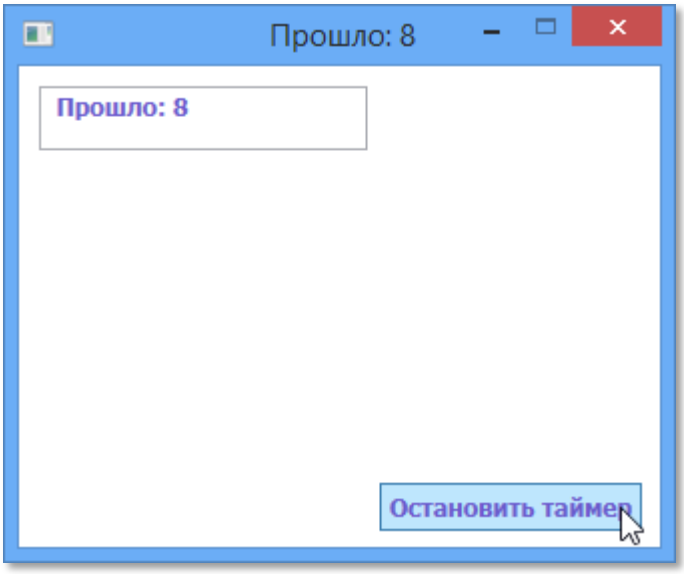


Рис. 6.13. Вторая кнопка-невидимка

Если на форме места для двух кнопок не хватает, их можно поместить одну над другой:

```
btnStop := Controls.AddButton('Остановить таймер', 8, height-32);
```

Свойство **LastTypedTextBox** действует аналогично *LastClickedButton*, но при вводе в *текстовое поле*.

И последний метод

Controls. Remove(**controlName** : string);

уничтожает ненужный элемент управления в работающем приложении и восстановлению он не подлежит.

Глава 7. Занимательная прямолинейность

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*

Одна точка - унылое зрелище. А вот через *две* точки уже можно провести прямую, поэтому попробуем теперь порисовать **линиями**.

Для этого в классе *GraphicsWindow* припасён метод:

```
DrawLine(x1 : double; y1 : double : x2 : double; y2 : double);
```

Точка с координатами *(x1, y1)* задаёт **начало** прямой, а точка *(x2, y2)* – её **конец**, то есть будет правильнее сказать, что этот метод вычерчивает **отрезок** прямой (Рис. 7.1).

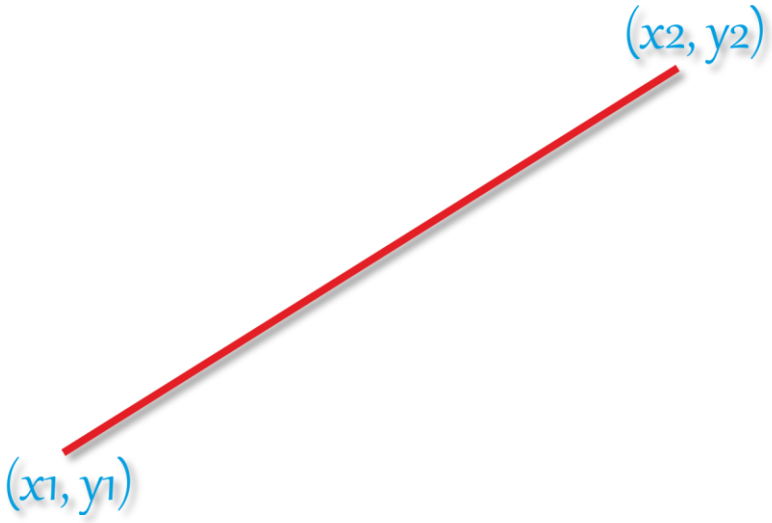


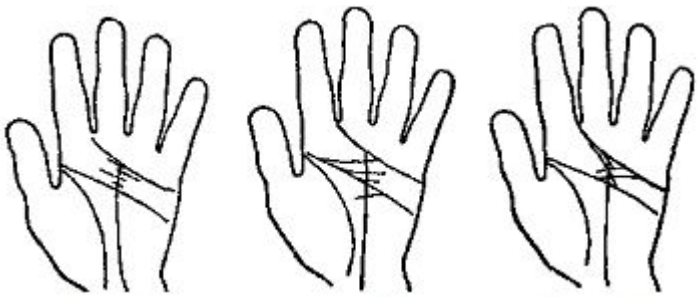
Рис. 7.1. Прямая линия

Толщина линий определяется свойством **PenWidth**, а её цвет – свойством **PenColor** того же класса *GraphicsWindow.*

Проект *Случайные линии*

Исходный код программы находится в папке **Случайные линии**.

Поскольку для рисования линий достаточно взять две точки, то новый проект будет совершенно бесхитростным:



```
{$apptype windows}

// Случайные линии
// ПРОГРАММА ДЛЯ ВЫЧЕРЧИВАНИЯ
// СЛУЧАЙНЫХ ЦВЕТНЫХ ЛИНИЙ

uses
   DrawUnit;

begin
   Draw.Prepare();
   Draw.Draw();
end.

unit DrawUnit;

uses
   Microsoft.SmallBasic.Library, System, System.Threading;

type
   Draw = class

      const GWWIDTH = 480;
      const GWHEIGHT = 320;

      class procedure Prepare();
      begin
         GraphicsWindow.Hide();
         GraphicsWindow.Title := 'Случайные линии';
         GraphicsWindow.Width := GWWIDTH;
         GraphicsWindow.Height := GWHEIGHT;
         GraphicsWindow.Show();
         GraphicsWindow.Left := (Desktop.Width –
                                 GraphicsWindow.Width) / 2;
         GraphicsWindow.Top := (Desktop.Height –
                                GraphicsWindow.Height) / 2;
```

```
        GraphicsWindow.CanResize := false;
        GraphicsWindow.BackgroundColor := 'Black';
    end;
```

Вместо координат одной точки, как при рисовании пикселей, нам понадобятся координаты *двух* точек, которые мы затем и соединим прямой линией посредством вызова метода *DrawLine* с соответствующими параметрами:

```
class procedure Draw();
begin
   var rand := new System.Random();
   var width := GWWIDTH;
   var height := GWHEIGHT;
   // толщина линий:
   var penWidth := 2;//20;

   GraphicsWindow.PenWidth := penWidth;
   // Чертим цветные линии
   while(true ) do
   begin
      // задаём случайные координаты начала и конца линии -->
      // координаты первой точки:
      var x1 := rand.Next(width - penWidth div 2);
      var y1 := rand.Next(height - penWidth div 2);
      // координаты второй точки:
      var x2 := rand.Next(width - penWidth div 2);
      var y2 := rand.Next(height - penWidth div 2);
      // выбираем случайный цвет линии:
      var clr := GraphicsWindow.GetRandomColor();
      GraphicsWindow.PenColor := clr;
      // проводим линию:
      GraphicsWindow.DrawLine(x1, y1, x2, y2);
      Thread.Sleep(TimeSpan.FromMilliseconds(30));
   end;
```

Запускаем программу – и тут уж ничего не добавишь: картина куда краше, чем с точками (Рис. 7.2). А мало этого, увеличьте толщину пера до 20 – и смотрите сами (Рис. 7.3):

```
var penWidth := 20;
```

Если толщина линий большая, то они больше напоминают **прямоугольники**.

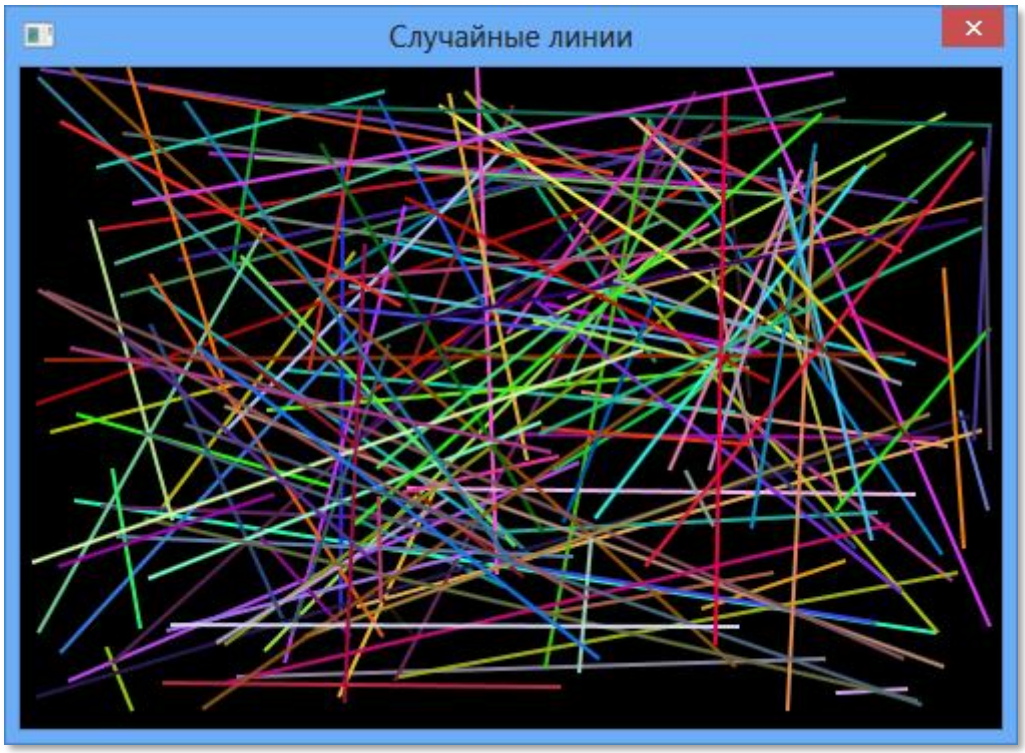


Рис. 7.2. Случайные линии

Рис. 7.3. Толстые случайные линии

Проект *Цветные линии*

Исходный код программы находится в папке **Цветные линии**.

Когда отрезки беспорядочно разбросаны по экрану, то глаз им не радуется. Однако есть много способов с помощью одних только прямых линий нарисовать шикарные узоры. Сейчас мы напишем совсем короткую программу, которая «напишет» великолепную картинку (Рис. 7.4).


```
{$apptype windows}

// Цветные линии

uses
   DrawUnit;

begin
   Draw.Prepare();
   Draw.Draw();
end.


unit DrawUnit;

uses
   Microsoft.SmallBasic.Library, System, System.Threading;

type
   Draw = class

      const GWWIDTH = 640;
      const GWHEIGHT = 480;

      class procedure Prepare();
      begin
         GraphicsWindow.Hide();
         GraphicsWindow.Title := 'Цветные линии';
```

```
        GraphicsWindow.Width := GWWIDTH;
        GraphicsWindow.Height := GWHEIGHT;
        GraphicsWindow.Show();
        GraphicsWindow.Left := (Desktop.Width -
                                GraphicsWindow.Width) / 2;
        GraphicsWindow.Top := (Desktop.Height -
                               GraphicsWindow.Height) / 2;
        GraphicsWindow.CanResize := false;
        GraphicsWindow.BackgroundColor := 'Black';
    end;

    class procedure Draw();
    begin
        var width := GWWIDTH;
        var height := GWHEIGHT;
        // толщина линий:
        var penWidth := 1;

        GraphicsWindow.PenWidth := penWidth;
        // отношение высоты окна к ширине:
        var ratio := (double)(height) / width;
        var step := 10;
        var x := 0;
        while (x < width) do
        begin
            GraphicsWindow.PenColor := 'Red';
            GraphicsWindow.DrawLine(0, x * ratio, width - x, 0);
            GraphicsWindow.PenColor := 'Yellow';
            GraphicsWindow.DrawLine(0, (width - x) * ratio, width - x,
                                    width * ratio);
            GraphicsWindow.PenColor := 'Blue';
            GraphicsWindow.DrawLine(width - x, 0 * ratio, width,
                                        (width - x) * ratio);
            GraphicsWindow.PenColor := 'Green';
            GraphicsWindow.DrawLine(width - x, width * ratio, width,
                                x * ratio);
            Thread.Sleep(TimeSpan.FromMilliseconds(30));
            x += step;
        end;
    end;
  end; //end of class

end.
```

Чтобы каждый из четырёх наборов линий имел свой собственный цвет, перед рисованием прямой линии мы задаём её цвет методом *Графического окна* **PenColor**. Как изменяются координаты начала и конца линии, хорошо видно на рисунке – они скользят по границам клиентской области окна с заданным шагом *Step*.

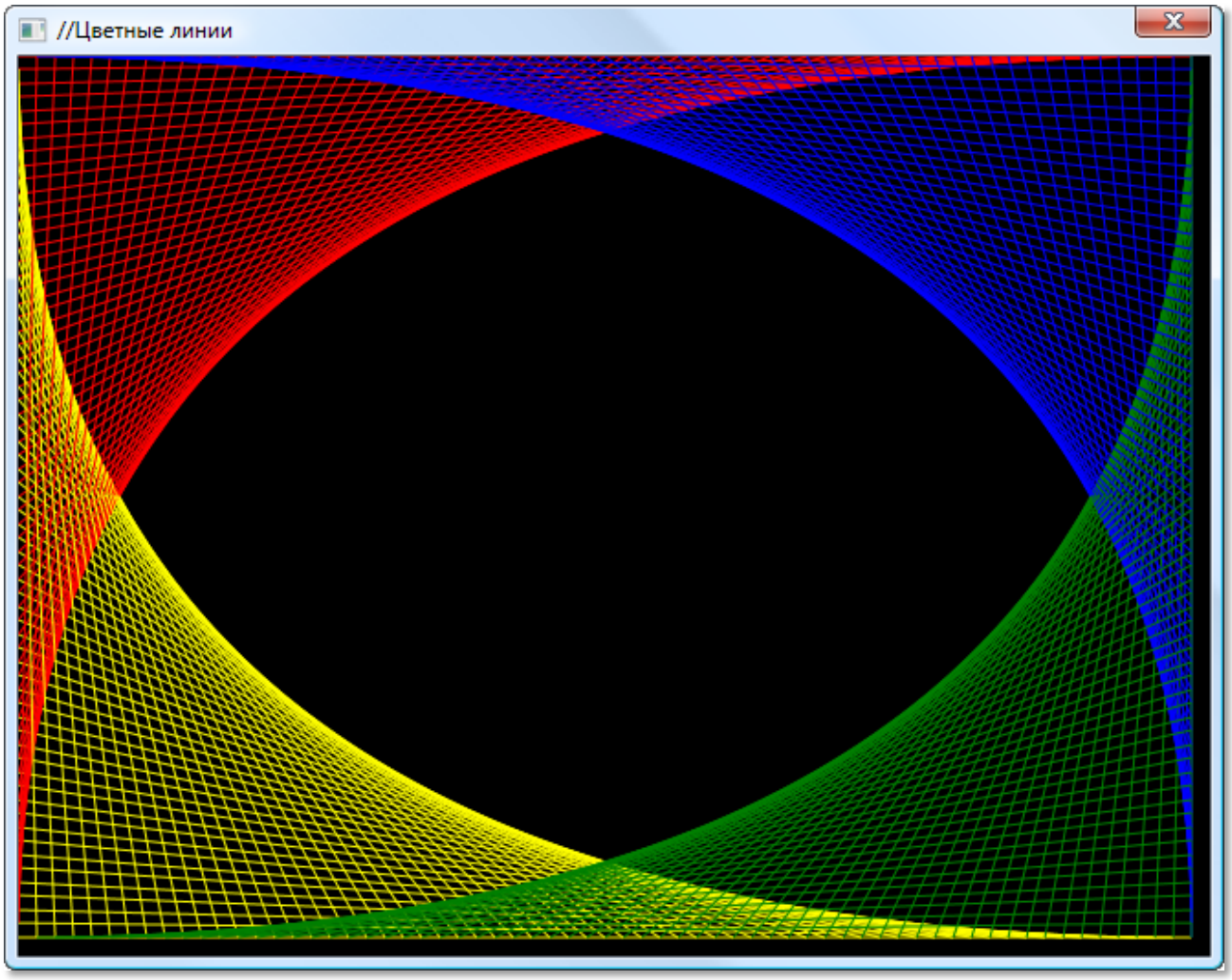


Рис. 7.4. Цветные линии

Изменяя этот параметр, а также цвет линий, вы сможете получить и другие узоры.

Проект *Градиентная заливка*

Исходный код программы находится в папке **Градиентная заливка**.

Если чертить горизонтальные линии вплотную друг к другу, то можно получить картину, ласкающую взор. Сейчас мы испытаем модную нынче **градиентную заливку**, в которой один цвет *плавно* переходит в другой. Этот приём часто применяется в заставках, которые появляются на экране при установке программ, а также в компьютерной графике для раскрашивания кнопок, баннеров и других объектов.

Хорошим примером натуральной «градиентной заливки» может служить безоблачное полуденное или закатное небо!

```
{$apptype windows}

// Градиент

// ПРОГРАММА ДЛЯ ГРАДИЕНТНОЙ
// ЗАЛИВКИ ПРЯМОУГОЛЬНИКА

uses
   DrawUnit;

begin
   var draw:= new Draw();
   draw.Prepare();
   draw.CreateGUI();
end.


unit DrawUnit;

uses
   Microsoft.SmallBasic.Library, System, System.Threading;

type
```

```
  Draw = class

     const GWWIDTH = 640;
     const GWHEIGHT = 480;

     // толщина линий:
     class penWidth := 1;
     // массив кнопок:
     class btn: array of string;
     // размеры окна:
     class height: integer;
     class width: integer;

     public constructor();
     begin
        width := GWWIDTH;
        height := GWHEIGHT;
     end;

     public procedure Prepare();
     begin
        GraphicsWindow.Hide();
        GraphicsWindow.Title := 'Градиент';
        GraphicsWindow.Width := GWWIDTH;
        GraphicsWindow.Height := GWHEIGHT;
        GraphicsWindow.Show();
        GraphicsWindow.Left := (Desktop.Width –
                                GraphicsWindow.Width) / 2;
        GraphicsWindow.Top := (Desktop.Height –
                               GraphicsWindow.Height) / 2;
        GraphicsWindow.CanResize := false;
        GraphicsWindow.BackgroundColor := 'Black';
     end;
```

Всего наша программа сможет выполнить 6 различных градиентных заливок, поэтому нам потребуется именно столько **кнопок**, чтобы пользователь мог по своему желанию выбирать последовательность их просмотра. Для сокращения исходного кода все кнопки мы объединим в массив **btn**:

```
// КНОПКИ
public procedure CreateGUI();
begin
   var y := 40;
   var dy := 32;
   var x := width - 80;
   var n := 0;
   btn := new string[6];
```

```
    btn[n] := Controls.AddButton('Ж --> К', x, y + dy * (n - 1));
    Controls.SetSize(btn[n], 80, 24);
    n += 1;
    btn[n] := Controls.AddButton('Ж --> З', x, y + dy * (n - 1));
    Controls.SetSize(btn[n], 80, 24);
    n += 1;
    btn[n] := Controls.AddButton('Ц --> З', x, y + dy * (n - 1));
    Controls.SetSize(btn[n], 80, 24);
    n += 1;
    btn[n] := Controls.AddButton('Ц --> С', x, y + dy * (n - 1));
    Controls.SetSize(btn[n], 80, 24);
    n += 1;
    btn[n] := Controls.AddButton('Л --> С', x, y + dy * (n - 1));
    Controls.SetSize(btn[n], 80, 24);
    n += 1;
    btn[n] := Controls.AddButton('Л --> К', x, y + dy * (n - 1));
    Controls.SetSize(btn[n], 80, 24);

    Controls.ButtonClicked += OnClick;
end;
```

Надписи на кнопках обозначают направление градиентного перехода, а начальный и конечный цвета указаны первой буквой названия цвета:

Ж – жёлтый

К – красный

Ц – циан

С – синий

Л – лиловый

После нажатия на любую из этих кнопок программа передаёт управление методу-обработчику **OnClick**, в котором выполняется метод, закреплённый за нажатой кнопкой:

```
// НАЧИНАЕМ ЗАКРАСКУ
private procedure OnClick();
begin
    var button := (string)(Controls.LastClickedButton);
    if (button = btn[0]) then YR()
    else if (button = btn[1]) then YG()
    else if (button = btn[2]) then CG()
    else if (button = btn[3]) then CB()
    else if (button = btn[4]) then MB()
    else if (button = btn[5]) then MR();
end;
```

Названия методов также обозначены первыми буквами цветов, образующих градиент, так что запутаться в них вам вряд ли удастся.

Поскольку сами методы, в отличие от заливок, весьма однообразны, то мы рассмотрим только два их них. Остальные вы можете посмотреть самостоятельно в исходном коде программы.

При переходе от **циана** к **синему** цвету (Рис. 7.5) *синяя* составляющая текущего цвета всегда равна 255 (максимальное значение), *красная* – нулю (отсутствует вообще), а *зелёная* составляющая постепенно уменьшает своё значение при перемещении линий сверху вниз от 255 до 0. Именно благодаря этому изменению и возникает плавный переход цвета по вертикали. Так как по горизонтали цвет остается без изменений, то достаточно провести горизонтальную линию текущего цвета:

```
// ПЕРЕХОД ОТ ЦИАНА  К СИНЕМУ - CYAN TO BLUE
private procedure CB();
begin
   var b := 255;
   var r := 0;
   var i := 0.0;
   while(i < height+9) do
   begin
      // вычисляем интенсивность зелёной составляющей цвета:
      var g := (int)(255 * (1 - i / height));
      // задаём цвет линии:
      var clr := GraphicsWindow.GetColorFromRGB(r, g, b);
      GraphicsWindow.PenColor := clr;
      // проводим горизонтальную линию:
      GraphicsWindow.DrawLine(0, i, width - 90, i);
      Thread.Sleep(TimeSpan.FromMilliseconds(1));
      i += 1.0;
   end;
end;
```

Для создания градиентного перехода от **лилового** цвета к **красному** (Рис. 7.6) мы поступаем аналогично. От предыдущего этот градиент отличается только составляющими цвета:

```
private procedure MR();
begin
   var r := 255;
   var g := 0;
   var i := 0.0;
   while(i < height+9) do
   begin
      // вычисляем интенсивность синей составляющей цвета:
```

```
        var b := (int)(255 * (1 - i / height));
        // задаём цвет линии:
        var clr := GraphicsWindow.GetColorFromRGB(r, g, b);
        GraphicsWindow.PenColor := clr;
        // проводим горизонтальную линию:
        GraphicsWindow.DrawLine(0, i, width - 90, i);
        Thread.Sleep(TimeSpan.FromMilliseconds(1));
        i += 1.0;
    end;
end;
```

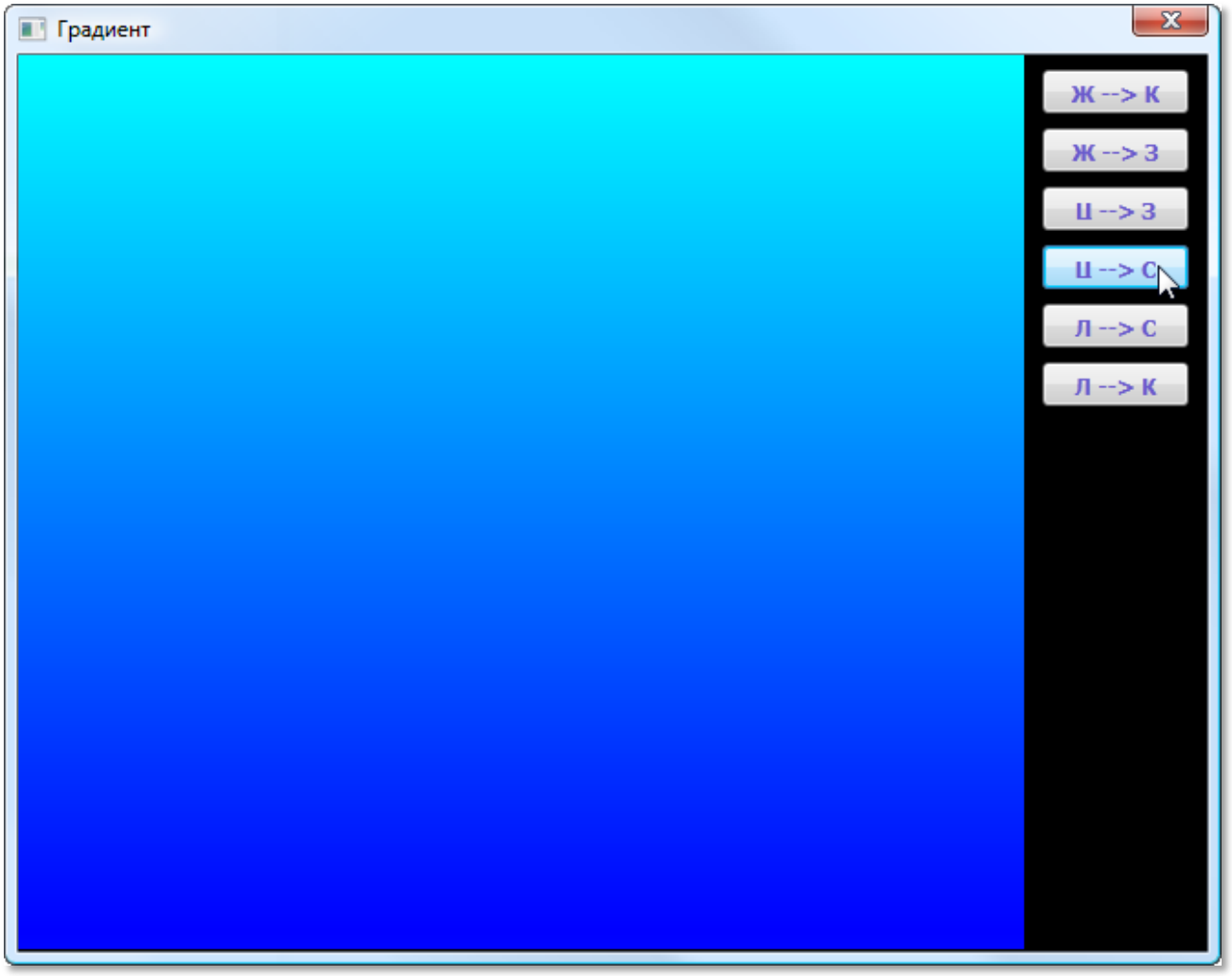


Рис. 7.5. Переход от **циана** к **синему**

Как видите, создать прямоугольную градиентную заливку совсем несложно, а результат получается впечатляющий! Запускайте программу, жмите на кнопки и наслаждайтесь **цветовыми** переходами!

Рис. 7.6. Переход от **лилового** цвета к **красному**

Задания для самостоятельного решения

Градиенты

1. Измените программу так, чтобы она чертила линии **вертикально** (горизонтальный градиент) (Рис. 7.7)!

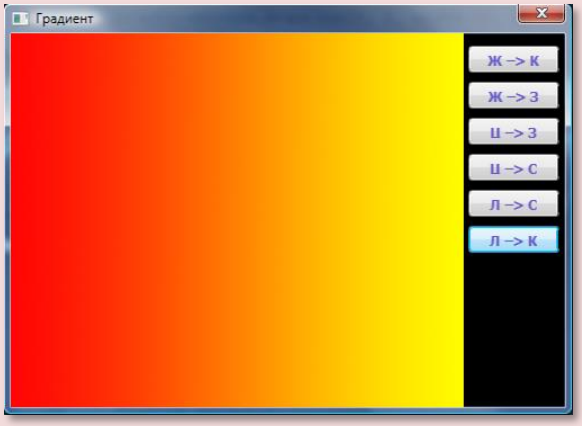


Рис. 7.7. Горизонтальный градиент

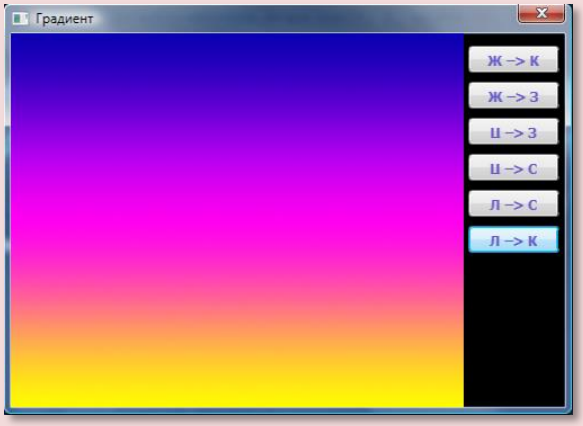


Рис. 7.8. Двойной градиент

2. Подумайте, как запрограммировать двойной (Рис. 7.8), радиальный (Рис. 7.9) и квадратный (Рис. 7.10) градиенты.

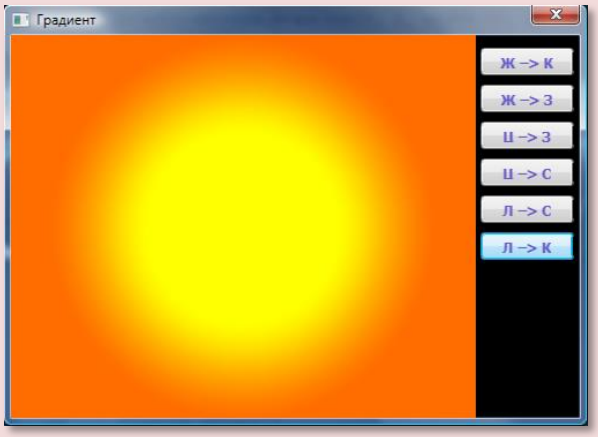


Рис. 7.9. Радиальный градиент

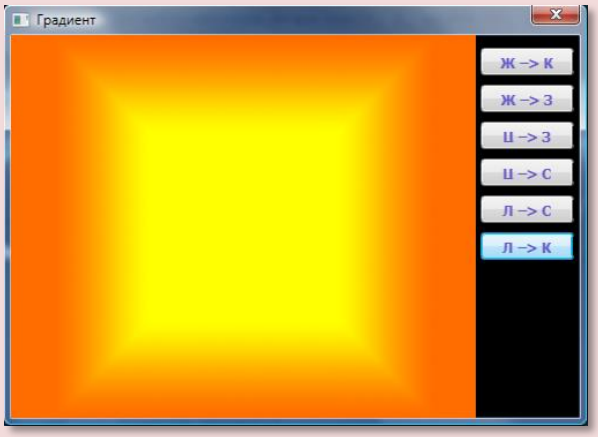


Рис. 7.10. Квадратный градиент

Глава 8. Геометрические фантазии

Из отрезков можно построить любые многоугольники, но для прямоугольников и треугольников в библиотеке *SmallBasicLibrary* имеются специальные методы.

Прямоугольники и квадраты

Полезный во всех отношениях метод

```
DrawRectangle (x, y, width, height : double);
```

рисует **контурный прямоугольник**, верхний левый угол которого задаётся координатами (*x, y*), а ширина и высота определяются параметрами *width* и *height*. Цвет контура устанавливается с помощью свойства **PenColor** *Графического окна*, а его толщина – с помощью свойства **PenWidth** (Рис. 8.1).

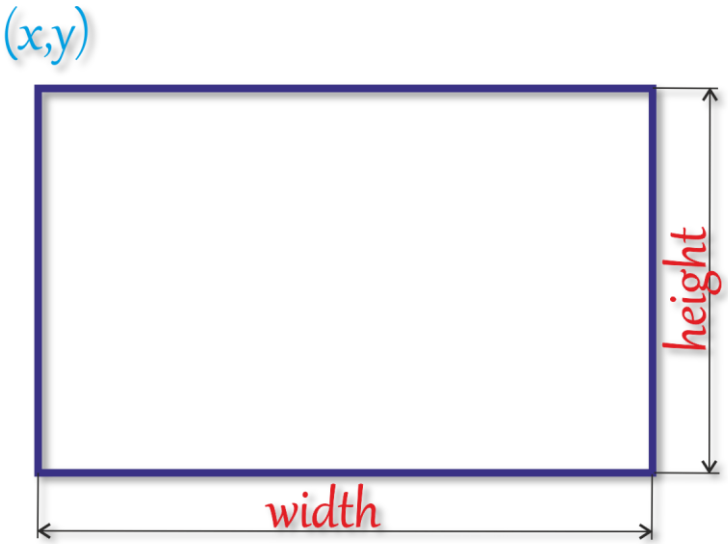


Рис. 8.1. *Контурный* прямоугольник

Для рисования **закрашенных прямоугольников** имеется метод:

```
FillRectangle (x, y, width, height : double);
```

Их размеры и положение на экране задаются точно так же, как и контурных прямоугольников, но они не имеют контура, а цвет заливки определяется цветом кисти **BrushColor** *Графического окна* (Рис. 8.2).

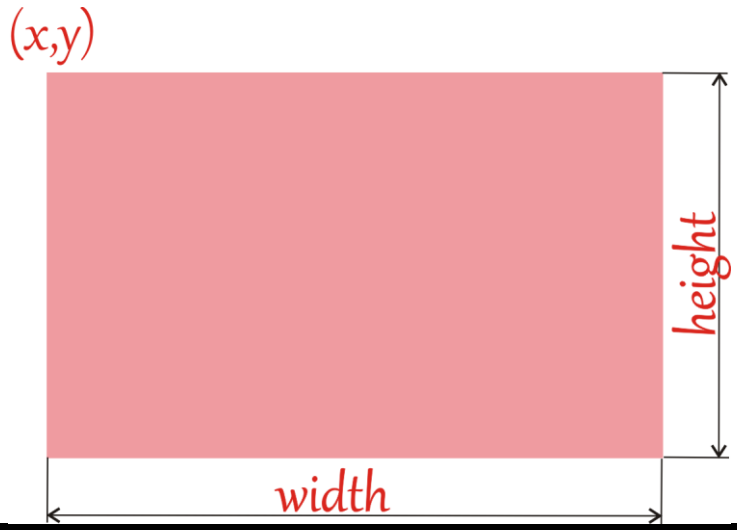


Рис. 8.2. *Закрашенный* прямоугольник

> Если ширина прямоугольника равна высоте, то получится **квадрат**.

Проект *Прямоугольники*

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

А теперь давайте позабавимся, рисуя случайные прямоугольники.

Начнём мы с контурных прямоугольников. Теперь вместо прямых линий мы будем рисовать прямоугольники:

```
{$apptype windows}

// Случайные прямоугольники

// ПРОГРАММА ДЛЯ ВЫЧЕРЧИВАНИЯ
```

```
// СЛУЧАЙНЫХ ПРЯМОУГОЛЬНИКОВ

uses
   DrawUnit;

begin
   Draw.Prepare();
   Draw.Draw();
end.

unit DrawUnit;

uses
   Microsoft.SmallBasic.Library, System, System.Threading;

type
   Draw = class

      const GWWIDTH = 480;
      const GWHEIGHT = 320;

      class procedure Prepare();
      begin
         GraphicsWindow.Hide();
         GraphicsWindow.Title := 'Случайные прямоугольники';
         GraphicsWindow.Width := GWWIDTH;
         GraphicsWindow.Height := GWHEIGHT;
         GraphicsWindow.Show();
         GraphicsWindow.Left := (Desktop.Width –
                                 GraphicsWindow.Width) / 2;
         GraphicsWindow.Top := (Desktop.Height –
                                 GraphicsWindow.Height) / 2;
         GraphicsWindow.CanResize := false;
         GraphicsWindow.BackgroundColor := 'Black';
      end;

      class procedure Draw();
      begin
         var width := GWWIDTH;
         var height := GWHEIGHT;
         // толщина контура:
         var penWidth := 5;
         GraphicsWindow.PenWidth := penWidth;
         var rand := new Random();
         // Чертим цветные прямоугольники
         while(true ) do
         begin
            // координаты левой верхней вершины прямоугольника:
            var x := rand.Next(width - penWidth div 2);
            var y := rand.Next(height - penWidth div 2);
```

```
            // размеры прямоугольника:
            var w := rand.Next(width - x - penWidth div 2);
            var h := rand.Next(height - y - penWidth div 2);
            // выбираем случайный цвет контура:
            var clr := GraphicsWindow.GetRandomColor();
            GraphicsWindow.PenColor := clr;
            // рисуем контурный прямоугольник:
            GraphicsWindow.DrawRectangle(x, y, w, h);
            Thread.Sleep(TimeSpan.FromMilliseconds(30));
        end;
    end;
  end; //end of class

end.
```

Запускаем программу и визуально радуемся нашим «канвасным» достижениям (Рис. 8.3).

Совсем немного изменим программу - и вместо контурных прямоугольников на нас обрушится лавина прямоугольников, окрашенных во все **цвета радуги** (Рис. 8.4):

```
// Чертим цветные прямоугольники
while(true ) do
begin
   // координаты левой верхней вершины прямоугольника:
   var x := rand.Next(width - penWidth div 2);
   var y := rand.Next(height - penWidth div 2);

   // размеры прямоугольника:
   var w := rand.Next(width - x - penWidth div 2);
   var h := rand.Next(height - y - penWidth div 2);
   // выбираем случайный цвет контура:
   var clr := GraphicsWindow.GetRandomColor();
   // рисуем контурный прямоугольник:
   //GraphicsWindow.PenColor := clr;
   //GraphicsWindow.DrawRectangle(x, y, w, h);
   //рисуем закрашенный прямоугольник:
   GraphicsWindow.BrushColor := clr;
   GraphicsWindow.FillRectangle(x, y, w, h);

   Thread.Sleep(TimeSpan.FromMilliseconds(30));
end;
```

Рис. 8.3. *Контурные* прямоугольники

Рис. 8.4. Закрашенные прямоугольники

Эллипсы и круги

Для рисования **эллипсов** *Графическое окно* также предоставляет два метода, которые почти ничем не отличаются от «прямоугольных». Единственное различие состоит в том, что у эллипса нет вершин, поэтому его положение на канве задаётся координатами верхнего левого угла описанного прямоугольника.

Метод

```
DrawEllipse(x, y, width, height : double);
```

рисует **контурный эллипс** (Рис. 8.5).

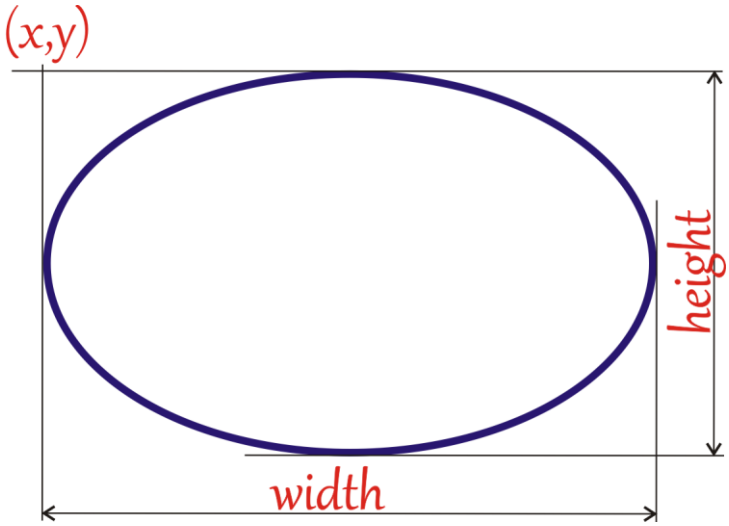


Рис. 8.5. Контурный эллипс

А второй метод

```
FillEllipse(x, y, width, height : double);
```

рисует **закрашенный эллипс** (Рис. 8.6).

Если ширина эллипса равна высоте, то получится **круг (окружность)**.

Измените программу для рисования прямоугольников так, чтобы она рисовали эллипсы!

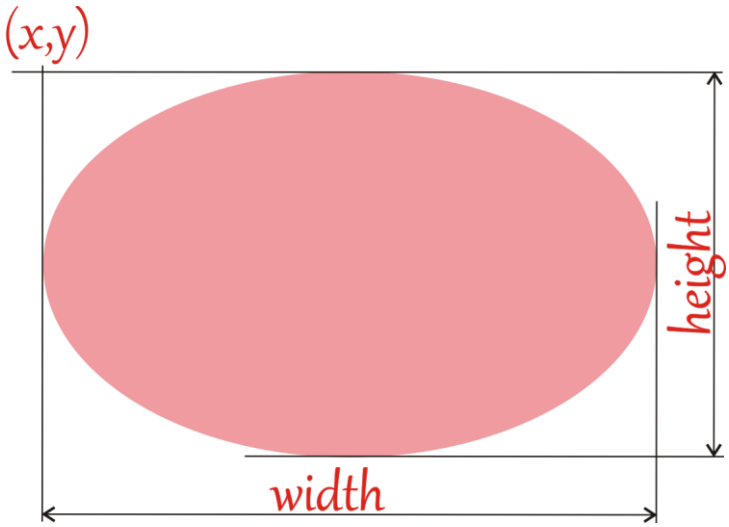


Рис. 8.6. Закрашенный эллипс

Треугольники

Вероятно, вы уже догадались, что и для вычерчивания **треугольников** *Графическое окно* также запаслось двумя методами (Рис. 8.7 и 8.8):

```
DrawTriangle(x1 : double; y1 : double,
             x2 : double; y2 : double,
             x3 : double; y3 : double);
```

```
FillTriangle(x1 : double; y1 : double,
             x2 : double; y2 : double,
             x3 : double; y3 : double);
```

Проще всего задать размеры треугольника и его положение на канве координатами трёх его вершин, что и сделано в этих методах.

Дабы лишний раз не повторяться, мы не будем рисовать случайные треугольники, а построим из маленьких треугольников один большой. Это будет свежо и экспрессивно!

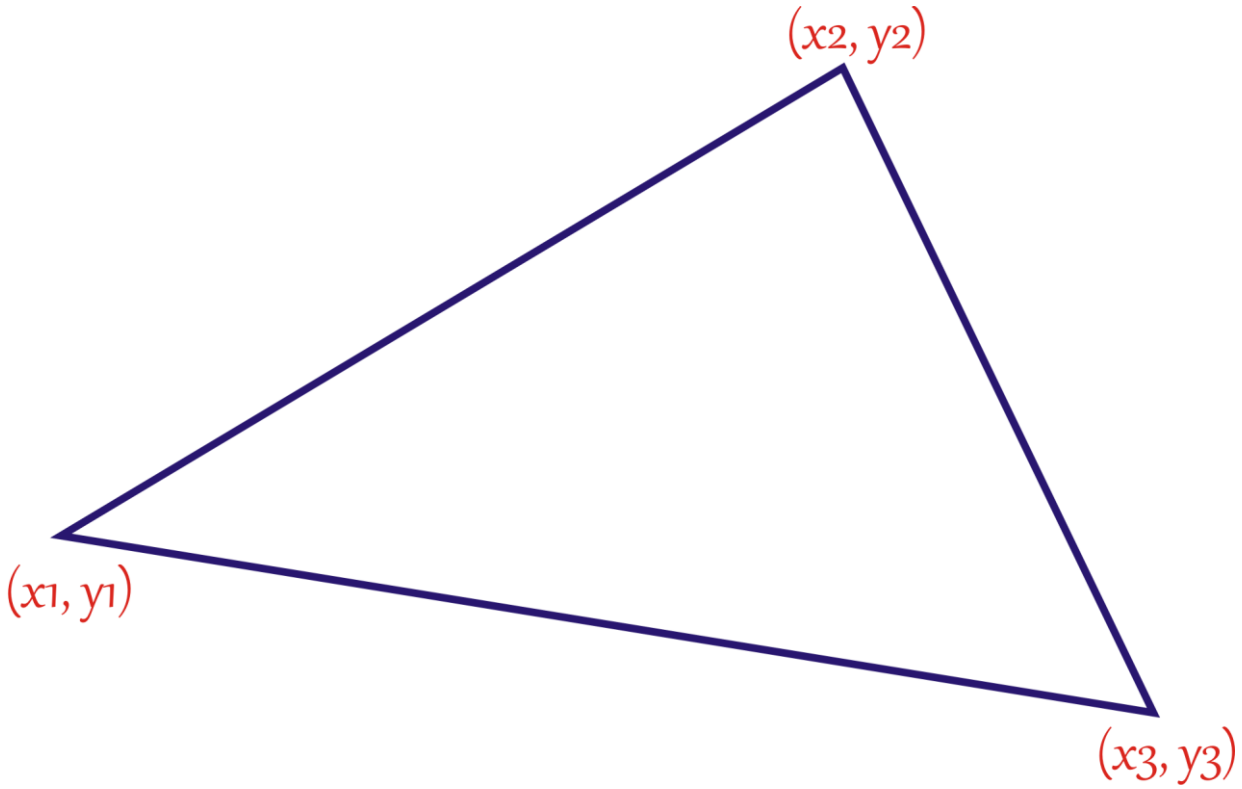


Рис. 8.7. *Контурный* треугольник

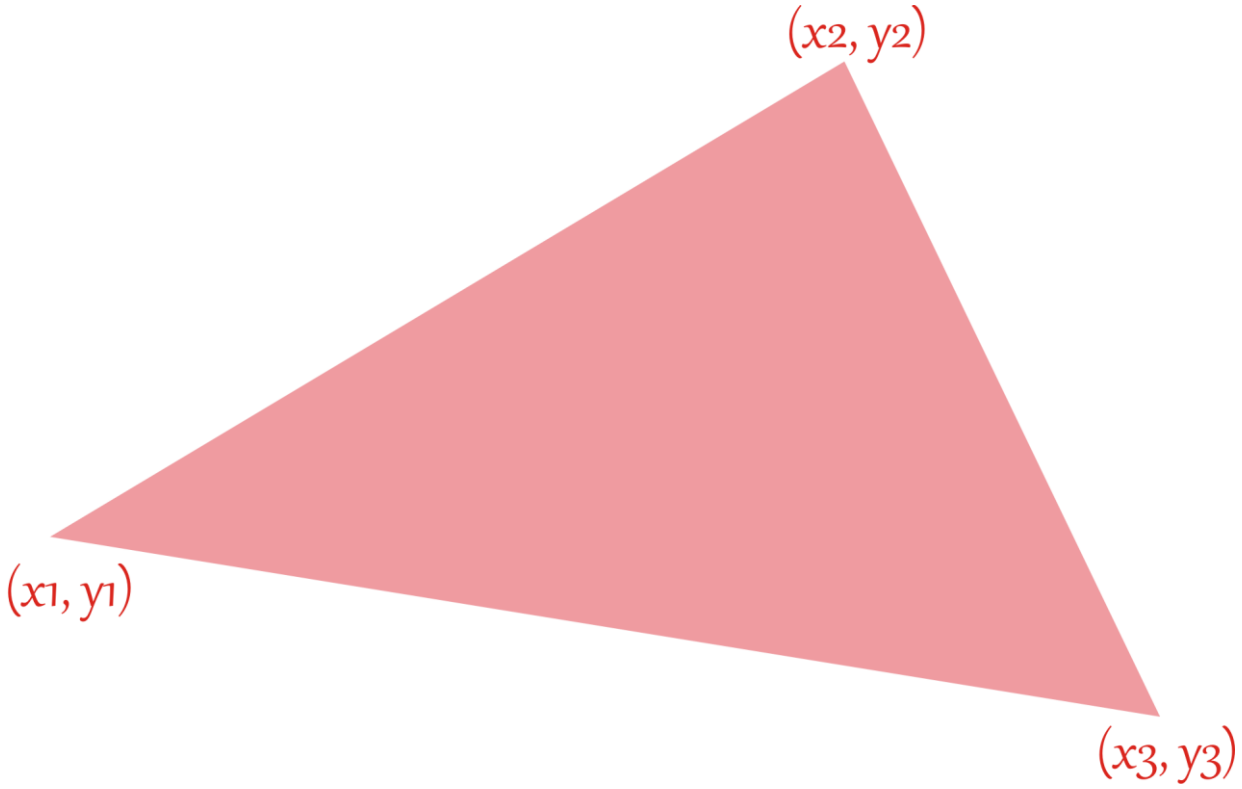


Рис. 8.8. Закрашенный треугольник

Проект *Треугольники*

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

За основу мы возьмём предыдущий проект, но увеличим высоту окна для гармонизации нашего произведения:

```
{$apptype windows}

// Треугольники

// ПРОГРАММА ДЛЯ ВЫЧЕРЧИВАНИЯ
// ТРЕУГОЛЬНИКА ИЗ ТРЕУГОЛЬНИКОВ


uses
   DrawUnit;

begin
   Draw.Prepare();
   Draw.Draw();
end.

unit DrawUnit;

uses
   Microsoft.SmallBasic.Library, System, System.Threading;

type
   Draw = class

      const GWWIDTH = 480;
      const GWHEIGHT = 440;

      //длина стороны треугольника:
      const SIZE = 32;

      class procedure Prepare();
      begin
         GraphicsWindow.Hide();
         GraphicsWindow.Title := 'Треугольники';
         GraphicsWindow.Width := GWWIDTH;
         GraphicsWindow.Height := GWHEIGHT;
         GraphicsWindow.Show();
         GraphicsWindow.Left := (Desktop.Width –
```

```
                                GraphicsWindow.Width) / 2;
    GraphicsWindow.Top := (Desktop.Height -
                           GraphicsWindow.Height) / 2;
    GraphicsWindow.CanResize := false;
    GraphicsWindow.BackgroundColor := 'Black';
end;
```

Маленькие треугольники обращены вершиной вниз и образуют столбики, высота которых уменьшается по мере удаления столбиков от центра.

Первый столбик начинается треугольником с координатами левой вершины (*x1, y1*), а следующие столбики смещены влево и вправо на *size/4*3*(j)* пикселей. Кроме того, каждый следующий столбик начинается ниже предыдущего на *j*dy*1.5* пикселей.

Почему так происходит, хорошо видно на готовой картинке (Рис. 8.9):

```
public class procedure Draw();
begin
   var width := GWWIDTH;
   var height := GWHEIGHT;
   // толщина контура:
   var penWidth := 2;
   // цвет контура:
   var penColor := 'Yellow';
   GraphicsWindow.PenWidth := penWidth;
   GraphicsWindow.PenColor := penColor;

   // координаты левого угла верхнего треугольника:
   var x1 := width / 2.0 - size / 2.0;
   var y1 := 10.0;
   // высота треугольников:
   var dy := size * System.Math.Sin(GetRadians(60));
   // запоминаем начальные значения:
   var yt := y1;
   var xt := x1;

    // рисуем столбики из треугольников:
   for var j := 0 to 14 do
   begin
      // левый столбик:
      y1 := yt + j * dy * 1.5;
      x1 := xt - size / 4 * 3 * j;
      var i := 1.0;
      while(i <= 15 - 1.5 * j) do
      begin
         DrawTre(x1, y1, dy);
```

```
            y1 += dy;
            i += 1.0;
         end;
         // правый столбик:
         y1 := yt + j * dy * 1.5;
         x1 := xt + size / 4 * 3 * j;
         i := 1.0;
         while(i <= 15 - 1.5 * j) do
         begin
            DrawTre(x1, y1, dy);
            y1 += dy;
            i += 1.0;
         end
      end;

   end;
end; //end of class

end.
```

В методе **DrawTre** мы рисуем *один* треугольник:

```
// ЧЕРТИМ ТРЕУГОЛЬНИК
private class procedure DrawTre(x1, y1, dy: double);
begin
   GraphicsWindow.DrawTriangle(x1, y1,
                               x1 + SIZE, y1,
                               x1 + SIZE / 2, y1 + dy);
   Thread.Sleep(TimeSpan.FromMilliseconds(30));
end;
```

Также нам понадобится метод для перевода градусов в радианы:

```
private class function GetRadians(a: double): double;
begin
   Result := a / 180 * System.Math.PI;
end;
```

Запускаем программу – пирамида из треугольников построена (Рис. 8.9)!

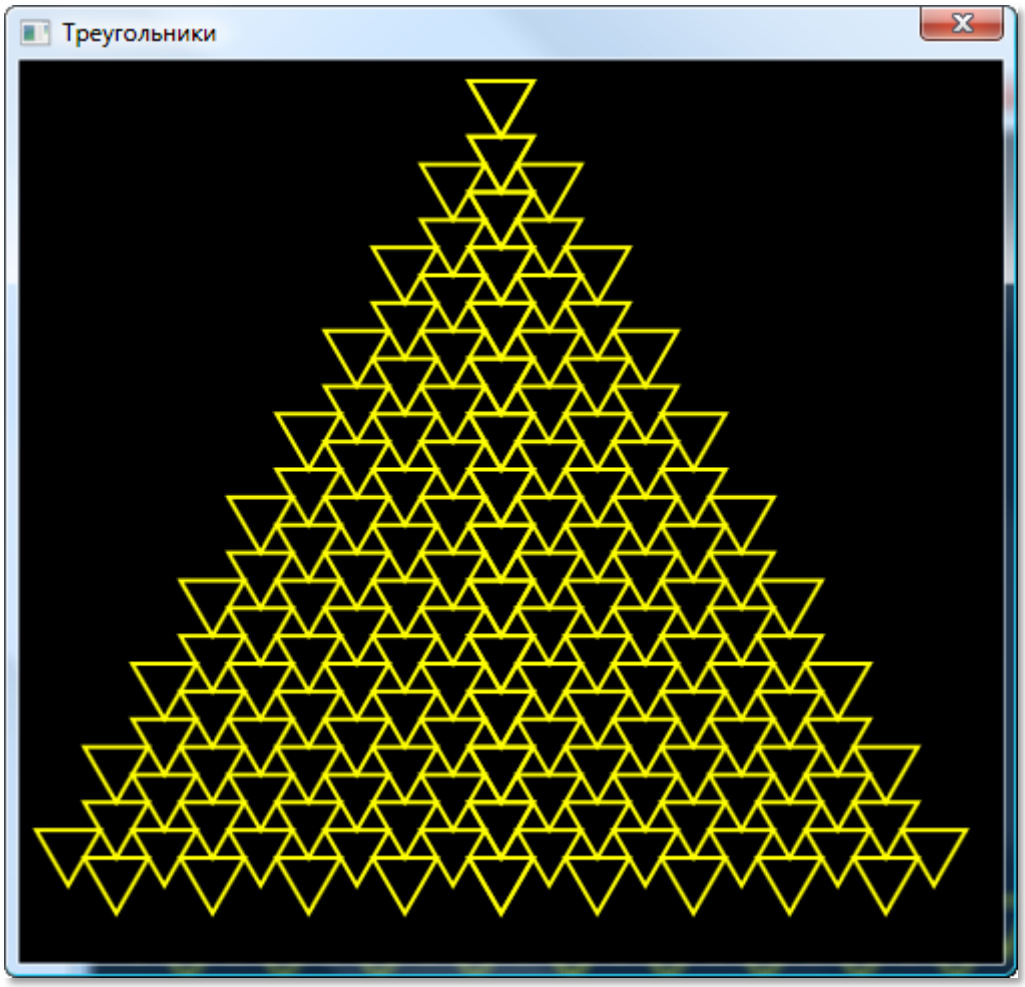


Рис. 8.9. Треугольный треугольник

Проект *Живые картинки*

Исходный код программы находится в папке **Живые картинки**.

До сих пор мы занимались *статическими* картинками: нарисовал их – и всё! А ведь гораздо интереснее наблюдать на экране за непрерывными изменениями геометрических фигур. Сейчас мы напишем программу, которая будет рисовать прямоугольники или эллипсы, постоянно обновляющие картину на экране.

Поскольку эта программа не столько сложная, сколько заковыристая, то и **переменных** нам потребуется немало:

```
{$apptype windows}

// Rectangular

// ПРОГРАММА ДЛЯ ВЫЧЕРЧИВАНИЯ
// ДИНАМИЧЕСКИХ ПРЯМОУГОЛЬНИКОВ и ЭЛЛИПСОВ

uses
   DrawUnit;

begin
   var draw := new Draw();
   draw.Prepare();
   draw.CreateGUI();
   draw.Draw();
end.
```

Так как справа мы разместили кнопки, то это обстоятельство следует учесть при рисовании:

```
unit DrawUnit;

uses
   Microsoft.SmallBasic.Library, System, System.Threading;

type
   Draw = class

      const GWWIDTH = 480 + 100;
      const GWHEIGHT = 320;
      width: integer;
      height: integer;
      // размеры области рисования:
      x0: integer;
      y0: integer;
      xmax: integer;
      ymax: integer;


     public constructor();
      begin
         width := GWWIDTH;
         height := GWHEIGHT;
         // размеры области рисования:
         x0 := 0;
         y0 := 0;
         xmax := width - 100;
         ymax := height + 9;
      end;


     public procedure Prepare();
      begin
         GraphicsWindow.Hide();
         GraphicsWindow.Title := 'Rectangular';
         GraphicsWindow.Width := GWWIDTH;
         GraphicsWindow.Height := GWHEIGHT;
         GraphicsWindow.Show();
         GraphicsWindow.Left := (Desktop.Width –
                                  GraphicsWindow.Width) / 2;
         GraphicsWindow.Top := (Desktop.Height –
                                 GraphicsWindow.Height) / 2;
         GraphicsWindow.CanResize := false;
         GraphicsWindow.BackgroundColor := 'Black';
      end;
```

Чтобы переключаться между прямоугольниками и эллипсами, установим пару **кнопок**:

```
// КНОПКИ
public procedure CreateGUI();
begin
   var y := 60;
   var dy := 48;
   var x := width - 85;
   var n := 0;
   btn := new string[2];
   btn[n] := Controls.AddButton('Пр-ки', x, y + dy * (n - 1));
   Controls.SetSize(btn[n], 80, 32);
   n += 1;
   btn[n] := Controls.AddButton('Эллипсы', x, y + dy * (n - 1));
   Controls.SetSize(btn[n], 80, 32);
   Controls.ButtonClicked += OnClick;
end;
```

После нажатия на эти кнопки мы попадём в метод-обработчик **OnClick**:

```
// ВЫБИРАЕМ ФИГУРУ ДЛЯ РИСОВАНИЯ
private procedure OnClick();
begin
   GraphicsWindow.BrushColor := 'Black';
   var button := (string)(Controls.LastClickedButton);
   if (button = btn[0]) then
   begin
      GraphicsWindow.FillRectangle(x0, y0, xmax + 4, ymax);
      GraphicsWindow.PenWidth := 2;
      figura := 'Rect';
   end
   else
   begin
      GraphicsWindow.FillRectangle(x0, y0, xmax + 4, ymax);
      GraphicsWindow.PenWidth := 1;
      figura := 'Ellipse';
   end;
end;
```

Здесь мы, прежде всего, стираем все предыдущие кривые, устанавливаем разную толщину контура для прямоугольников и эллипсов (вы можете выбрать свои значения) и сообщаем методу *Draw*, какими фигурами желаем рисовать.

В методе **Draw** переменные *x1, y1* и *x2, y2* задают координаты верхнего левого и правого нижнего углов прямоугольника, а при вызове метода

DrawRectangle нам потребуется длина сторон прямоугольника, которую нам придётся вычислять дополнительно. Рисовать мы будет прямоугольники и эллипсы, а переменная *figura* хранит название текущей геометрической фигуры.

Далее, в бесконечном цикле *while* мы вычерчиваем прямоугольники или эллипсы. Начинаем с самой большой фигуры, размеры которой равны области рисования, затем постепенно координаты вершин прямоугольника всё больше и больше приближаются к центру и, наконец, наступает момент, когда левый верхний угол занимает место правее и/или ниже правого нижнего. Вот тут обязательно нужно поменять эти координаты в вызове метода *DrawRectangle*, иначе он будет работать неправильно!

```
// ЧЕРТИМ ЦВЕТНЫЕ ПРЯМОУГОЛЬНИКИ
public procedure Draw();
begin
   // текущие координаты прямоугольника:
   var x1 := 0;
   var y1 := 0;
   var x2 := 0;
   var y2 := 0;
   // координаты верхнего левого угла фигуры:
   var x := 0;
   var y := 0;
   while(true) do
   begin
      for var i := 6 downto 1 do
      begin
         for var j := 0 to 5 do
         begin
            // стартовый прямоугольник:
            x1 := x0;
            y1 := y0;
            x2 := xmax;
            y2 := ymax;
            // выбираем случайный цвет контура:
            var clr := GraphicsWindow.GetRandomColor();
            GraphicsWindow.PenColor := clr;
            // корректируем координаты верхнего
            // левого угла прямоугольника:
            while (x1 <= xmax) and (y1 <= ymax) do
            begin
               if (x1 > x2) then
                  x := x2
               else
                  x := x1;
               if (y1 > y2) then
                  y := y2
```

```
                else
                    y := y1;
                if (figura = 'Rect') then
                    // рисуем прямоугольник:
                    GraphicsWindow.DrawRectangle(x, y,
                                System.Math.Abs(x2 - x1),
                                System.Math.Abs(y2 - y1))
                else
                    // рисуем эллипс:
                    GraphicsWindow.DrawEllipse(x, y,
                                System.Math.Abs(x2 - x1),
                                System.Math.Abs(y2 - y1));
                Thread.Sleep(TimeSpan.FromMilliseconds(16));
                // новые координаты вершин прямоугольника:
                x1 += i;
                y1 += j;
                x2 -= i;
                y2 -= j;
            end
        end
    end
  end
end;
```

Правила изменения размеров фигур очень простые, но образующиеся при работе программы динамические узоры весьма любопытны, и разглядывать их можно не одну минуту (Рис. 8.10 и 8.11)!

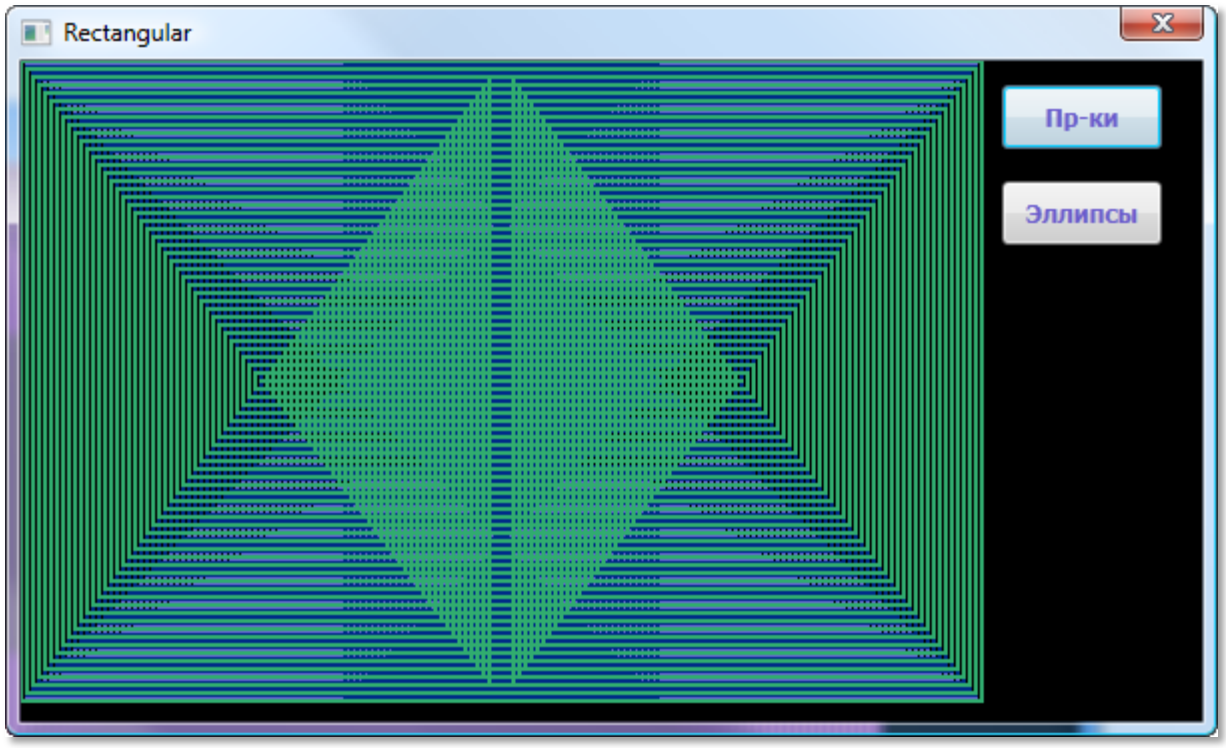


Рис. 8.10. Динамические прямоугольники

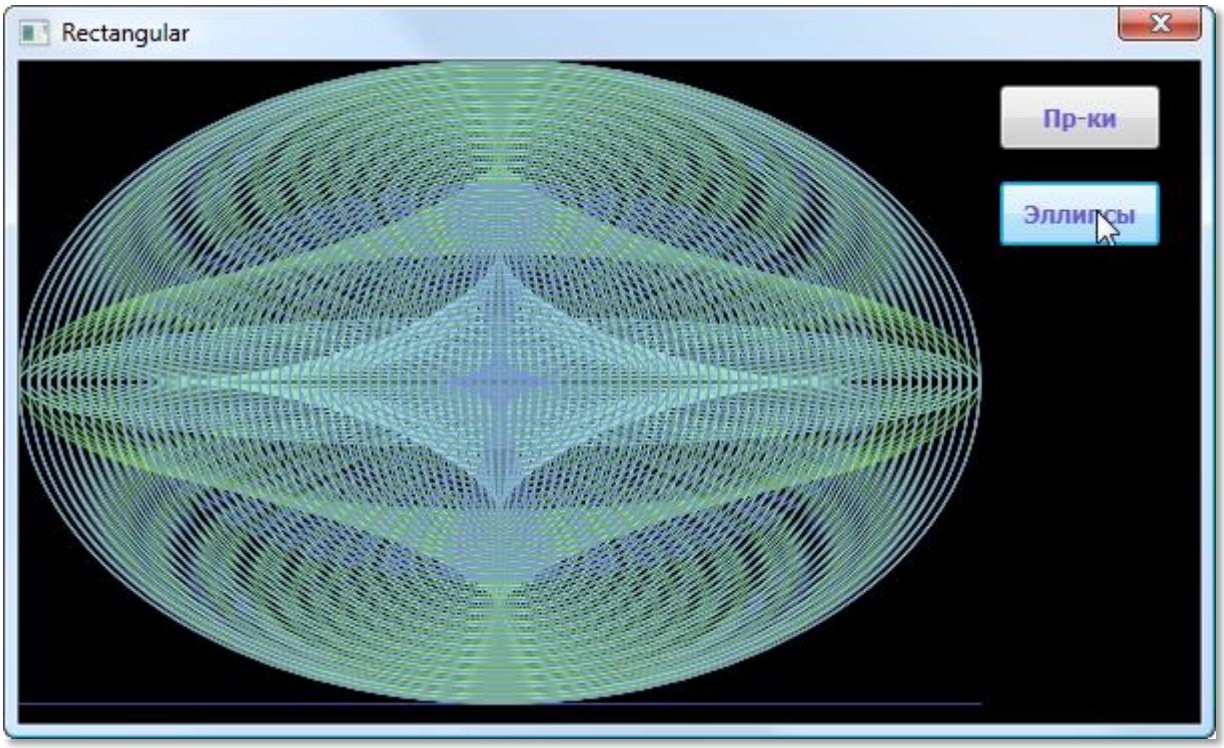


Рис. 8.11. Динамические эллипсы

Глава 9. Картинная галерея

В этой главе мы полюбуемся готовыми картинами и даже замахнёмся на их «»улучшение!

Проект *Фильтруем снимки*

Исходный код программы находится в папке **Фильтруем снимки**.

Вам, без сомнения, не раз попадались на глаза необычные снимки, как бы составленные из цветных квадратиков или точек. Может быть, вы и сами делали такие картинки в *Фотошопе*, который имеет немало *фильтров* для придания фотографиям нового облика. Например, фильтр **Pointilize** (Рис. 9.1), который может увеличивать пиксели исходного изображения в несколько раз. В итоге получается «художественное» изображение, похожее на картины, написанные в технике *пуантилизма* (цветными точками) (Рис. 9.2).

Мы не будем замахиваться на достижения искусства или даже всеми нами любимого *Фотошопа,* а напишем простенький, но свой фильтр, который превратит любую фотографию в «мозаику».

> Для экспериментов всё-таки лучше взять не *любую* фотографию, а красочную и без мелких деталей, которые неизбежно исчезнут при *пикселизации*. Если вам приходилось видеть картины, вышитые крестиком, то смело берите пример с них.

За основу проекта возьмите исходный код *Пипетки* и подгоните размеры окна под выбранную вами фотографию. Поместите её в папку с программой и при запуске программы впечатайте в канву *Графического окна*. В качестве примера я взял фотографию с красными тюльпанами, которая в *Графическом окне* смотрится совсем неплохо (Рис. 9.3).

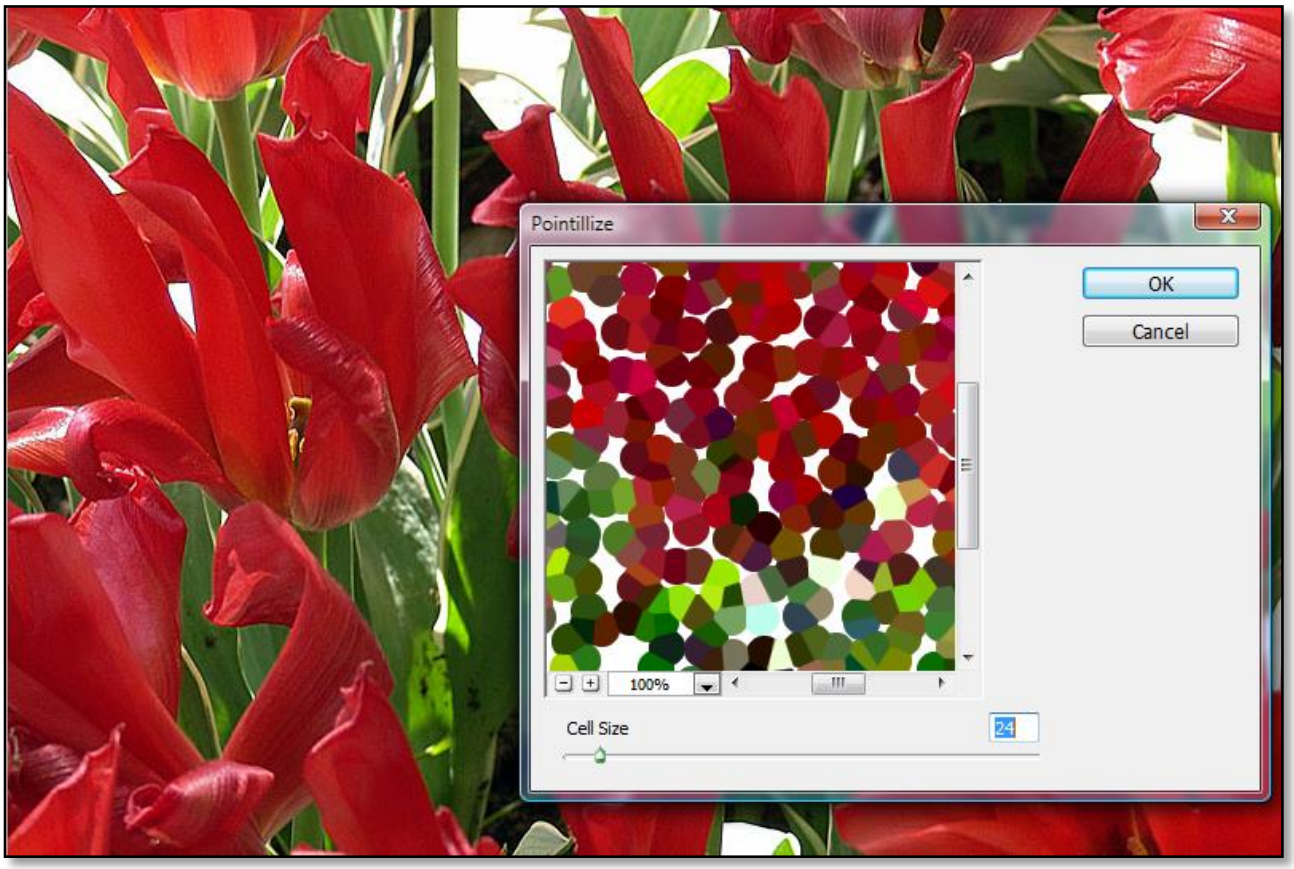


Рис. 9.1. Фильтр *Pointilize* в графическом редакторе *Фотошоп*

```
{$apptype windows}

// Фильтр

// ПРОГРАММА ДЛЯ ПИКСЕЛИЗАЦИИ ИЗОБРАЖЕНИЯ

uses
   DrawUnit;

begin
   Draw.Prepare();
   Draw.Draw();
end.

unit DrawUnit;

uses
```

```
  Microsoft.SmallBasic.Library, System;

type
   Draw = class

      const GWWIDTH = 800-10;
      const GWHEIGHT = 533-10;

      class procedure Prepare();
      begin
         GraphicsWindow.Hide();
         GraphicsWindow.Title := 'Фильтр';
         GraphicsWindow.Width := GWWIDTH;
         GraphicsWindow.Height := GWHEIGHT;
         GraphicsWindow.Show();
         GraphicsWindow.Left := (Desktop.Width –
                                 GraphicsWindow.Width) / 2;
         GraphicsWindow.Top := (Desktop.Height –
                                GraphicsWindow.Height) / 2;
         GraphicsWindow.CanResize := false;
      end;

     class procedure Draw();
      begin
         var width := GWWIDTH + 10;
         var height := GWHEIGHT + 10;
         var path:= Environment.CurrentDirectory;
         //Application.StartupPath;
         var background := ImageList.LoadImage(path +
                                               '\тюльпаны.jpg');
        GraphicsWindow.DrawImage(background, 0, 0);

         while (true) do
         begin
            var x := GraphicsWindow.MouseX;
            var y := GraphicsWindow.MouseY;
            var clr := GraphicsWindow.GetPixel(x, y);
            //окошко:
            GraphicsWindow.BrushColor := clr;
            GraphicsWindow.FillRectangle(width - 32 - 6, 20,
                                         32, 32);
            var str := 'Цвет пикселя (' + x.ToString() + ','  +
                       y.ToString() + ') = ' + (string)(clr);
            GraphicsWindow.Title := str;
         end;
      end;
   end; //end of class

end.
```

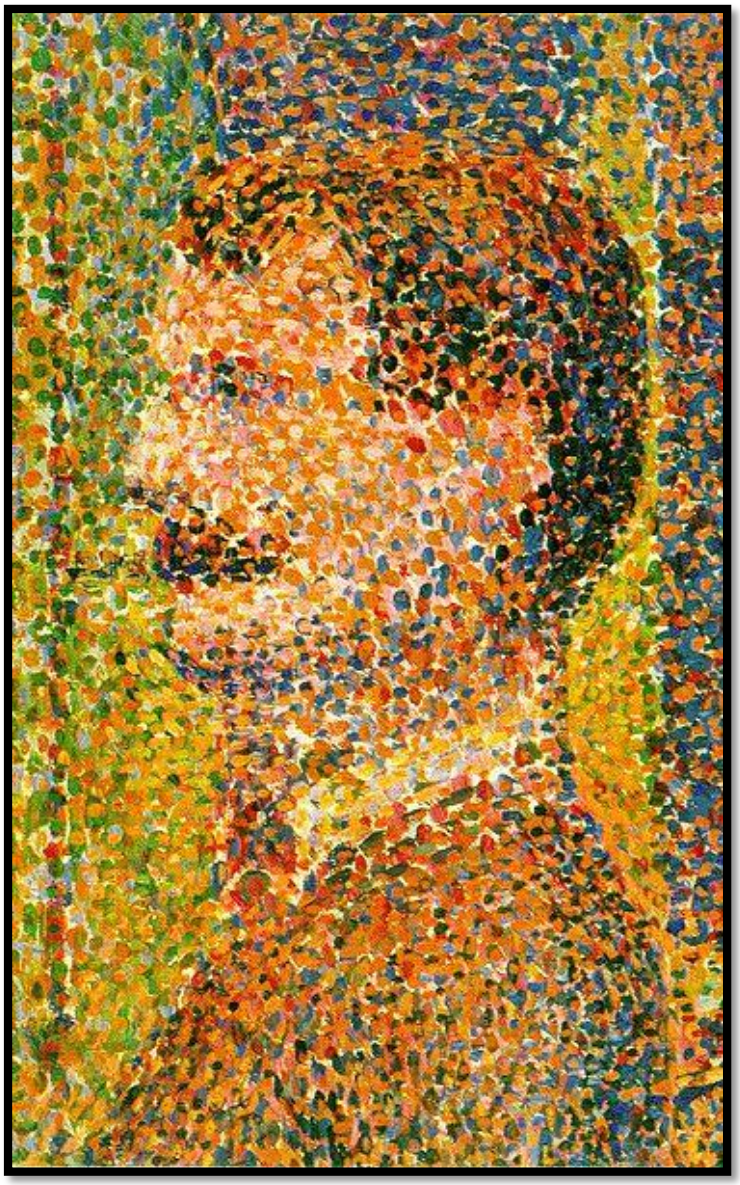


Рис. 9.2. Фрагмент картины Ж. Сёра

Обычно все картинки хранятся в одной папке, поэтому вы можете изменить полный путь к этой папке так, чтобы загружать нужную картинку из неё.

Если же вы хотите использовать в программе только конкретную картинку, то поместите её в папку с исходным кодом программы и укажите путь к картинке в этой папке.

Мы напишем *два* фильтра, которые отличаются друг от друга только тем, что в первом точки *квадратные*, а во втором - *круглые*. С точки зрения геометрии, разница небольшая, но в искусстве, как известно мелочей нет, поэтому пробуйте разные варианты, пока не добьётесь совершенства.

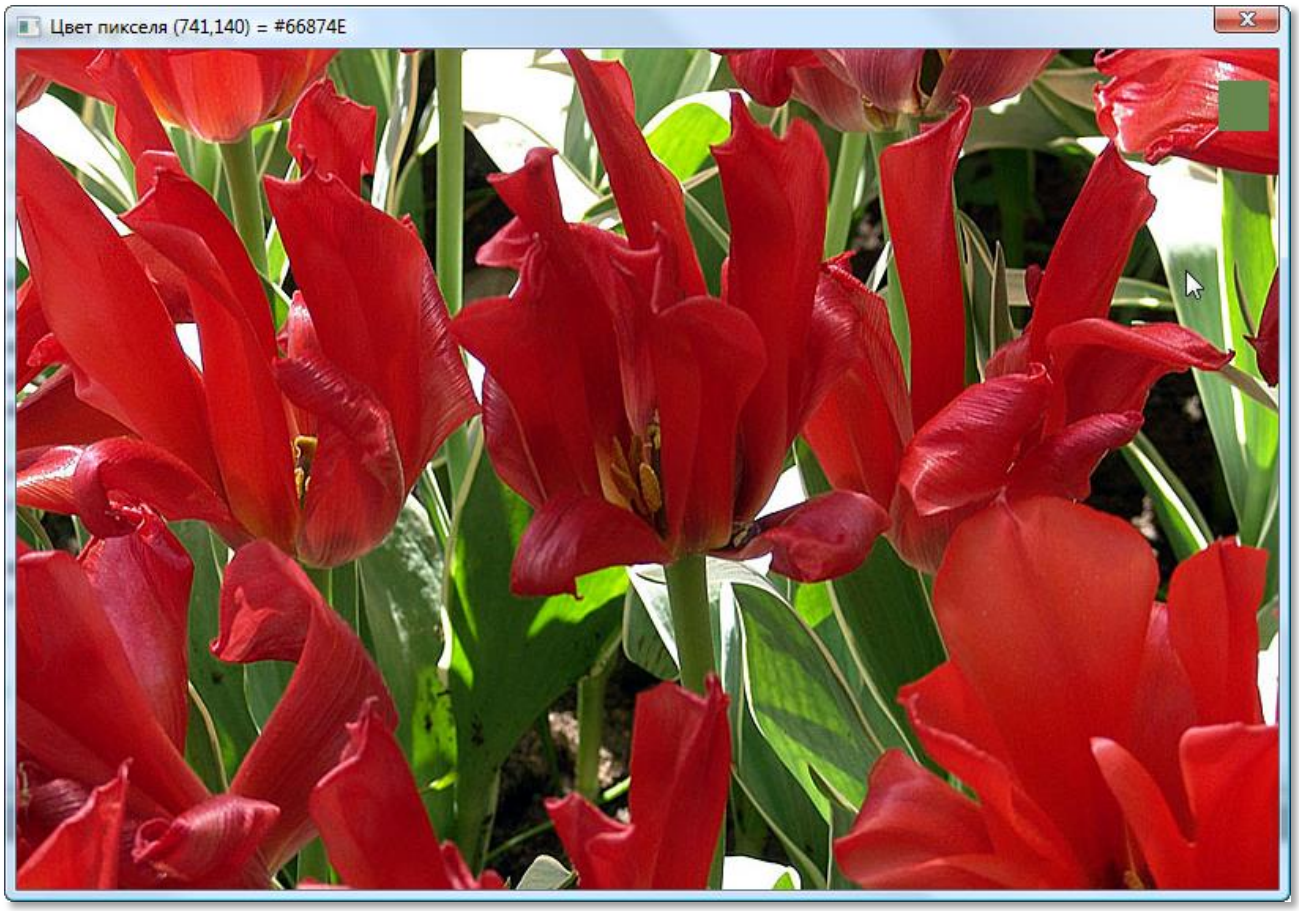


Рис. 9.3. Исходное изображение

Первый фильтр действует очень просто. Мы разбиваем всё изображение на квадратики со стороной *l* пикселей и с помощью двух циклов *for* сканируем изображение. Для каждого квадратика находим цвет пикселя в его центре, а затем всё изображение внутри квадратика заливаем этим цветом (Рис. 9.4).

```
// Фильтруем изображение
private class  procedure Filter(r: integer);
begin
   // r - радиус точки
   var l := 2 * r;
   var width := GWWIDTH;
   var height := GWHEIGHT;
   for var j := 0 to height div l do
      for var i := 0 to width div l do
      begin
          // определяем цвет пикселя в центре точки:
         var xc := i * l + r;
         var yc := j * l + r;
         var clr := GraphicsWindow.GetPixel(xc, yc);
         GraphicsWindow.BrushColor := clr;
        // чертим квадрат:
```

```
            GraphicsWindow.FillRectangle(xc - r, yc - r, l, l);
        end
end;
```

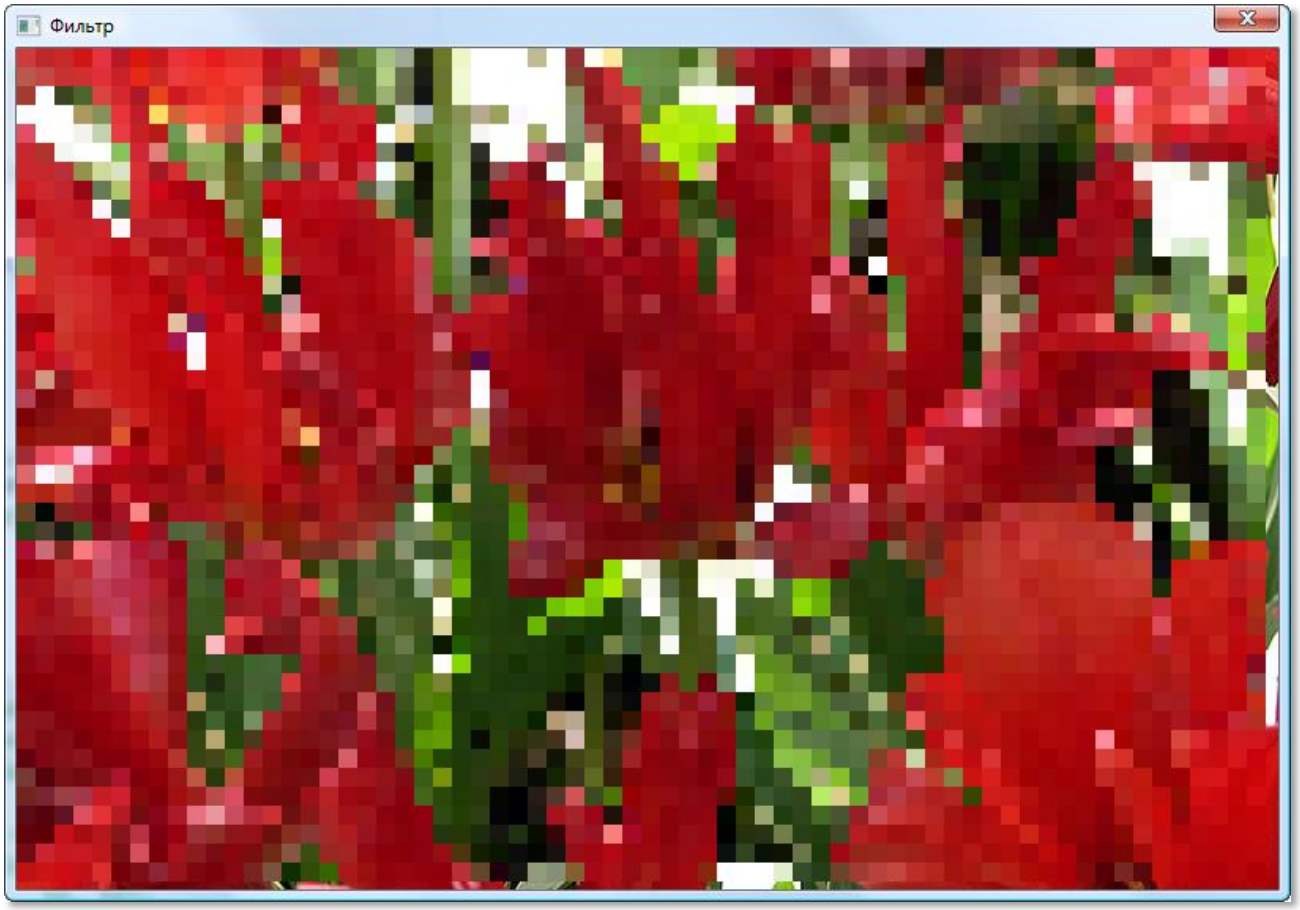


Рис. 9.4. Отфильтрованные тюльпаны

В результате изображение становится более грубым, как бы состоящим из огромных пикселей. Подобный эффект вы можете наблюдать при большом увеличении фотографий. Правда, в этом случае увеличиваются и размеры картинки, поэтому все пиксели исходного изображения остаются в целости и сохранности, наш же фильтр изменяет цвет большинства пикселей.

Второй фильтр более «изощрённый». Сначала мы запоминаем цвет пикселя в центре квадрата, затем закрашиваем его **чёрным** цветом.

> Вы можете вообще не закрашивать квадрат, или закрашивать его другими цветом. Иногда таким простым способом удаётся получить хорошие результаты!

А уже потом, «по-чёрному» рисуем круг заданного цвета:

```
private class  procedure Filter2(r: integer);
```

```
begin
   // r - радиус точки
   var l := 2 * r;
   var width := GWWIDTH;
   var height := GWHEIGHT;
   for var j := 0 to height div l do
      for var i := 0 to width div l do
      begin
          // определяем цвет пикселя в центре точки:
         var xc := i * l + r;
         var yc := j * l + r;
         var clr := GraphicsWindow.GetPixel(xc, yc);
         // рисуем квадрат:
         GraphicsWindow.BrushColor := 'Black';
         //GraphicsWindow.BrushColor := 'White';
         GraphicsWindow.FillRectangle(xc - r, yc - r, l, l);
         GraphicsWindow.BrushColor := clr;
         // рисуем круг:
         GraphicsWindow.BrushColor := clr;
         GraphicsWindow.FillEllipse(xc - r, yc - r, l, l);
      end
end;
```

Если каждый цветной кружок вышить крестиком (а лучше не полениться и вышить *двойным* крестиком), то получится прекрасная вышитая картина (Рис. 9.5). Особенно если к выбору исходного изображения подойти с полной ответственностью и бездной вкуса!

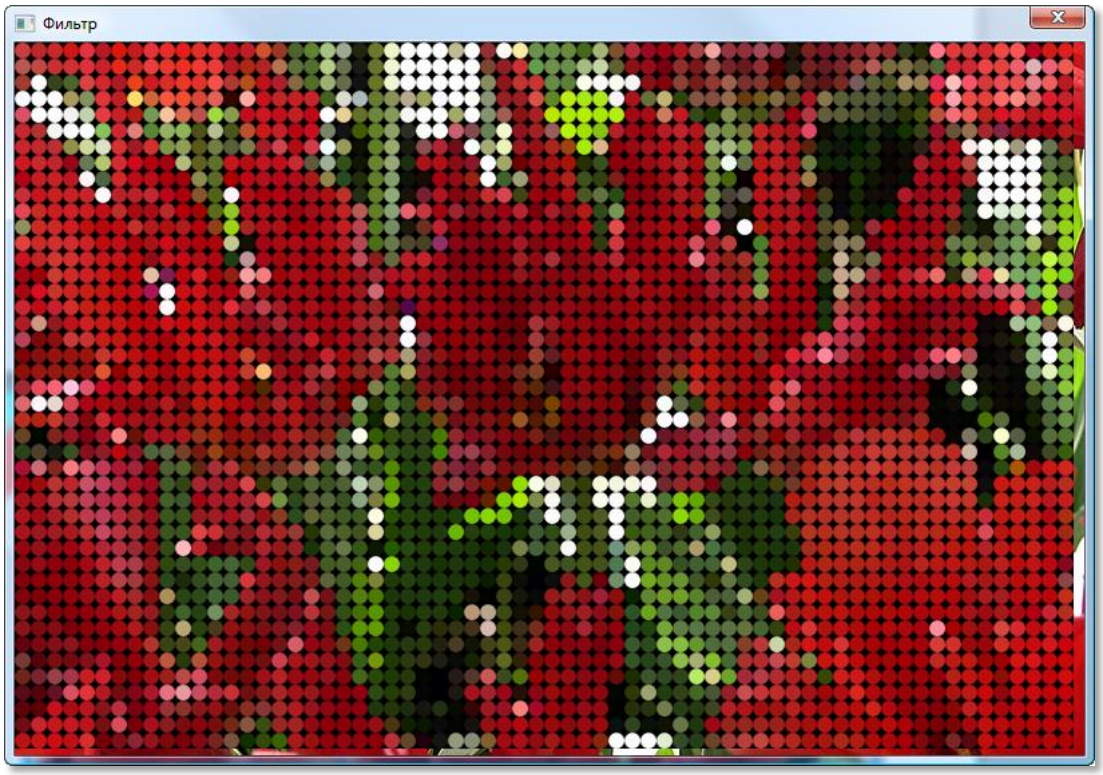


Рис. 9.5. Фильтруем тюльпаны через круглые «дырочки»

Проект *Картинки из Интернета*

Исходный код программы находится в папке **Фильтруем снимки**.

Если у вас имеется собственный сайт, то вы можете хранить картинки непосредственно на нём. Иногда такой способ хранения картинок удобнее, чем на диске, ведь Интернет доступен на любом компьютере или мобильном устройстве.

Например, я создал папку *LINQPadImage* на своём сайте и загрузил туда картинку с тюльпанами, и теперь вы можете использовать её в программе:

```
public class procedure Draw();
begin
   . . .
  //картинка из Интернета:
  //var path := 'http://rvgames.de/LINQPadImage';
  //var background := ImageList.LoadImage(path +
                                '/tulips.jpg');
  var background :=
    ImageList.LoadImage('http://rvgames.de/LINQPadImage/tulips.jpg');
  // или http://детскиекниги.рф/LINQPadImage/tulips.jpg
  GraphicsWindow.DrawImage(background, 0, 0);
  . . .
end;
```

Обратите внимание, что русские буквы лучше не использовать в названиях файлов, иначе картинка может и не загрузиться! Именно поэтому я и переименовал файл с тюльпанами.

Если путь к файлу на сайте не очень длинный, его можно **целиком** поместить в вызов метода *LoadImage*:

```
// картинка из Интернета:
var background := ImageList.LoadImage('http://rvgames.de/LINQPadImage/tulips.jpg');
```

Проект *Network*

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

Для загрузки файлов из Интернета в библиотеке *SmallBasicLibrary* имеется класс - **Network**, в котором всего 2 метода.

Метод

Network. DownloadFile(**FilePath** : string) : **string**;

загружает указанный файл любого типа во временную папку, откуда его можно распечатать в программе или скопировать в другую папку для дальнейшего использования.

Допустим, вам так полюбились мои тюльпанчики, что вы решили сохранить их на долгую память.

Поскольку вы не сможете предугадать, куда файл попадёт, то нужно **распечатать** полный путь к месту его бережного хранения:

```
{$apptype windows}

// Network

uses
   DrawUnit;

begin
   Draw.Prepare();
   Draw.Draw();
end.

unit DrawUnit;

uses
   Microsoft.SmallBasic.Library, System;

type
   Draw = class
```

```
      const GWWIDTH = 800 - 10;
      const GWHEIGHT = 533 - 10;

      public class procedure Prepare();
      begin
         GraphicsWindow.Hide();
         GraphicsWindow.Title := 'Network';
         GraphicsWindow.Width := GWWIDTH;
         GraphicsWindow.Height := GWHEIGHT;
         GraphicsWindow.Show();
         GraphicsWindow.Left := (Desktop.Width –
                                 GraphicsWindow.Width) / 2;
         GraphicsWindow.Top := (Desktop.Height –
                                GraphicsWindow.Height) / 2;
         GraphicsWindow.CanResize := false;
      end;

      public class procedure Draw();
      begin
         var width := GWWIDTH;
         var height := GWHEIGHT;

         var filePath := 'http://rvgames.de/LINQPadImage/tulips.jpg';
         var downloadFile := Network.DownloadFile(FilePath);
         Console.WriteLine('Downloaded File:' +
                           downloadFile.ToString());
         GraphicsWindow.DrawImage(downloadFile, 0, 0);
      end;
   end; //end of class

end.
```

После запуска программы картинка будет напечатана в окне приложения (Рис. 9.6)

И сохранена на диске (Рис. 9.7).

Найдя этот файл, вы должны переименовать его и задать правильное расширение, в данном случае – *jpg*. Но, естественно, файлы могут быть любого типа.

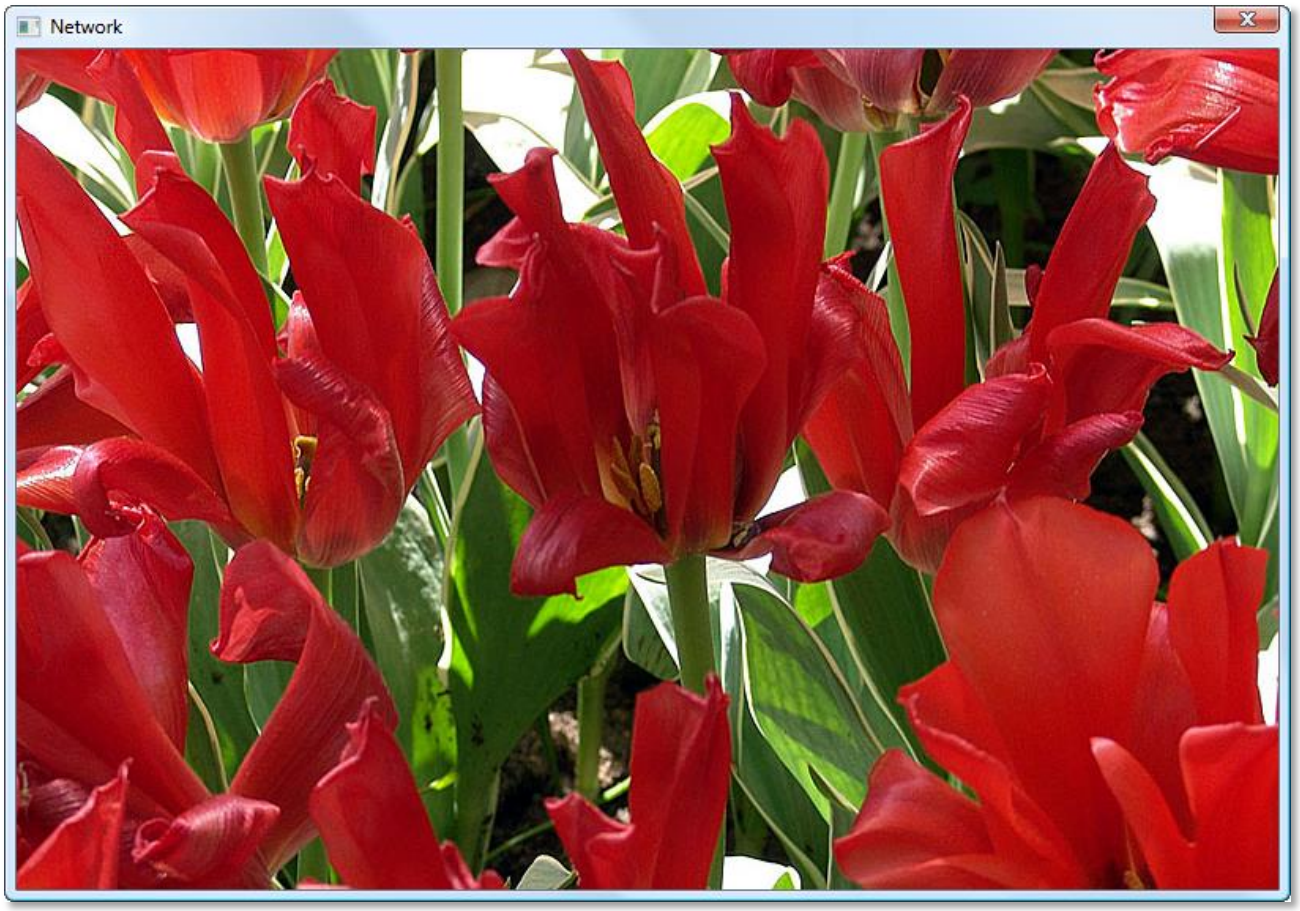


Рис. 9.6. Цветы из *Интернета*

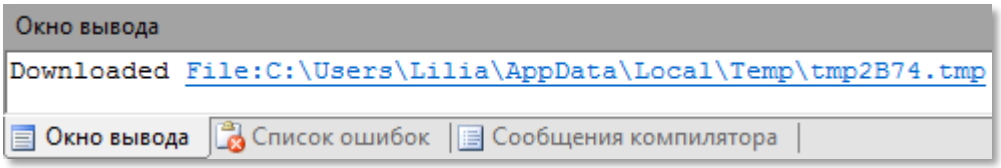


Рис. 9.7. Место хранения изменить нельзя

Проект *Network2*

Исходный код программы находится в папке **Network2**.

Второй метод класса *Network*

Network. GetWebPageContents (**FilePath** : string) : **string**;

загружает исходный код заданной страницы:

```
{$apptype windows}

// Network 2

uses
   DrawUnit;

begin
   Draw.Prepare();
   Draw.Draw();
end.

unit DrawUnit;

uses
   Microsoft.SmallBasic.Library, System;

type
   Draw = class

      const GWWIDTH = 800 - 10;
      const GWHEIGHT = 533 - 10;

      public class procedure Prepare();
      begin
         GraphicsWindow.Hide();
         GraphicsWindow.Title := 'Network';
         GraphicsWindow.Width := GWWIDTH;
         GraphicsWindow.Height := GWHEIGHT;
         GraphicsWindow.Show();
         GraphicsWindow.Left := (Desktop.Width –
                                 GraphicsWindow.Width) / 2;
         GraphicsWindow.Top := (Desktop.Height –
                                GraphicsWindow.Height) / 2;
         GraphicsWindow.CanResize := false;
      end;

      public class procedure Draw();
      begin

         var FilePath := 'http://rvgames.de/';
         var WebPageContent := Network.GetWebPageContents(FilePath);
         Console.WriteLine('Page Content: ');
         Console.WriteLine(WebPageContent.ToString());
      end;
   end; //end of class
```

```
end.
```

На Рис. 9.8 видно, что это обычный *HTML*-код.

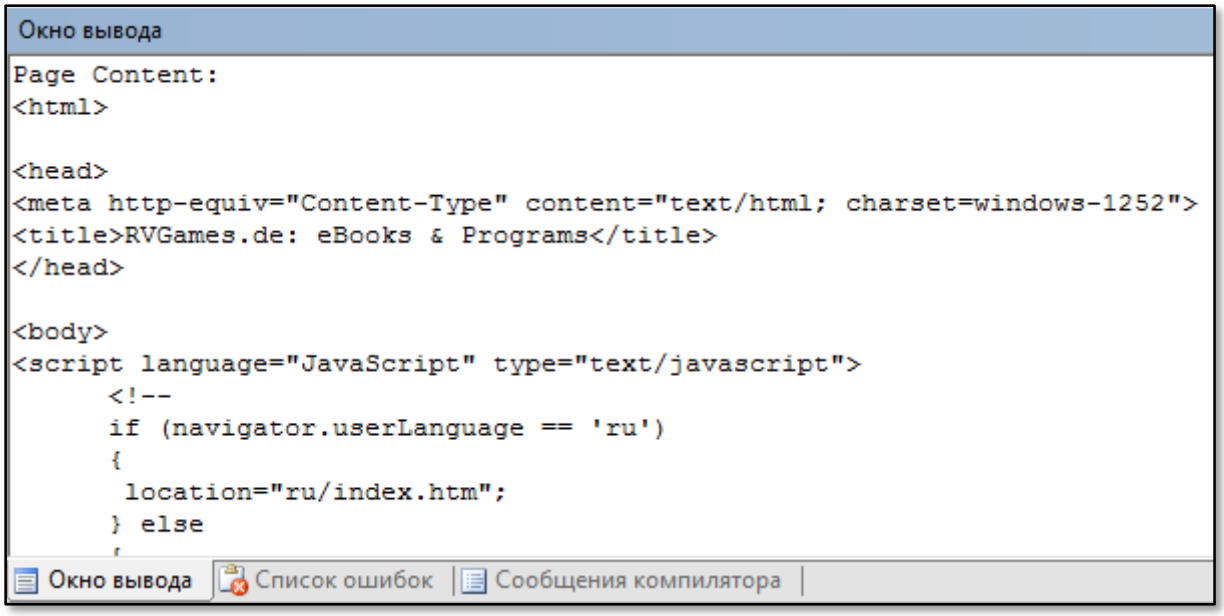


Рис. 9.8. Исходный код веб-страницы

Задания для самостоятельного решения

Фильтрование изображений

1. В программе *Фильтр* мы устанавливали цвет точки по цвету пикселя в её центре. С увеличением размеров точки отфильтрованная картинка всё больше отдаляется от оригинала, поскольку мы не учитываем цвет других пикселей этой точки. Подумайте, как **усреднить** цвет точки в фильтре.

2. Добавьте к программе метод рисования сетки после применения квадратного фильтра (Рис. 9.9):

```
private class procedure DrawGrid(r: integer);
begin
   var l := 2 * r;
   GraphicsWindow.PenColor := 'Black';
   GraphicsWindow.PenWidth := 1;
   var width := GWWIDTH;
   var height := GWHEIGHT;
   // горизонтали:
   for var j := 0 to height div l do
      GraphicsWindow.DrawLine(0, j * l, width, j * l);
   // вертикали:
   for var i := 0 to width div l do
      GraphicsWindow.DrawLine(i * l, 0, i * l, height);
end;

Filter(6);
//Filter2(6);
DrawGrid(6);
```

Рис. 9.9. С сеткой мозаика выглядит не в пример лучше!

Глава 10. Цвет в компьютерной графике

На вкус и цвет товарища нет.

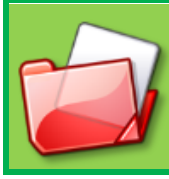
Программистская пословица

В библиотеке *SmallBasicLibrary* цвета обозначаются их английскими **названиями**. Например:

Black – чёрный
White – белый
Yellow – жёлтый
Red – красный

Названия некоторых цветов вам, возможно, вообще неизвестны. Тем более что трудно предугадать, какие из них вообще имеются в этой библиотеке. Поэтому давайте напишем проект, который наяву покажет нам, как называются все «стандартные» цвета и как они выглядят на экране.

Проект *Цветовая палитра*



Исходный код программы находится в папке **Цветовая палитра**.

Самое сложное в этом проекте – правильно пересчитать цвета и удачно распределить их в окне приложения!

У меня получился длинный список из 141 названия. Надеюсь, я ничего не пропустил и ни один цвет не записал дважды:

```
{$apptype windows}

// Colors

uses
   DrawUnit;

begin
   var draw := new Draw();
```

```
   draw.Prepare();
   draw.Draw();
end.

unit DrawUnit;

uses
   Microsoft.SmallBasic.Library, System;

type
   Draw = class

      // размеры клеток:
      const WIDTH = 60;
      const HEIGHT = 40;
      // зазор между клетками:
      const GAP = 4;
      // число клеток по горизонтали и вертикали:
      const SIZE = 12;

      COLOR := new string[] ('AliceBlue',
                             'AntiqueWhite',
                             'Aqua',
                             'Aquamarine',
                             'Azure',
                             'Beige',
                             'Bisque',
                             'Black',
                             'BlanchedAlmond',
                             'Blue',
                             'BlueViolet',
                             'Brown',
                             'BurlyWood',
                             'CadetBlue',
                             'Chartreuse',
                             'Chocolate',
                             'Coral',
                             'CornflowerBlue',
                             'Cornsilk',
                             'Crimson',
                             'Cyan',
                             'DarkBlue',
                             'DarkCyan',
                             'DarkGoldenrod',
                             'DarkGray',
                             'DarkGreen',
                             'DarkKhaki',
                             'DarkMagenta',
                             'DarkOliveGreen',
                             'DarkOrange',
                             'DarkOrchid',
```

```
                                'DarkRed',
                                'DarkSalmon',
                                'DarkSeaGreen',
                                'DarkSlateBlue',
                                'DarkSlateGray',
                                'DarkTurquoise',
                                'DarkViolet',
                                'DeepPink',
                                'DeepSkyBlue',
                                'DimGray',
                                'DodgerBlue',
                                'FireBrick',
                                'FloralWhite',
                                'ForestGreen',
                                'Fuchsia',
                                'Gainsboro',
                                'GhostWhite',
                                'Gold',
                                'Goldenrod',
                                'Gray',
                                'Green',
                                'GreenYellow',
                                'Honeydew',
                                'HotPink',
                                'IndianRed',
                                'Indigo',
                                'Ivory',
                                'Khaki',
                                'Lavender',
                                'LavenderBlush',
                                'LawnGreen',
                                'LemonChiffon',
                                'LightBlue',
                                'LightCoral',
                                'LightCyan',
                                'LightGoldenRodYellow',
                                'LightGray',
                                'LightGreen',
                                'LightPink',
                                'LightSalmon',
                                'LightSalmon',
                                'LightSeaGreen',
                                'LightSkyBlue',
                                'LightSlateGray',
                                'LightSteelBlue',
                                'LightYellow',
                                'Lime',
                                'LimeGreen',
                                'Linen',
                                'Magenta',
                                'Maroon',
```

```
                                'MediumAquamarine',
                                'MediumBlue',
                                'MediumOrchid',
                                'MediumPurple',
                                'MediumSeaGreen',
                                'MediumSlateBlue',
                                'MediumSpringGreen',
                                'MediumTurquoise',
                                'MediumVioletRed',
                                'MidnightBlue',
                                'MintCream',
                                'MistyRose',
                                'Moccasin',
                                'NavajoWhite',
                                'Navy',
                                'OldLace',
                                'Olive',
                                'OliveDrab',
                                'Orange',
                                'OrangeRed',
                                'Orchid',
                                'PaleGoldenrod',
                                'PaleGreen',
                                'PaleTurquoise',
                                'PaleVioletRed',
                                'PapayaWhip',
                                'PeachPuff',
                                'Peru',
                                'Pink',
                                'Plum',
                                'PowderBlue',
                                'Purple',
                                'Red',
                                'RosyBrown',
                                'RoyalBlue',
                                'SaddleBrown',
                                'Salmon',
                                'SandyBrown',
                                'SeaGreen',
                                'Seashell',
                                'Sienna',
                                'Silver',
                                'SkyBlue',
                                'SlateBlue',
                                'SlateGray',
                                'Snow',
                                'SpringGreen',
                                'SteelBlue',
                                'Tan',
                                'Teal',
                                'Thistle',
```

```
                              'Tomato',
                              'Turquoise',
                              'Violet',
                              'Wheat',
                              'White',
                              'WhiteSmoke',
                              'Yellow',
                              'YellowGreen'
    );
```

К сожалению, 141 цвет невозможно расположить в виде прямоугольной таблицы. Наиболее близкий результат даёт число 144 = 12 x 12. В этом случае 3 ячейки таблицы останутся пустыми.

В методе **Prepare** мы подгоняем размеры окна под размеры таблицы, чтобы не осталось ненужных пустот, но при этом не забываем оставить внизу место для печати названия цвета, который находится под курсором мышки:

```
    public procedure Prepare();
    begin
       GraphicsWindow.Hide();
       GraphicsWindow.Title := 'Colors';
       GraphicsWindow.Width := SIZE * (WIDTH + GAP) - GAP - 2;
       GraphicsWindow.Height := SIZE * (HEIGHT + GAP) - GAP + 40;
       GraphicsWindow.Show();
       GraphicsWindow.Left := (Desktop.Width –
                               GraphicsWindow.Width) / 2;
       GraphicsWindow.Top := (Desktop.Height –
                              GraphicsWindow.Height) / 2;
       GraphicsWindow.CanResize := false;
       GraphicsWindow.BackgroundColor := 'Black';
       GraphicsWindow.FontName := 'Arial';
       GraphicsWindow.FontSize := 32;
       GraphicsWindow.MouseMove += OnMouseMove;
    end;
```

Имея массив *COLOR* с названиями цветов, мы легко нарисуем **таблицу**:

```
private procedure DrawTable();
begin
   Console.WriteLine('Число цветов = ' +
                     COLOR.Count().ToString());
   for var y := 0 to SIZE - 1 do
      for var x := 0 to SIZE - 1 do
```

```
        begin
           if (y * SIZE + x < COLOR.Count()) then
              GraphicsWindow.BrushColor := COLOR[y * SIZE + x]
           else
              GraphicsWindow.BrushColor := 'Black';

           GraphicsWindow.FillRectangle(GAP + x * (WIDTH + GAP),
                                        GAP + y * (HEIGHT + GAP),
                                        WIDTH, HEIGHT);
        end;
 end;
```

При перемещении мышки вызывается метод **OnMouseMove**. Здесь мы по её координатам определяем столбец *yc* и строку *xc*, на пересечении которых находится цветная клетка с номером *nc* в массиве *COLOR*:

```
private procedure OnMouseMove();
begin
   var x := (integer)(GraphicsWindow.MouseX);
   var y := (integer)(GraphicsWindow.MouseY);
   var xc := x div (WIDTH + GAP);
   var yc := y div (HEIGHT + GAP);
   // номер цвета в массиве:
   var nc := SIZE * yc + xc;
```

Название актуального цвета мы печатаем под таблицей, и для этого всякий раз стираем предыдущую надпись:

```
var ty := (integer)(GraphicsWindow.Height) - 40;
var tx := 30;
GraphicsWindow.BrushColor := 'Black';
GraphicsWindow.FillRectangle(0, ty, GraphicsWindow.Width, 40);
```

Также мы обязательно должны учесть, что мышка может выйти и за пределы цветной таблицы, и тогда мы должны игнорировать её действия, иначе индекс цвета в массиве будет вычислен неверно, что приведёт к ошибке:

```
if (xc > SIZE - 1) or (yc > SIZE - 1) or (nc >= COLOR.Count()) then
    exit;

// печатаем название цвета:
GraphicsWindow.BrushColor := 'White';
```

```
GraphicsWindow.DrawText(tx, ty, COLOR[nc]);
```

На этом длинные подготовительные работы заканчиваются и можно запускать приложение на полную мощь! Водите мышкой по таблице, читайте и запоминайте интересные цвета (Рис. 10.1)!

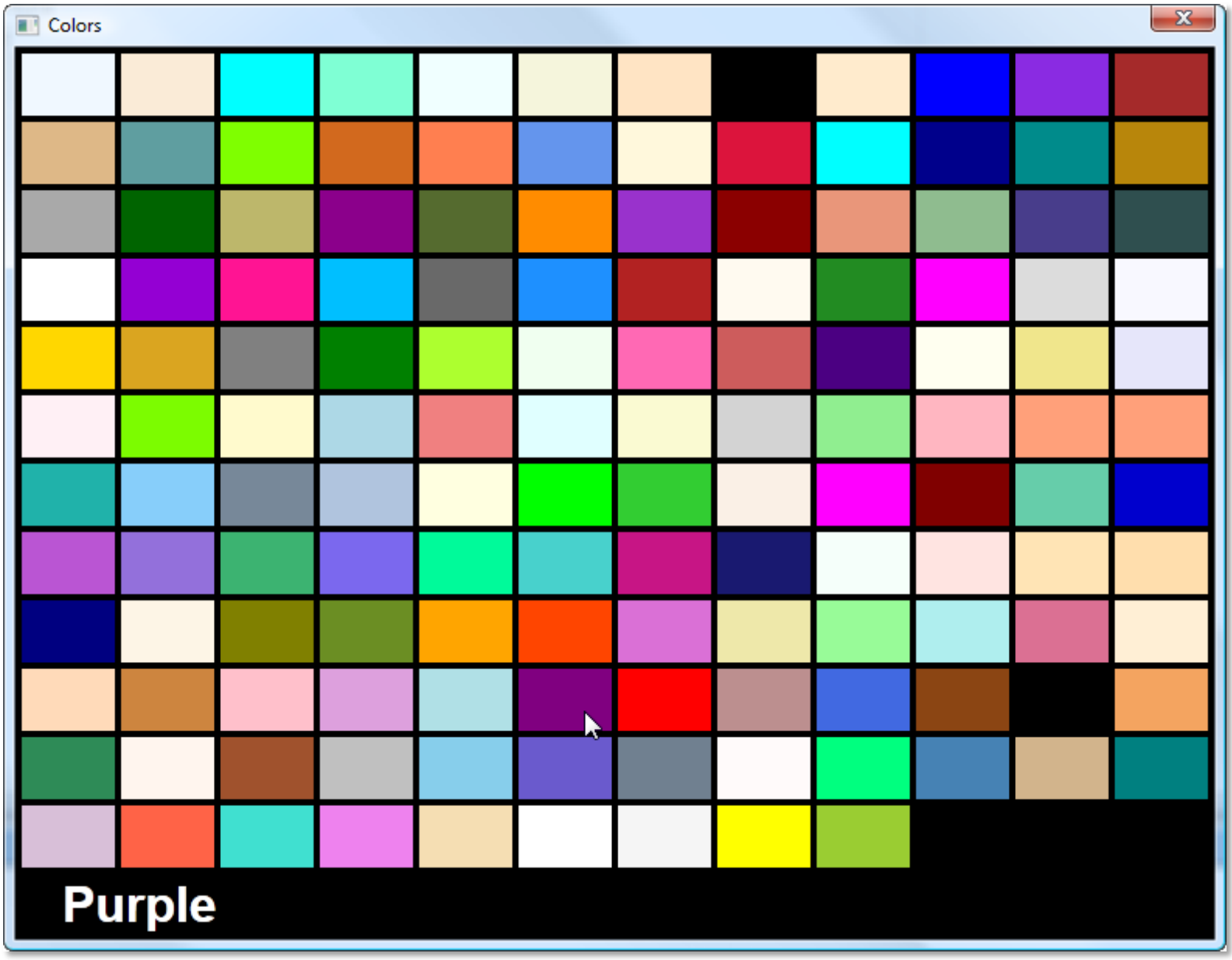


Рис. 10.1. Богатая цветовая палитра!

Проект *Случайные цвета*

Исходный код программы находится в папке **Случайные цвета**.

Названия цветов можно использовать в **свойствах** *Графического окна:*

- **BackgroundColor** - для окрашивания фона *Графического окна*
- **BrushColor -** для закрашивания геометрических фигур и символов
- **PenColor** - для задания цвета контуров фигур и линий

Однако пиксель, а, значит, и все фигуры, линии и символы могут быть окрашены в гораздо большее число цветов, чем мы рассмотрели в предыдущем проекте!

Глубина цвета

Каждый пиксель на экране занимает в памяти компьютера 32 бита памяти – по 8 битов (= 1 байту) на каждый цветной **компонент** (Рис. 10.2). Под компонентами понимают составляющие цвета:

- **A**(lpha) - прозрачность
- **R**(ed) - красная
- **G**(reen) - зелёная
- **B**(lue) - синяя

Таким образом, **глубина цвета** - 32 бита. Обычно она и выражается числом битов на один пиксель (*bits per pixel, bpp*). Чем больше это значение, тем больше цветов можно показать на экране. Качество цветопередачи зависит также от установленной на компьютере видеокарты и выбирается в диалоговом окне *Параметры дисплея*. В операционной системе *Windows Vista* оно выглядит так (Рис. 10.3).

Если развернуть список *Качество цветопередачи*, то можно найти еще одну глубину цвета, которую поддерживает видеокарта, – 16 бит на пиксель.

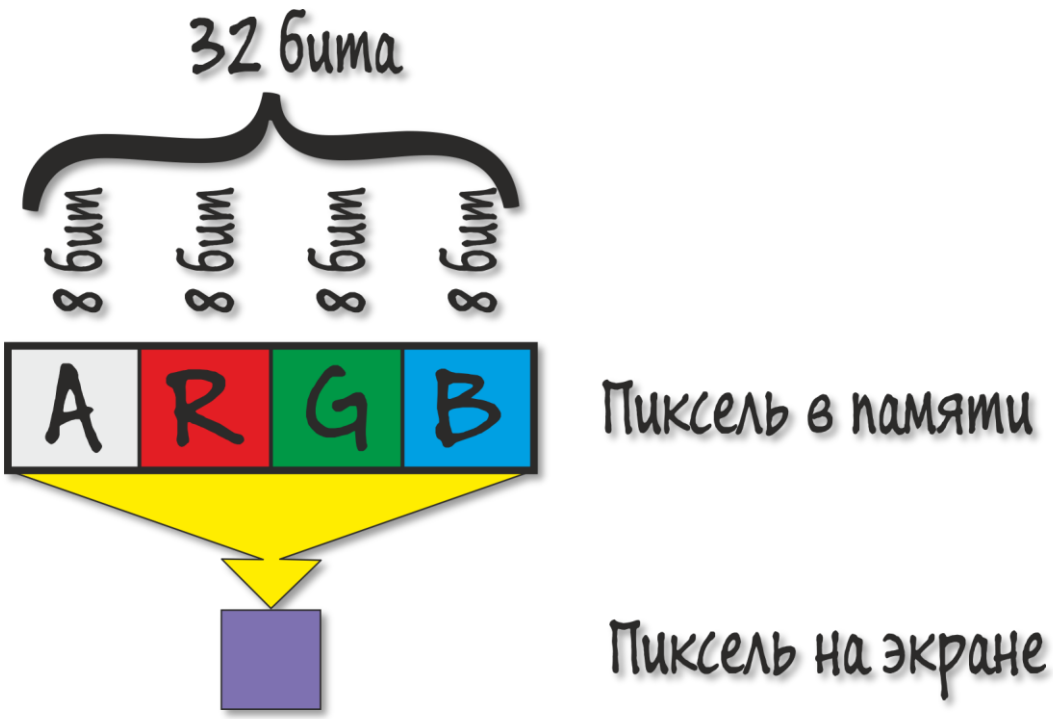


Рис. 10.2. Пиксель

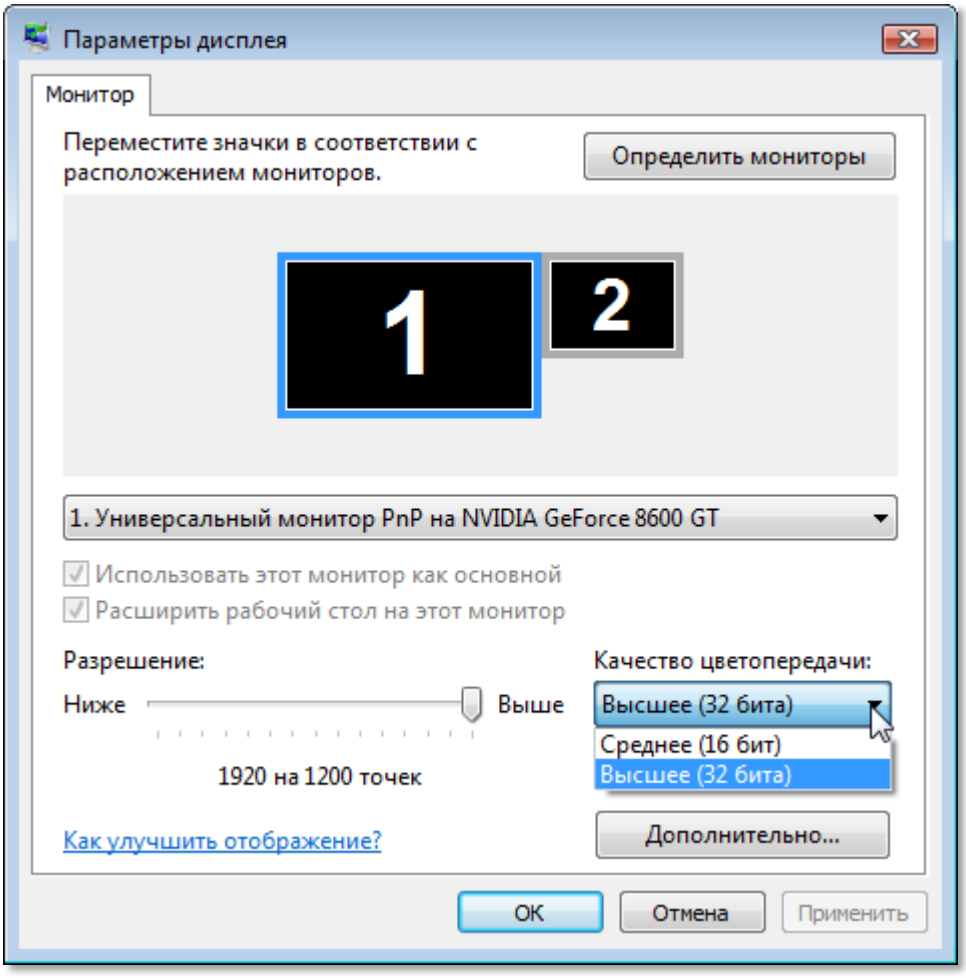


Рис. 10.3. Выбираем многоцветье!

Существуют и другие *цветовые схемы*, которые используются (или использовались раньше) значительно реже:

Глубина цвета	Описание
24 bpp	**24-битный цвет TrueColor** Для каждой составляющей цвета используется 8 бит, что дает 2^{24} = **16 777 216** цветов.
32 bpp	**32-битный цвет TrueColor** Поскольку многие современные процессоры именно 32-битные, то вполне разумно и цвет кодировать 32 битами. Для этого к 24 битам для компонентов *RGB* добавлены ещё 8 бит для хранения *альфа-канала*, который определяет *прозрачность* каждого пикселя на экране. Поскольку цвет задается всё теми же 24 битами, то количество цветов и в этой схеме также равно **16 777 216**.

Для вывода изображений на экран обычно используется цветовая модель **RGB** (Red-Green-Blue, Красный-Зеленый-Синий), которая относится к *аддитивным*.

Все цвета получаются **добавлением** (откуда и название: английское слово *addition* значит *добавление*, *сложение*) компонентов (первичных цветов) различной интенсивности к чёрному цвету. Все цвета, которые различает глаз человека, можно создать из этих трёх цветов, изменяя их интенсивность. При добавлении синего и красного цветов (полной интенсивности) мы получаем пурпурный, зелёного и красного – жёлтый, зелёного и синего – циановый. При смешивании всех цветов получается белый цвет. При отсутствии основных цветов мы получаем чёрный.

Эта цветовая модель используется для формирования изображения на экранах телевизоров и мониторов, которые излучают свет.

В библиотеке *SmallBasicLibrary* из четырёх составляющих цвета используются только три, то есть **альфа-составляющая** (прозрачность) отсутствует (но всё-таки может быть при желании добавлена).

Поскольку байты удобнее представлять в 16-ричном виде, то при задании цвета используется именно этот формат, но опять же в виде **строки**, начинающейся с решётки. Например, упомянутые выше цвета можно задать двумя способами:

Название цвета	Цвет	16-ричное значение
'Black'	чёрный	'#000000'
'White'	белый	'#FFFFFF'
'Yellow'	жёлтый	'#FFFF00'
'Red'	красный	'#FF0000'

Поэтому красный цвет фона можно установить и так:

```
GraphicsWindow.BackgroundColor := 'Red';
```

И так:

```
GraphicsWindow.BackgroundColor = '#FF0000';
```

Как вы уже знаете, класс *GraphicsWindow* имеет специальный метод **GetRandomColor** для генерирования случайного цвета.

Немного переделав предыдущий проект, мы заполним всю таблицу случайными цветами, а затем с помощью мышки узнаем 16-ричное значение любого из них – тоже неплохой способ для выбора цветов для своих проектов:

```
public procedure Prepare();
begin
   GraphicsWindow.Hide();
   GraphicsWindow.Title := 'Random Colors';
   . . .
   // создаём массив случайных цветов:
   CreateTable();
end;
```

```
private procedure CreateTable();
begin
   for var y := 0 to SIZE - 1 do
      for var x := 0 to SIZE - 1 do
         COLOR[y * SIZE + x] := GraphicsWindow.GetRandomColor();
end;
```

При каждом запуске программы будет формироваться новая таблица (Рис. 10.4). Путешествуя по ней с мышкой в руке, вы живо изучите все компьютерные цвета!

Рис. 10.4. Случайные цвета

Проект *Цвет своими руками*

Исходный код программы находится в папке **Цвет своими руками**.

Как говорили герои комедии *Ирония судьбы, или С лёгким паром*, мы не станем полагаться на случай, а будем самостоятельно задавать цвет по собственному выбору. Это легко сделать с привлечением метода **GetColorFromRGB**, которому следует передать три составляющие цвета. Он же в благодарность вернёт нам полноценный цвет, собранный из этих компонентов.

Для ввода данных нам понадобятся 3 **текстовых окна**:

```
/ /составляющие цвета:
txtR: string;
txtG: string;
txtB: string;
```

Нам, конечно, удобнее вводить десятичные значения – от 0 до 255, а в программах используются 16-ричные значения, для которых мы отведём ещё 3 **текстовых поля**:

```
txtRH: string;
txtGH: string;
txtBH: string;
```

Вводить в эти текстовые поля ничего не нужно. Наоборот, мы будем печатать в них соответствующие 16-ричные значения составляющих цвета.

И последний элемент управления нашей программы – **кнопка**:

```
// кнопка:
btnCreate: string;
```

После нажатия на неё канва окрасится в заданный цвет.

В методе **Prepare** мы, как обычно, настраиваем окно приложения:

```
public procedure Prepare();
begin
   GraphicsWindow.Hide();
   GraphicsWindow.Title := 'Create Colors';
   GraphicsWindow.Width := GWWIDTH;
   GraphicsWindow.Height := GWHEIGHT;
   GraphicsWindow.Show();
   GraphicsWindow.Left := (Desktop.Width –
                           GraphicsWindow.Width) / 2;
   GraphicsWindow.Top := (Desktop.Height –
                          GraphicsWindow.Height) / 2;
   GraphicsWindow.CanResize := false;
   GraphicsWindow.BackgroundColor := 'Black';
end;
```

Если элементов управления много, то лучше весь код, связанный с их созданием, вынести в **отдельный метод**:

```
// создаём элементы управления:
public procedure CreateGUI();
begin
   GraphicsWindow.FontName := 'Arial';
   GraphicsWindow.FontSize := 30;

   var x := 10;
   var y := 10;
   var w := 70;
   var h := 42;
   var dy := h + 12;
   txtR := Controls.AddMultiLineTextBox(x, y);
   Controls.SetSize(txtR, w, h);

   txtRH := Controls.AddMultiLineTextBox(x + w + 12, y);
   Controls.SetSize(txtRH, w, h);

   y += dy;
   txtG := Controls.AddMultiLineTextBox(x, y);
   Controls.SetSize(txtG, w, h);
   txtGH := Controls.AddMultiLineTextBox(x + w + 12, y);
   Controls.SetSize(txtGH, w, h);

   y += dy;
   txtB := Controls.AddMultiLineTextBox(x, y);
   Controls.SetSize(txtB, w, h);
   txtBH := Controls.AddMultiLineTextBox(x + w + 12, y);
   Controls.SetSize(txtBH, w, h);
```

```
    GraphicsWindow.FontSize := 18;
    y += dy;
    btnCreate := Controls.AddButton('Цвет!', x, y);
    Controls.SetSize(btnCreate, w, h);
    Controls.ButtonClicked += OnClick;
end;
```

И вот пользователь ввёл в три левых поля значения составляющих цвета и нажал кнопку. В методе **OnClick** мы считываем текст в строковую переменную *s*, которую отсылаем методу *GetComp*, который возвращает правильное числовое значение заданной строки:

```
private procedure OnClick();
begin
   var s: string := Controls.GetTextBoxText(txtR);
   var r := GetComp(s);
   Controls.SetTextBoxText(txtRH, string.Format('#{0:X2}', r));

   s := Controls.GetTextBoxText(txtG);
   var g := GetComp(s);
   Controls.SetTextBoxText(txtGH, string.Format('#{0:X2}', g));

   s := Controls.GetTextBoxText(txtB);
   var b := GetComp(s);
   Controls.SetTextBoxText(txtBH, string.Format('#{0:X2}', b));
```

Получив составляющие цвета и распечатав их в 16-ричном виде в правых текстовых полях, мы собираем из них цвет, в который и окрашиваем окно приложения:

```
   var clr := GraphicsWindow.GetColorFromRGB(r, g, b);
   GraphicsWindow.BackgroundColor := clr;
end;
```

Метод **GetComp** получает строку, состоящую из цифр (если это не так, то программа будет испорчена!), превращает её в число, отбрасывает возможный знак минус и «вбивает» его в допустимый диапазон значений для составляющих цвета 0..255:

```
private function GetComp(s: string): integer;
begin
```

```
   Result := 0;
   if (s <> String.Empty) then
   begin
      Result := Int32.Parse(s);
      Result := System.Math.Abs(Result mod 256);
   end;
end;
```

Теперь вы можете самостоятельно перемешивать краски, сгущать их и окрашивать канву во все цвета радуги (Рис. 10.5)!

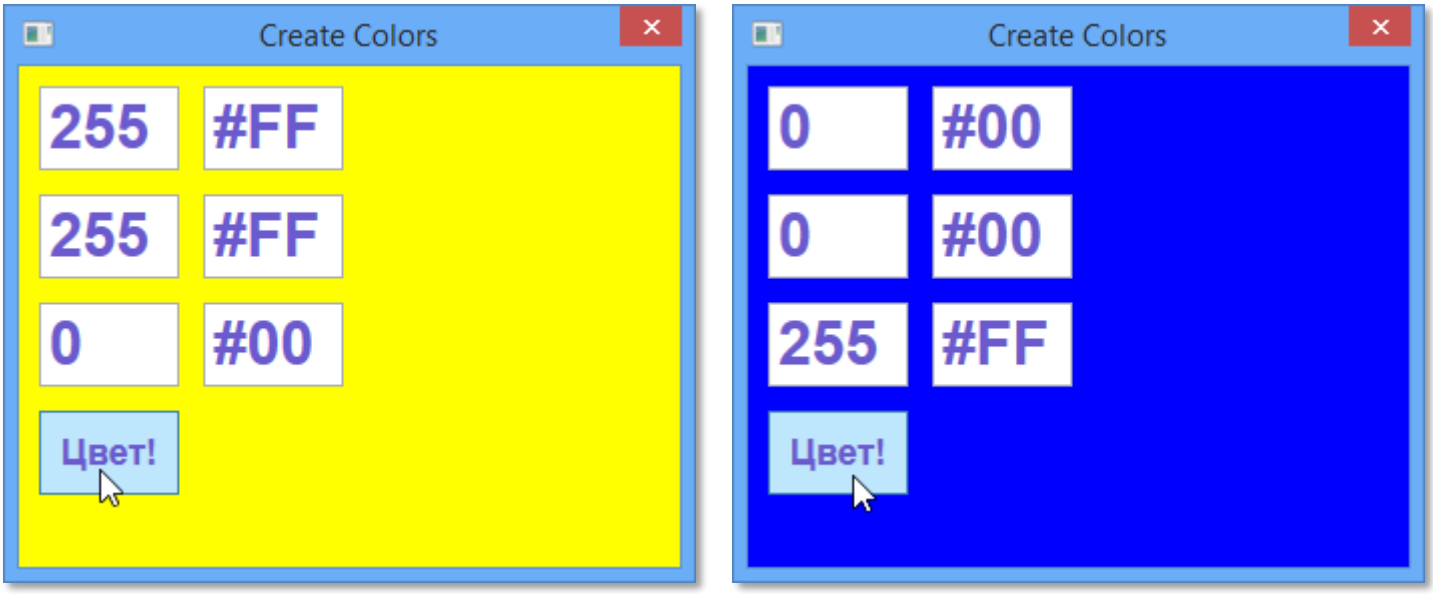


Рис. 10.5. Компьютерный краскосмеситель

Проект *Библиотека цветов*

Исходный код программы находится в папке **Библиотека цветов**.

Начните новый проект и добавьте к нему, как было описано выше, библиотеку **Colours.dll** и пространство имён **Colours**.

Вся библиотека состоит из огромного числа **свойств**, которые возвращают 16-ричное представление цвета. Поскольку имя свойства совпадает с названием цвета, то пользоваться этими свойствами удобно и просто!

Для примера окрасим окно в красный цвет (Рис. 10.6):

```
{$apptype windows}
// Colors DLL

uses
   DrawUnit;

begin
   var draw := new Draw();
   draw.Prepare();
end.

unit DrawUnit;
uses
   Microsoft.SmallBasic.Library, Colours, System;

type
   Draw = class
      const GWWIDTH = 320;
      const GWHEIGHT = 240;

      public procedure Prepare();
      begin
         GraphicsWindow.Hide();
         GraphicsWindow.Title := 'Colors DLL';
         GraphicsWindow.Width := GWWIDTH;
         GraphicsWindow.Height := GWHEIGHT;
         GraphicsWindow.Show();
         GraphicsWindow.Left := (Desktop.Width –
                                 GraphicsWindow.Width) / 2;
         GraphicsWindow.Top := (Desktop.Height –
                                 GraphicsWindow.Height) / 2;
         GraphicsWindow.CanResize := false;
```

```
        Console.WriteLine(Colour.Red.ToString());
        GraphicsWindow.BackgroundColor := Colour.Red;
      end;
   end; //end of class
end.
```

Рис. 10.6. Красный цвет – проблем нет!

В *Окне вывода* можно прочитать 16-ричное значение выбранного цвета (Рис. 10.7).

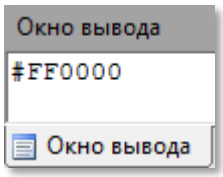


Рис. 10.7. 16-ричное значение цвета

Для удобства пользования названия всех свойств класса *Colour* и 16-ричные значения цветов сведены в **таблицу**:

Red Colors	
IndianRed	'#CD5C5C'
LightCoral	'#F08080'
Salmon	'#FA8072'
DarkSalmon	'#E9967A'
LightSalmon	'#FFA07A'
Crimson	'#DC143C'
Red	'#FF0000'
FireBrick	'#B22222'
DarkRed	'#8B0000'

Pink Colors	
Pink	'#FFC0CB'
LightPink	'#FFB6C1'
HotPink	'#FF69B4'
DeepPink	'#FF1493'
MediumVioletRed	'#C71585'
PaleVioletRed	'#DB7093'
Orange-Yellow Colors	
LightSalmon	'#FFA07A'
Coral	'#FF7F50'
Tomato	'#FF6347'
OrangeRed	'#FF4500'
DarkOrange	'#FF8C00'
Orange	'#FFA500'
Gold	'#FFD700'
Yellow	'#FFFF00'
LightYellow	'#FFFFE0'
LemonChiffon	'#FFFACD'
LightGoldenrodYellow	'#FAFAD2'
PapayaWhip	'#FFEFD5'
Moccasin	'#FFE4B5'
PeachPuff	'#FFDAB9'
PaleGoldenrod	'#EEE8AA'
Khaki	'#F0E68C'
DarkKhaki	'#BDB76B'
Purple Colors	
Lavender	'#E6E6FA'
Thistle	'#D8BFD8'
Plum	'#DDA0DD'
Violet	'#EE82EE'
Orchid	'#DA70D6'
Fuchsia	'#FF00FF'
Magenta	'#FF00FF'
MediumOrchid	'#BA55D3'
MediumPurple	'#9370DB'
BlueViolet	'#8A2BE2'
DarkViolet	'#9400D3'
DarkOrchid	'#9932CC'
DarkMagenta	'#8B008B'
Purple	'#800080'
Indigo	'#4B0082'
SlateBlue	'#6A5ACD'
DarkSlateBlue	'#483D8B'

MediumSlateBlue	'#7B68EE'
Green Colors	
GreenYellow	'#ADFF2F'
Chartreuse	'#7FFF00'
LawnGreen	'#7CFC00'
Lime	'#00FF00'
LimeGreen	'#32CD32'
PaleGreen	'#98FB98'
LightGreen	'#90EE90'
MediumSpringGreen	'#00FA9A'
SpringGreen	'#00FF7F'
MediumSeaGreen	'#3CB371'
SeaGreen	'#2E8B57'
ForestGreen	'#228B22'
Green	'#008000'
DarkGreen	'#006400'
YellowGreen	'#9ACD32'
OliveDrab	'#6B8E23'
Olive	'#808000'
DarkOliveGreen	'#556B2F'
MediumAquamarine	'#66CDAA'
DarkSeaGreen	'#8FBC8F'
LightSeaGreen	'#20B2AA'
DarkCyan	'#008B8B'
Teal	'#008080'
Blue Colors	
Aqua	'#00FFFF'
Cyan	'#00FFFF'
LightCyan	'#E0FFFF'
PaleTurquoise	'#AFEEEE'
Aquamarine	'#7FFFD4'
Turquoise	'#40E0D0'
MediumTurquoise	'#48D1CC'
DarkTurquoise	'#00CED1'
CadetBlue	'#5F9EA0'
SteelBlue	'#4682B4'
LightSteelBlue	'#B0C4DE'
PowderBlue	'#B0E0E6'
LightBlue	'#ADD8E6'
SkyBlue	'#87CEEB'
LightSkyBlue	'#87CEFA'
DeepSkyBlue	'#00BFFF'
DodgerBlue	'#1E90FF'

CornflowerBlue	'#6495ED'
MediumSlateBlue	'#7B68EE'
RoyalBlue	'#4169E1'
Blue	'#0000FF'
MediumBlue	'#0000CD'
DarkBlue	'#00008B'
Navy	'#000080'
MidnightBlue	'#191970'
Brown Colors	
Cornsilk	'#FFF8DC'
BlanchedAlmond	'#FFEBCD'
Bisque	'#FFE4C4'
NavajoWhite	'#FFDEAD'
Wheat	'#F5DEB3'
BurlyWood	'#DEB887'
Tan	'#D2B48C'
RosyBrown	'#BC8F8F'
SandyBrown	'#F4A460'
Goldenrod	'#DAA520'
DarkGoldenrod	'#B8860B'
Peru	'#CD853F'
Chocolate	'#D2691E'
SaddleBrown	'#8B4513'
Sienna	'#A0522D'
Brown	'#A52A2A'
Maroon	'#800000'
White Colors	
White	'#FFFFFF'
Snow	'#FFFAFA'
Honeydew	'#F0FFF0'
MintCream	'#F5FFFA'
Azure	'#F0FFFF'
AliceBlue	'#F0F8FF'
GhostWhite	'#F8F8FF'
WhiteSmoke	'#F5F5F5'
Seashell	'#FFF5EE'
Beige	'#F5F5DC'
OldLace	'#FDF5E6'
FloralWhite	'#FFFAF0'
Ivory	'#FFFFF0'
AntiqueWhite	'#FAEBD7'
Linen	'#FAF0E6'
LavenderBlush	'#FFF0F5'

MistyRose	'#FFE4E1'
Gray Colors	
Gainsboro	'#DCDCDC'
LightGray	'#D3D3D3'
Silver	'#C0C0C0'
DarkGray	'#A9A9A9'
Gray	'#808080'
DimGray	'#696969'
LightSlateGray	'#778899'
SlateGray	'#708090'
DarkSlateGray	'#2F4F4F'
Black	'#000000'

Все цвета разбиты по категориям, так что выбрать нужный цвет вам будет проще.

Очень наглядная **таблица** цветов представлена на сайте *msdn.microsoft.com*:

AliceBlue	#FFF0F8FF	DarkTurquoise	#FF00CED1	LightSeaGreen	#FF20B2AA	PapayaWhip	#FFFFEFD5
AntiqueWhite	#FFFAEBD7	DarkViolet	#FF9400D3	LightSkyBlue	#FF87CEFA	PeachPuff	#FFFFDAB9
Aqua	#FF00FFFF	DeepPink	#FFFF1493	LightSlateGray	#FF778899	Peru	#FFCD853F
Aquamarine	#FF7FFFD4	DeepSkyBlue	#FF00BFFF	LightSteelBlue	#FFB0C4DE	Pink	#FFFFC0CB
Azure	#FFF0FFFF	DimGray	#FF696969	LightYellow	#FFFFFFE0	Plum	#FFDDA0DD
Beige	#FFF5F5DC	DodgerBlue	#FF1E90FF	Lime	#FF00FF00	PowderBlue	#FFB0E0E6
Bisque	#FFFFE4C4	Firebrick	#FFB22222	LimeGreen	#FF32CD32	Purple	#FF800080
Black	#FF000000	FloralWhite	#FFFFFAF0	Linen	#FFFAF0E6	Red	#FFFF0000
BlanchedAlmond	#FFFFEBCD	ForestGreen	#FF228B22	Magenta	#FFFF00FF	RosyBrown	#FFBC8F8F
Blue	#FF0000FF	Fuchsia	#FFFF00FF	Maroon	#FF800000	RoyalBlue	#FF4169E1
BlueViolet	#FF8A2BE2	Gainsboro	#FFDCDCDC	MediumAquamarine	#FF66CDAA	SaddleBrown	#FF8B4513
Brown	#FFA52A2A	GhostWhite	#FFF8F8FF	MediumBlue	#FF0000CD	Salmon	#FFFA8072
BurlyWood	#FFDEB887	Gold	#FFFFD700	MediumOrchid	#FFBA55D3	SandyBrown	#FFF4A460
CadetBlue	#FF5F9EA0	Goldenrod	#FFDAA520	MediumPurple	#FF9370DB	SeaGreen	#FF2E8B57
Chartreuse	#FF7FFF00	Gray	#FF808080	MediumSeaGreen	#FF3CB371	SeaShell	#FFFFF5EE
Chocolate	#FFD2691E	Green	#FF008000	MediumSlateBlue	#FF7B68EE	Sienna	#FFA0522D
Coral	#FFFF7F50	GreenYellow	#FFADFF2F	MediumSpringGreen	#FF00FA9A	Silver	#FFC0C0C0
CornflowerBlue	#FF6495ED	Honeydew	#FFF0FFF0	MediumTurquoise	#FF48D1CC	SkyBlue	#FF87CEEB
Cornsilk	#FFFFF8DC	HotPink	#FFFF69B4	MediumVioletRed	#FFC71585	SlateBlue	#FF6A5ACD
Crimson	#FFDC143C	IndianRed	#FFCD5C5C	MidnightBlue	#FF191970	SlateGray	#FF708090
Cyan	#FF00FFFF	Indigo	#FF4B0082	MintCream	#FFF5FFFA	Snow	#FFFFFAFA
DarkBlue	#FF00008B	Ivory	#FFFFFFF0	MistyRose	#FFFFE4E1	SpringGreen	#FF00FF7F
DarkCyan	#FF008B8B	Khaki	#FFF0E68C	Moccasin	#FFFFE4B5	SteelBlue	#FF4682B4
DarkGoldenrod	#FFB8860B	Lavender	#FFE6E6FA	NavajoWhite	#FFFFDEAD	Tan	#FFD2B48C
DarkGray	#FFA9A9A9	LavenderBlush	#FFFFF0F5	Navy	#FF000080	Teal	#FF008080
DarkGreen	#FF006400	LawnGreen	#FF7CFC00	OldLace	#FFFDF5E6	Thistle	#FFD8BFD8
DarkKhaki	#FFBDB76B	LemonChiffon	#FFFFFACD	Olive	#FF808000	Tomato	#FFFF6347
DarkMagenta	#FF8B008B	LightBlue	#FFADD8E6	OliveDrab	#FF6B8E23	Transparent	#00FFFFFF
DarkOliveGreen	#FF556B2F	LightCoral	#FFF08080	Orange	#FFFFA500	Turquoise	#FF40E0D0
DarkOrange	#FFFF8C00	LightCyan	#FFE0FFFF	OrangeRed	#FFFF4500	Violet	#FFEE82EE
DarkOrchid	#FF9932CC	LightGoldenrodYellow	#FFFAFAD2	Orchid	#FFDA70D6	Wheat	#FFF5DEB3
DarkRed	#FF8B0000	LightGray	#FFD3D3D3	PaleGoldenrod	#FFEEE8AA	White	#FFFFFFFF
DarkSalmon	#FFE9967A	LightGreen	#FF90EE90	PaleGreen	#FF98FB98	WhiteSmoke	#FFF5F5F5
DarkSeaGreen	#FF8FBC8F	LightPink	#FFFFB6C1	PaleTurquoise	#FFAFEEEE	Yellow	#FFFFFF00
DarkSlateBlue	#FF483D8B	LightSalmon	#FFFFA07A	PaleVioletRed	#FFDB7093	YellowGreen	#FF9ACD32
DarkSlateGray	#FF2F4F4F						

Проект *Игра ПараЦвет*

Исходный код программы находится в папке **Игра ПараЦвет**.

А.С. Пушкин, *Сказка о попе и работнике его Балде*

Эта игра - не для дальтоников! Прямоугольная таблица 8 х 8 клеток составлена из 32 пар одинаково окрашенных в стандартные цвета клеток. **Задача** незадачливого игрока заключается в том, чтобы отыскать и отщёлкать *все пары клеток*.

Для этого нужно навести курсор мышки на **первую** клетку (Рис. 10.8) и щёлкнуть её. Клетка будет выделена **красным** контуром (Рис. 10.9).

Если вы хотите снять выделение с первой клетки, кликните её ещё раз.

Теперь нужно найти в таблице вторую клетку точно такого же цвета (Рис. 10.10) и щёлкнуть её. В тренировочном режиме под таблицей печатаются названия цветов, так что с поисками парной клетки проблем не будет. Если же вы ошибётесь и щёлкнете клетку другого цвета, то звуковой сигнал оповестит вас об этой неудаче, и вы сможете выбрать другую клетку.

Надо сказать, что многие цвета практически неотличимы друг от друга, поэтому несколько цветов я исключил из массива *COLOR*. Если вас преследуют неудачи при выборе цветов, то сократите этот массив ещё больше.

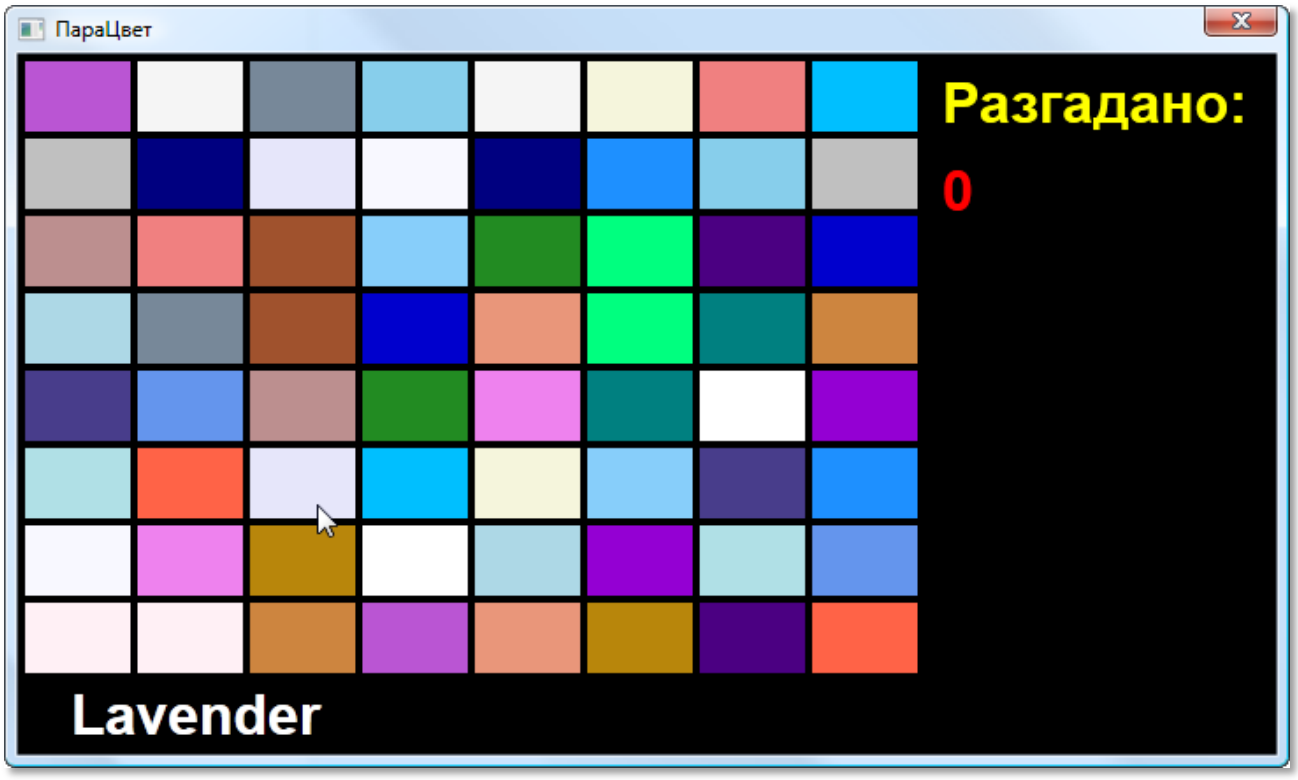


Рис. 10.8. Пора начать!

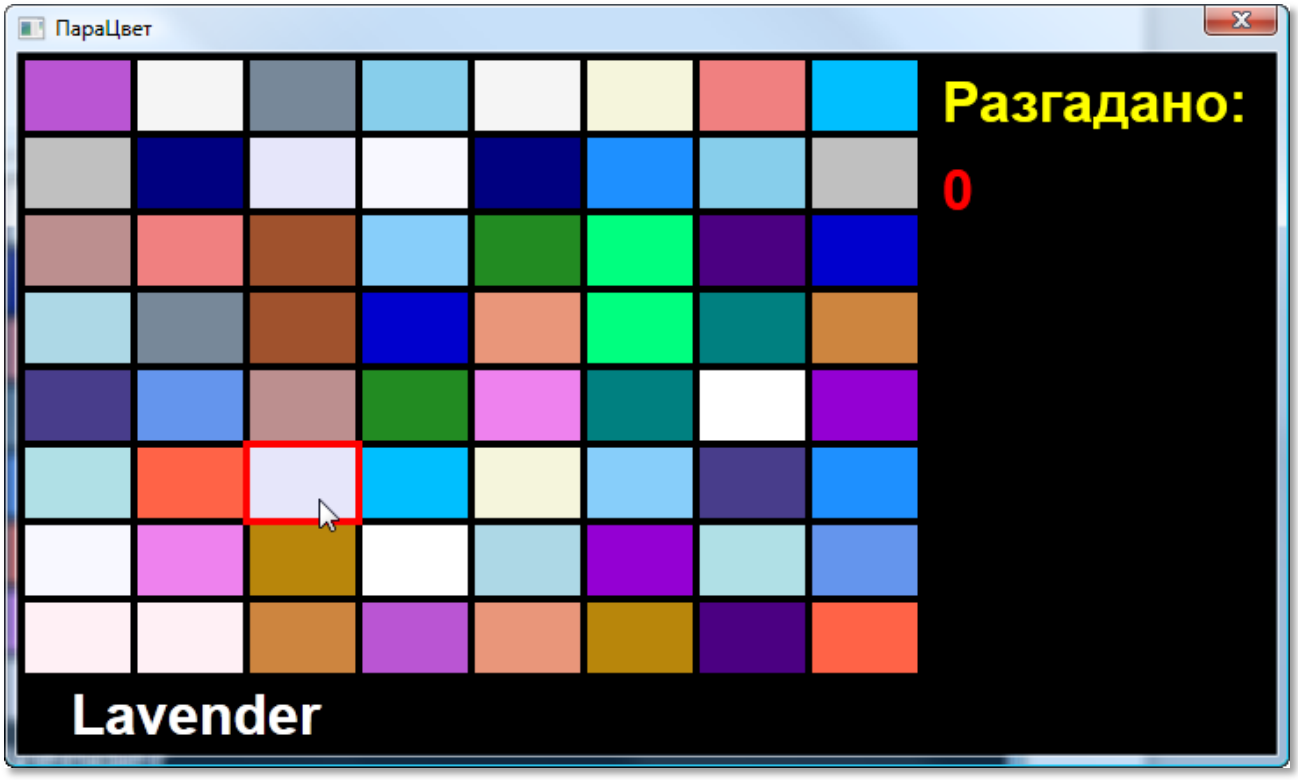


Рис. 10.9. С первым щелчком!

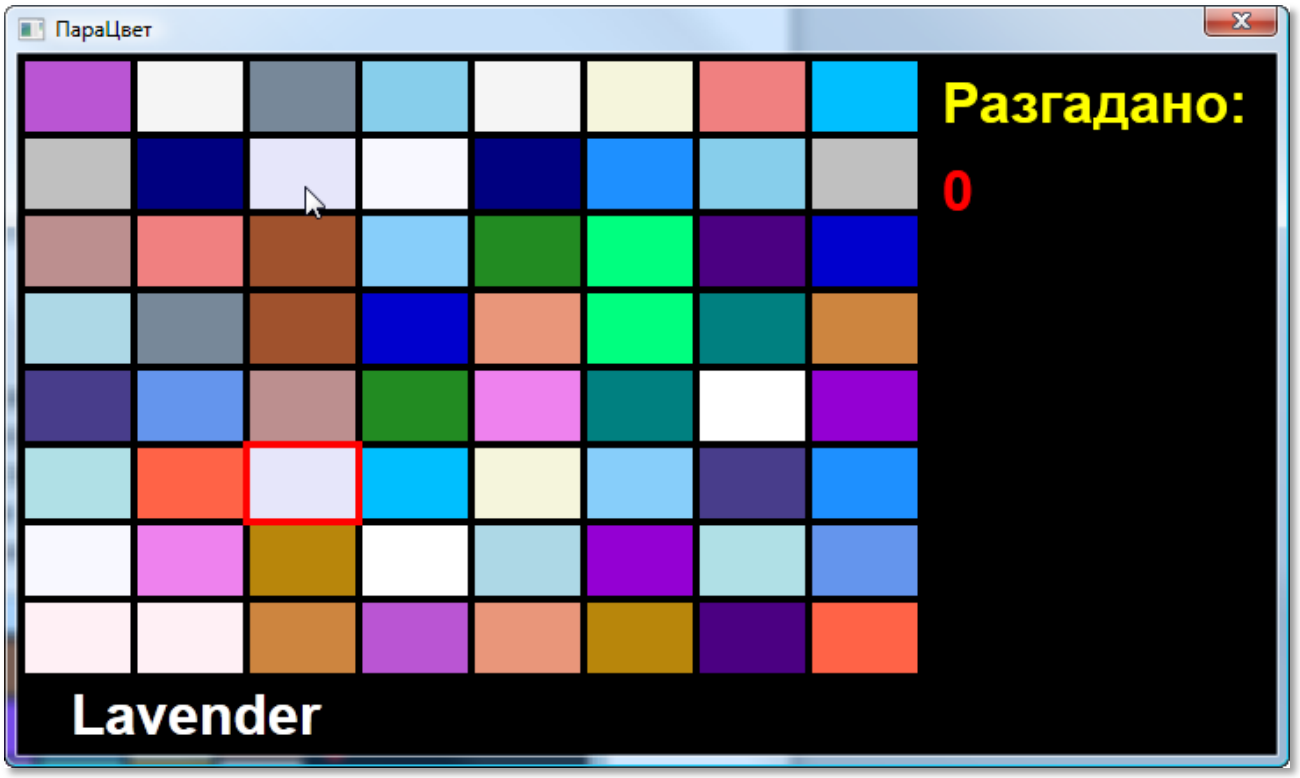


Рис. 10.10. Выделенная клетка

Если пара цветов подобрана верно, то две клетки перекрасятся в **чёрный** цвет и выйдут из дальнейшей игры, а ваш лицевой счёт пополнится двумя очками (Рис. 10.11).

Рис. 10.11. Глаз не подвёл!

Дальше всё то же самое: находите пару цветов и выщёлкиваете их из игры (Рис. 10.12).

Когда на поле останется совсем немного цветных клеток, игра пойдёт бойчее и веселее (Рис. 10.13)!

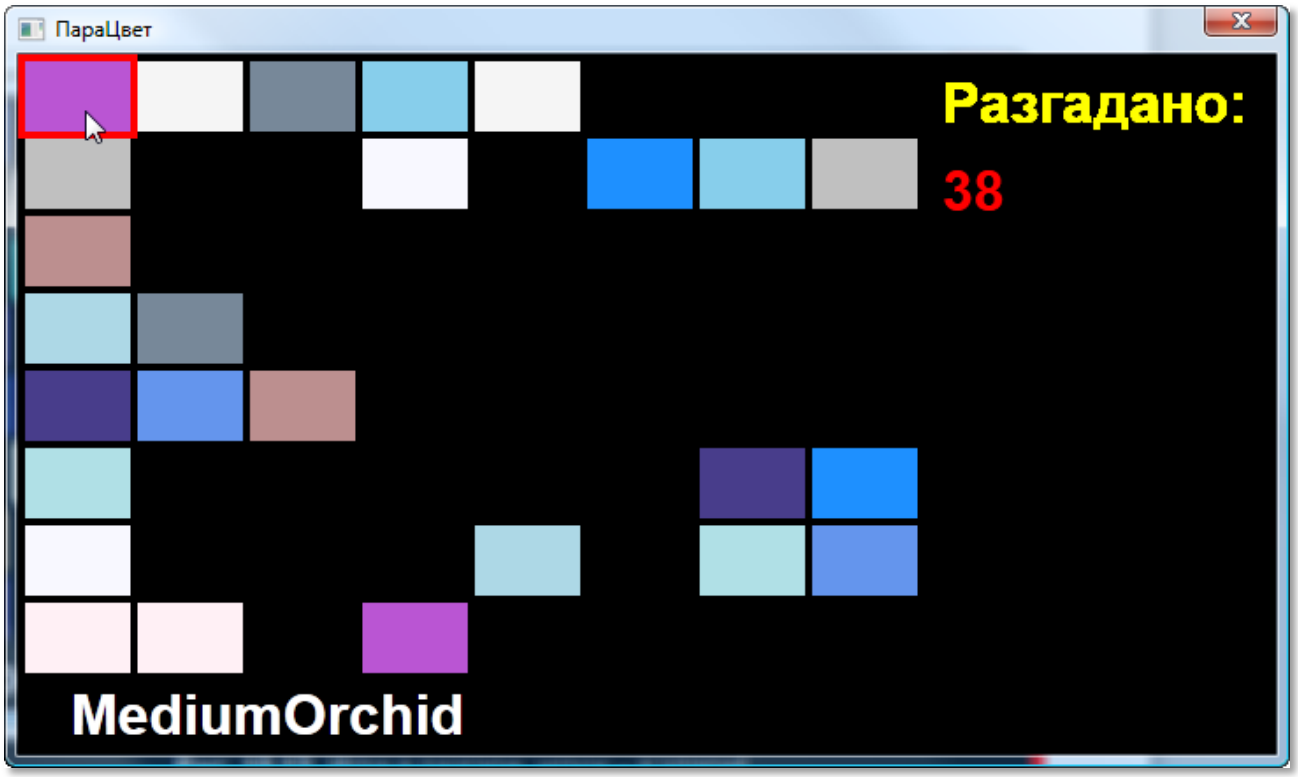


Рис. 10.12. Игра в разгаре, игрок – в ударе!

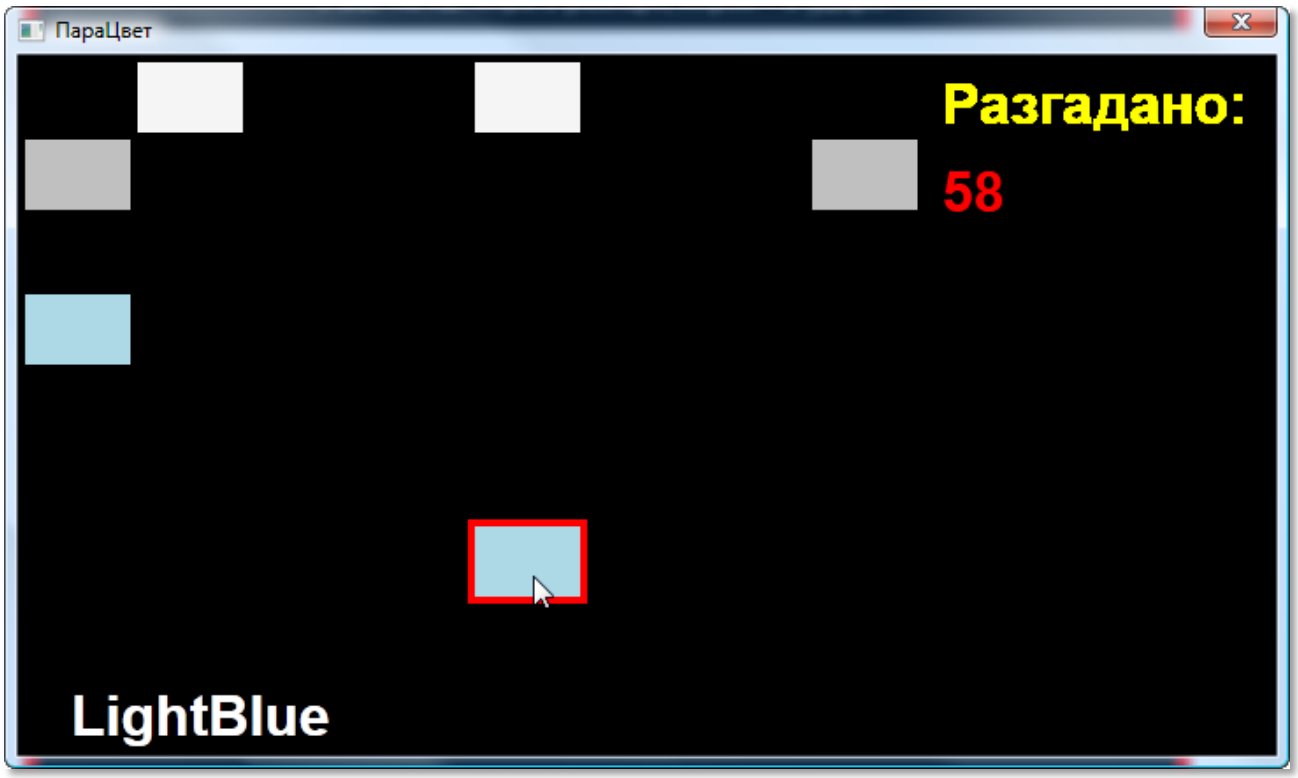


Рис. 10.13. Развязка близится!

Если глаз у вас не замылился, как у разведчика Володи Шарапова, то следом за полной зачисткой местности вас ждёт заслуженная *ГЕЙМ‘ОВА* (Рис. 10.14)!

Рис. 10.14. Чистая победа!

Впрочем, тут же начнётся новая партия игры, так что долго почивать на лаврах вам не удастся…

Как видите, на *паскале* в содружестве с программой *PascalABC.NET* и библиотекой *SmallBasicLibrary* можно писать вполне конкретные игры. Кроме собственно, тренировочной составляющей, игра *ПараЦвет* заостряет и обостряет цветовое зрение, целкость, зоркость, верность рук, хладнокровие и смётку. А также помогает выучить причудливые английские названия затейливых цветов.

Казалось бы, на этом всё – но нет: **игру ещё нужно написать**!

За основу мы примем проект *Цветовая палитра*, который мы обильно оснастим игровым моментом.

Вполне разумно ограничить игровое поле и наши усилия квадратной таблицей на 64 клетки:

```
unit DrawUnit;
```

```
uses
   Microsoft.SmallBasic.Library, System, System.Threading;

type
   Draw = class

      // размеры клеток:
      const WIDTH = 60;
      const HEIGHT = 40;
      // зазор между клетками:
      const GAP = 4;
      // число клеток по горизонтали и вертикали:
      const SIZE = 8;

      // игровое поле:
      Field := new integer[SIZE, SIZE];
      rand := new Random();
```

Координаты первой кликнутой клетки и общее число разгаданных клеток мы запомним в целочисленных **полях**:

```
      // индексы кликнутой клетки:
      aktivX := -1;
      aktivY := -1;
      // угадано клеток:
      solved := 0;
```

Список цветов я предусмотрительно не привожу, чтобы не затягивать удовольствие их перечислением:

```
    COLOR := new string[] ('AliceBlue',
                           'AntiqueWhite',
                           'Aqua',
                           'Aquamarine',
                            . . .
                           );
```

Вы вполне можете подредактировать его в сторону сокращения.

Метод **Prepare** нужно лишь незначительно поправить и дополнить:

```
public procedure Prepare();
begin
   GraphicsWindow.Hide();
   GraphicsWindow.Title := 'ПараЦвет';
   GraphicsWindow.Width := SIZE * (WIDTH + GAP) - GAP - 2 + 200;
   GraphicsWindow.Height := SIZE * (HEIGHT + GAP) - GAP + 40;
   GraphicsWindow.Show();
   GraphicsWindow.Left := (Desktop.Width –
                           GraphicsWindow.Width) / 2;
   GraphicsWindow.Top := (Desktop.Height –
                          GraphicsWindow.Height) / 2;
   GraphicsWindow.CanResize := false;
   GraphicsWindow.BackgroundColor := 'Black';
   GraphicsWindow.FontName := 'Arial';
   GraphicsWindow.FontSize := 32;
   GraphicsWindow.MouseMove += OnMouseMove;
end;
```

Размеры окна мы увеличили на 200 пикселей, чтобы информировать игрока о набранных им в упорной борьбе очках. Весь игровой цикл заключён в методе **Game**.

Каждый **игровой цикл** начинается с *подготовки* к новой баталии:

```
public procedure Game();
begin
   PrepareGame();
```

В методе **PrepareGame** мы, прежде всего, случайным образом, отбираем 32 разных цвета для заполнения таблицы:

```
// ГОТОВИМСЯ К ИГРЕ
private procedure PrepareGame();
begin
   // отбираем 32 разных цвета:
   var nc := SIZE * SIZE div 2;
   var iclr := new int32[nc];
   var ncall := COLOR.Count();
   // случайный цвет:
   var rclr := 0;
   for var i := 0 to nc - 1 do
   begin
```

```
      repeat
         // случайный цвет:
         rclr := rand.Next(ncall);
      until not (iclr.Contains(rclr));
      iclr[i] := rclr;
   end;
```

Теперь в эти цвета опять же случайно нужно окрасить по 2 клетки игрового поля **Field**:

```
   // расставляем цвета парами в массиве Field:
   System.Array.Clear(Field, 0, Field.Length);
   var x := 0;
   var y := 0;
   for var i := 0 to nc - 1 do
   begin
      // очередной цвет:
      var clr := iclr[i];
      // первая случайная клетка в массиве Field:
      repeat
         x := rand.Next(SIZE);
         y := rand.Next(SIZE);
      until not (Field[x, y] <> 0);
      Field[x, y] := clr;
      // вторая случайная клетка в массиве Field:
      repeat
         x := rand.Next(SIZE);
         y := rand.Next(SIZE);
      until not (Field[x, y] <> 0);
      Field[x, y] := clr;
   end;
end;
```

Теперь в методе *Game* можно **показать** игровое поле во всей красе:

```
   DrawTable();
```

Метод **DrawTable** вам уже знаком, поэтому его нужно только слегка подправить: если клетка разгадана, то она должна быть залита чёрным цветом (по этой причине чёрный цвет из списка цветов исключён и в игре не используется):

```
// РИСУЕМ ЦВЕТНУЮ ТАБЛИЦУ
private procedure DrawTable();
begin
   for var y := 0 to SIZE - 1 do
      for var x := 0 to SIZE - 1 do
      begin
         if (y * SIZE + x < COLOR.Count()) and
            (Field[x, y] <> -1) then
            GraphicsWindow.BrushColor := COLOR[Field[x, y]]
         else
            GraphicsWindow.BrushColor := 'Black';

         GraphicsWindow.FillRectangle(GAP + x * (WIDTH + GAP),
                                      GAP + y * (HEIGHT + GAP),
                                      WIDTH, HEIGHT);
      end
end;
```

Опять возвращаемся в метод *Game* и продолжаем подготовительные работы:

```
Neaktiv(0, 0);
// угадано клеток:
solved := 0;
PrintScore();
GraphicsWindow.PenWidth := GAP;
```

В методе **Neaktiv** мы сбрасываем выделение всех клеток:

```
// КЛЕТКА СТАНОВИТСЯ НЕАКТИВНОЙ
private procedure Neaktiv(x, y: integer);
begin
   aktivX := -1;
   aktivY := -1;
   GraphicsWindow.PenColor := 'Black';
   GraphicsWindow.DrawRectangle(x * (WIDTH + GAP) + 2,
                                y * (HEIGHT + GAP) + 2,
                               WIDTH + 2 * GAP - 4,
                               HEIGHT + 2 * GAP - 4);
end;
```

В методе **PrintScore** печатаем число разгаданных клеток, которое равно значению поля *solved*:

```
private procedure PrintScore();
begin
   GraphicsWindow.BrushColor := 'Yellow';
   var tx := (Int32)(GraphicsWindow.Width) - 190;
   var ty := 10;
   GraphicsWindow.DrawText(tx, ty, 'Разгадано:');

   ty := 60;
   GraphicsWindow.BrushColor := 'Black';
   GraphicsWindow.FillRectangle(tx, ty, 190, 40);
   GraphicsWindow.BrushColor := 'Red';
   GraphicsWindow.DrawText(tx, ty, solved);
end;
```

Запускаем программу и водим мышку по полю. При этом вызывается метод **OnMouseMove**, который мы используем для вывода подсказок - названий тех цветов, на которых находится мышка. Этот метод вам также хорошо известен, но в этом проекте мы иначе вычисляем цвет клетки - берём его из массива *Field*:

```
// ПЕЧАТАЕМ ПОДСКАЗКУ
private procedure OnMouseMove();
begin
   var x := (integer)(GraphicsWindow.MouseX);
   var y := (integer)(GraphicsWindow.MouseY);
   var xc := x div (WIDTH + GAP);
   var yc := y div (HEIGHT + GAP);

   if (xc < 0) or (yc < 0) or (xc >= SIZE) or
      (yc >= SIZE) then
      exit;
   // номер цвета в массиве:
   var nc := Field[xc, yc];
   if (nc = -1) then
      exit;
   var ty := (Int32)(GraphicsWindow.Height) - 40;
   var tx := 30;
   GraphicsWindow.BrushColor := 'Black';
   GraphicsWindow.FillRectangle(0, ty,
                                GraphicsWindow.Width, 40);

   if (xc > SIZE - 1) or (yc > SIZE - 1) or
      (nc >= COLOR.Count()) then
```

```
      exit;

   // печатаем название цвета:
   GraphicsWindow.BrushColor := 'White';
   GraphicsWindow.DrawText(tx, ty, COLOR[nc]);
end;
```

А в методе *Game* уже началась текущая партия:

```
while(true ) do
begin
   Thread.Sleep(TimeSpan.FromMilliseconds(160));
   var x := (Int32)(Mouse.MouseX - GraphicsWindow.Left) - 7;
   var y := (Int32)(Mouse.MouseY - GraphicsWindow.Top) - 26;
   if (x < 0) or (y < 0) then
      continue;
```

Здесь мы снимаем координаты мышки, используя свойства **MouseX** и **MouseY** класса *Mouse*, учитывая, что они возвращают координаты мышки на *Рабочем столе*, а не в окне приложения, поэтому их необходимо дополнительно преобразовать. Если мышка находится вне клиентской области окна, то её дальнейшие действия мы тут же и пресекаем.

Если мышка находится в клиентской области, то мы вычисляем **координаты** клетки и отвергаем те из них, которые выходят за границы игрового поля:

```
var xc := x div (WIDTH + GAP);
var yc := y div (HEIGHT + GAP);
if (xc < 0) or (yc < 0) or (xc >= SIZE) or (yc >= SIZE) then
   continue;
```

В игре нас интересуют не прогулки мышки по игровому полю, а отмечаемые ею с помощью клика клетки. А для этого игрок должен нажать левую кнопку мышки:

```
// если нажата левая кнопка мышки, то:
if (Mouse.IsLeftButtonDown) then
begin
```

И вот тут могут возникнуть различные ситуации.

Если щелчок пришёлся на уже разгаданную клетку (на экране она окрашена в чёрный цвет, а в массиве *Field* имеет значение -1), то ничего делать не нужно:

```
// если это уже разгаданная клетка:
if (Field[xc, yc] = -1) then
   continue;
```

Если кликнутая клетка уже была кликнута раньше, то есть сейчас она *активна* (её координаты хранятся в полях *aktivX* и *aktivY*), то она становится **неактивной**:

```
// если клетка кликнута повторно, то она
// становится неактивной:
if (xc = aktivX) and (yc = aktivY) then
begin
   Neaktiv(xc, yc);
   continue;
end;
```

Если до этого клика активной клетки не было (тогда поле *aktivX* равно -1), то кликнутая клетка превращается в **активную**:

```
// если это первая кликнутая клетка:
if (aktivX = -1) then
begin
   // запоминаем кликнутую клетку:
   aktivX := xc;
   aktivY := yc;
   GraphicsWindow.PenColor := 'Red';
   GraphicsWindow.DrawRectangle(xc * (WIDTH + GAP) + 2,
                                yc * (HEIGHT + GAP) + 2,
                                WIDTH + 2 * GAP - 4,
                                HEIGHT + 2 * GAP - 4);
end
```

Мы запоминаем её координаты в полях **aktivX** и **aktivY** и обводим **красной** рамкой.

Если клик пришёлся на неактивную клетку при существующей активной клетке, значит, игрок щёлкнул по предполагаемой **парной** клетке:

```
else // вторая кликнутая клетка:
begin
```

Тут мы, естественно, должны **сравнить** цвет первой (активной) клетки, который равен *Field[aktivX,aktivY]* с цветом второй клетки, который равен *Field[xc,yc]:*

```
 // её цвет:
 var clr2 := Field[xc, yc];
 // цвет первой клетки:
 var clr1 := Field[aktivX, aktivY];
```

Если они **не совпали**, то программа грустно пискнет и отправится в начало цикла *while*:

```
if (clr2 <> clr1) then
begin
   Sound.PlayChimes();
end
```

Если же клетки оказались одноцветными, то игрок получает 2 очка и приятную информацию от программы:

```
else // правильная пара!
begin
   solved += 2;
   PrintScore();
```

Если это была последняя пара клеток, то **игра заканчивается**:

```
// игра закончена:
if (solved = SIZE * SIZE) then
```

```
begin
   // стираем обе клетки:
   Clear2(xc, yc);
   GraphicsWindow.ShowMessage('GAME OVER!', 'Парацвет');
   exit;
end
```

В противном случае мы **окрашиваем** эту пару клеток в чёрный цвет:

```
// СТИРАЕМ ПАРУ КЛЕТОК
private procedure Clear2(x2, y2: integer);
begin
   Field[aktivX, aktivY] := -1;
   Field[x2, y2] := -1;
   Neaktiv(aktivX, aktivY);
   DrawTable();
end;
```

И продолжаем игру дальше – до победного конца:

```
                else // игра продолжается:
                begin
                    // стираем обе клетки:
                   Clear2(xc, yc);
                end;
            end;
         end;
      end;
    end;
 end;
```

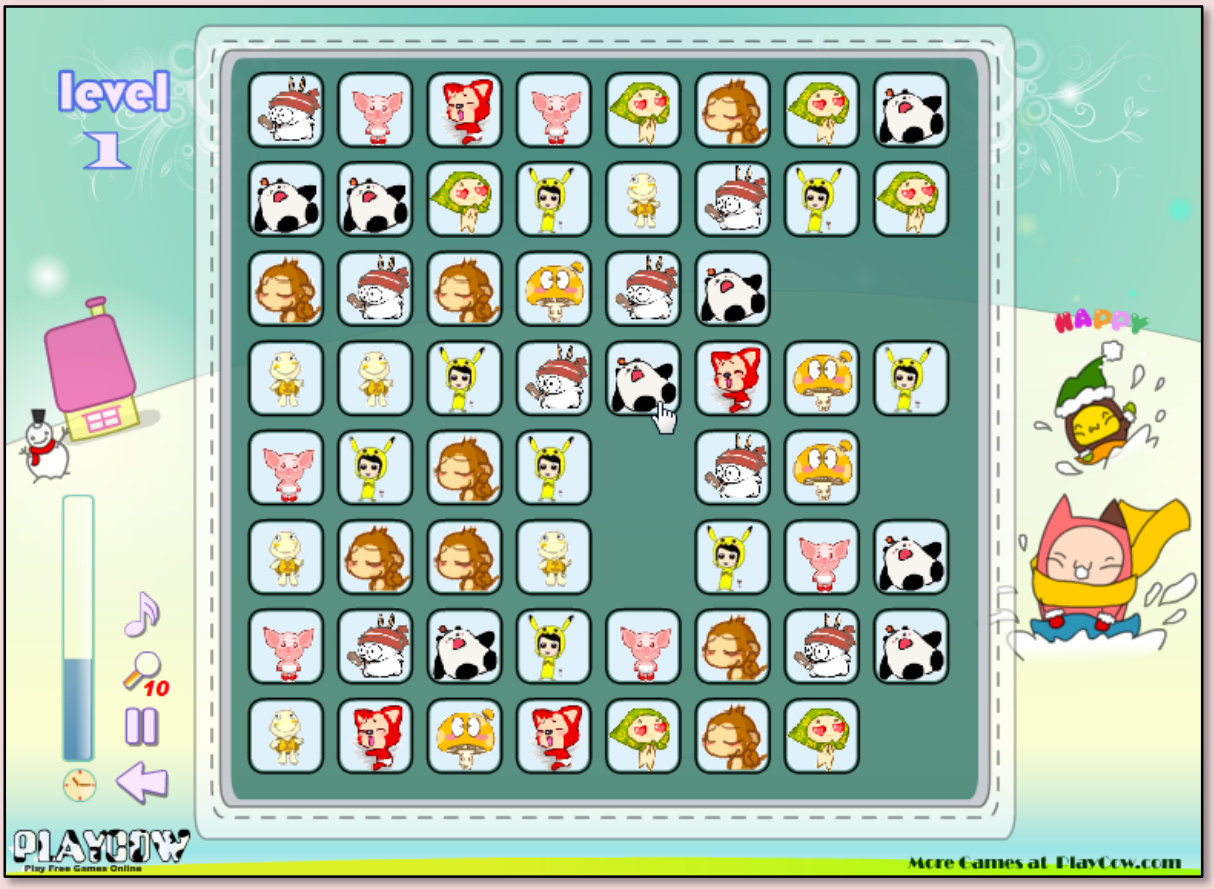
Задания для самостоятельного решения

ПараЦвет

Чтобы обострить игру:

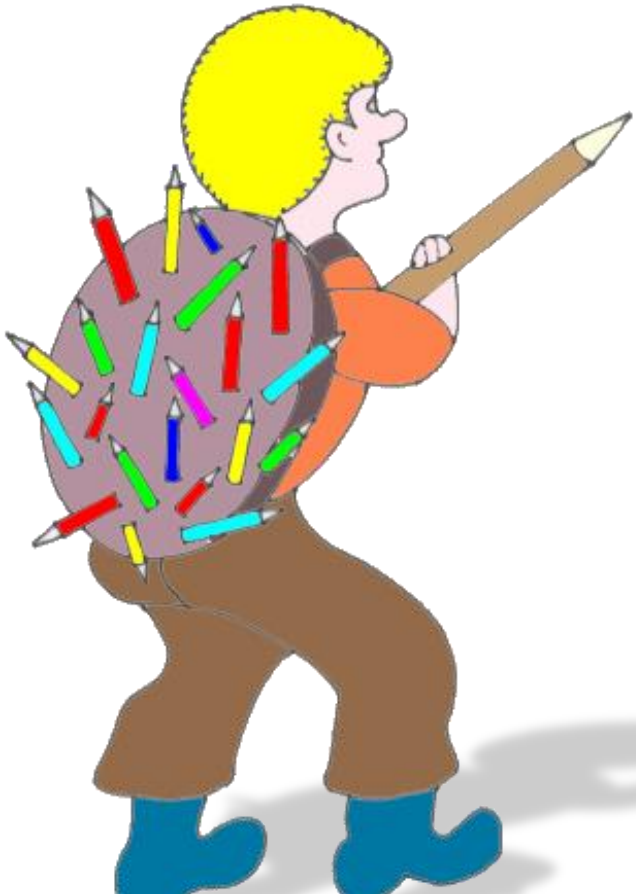
1. Добавьте таймер, и ограничьте игрока во времени.
2. Введите штрафные санкции за неправильный выбор пар клеток, то есть наказывайте игрока за ошибки.
3. Добавьте кнопку, нажав на которую, игрок избавится от подсказок.

Нашу программу легко переделать в другие игры. Например, игры **Парные картинки** или **Найди пару** (Рис. 10.15) почти дословно повторяют наш *ПараЦвет*.


level
1
10
PLAYCOW
Play Free Games Online
More Games at PlayCow.com


v.1.01
Уровень 1/10
100

Рис. 10.15. Игры на вырост



Глава 11. Полярная система координат

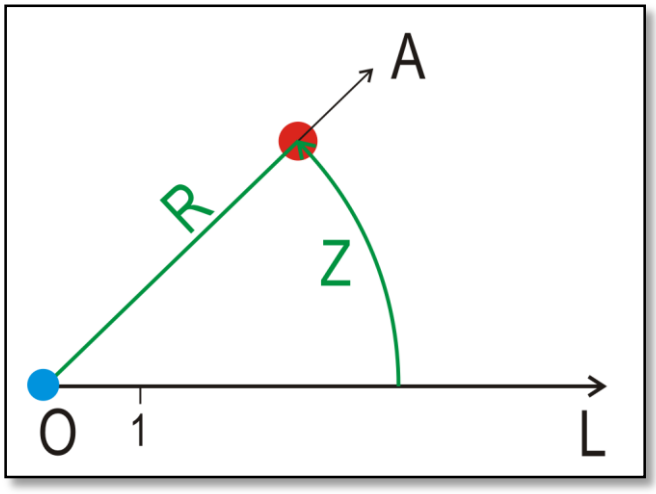
Трутся об ось медведи,-
Вертится Земля…

Песенка из комедии *Кавказская пленница*

Кроме прямоугольной системы координат, которая известна всем и каждому, существуют и другие. Если вы хотите поставить точку на плоскости, вам обязательно понадобятся *два* числа. В декартовой системе координат это абсцисса и ордината, а в **полярной**, которая нам понадобится в этой главе, - *полярный радиус R* и *полярный угол Z*.

> Полярный радиус называется также **радиус-вектором**. Длина радиус-вектора называется **модулем**.

Начало полярных координат находится, как легко догадаться, в **полюсе**. Из него можно провести в любом направлении луч, образующий с полярной осью *OL* угол **Z**, который отсчитывается *против* часовой стрелки. Если луч проходит через заданную точку, то её положение в полярной системе координат определяется значениями полярного угла **Z** и полярного радиуса **R**, который равен расстоянию от точки до полюса. Очень хорошо это видно на картинке (Рис. 11.1).



O – полюс
OL – полярная ось
OA – луч, проходящий через заданную точку на плоскости

R – полярный радиус точки
Z – её полярный угол

Рис. 11.1. Полярная система координат

Поскольку координаты пикселей на экране удобнее задавать в прямоугольной системе координат, то давайте выведем **формулы**, по которым можно пересчитать координаты точки в *полярной* системе в координаты в

прямоугольной системе. Для этого совместим начало прямоугольной системы координат с полюсом, а ось абсцисс – с полярной осью (Рис. 11.2).

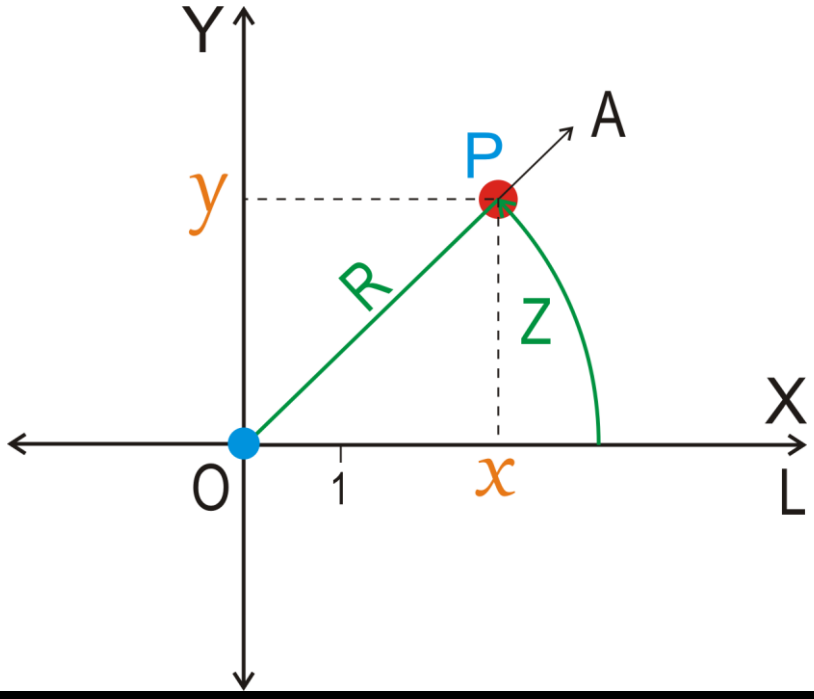


Рис. 11.2. Координаты точки в двух системах координат

Опустим перпендикуляры из точки *P* на оси абсцисс и ординат. Точки пересечения перпендикуляров обозначим буквами *x* и *y* и рассмотрим прямоугольный треугольник *OPx*, из которого легко найдём координаты точки *P*:

x = R * cos(Z) (1)
y = R * sin(Z) (2)

Возникает вопрос: если от полярных координат так просто перейти к прямоугольным, то для чего вообще их использовать? – Дело в том, что в полярных координатах некоторые функции задавать гораздо проще, чем в прямоугольных.

Например, **окружности** можно описать такими формулами:

R = 1 – для единичной окружности или
R = r – для окружности радиуса *r*

Те же самые окружности в прямоугольных координатах имеют совсем непростой вид:

$x^2 + y^2 = 1$ и
$x^2 + y^2 = r^2$

И многие другие кривые - спирали, улитки, розы, эллипсы, кардиоиды – удобнее задавать именно в полярных координатах. Чтобы убедиться в этом, давайте построим графики нескольких знаменитых кривых, описанных в полярных координатах.

Проект *Полярные кривые*

Исходный код программы находится в папке **Полярные кривые**.

Начните проект **Полярные кривые** и сохраните его в новой папке.

Для построения графиков нам нужны несколько **полей**, назначение которых вполне очевидно:

```
{$apptype windows}

// Polar

// ПРОГРАММА ДЛЯ ПОСТРОЕНИЯ ГРАФИКОВ
// В ПОЛЯРНЫХ КООРДИНАТАХ

uses
   DrawUnit;

begin
   var draw := new Draw();
   draw.Prepare();
   draw.Draw();
end.

unit DrawUnit;

uses
   Microsoft.SmallBasic.Library, System, System.Threading;

type
   Draw = class
```

```
    const GWWIDTH = 320;
    const GWHEIGHT = 320;

    // размеры окна:
    width: integer;
    height: integer;
    // координаты центра окна:
    CX: integer;
    CY: integer;

    // координаты точки:
    x: double;
    y: double;

    // название фигуры:
    figura: string;


    procedure Prepare();
    begin
       GraphicsWindow.Hide();
       GraphicsWindow.Title := 'Графики в полярных координатах';
       GraphicsWindow.Width := GWWIDTH;
       GraphicsWindow.Height := GWHEIGHT;
       GraphicsWindow.Show();
       GraphicsWindow.Left := (Desktop.Width –
                               GraphicsWindow.Width) / 2;
       GraphicsWindow.Top := (Desktop.Height –
                              GraphicsWindow.Height) / 2;
       GraphicsWindow.CanResize := false;
       GraphicsWindow.BackgroundColor := 'White';
    end;
```

В методе **Draw** в строковой переменной *figura* мы задаём имя вычерчиваемой фигуры и вызываем метод *DrawFig*:

```
public procedure Draw();
begin
   width := GWWIDTH;
   height := GWHEIGHT;
   var penWidth := 1;
   GraphicsWindow.PenWidth := penWidth;

   // координаты центра окна:
   CX := width div 2;
   CY := height div 2;
```

```
   // цвет линий:
   GraphicsWindow.PenColor := 'Black';
   //         figura := 'Circle';
   //         figura := 'Clever';
   //         figura := 'Ulitka';
   //         figura := 'Spiral';
   //         DrawFig(1);
   figura := 'Star';
   DrawFig(7);
end;
```

Чтобы построить график, мы изменяем полярный угол **Z** от нуля до 360 градусов (в радианах это – *0 ... 2*Math.PI*). При дальнейшем увеличении полярного угла новые точки будут просто накладываться на уже существующие, поэтому такое ограничение угла вполне уместно.

> Внешний цикл *for* нужен только для вычерчивания *звёздочки*.

Важно задать небольшой **шаг** (*0.01*) изменения угла, чтобы точки графика создавали *цельную* кривую. В некоторых случаях не помогает и мелкий шаг, поэтому мы не будем ставить отдельные точки, а проведём **линии** из каждой последующей точки в предыдущую. Так мы застрахуем себя от разрывов в графике кривой. У самой первой точки нет предыдущей, поэтому мы должны учесть это исключение:

```
// СТРОИМ КРИВУЮ
private procedure DrawFig(n: integer);
begin
   // полярный угол:
   var Z := 0.0;
   // координаты точки:
   var x1 := 0.0;
   var y1 := 0.0;

   for var i := 1 to n do
   begin
      Z := 0.0;
      while (Z <= 2 * System.Math.PI + 0.01) do
      begin
         Calc(Z, i);
         // запоминаем координаты первой точки:
         if (Z = 0) then
         begin
            x1 := x;
            y1 := y;
```

```
         end;
         Z += 0.01;
         // координаты точки выходят
         // за границы области рисования:
         if (x < 0) or (x > width) or (y < 0) or (y > height) then
            continue;

         // ставим "точку";
         GraphicsWindow.DrawLine(x, y, x1, y1);

         // координаты "предыдущей" точки:
         x1 := x;
         y1 := y;
         Thread.Sleep(TimeSpan.FromMilliseconds(1));
      end
   end
end;
```

Построение всех кривых оформлено в виде **отдельных методов**, что, конечно, гораздо удобнее, чем записывать соответствующий код одним куском.

Для каждого угла *Z* мы вызываем метод **Calc**, в котором вычисляем координаты точки *(x,y)* для этого угла:

```
// ВЫЧИСЛЯЕМ СЛЕДУЮЩУЮ ТОЧКУ КРИВОЙ
private procedure Calc(Z: double; i: integer);
begin
   if (figura = 'Circle') then
      Circle(Z)
   else if (figura = 'Clever') then
      Clever(Z)
   else if (figura = 'Ulitka') then
      Ulitka(Z)
   else if (figura = 'Spiral') then
      Spiral(Z)
   else Star(Z, i);
end;
```

Полярный радиус мы задаём индивидуально для каждой кривой так, чтобы она гармонично вписалась в область рисования.

Проще всего построить **окружность** (Рис. 11.3):

```
// Окружность
```

```
private procedure Circle(Z: double);
begin
   // радиус окружности:
   var R := 150;
   x := CX + System.Math.Cos(Z) * R;
   y := CY + System.Math.Sin(Z) * R;
end;
```

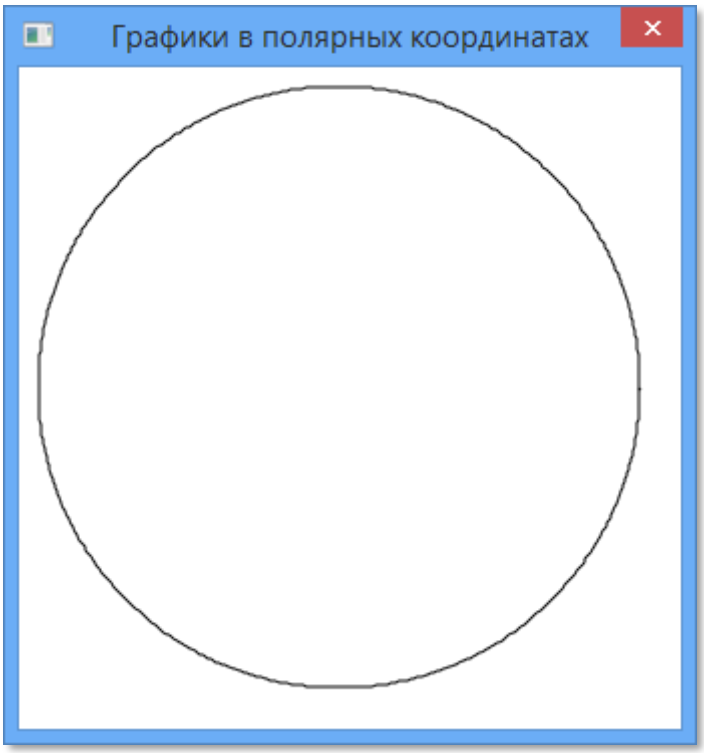


Рис. 11.3. Окружность R = 150

Глобальные переменные *x* и *y* хранят координаты очередной точки кривой, пересчитанные из полярных координат в прямоугольные (оконные) по тем формулам (1) и (2), что мы вывели в начале главы. Радиус окружности можно задавать произвольно, но так, чтобы она целиком помещалась в окне!

Следующие методы просто вычисляют прямоугольные координаты точек по полярным формулам соответствующих кривых. Важно помнить, что некоторые формулы содержат **параметры**, при изменении которых график приобретает другой вид. Это значит, что с помощью этих формул вы можете начертить гораздо больше красивых кривых, чем показано в книге.

Например, по формуле *R= sin(2*Z)* (Рис. 11.4, слева) будет вычерчен лист **клевера**, а по формуле *R= sin(12*Z)* (Рис . 11.4, справа) – **ромашка**. В дан-

ном случае можно рассматривать формулу *R= sin(k*Z)* с параметром *k*, который задаёт число лепестков кривой.

```
// Клевер, ромашки
private procedure Clever(Z: double);
begin
   var k := 2; //12
   var R := System.Math.Sin(k * Z) * 160.0;
   x := CX + System.Math.Cos(Z) * R;
   y := CY + System.Math.Sin(Z) * R;
end;
```

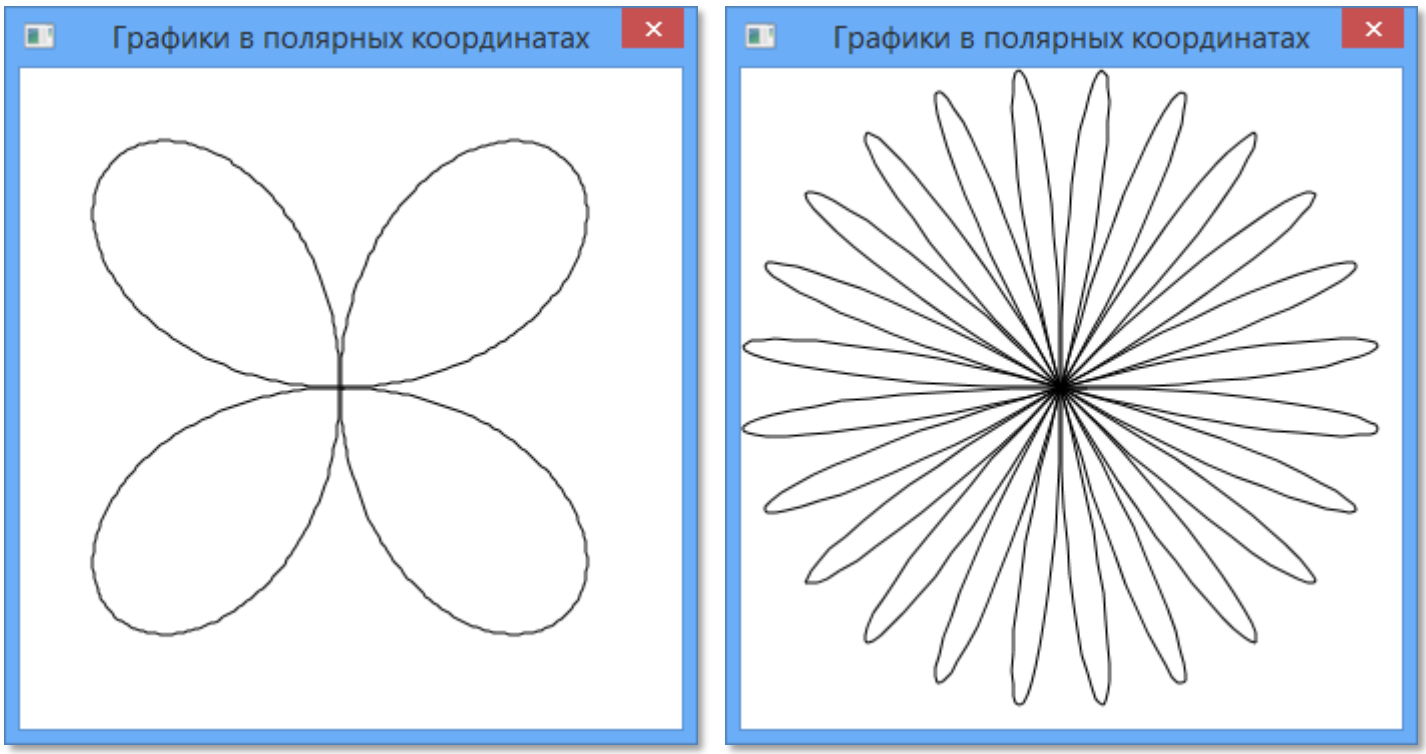


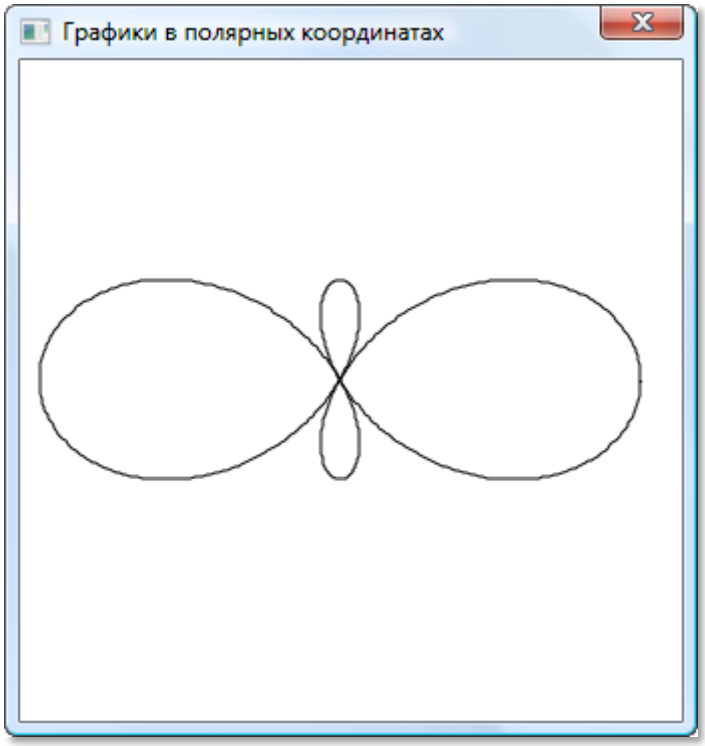
Рис. 11.4. R = sin(2*Z) - клевер R= sin(12*Z) - ромашка

Обязательно попробуйте поиграть параметрами или добавить в формулу новые члены, чтобы найти другие кривые!

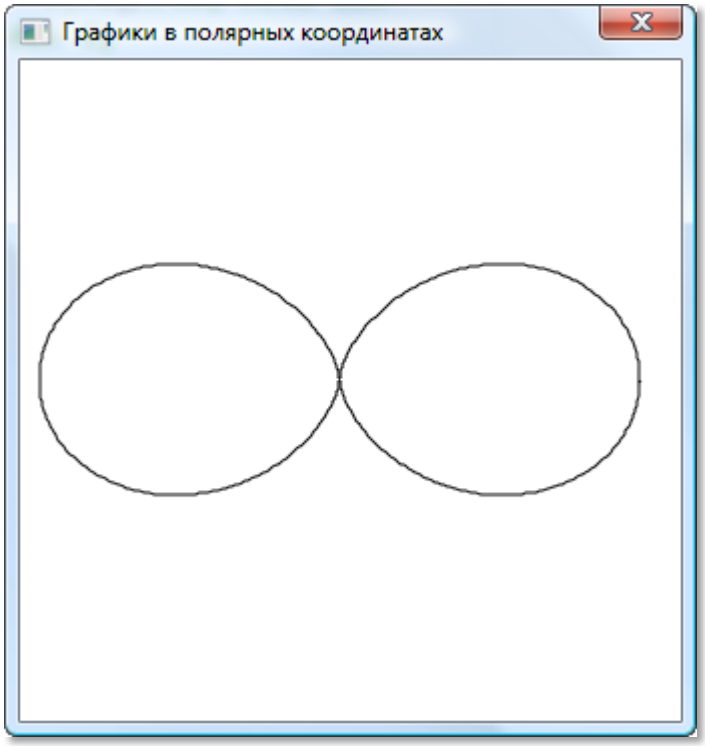
Например, с помощью следующего метода можно вычертить несколько замечательных кривых (Рис. 11.5):

```
// Улитка Паскаля
private procedure Ulitka(Z: double);
begin
   var k := 2.0;//2;
```

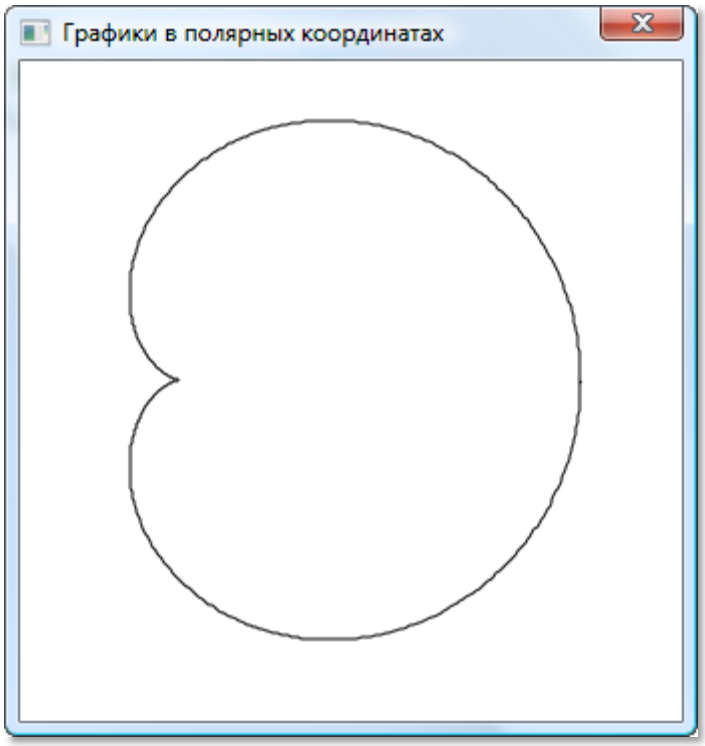
```
    var l := 1.0;//1;//2;
    var R := (1.0 + l * System.Math.Cos(k * Z)) * 75.0;//50; //80 100
    x := CX + System.Math.Cos(Z) * R;// - width div 4;
    y := CY + System.Math.Sin(Z) * R;
end;
```



R= 1 + 2*cos(2*Z) – петельное сцепление



R= 1 + cos(2*Z) – знак бесконечности



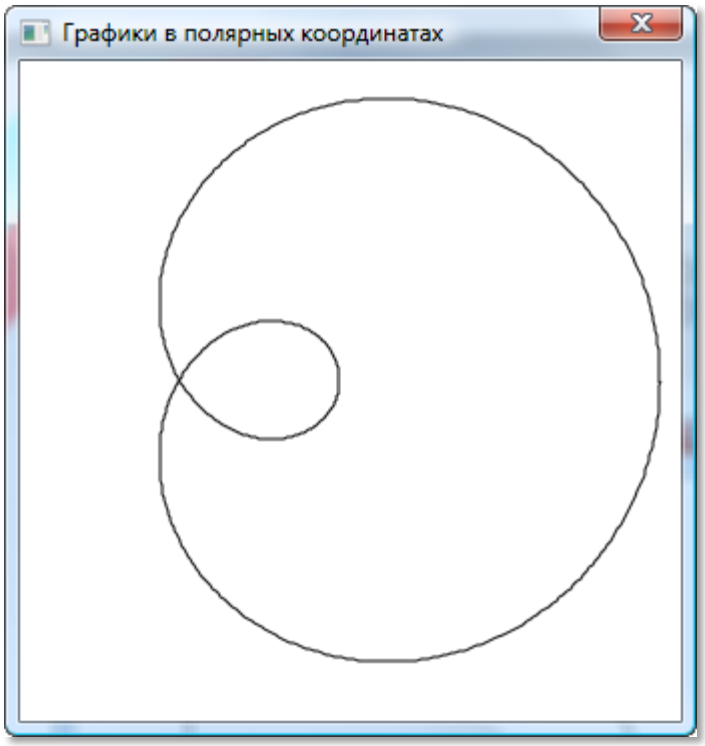


Рис. 11.5. R= 1 + cos(Z) - кардиоида R= 1 + 2*cos(Z) – улитка Паскаля

И ещё парочка любопытных полярных «загогулин» - **спираль** (Рис. 11.6) и **звёздочка** (Рис. 11.7).

```
// Спираль
private procedure Spiral(Z: double);
begin
   var k := 5.0;
   var R := Z / k * 100.0;
   x := CX + System.Math.Cos(Z) * R;
   y := CY + System.Math.Sin(Z) * R;
end;
```

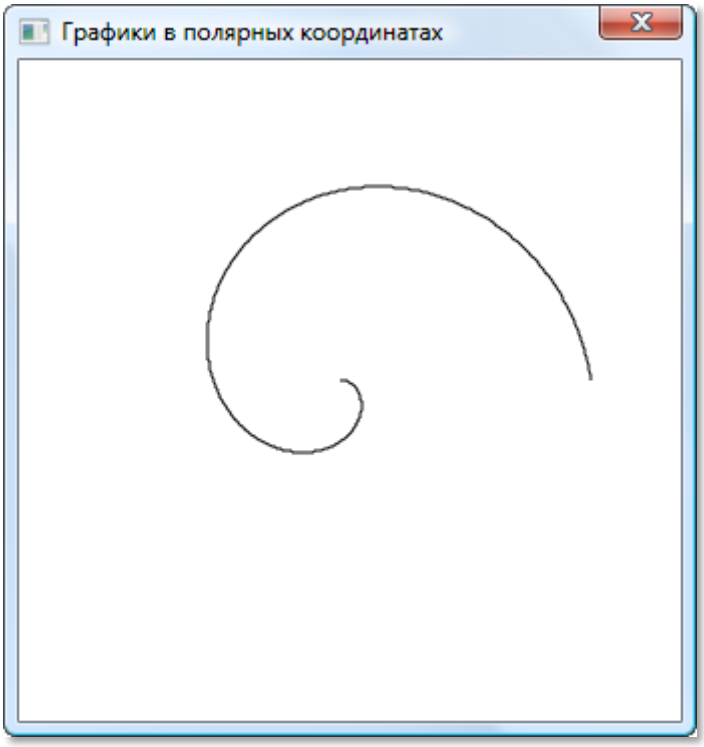


Рис. 11.6. R= Z/k – спираль

```
// Звёздочка
private procedure Star(Z: double; i: integer);
begin
   var k := 5;
   var F := Z + i * System.Math.PI * 2;
   var R := 60 / (2 + System.Math.Cos(F * k +
         ((System.Math.Floor(0.1 * F) / 5 -
         System.Math.Floor(System.Math.Floor(0.1 * F) / 5))))) * 2.5;
   x := CX + System.Math.Cos(F) * R;
   y := CY + System.Math.Sin(F) * R;
end;
```

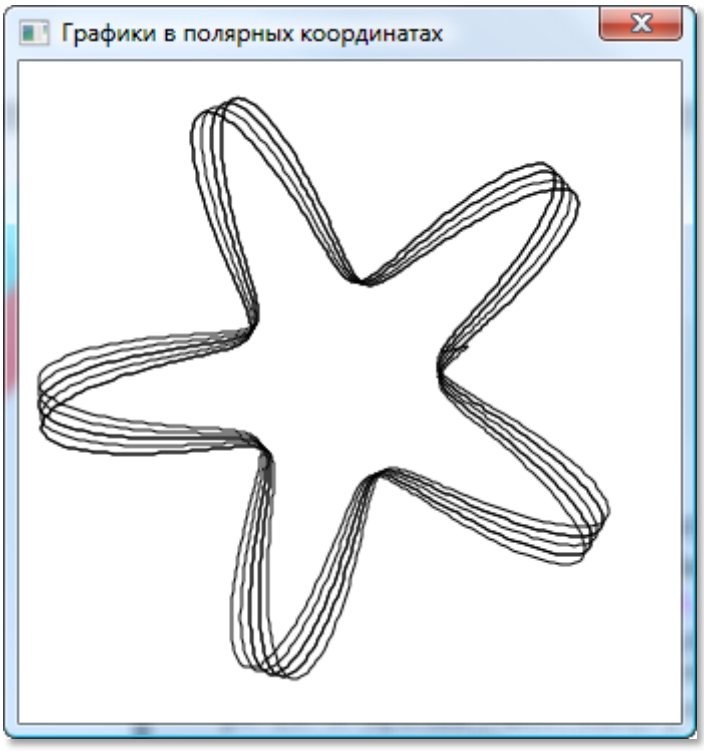


Рис. 11.7. Звёздочка

Задания для самостоятельного решения

Полярные кривые

1. Установите на форме **кнопки** для выбора нужной кривой.

2. Поэкспериментируйте с различными значениями параметров в формулах. Так, например, можно получить ромашки с разным числом лепестков или улитки Паскаля разной формы.

3. Добавьте к программе возможность автоматического **масштабирования** графиков под размеры клиентской области окна приложения. Для этого перед началом построения графиков следует найти максимальные значения координат **x** и **y** и вычислить коэффициент масштабирования, на который затем умножать все значения текущих координат.

Глава 12. Черепашья графика

Черепаха - русский, украинский
Чарапаха - белорусский
Tortoise, **turtle** - английский
Galápago - испанский
Testudo - латинский
Tartaruga - португальский

Вы уже научились рисовать картинки с помощью графических примитивов, которые предоставляет класс *GraphicsWindow*, - окружностей и эллипсов, прямоугольников и квадратов, треугольников и прямых линий. Их вполне достаточно, чтобы вычертить на канве любую фигуру. Для задания координат характерных точек этих примитивов мы используем прямоугольную систему координат канвы. Однако вы знаете, что в некоторых случаях удобнее перейти к полярным координатам, чтобы вычертить те или иные кривые (вы с ними познакомились в предыдущей главе). Впрочем, нам всё равно пришлось пересчитывать полярные координаты точек кривых в прямоугольные, поскольку координаты пикселей канвы мы не можем задавать иначе. А ведь было бы, наверное, зд***о***рово рисовать кривые *сразу* в полярных координатах!

Впервые такая возможность появилась в языке программирования *Лого* (*Logo*), который был разработан в 1967 году *Сеймуром Пейпертом* и *Идит Харель*. *Лого* создавался как средство обучения детей дошкольного и младшего школьного возраста основам программирования. Главными объектами языка *Лого* были слова и предложения, что и определило выбор названия для самого языка – *лого* по-гречески значит *слово*. Но известность этому языку программирования принесла маленькая **Черепашка** (*Turtle*), которая умела выполнять несложные команды и вычерчивать линии. Именно благодаря *Черепашке* этот язык стал очень популярным в 70-80-е годы прошлого века, а *Черепашка* появилась и в других языках

программирования – *LISP, SCHEME* и многих версиях *бейсика*. Есть *Черепашка* и в библиотеке *SmallBasicLibrary* - она «спрятана» в классе **Turtle**.

Проект *Знакомство с Черепашкой*

Исходный код программы находится в папке **Знакомство с Черепашкой**.

Морская черепашка
По имени Наташка,
С очками из Китая -
Такая вот крутая.

Натали, *Морская черепашка*

Чтобы увидеть *Черепашку* на экране, достаточно выполнить метод **Turtle.Show** (Рис. 12.1). Она всегда появляется в центре окна и смотрит вверх. Направление её взгляда знать очень важно, потому что *Черепашка* может ползти только туда, куда глаза глядят!

> Если вы хотите, чтобы *Черепашка* поползла в каком-либо направлении, сначала поверните её!

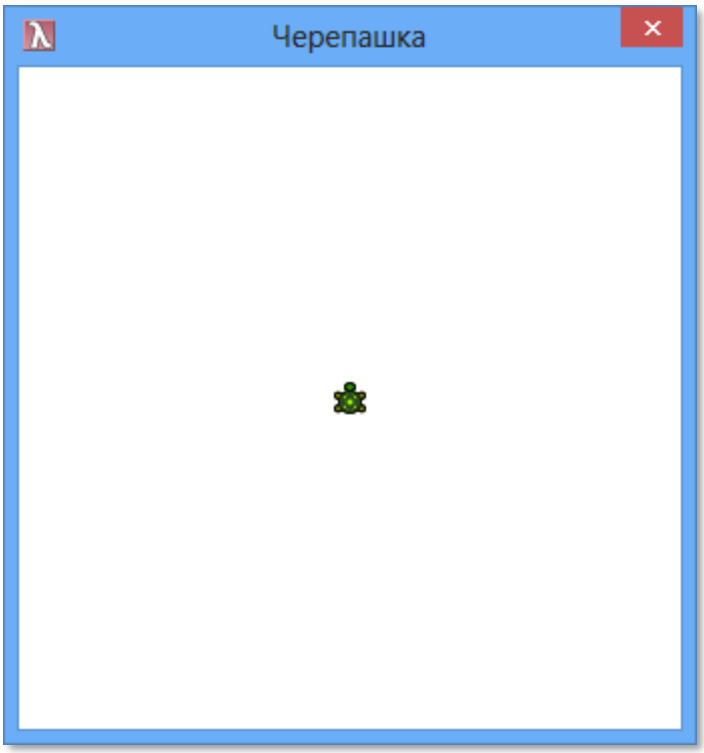


Рис. 12.1. «Новорождённая» *Черепашка*

В разных системах программирования *Черепашки* отличаются по форме. Обычно это стилизованные черепахи, но встречаются и примитивные виды *Черепашек* – равносторонние треугольники и наконечники стрелок (Рис. 12.2). Их объединяет желание ползать по экрану и возможность указывать направление их перемещения.

Рис. 12.2. *Черепашки* разных видов

Если вы уменьшите размеры окна, то *Черепашка* появится либо у правой нижней границы окна, либо вообще не появится, поэтому лучше устанавливать *Черепашку* в нужном месте окна «принудительно», задавая свойствам **X** и **Y** *Черепашки* нужные значения координат.

Например, если вы хотите, чтобы она возникла в *центре* окна (Рис. 12.3), то начните программу с такого кода:

```
{$apptype windows}

// Turtle

uses
   DrawUnit;

begin
   var draw := new Draw();
   draw.Prepare();
   draw.Show();
end.

unit DrawUnit;

uses
   Microsoft.SmallBasic.Library, System;

type
   Draw = class
      const GWWIDTH = 320;
      const GWHEIGHT = 320;

      width := GWWIDTH;
      height := GWHEIGHT;

      CX := width div 2;
```

```
      CY := height div 2;

      procedure Prepare();
      begin
         GraphicsWindow.Hide();
         GraphicsWindow.Title := 'Черепашка';
         GraphicsWindow.Width := GWWIDTH;
         GraphicsWindow.Height := GWHEIGHT;
         GraphicsWindow.Show();
         GraphicsWindow.Left := (Desktop.Width -
                                 GraphicsWindow.Width) / 2;
         GraphicsWindow.Top := (Desktop.Height -
                                GraphicsWindow.Height) / 2;
         GraphicsWindow.CanResize := false;
         GraphicsWindow.BackgroundColor := 'White';
      end;

      public procedure Show();
      begin
          // создаём Черепашку:
         Turtle.Show();
          // устанавливаем Черепашку в центре окна:
         Turtle.X := CX;
         Turtle.Y := CY;
      end;
   end; //end of class
end.
```

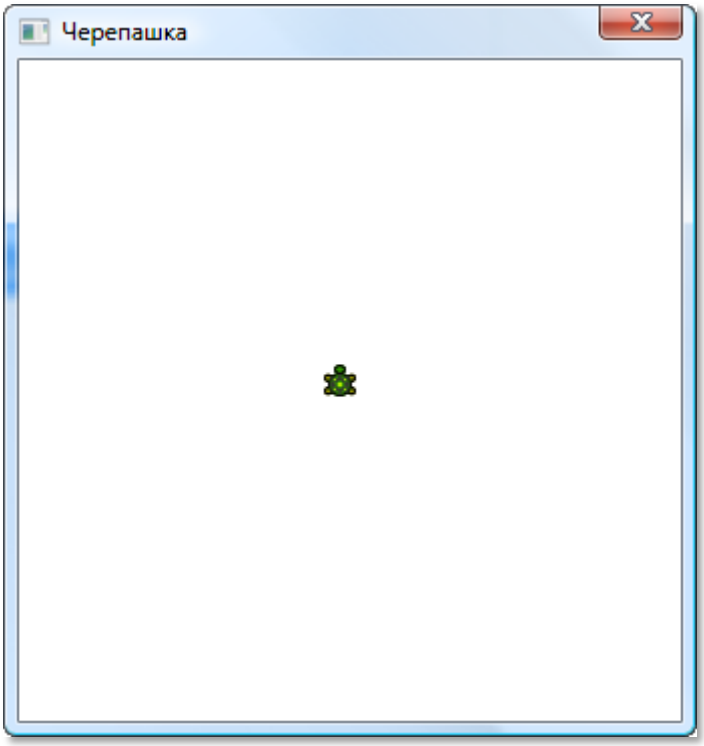


Рис. 12.3. Центровая *Черепашка*

Имейте в виду, что координаты *Черепашки* отсчитываются в системе координат канвы, то есть начало координат находится в левом верхнем углу, а ось *Y* направлена вниз!

Черепашка может выполнять несколько простых *команд*. Например, попросим её ползти **вперёд**:

```
public procedure MoveForward();
begin
   self.Show();
   Turtle.Move(100);
end;
```

Как и было обещано, *Черепашка* выполнит команду и проползёт *вперёд* (то есть *вверх*, потому что она туда смотрит!) 100 пикселей (Рис. 12.4).

А всё-таки *Черепашка* умеет двигаться и в противоположном взгляду направлении! Дайте ей команду

```
Turtle.Move(-100);
```

и она, смотря вперёд, попятится назад (Рис. 12.5).

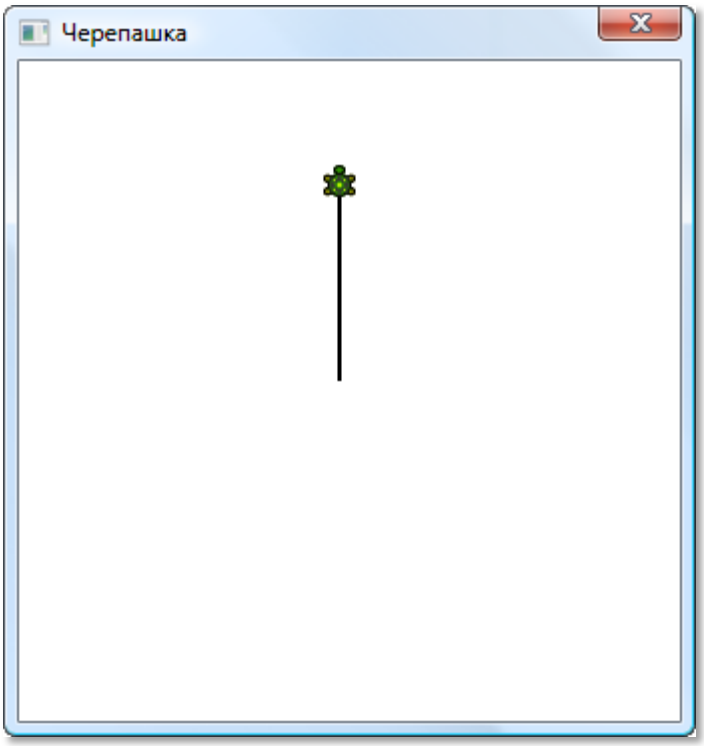


Рис. 12.4. Полный вперёд!

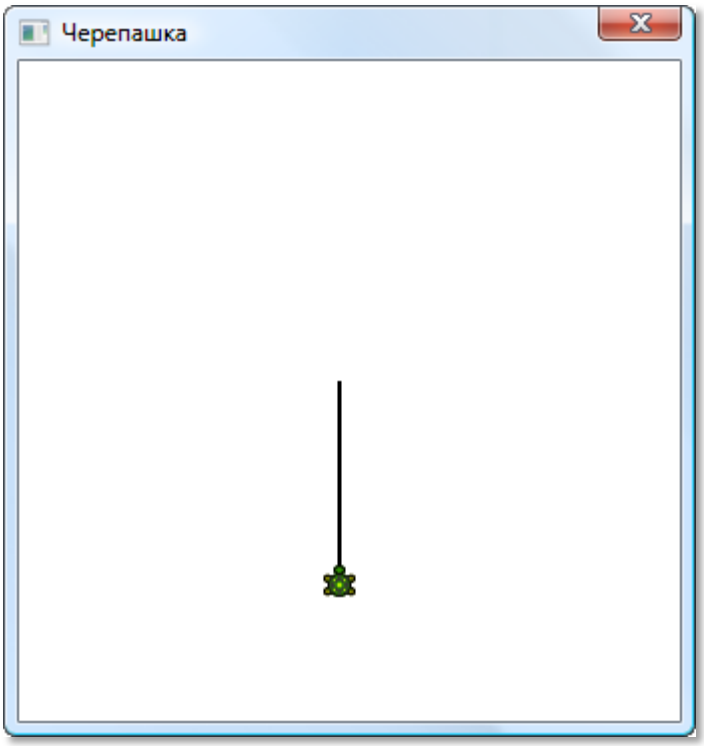


Рис. 12.5. Полный назад!

За ней остался след - тонкая прямая **чёрная** линия. Это всё потому, что *Черепашка* не простая, а вооруженная – карандашом! Вы можете дать *Черепашке* карандаш любого цвета, а также с грифелем большего диаметра, чтобы линии стали **толще**. Правда, для этого вам придётся обратиться к свойствам *Графического окна*, которое и выдаст *Черепашке* нужный *карандаш*:

```
GraphicsWindow.PenColor := 'Red';
GraphicsWindow.PenWidth := 10;
```

Теперь *Черепашка* проведёт **толстую красную** линию (Рис. 12.6).

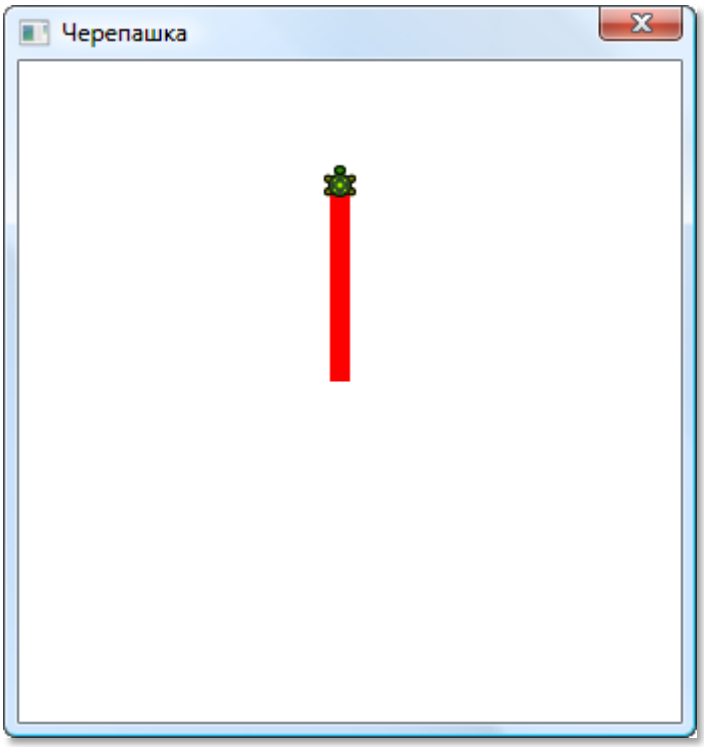


Рис. 12.6. *Черепашка* бороздит просторы *Графического окна*

Но *Черепашка* при ползании *не всегда* прочерчивает линию. Дело в том, что по умолчанию карандаш находится в боевом положении, поэтому от него остаётся след. За положение карандаша отвечают методы *Черепашки* **PenDown** и **PenUp**. Первый **опускает** карандаш, а второй – **поднимает** его. Соответственно этому *Черепашка* либо чертит линию, либо просто перемещается в новое положение.

Немного исправим программу так, чтобы *Черепашка* чертила *пунктирную* линию (Рис. 12.7).

```
public procedure Punktir();
begin
   self.Show();
   GraphicsWindow.PenColor := 'Green';
   GraphicsWindow.PenWidth := 4;

   for var i := 1 to 10 do
   begin
      if (i mod 2 = 0) then
         Turtle.PenUp()
      else
         Turtle.PenDown();

      Turtle.Move(10);
   end;
end;
```

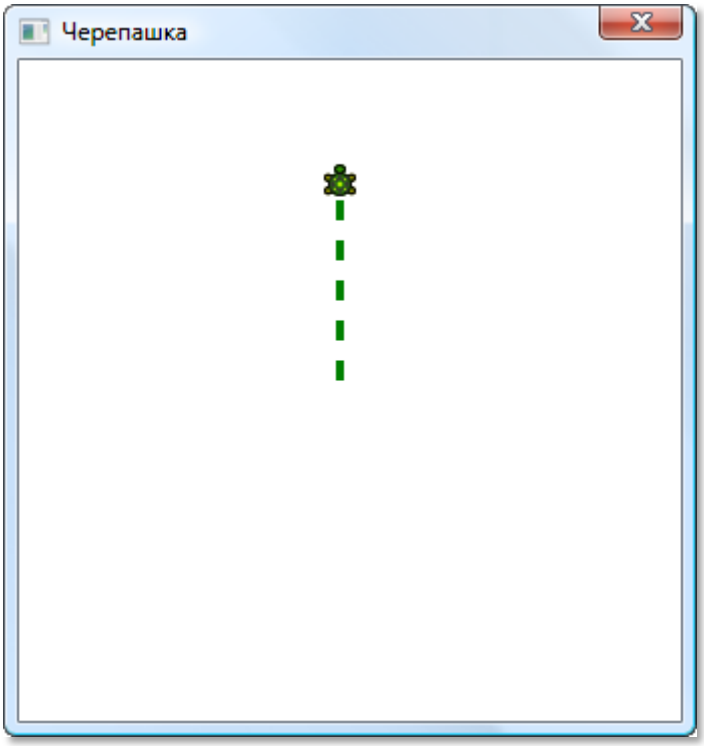


Рис. 12.7. *Черепашка* оставляет за собой следы

Иногда бывает полезно **спрятать** *Черепашку* после того как она выполнила свою работу, что часто бывает и в жизни: исполнители не должны заслонять руководящие кадры. Вызвав метод

```
Turtle.Hide()
```

вы превратите *Черепашку* в *Черепашку-невидимку*. И что особенно приятно, она при этом не утратит способности вычерчивать линии (Рис. 12.8).

Несмотря на то, что *Черепашка*, как это у них, у черепашек, и водится, довольно медлительна, но, задав свойству

Turtle. Speed

значение десять или близкое к десяти, вы слегка приободрите её. Вы можете выбирать произвольные значения, но у *Черепашки* всего 10 скоростей – от 1 до 10.

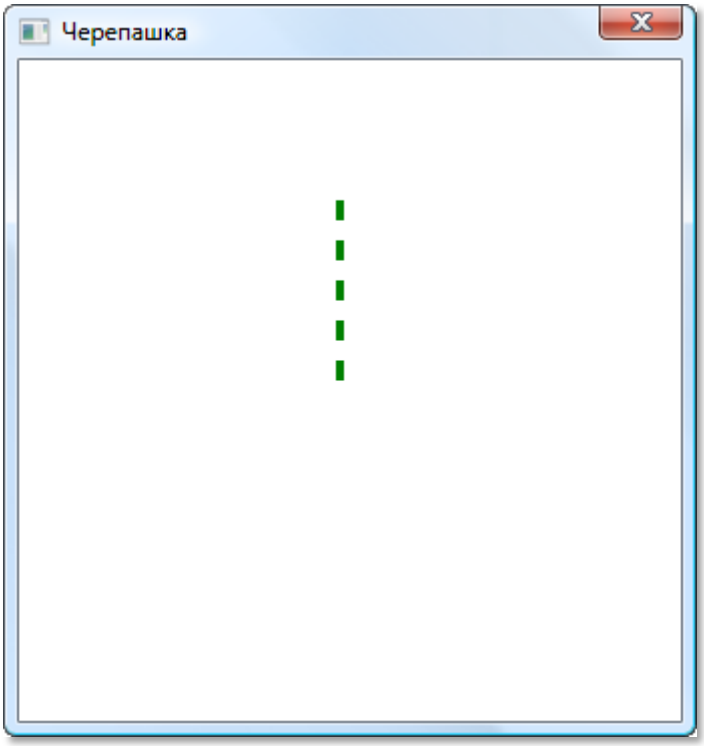


Рис. 12.8. *Черепашка* спряталась

Наша *Черепашка* может перемещаться в указанную точку канвы, то есть попросту чертить линию от точки, в которой она находится, до любой другой. Для этого при вызове метода

Turtle. MoveTo(**x**, **y** : double);

следует указать прямоугольные координаты новой точки.

После установки *Черепашки* в центре окна давайте-ка погоняем её по канве:

```
public procedure Ugol();
begin
   self.Show();
   GraphicsWindow.PenColor := 'Green';
   GraphicsWindow.PenWidth := 4;

   Turtle.MoveTo(300,300);
   Turtle.MoveTo(20,300);
end;
```

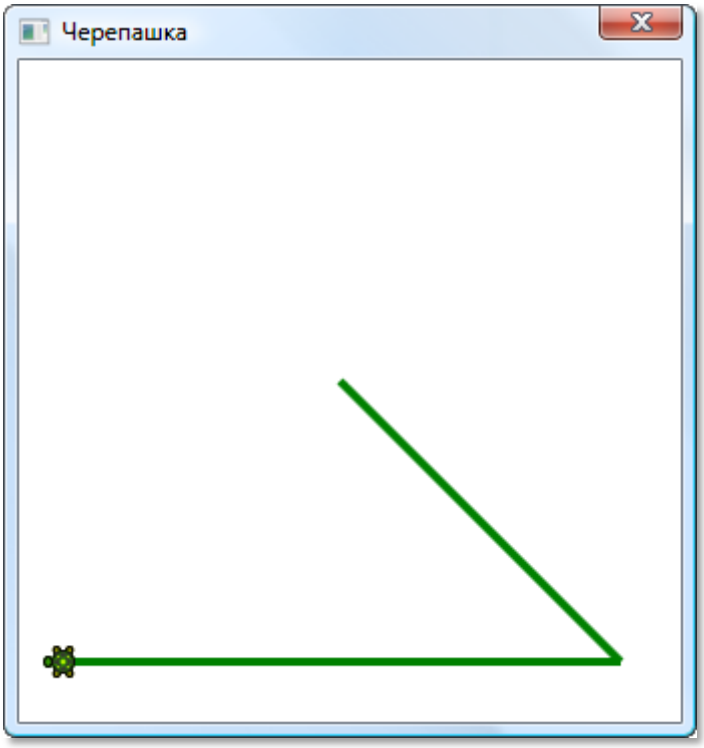


Рис. 12.9. *Черепашка* выполняет команды *MoveTo*

Очень забавно наблюдать, как *Черепашка* самостоятельно поворачивается так, чтобы её глаза были направлены в заданную точку (Рис. 12.9). Значит, она не вслепую перемещается по канве, а именно так, как мы и договаривались в начале главы!

Таким образом, *Черепашка* неплохо ориентируется и в прямоугольной системе координат. Однако от неё было бы не много пользы, если бы она просто перемещалась от одной точки к другой в этой системе координат.

Проект *Полярная Черепашка*

Исходный код программы находится в папке **Полярная Черепашка**.

Главное достоинство *Черепашки* в том, что она может передвигаться и в *полярной* системе координат. Действительно, при запуске программы мы можем установить *Черепашку* в центре окна. Будем считать, что это *полюс* полярной системы координат.

Три метода класса *Turtle:*

```
Turtle.TurnLeft();
```

```
Turtle.TurnRight();
```

```
Turtle.Turn(угол : double);
```

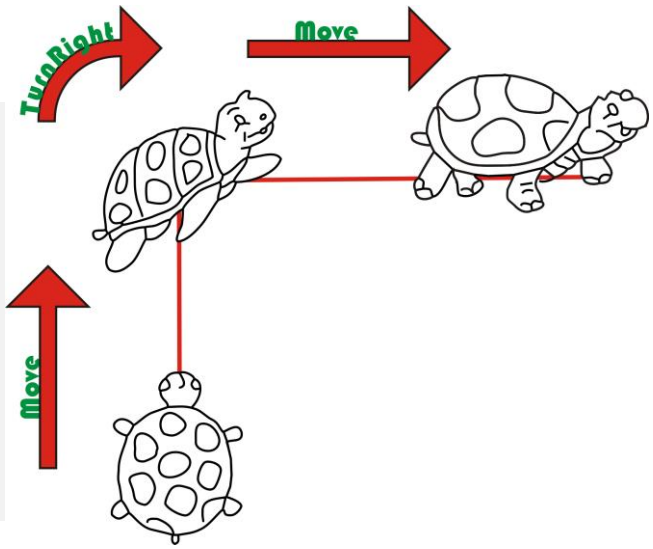
поворачивают *Черепашку* вокруг собственной оси. Первый метод – на 90 градусов *против* часовой стрелки, второй – на 90 градусов *по* часовой стрелке, третий – на произвольный угол. В последнем случае *Черепашка* повернётся против часовой стрелки, если угол отрицательный, и по часовой стрелке – если положительный.

Например, мы легко заставим нашу *Черепашку* танцевать брейк:

```
// брейк:
```

```
public procedure Breakdance();
begin
   self.Show();
   for var i := 1 to 10 do
   begin
      Turtle.TurnLeft();
      Turtle.TurnRight();
   end;
end;
```

Или так:




```
public procedure Breakdance2();
begin
   self.Show();
   for var i := 1 to 10 do
   begin
      Turtle.Turn(300);
      Turtle.Turn(-320);
   end;
end;
```

С помощью этих методов вы легко сможете переместить *Черепашку* в любую точку, заданную в полярных координатах. Например, выбрав полярный угол и полярный радиус

```
public procedure TurnMove();
begin
   self.Show();
   GraphicsWindow.PenColor := 'Green';
   GraphicsWindow.PenWidth := 4;
   var Z := 45;
   var R := 120;
   Turtle.Turn(Z);
   Turtle.Move(R);
end;
```

вы отправите *Черепашку* в точку, координаты которой заданы в полярной системе координат (Рис. 12.10).

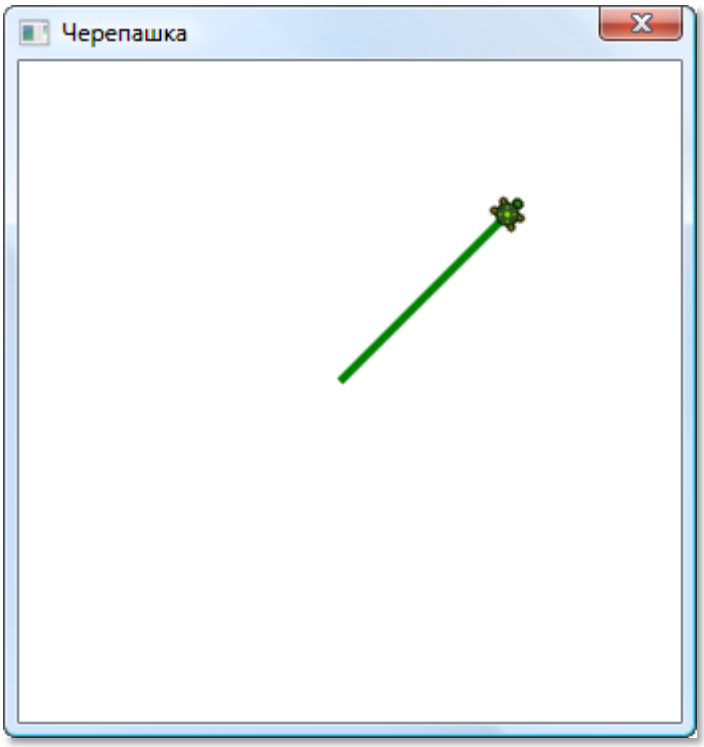


Рис. 12.10. Полярная *Черепашка*

Всякий раз возвращая *Черепашку* в начало координат и поворачивая её головой вверх, вы можете наставить сколько угодно точек в полярной системе координат:

```
public procedure Dots8();
begin
   self.Show();
   GraphicsWindow.PenColor := 'Green';
   GraphicsWindow.PenWidth := 4;
   var Z := 0;
   var R := 120;
   Turtle.Turn(Z);
   for var i := 1 to 8 do
   begin
      Turtle.PenUp();
      Turtle.Move(R - 4);
      Turtle.PenDown();
      Turtle.Move(4);
      Turtle.PenUp();
      Turtle.MoveTo(CX, CY);
      Turtle.Angle := Z;
      Turtle.Turn(360 / 8 * i);
   end;
end;
```

Например, вы можете расставить точки в вершинах правильного восьмиугольника (Рис. 12.11).

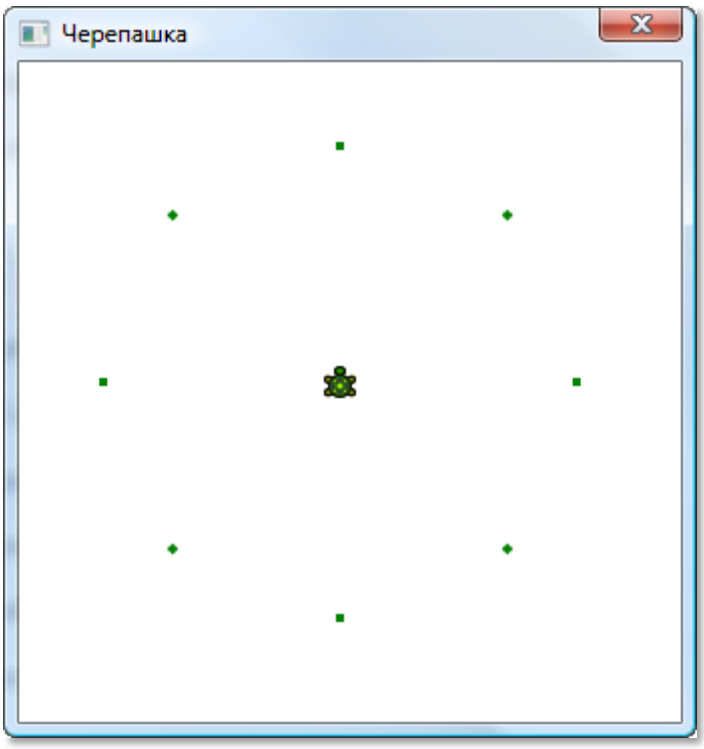


Рис. 12.11. *Черепашка* ставит точки в полярной системе координат

А что получится, если продолжить перемещения *Черепашки*, не возвращая её в прежнее положение? – Тогда начало полярных координат переместится в точку, где находится *Черепашка*, а полярная ось повернётся по направлению взгляда *Черепашки*. Следующий поворот будет отсчитываться уже не от горизонтальной оси, а от *нового* положения полярной оси.

Вы знаете, что все внутренние углы правильного треугольника равны 60 градусов, а внешние – 120, поэтому легко построите **правильный треугольник** (Рис. 12.12), если вычертите три его стороны, начиная каждую новую из конца предыдущей. Так как длина всех сторон одинакова, то достаточно повернуть *Черепашку* на нужный угол:

```
// треугольник:
public procedure Triangle();
begin
   self.Show();
   GraphicsWindow.PenColor := 'Green';
   GraphicsWindow.PenWidth := 4;
   var R := 120;
   Turtle.TurnRight();
   for var i := 1 to 3 do
```

```
   begin
      Turtle.Turn(-120);
      Turtle.Move(R);
   end;
end;
```

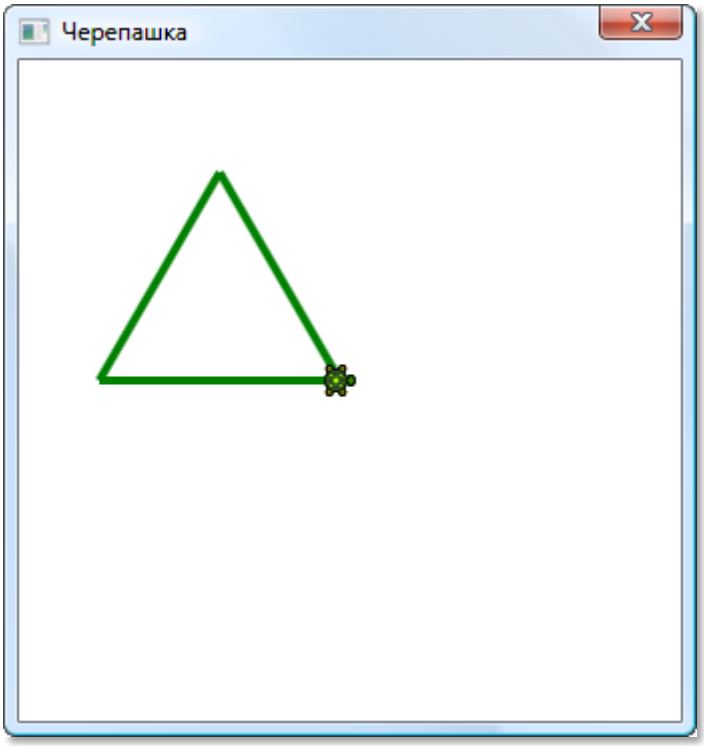


Рис. 12.12. *Черепашка* построила правильный треугольник

Мы сделаем формулу более универсальной, если введём параметр **n**, равный числу сторон правильного многоугольника:

```
// многоугольники:
public procedure Polygons();
begin
   //self.Show();
   GraphicsWindow.PenColor := 'Green';
   GraphicsWindow.PenWidth := 4;
   Turtle.X := width - 100;
   Turtle.Y := height - 20;
   var R := 120;
   Turtle.TurnRight();
   var n := 6;//8 6;
   for var i := 1 to n do
   begin
      Turtle.Turn(-360 / n);
      Turtle.Move(R);
   end;
end;
```

Для построения *любого* правильного многоугольника (Рис. 12.13) нам потребовалось написать всего несколько строк кода и при этом – никаких расчётов координат вершин!

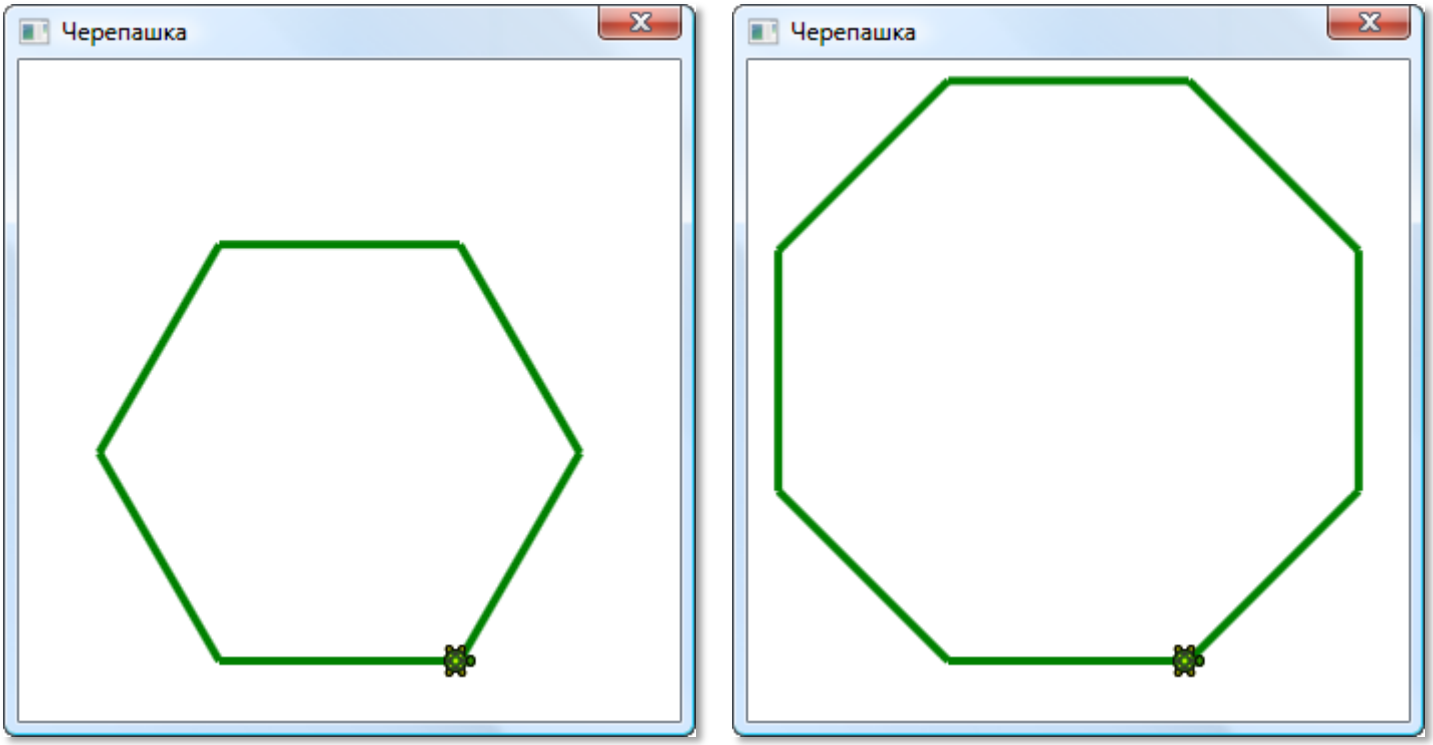


Рис. 12.13. *Черепашка* построила правильные многоугольники

Нетрудно догадаться, что если задать достаточно большое число вершин, то многоугольник превратится в **окружность** (Рис. 12.14).

```
// окружность:
public procedure Circle();
begin
   GraphicsWindow.PenColor := 'Green';
   GraphicsWindow.PenWidth := 4;
   Turtle.X := width - 150;
   Turtle.Y := height - 5;
   var R := 12;
   Turtle.TurnRight();
   var n := 80;
   for var i := 1 to n do
   begin
      Turtle.Turn(-360 / n);
      Turtle.Move(R);
   end;
end;
```

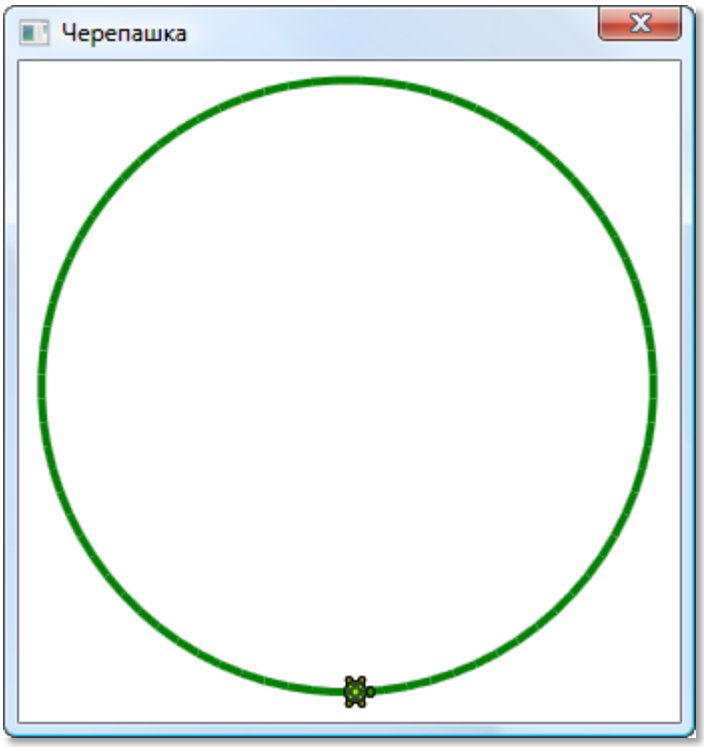
Черепашке нетрудно вычертить и **спираль** (Рис. 12.15).

```
// спираль:
public procedure Spiral();
begin
   self.Show();
   GraphicsWindow.PenColor := 'Green';
   GraphicsWindow.PenWidth := 4;
   var R := 1.0;
   Turtle.TurnRight();
   var n := 240;
   for var i := 1 to n do
   begin
      Turtle.Move(R);
      Turtle.Turn(10);
      R += 0.1;
   end;
end;
```

Если при повороте *Черепашки* задать *отрицательный* угол

`Turtle.Turn(-10),`

то вместо правой спирали получится **левая** (она закручена в противоположном направлении).



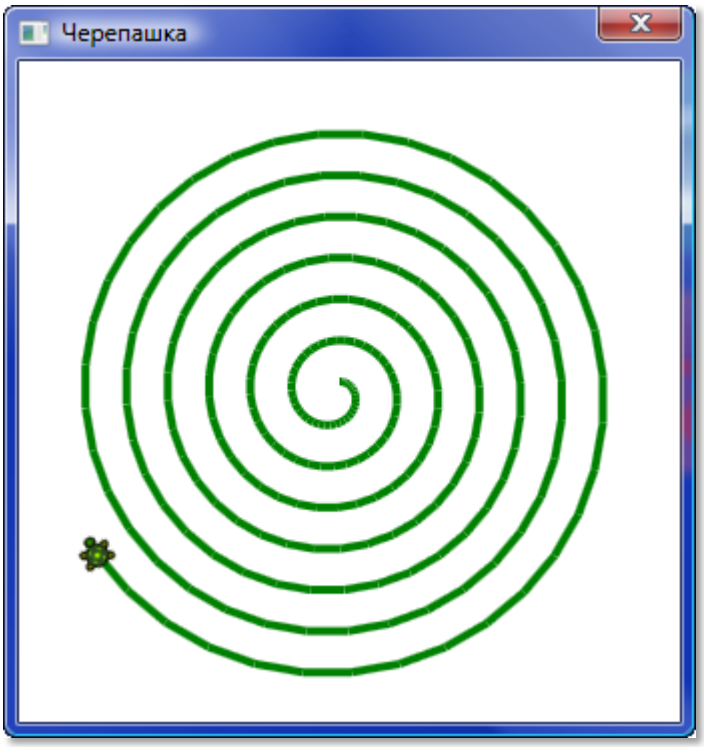


Рис. 12.14. Черепашка чертит окружность **Рис. 12.15.** Черепашка рисует спираль

Проект *Черепашьи загогулины*

Исходный код программы находится в папке **Черепашьи загогулины**.

А теперь давайте перейдём к более сложным кривым, которые наверняка убедят вас, что вы не напрасно познакомились с *Черепашкой.*

Для начала мы заставим многоугольники **вращаться** вокруг одной из своих вершин:

```
private procedure PolyStop();
begin
   repeat
      Turtle.Move(side);
      Turtle.Turn(angle1);
      _turn += angle1;
   until not (_turn mod 360 <> 0);
end;

// Вращающиеся фигуры
private procedure PolyRoll();
begin
   for var i := 1 to 12 do
   begin
      PolyStop();
      Turtle.Turn(angle2);
   end
end;
```

Изменяя значения переменных *angle1* и *angle2*, вы получите разные многоугольники, вращающиеся вокруг центра окна (Рис. 12.16).

```
// Вращающиеся многоугольники:
public procedure RotatePolygons();
begin
   self.Show();
   GraphicsWindow.PenColor := 'Black';
   GraphicsWindow.PenWidth := 2;
   _turn := 0;
   angle1 := 90;
   angle2 := 30;
   side := 100;
```

```
   PolyRoll();
end;
```

Или:

```
public procedure RotatePolygons2();
begin
   self.Show();
   GraphicsWindow.PenColor := 'Black';
   GraphicsWindow.PenWidth := 2;
   _turn := 0;
   angle1 := 60;
   angle2 := 45;
   side := 80;
   PolyRoll();
end;
```

```
{$apptype windows}


uses
   DrawUnit;

begin
   var draw := new Draw();
   draw.Prepare();
   //draw.RotatePolygons();
   draw.RotatePolygons2();
end.

unit DrawUnit;

uses
   Microsoft.SmallBasic.Library, System;

type
   Draw = class

      const GWWIDTH = 320;
      const GWHEIGHT = 320;

      width := GWWIDTH;
      height := GWHEIGHT;

      CX := width div 2;
      CY := height div 2;
```

```
_turn: integer;
angle1: integer;
angle2: integer;
side: integer;

public procedure Prepare();
begin
   GraphicsWindow.Hide();
   GraphicsWindow.Title := 'Черепашка';
   GraphicsWindow.Width := GWWIDTH;
   GraphicsWindow.Height := GWHEIGHT;
   GraphicsWindow.Show();
   GraphicsWindow.Left := (Desktop.Width –
                           GraphicsWindow.Width) / 2;
   GraphicsWindow.Top := (Desktop.Height –
                           GraphicsWindow.Height) / 2;
   GraphicsWindow.CanResize := false;
   GraphicsWindow.BackgroundColor := 'White';
end;

private procedure Show();
begin
    // создаём Черепашку:
   Turtle.Show();
    // устанавливаем Черепашку в центре окна:
   Turtle.X := CX;
   Turtle.Y := CY;
end;

private procedure PolyStop();
begin
   repeat
      Turtle.Move(side);
      Turtle.Turn(angle1);
      _turn += angle1;
   until not (_turn mod 360 <> 0);
end;

// Вращающиеся фигуры
private procedure PolyRoll();
begin
   for var i := 1 to 12 do
   begin
      PolyStop();
      Turtle.Turn(angle2);
   end
end;

// Вращающиеся многоугольники:
public procedure RotatePolygons();
begin
```

```
            self.Show();
            GraphicsWindow.PenColor := 'Black';
            GraphicsWindow.PenWidth := 2;
            _turn := 0;
            angle1 := 90;
            angle2 := 30;
            side := 100;
            PolyRoll();
        end;

        public procedure RotatePolygons2();
        begin
            self.Show();
            GraphicsWindow.PenColor := 'Black';
            GraphicsWindow.PenWidth := 2;
            _turn := 0;
            angle1 := 60;
            angle2 := 45;
            side := 80;
            PolyRoll();
        end;

    end; //end of class
end.
```

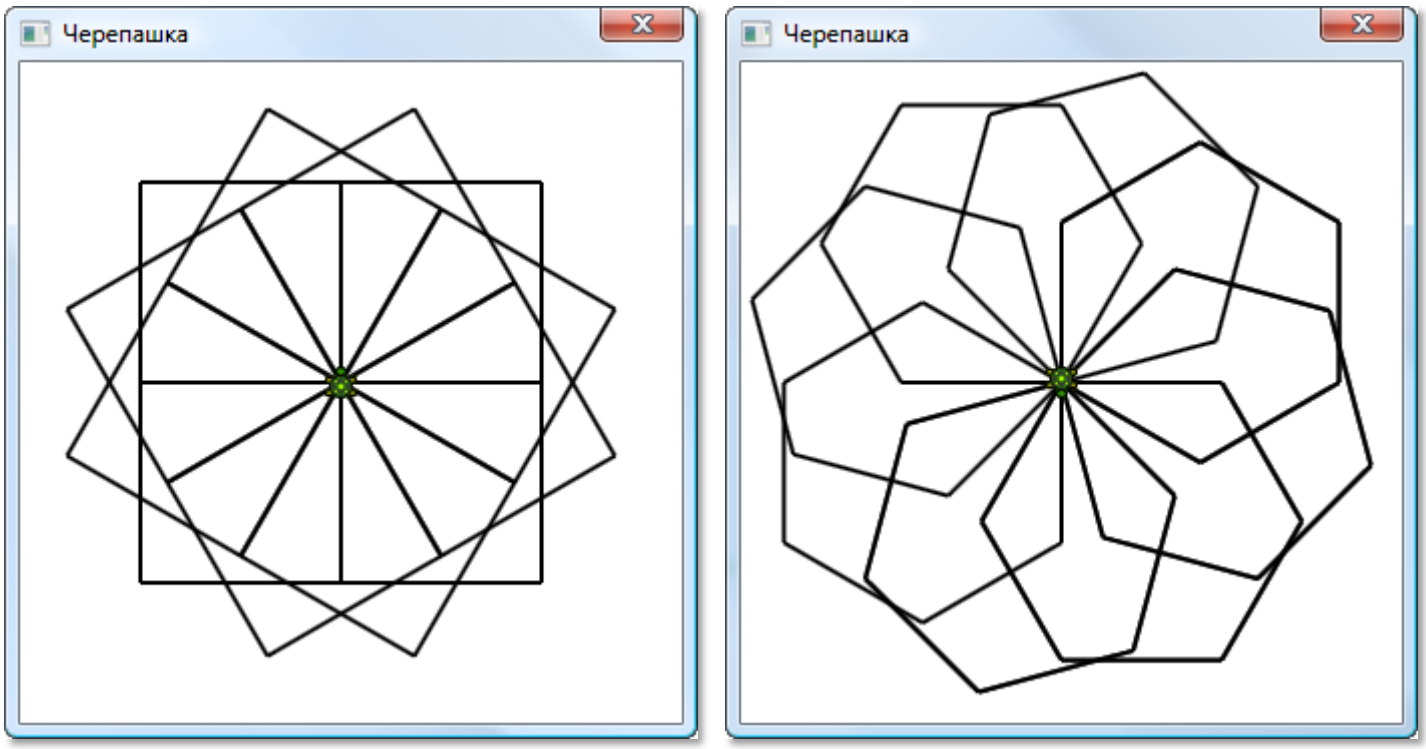


Рис. 12.16. Вращающиеся фигуры

Проект *Рекурсивная Черепашка*

Исходный код программы находится в папке **Рекурсивная Черепашка**.

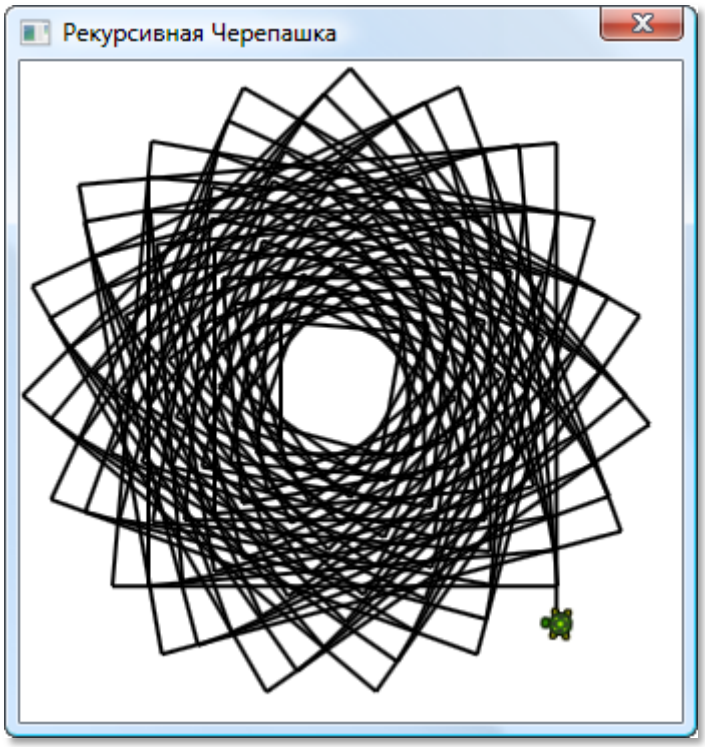
Необыкновенно простой рекурсивный метод **PolySpi** при разных параметрах даёт потрясающее многообразие великолепных *спиралей* (Рис. 12.17)!

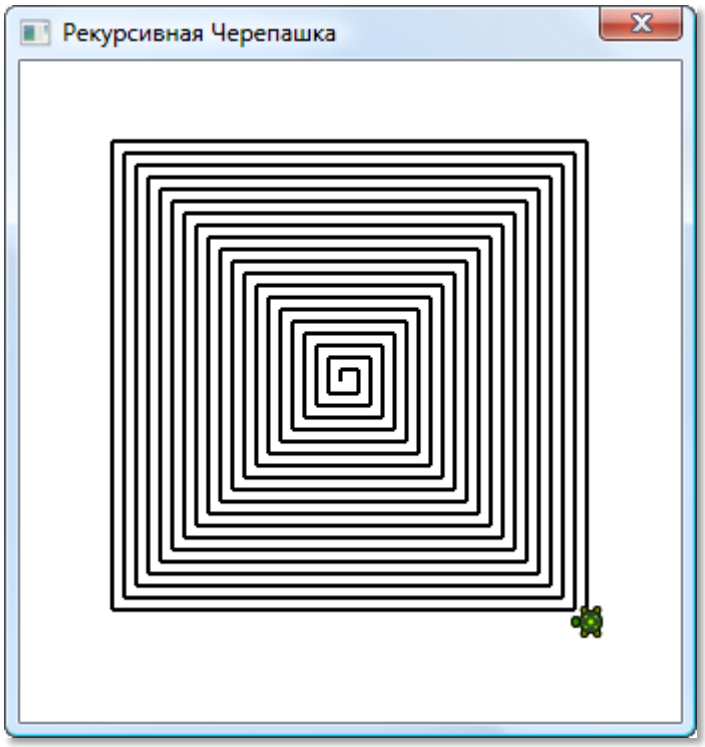
```
{$apptype windows}

uses
   DrawUnit;

begin
   var draw := new Draw();
   draw.Prepare();
   //draw.Spiral1();
   //draw.QSpiral();
   //draw.TSpiral1();
   //draw.TSpiral2();
   draw.Spiral2();
end.

unit DrawUnit;
uses
   Microsoft.SmallBasic.Library, System;

type
   Draw = class

      const GWWIDTH = 320;
      const GWHEIGHT = 320;
      width := GWWIDTH;
      height := GWHEIGHT;
      CX := width div 2;
      CY := height div 2;
      angle: double;
      inc: integer;
      side: double;
      maxSide: double;

      public procedure Prepare();
      begin
         GraphicsWindow.Hide();
         GraphicsWindow.Title := 'Рекурсивная Черепашка';
         GraphicsWindow.Width := GWWIDTH;
         GraphicsWindow.Height := GWHEIGHT;
```

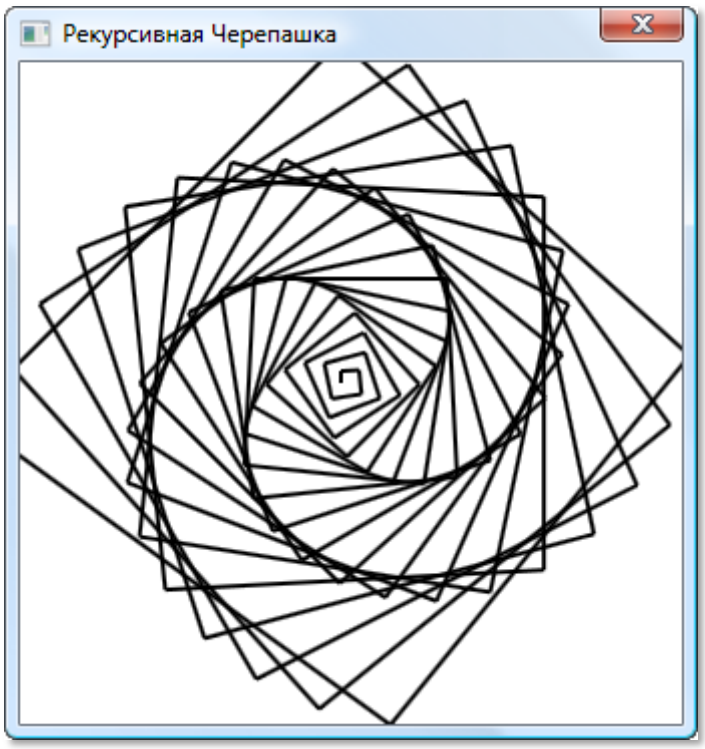
```
        GraphicsWindow.Show();
        GraphicsWindow.Left := (Desktop.Width –
                                GraphicsWindow.Width) / 2;
        GraphicsWindow.Top := (Desktop.Height –
                               GraphicsWindow.Height) / 2;
        GraphicsWindow.CanResize := false;
        GraphicsWindow.BackgroundColor := 'White';
    end;

    private procedure Show();
    begin
        // создаём Черепашку:
        Turtle.Show();
        // устанавливаем Черепашку в центре окна:
        Turtle.X := CX;
        Turtle.Y := CY;
    end;

    // Рекурсивные спирали
    private procedure PolySpi();
    begin
        if (side > maxside) then
            exit;
        Turtle.Move(side);
        Turtle.Turn(angle);
        side += inc;
        PolySpi();
    end;

    // рекурсивная спираль 1:
    public procedure Spiral1();
    begin
        // устанавливаем Черепашку:
        Turtle.X := CX - 30;
        Turtle.Y := CY + 30;
        GraphicsWindow.PenColor := 'Black';
        GraphicsWindow.PenWidth := 2;
        Turtle.Speed := 10;
        maxside := 240;
        inc := 1;
        side := 60;
        angle := 95;
        PolySpi();
    end;

  end; //end of class
end.

    // квадратная спираль:
    public procedure QSpiral();
    begin
```

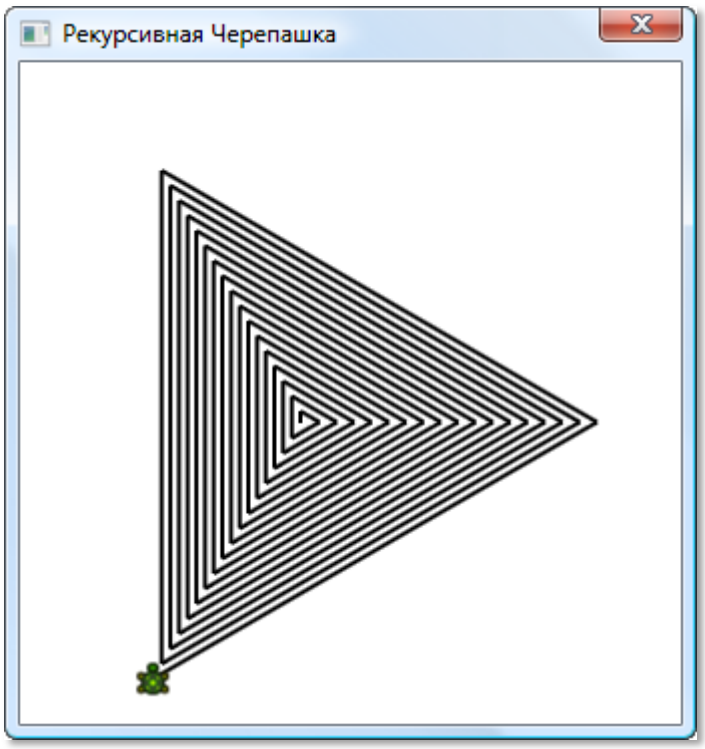
```
        // устанавливаем Черепашку:
        Turtle.X := CX;
        Turtle.Y := CY;
        GraphicsWindow.PenColor := 'Black';
        GraphicsWindow.PenWidth := 2;
        Turtle.Speed := 10;
        maxside := 240;
        inc := 3;
        side := 6;
        angle := 90;
        //angle := 87;
        PolySpi();
    end;

    public procedure TSpiral1();
    begin
        // устанавливаем Черепашку:
        Turtle.X := CX-20;
        Turtle.Y := CY+20;
        GraphicsWindow.PenColor := 'Black';
        GraphicsWindow.PenWidth := 2;
        Turtle.Speed := 10;
        maxside := 260;
        inc := 5;
        side := 6;
        angle := 120;
        PolySpi();
    end;

    public procedure TSpiral2();
    begin
        // устанавливаем Черепашку:
        Turtle.X := CX;
        Turtle.Y := CY+20;
        GraphicsWindow.PenColor := 'Black';
        GraphicsWindow.PenWidth := 2;
        Turtle.Speed := 10;
        maxside := 300;
        inc := 4;
        side := 6;
        angle := 117;
        PolySpi();
    end;

    // рекурсивная спираль 2:
    public procedure Spiral2();
    begin
        // устанавливаем Черепашку:
        Turtle.X := CX;
        Turtle.Y := CY + 20;
        GraphicsWindow.PenColor := 'Black';
```

```
        GraphicsWindow.PenWidth := 2;
        Turtle.Speed := 10;
        maxside := 280;
        inc := 1;
        side := 6;
        angle := 30;
        PolySpi();
    end;
```









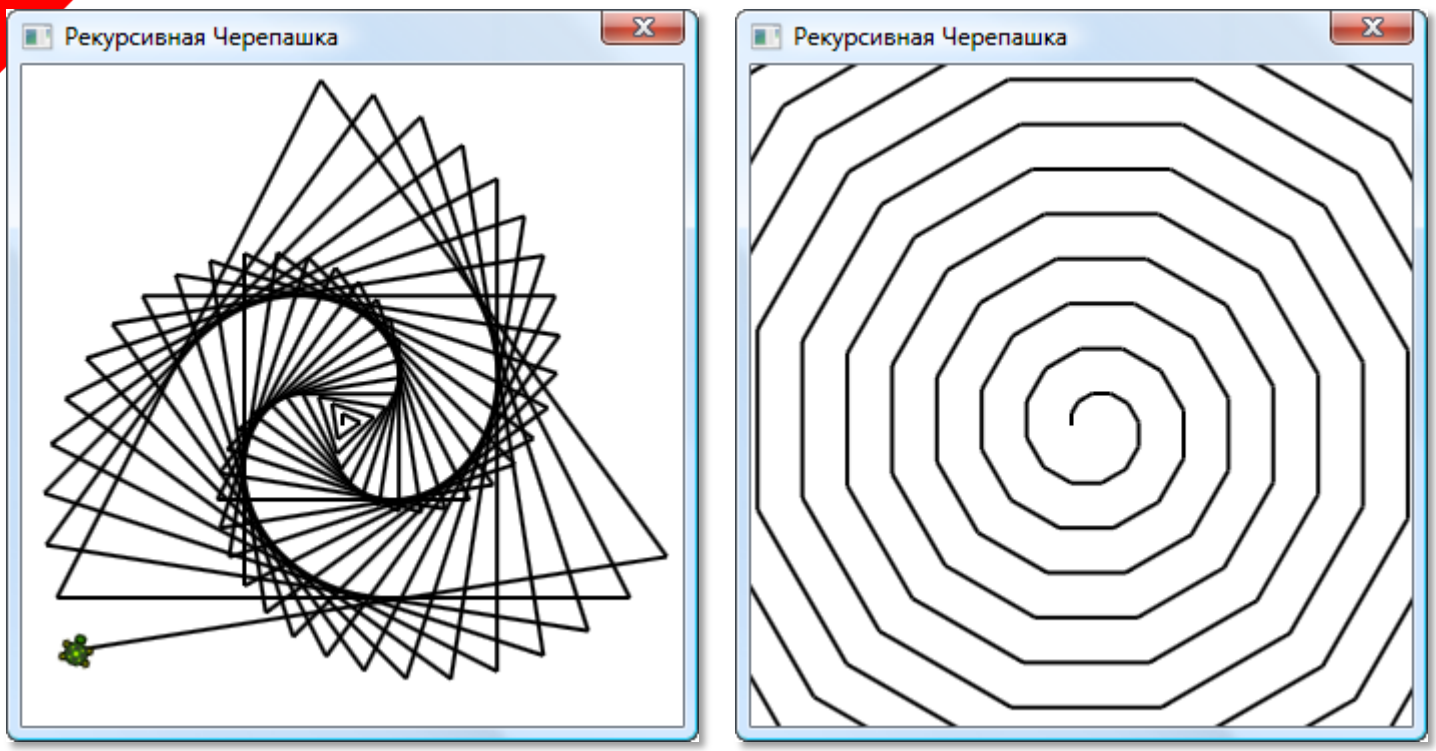


Рис. 12.17. Рекурсивные спирали

Проект *Звёздчатая Черепашка*

Исходный код программы находится в папке **Звёздчатая Черепашка**.

И последний проект этой главы – вычерчивание разнообразных **звёздчатых многоугольников** (Рис. 12.18).

```
{$apptype windows}

// Звёздчатые многоугольники

uses
   DrawUnit;

begin
   var draw := new Draw();
   draw.Prepare();
   //draw.Star();
   draw.Star2();
```

```
end.


// зубчатое колесо:
public procedure Star2();
begin
   // устанавливаем Черепашку:
   Turtle.X := CX - 70;
   Turtle.Y := CY - 100;
   Turtle.Angle := 0;
   GraphicsWindow.PenColor := 'Black';
   GraphicsWindow.PenWidth := 2;
   Turtle.Speed := 10;
   side := 36;
   angle := 125;
   NewPoly(24);
end;
```

Количество **зубцов** определяется углом *angle*.

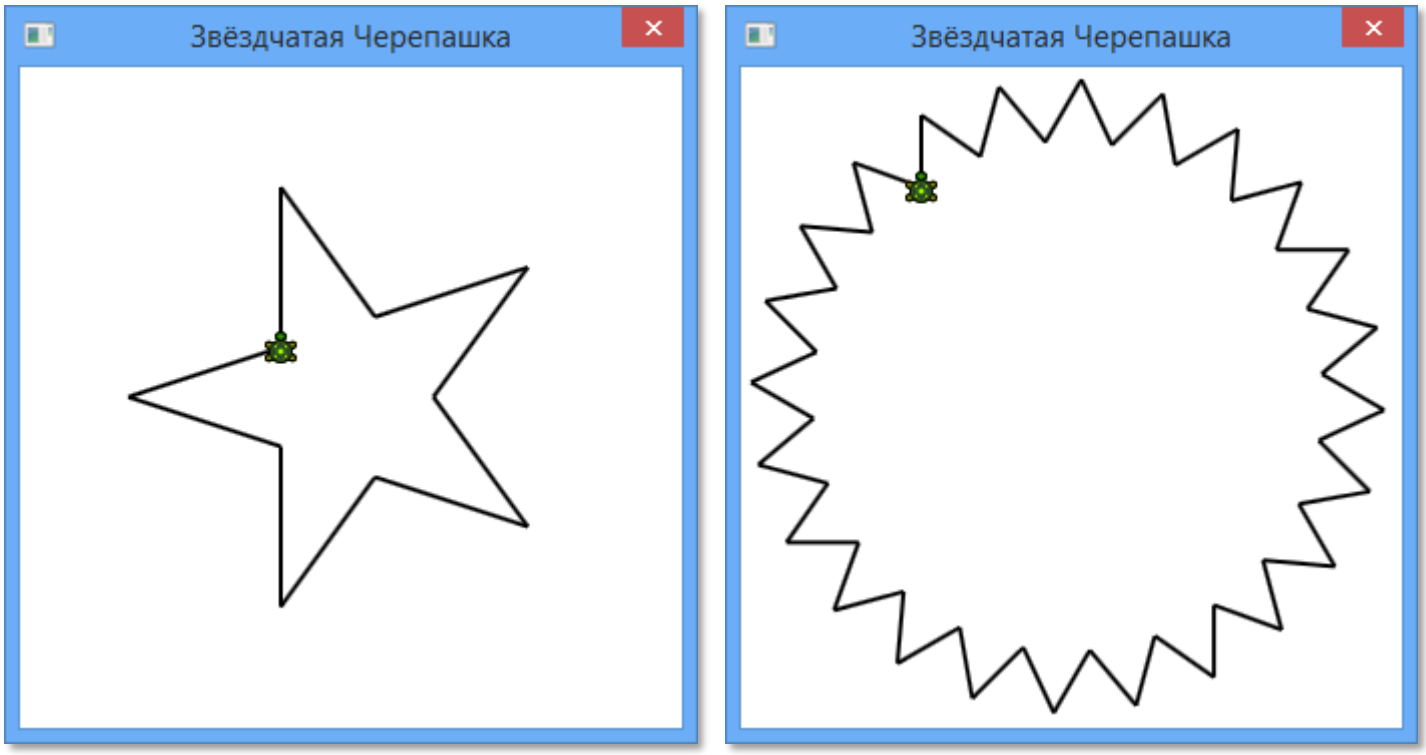


Рис. 12.18. Звёздчатые многоугольники

Задания для самостоятельного решения

Черепашка

Напишите такие программы для *Черепашки*, чтобы она построила следующие фигуры (Рис. 12.19).

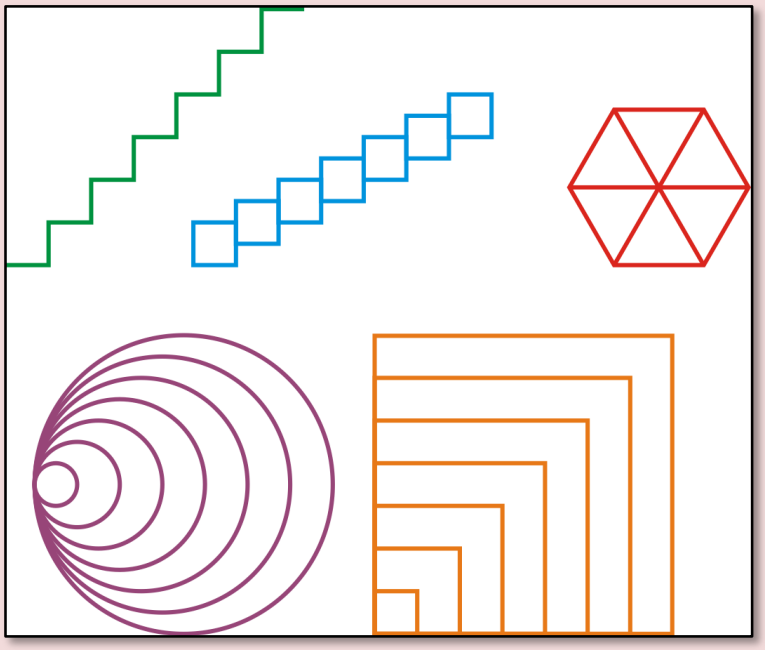


Рис. 12.19. Черепашьи фигуры

Глава 13. Фрактальная Киберчерепашка

В предыдущей главе *Черепашка* выполняла наши команды, записанные на языке *паскаль*. Но ведь в каждом живом существе, в его геноме записана информация о том, как оно должно расти и развиваться, а также инстинкты, управляющие поведением достаточно примитивных организмов. *Черепашка* также может выполнять не только внешнюю программу, но и внутреннюю – записанную в её геноме и реализуемую через инстинкты. Эту главу мы и посвятим моделированию такой *кибернетической Черепашки*.

Проект *Кибер-Черепашка*

Исходный код программы находится в папке **Кибер-Черепашка**.

Черепашка рождается в условленном месте, обозначенном координатами **x0** и **y0**, а её голова направлена в сторону света, заданную начальным углом **a0**:

```
unit DrawUnit;

uses
   Microsoft.SmallBasic.Library, System;

type
   Draw = class
   private
      const GWWIDTH = 320;
      const GWHEIGHT = 320;
      // размеры окна:
      width := GWWIDTH;
      height := GWHEIGHT;
      // координаты центра окна:
      CX := width div 2;
      CY := height div 2;

      // начальное положение Черепашки:
      x0 := 0.0;
      y0 := 0.0;
      a0 := 0.0;
```

Основное предназначение *Черепашки* (судьба) - перемещение по плоскости и вычерчивание линий, поэтому нам необходимо задать её основные **свойства**:

```
      // длина единичного перемещения Черепашки:
      size := 0.0;
      // угол единичного поворота Черепашки:
      teta := 0;
      //скорость движения Черепашки:
      speed := 9;
      //цвет линии:
      penColor := 'Blue';
      //толщина линии:
      penWidth := 2;
      // список команд, которые должна выполнить Черепашка:
      instinct: string;
```

Здесь всё просто и понятно, кроме, пожалуй, **инстинкта**, который и определяет поведение *Черепашки*. Дальше мы зададим ей жизненную программу (инстинкт), и вы сможете сами программировать *Черепашку*.

Настройка **окна** приложения не должна вызвать у вас вопросов:

```
// координаты центра окна:
int CX;
int CY;
// размеры окна:
int height;
int width;

     public procedure Prepare();
      begin
         GraphicsWindow.Hide();
         GraphicsWindow.Title := 'КиберЧерепашка - CyberTurtle';
         GraphicsWindow.Width := GWWIDTH;
         GraphicsWindow.Height := GWHEIGHT;
         GraphicsWindow.Show();
         GraphicsWindow.Left := (Desktop.Width –
                                  GraphicsWindow.Width) / 2;
         GraphicsWindow.Top := (Desktop.Height –
                                 GraphicsWindow.Height) / 2;
         GraphicsWindow.CanResize := false;
         GraphicsWindow.BackgroundColor := 'White';
      end;
```

Теперь мы вполне можем перейти к программированию инстинкта *Черепашки*. Пусть её жизненный путь начнётся с **прямой линии**. Задаём место рождения *Черепашки*:

```
public procedure Line();
begin
   // устанавливаем Черепашку:
   x0 := CX - 100;
   y0 := CY + 100;
```

Её направление:

```
a0 := 0;
```

Длину «шага»:

```
size := 100;
```

И самое главное – **инстинкт**:

```
// прямая линия:
instinct := 'F';
```

Процесс рождения *Черепашки* не зависит от самой *Черепашки*, а полностью лежит на совести её родителей, то есть на нас с вами:

```
// Черепашка занимает исходное положение
private procedure Start();
begin
   Turtle.Speed := speed;
   GraphicsWindow.PenColor := penColor;
   GraphicsWindow.PenWidth := penWidth;
   Turtle.PenDown();
   Turtle.X := x0;
   Turtle.Y := y0;
   Turtle.Angle := a0;
   Turtle.Show();
```

```
end;
```

В большинстве языков программирования, поддерживающих черепашью графику, движение *Черепашки* вперёд обозначается командой ***F****orward*. Первую букву этого слова мы будем использовать для той же цели. После того как *Черепашка* заняла исходное положение, она должна выполнить те действия, которые ей предписывает инстинкт. Для этого у неё имеется мозг, который умеет обрабатывать и исполнять команды инстинкта:

```
// Черепашка выполняет инстинкт
private procedure Execute();
begin
   // Черепашка считывает команды:
   for var i := 1 to instinct.Length do
   begin
      // очередная команда:
      var cmd := instinct[i];
      // двигаться вперед на один шаг:
      if (cmd = 'F') then
         Turtle.Move(size);
   end
end;
```

Пока *Черепашка* маленькая, она может только двигаться вперёд, но и этого вполне достаточно, чтобы дать ей путёвку в жизнь:

```
   Start();
   Execute();
end;
```

Запускаем программу, и *Черепашка* проводит вертикальный отрезок длиной в *size* пикселей (Рис. 13.1).

На следующем этапе своей черепашьей жизни наша *Киберчерепашка* сможет выполнять команды **+** и **-**, то есть поворачиваться направо и налево на заданный угол *teta*. В её окрепшем мозгу появятся новые шишки для обработки более сложного инстинкта:

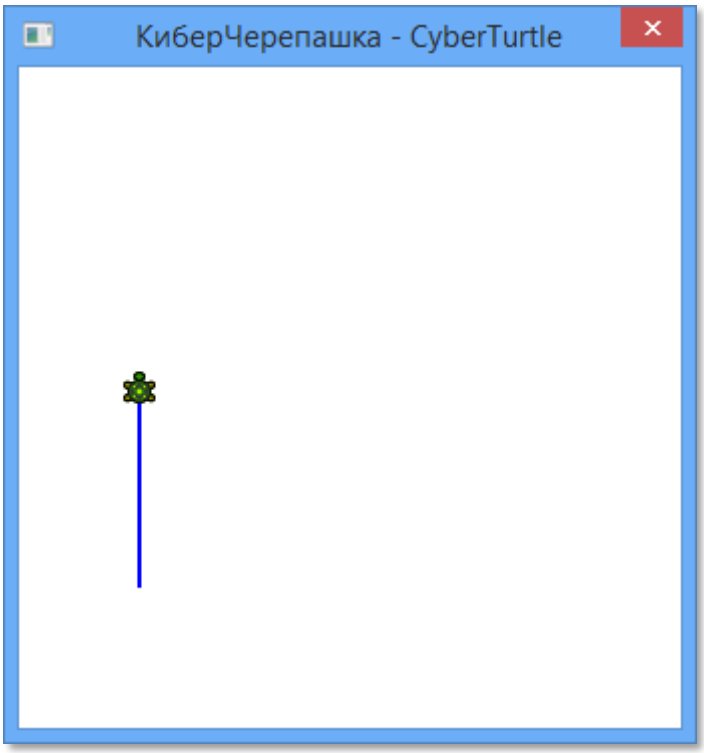


Рис. 13.1. *Черепашка* в ползунковом возрасте

```
private procedure Execute();
begin
   // Черепашка считывает команды:
   for var i := 1 to instinct.Length do
   begin
      // очередная команда:
      var cmd := instinct[i];
      // двигаться вперед на один шаг:
      if (cmd = 'F') then
         Turtle.Move(size)
      else if (cmd = '+') then
         Turtle.Turn(teta)
      else if (cmd = '-') then
         Turtle.Turn(-teta);
   end
end;
```

Треугольник вы уже вычерчивали с помощью обычной *Черепашки*. Как вы помните, для этого *Черепашка* должна вычертить одну сторону, повернуться на угол 120 градусов, вычертить вторую сторону, опять повернуться на 120 градусов и, наконец, вычертить третью сторону. Отсюда следует, что угол единичного поворота равен 120 градусам, а инстинкт запишется строкой:

```
// треугольник:
public procedure Triangle();
begin
   // устанавливаем Черепашку:
   x0 := CX - 100;
   y0 := CY + 100;
   a0 := 0;
   size := 100;

   teta := 120;
   instinct := 'F+F+F+';

   Start();
   Execute();
end;
```

Для проверки нашей гипотезы запускаем программу. *Черепашка* нас не подвела – треугольник удался на славу (Рис. 13.2)!

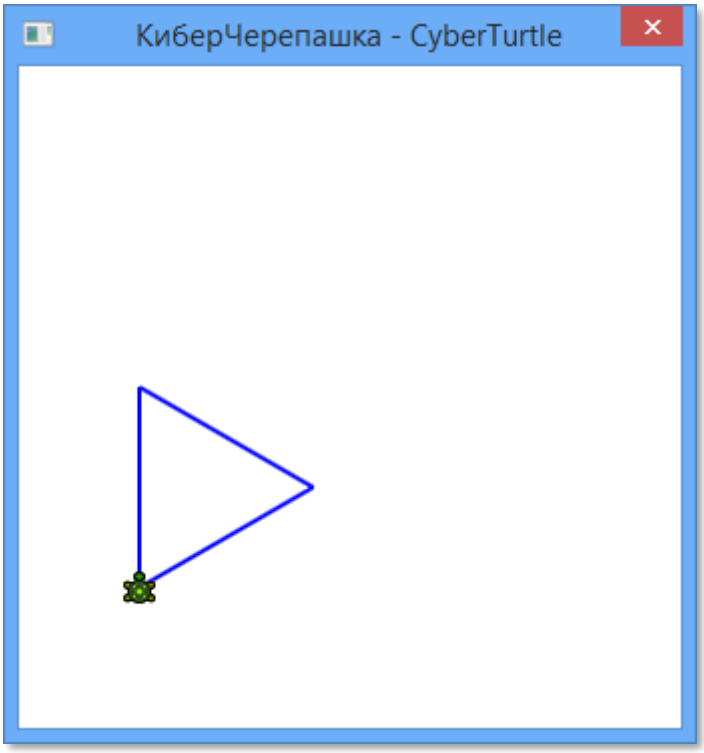


Рис. 13.2. Черепашковый треугольник

Совершенно аналогично вы можете сформировать инстинкты для вычерчивания любых правильных **многоугольников** (Рис. 13.3):

```
// квадрат:
public procedure Quadrat();
begin
   // устанавливаем Черепашку:
   x0 := CX - 100;
   y0 := CY + 100;
   a0 := 0;
   size := 100;

   teta := 90;
   instinct := 'F+F+F+F+';

   Start();
   Execute();
end;

// пятиугольник:
public procedure Pentagon();
begin
   // устанавливаем Черепашку:
   x0 := CX - 100;
   y0 := CY + 100;
   a0 := 0;
   size := 100;

   teta := 72;
   instinct := 'F+F+F+F+F+';

   Start();
   Execute();
end;

// шестиугольник:
public procedure Shestigon();
begin
   // устанавливаем Черепашку:
   x0 := CX - 100;
   y0 := CY + 100;
   a0 := 0;
   size := 100;

   teta := 60;
   instinct := 'F+F+F+F+F+F+';

   Start();
   Execute();
end;
```

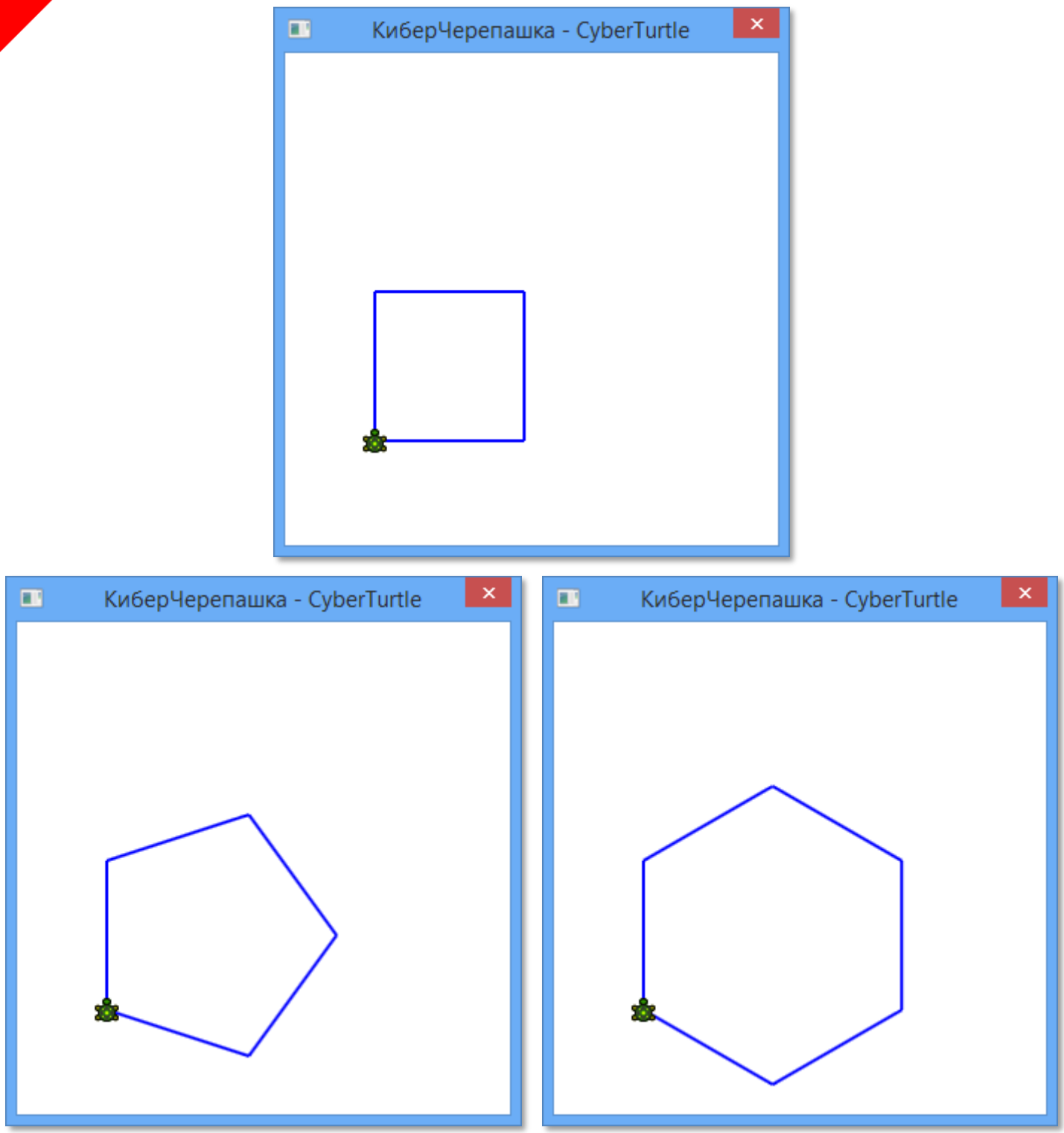


Рис. 13.3. Многоугольники

Как вы видите, инстинкт становится всё более длинным, при этом одна и та же пара команд *F+* просто повторяется *n* раз, если через *n* обозначить число вершин правильного многоугольника. Давайте научимся **выращивать инстинкты**!

Пусть при рождении *Черепашка* получает простейший инстинкт, который принято называть **аксиомой**, а затем, с каждым годом инстинкт развивается и становится всё более сложным благодаря *правилам* роста. Для треугольника и других правильных многоугольников аксиома самая простая:

```
axiom := 'F';
```

Она означает, что *Черепашка* должна двигаться вперёд на расстояние *size*. Наша первая *Черепашка* только это и умеет делать.

Поскольку все многоугольники строятся по одному и тому же сценарию, то и правило для них одно и то же:

```
newF := 'F+F';
```

А действует оно так. Через год (возраст *Черепашки* определяется переменной *iter*) каждая команда *F*, в соответствии с правилом роста, заменяется значением переменной *newF,* то есть *F+F*:

```
iter = 1 → axiom := 'F+F'
```

Это значит, что через год (а у *Черепашек* год короткий!) наша *Черепашка* научится вычерчивать **угол** (Рис. 13.4).

На второй год с поведение *Черепашки* усложнится:

```
iter = 2 → axiom := '(F)+(F)' → '(F+F)+(F+F)' → 'F+F+F+F'
```

Опять в аксиоме каждая буква *F* заменяется выражением *F+F*. Для удобства преобразований мы воспользовались скобками, но самой *Черепашке* они не нужны.

Вот теперь *Черепашка* может вычертить настоящий *треугольник* (Рис. 13.5).

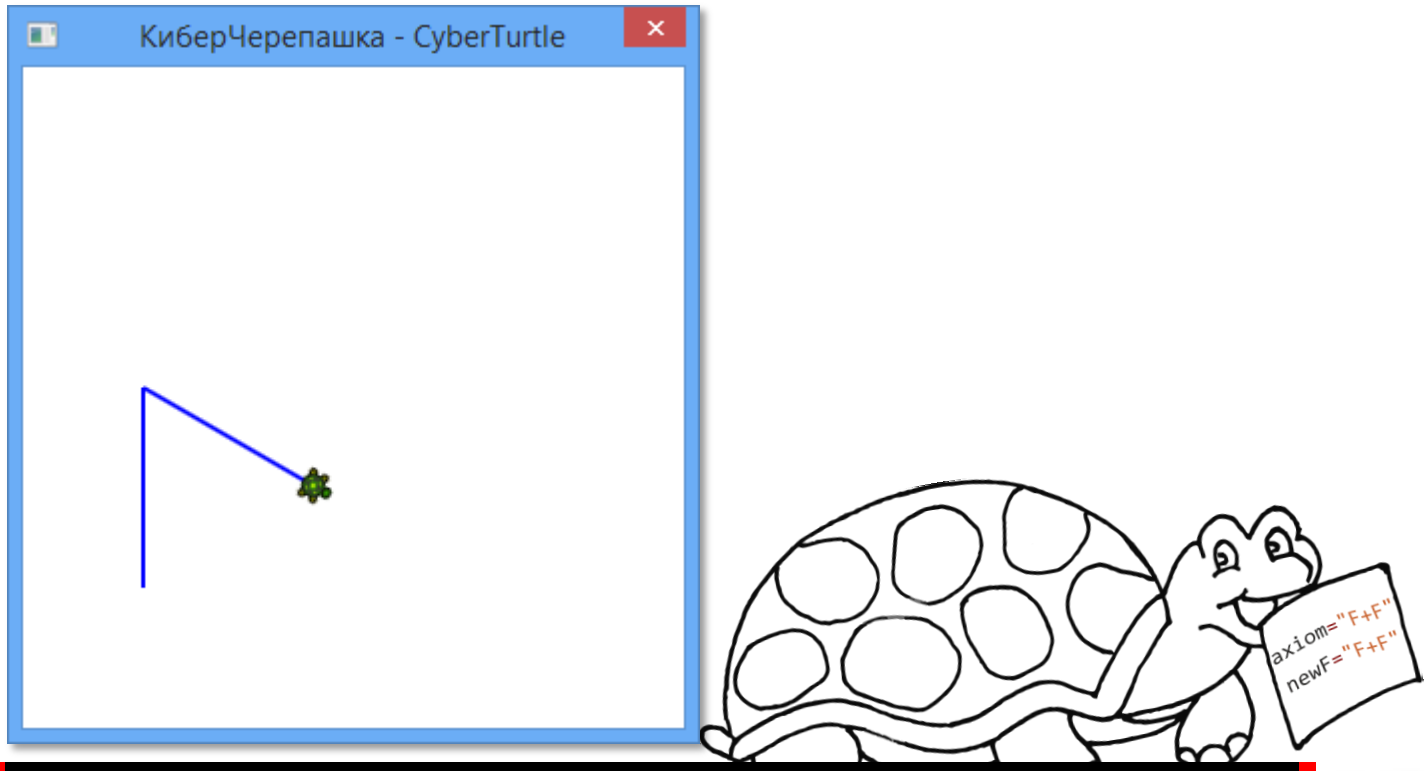


Рис. 13.4. Молодая *Черепашка* - «угловатая»

Вы, должно быть, заметили, что для построения треугольника достаточно инстинкта `'F+F+F+'`, то есть одну сторону *Черепашка* чертит дважды. Увы, все инстинкты несовершенны, но зато достаточно просты.

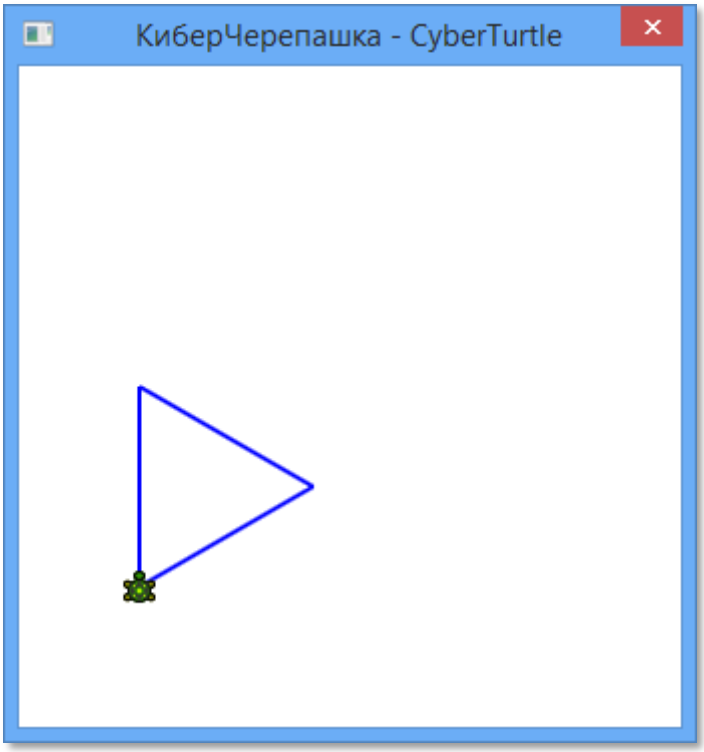


Рис. 13.5. Двухлетняя *Черепашка* умеет чертить треугольник

Инстинкт *Черепашки* формируется, конечно, не вручную - она имеет для этого в мозгу специальный **метод**:

```
// Черепашка формирует инстинкт
private procedure CreateInstinct();
begin
   for var i := 1 to iter do
   begin
      instinct := string.Empty;
      for var j := 1 to axiom.Length do
      begin
         // очередная команда аксиомы:
         var cmd := axiom[j];
         if (cmd = 'F') then
            instinct += newF
         else
            instinct += cmd;
      end;
      axiom := instinct;
   end
end;
```

Его действие мы уже разобрали. Каждая буква *F* заменяется значением переменной *newF*, а все остальные символы (команды поворота + и -) переходят в новый инстинкт без изменения. Так как мы привыкли использовать при моделировании поведения *Черепашки* переменную *instinct*, то результатом работы метода *CreateInstinct* пусть будет значение этой переменной, хотя мы могли бы заменить её и переменной *axiom*.

Выделим «**треугольный**» инстинкт в отдельный метод:

```
// Треугольник:
public procedure Triangle();
begin
   // устанавливаем Черепашку:
   x0 := CX - 100;
   y0 := CY + 100;
   a0 := 0;
   teta := 120;
   size := 100;
   speed := 5;
   iter := 2;
   axiom := 'F';
   newF := 'F+F';
   CreateInstinct();
```

```
    Start();
    Execute();
end;
```

Вы научились формировать простейший инстинкт, повинуясь которому *Черепашка* рисует треугольник (см. Рис. 13.5). Достаточно изменить значение угла *teta,* и новый инстинкт – для построения **квадрата** – готов:

```
// Квадрат:
public procedure Quadrat();
begin
   // устанавливаем Черепашку:
   x0 := CX - 100;
   y0 := CY + 100;
   a0 := 0;
   teta := 90;
   size := 100;
   iter := 2;
   speed := 10;
   axiom := 'F';
   newF := 'F+F';
   CreateInstinct();
   Start();
   Execute();
end;
```

На третий год жизни *Черепашка* сумеет вычертить **пятиугольник**, **шестиугольник** и **восьмиугольник** (Рис. 13.6).

```
// Пятиугольник:
public procedure Pentagon();
begin
    // устанавливаем Черепашку:
    x0 := CX - 100;
    y0 := CY + 100;
    a0 := 0;
    teta := 72;
    size := 100;
    iter := 3;
    speed := 10;
    axiom := 'F';
    newF := 'F+F';
    CreateInstinct();
    Start();
```

```
   Execute();
end;

// Шестиугольник:
public procedure Hexagon();
begin
   // устанавливаем Черепашку:
   x0 := CX - 100;
   y0 := CY + 100;
   a0 := 0;
   teta := 60;
   size := 100;
   iter := 3;
   speed := 10;
   axiom := 'F';
   newF := 'F+F';
   CreateInstinct();
   Start();
   Execute();
end;

// Восьмиугольник:
public procedure Octagon();
begin
   // устанавливаем Черепашку:
   x0 := CX - 100;
   y0 := CY + 100;
   a0 := 0;
   teta := 45;
   size := 85;
   iter := 3;
   speed := 10;
   axiom := 'F';
   newF := 'F+F';
   CreateInstinct();
   Start();
   Execute();
end;
```

Мы могли бы продолжать этот процесс и дальше, но совершенно понятно, что с годами *Черепашка* сможет выстраивать любые правильные многоугольники, поэтому давайте усложним поведение *Черепашки* с помощью более изощрённых инстинктов и научим её чертить *фракталы*.

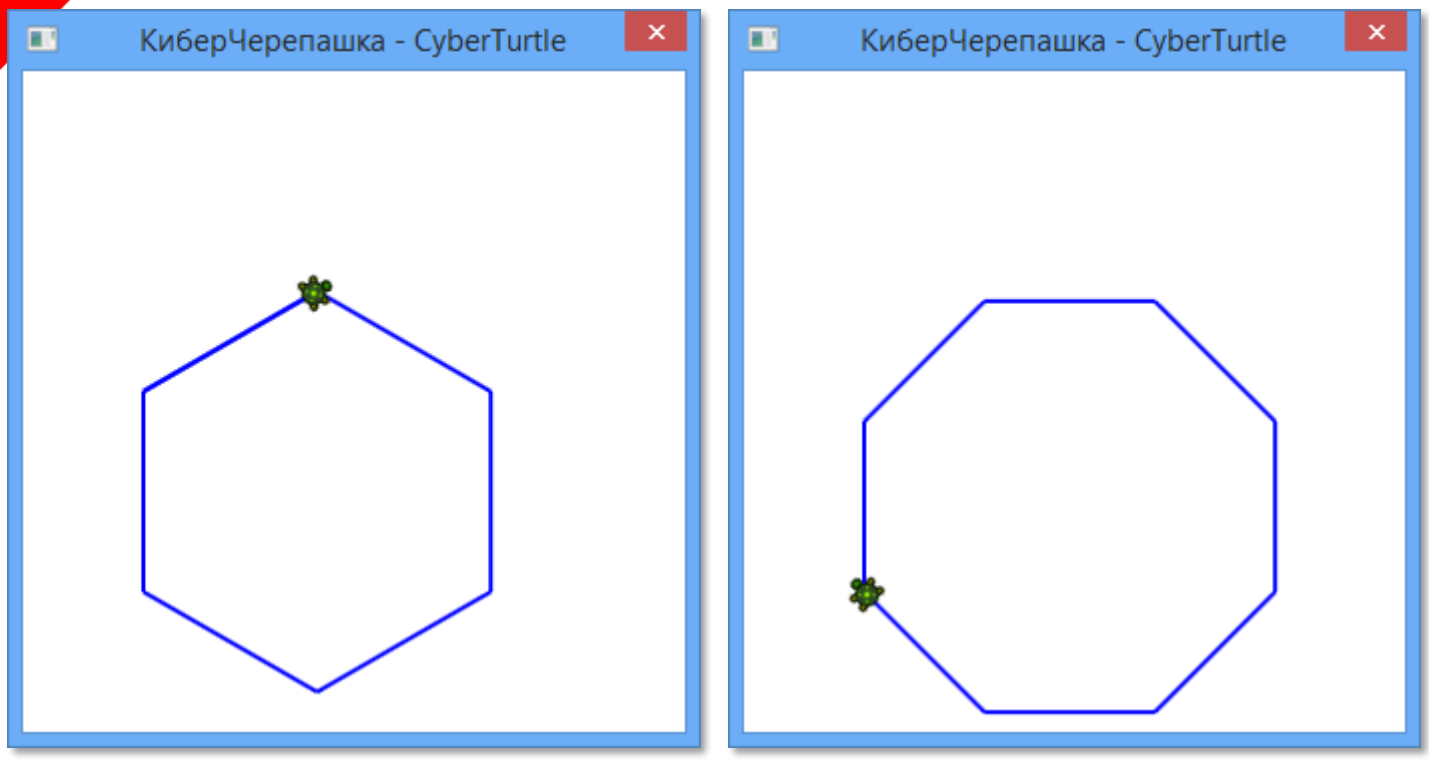


Рис. 13.6. Черепашковые многоугольники

Фракталы

Фракталы – это замечательные геометрические фигуры, составленные из точно таких же фигур, но меньшего размера. Например, из маленьких отрезков вы можете составить отрезок большей длины, из него – ещё более длинный, и так до бесконечности (Рис. 13.7).

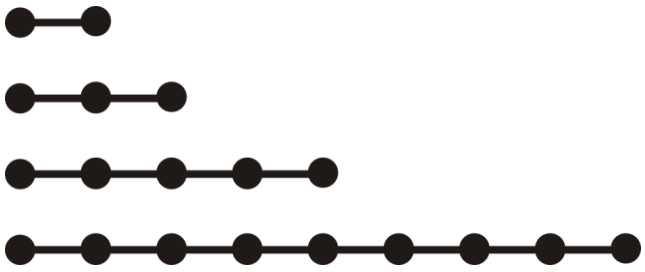


Рис. 13.7. Фрактальные отрезки

Аналогично из квадратиков вы можете построить большой квадрат (Рис. 13.8).

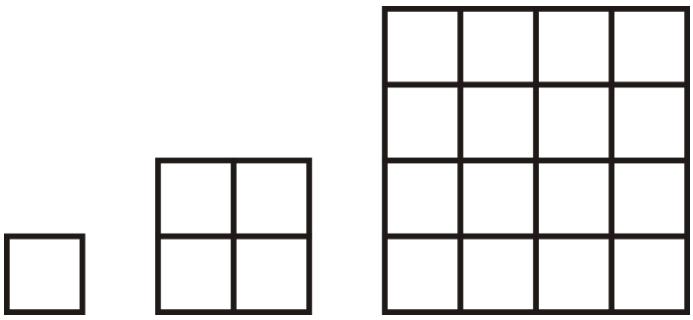


Рис. 13.8. Фрактальные квадраты

Примерами фракталов в природе могут служить: кровеносная система, береговая линия, горный рельеф, перистые облака (Рис. 13.9), разряд молнии, системы рек с притоками, трещины в почве, ветвистые растения, фьорды, прожилки на листьях (Рис. 13.10), ледяные узоры на стёклах…

Рис. 13.9. Перистые облака

Рис. 13.10. Жилкование листа

Снежинка Коха

Более сложный пример фрактальной кривой придумал в 1904 году *Гельг фон Кох*. Построение начинается с отрезка, который можно условно разделить на три равные части (Рис. 13.11а). Среднюю часть отрезка мы заменяем двумя отрезками, каждый из которых равен трети исходного отрезка (Рис. 13.11б). Затем мы повторяем эту операцию сколько угодно раз (Рис. 13.11в).

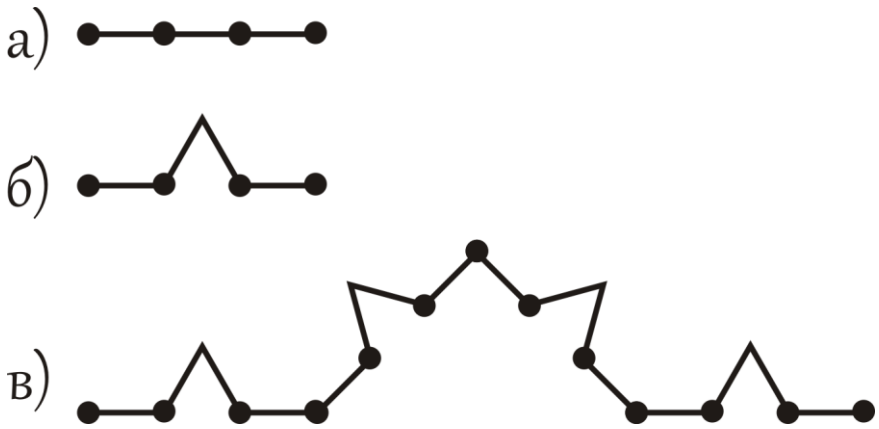
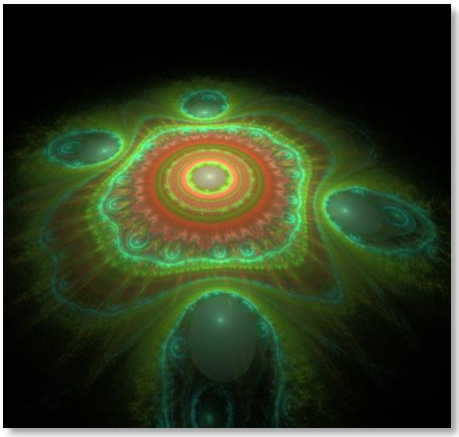


Рис. 13.11. Построение снежинки Коха

Проект *Фрактальная Черепашка*

Исходный код программы находится в папке **Фрактальная Черепашка**.



Если вы и дальше продолжите построение кривой Коха, то быстро убедитесь, что вручную её строить весьма утомительно. Поэтому лучше научить этому занятию *Черепашку*, только строить она будет не кривую, а **снежинку Коха**, которая отличается от кривой только тем, что её звенья не выстраиваются в прямую линию, а поворачиваются на 60 градусов, вследствие чего кривая становится замкнутой. Итак, с углом *teta* мы определились. С аксиомой тоже можно справиться, а вот правило, порождающее инстинкт, придётся позаимствовать у самого господина Коха:

```
// Снежинка Коха:
public procedure  Koch();
begin
   x0 := CX - 50;
   y0 := CY + 50;
   a0 := 0;

   teta := 60;
   size := 40;
   iter := 1;

   x0 := CX - 80;
   y0 := CY + 120;
   size := 3;
   iter := 4;

   speed := 10;
   axiom := 'F++F++F';
   newF := 'F-F++F-F';
   CreateInstinct();
   Start();
   Execute();
end;
```

Запускаем программу - на первом году жизни *Черепашка* нарисует шестиконечную звезду, в которой легко узнать кривую Коха, повторенную 3 раза (Рис. 13.13 слева).

Вообще говоря, настоящие снежинки также имеют 6 лучей, но более сложной формы (Рис. 13.12).

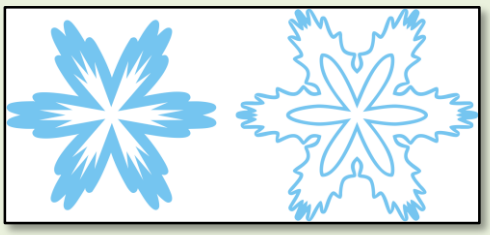


Рис. 13.12. Стилизованные снежинки

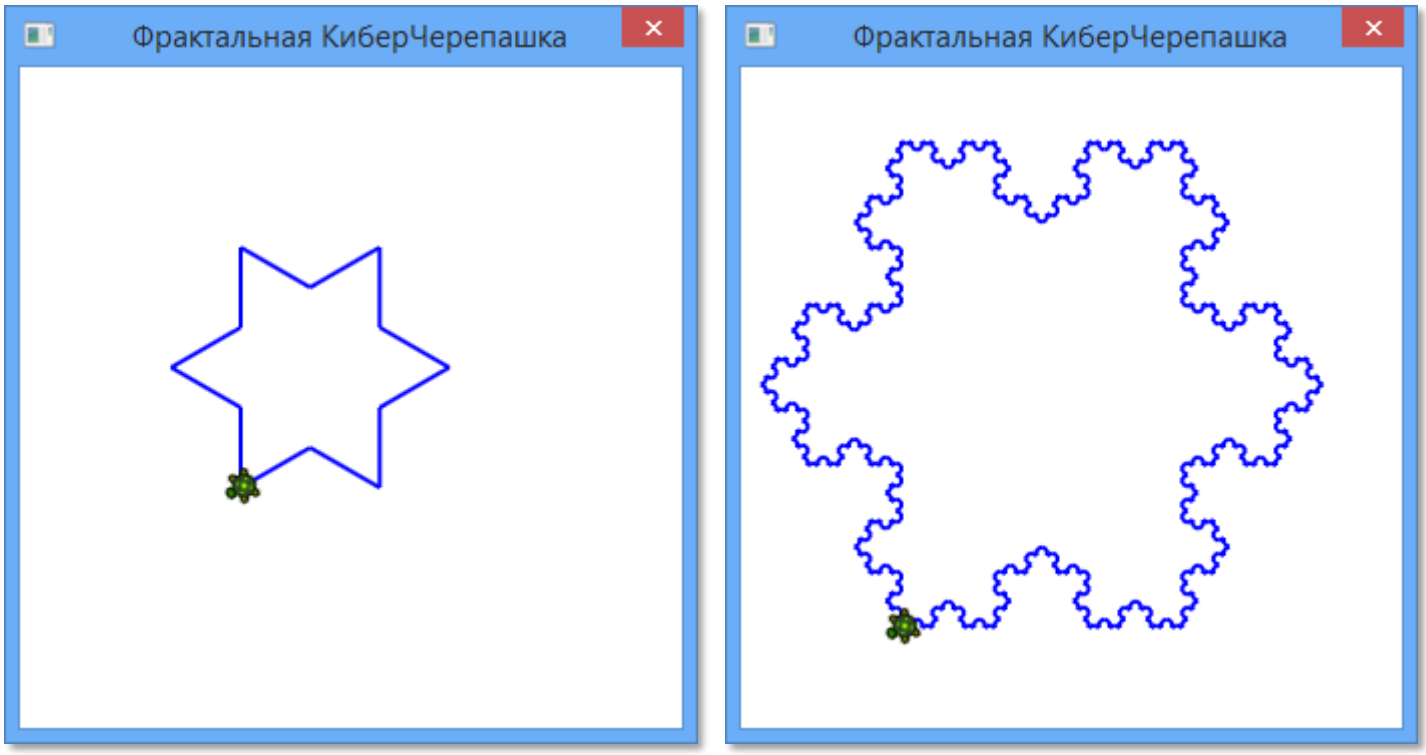


Рис. 13.13. Кривая Коха при *iter=1* и *iter=4*

С возрастом *Черепашка* рисует всё более и более сложные узоры, так что в четырёхлетнем возрасте снежинка у неё получается не хуже настоящей (Рис. 13.13, справа)!

В 1890 году *Джузеппе Пеано* построил функцию, график которой, названный **кривой Пеано**, покрывает квадратами (или, если угодно, ромбами) всю плоскость. И хотя кривая Пеано не является фракталом в полном смысле этого слова, но всё равно очень забавно наблюдать, как *Черепашка* её вычерчивает (Рис. 13.14). С годами (то есть с увеличением значения *iter*) она «квадрирует» все б***о***льшую и б***о***льшую поверхность.

```
// Кривая Пеано:
public procedure Peano();
begin
   x0 := CX - 100;
   y0 := CY + 100;
   a0 := 45;

   teta := 90;
   size := 10;
   iter := 3;
   speed := 10;
   axiom := 'F';
   newF := 'F-F+F+F+F-F-F-F+F';
   CreateInstinct();
   Start();
   Execute();
end;
```

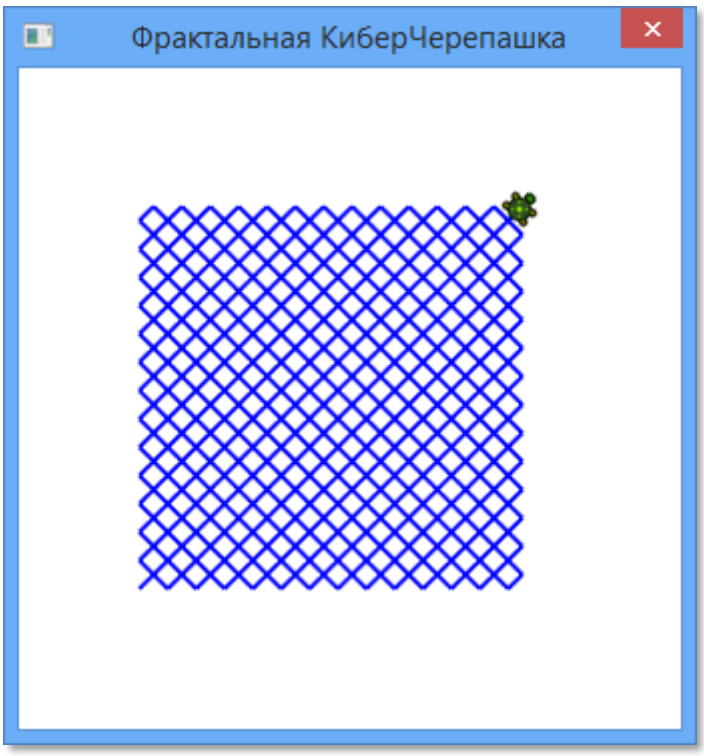


Рис. 13.14. Кривая Пеано при *iter= 3*

Ещё более усложнив инстинкт, мы научим *Черепашку* выделывать очень сложную кривую (Рис. 13.15).

```
      // 32-сегментная кривая:
      public procedure SC32();
```

```
begin
   x0 := CX - 40;
   y0 := CY + 40;
   a0 := 0;

   teta := 90;
   size := 2;
   iter := 2;
   speed := 10;
   axiom := 'F+F+F+F';
   newF :=
   '-F+F-F-F+F+FF-F+F+FF+F-F-FF+FF-FF+F+F-FF-F-F+FF-F-F+F+F-F+';
   CreateInstinct();
   Start();
   Execute();
end;
```

Добавим к инстинкту *Черепашки* новую команду, которая в генетическом коде обозначается буквой **b**. Получив такую команду, *Черепашка* переползает вперёд, но линию за собой не чертит.

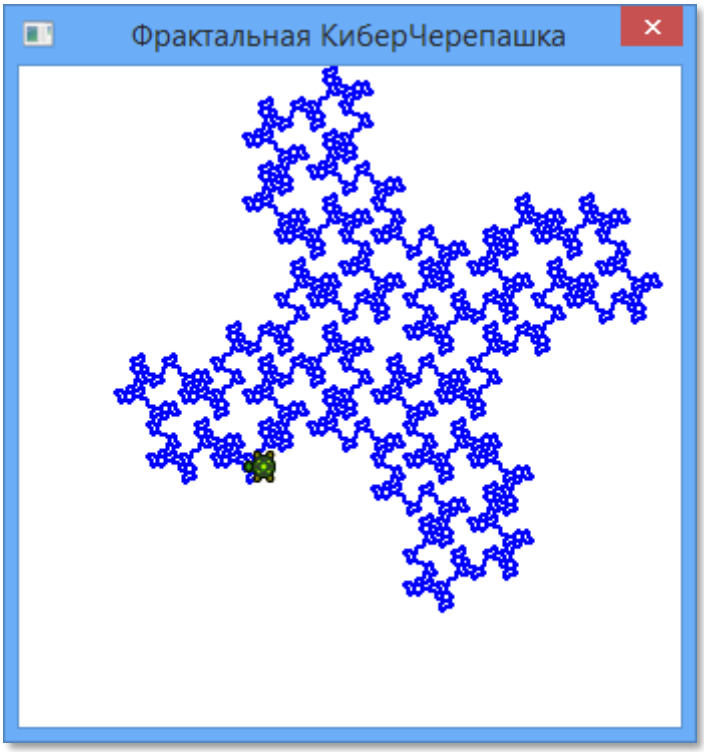


Рис. 13.15. 32-сегментная кривая при *iter= 2*

Для обработки нового гена мы прибавим в метод *CreateInstinct* пару строк:

```
. . .
```

```
    else if (cmd = 'b') then
        instinct += newb
    . . .
```

То есть новый ген *b* входит в состав инстинкта аналогично гену *F*. А вот в методе *Execute Черепашка* должна сначала поднять карандаш, а после перемещения снова опустить его:

```
else if (cmd = 'b') then
begin
   Turtle.PenUp();
   Turtle.Move(size);
   Turtle.PenDown();
end
```

С помощью этого гена *Черепашка* может рисовать фигуры, состоящие из нескольких не связанных между собой частей (Рис. 13.16).

```
// Мозаика:
public procedure Mozaic();
begin
   x0 := CX + 60;
   y0 := CY + 60;
   a0 := 0;

   teta := 90;
   size := 4;
   iter := 2;

   speed := 10;
   axiom := 'F-F-F-F';
   newF := 'F-b+FF-F-FF-Fb-FF+b-FF+F+FF+Fb+FFF';
   newb := 'bbbbbb';
   CreateInstinct();
   Start();
   Execute();
end;
```

Обратите внимание: каждое вхождение нового гена *b* в инстинкт заменяется значением переменной *newb*!

Вы можете добавлять к инстинкту *Черепашки* и другие гены, которые обозначаются буквами латинского алфавита. Они участвуют в формировании инстинкта, но при его выполнении игнорируются.

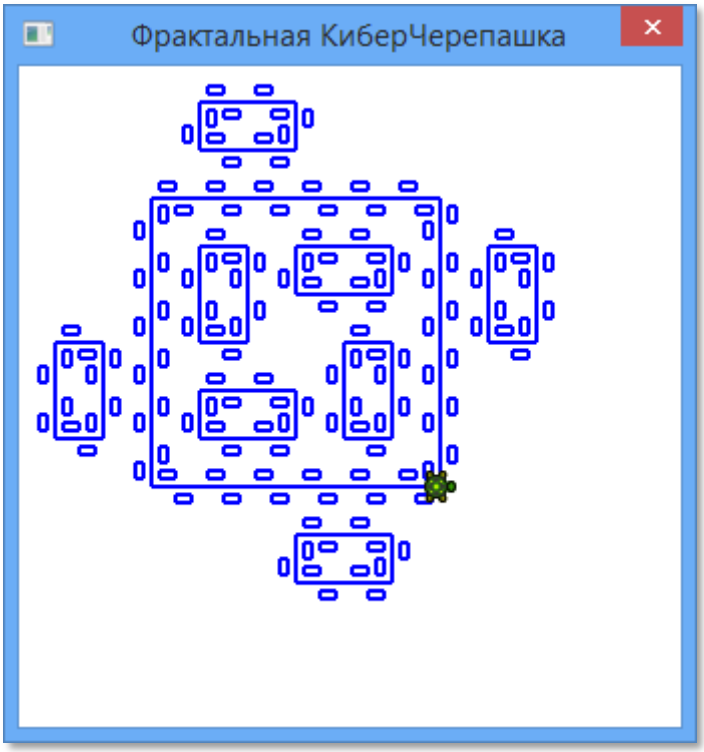


Рис. 13.16. Мозаика при *iter= 2*

Для каждого нового гена должно быть записано правило, по которому он заменяется в инстинкте другими генами. Нашей Ч*ерепашке* будет достаточно четырёх неисполняемых генов, которые формируют инстинкт в методе *CreateInstinct*:

```
else if (cmd = 'W') then
   instinct += newW
else if (cmd = 'X') then
   instinct += newX
else if (cmd = 'Y') then
   instinct += newY
else if (cmd = 'Z') then
   instinct += newZ
```

В метод *Execute* никаких изменений вносить не следует, поскольку эти команды *Черепашка* не выполняет. Зато новые гены могут настолько усложнить инстинкт *Черепашки*, что она начнёт вычерчивать кривые удивительной красоты!

Помните, как в главе *Геометрические фантазии* мы построили *Треугольный треугольник* (см. Рис. 8.9)? – А теперь мы обучим этому искусству нашу *КиберЧерепашку*:

```
// Треугольники:
public procedure Triangles();
begin
   x0 := CX - 10;
   y0 := CY + 20;
   a0 := 30;

   teta := 120;
   size := 16;
   iter := 6;
   speed := 10;
   axiom := 'bX';
   newb := 'b';
   newF := 'F';
   newX := '--FXF++FXF++FXF--';
   CreateInstinct();
   Start();
   Execute();
end;
```

Хотя задача оказалась непростой, но *КиберЧерепашка* справилась с заданием отлично (Рис. 13.17).

С двумя новыми генами *Черепашка* построит **кривую Пеано-Госпера**, которая, как и кривая Коха, также напоминает снежинку, но заполненную внутри замысловатыми линиями (Рис. 13.18).

```
// Кривая Пеано-Госпера:
public procedure PeanoGosper();
begin
   x0 := CX - 70;
   y0 := CY + 80;
   a0 := 0;

   teta := 60;
   size := 10;
   iter := 4;
   speed := 10;
   axiom := 'FX';
   newF := 'F';
   newX := 'X+YF++YF-FX--FXFX-YF+';
```

```
   newY := '-FX+YFYF++YF+FX--FX-Y';
   CreateInstinct();
   Start();
   Execute();
end;
```

Рис. 13.17. Треугольники при *iter= 6*

А вот чтобы научить *Черепашку* рисовать почти настоящую снежинку, нам придётся наделить ее **памятью**. По команде **[** она будет запоминать свои координаты и угол поворота, а по команде **]** возвращаться в это состояние.

Поскольку эти гены не влияют на формирование инстинкта, то метод *CreateInstinct* останется без изменений, а в метод *Execute* нужно добавить строчки:

```
else if (cmd = '[') then
begin
   Stack.PushValue(_stack, Turtle.Angle);
   Stack.PushValue(_stack, Turtle.X);
   Stack.PushValue(_stack, Turtle.Y);
end
else if (cmd = ']') then
begin
   Turtle.Y := Stack.PopValue(_stack);
```

```
  Turtle.X := Stack.PopValue(_stack);
  Turtle.Angle := Stack.PopValue(_stack);
end
```

Как видите, память *Черепашки* устроена по принципу **стека**: *Черепашка* вспоминает *последние* запомненные события, после чего они стираются из её памяти.

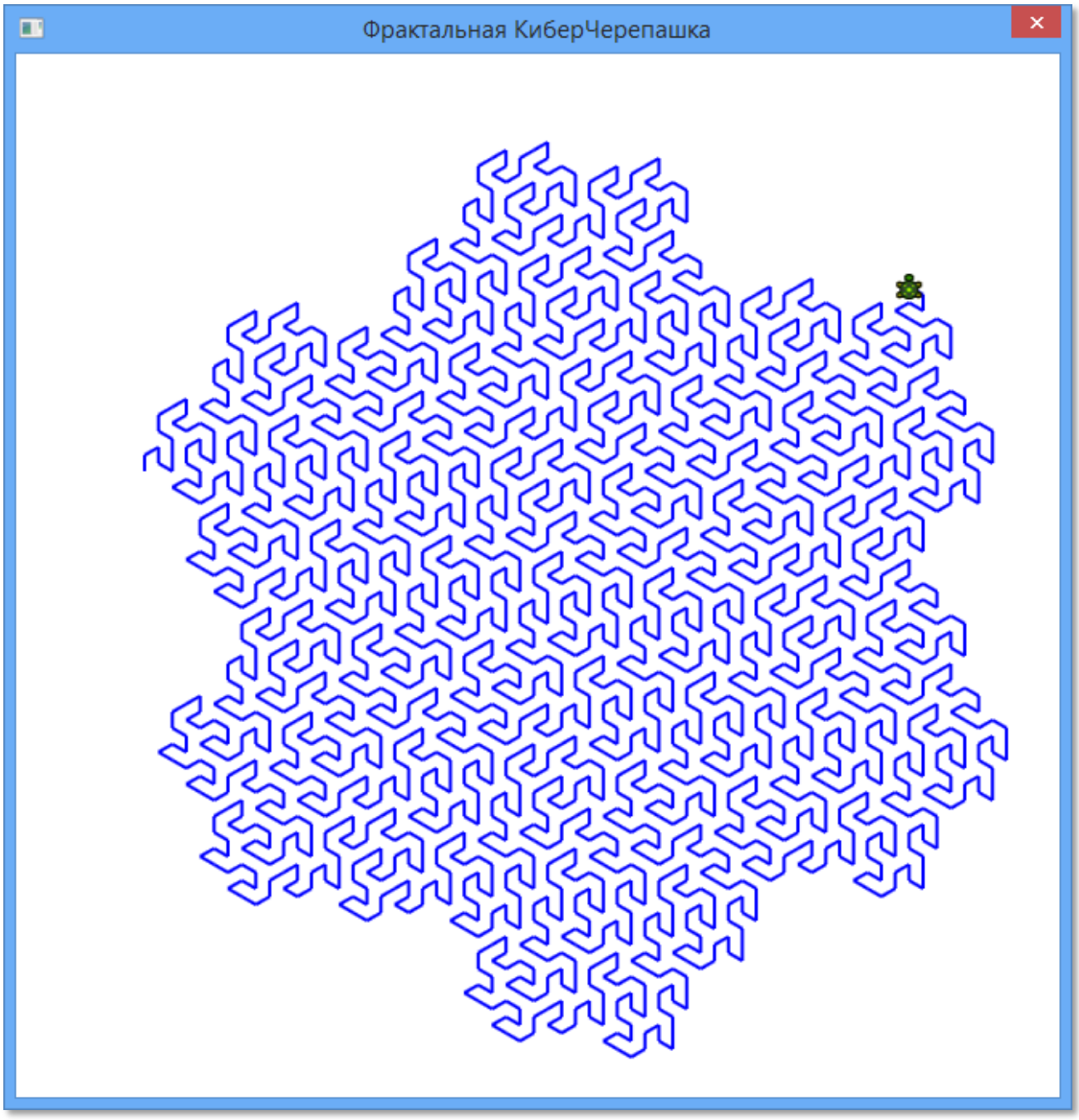


Рис. 13.18. Кривая Пеано-Госпера при *iter= 4*

Теперь *Черепашка* не только умеет выполнять команды инстинкта, но и обладает памятью. А вот так она строит почти настоящую снежинку (Рис. 13.19):

```
// Снежинка:
public procedure Snowflake();
begin
   x0 := CX;
   y0 := CY - 20;
   a0 := 0;

   size := 4;
   iter := 2;
   teta := 60;

   speed := 10;
   axiom := '[F]+[F]+[F]+[F]+[F]+[F]';
   newF := 'F[++F][-FF]FF[+F][-F]FF';
   CreateInstinct();
   Start();
   Execute();
end;
```

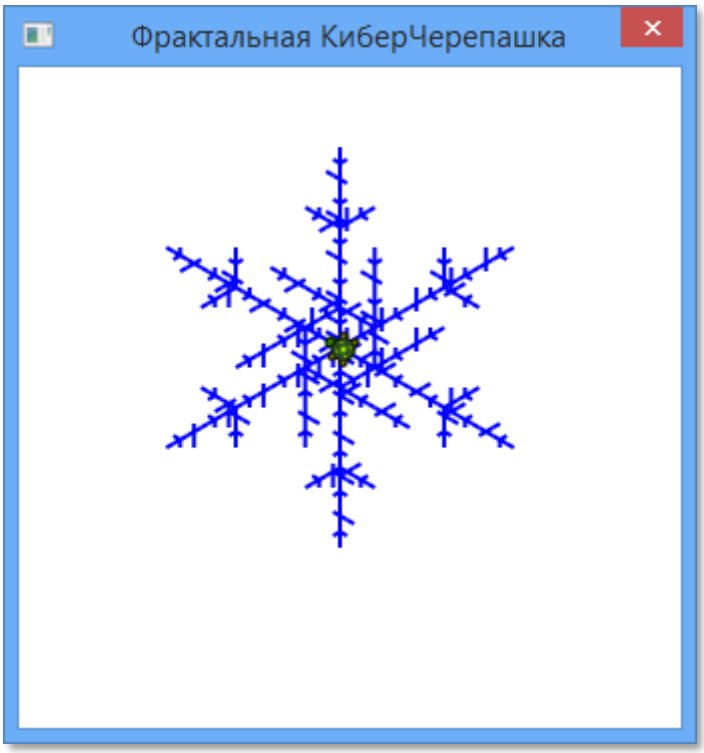


Рис. 13.19. Снежинка при *iter= 2*

С помощью очень простого инстинкта и памяти *Черепашка* нарисует очень красивую картинку, состоящую из **ромбов** (Рис. 13.20):

```
// Ромбы:
public procedure Rhombes();
begin
   x0 := CX;
   y0 := CY;
   a0 := 0;

   teta := 60;
   size := 10;
   iter := 6;

   speed := 10;
   axiom := 'F';
   newF := '-F+F+[+F+F]-';
   CreateInstinct();
   Start();
   Execute();
end;
```

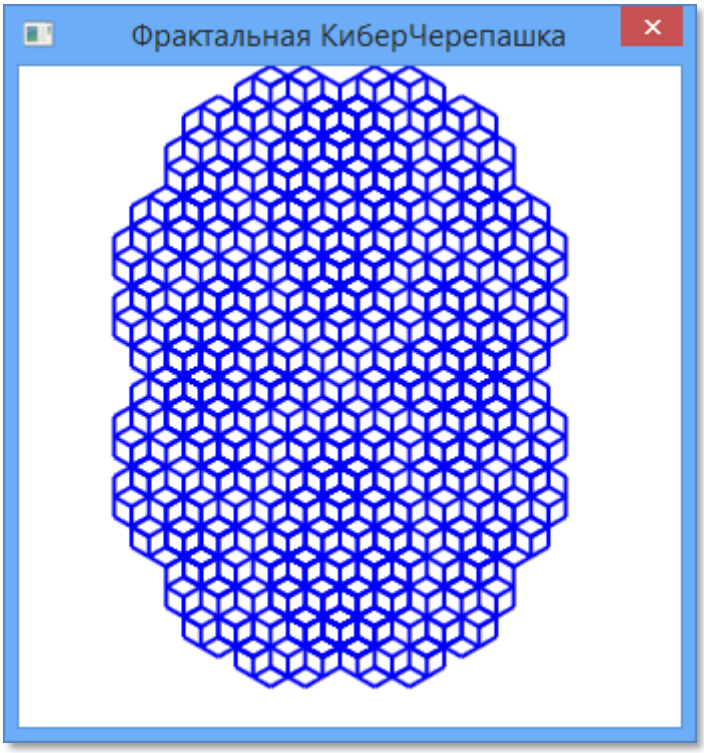


Рис. 13.20. Ромбы при *iter= 6*

Кривая Пенроуза (Рис. 13.21) также состоит из ромбов, но для её построения *Черепашке* понадобится и очень сложный инстинкт, и крепкая память:

```
// Кривая Пенроуза:
public procedure Penrose();
begin
   x0 := CX + 80;
   y0 := CY + 80;
   a0 := 0;
   teta := 36;
   size := 24;
   iter := 4;
   speed := 10;
   axiom := '[Y]++[Y]++[Y]++[Y]++[Y]';
   newW := 'YF++ZF----XF[-YF----WF]++';
   newX := '+YF--ZF[---WF--XF]+';
   newY := '-WF++XF[+++YF++ZF]-';
   newZ := '--YF++++WF[+ZF++++XF]--XF';
   newF := '';
   CreateInstinct();
   Start();
   Execute();
end;
```

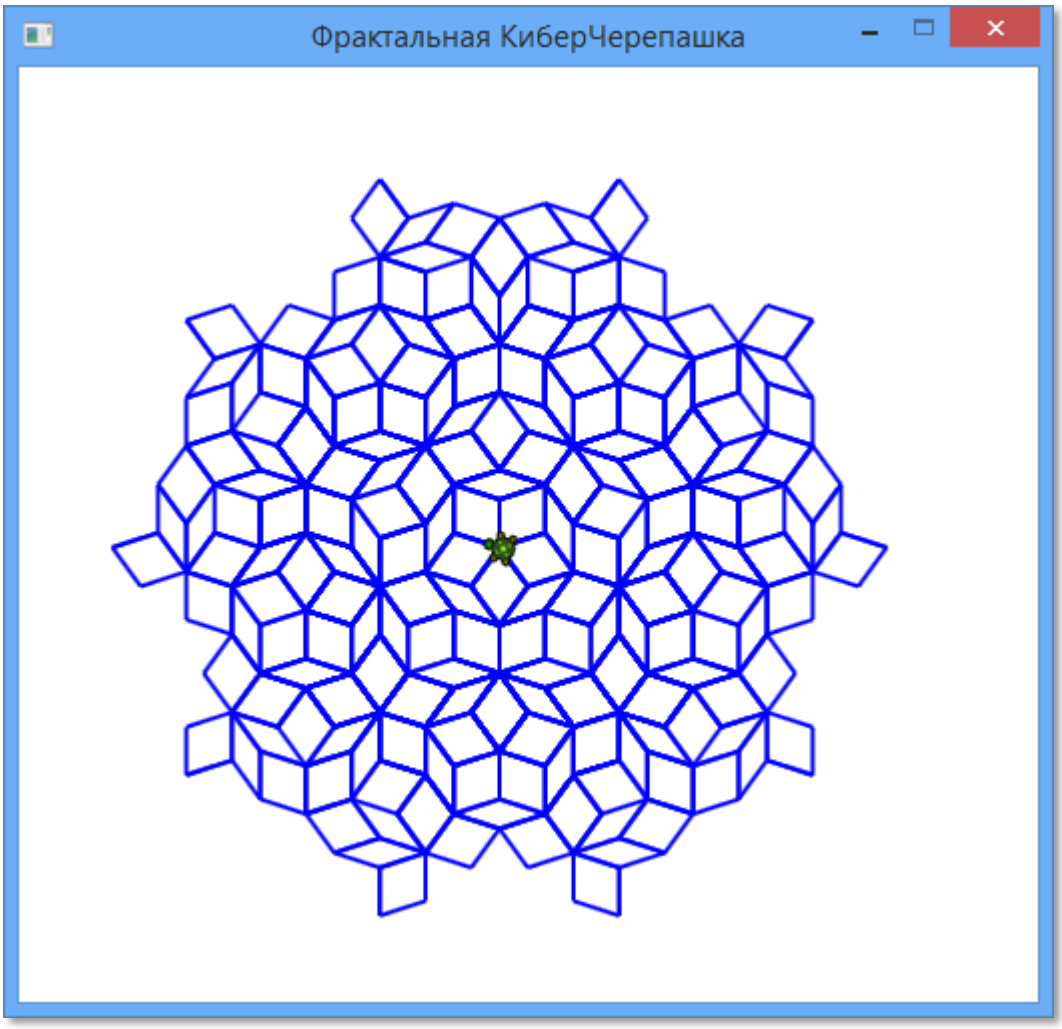


Рис. 13.21. Кривая Пенроуза при *iter= 4*

О *мозаиках Пенроуза* вы можете прочитать в книге Мартина Гарднера **[ГМ93]**.

И последнее, чему мы научим нашу *Черепашку*, - это строить кривую, которая очень напоминает настоящее **растение** (Рис. 13.22):

```
// Куст:
public procedure Bush();
begin
   x0 := CX;
   y0 := CY + 100;
   a0 := 0;
   size := 8;
   iter := 4;
   teta := 180 div 8;
   speed := 10;
   axiom := 'F';
   newF := '-F+F+[+F-F-]-[-F+F+F]';
   CreateInstinct();
   Start();
   Execute();
end;
```

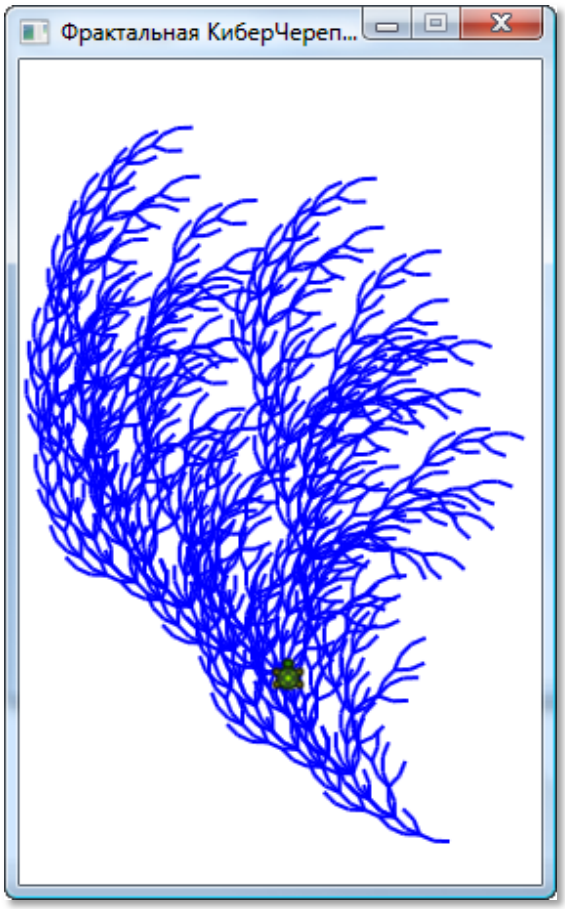


Рис. 13.22. Куст при *iter= 4*

Проект *L-системы*

Исходный код программы находится в папке **L-системы**.

И вот, глядя на этот очаровательный «куст», мы можем, наконец, раскрыть тайну поведения нашей *Черепашки*. Оказывается, инстинкт *Черепашки* описывается с помощью **L-системы**, которая была предложена шведским биологом *Аристидом Линденмайером* (поэтому иначе они называются *системами Линденмайера*) для описания растений. Пример с кустом и следующий – с деревом (Рис. 13.23) - убедительно доказывают, что с помощью этой системы можно конструировать весьма правдоподобные растения (в некоторых графических редакторах они генерируются практически так же).

```
// Дерево:
public procedure Tree();
begin
   x0 := CX;
   y0 := CY + 120;
   a0 := 0;

   size := 8;
   iter := 4;
   teta := 180 div 6;
   speed := 10;
   axiom := 'F';
   newF := 'F[-F]F[+F][F]';
   CreateInstinct();
   Start();
   Execute();
end;
```

Все древовидные структуры, а также многие рассмотренные нами фракталы описываются аксиомой и несколькими правилами, которые с увеличением числа итераций *iter* могут создать кривые любой степени детализации (в том числе и бесконечной, если у вас есть бесконечно много времени и других ресурсов). Неожиданно оказалось, что построение кривых по системе Линденмайера очень удобно реализовать с помощью черепашьей графики, поскольку *Черепашка* умеет выполнять все команды, используемые в *L-системах*.

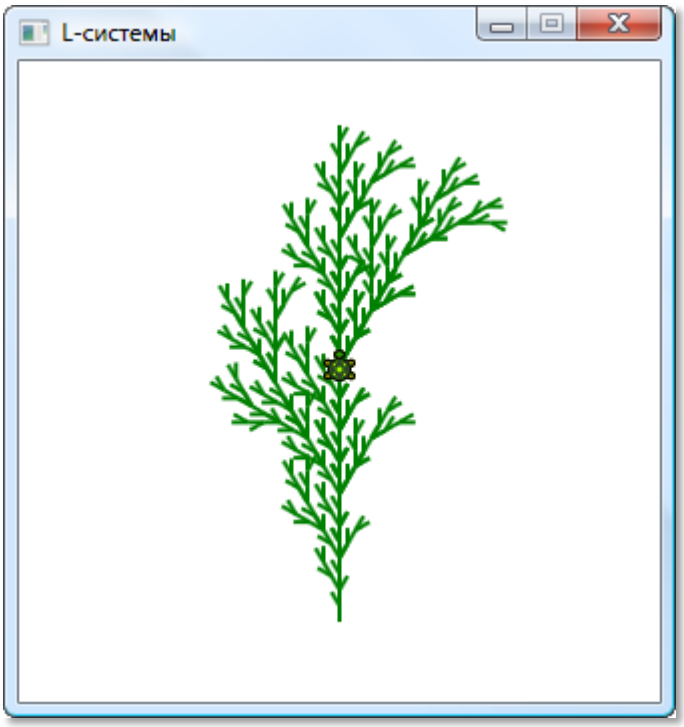


Рис. 13.23. Дерево

И ещё парочка деревьев (Рис. 13.24):

```
// Дерево2:
public procedure Tree2();
begin
   x0 := CX;
   y0 := CY + 180;
   a0 := 0;

   size := 4;
   iter := 4;
   teta := 180 div 6;
   speed := 10;
   axiom := 'F';
   newF := 'F[+F]F[-F]F';
   CreateInstinct();
   Start();
   Execute();
end;

// Дерево3:
public procedure Tree3();
begin
   x0 := CX + 40;
   y0 := CY + 160;
   a0 := 0;
```

```
    size := 6;
    iter := 4;
    teta := 180 div 6;
    speed := 10;
    axiom := 'F';
    newF := 'FF-[-F+F+F]+[+F-F-F]';
    CreateInstinct();
    Start();
    Execute();
end;
```

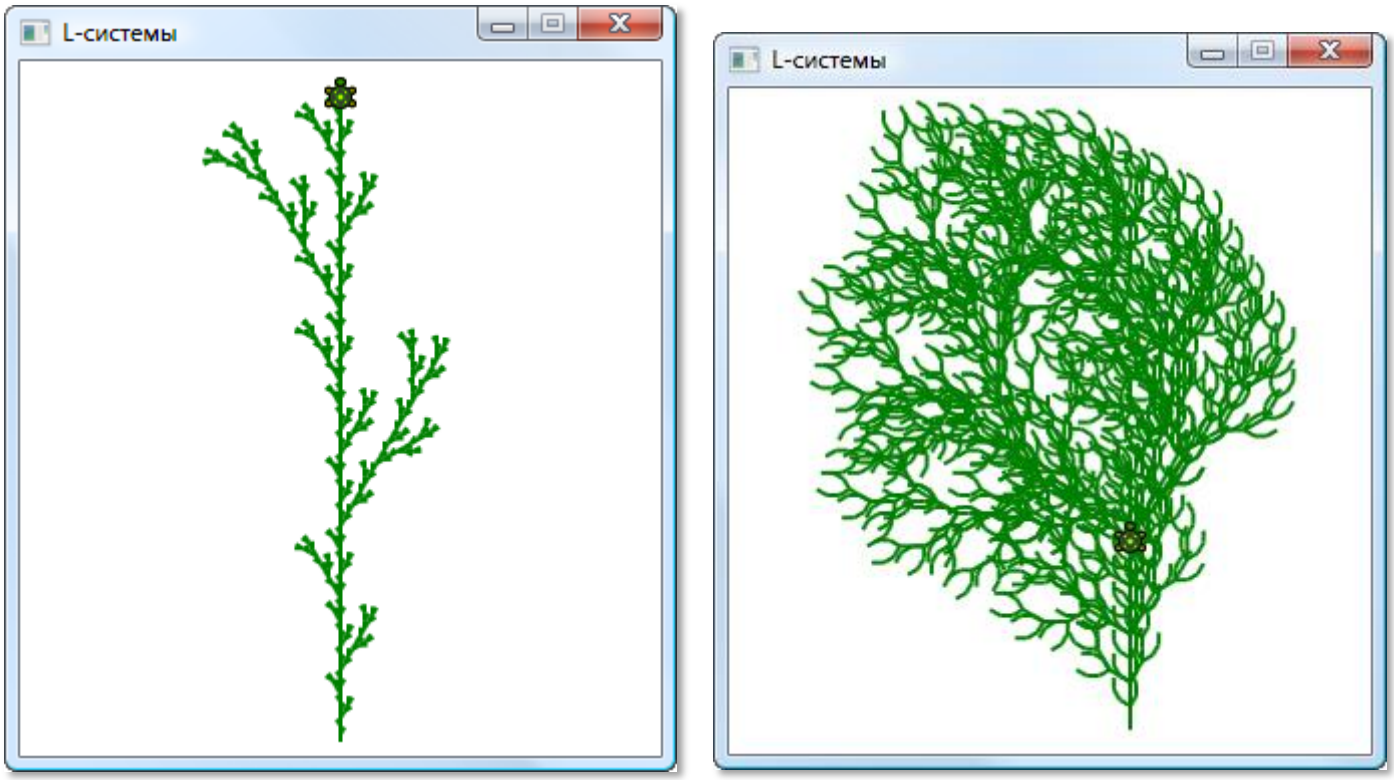


Рис. 13.24. Деревья

Проект *Очень БОЛЬШАЯ Черепашка*

Исходный код программы находится в папке **Очень БОЛЬШАЯ Черепашка**.

Седлай черепаху!

Со времён языка *Лого Черепашка* используется как наглядное средство при обучении детей программированию. Эта славная традиция не увяла и до настоящего времени. Например, в книге *Python für Kids* (Рис. 13.25), которая выдержала в Германии уже 4 издания, всё обучение программированию строится вокруг этого милого животного (это я о *Черепашке*, а не о *Питоне* – хотя упомянутая недобрым словом змея не имеет прямого отношения к одноимённому языку).

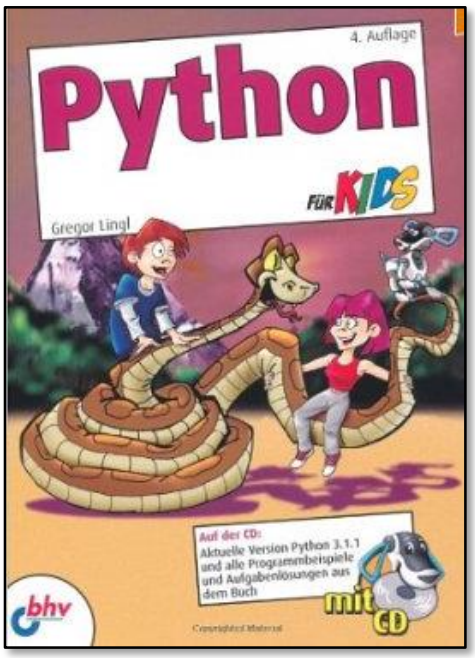


Рис. 13.25. Обложка книги

Справедливости ради следует отметить, что немецкая *Черепашка* (*Schildkröte*) и бойчее, и мощнее Смоллбейсиковской. Например, наша *Черепашка* только и умеет что вычерчивать линии, а немецкая - ещё и заливные фигуры (Рис. 13.26).

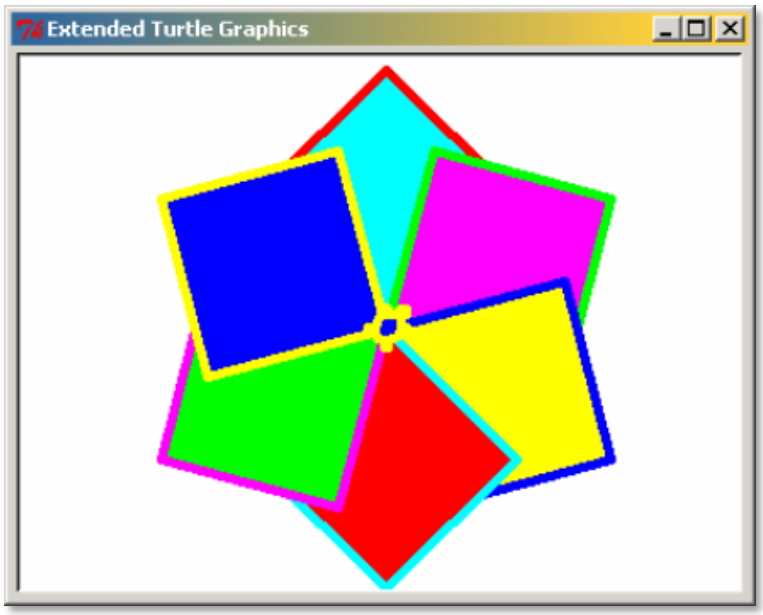


Рис. 13.26. Окрашенные квадраты

Наша *Черепашка* всегда пребывает в одиночестве, а немецких черепах можно плодить целыми стадами (Рис. 13.27) и беспощадно топтать ими одновременно всю канву окна приложения.

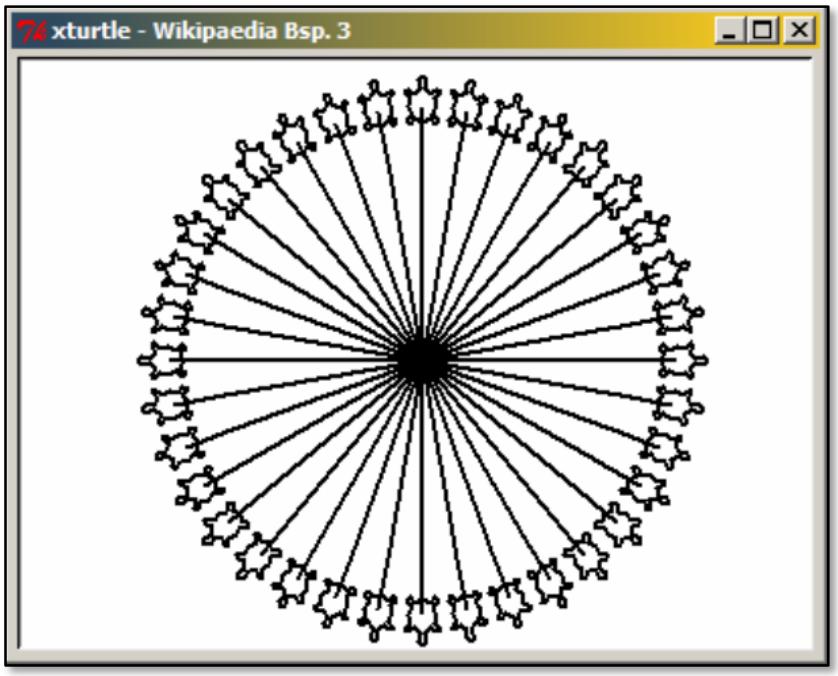


Рис. 13.27. 36 *Черепашек*, разбегающихся во все стороны

Благодаря невероятному совпадению или везению *Черепашка* в качестве средства обучения попала не в суп, а ещё и в американскую книжку – и тоже по программированию на *Питоне*, и тоже для детей (Рис. 13.28).

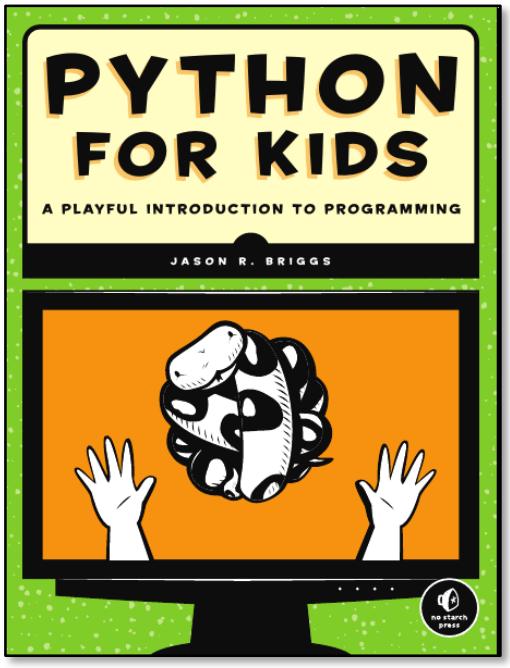


Рис. 13.28. Другая книга для детей

Американская *Черепашка* так же одинока, как и наша, и даже на настоящую черепаху не похожа, но зато умеет рисовать закрашенные фигуры (Рис. 13.29).

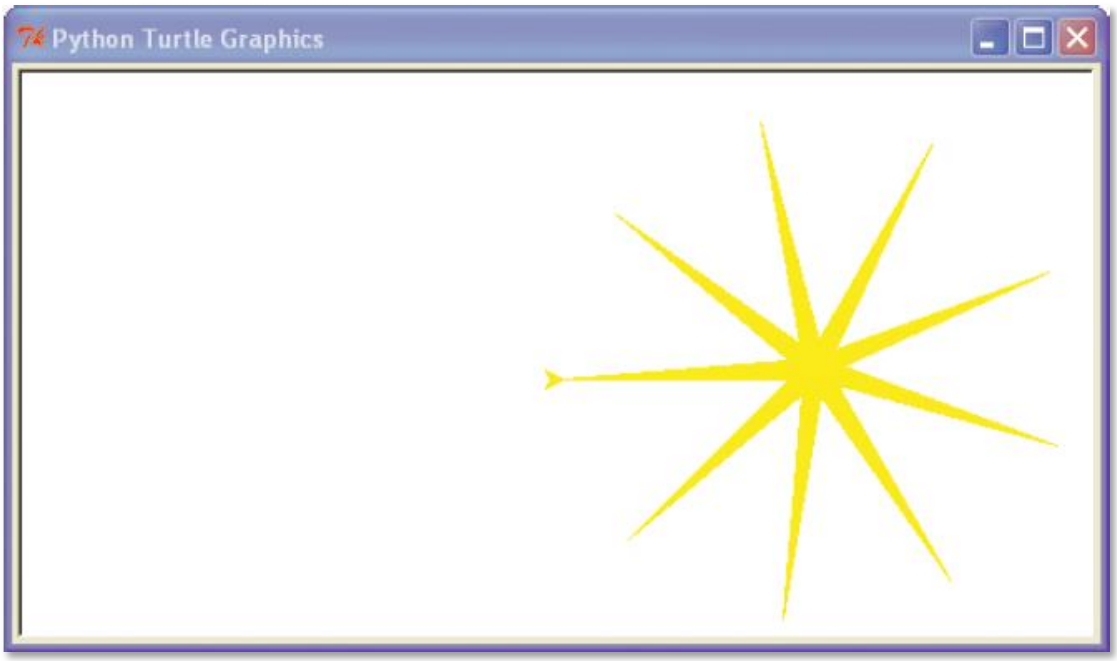


Рис. 13.29. Закрашенная звёздочка

Не обошёлся без *Черепашки* и видеокурс *PluralSight - Teaching Kids Programming with C#* (Рис. 13.30), посвящённый обучению детей, начиная с 10-летнего возраста, программированию на языке *Си-шарп*.

> Этот видеокурс вы можете загрузить с сайта *TeachingKidsProgramming.org*.

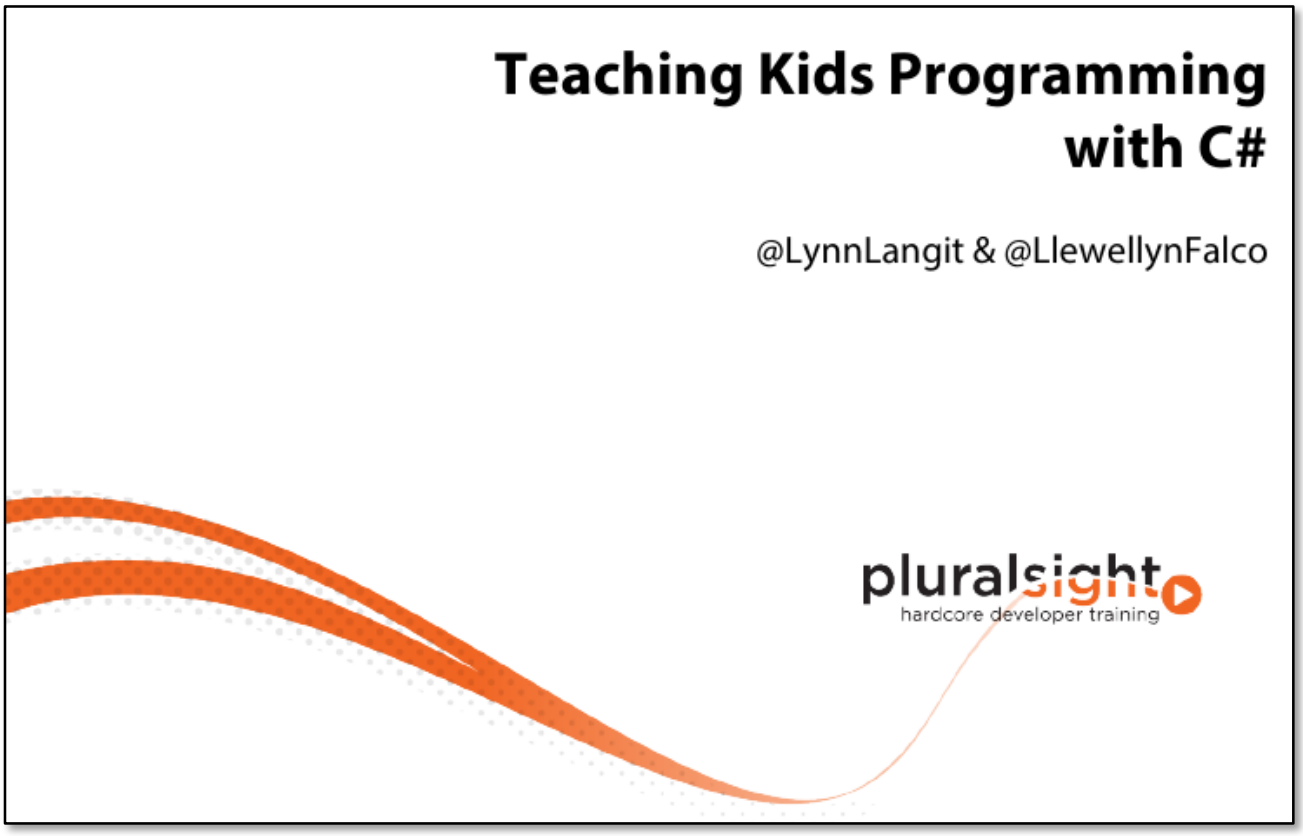


Рис. 13.30. Всё лучшее – детям!

Надо отметить, что видеокурс очень короткий, поэтому пользы от него будет немного, хотя смотреть, как программируют другие, - большое удовольствие (Рис. 13.31)!

Впрочем, нам никто и ничто не мешает использовать *Черепашку*-курсистку в своих проектах.

Для этого нужно добавить ссылку на динамическую библиотеку **SmallBasicFun.dll**.

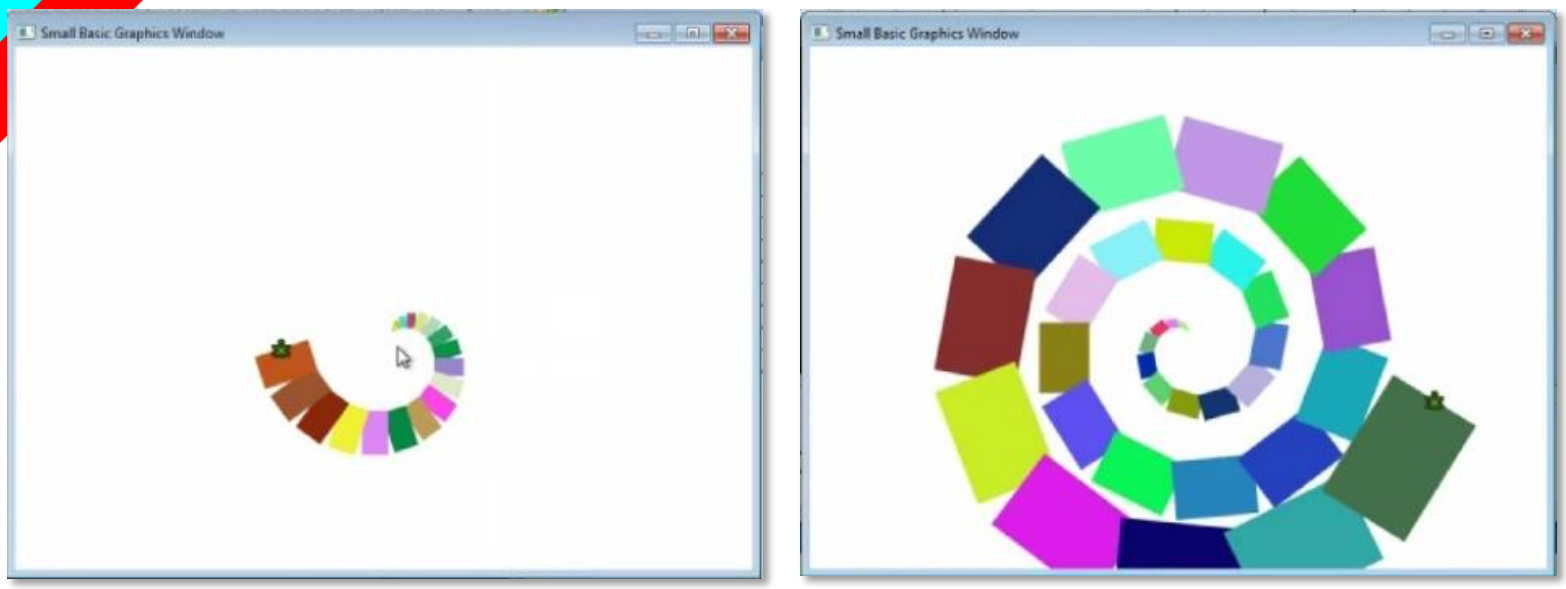


Рис. 13.31. Ломаные спирали

Вообще говоря, в библиотеке *SmallBasicFun* не одна, а две *Черепашки* – БОЛЬШАЯ и маленькая. Большая *Черепаха*, хотя и большая, но довольно глупая: она умеет рисовать только квадраты и правильные многоугольники (Рис. 13.32). Для этого она имеет 2 метода:

GiantTortoise. DrawSquare(**color** : string);

GiantTortoise. DrawShape(**sides** : integer; **color** : string);

```
{$apptype windows}


uses
   DrawUnit;

begin
   var draw := new Draw();
   draw.Prepare();
   //draw.Quadrat();
   draw.Polygon();
end.

unit DrawUnit;

uses
   Microsoft.SmallBasic.Library, SmallBasicFun, System;
```

```
type
   Draw = class
   private
      const GWWIDTH = 480;
      const GWHEIGHT = 480;
      // размеры окна:
      width := GWWIDTH;
      height := GWHEIGHT;
      // координаты центра окна:
      CX := width div 2;
      CY := height div 2;


      public procedure Prepare();
      begin
         GraphicsWindow.Hide();
         GraphicsWindow.Title := 'Класс GiantTortoise';
         GraphicsWindow.Width := GWWIDTH;
         GraphicsWindow.Height := GWHEIGHT;
         GraphicsWindow.Show();
         GraphicsWindow.Left := (Desktop.Width –
                                 GraphicsWindow.Width) / 2;
         GraphicsWindow.Top := (Desktop.Height –
                                 GraphicsWindow.Height) / 2;
         // GraphicsWindow.CanResize := false;
         GraphicsWindow.BackgroundColor := 'White';
      end;


      // Квадрат:
      public procedure Quadrat();
      begin
         // рисуем квадрат:
         GiantTortoise.DrawSquare(Colors.Red);
         GiantTortoise.DrawSquare('random');
      end;

      // Многоугольник:
      public procedure Polygon();
      begin
         // рисуем многоугольник:
         GiantTortoise.DrawShape(9, 'random');
      end;


   end; //end of class
end.
```

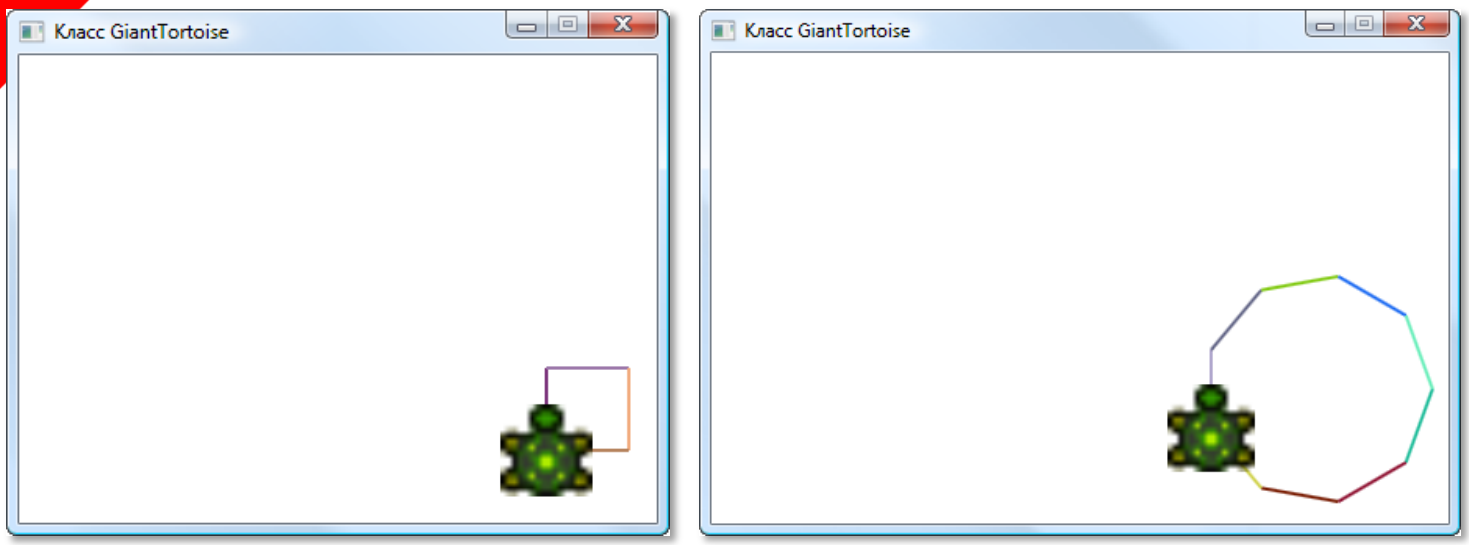


Рис. 13.32. Многоугольники большой *Черепашки*

Метод **DrawSquare** помогает вычертить *Черепахе* квадрат со сторонами заданного цвета. Обратите внимание, что можно задать случайный цвет "*random*", и тогда все стороны квадрата будут вычерчены линиями случайного цвета. Ни размеры квадрата, ни его положение на канве изменить нельзя.

Этот метод использует для рисования квадрата другой метод - **DrawShape**, которому он передаёт значение параметра *sides* = 4.

Вы можете задать другое значение этому параметру, и *Черепаха* послушно вычертит правильный многоугольник.

Проект *Библиотечная Черепашка*

Исходный код программы находится в папке **Библиотечная Черепашка**.

Вторая *Черепашка* – маленькая, да удаленькая! Она может выполнять всё, что умеет делать и стандартная *Черепашка*, и кое-что у неё есть в запасе!

Чтобы *Черепашка* появилась на экране, нужно выполнить метод **Show**:

```
Tortoise.Show();
```

Метод **Hide**, наоборот, скрывает *Черепашку*, но при этом она продолжает выполнять все команды:

```
Tortoise.Hide();
```

Метод **Reset** воссоздаёт *Черепашку* с параметрами по умолчанию:

```
Tortoise.Reset();
```

Два метода – **SetX** и **SetY** – переносят *Черепашку* в указанную точку клиентской области окна без вычерчивания линии:

```
Tortoise.SetX(x : double);
```

```
Tortoise.SetY(y : double);
```

Метод **SetPosition** одновременно изменяет обе координаты *Черепашки*:

```
Tortoise.SetPosition(x, y : integer);
```

Два других метода – **GetX** и **GetY** – сообщают нам текущие координаты *Черепашки*:

```
Tortoise.GetX() : double;
```

```
Tortoise.GetY() : double;
```

Теперь мы можем создать *Черепашку* и поставить её перед фактом в самом центре окна (Рис. 13.33):

```
{$apptype windows}

// Tortoise

uses
   DrawUnit;
```

```
begin
   var draw := new Draw();
   draw.Prepare();
   // треугольная спираль:
   draw.Spiral(400, 120, 8);
end.


unit DrawUnit;

uses
   Microsoft.SmallBasic.Library, SmallBasicFun, System;

type
   Draw = class
   private
      const GWWIDTH = 480;
      const GWHEIGHT = 480;
      // размеры окна:
      width := GWWIDTH;
      height := GWHEIGHT;
      // координаты центра окна:
      CX := width div 2;
      CY := height div 2;


      public procedure Prepare();
      begin
         GraphicsWindow.Hide();
         GraphicsWindow.Title := 'Класс Tortoise';
         GraphicsWindow.Width := GWWIDTH;
         GraphicsWindow.Height := GWHEIGHT;
         GraphicsWindow.Show();
         GraphicsWindow.Left := (Desktop.Width –
                                  GraphicsWindow.Width) / 2;
         GraphicsWindow.Top := (Desktop.Height –
                                 GraphicsWindow.Height) / 2;
         // GraphicsWindow.CanResize := false;
         GraphicsWindow.BackgroundColor := 'White';

         // создаём Черепашку:
         Tortoise.Show();
          // устанавливаем Черепашку в центре окна:
         Tortoise.SetX(CX);
         Tortoise.SetY(CY);
      end;

   end; //end of class
end.
```

Зная, что все *Черепашки* любят выделывать спирали, мы не можем отказать и нашей *Черепашке* в этом удовольствии!

Метод **Move** заставляет двигаться её вперёд на *lengthInPixels* пикселей:

```
Tortoise.Move(lengthInPixels : double);
```

Метод **MoveTo** перемещает *Черепашку* в указанную точку с координатами *(x,y)*:

```
Tortoise.MoveTo(x, y : double);
```

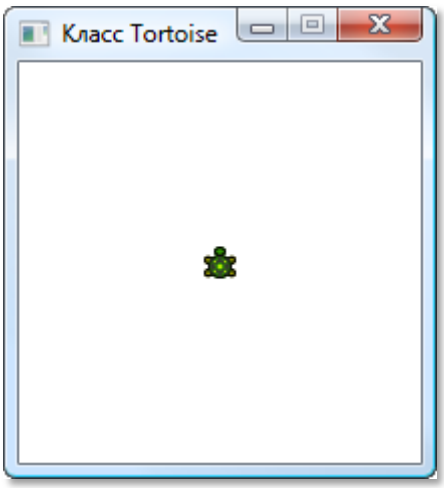


Рис. 13.33. Маленькая *Черепашка* в центре внимания

Метод **Turn** поворачивает *Черепашку* на *degrees* градусов: по часовой стрелке, если угол положительный, или против часовой стрелки, если угол отрицательный:

```
Tortoise.Turn(degrees : double);
```

Если предыдущий метод отсчитывает угол от текущего поворота *Черепашки*, то метод **SetAngle** поворачивает *Черепашку* точно в заданном направлении. Если *angle = 0*, то *Черепашка* смотрит вверх, если *angle = 180*, то *Черепашка* смотрит вниз.

```
Tortoise.SetAngle (angle : double);
```

Метод **GetAngle** возвращает текущий угол поворота *Черепашки*:

```
Tortoise.GetAngle() : double;
```

Метод **SetOrientation** объединяет усилия двух методов – *SetPosition* и *SetAngle:*

```
Tortoise. SetOrientation( x,  y, angle : integer)  : double;
```

Однако вернёмся к **спиралям** и напишем два простых метода для их вычерчивания:

```
private procedure Step(len, angle: double);
begin
   Tortoise.Move(len);
   Tortoise.Turn(angle);
end;
public procedure Spiral(len, angle, _step: double);
begin
   // устанавливаем Черепашку:
   Tortoise.SetX(80);
   Tortoise.SetY(CY + 200);
   Tortoise.SetSpeed(9);
   Tortoise.SetPenColor('Green');
   var i := len;
   while(i >= 10) do
   begin
      Step(i, angle);
      i -= _step;
   end;
end;
```

Задавая разные значения параметру **angle**, вы будете получать очень красивые *многоугольные спирали* (Рис. 13.34):

```
  // треугольная спираль:
  draw.Spiral(400, 120, 8);
```

```
  // квадратная спираль:
  draw.Spiral(400, 90, 4);
```

```
  // пятиугольная спираль:
  draw.Spiral(260, 72, 2);
```

```
// шестиугольная спираль:
draw.Spiral(200, 60, 2);
```

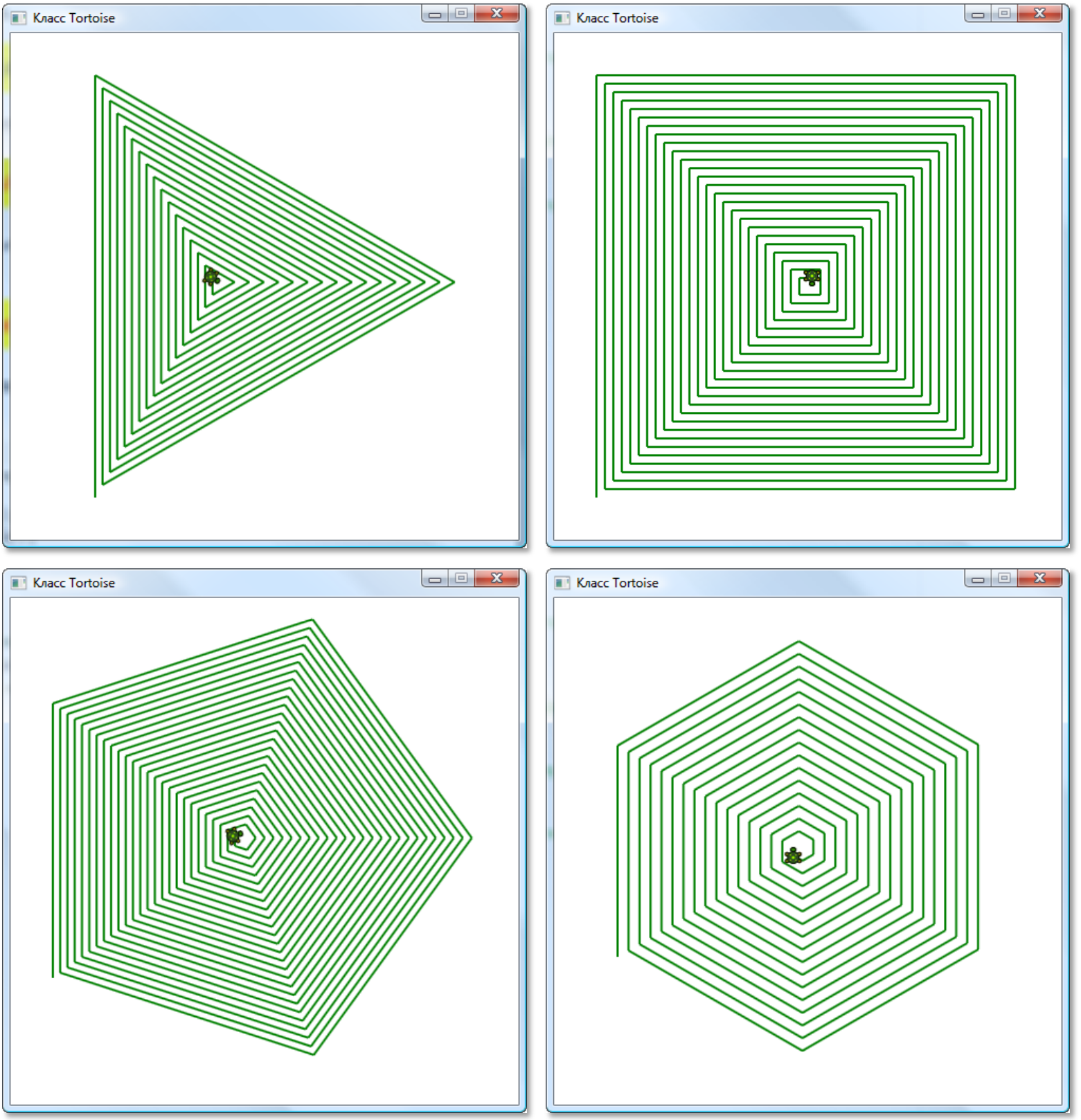


Рис. 13.34. Многоугольные спирали

Ещё более интересные спирали получаются, если задавать «неправильный» угол. Тогда спирали дополнительно **закручиваются** относительно центра (Рис. 13.35):

```
// треугольная спираль с закруткой:
draw.Spiral(400, 120 + 1, 8);
```

```
// квадратная спираль  с закруткой:
draw.Spiral(400, 90+0.5, 4); //+2 +1
```

```
// шестиугольная спираль с закруткой:
draw.Spiral(200, 60.3, 1);
```

Метод **SetPenColor** устанавливает цвет линий:

Tortoise. SetPenColor(**color** : string);

А метод **SetPenWidth** – их толщину:

Tortoise. SetPenWidth(**widthInPixels** : double);

Методы **GetPenColor** и **GetPenWidth** возвращают текущий цвет линий и их толщину:

Tortoise.GetPenColor() : string;

Tortoise.GetPenWidth() : double;

Метод **PenDown** опускает карандаш, и тогда *Черепашка* оставляет за собой след, а метод **PenUp** поднимает его, и *Черепашка* заметает следы:

Tortoise.PenDown();

Tortoise.PenUp();

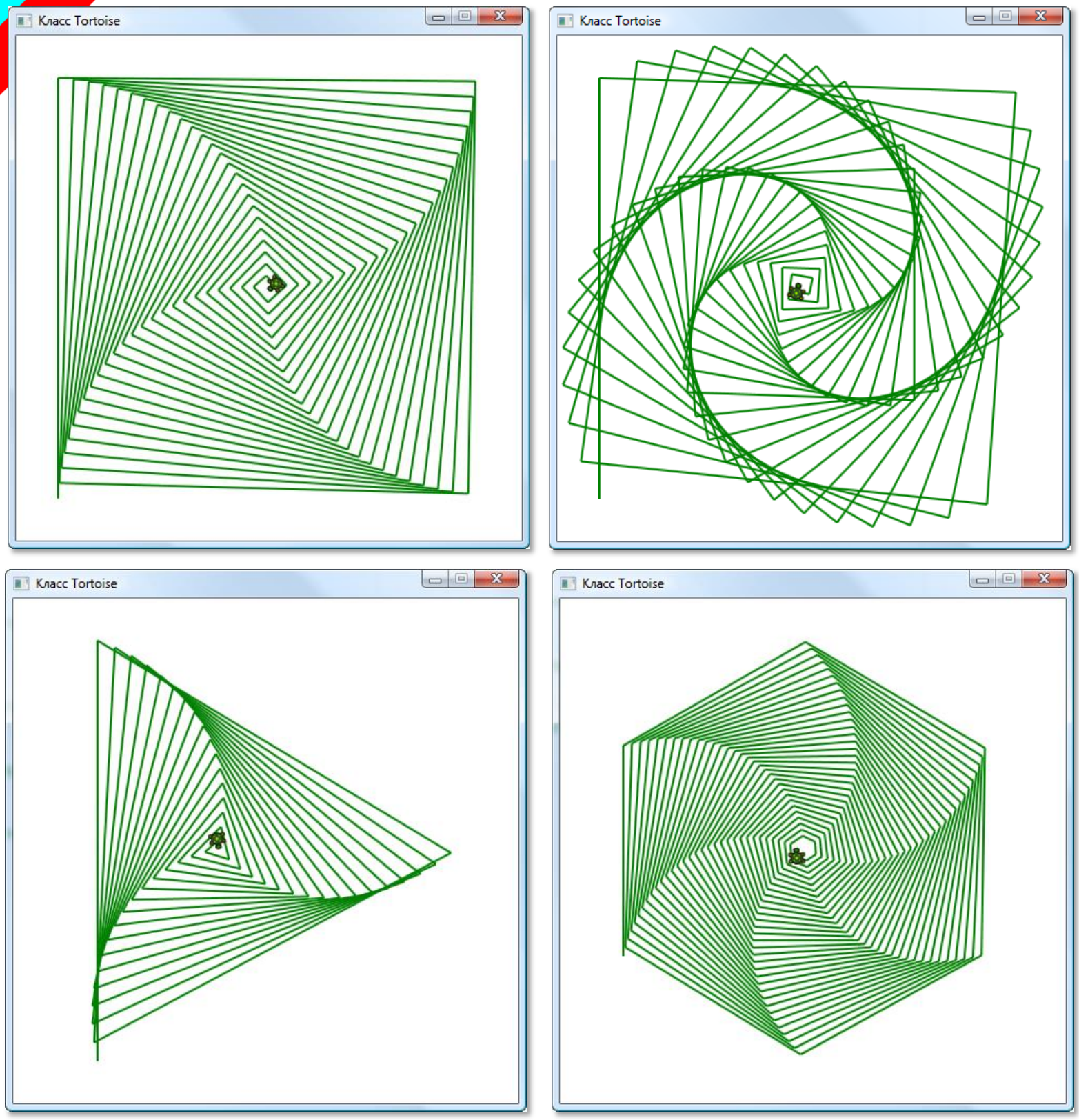


Рис. 13.35. Закрученные спирали

И последние методы отвечают за **скорость** движения *Черепашки*. Метод **SetSpeed** задаёт перцу и скорости *Черепашке*. Самая медленная скорость –

при значении параметра *speed*, равном 1, самая быстрая – при значении *speed*, равном 10:

```
Tortoise. SetSpeed(speed : integer);
```

Когда нужно максимально ускорить *Черепашку*, следует воспользоваться методом **InstantSpeed** с параметром **goFast** = *true*:

```
Tortoise.InstantSpeed( goFast : boolean);
```

Метод **GetSpeed** возвращает текущую скорость Черепашки:

```
Tortoise.GetSpeed() : integer;
```

А теперь давайте вычертим **фрезу** (Рис. 13.36):

```
   //draw.Squaggle(20, 3, 1);
   draw.Squaggle(77, 3.5, 1.01);


// фреза:
private procedure Squaggle(zoom, z: double);
begin
   Tortoise.Move(50*zoom);
   Tortoise.Turn(150/z);
   Tortoise.Move(60*zoom);
   Tortoise.Turn(100/z);
   Tortoise.Move(30*zoom);
   Tortoise.Turn(90/z);
end;

public procedure Squaggle(n: integer; zoom: double; z: double);
begin
   Tortoise.SetSpeed(10);
   Tortoise.SetX(CX-100);
   Tortoise.SetY(CY);
   Tortoise.SetPenColor('Green');
   for var i := 1 to n do
        Squaggle(zoom,z);
end;
```

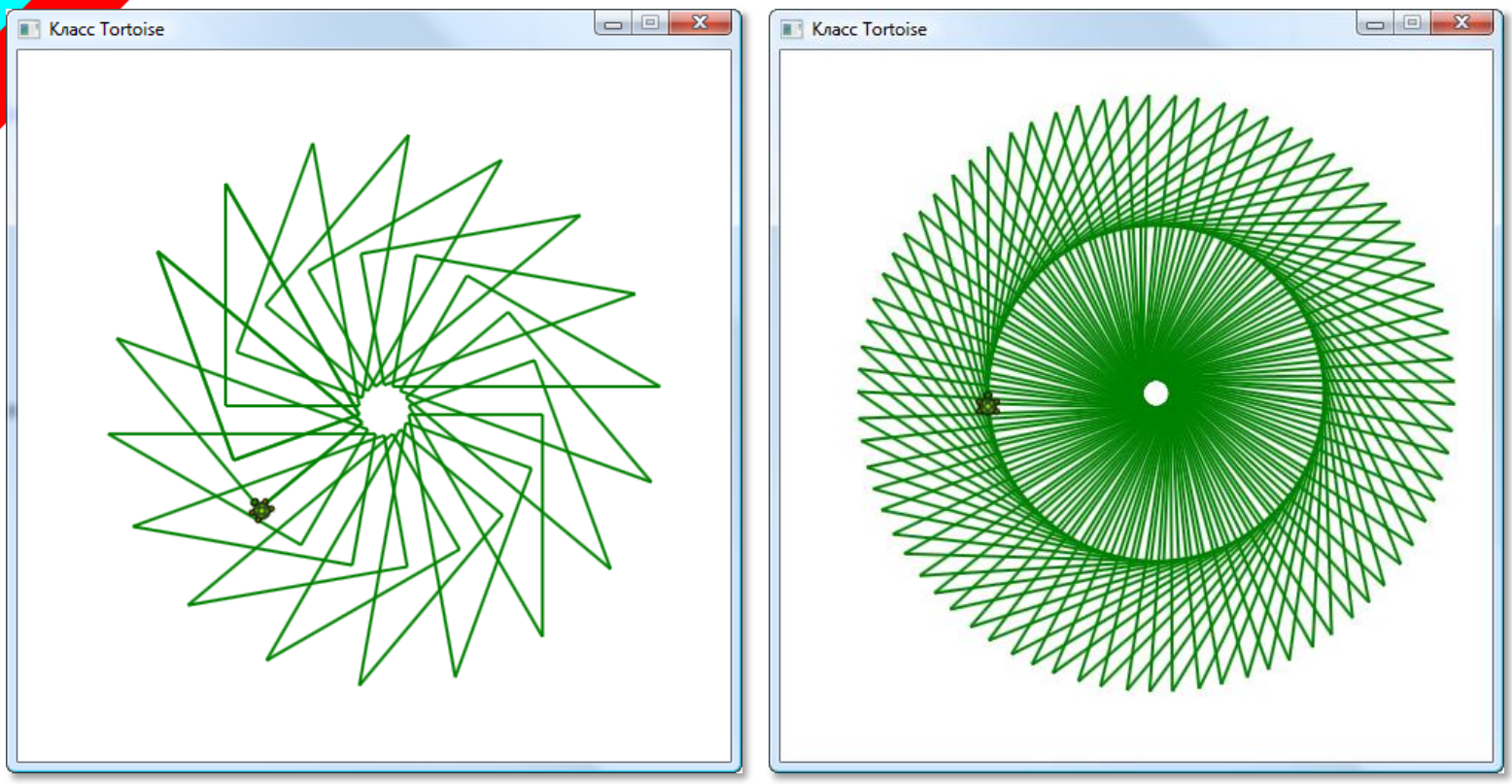


Рис13.36. Фрезерные фигуры

А можно и немного **почеркаться** прямо в окне приложения:

```
// почеркушки:
draw.Inspi();

   //почеркушки:
   private procedure Inspi(side, angle, inc: double);
   begin
      for var i := 0 to 10 - 1 do
      begin
         Tortoise.Move(side);
         Tortoise.Turn(angle + inc * i);
      end;
   end;

   public procedure Inspi();
   begin
      Tortoise.SetX(CX);
      Tortoise.SetY(CY);
      Tortoise.SetSpeed(10);
      while(true) do
      begin
         Tortoise.SetX(GraphicsWindow.MouseX);
         Tortoise.SetY(GraphicsWindow.MouseY);
```

```
            Tortoise.SetPenColor(GraphicsWindow.GetRandomColor());
            Inspi(100, 150, 1.5);
        end;
    end;
```

Запускаем программу и водим мыша за нос (Рис. 13.37).

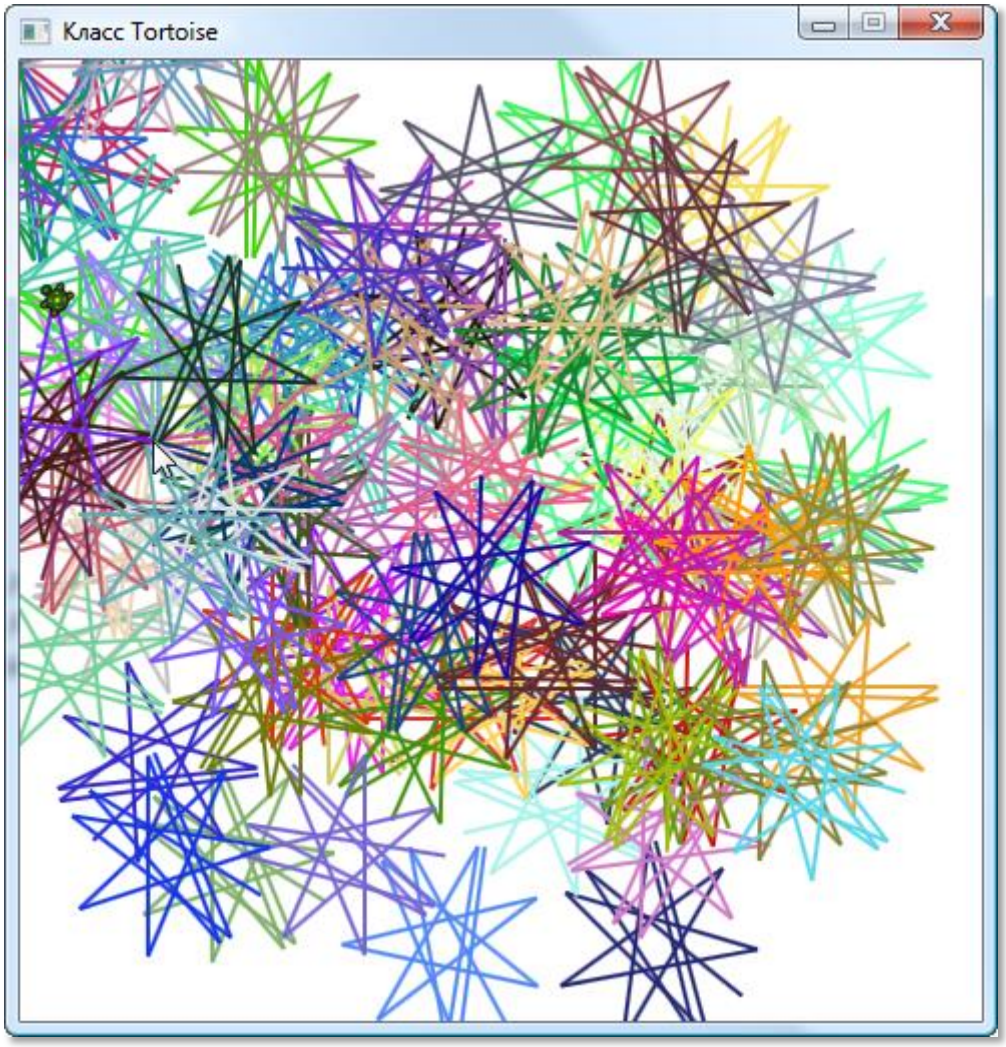


Рис. 13.37. Черепашьи почеркушки

Проект *Блуждающие Черепашки*

Исходный код программы находится в папке **Блуждающие Черепашки**.

Обе *Черепашки* – и большая, и маленькая – ведут свой род от совершенно дикой *Черепашки* из класса **MyTurtle**. Она плохо поддаётся дрессировке, но у неё есть одна способность, которая напрочь отсутствует у её одомашненных родственниц, - она умеет плодиться. И при создании каждой новой *Черепашки* ей можно задать свой любимый размерчик.

Давайте создадим для начала пару разнокалиберных *Черепашек*:

```
{$apptype windows}
// MyTurtle

uses
   DrawUnit;

begin
   var draw := new Draw();
   draw.Prepare();
end.

unit DrawUnit;

uses
   Microsoft.SmallBasic.Library, SmallBasicFun, System;

type
   Draw = class
   private
      const GWWIDTH = 480;
      const GWHEIGHT = 480;
      // размеры окна:
      width := GWWIDTH;
      height := GWHEIGHT;
```

```
      // координаты центра окна:
      CX := width div 2;
      CY := height div 2;


      public procedure Prepare();
      begin
         GraphicsWindow.Hide();
         GraphicsWindow.Title := 'Класс MyTurtle';
         GraphicsWindow.Width := GWWIDTH;
         GraphicsWindow.Height := GWHEIGHT;
         GraphicsWindow.Show();
         GraphicsWindow.Left := (Desktop.Width –
                                 GraphicsWindow.Width) / 2;
         GraphicsWindow.Top := (Desktop.Height –
                                GraphicsWindow.Height) / 2;
         // GraphicsWindow.CanResize := false;
         GraphicsWindow.BackgroundColor := 'White';
         // создаём первую Черепашку:
         var turtle1 := new MyTurtle();
         turtle1.Show();
         // устанавливаем Черепашку в центре окна:
         turtle1.X := CX;
         turtle1.Y := CY;
         // создаём вторую Черепашку:
         var turtle2 := new MyTurtle(1.6);
         turtle2.Show();
         turtle2.X := CX + 20;
         turtle2.Y := CY + 20;
      end;
   end; //end of class
end.
```

Как вы видите, дело это совершенно нехитрое – и вот уже новорождённые *Черепашки* предстали перед нашими глазами (Рис. 13.38).

Рис. 13.38. Двойняшки-*Черепашки*

Ооднииин ооочееень мееедлиииитееельныыый
прииибаааааалтиииееееец устроился работать
в зоопарк сторожем.
На следующий день все черепахи
расползлись во все стороны.
Директор зоопарка недоумевает:
- Как такое могло случиться?
Сторож отвечает:
- Неее знаааю.
Яяяя тоооолькооо ооооткрыыыл
двеерцууу клееетккиии,
аааа ооониии каааак лооомаааануууутсяяяяя!

Анекдот из книги *Европейский тормозной путь*

Немного поднатужившись, мы можем породить и 36 *Черепашек* – что мы глупее немцев с их питоновыми *Черепашками*?

Но породить, как говорил Тарас Бульба, мало – нужно ещё придать *Черепашкам* правильное направление и дать им крепкого родительского пинка под зад, чтобы они разбежались во все стороны:

```
public procedure Running();
begin
   var turtles := new MyTurtle[36];
   for var i := 0 to 36 - 1 do
   begin
      turtles[i] := new MyTurtle(1.6);
      turtles[i].X := CX;
      turtles[i].Y := CY;
      turtles[i].Angle := i * 360 / 36;
      turtles[i].Speed := 10;
      turtles[i].Show();
   end;
   foreach var t in turtles do
   begin
      t.Move(222);
   end;
end;
```

Моментальное семейное фото с места событий демонстрирует полную победу наших *Черепашек* над иноземными панцерами (Рис. 13.39).

А сейчас давайте смоделируем ситуацию поиска жизненного пути неориентированными в жизни созданиями.

В методе **RandomWalk** мы производим на свет 12 юных *Черепашек*, которые тут же пускаются в бесконечный путь познания добра и зла:

```
public procedure RandomWalk();
begin
   var rand := new Random();
   var turtles := new MyTurtle[12];
   for var i := 0 to 12 - 1 do
   begin
      turtles[i] := new MyTurtle(1.2);
      turtles[i].X := rand.Next(CX - 50, CX + 50);
      turtles[i].Y := rand.Next(CY - 50, CY + 50);
      turtles[i].LineColor :=
                  GraphicsWindow.GetRandomColor().AsColor();
      turtles[i].Speed := 10;
      turtles[i].Show();
   end;
   while(true) do
   begin
      for var i := 0 to 12 - 1 do
      begin
         turtles[i].Angle := rand.Next(360);
         turtles[i].Move(rand.Next(10, 50));
      end;
   end;
end;
```

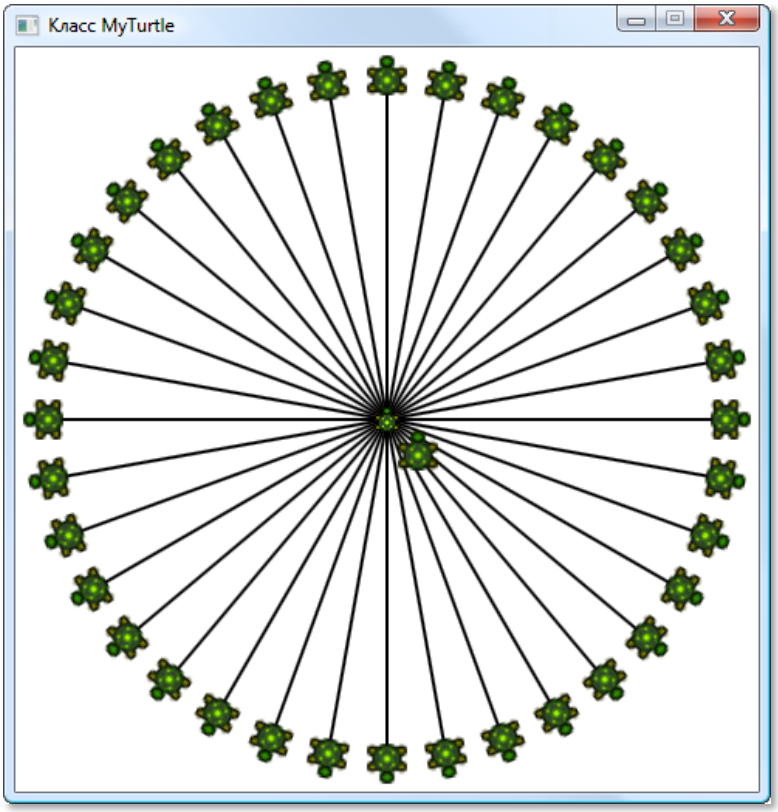


Рис. 13.39. Фигурно!

Поскольку ума у них ещё нет, а рвения хоть отбавляй, то они будут суетиться, ища место под солнцем, а их жизненный путь предстанет перед нашими глазами в виде перепутанного клубка нитей или своеобразного броуновского движения, которое англосаксоны называют *случайным блужданием* (Рис. 13.40).

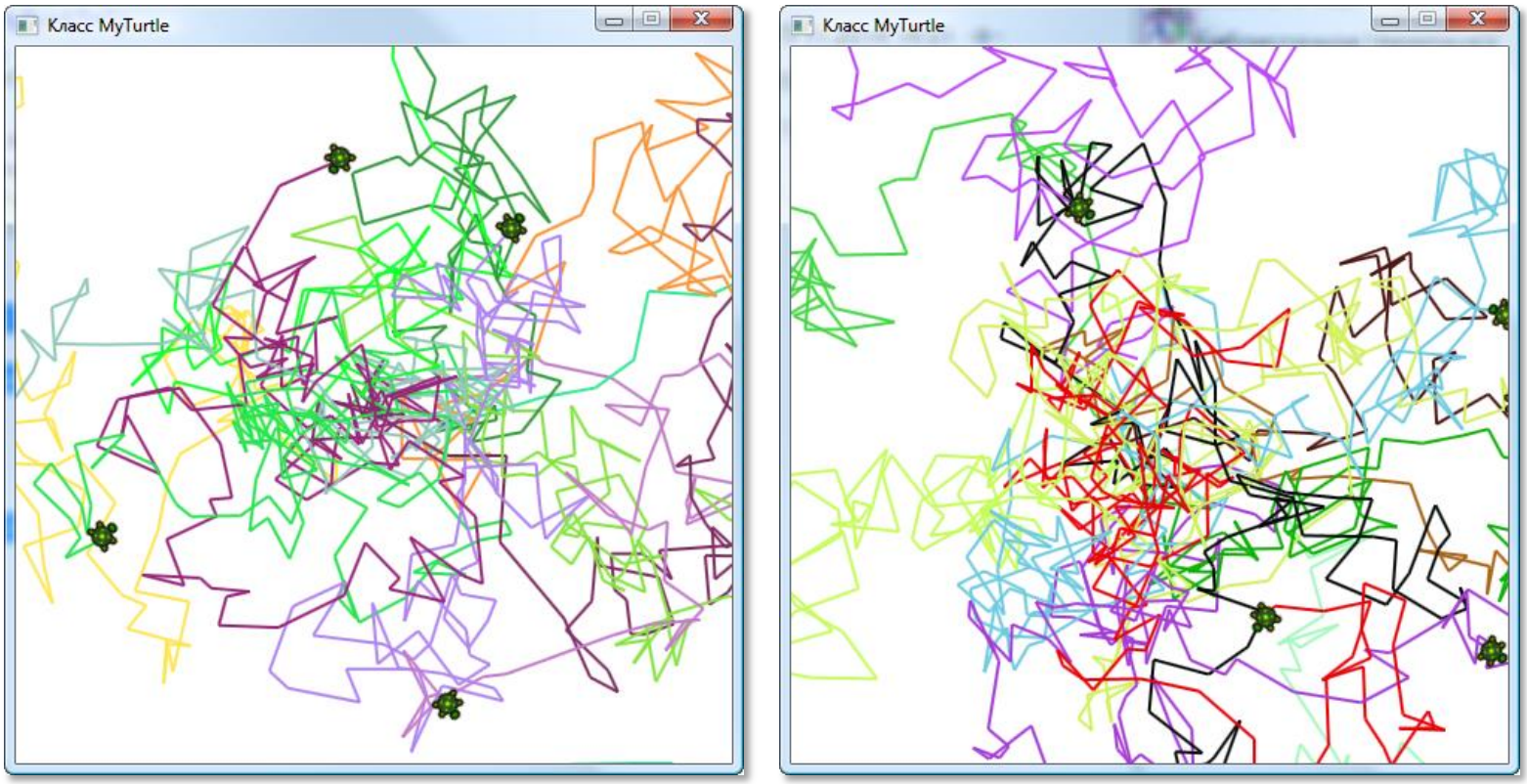


Рис. 13.40. Эх, куда глаза глядят!

А чего больше в этом эксперименте - физики или педагогики, - решайте сами!

Задания для самостоятельного решения

Черепашка *Tortoise*

Какую фигуру вычертит Черепашка, выполняя следующий код:

```
Tortoise.Move(100);
Tortoise.Turn(90);
Tortoise.Move(100);
```

А такой код:

```
for var i := 1 to 4 do
begin
  Tortoise.Move(100);
  Tortoise.Turn(90);
end;
```

Глава 14. Занимательная психология

Очень часто в популярных тестах по психологии встречаются задания, в которых нужно быстро и правильно *сосчитать* какие-нибудь предметы: цветки, бабочек, якоря – да что угодно. Только не думайте, что это пустая забава!

В замечательной книге *Вам – взлёт!* Анатолий Маркуша рассказал о таком случае из фронтовой жизни.

Когда в нелётную погоду все лётчики полка скучали и слонялись по аэродрому, старший лейтенант Нико Ломия играл в спички: бросит несколько штук, быстро взглянет на них, сгребёт в кулак, снова бросит… И так просиживал он целые дни. Конечно, товарищи начали переживать за него – не сошёл ли он с ума от тоски? Или гадает на хорошую погоду? Дошла эта история и до командира полка. Он вызвал лейтенанта, а тот объяснил ему, что таким необычным способом он тренирует зрительную память: бросит горсть спичек и пытается с одного взгляда определить, сколько их. Он уже справлялся с дюжиной спичек, но цель у него была – 50 спичек.

А через полгода он стал лучшим воздушным разведчиком фронта. На большой скорости пролетая над вражеской территорией, он мгновенно подсчитывал и запоминал: орудий - столько, самолетов – столько, танков – столько. И никогда не ошибался в своих подсчётах.

На следующей странице вы найдёте одно из таких заданий (Рис. 14.1). Вы можете просто пересчитать на время, сколько там разных цветков и листьев, но ещё лучше - сначала прикиньте, а потом проверьте, насколько вы ошиблись в своих предположениях.

Вы и сами легко можете сделать такие задачки! Возьмите чистый лист белой бумаги и в беспорядке нарисуйте какие-нибудь предметы. Можно использовать и красивые наклейки, тогда решать такие задачки будет ещё интереснее.

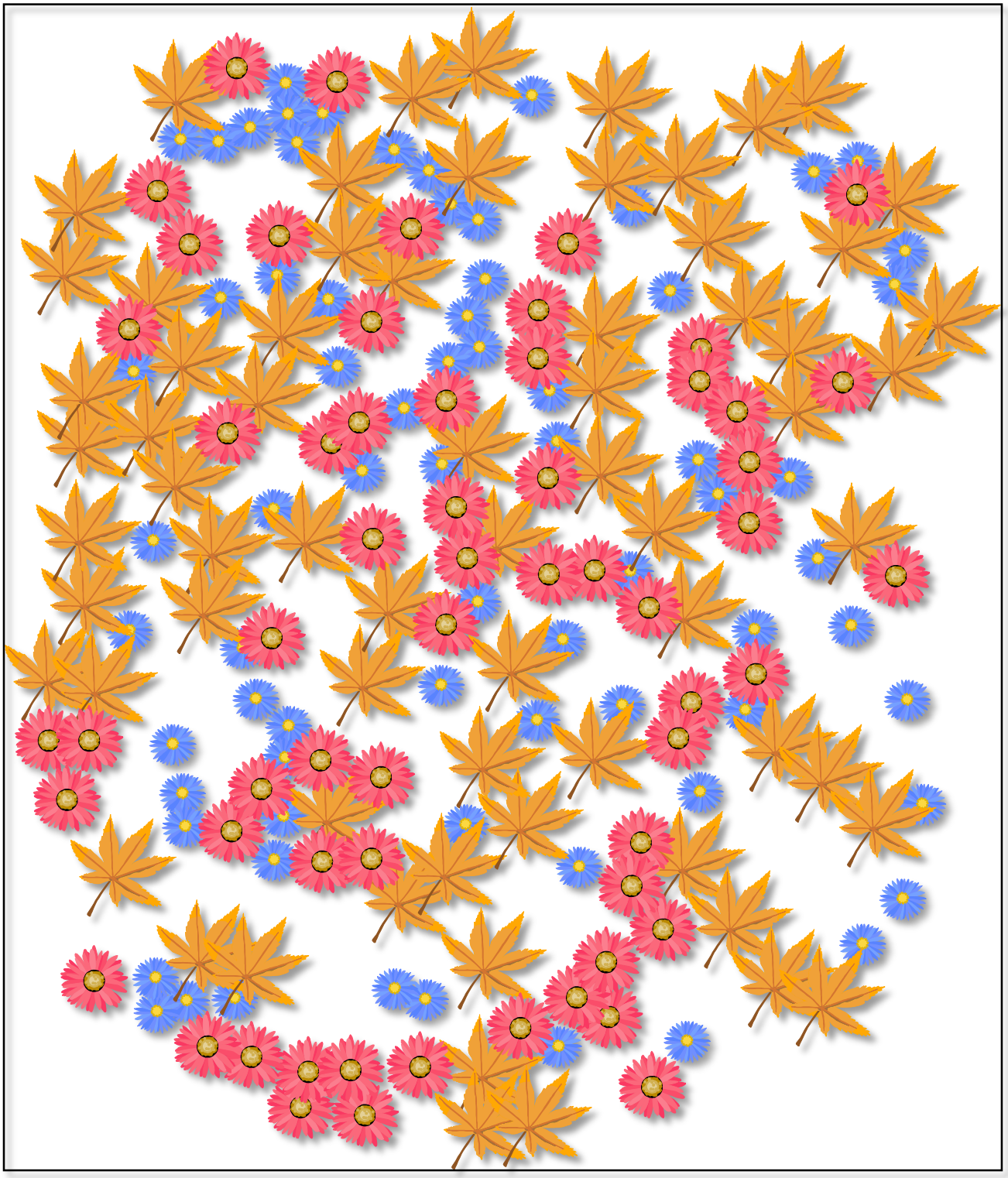


Рис. 14.1. Сосчитайте как можно быстрее, сколько здесь:

1. **Синих** цветков _____
2. **Красных** цветков _____
3. **Жёлтых** листьев _____

Проект *Психологическая считалка*

Исходный код программы находится в папке **Психологическая считалка**.

Но самоделки - это потом, а сейчас мы поступим иначе – напишем программу, которая будет сама делать за нас такие задачки. Для почина мы будем рисовать разноцветные кружочки, но вы можете заменить их (или добавить к ним) квадратами или треугольниками. В другой главе мы усовершенствуем программу так, чтобы она могла выводить на экран любые картинки, а не только простые геометрические фигуры.

А начнём мы новый проект **Психологическая считалка** с того, что создадим *Графическое окно* для нового приложения:

```
unit DrawUnit;

uses
   Microsoft.SmallBasic.Library, System;

type
   Draw = class
   private
      const GWWIDTH = 640;
      const GWHEIGHT = 480;
      const BackgroundColor = '#2F4F4F';
      // размеры окна:
      width := GWWIDTH;
      height := GWHEIGHT;
      // координаты центра окна:
      CX := width div 2;
      CY := height div 2;

      public procedure Prepare();
      begin
         GraphicsWindow.Hide();
         GraphicsWindow.Title := 'Психологическая считалка';
         GraphicsWindow.Width := GWWIDTH;
         GraphicsWindow.Height := GWHEIGHT;
         GraphicsWindow.Show();
         GraphicsWindow.Left := (Desktop.Width –
                                 GraphicsWindow.Width) / 2;
         GraphicsWindow.Top := (Desktop.Height –
                                GraphicsWindow.Height) / 2;
         GraphicsWindow.CanResize := false;
```

```
        GraphicsWindow.BackgroundColor := BackgroundColor;
    end;
```

Для определения времени, затраченного на выполнение теста, нам потребуется **таймер**, который будет срабатывать каждую секунду:

```
// таймер:
Timer.Interval := 1000;
Timer.Pause();
Timer.Tick += OnTick;
```

Метод-обработчик выводит время в **текстовое поле**:

```
// Отсчитываем время тестирования:
private procedure OnTick();
begin
   time += Timer.Interval / 1000;
   Controls.SetTextBoxText(txtTime, 'Время: ' + time.ToString());
end;
```

Его нужно создать:

```
// ТЕКСТОВЫЕ ПОЛЯ

// Время тестирования:
txtTime := Controls.AddTextBox(340, 10);
Controls.SetSize(txtTime, 100, 26);
Controls.HideControl(txtTime);
```

Не удивляйтесь комментарию *ТЕКСТОВЫЕ ПОЛЯ*: мы создадим ещё одно *текстовое поле* – для ввода ответа игрока:

```
// Проверка ответа:
txtOtvet := Controls.AddTextBox(180, 10);
Controls.SetSize(txtOtvet, 32, 26);
Controls.HideControl(txtOtvet);
```

Не обойтись нам и без пары **кнопок**:

```
// КНОПКИ

//B utton1 - начинаем тест:
btnStart := Controls.AddButton('Начать тестирование', 10, 10);
// Button2 - проверка:
btnProv := Controls.AddButton('Проверить', 220, 10);
Controls.ButtonClicked += OnClick;
Controls.HideControl(btnProv);
```

На этом создание нехитрого **интерфейса** программы закончено - показываем окно на экране (Рис. 14.2):

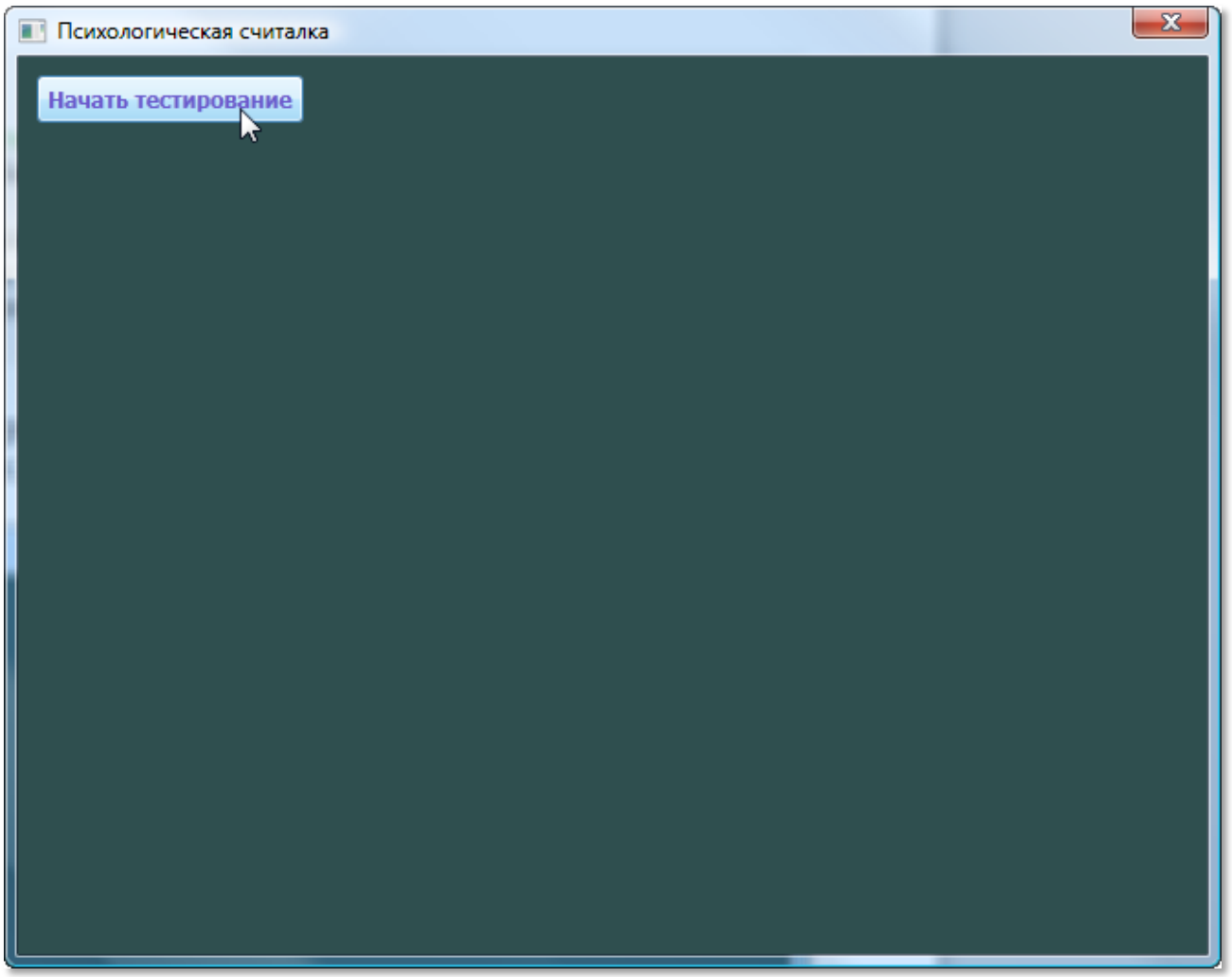


Рис. 14.2. Окно приложения

Одинокая кнопка не даст пользователю ни единого шанса запутаться в интерфейсе. И, поскольку выбора у него нет, он нажмёт кнопку *Начать тестирование*, и, как мы видим по коду, попадёт в метод-обработчик нажатия на кнопку **OnClick**:

```
private procedure OnClick();
begin
   var btn := (string)(Controls.LastClickedButton);
   var s := String.Empty;
   if (btn = btnStart) then / /начинаем тестирование
   begin
      Controls.HideControl(btnStart);
      Controls.ShowControl(btnProv);
      CreateTest();
      Controls.SetTextBoxText(txtTime, '');
      Controls.ShowControl(txtTime);
      Controls.SetTextBoxText(txtOtvet, '');
      Controls.ShowControl(txtOtvet);
      time := 0;
      Timer.Resume();
   end
   else   // проверяем ответ
   begin
      Controls.HideControl(btnProv);
      Controls.ShowControl(btnStart);
      Timer.Pause();
      if (nKrug = Int32(Controls.GetTextBoxText(txtOtvet))) then
      begin
         s := 'ПРАВИЛЬНО!';
         GraphicsWindow.BrushColor := 'Yellow';
      end
      else
      begin
         s := 'НЕПРАВИЛЬНО!';
         GraphicsWindow.BrushColor := 'Red';
      end;
      GraphicsWindow.FontSize := 48;
      GraphicsWindow.DrawText(140, height / 2 - 24, s);
   end
end;
```

Так как нажатие всех кнопок в нашей программе обрабатывает один и тот же метод, то нам нужно, в первую очередь, узнать, какую именно кнопку нажал пользователь:

```
var btn := (string)(Controls.LastClickedButton);
```

Если это кнопка **btnStart**, то мы начинаем тестирование, если **btnProv** - проверяем ответ пользователя.

Для тестирования нам нужно создать тест с помощью метода **CreateTest**, а также спрятать одни элементы управления и показать другие, чтобы пользователь в них не запутался.

Итак, кнопка нажата – **создаём новый тест**. Для этого мы просто разбрасываем по клиентской области окна **разноцветные** кружочки:

```
private procedure CreateTest();
begin
   // чистим окно:
   GraphicsWindow.BrushColor := BackgroundColor;
   GraphicsWindow.FillRectangle(0, 40, width, height - 40);
   // число кружков::
   nKrug := rand.Next(30) + 21;
   // отладка:
   //GraphicsWindow.Title := nKrug;
   for var i := 1 to nKrug do
   begin
      // выбираем случайный радиус кружка:
      var radius := rand.Next(16) + 20;
      // выбираем случайные координаты  кружка:
      var x := rand.Next(width - 2 * radius);
      var y := rand.Next(height - 2 * radius - 40) + 40;
      // выбираем случайный цвет для  кружка:
      var clr := GraphicsWindow.GetRandomColor();
      GraphicsWindow.BrushColor := clr;
      // рисуем цветной кружок:
      GraphicsWindow.FillEllipse(x, y, 2 * radius, 2 * radius);
   end;
end;
```

Прежде всего, нужно подготовить рабочее место, то есть убрать с канвы всё лишнее:

```
//чистим окно:
GraphicsWindow.BrushColor := BackgroundColor;
GraphicsWindow.FillRectangle(0, 40, width, height - 40);
```

Метод *GraphicsWindow.Clear* здесь не проходит, потому что под его горячую руку попадет всё, что уже нарисовано на канве, в том числе и элементы управления! Поэтому мы нарисуем закрашенный цветом фона окна прямоугольник нужных нам размеров. Глядя на первую картинку, можно подумать, что канва и без того чистая, но вот что с ней случится после нажатия на кнопку (Рис. 14.3).

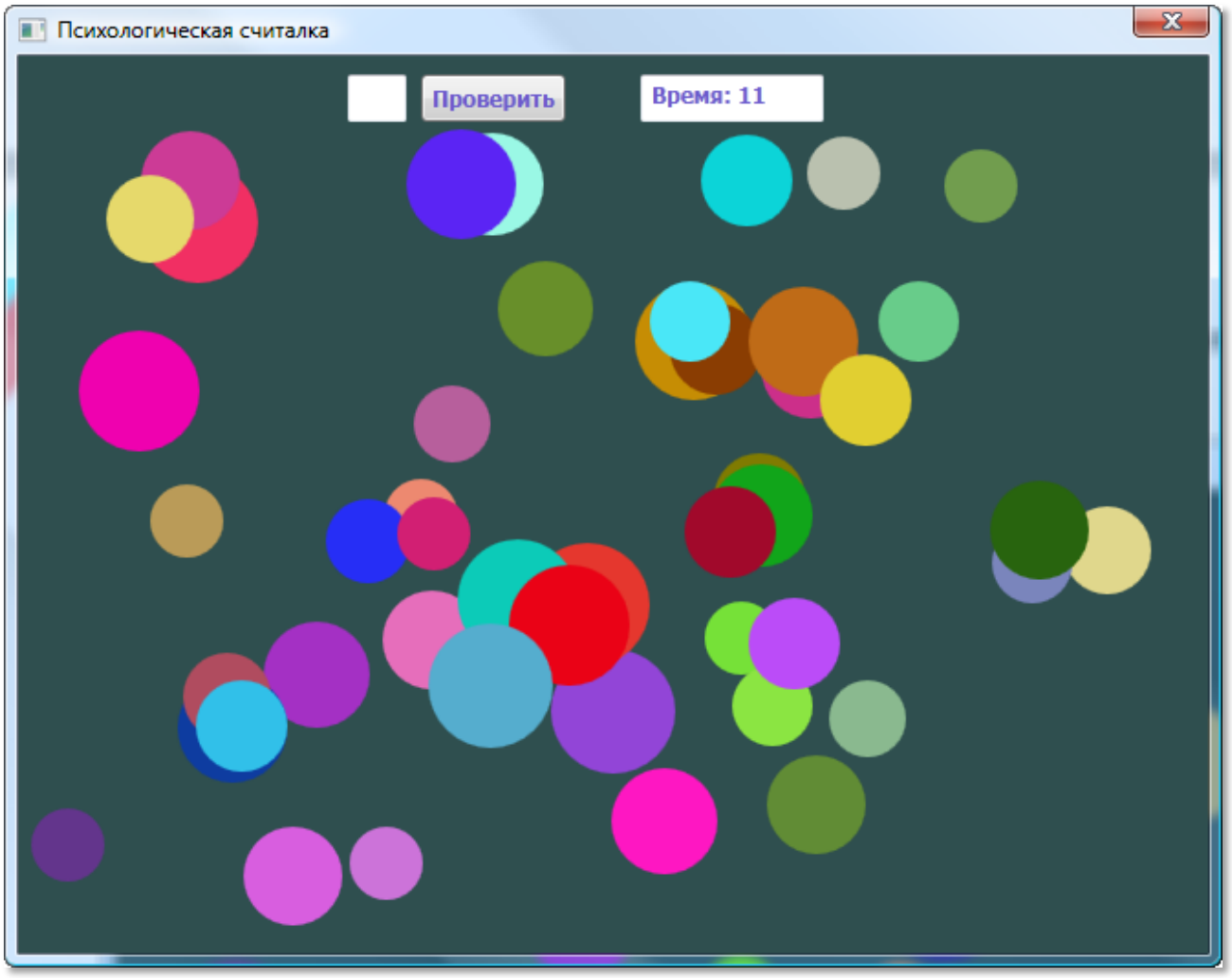


Рис. 14.3. А вот и цветные кружочки!

Если вы захотите продолжить тестирование, то следующие кружочки будут нарисованы поверх прежних, и вы вместо теста получите кашу!

Число кружочков для каждого тестирования нужно выбирать другим, иначе будет неинтересно. В данном примере их будет от 21 до 50:

```
time: double;
txtTime: string;
txtOtvet: string;
btnStart: string;
btnProv: string;
nKrug: integer;
rand := new Random();


// число кружков::
nKrug := rand.Next(30) + 21;
```

Вы легко можете выбрать другой интервал, например, для младших школьников лучше подойдет диапазон от 3 до 11.

При отладке программы очень важно знать, сколько кружков на поле, чтобы не пересчитывать их всякий раз заново, ведь программу придётся доводить до ума довольно долго:

```
// отладка:
//GraphicsWindow.Title := nKrug;
```

Всегда вставляйте в свои программы **отладочные операторы**. А когда они станут не нужны, просто закомментируйте их.

Теперь можно нарисовать на экране заданное число кружков. Их размеры, цвет и положение мы также выбираем случайным образом, чтобы задания отличались друг от друга:

```
// выбираем случайный радиус кружка:
var radius := rand.Next(16) + 20;
```

Радиус кружков задайте по своему вкусу!

Выбирать место для нового кружка следует так, чтобы он целиком поместился на канве, а также не вторгался в ту область канвы, где расположены элементы управления программой:

```
// выбираем случайные координаты  кружка:
var x := rand.Next(width - 2 * radius);
var y := rand.Next(height - 2 * radius - 40) + 40;
```

Осталось **нарисовать** кружок на экране:

```
// рисуем цветной кружок:
GraphicsWindow.FillEllipse(x, y, 2 * radius, 2 * radius);
```

Когда построение теста закончено, программа начинает отсчёт времени:

```
Controls.SetTextBoxText(txtTime, '');
Controls.ShowControl(txtTime);
Controls.SetTextBoxText(txtOtvet, '');
Controls.ShowControl(txtOtvet);
time := 0;
Timer.Resume();
```

Испытуемый, подсчитав кружки на поле, нажимает кнопку **Проверить**, после чего программа переходит к обработке этого события в методе *OnClick*.

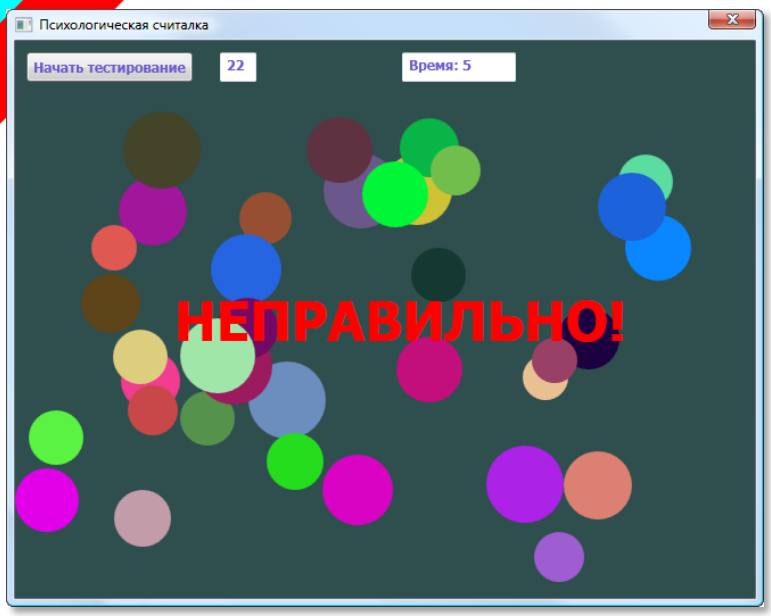
Таймер останавливается и показывает затраченное на тестирование **время**:

```
Timer.Pause();
```

Затем в методе *OnClick* сравнивается число, введённое пользователем с клавиатуры в *текстовое поле txtOtvet*, с действительным числом кружков, и выдаёт **результат** теста (Рис. 14.4):

```
if (nKrug = Int32(Controls.GetTextBoxText(txtOtvet))) then
begin
   s := 'ПРАВИЛЬНО!';
   GraphicsWindow.BrushColor := 'Yellow';
end
         else
begin
   s := 'НЕПРАВИЛЬНО!';
   GraphicsWindow.BrushColor := 'Red';
end;
```

На рисунках видно (и по исходному тексту тоже!), что опять появилась призывная кнопка **Начать тестирование** - *Психологическая считалка* снова в бою!



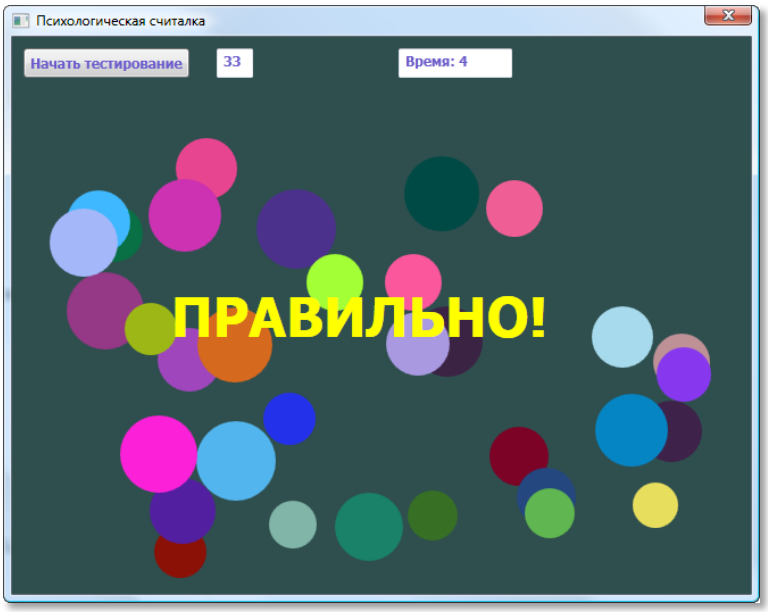


Рис. 14.4. Пользователь погорячился! Результат тренировок налицо!

Задания для самостоятельного решения

Психологическая считалка

1. В случае неправильного ответа программа только сообщает пользователю, что он неверно подсчитал кружочки. Добавьте к строке **правильное** число кружков, чтобы пользователь мог оценить, насколько он ошибся.

2. Так как цвет кружков выбирается случайно, то он может совпасть с цветом фона *BackgroundColor='#2F4F4F'* и тогда будет невидим для пользователя. Вероятность такого события очень невелика, но её следует учесть, например, так:

```
repeat
   //выбираем случайный цвет для  кружка:
   clr := GraphicsWindow.GetRandomColor();
until not(clr = BackgroundColor);

GraphicsWindow.BrushColor := clr;
```

3. Другая неприятность заключается в том, что БОЛЬШОЙ кружок может полностью закрыть маленький, который появился на экране раньше. То, что некоторые кружки частично перекрывают друг друга, это неплохо, так как это делает задачу более трудной. Однако полностью закрытый кружок вообще не видно, поэтому результат тестирования будет неверным. Чтобы этого избежать, будем проверять, куда попадает центр нового кружка. Если на пустое место, то всё нормально, кружок можно рисовать, иначе нужно повторить попытку:

```
for var n := 0 to 100-1 do
begin
      //выбираем случайный радиус кружка:
      radius := rand.Next(16)+ 20;
      //выбираем случайные координаты  кружка:
      var x := rand.Next(width - 2*radius);
      var y := rand.Next(height - 2*radius - 40) + 40;
      if (GraphicsWindow.GetPixel(x+radius,y+radius) =
        BackgroundColor) then
      break;
end;
```

Для чего нужен цикл *for*? - Переменная цикла *n* подсчитывает число неудачных попыток нарисовать кружок в чистом поле. После 100 попыток кружок будет нарисован наудачу. Действительно, на поле может возникнуть ситуация, когда место для нового кружка будет трудно найти (а если кружков много и они большого размера, то вообще не найти) и программа будет тратить много времени на создание теста (или вообще зациклится!). Конечно, такую ситуацию мы не должны допустить.

Если размеры кружков очень сильно различаются, то и в этом случае может произойти полное «затмение» некоторых кружков со всеми вытекающими отсюда последствиями. Можно несколько улучшить ситуацию, но полностью предотвратить её нельзя. Поэтому ограничивайте количество кружков и не делайте их слишком разными «по росту»!

Глава 15. Звёздное небо

Открылась бездна, звезд полна…

М.В.Ломоносов

Трудно быть богом

Братья Стругацкие

Если бы мы жили в центре Галактики, то звёздное небо очень бы напоминало картинку из *Психологической считалки*. Но нам в смысле звёздной красоты повезло значительно меньше, поскольку мы живем на периферии, на забытой богом спирали. Куда ни глянь: звёзды мелкие и **жёлтые**.

> На самом деле звёзды БОЛЬШИЕ и **разноцветные**, просто они нам такими кажутся из-за огромного расстояния до них. Правда, даже в самый сильный телескоп звёзды всё равно выглядят точками, хотя и **цветными**.

Если усыпать канву маленькими кружочками, то мы получим картину, вполне *напоминающую* настоящее небо.

> Чтобы небо выглядело натурально, придётся задать координаты *всех* видимых глазом звёзд, а это несколько тысяч штук! Также необходимо учесть их цвет и блеск (звёздную величину).

Поскольку нам не нужно заботиться о перекрывании звёзд и прочих премудростях, с которыми мы столкнулись в психологическом тесте, то программа получится очень простая.

> Звёзды и на самом деле закрывают друг друга, например, затменные переменные звёзды делают это периодически.

Проект *Звёздное небо*

Исходный код программы находится в папке **Звёздное небо**.

Начните новый проект **Звёздное небо** с заголовка окна и цвета неба (Рис. 15.1, слева):

```
unit DrawUnit;

uses
   Microsoft.SmallBasic.Library, System;

type
   Draw = class
   private
      const GWWIDTH = 640;
      const GWHEIGHT = 480;
      // цвет неба:
      const BackgroundColor = 'MidnightBlue'; //'Black';

      // размеры окна:
      width := GWWIDTH;
      height := GWHEIGHT;
      // координаты центра окна:
      CX := width div 2;
      CY := height div 2;

      btnStart: string;
      rand := new Random();

      public procedure Prepare();
      begin
         GraphicsWindow.Hide();
         GraphicsWindow.Title := 'Звёздное небо';
         GraphicsWindow.Width := GWWIDTH;
         GraphicsWindow.Height := GWHEIGHT;
         GraphicsWindow.Show();
         GraphicsWindow.Left := (Desktop.Width –
                                 GraphicsWindow.Width) / 2;
         GraphicsWindow.Top := (Desktop.Height –
                                GraphicsWindow.Height) / 2;
         GraphicsWindow.CanResize := false;
         GraphicsWindow.BackgroundColor := BackgroundColor;
```

Если хотите, можете сделать небо совсем **чёрным** (Рис. 15.1, справа), тогда звёзды будут казаться более яркими:

```
GraphicsWindow.BackgroundColor := 'Black';
```

Нам будет достаточно одной **кнопки**:

```
    // КНОПКА
    btnStart := Controls.AddButton('Новое небо', 10, 10);
    Controls.ButtonClicked += OnClick;
end;
```

Вот так, с помощью единственной кнопки, мы будем менять картину мира в галактических масштабах!

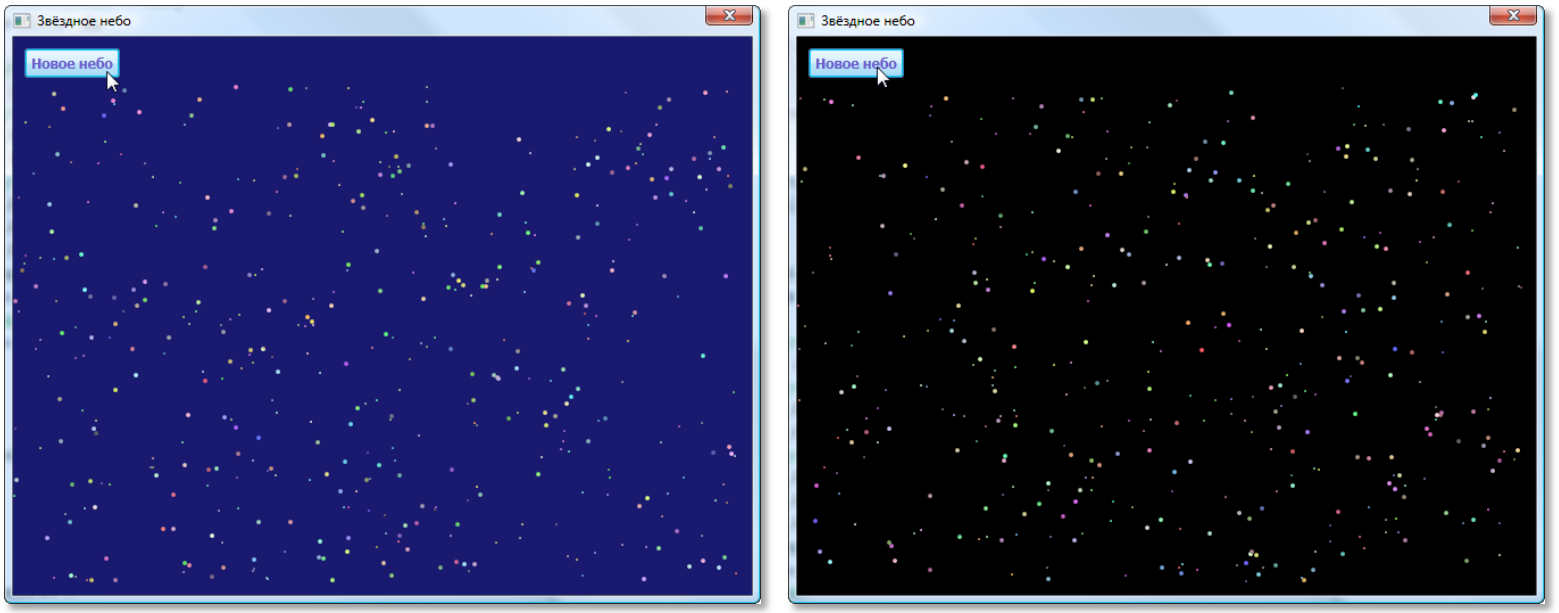


Рис. 15.1. Звёзд много и все они разные, но нет на искусственном небе наших родных созвездий Ориона, Лиры, Пегаса, Лебедя и обеих Медведиц!

Как только вы нажмёте заветную кнопку Творца

```
private procedure OnClick();
begin
   CreateSky();
end;
```

будет вызван метод **CreateSky**, где произойдет *Большой взрыв*, и небо украсится мириадами звёзд:

```
private procedure CreateSky();
begin
   // чистим небо:
   GraphicsWindow.BrushColor := BackgroundColor;
   GraphicsWindow.FillRectangle(0, 40, width+8, height - 40+8);

   // число звезд:
   var nStar := rand.Next(101) + 400;
   for var i := 1 to nStar do
   begin
      // выбираем случайный радиус звезды:
      var radius := rand.Next(2) + 1;
      // выбираем случайные координаты  звезды:
      var x := rand.Next(width - 2 * radius);
      var y := rand.Next(height - 2 * radius - 40) + 40;
      //  выбираем случайный цвет для  звезды:
      var r := rand.Next(164) + 92;
      var g := rand.Next(164) + 92;
      var b := rand.Next(164) + 92;
      var clr := GraphicsWindow.GetColorFromRGB(r, g, b);
      GraphicsWindow.BrushColor := clr;
      //рисуем звезду:
      GraphicsWindow.FillEllipse(x, y, 2 * radius, 2 * radius);
   end
end;
```

С точки зрения программирования, ничего таинственного в этом созидательном акте нет. Единственное, на что следует обратить внимание: мы выбираем случайный цвет звезды не с помощью метода *GetRandomColor,* что было бы проще. Но нам нужны достаточно яркие звезды (иначе они потеряются на небе), поэтому мы задаём такие составляющие цвета *r, g, b*, чтобы они были от 92 (самые тусклые) до 255 (самые яркие), а далее формируем из них цвет в методе *GetColorFromRGB:*

```
var clr := GraphicsWindow.GetColorFromRGB(r, g, b);
```

Если вы останетесь недовольны видом созданного вами неба, смело жмите на кнопку *Новое небо*!

Задания для самостоятельного решения

Звёздное небо

1. Кроме одиночных и кратных звёзд, с рисованием которых наша программа неплохо справляется, на небе можно увидеть и **звёздные скопления** – группы звезд, связанных общим происхождением и силами гравитации. Они бывают *рассеянные* и *шаровые*.

Самое известное рассеянное звёздное скопление в наших широтах – это *Плеяды*, или *Стожары*. Оно находится в созвездии Тельца и сравнительно недалеко от Солнца, поэтому многие звёзды этого скопления можно видеть невооружённым глазом.

Давайте добавим к нашему звёздному небу несколько рассеянных скоплений (Рис. 15.2). Для нас важно знать, что в них не очень много звёзд и они белого и **голубого** цвета.

Для создания рассеянных звездных скоплений (РЗС) нам потребуется ещё одна **кнопка**:

```
btnAddOpenCluster: string;
btnAddOpenCluster := Controls.AddButton('Добавить РЗС', 100, 10);
```

Потребуется переписать и метод-обработчик:

```
private procedure OnClick();
begin
   var btn := Controls.LastClickedButton.ToString();
   if (btn = btnStart) then
      CreateSky()
   else if (btn = btnAddOpenCluster) then
      CreateOpenCluster();
end;
```

Но самое главное – в новом **методе**:

```
// СОЗДАЁМ РАССЕЯННОЕ ЗВЕЗДНОЕ СКОПЛЕНИЕ
private procedure CreateOpenCluster();
begin
   //число звёзд:
   var nStar := rand.Next(10) + 8;
   //диаметр скопления:
   var rOC := rand.Next(20) + 21;
   //левый верхний угол скопления:
```

```
   var xOC := rand.Next(width - 2 * rOC);
   var yOC := rand.Next(height - 2 * rOC - 40) + 40;

   for var i := 1 to nStar do
   begin
      // выбираем случайный радиус звезды:
      var radius := rand.Next(2) + 1;
      // выбираем случайные координаты  звезды:
      var x := rand.Next(2 * rOC) + xOC;
      var y := rand.Next(2 * rOC - 40) + 40 + yOC;
      // выбираем случайный цвет для  звезды:
      var c := rand.Next(4);
      var clr := '';
      if (c = 0) then
         clr := 'White'
      else if (c = 1) then
         clr := 'AliceBlue'
      else if (c = 2) then
         clr := 'LightSkyBlue'
      else
         clr := 'DeepSkyBlue';

      GraphicsWindow.BrushColor := clr;
      // рисуем звезду:
      GraphicsWindow.FillEllipse(x, y, 2 * radius, 2 * radius);
   end
end;
```

Пользуясь этими подсказками, **напишите новый вариант программы** (Рис. 15.3)!

Рис. 15.2. Семь рассеянных скоплений

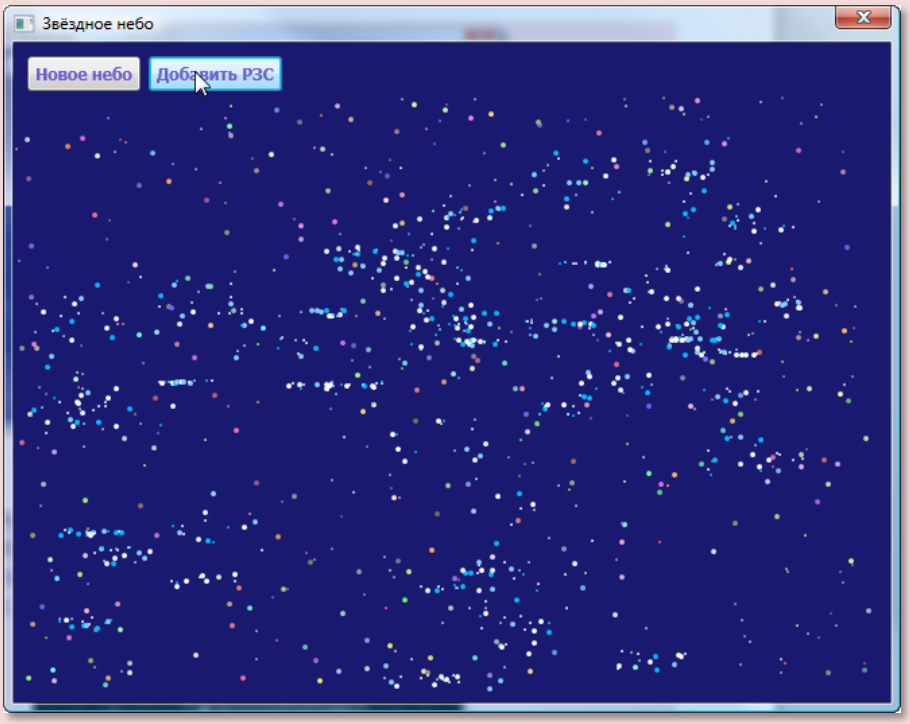


Рис. 15.3. Звёздное небо стало живописнее!

2. Любуясь звёздами, вы, конечно, не могли забыть о **шаровых** звёздных скоплениях, в которых звёзд гораздо больше, чем в рассеянных, и форма у них сферическая, причем концентрация звёзд увеличивается к центру скопления (Рис. 15.4).

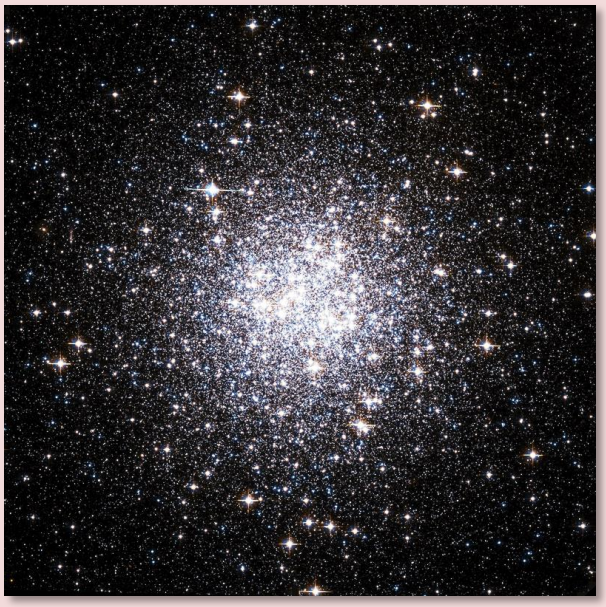


Рис. 15.4. Шаровое звёздное скопление

Цвет звёзд в таких скоплениях – от **белого** до **красного**, то есть можно брать такие же цвета, как для одиночных звёзд в методе *createSky*.

Исходный код программы находится в папке **Звёздное небо 2**.

Глава 16. С первого взгляда!

Когда видишь деньги,
не теряй времени!

Кинокомедия *Бриллиантовая рука*

В главе *Занимательная психология* мы говорили о задачах, в которых требуется очень быстро, с первого взгляда определить количество предметов. Пример такого психологического теста вы найдёте на Рис. 16.1.

Конечно, решая задание в книге, вы можете и схитрить - подглядывать или считать монетки пальцем. Да и задание только одно: решил - и сказочке конец. А ведь так хочется продлить удовольствие... И тут нам опять поможет компьютер!

Проект *С первого взгляда!*

Исходный код программы находится в папке **С первого взгляда!**.

Начните проект **С первого взгляда!**, сохраните его в папке и объявите **константы** и **переменные**:

```
unit DrawUnit;

uses
   Microsoft.SmallBasic.Library, System;

type
   Draw = class
   private
      const GWWIDTH = 640;
      const GWHEIGHT = 480;
      // цвет фона:
      const  BackgroundColor = '#FFFFF0';// 'Ivory';
```

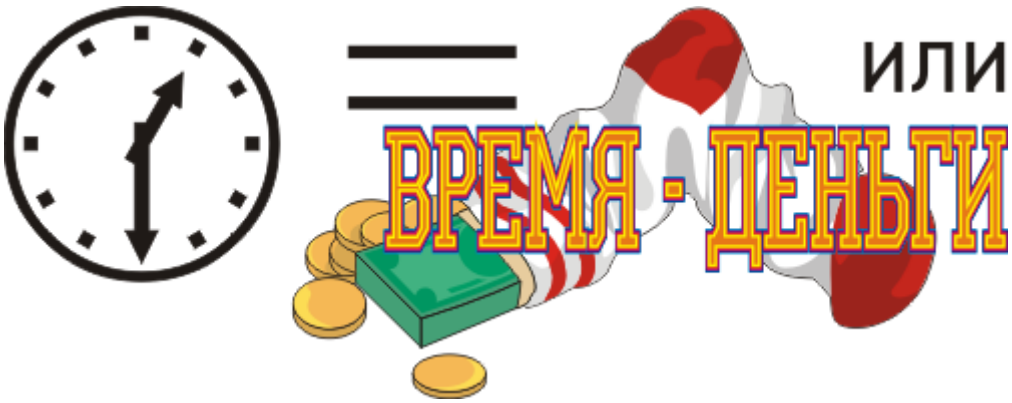
```
    // размеры окна:
    width := GWWIDTH;
    height := GWHEIGHT;
    // координаты центра окна:
    CX := width div 2;
    CY := height div 2;

    // время:
    time: double;
    txtTime: string;
    txtOtvet: string;
    btnStart: string;
    btnProv: string;
    // число разных картинок:
    const N_IMAGE = 6;
    img := new string[N_IMAGE];
    // число картинок на экране:
    nImage: integer;

    rand := new Random();
```

Настройте **окно** приложения:

```
    public procedure Prepare();
    begin
       GraphicsWindow.Hide();
       GraphicsWindow.Title := 'С первого взгляда!';';
       GraphicsWindow.Width := GWWIDTH;
       GraphicsWindow.Height := GWHEIGHT;
       GraphicsWindow.Show();
       GraphicsWindow.Left := (Desktop.Width –
                               GraphicsWindow.Width) / 2;
       GraphicsWindow.Top := (Desktop.Height –
                              GraphicsWindow.Height) / 2;
       GraphicsWindow.CanResize := false;
       GraphicsWindow.BackgroundColor := BackgroundColor;
```

Для отсчёта заданного времени потребуется **таймер**:

```
// таймер:
time := 0;
Timer.Interval := 1000;
Timer.Pause();
Timer.Tick += OnTick;
```



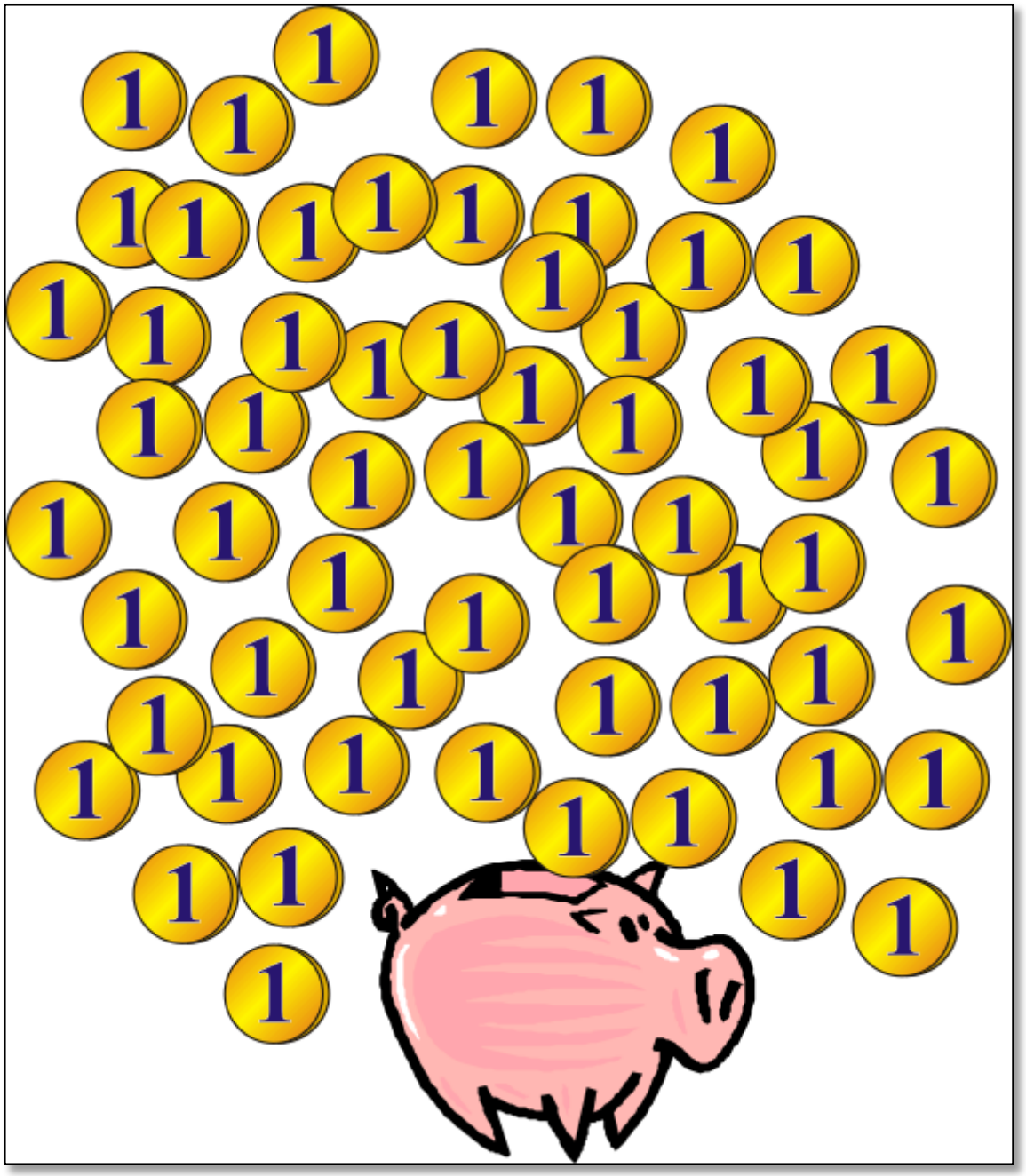


Рис. 16.1. Определите с первого взгляда, **сколько здесь монет**

А ещё - **картинки** с монетами. Согласно константе *N_IMAGE*, их должно быть шесть штук. Хранятся они в папке с исходным кодом программы, поэтому путь к ним легко найти:

```
var path := Environment.CurrentDirectory + '\';
```

Загрузите картинки в *Список картинок* **ImageList** с помощью метода *LoadImage:*

```
img[0] := ImageList.LoadImage(path + '1.png');
img[1] := ImageList.LoadImage(path + '2.png');
img[2] := ImageList.LoadImage(path + '3.png');
img[3] := ImageList.LoadImage(path + '4.png');
img[4] := ImageList.LoadImage(path + '5.png');
img[5] := ImageList.LoadImage(path + '6.png');
```

Всё – картинки у вас в кармане! Теперь вы можете рисовать их на канве - где угодно и сколько угодно.

Элементы управления - точно такие же, как в программе *Психологическая считалка*, как и б***о***льшая часть этой программы.

Метод **OnClick** изменился незначительно:

```
private procedure OnClick();
begin
   var btn := (string)(Controls.LastClickedButton);
   var s := String.Empty;
   if (btn = btnStart) then // начинаем тестирование
   begin
      Controls.HideControl(btnStart);
      Controls.HideControl(txtOtvet);
      CreateTest();
      Controls.SetTextBoxText(txtTime, 'Время: 0');
      Controls.ShowControl(txtTime);
      Controls.SetTextBoxText(txtOtvet, '');
      Controls.ShowControl(txtOtvet);
      time := 0;
      Timer.Resume();
      Controls.ShowControl(txtTime);
   end
   else   // проверяем ответ
```

```
    begin
      Controls.HideControl(btnProv);
      Controls.ShowControl(btnStart);
      Controls.HideControl(txtTime);
      Timer.Pause();
      if (nImage = Int32(Controls.GetTextBoxText(txtOtvet))) then
      begin
        s := 'ПРАВИЛЬНО!   ' + nImage.ToString();
        GraphicsWindow.BrushColor := 'Green';
      end
            else
      begin
        s := 'НЕПРАВИЛЬНО!   ' + nImage.ToString();
        GraphicsWindow.BrushColor := 'Red';
      end;
      GraphicsWindow.FontSize := 48;
      GraphicsWindow.DrawText(90, height / 2 - 24, s);
    end
end;
```

Обратите внимание на то, как элементы управления появляются на экране и удаляются с него!

Тест формируется из загруженных **картинок**, а не из простых геометрических фигур, поэтому для программы можно использовать любые изображения!

```
// СОЗДАЁМ ТЕСТ
private procedure CreateTest();
begin
   GraphicsWindow.BrushColor := BackgroundColor;
   GraphicsWindow.FillRectangle(0, 40, width, height - 40);
   // число картинок на экране:
   nImage := rand.Next(30) + 21;
   // размеры картинок:
   var w: integer := ImageList.GetWidthOfImage(img[1]);
   var h: integer := ImageList.GetHeightOfImage(img[1]);
   // отладка:
   //GraphicsWindow.Title = nImages.ToString();
   var x := 0;
   var y := 0;
   for var i := 0 to nImage - 1 do
   begin
      // выбираем случайную картинку:
      var nImg := rand.Next(N_IMAGE);
      // Выбираем случайные координаты  картинки  -->
      for var n := 0 to 100 - 1 do
```

```
      begin
         // задаём случайные координаты  картинки:
         x := rand.Next(width - w);
         y := rand.Next(height - h - 40) + 40;

         // центр картинки:
         if (GraphicsWindow.GetPixel(x + w div 2,
             y + h div 2).ToString() <>
             BackgroundColor) then
            continue;

          //углы:
         if (GraphicsWindow.GetPixel(x, y).ToString() <>
            BackgroundColor) then
            continue;
         if (GraphicsWindow.GetPixel(x + w, y).ToString() <>
             BackgroundColor) then
            continue;
         if (GraphicsWindow.GetPixel(x, y + h).ToString() <>
             BackgroundColor) then
            continue;
         if (GraphicsWindow.GetPixel(x + w, y + h).ToString() <>
             BackgroundColor) then
            continue;
         break;
      end;
      // рисуем картинку:
      GraphicsWindow.DrawResizedImage(img[nImg], x, y, w, h);
   end
end;
```

Здесь важно позаботиться о том, чтобы картинки не закрывали друг друга. Для этого мы 100 раз пытаемся вывести картинку на свободное место канвы и только потом бросаем монетку наудачу. При небольшом числе картинок этот способ создания теста действует очень хорошо.

Время тестирования мы ограничим *десятью* секундами:

```
// Отсчитываем время тестирования:
private procedure OnTick();
begin
   time += Timer.Interval / 1000;
   Controls.SetTextBoxText(txtTime, 'Время: ' + time.ToString());
   if (time >= 10) then
   begin
      GraphicsWindow.BrushColor := BackgroundColor;
      GraphicsWindow.FillRectangle(0, 40, width, height - 40);
      Timer.Pause();
```

```
      Controls.ShowControl(btnProv);
      Controls.ShowControl(txtOtvet);
   end
end;
```

Когда время истечёт, все монеты исчезнут с экрана, а в окне приложения появится **кнопка** и **текстовое поле** (Рис. 16.2). Тестируемый должен ввести число – соответственно тому, сколько монет он успел насчитать, и нажать кнопку *Проверить*.

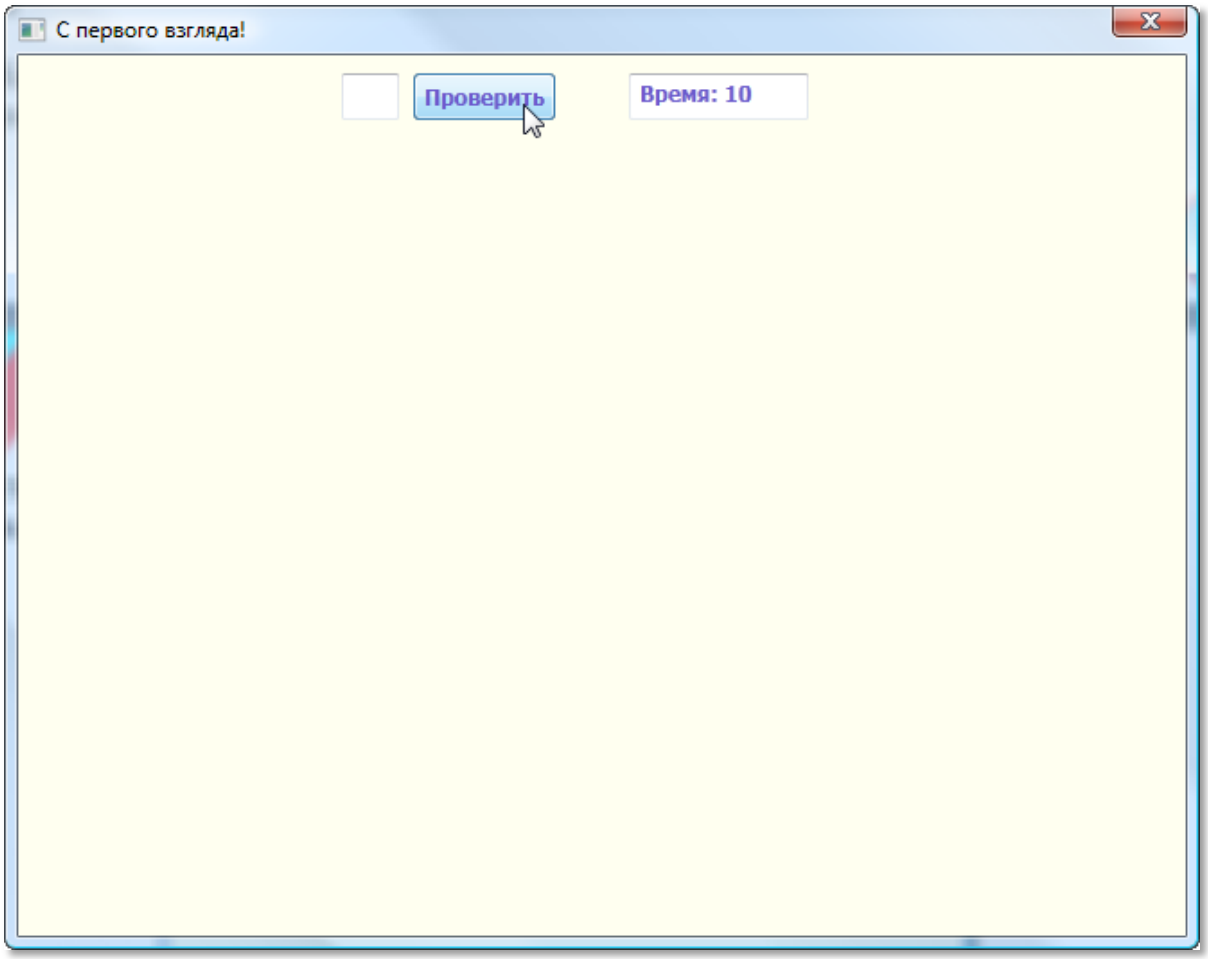


Рис. 16.2. Время вышло!

Как говорится, вердикт не заставит себя долго ждать (Рис. 16.3).

Ну вот, к тесту вы подготовились. Запускайте программу, нажимайте кнопку *Начать тест* и старайтесь уложиться в десять секунд (Рис. 16.4).

Если вам это удастся, вы смело можете идти в банкиры и считать чужие деньги!

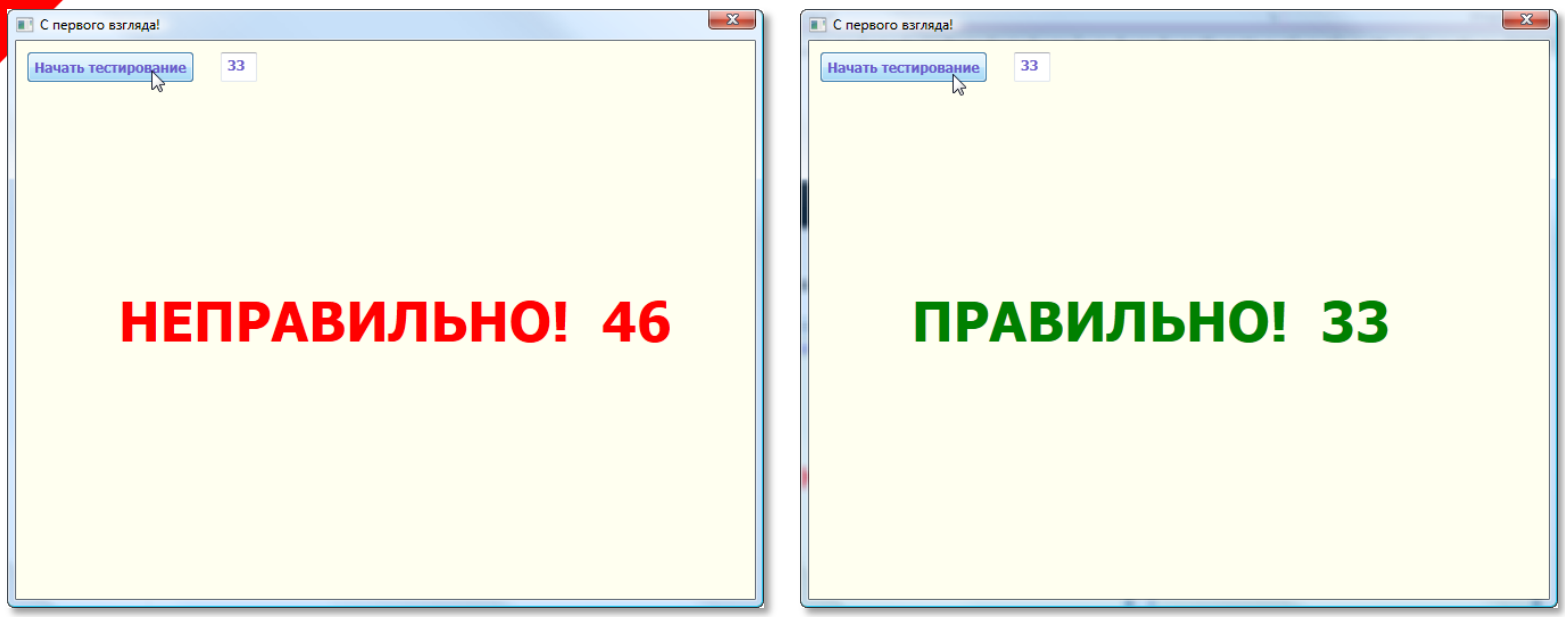


Рис. 16.3. Оттестировались по-разному!

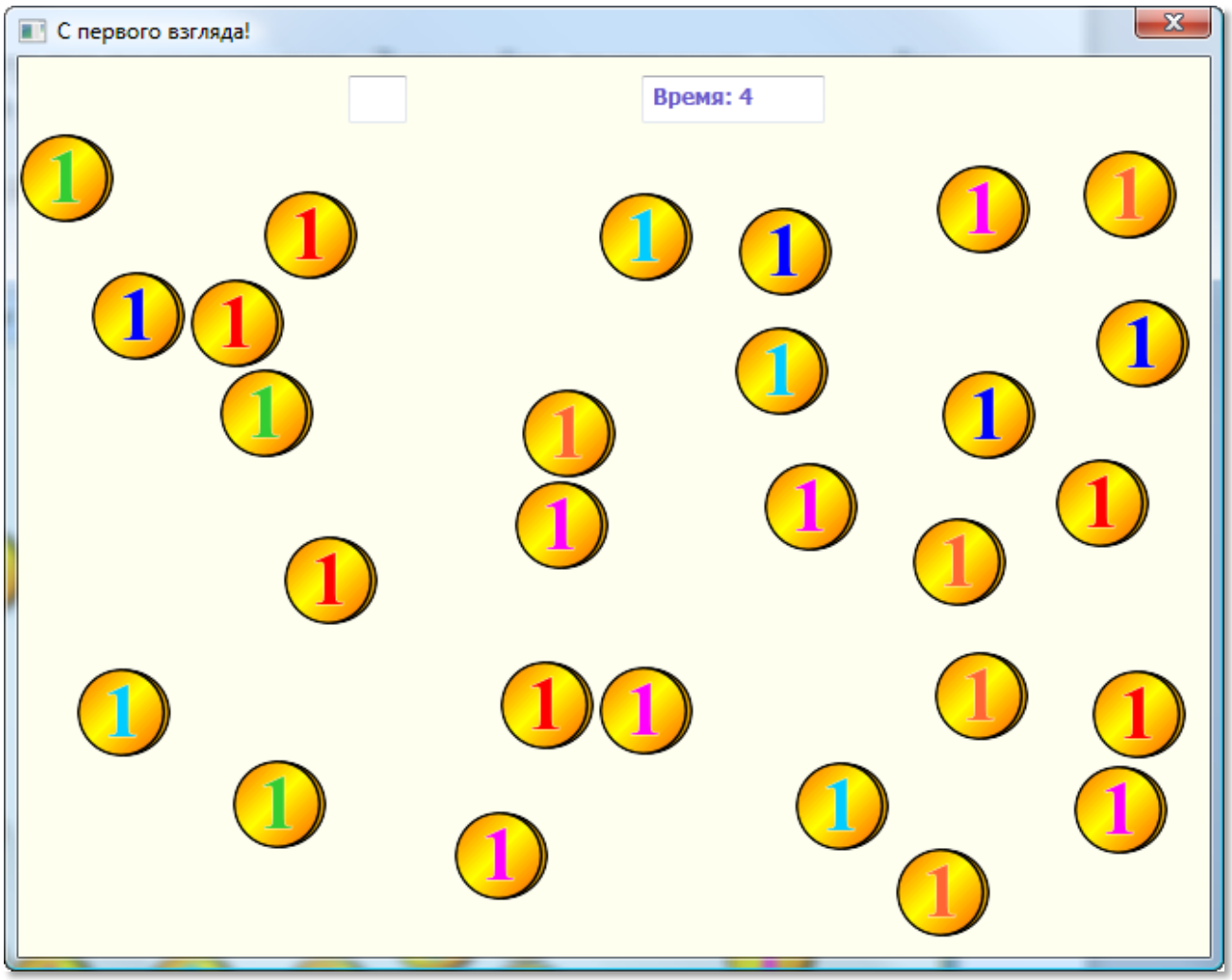


Рис. 16.4. Осталось 6 секунд… Ма-ма!

Глава 17. Занимательная биология

Три проекта, которые мы разработаем в этой главе, и достаточно занимательны, и несомненно полезны, поскольку вы сможете доставить немало приятных минут своим близким, родственникам и друзьям, подвергнув их компьютерному тестированию. Не нужно относиться к его результатам излишне серьёзно, но и полностью отвергать их тоже не стоит.

Проект Контрольное взвешивание, или Веское приложение

Исходный код программы находится в папке **Контрольное взвешивание**.

Хорошего человека должно быть много.

Коварное заблуждение

Сначала мы поможем себе и своим близким приблизиться к идеалу. Пока только в смысле веса.

Некоторые специалисты считают, что **идеальный вес** здорового человека можно определить по формулам:

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,

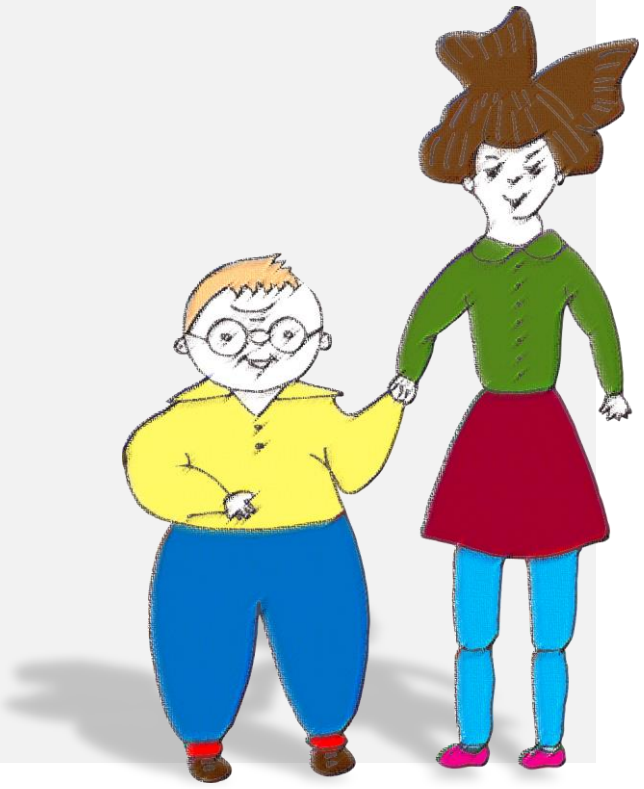
где под загадочными буквами *A* и *B* скрываются **рост** испытуемого (в сантиметрах) и **возраст** (в годах), соответственно.

Согласимся 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 может быть программа, основанная на этих незамысловатых формулах? - Ясно, что главную роль в ней должен играть **метод**, вычисляющий идеальный вес по заданным параметрам – полу «подопытного ин-

дивидуума», его возрасту и росту. С написанием метода проблем не будет, ведь достаточно перевести на компьютерный язык обычные математические выражения.

```
unit DrawUnit;

uses
   Microsoft.SmallBasic.Library, System;

type
   Draw = class
   private
      const GWWIDTH = 590;
      const GWHEIGHT = 390;

      // размеры окна:
      width := GWWIDTH;
      height := GWHEIGHT;
      // координаты центра окна:
      CX := width div 2;
      CY := height div 2;

      // ЭУ
      txtVozrast: string;
      txtRost: string;
      txtVes: string;
      btnM: string;
      btnW: string;
```



Размер **окна** и **картинку** для *фона* вы можете выбрать по своему вкусу и настроению:

```
public procedure Prepare();
begin
   GraphicsWindow.Hide();
   GraphicsWindow.Title := 'Вес';
   GraphicsWindow.Width := GWWIDTH;
   GraphicsWindow.Height := GWHEIGHT;
   GraphicsWindow.Show();
   GraphicsWindow.Left := (Desktop.Width –
                           GraphicsWindow.Width) / 2;
   GraphicsWindow.Top := (Desktop.Height –
                          GraphicsWindow.Height) / 2;
   GraphicsWindow.CanResize := false;

   var path := Environment.CurrentDirectory;
   var background := ImageList.LoadImage(path + '\mw.jpg');
```

```
  GraphicsWindow.DrawImage(background, 0, 0);

  // шрифт:
  GraphicsWindow.BrushColor := 'Red';
  GraphicsWindow.FontBold := 'True';
  GraphicsWindow.FontSize := 16;
```

Нам нужно позаботиться о том, чтобы пользователь мог быстро и удобно передавать в программу необходимые значения. Так как пол может быть либо **мужской**, либо **женский**, то поставим на форму две **кнопки** с буквами *Эм и Жо*, как говаривал Лёлик из комедии *Бриллиантовая рука*. Пользователь выберет ту кнопку, которую пожелает (например, кто-то захочет узнать свой идеальный вес при перемене пола или больше узнать о девочках).

```
// КНОПКИ

// Вычисляем вес для мужчин:
btnM := Controls.AddButton('М'.ToString(), 290, height - 40);
Controls.SetSize(btnM, 48, 32);
// Вычисляем вес для женщин:
btnW := Controls.AddButton('Ж'.ToString(), 10, height - 40);
Controls.SetSize(btnW, 48, 32);
Controls.ButtonClicked += OnClick;
```

Возраст и рост могут принимать большое количество значений, поэтому их мы будем вводить с клавиатуры в два **текстовых поля**:

```
// ТЕКСТОВЫЕ ПОЛЯ

// Возраст:
txtVozrast := Controls.AddTextBox(180, 210);
Controls.SetSize(txtVozrast, 160, 26);
GraphicsWindow.DrawText(350, 214, '<- Возраст в годах');

// Рост:
txtRost := Controls.AddTextBox(180, 248);
Controls.SetSize(txtRost, 160, 26);
GraphicsWindow.DrawText(350, 252, '<- Рост в сантиметрах');
```

Нам потребуется ещё одно **текстовое поле** – для того, чтобы предъявить пользователю его идеальный вес:

```
/ /Идеальный вес:
txtVes := Controls.AddMultiLineTextBox(180, 290);
Controls.SetSize(txtVes, 300, 50);
```

Вся премудрость нашей программы заключена в методе-обработчике нажатия на кнопку **OnClick**:

```
// Печатаем результат тестирования
private procedure OnClick();
begin
   var btn := (string)(Controls.LastClickedButton);
   var s := String.Empty;
   var ves: double;
   var rost: integer;
   var vozrast: integer;

   s := Controls.GetTextBoxText(txtVozrast);
   if (s = String.Empty) then
      vozrast := 0
   else
      vozrast := Int32.Parse(s);

   s := Controls.GetTextBoxText(txtRost);

   if (s = String.Empty) then
      rost := 0
   else
      rost := Int32.Parse(s);

   s := String.Empty;
   if (vozrast < 1) or (vozrast > 120) then
      s := ' Проверьте возраст!'
   else if (rost < 60) or (rost > 240) then
      s := ' Проверьте рост!';
   // ошибка!
   if (s <> String.Empty) then
   begin
      Controls.SetTextBoxText(txtVes, s);
      exit;
   end;

   if (btn = btnM) then
      ves := (3 * rost  - 450 + vozrast) * 0.25 + 45 //-  для мужчин
   else
      ves := (3 * rost  - 450 + vozrast) * 0.225 + 40.4; //- для женщин

   // корректируем отрицательные значения:
   if (ves < 0) then
      ves := 4;
```

```
    GraphicsWindow.BrushColor := 'Blue';
    s := ' Идеальный вес равен ' + System.Math.Round(ves).ToString() +
         ' кг';
    Controls.SetTextBoxText(txtVes, s);
end;
```

Действует он так.

Считываем введённые пользователем данные в строковую переменную **s**, а затем конвертируем строку в числа и сохраняем их в переменных **vozrast** и **rost**, по ходу проверяя биологическую грамотность пользователя или злобный юмор шутника. Если данные выходят за пределы разумения, то расчёты вести глупо, поэтому мы сразу же печатаем последнее предупреждение.

Если пользователь успешно преодолел ввод анкетных данных, то по имени нажатой кнопки мы определяем пол «персонажа», в соответствии с которым и выбираем одну из двух формул.

На всякий случай заменяем отрицательный вес более осмысленным. Это вызвано тем, что формулы для расчёта веса не универсальны, то есть действительны только для разумных сочетаний параметров, поэтому идеальный вес Кощея Бессмертного или дядьки Черномора по ней вычислять нельзя.

Найдя идеальный вес, мы округляем его до целого числа (формулы, естественно, не настолько точны, чтобы рассчитывать ещё и граммы), и выводим в текстовое поле **txtVes** строку с полезной информацией (Рис. 17.1).

Рис. 17.1. Пора худеть?

Проект *Жиропонижающее средство*

Исходный код программы находится в папке **Жиропонижающее средство**.

90-60-90

Фигурка мирового класса

Продолжаем исследования рода человеческого с помощью компьютера. Немного подправив 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 – около 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.

Мы не будем целиком переписывать всю программу *Вес*, а адаптируем её к «новым реалиям». Добавим ещё одно **текстовое окно** – для ввода *действительного* веса пользователя. Заодно нужно подправить и другие *текстовые поля*:

```
// ТЕКСТОВЫЕ ПОЛЯ

// Возраст:
txtVozrast := Controls.AddTextBox(180, 190);
Controls.SetSize(txtVozrast, 160, 26);
GraphicsWindow.DrawText(350, 194, '<- Возраст в годах');

// Рост:
txtRost := Controls.AddTextBox(180, 224);
Controls.SetSize(txtRost, 160, 26);
GraphicsWindow.DrawText(350, 227, '<- Рост в сантиметрах');

// Вес:
txtVes := Controls.AddMultiLineTextBox(180, 259);
Controls.SetSize(txtVes, 160, 28);
GraphicsWindow.DrawText(350, 262, '<- Вес в килограммах');
```

```
// Жирность:
txtZhir := Controls.AddMultiLineTextBox(180, 300);
Controls.SetSize(txtZhir, 300, 50);
```

Тут, конечно, нужно потрудиться и над обработкой данных пользователя с учётом новых формул:

```
// Печатаем результат тестирования
private procedure OnClick();
begin
   var btn := (string)(Controls.LastClickedButton);
   var s := String.Empty;
   var ves: double;
   var rost: integer;
   var vozrast: integer;
   var zhir: double;
   var v: integer;

   s := Controls.GetTextBoxText(txtVozrast);
   if (s = String.Empty) then
      vozrast := 0
   else
      vozrast := Int32.Parse(s);

   s := Controls.GetTextBoxText(txtRost);

   if (s = String.Empty) then
      rost := 0
   else
      rost := Int32.Parse(s);

   s := Controls.GetTextBoxText(txtVes);
   if (s = String.Empty) then
      v := 0
   else
      v := Int32.Parse(s);

   s := String.Empty;
   if (vozrast < 1) or (vozrast > 120) then
      s := ' Проверьте возраст!'
   else if (rost < 60) or (rost > 240) then
      s := ' Проверьте рост!'
   else if (v < 10) or (v > 240) then
      s := ' Проверьте вес!';

   // ошибка!
   if (s <> String.Empty) then
   begin
```

```
      Controls.SetTextBoxText(txtVes, s);
      exit;
   end;

   if (btn = btnM) then
   begin
      ves := (3 * rost  - 450 + vozrast) * 0.25 + 45; //- для мужчин
      zhir := (v - ves) / v * 100 + 15;
   end
   else
   begin
      ves := (3 * rost  - 450 + vozrast) * 0.225 + 40.4; //- для женщин
      zhir := (v - ves) / v * 100 + 22;
   end;

   // корректируем отрицательные значения:
   if (zhir < 0) then
      zhir := 0
   else if (zhir > 100) then
      zhir := 100;

   GraphicsWindow.BrushColor := 'Blue';
   s := ' Жирность равна ' + System.Math.Round(zhir).ToString() + '%';
   Controls.SetTextBoxText(txtZhir, s);
end;
```

Добавьте проверку для введённого веса, а также расчёт жирности по формулам, после чего можете переходить к контрольному взвешиванию (Рис. 17.2).

Рис. 17.2. А девочки-то «жирнее» мальчиков!

Проект *Сколько кожи на человеке?*

Исходный код программы находится в папке **Сколько кожи на человеке**.

Не лезьте из кожи!

Царевна Лягушка

Составим ещё одну познавательную программу по биологии. На этот раз мы вычислим площадь поверхности человеческого тела (то есть кожи). С точки зрения геометрии, фигура человека весьма «причудлива», поэтому не существует точных формул для определения площади тела.

Мы воспользуемся **формулой Бойде**, которая позволяет приближённо вычислить нужную нам величину (не пугайтесь - формула заковыристая):

$$S = (P \times 1000)^{(\lg(1/P)+35{,}75)\,/\,53{,}2} \times H^{0{,}3} : 3118{,}2,$$

где

- **S – площадь** кожи в квадратных метрах
- **P – вес** человека в килограммах
- **H** – его **рост** в сантиметрах

Так как для вычисления площади тела нужно знать *рост* и *вес* человека, то за основу нашей программы мы возьмём предыдущее наше творени:

```
unit DrawUnit;

uses
   Microsoft.SmallBasic.Library, System;

type
   Draw = class
   private
      const GWWIDTH = 590;
      const GWHEIGHT = 390;
      // размеры окна:
      width := GWWIDTH;
```

```
    height := GWHEIGHT;
    // координаты центра окна:
    CX := width div 2;
    CY := height div 2;
    // ЭУ
    txtRost: string;
    txtVes: string;
    txtKozha: string;
    btnKozha: string;
```

Подправим **элементы управления**:

```
public procedure Prepare();
begin
   GraphicsWindow.Hide();
   GraphicsWindow.Title := 'Кожа';
   GraphicsWindow.Width := GWWIDTH;
   GraphicsWindow.Height := GWHEIGHT;
   GraphicsWindow.Show();
   GraphicsWindow.Left := (Desktop.Width –
                           GraphicsWindow.Width) / 2;
   GraphicsWindow.Top := (Desktop.Height –
                          GraphicsWindow.Height) / 2;
   GraphicsWindow.CanResize := false;

   var path := Environment.CurrentDirectory;
   var background := ImageList.LoadImage(path + '\mw.jpg');
   GraphicsWindow.DrawImage(background, 0, 0);

   // шрифт:
   GraphicsWindow.BrushColor := 'Red';
   GraphicsWindow.FontBold := 'True';
   GraphicsWindow.FontSize := 16;

   // КНОПКА
   // Вычисляем площадь кожи:
   btnKozha := Controls.AddButton('Вычислить!', 180, height - 40);
   Controls.SetSize(btnKozha, 120, 32);
   Controls.ButtonClicked += OnClick;


   // ТЕКСТОВЫЕ ПОЛЯ

   // Вес:
   txtVes := Controls.AddTextBox(180, 210);
   Controls.SetSize(txtVes, 160, 26);
   GraphicsWindow.DrawText(350, 214, '<- Вес в килограммах');

   // Рост:
   txtRost := Controls.AddTextBox(180, 248);
```

```
    Controls.SetSize(txtRost, 160, 26);
    GraphicsWindow.DrawText(350, 252, '<- Рост в сантиметрах');

    // Площадь кожи:
    txtKozha := Controls.AddMultiLineTextBox(180, 290);
    Controls.SetSize(txtKozha, 360, 50);

end;
```

Довольно странно, но формула Бойде никак не учитывает половую принадлежность испытуемого, хотя пропорции мужского и женского тела заметно отличаются. Однако доверимся науке и уберём лишнюю кнопку.

Вычислять площадь тела, как обычно, мы будем в процедуре **OnClick**:

```
// Печатаем результат тестирования
private procedure OnClick();
begin
    var str := String.Empty;
    var ves: double;
    var rost: integer;

    str := Controls.GetTextBoxText(txtRost);
    if (str = String.Empty) then
        rost := 0
    else
        rost := Int32.Parse(str);

    str := Controls.GetTextBoxText(txtVes);
    if (str = String.Empty) then
        ves := 0
    else
        ves := Int32.Parse(str);

    if (ves < 10) or (ves > 240) then
        str := ' Проверьте вес!'
    else if (rost < 60) or (rost > 240) then
        str := ' Проверьте рост!';

    str := String.Empty;
    // ошибка!
    if (str <> String.Empty) then
    begin
        Controls.SetTextBoxText(txtKozha, str);
        exit;
    end;
    // вычисляем площадь кожи по формуле Бойде:
    var s := (System.Math.Log10(1.0 / ves) + 35.75) / 53.2;
    s := System.Math.Pow((ves * 1000), s) *
```

```
            System.Math.Pow(rost, 0.3) / 0.31182;
    str := ' Площадь кожи равна ' + System.Math.Round(s).ToString() +
           ' кв. см';
    GraphicsWindow.BrushColor := 'Blue';
    Controls.SetTextBoxText(txtKozha, str);

end;
```

Переводим формулу Бойде на математический диалект языка *паскаль* и выводим на экран площадь тела в квадратных *сантиметрах* (отбрасываем десятичные знаки с помощью метода *Round*). Как говорил Удав: *А в попугаях я гораздо длиннее!* (Рис. 17.3).

Обратите внимание, что буквой *s* мы обозначили площадь тела, как это и принято в математике, поэтому идентификатор строки для вывода результатов пришлось изменить - *str*.

С помощью этой программы вы сможете вычислить площади всех доступных вам тел, к вящему удовольствию их «владельцев» (если они ещё не потеряли вкус к жизни после первых двух экспериментов).

Рис. 17.3. С наукой не поспоришь!

Задания для самостоятельного решения

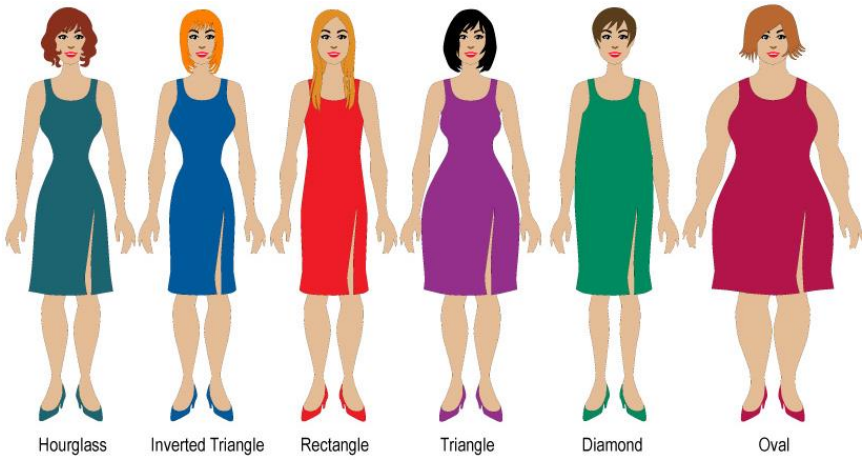
Биологические тесты

1. Перед тем как предлагать программу **Вес** другим людям, очень полезно испытать её на себе. Запустите программу, введите данные, оцените все достоинства и недостатки интерфейса. И – непременно! – постарайтесь её улучшить.

2. Протестируйте программу **Жир** на ком только сможете. Наверняка это доставит и вам, и всем испытуемым массу удовольствия и за-помнится на всю жизнь.

3. Посчитайте, сколько татуировок сможет разместить на себе Владимир Винокур, если средняя площадь татуировки равна 100 квадратным сантиметрам.

Глава 18. Класс *Shapes*



Неподвижные изображения можно легко нарисовать пикселями, линиями, прямоугольниками, эллипсами, треугольниками, картинками и символами. Если же эти объекты *перемещаются* по экрану, то необходимо стирать их в старом положении и рисовать в новом. Если фон одноцветный, то это задача несложная, но на картинном фоне придётся перерисовывать и фон, то есть каждый раз рисовать всю сцену заново. А этот процесс уже довольно долгий и утомительный.

Для таких случаев в библиотеке *SmallBasicLibrary* имеется специальный класс **Shapes** (*Формы, Фигуры*), который самостоятельно занимается всеми производственными процессами при перемещении своих объектов. А это могут быть все те фигуры, о которых мы говорили выше как о неподвижных. Но геометрические фигуры класса *Shapes* могут одновременно иметь и заливку, и контур.

Класс *Shapes* можно представить как набор или коллекцию фигур разных типов. Каждая фигура при её создании и добавлении в коллекцию получает **имя**, которое необходимо указывать при вызове некоторых методов этого класса.

Чтобы добавить в коллекцию **отрезок прямой**, нужно выполнить метод **AddLine**:

```
AddLine(x1, y1, x2, y2 : double) : string;
```

Этот и все подобные методы возвращают *строку* – имя новой фигуры в коллекции. По этому имени можно обратиться к нужной фигуре при её трансформациях.

Как вы уже знаете, для того чтобы нарисовать линию, нужно указать координаты её начальной *(x1,y1)* и конечной точки *(x2,y2).*

Треугольник однозначно определяется координатами своих вершин:

```
AddTriangle (x1 : double; y1 : double;
             x2 : double; y2 : double;
             x3 : double; y3 : double) : string;
```

Все остальные фигуры создаются так, что их левый верхний угол находится в точке с координатами *(0,0),* поэтому их затем нужно переместить в нужное положение методом *Move*.

Метод **AddEllipse** добавляет в коллекцию **эллипс** (круг):

```
AddEllipse(width, height : double) : string;
```

Параметры **width и height** задают размеры эллипса**.**

Метод **AddRectangle** отличается от предыдущего метода только тем, что добавляет в коллекцию **прямоугольник** (квадрат):

```
AddRectangle(width, height : double) : string;
```

Метод **AddText** добавляет в коллекцию **строку** *text*:

```
AddText(text) : string;
```

Текст этой фигуры можно изменить в любой момент с помощью метода **SetText**, при вызове которого следует указать имя фигуры и новый текст:

```
SetText(shapeName, text : string);
```

Цвет заливки фигур определяется текущим значением свойства *GraphicsWindow.BrushColor*, а цвет контура (линий) – значением свойства *GraphicsWindow.PenColor*. Толщина контура (линий) задаётся свойством *GraphicsWindow.PenWidth*.

И последняя фигура, которую можно добавить в коллекцию, – это растровая **картинка**:

```
AddImage(string imageName : string) : string;
```

Здесь под именем картинки **imageName** надо понимать полный путь к файлу.

Любую фигуру из коллекции можно переместить в новое место в окне приложения. Для этого методу **Move** следует указать имя фигуры и её новые координаты:

```
Move(shapeName : string; x, y : double);
```

По умолчанию новые фигуры сразу же видны на экране. Если же какую-либо фигуру нужно на время убрать с глаз долой, то есть сделать *невидимой*, то следует вызвать метод **HideShape**:

```
HideShape(shapeName : string);
```

Чтобы вернуть картинку на экран, выполните метод **ShowShape**:

```
ShowShape(shapeName : string);
```

И наконец, метод **RemoveShape** навсегда удаляет фигуру *shapeName* из коллекции:

```
RemoveShape(shapeName : string);
```

Проект *Класс Shapes*

Исходный код программы находится в папке **Класс Shapes**.

Чтобы лучше изучить фигуры класса *Shapes*, мы напишем новый **проект**.

С помощью кнопок мы будем добавлять к коллекции и показывать на экране все возможные фигуры. Их названия мы сохраним в строковом массиве **name**. Я думаю, что 100 фигур нам будет вполне достаточно для близкого знакомства с ними.

Для картинок нам потребуется отдельный массив **img**, из которого мы будем выбирать случайную картинку:

```
{$apptype windows}

// Класс Shapes


uses
   DrawUnit;

begin
   var draw := new Draw();
   draw.Prepare();

end.

unit DrawUnit;

uses
   Microsoft.SmallBasic.Library, System;

type
   Draw = class
   private
      const GWWIDTH = 640;
      const GWHEIGHT = 480;
      // размеры окна:
      width := GWWIDTH;
      height := GWHEIGHT;
      // координаты центра окна:
      CX := width div 2;
      CY := height div 2;
```

```
    // массив кнопок:
    btn: array of string;

    // число разных картинок:
    const N_IMAGE = 6;
    img := new string[N_IMAGE];

    // макс. число фигур:
    const N_SHAPES = 100;
    name := new string[N_SHAPES];
    // число фигур:
    nShapes := 0;

    rand := new Random();
```

Метод **Prepare** создаёт окно приложения, украшает его кнопками и загружает картинки в массив:

```
public procedure Prepare();
begin
   GraphicsWindow.Hide();
   GraphicsWindow.Title := 'Класс Shapes';
   GraphicsWindow.Width := GWWIDTH;
   GraphicsWindow.Height := GWHEIGHT;
   GraphicsWindow.Show();
   GraphicsWindow.Left := (Desktop.Width –
                           GraphicsWindow.Width) / 2;
   GraphicsWindow.Top := (Desktop.Height –
                          GraphicsWindow.Height) / 2;
   GraphicsWindow.CanResize := false;
   GraphicsWindow.BackgroundColor := 'Black';

   // КНОПКИ

   var y := 10;
   var dy := 40;
   var x := width - 80;
   var n := 0;
   btn := new string[6];

   btn[n] := Controls.AddButton('Эллипс', x, y + dy * (n));
   Controls.SetSize(btn[n], 80, 30);
   n += 1;
   btn[n] := Controls.AddButton('Отрезок', x, y + dy * (n));
   Controls.SetSize(btn[n], 80, 30);
   n += 1;
   btn[n] := Controls.AddButton('Прямоуг.', x, y + dy * (n));
   Controls.SetSize(btn[n], 80, 30);
```

```
    n += 1;
    btn[n] := Controls.AddButton('Треугольн.', x, y + dy * (n));
    Controls.SetSize(btn[n], 80, 30);
    n += 1;
    btn[n] := Controls.AddButton('Текст', x, y + dy * (n));
    Controls.SetSize(btn[n], 80, 30);
    n += 1;
    btn[n] := Controls.AddButton('Картинка', x, y + dy * (n));
    Controls.SetSize(btn[n], 80, 30);

    Controls.ButtonClicked += OnClick;

    var path := Environment.CurrentDirectory;
    img[0] := ImageList.LoadImage(path + '\1.png');
    img[1] := ImageList.LoadImage(path + '\2.png');
    img[2] := ImageList.LoadImage(path + '\3.png');
    img[3] := ImageList.LoadImage(path + '\4.png');
    img[4] := ImageList.LoadImage(path + '\5.png');
    img[5] := ImageList.LoadImage(path + '\6.png');
end;
```

В целом **интерфейс** приложения получился простым и понятным (Рис. 18.1).

Пора жать на кнопки, а для этого вам нужно позаботиться о наполнении метода **OnClick** таким кодом:

```
private procedure OnClick();
begin
    if (nShapes = N_SHAPES) then
        exit;

    // общее число фигур:
    nShapes += 1;

    var button := (string)(Controls.LastClickedButton);
    if (button = btn[0]) then
        AddEllipse()
    else if (button = btn[1]) then
        AddLine()
    else if (button = btn[2]) then
        AddRectangle()
    else if (button = btn[3]) then
        AddTriangle()
    else if (button = btn[4]) then
        AddText()
    else if (button = btn[5]) then
        AddImage();
```

```
end;
```

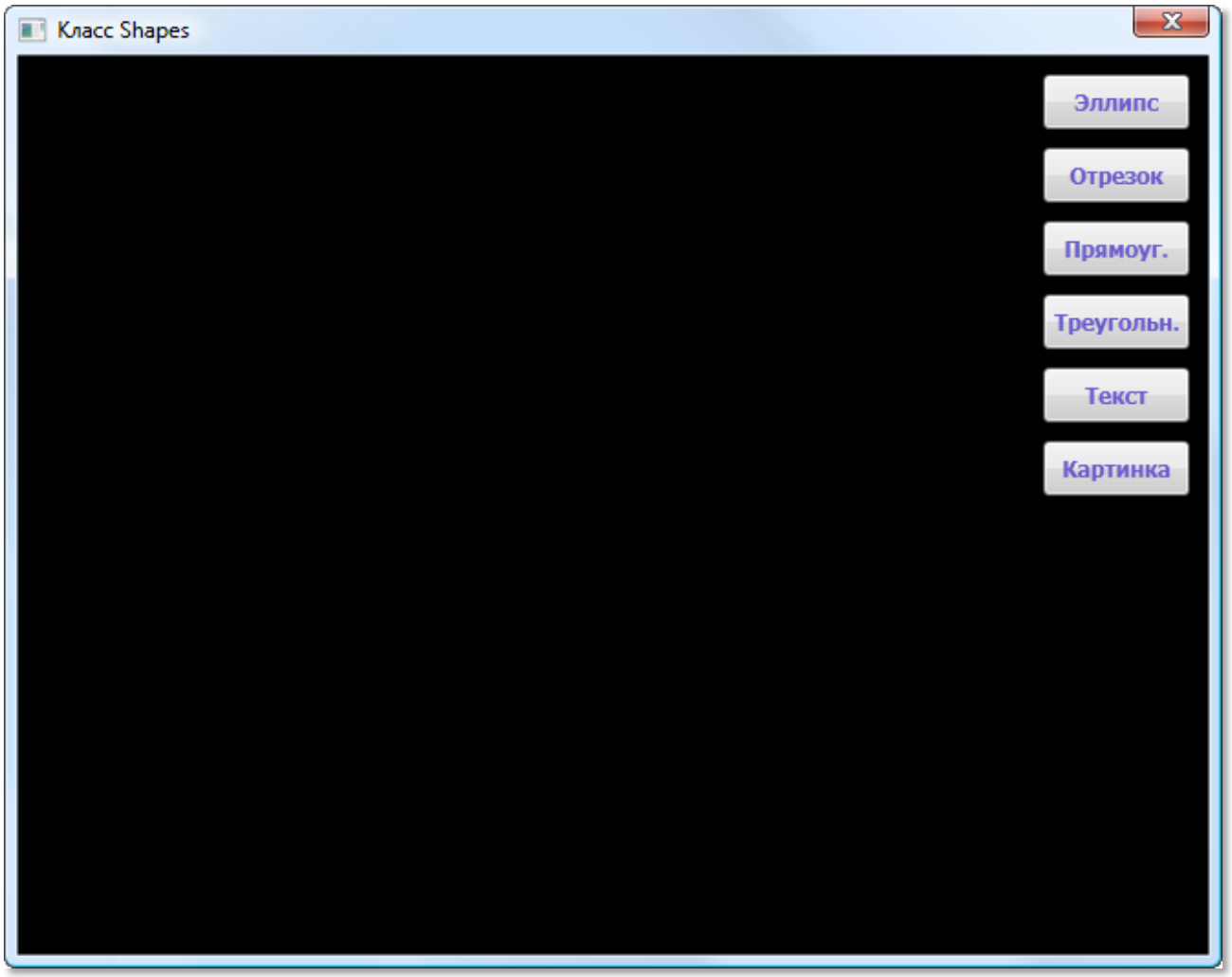


Рис. 18.1. Чистое поле

Здесь мы не забываем считать фигуры и контролировать наполнение массива *name*, чтобы не переполнить его чашу терпения.

Поскольку простого нажатия на кнопку недостаточно для появления фигуры на экране, то для каждой «коллекционерки» следует написать отдельный метод.

Чтобы фигуры одного типа отличались друг от друга, мы задаём им случайные размеры, координаты и внешний облик, то есть цвет закраски и контура:

```
// Добавляем случайный эллипс
private procedure AddEllipse();
begin
   // ширина:
```

```
   var w := rand.Next(40, 160);
   // высота:
   var h := rand.Next(40, 160);

   // цвет заливки:
   var fill := GraphicsWindow.GetRandomColor();
   GraphicsWindow.BrushColor := fill;
   // цвет контура:
   var draw := GraphicsWindow.GetRandomColor();
   GraphicsWindow.PenColor := draw;
   // толщина контура:
   GraphicsWindow.PenWidth := 2;

   name[nShapes] := Shapes.AddEllipse(w, h);

   // координаты:
   var x := rand.Next(width - 160);
   var y := rand.Next(height - 160);
   Shapes.Move(name[nShapes], x, y);
end;
```

Новорождённые эллипсы всегда получают координаты по умолчанию – *(0,0)*. Если не позаботиться об их расселении, то они все кучно сгрудятся в верхнем левом углу канвы и напрочь затмят друг друга. Это нам не будет интересно…

Понажимав кнопку **Эллипс**, вы получите на экране дружную коллекцию этих яйцеобразных фигур (Рис. 18.2).

Эти эллипсы почти не отличаются от своих статичных родственников, не считая одновременного наличия контура и заливки. Но они не просто впечатаны навечно в канву окна приложения, а могут перемещаться и выделывать кульбиты и прочие кувырки, с которыми вы вскоре познакомитесь.

Практически аналогично мы добавляем к коллекции и **прямоугольные** фигуры:

```
// Добавляем случайный прямоугольник
private procedure AddRectangle();
begin
   // ширина:
   var w := rand.Next(40, 160);
   // высота:
   var h := rand.Next(40, 160);
```

```
    // цвет заливки:
    var fill := GraphicsWindow.GetRandomColor();
    GraphicsWindow.BrushColor := fill;
    // цвет контура:
    var draw := GraphicsWindow.GetRandomColor();
    GraphicsWindow.PenColor := draw;
    // толщина контура:
    GraphicsWindow.PenWidth := 2;

    name[nShapes] := Shapes.AddRectangle(w, h);

    // координаты:
    var x := rand.Next(width - 160);
    var y := rand.Next(height - 160);
    Shapes.Move(name[nShapes], x, y);
end;
```

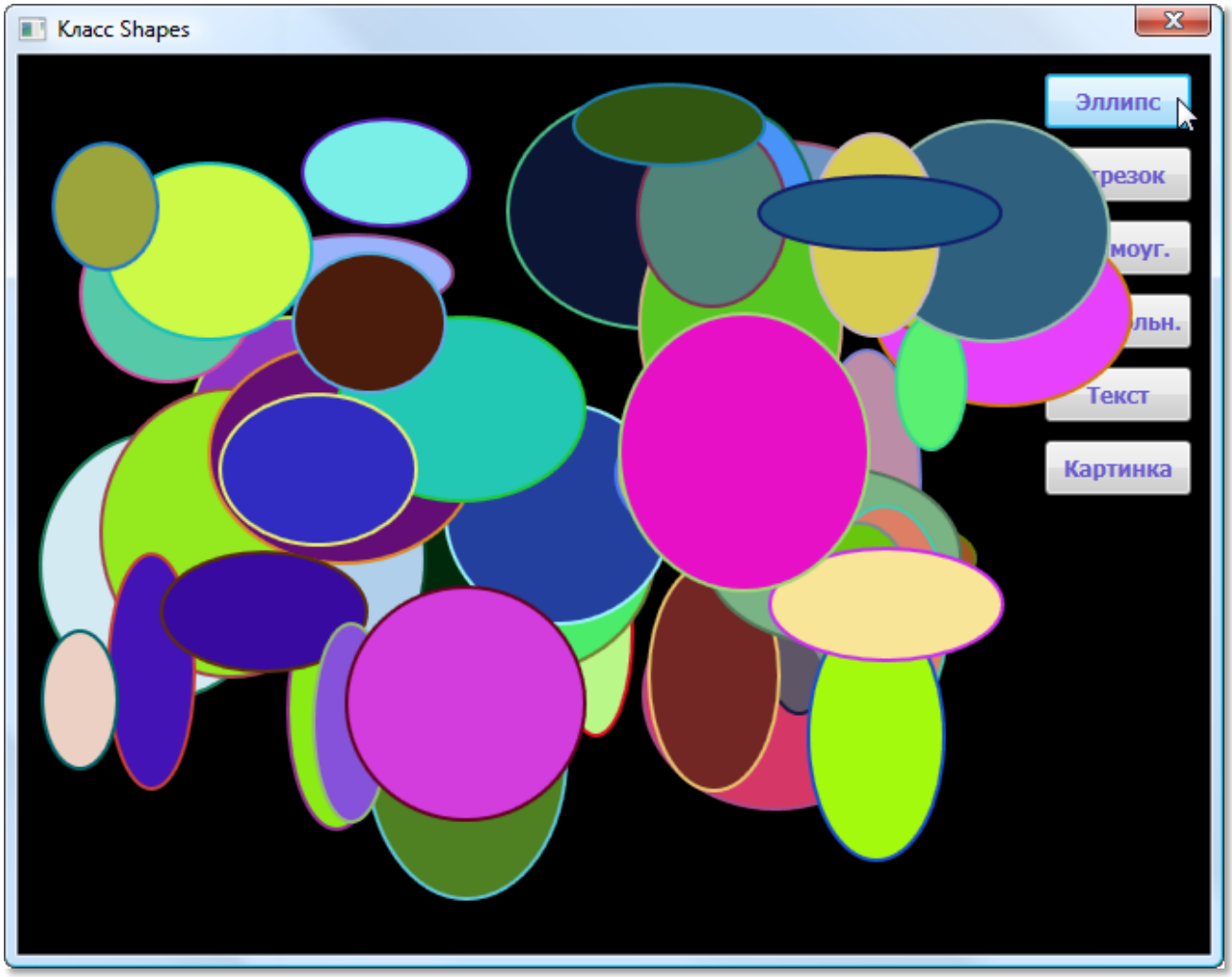


Рис. 18.2. Эллипсизация канвы прошла успешно!

Линия и **треугольник** сразу получают свои координаты, поэтому перемещать их в новое положение уже не нужно:

```
// Добавляем случайный отрезок
private procedure AddLine();
begin
   // цвет линии:
   var draw := GraphicsWindow.GetRandomColor();
   GraphicsWindow.PenColor := draw;

   // толщина линии:
   GraphicsWindow.PenWidth := rand.Next(1, 8);

   // координаты:
   var x1 := rand.Next(width - 160);
   var y1 := rand.Next(height - 160);
   var x2 := rand.Next(width - 160);
   var y2 := rand.Next(height - 160);

   name[nShapes] := Shapes.AddLine(x1, y1, x2, y2);
end;

// Добавляем случайный треугольник
private procedure AddTriangle();
begin
   // цвет заливки:
   var fill := GraphicsWindow.GetRandomColor();
   GraphicsWindow.BrushColor := fill;
   // цвет контура:
   var draw := GraphicsWindow.GetRandomColor();
   GraphicsWindow.PenColor := draw;
   // толщина контура:
   GraphicsWindow.PenWidth := 2;

   // координаты:
   var x1 := rand.Next(width - 160);
   var y1 := rand.Next(height - 160);
   var x2 := rand.Next(width - 160);
   var y2 := rand.Next(height - 160);
   var x3 := rand.Next(width - 160);
   var y3 := rand.Next(height - 160);

   name[nShapes] := Shapes.AddTriangle(x1, y1, x2, y2, x3, y3);
end;
```

Строку уже после её создания можно изменить с помощью метода *SetText*. У нас каждая новая строка будет показывать число всех созданных нами фигур:

```
// Добавляем строку
private procedure AddText();
```

```
begin
   // цвет букв:
   var fill := GraphicsWindow.GetRandomColor();
   GraphicsWindow.BrushColor := fill;

   GraphicsWindow.FontName := 'Arial';
   GraphicsWindow.FontBold :=  true;
   GraphicsWindow.FontSize := rand.Next(12, 49);
   name[nShapes] := Shapes.AddText('Text');
   Shapes.SetText(name[nShapes], 'Всего фигур: ' +
                  nShapes.ToString());

   // координаты:
   var x := rand.Next(160);
   var y := rand.Next(height - 160);
   Shapes.Move(name[nShapes], x, y);
end;
```

По умолчанию **картинка** имеет оригинальные размеры, так что нам остаётся только транспортировать её в случайное место окна приложения:

```
// Добавляем случайную картинку
private procedure AddImage();
begin
   var n := rand.Next(N_IMAGE);
   var s := img[n];
   name[nShapes] := Shapes.AddImage(s);

   // координаты:
   var x := rand.Next(width - 160);
   var y := rand.Next(height - 60);
   Shapes.Move(name[nShapes], x, y);
end;
```

Пройдясь мышкой по кнопкам, мы породим все возможные коллекционные фигуры в полном изобилии и достатке (Рис. 18.3).

Но давайте добавим *оживляжа* нашей мизансцене. Для этого установим **таймер**

```
//таймер:
Timer.Interval := 100;
Timer.Tick += OnTick;
```

который будет случайным образом изменять видимость-невидимость наших фигур (Рис. 18.4):

```
private procedure OnTick();
begin
   foreach var s in name do
   begin
      if (rand.Next(2) = 0) then
         Shapes.HideShape(s)
      else
         Shapes.ShowShape(s);
   end
end;
```

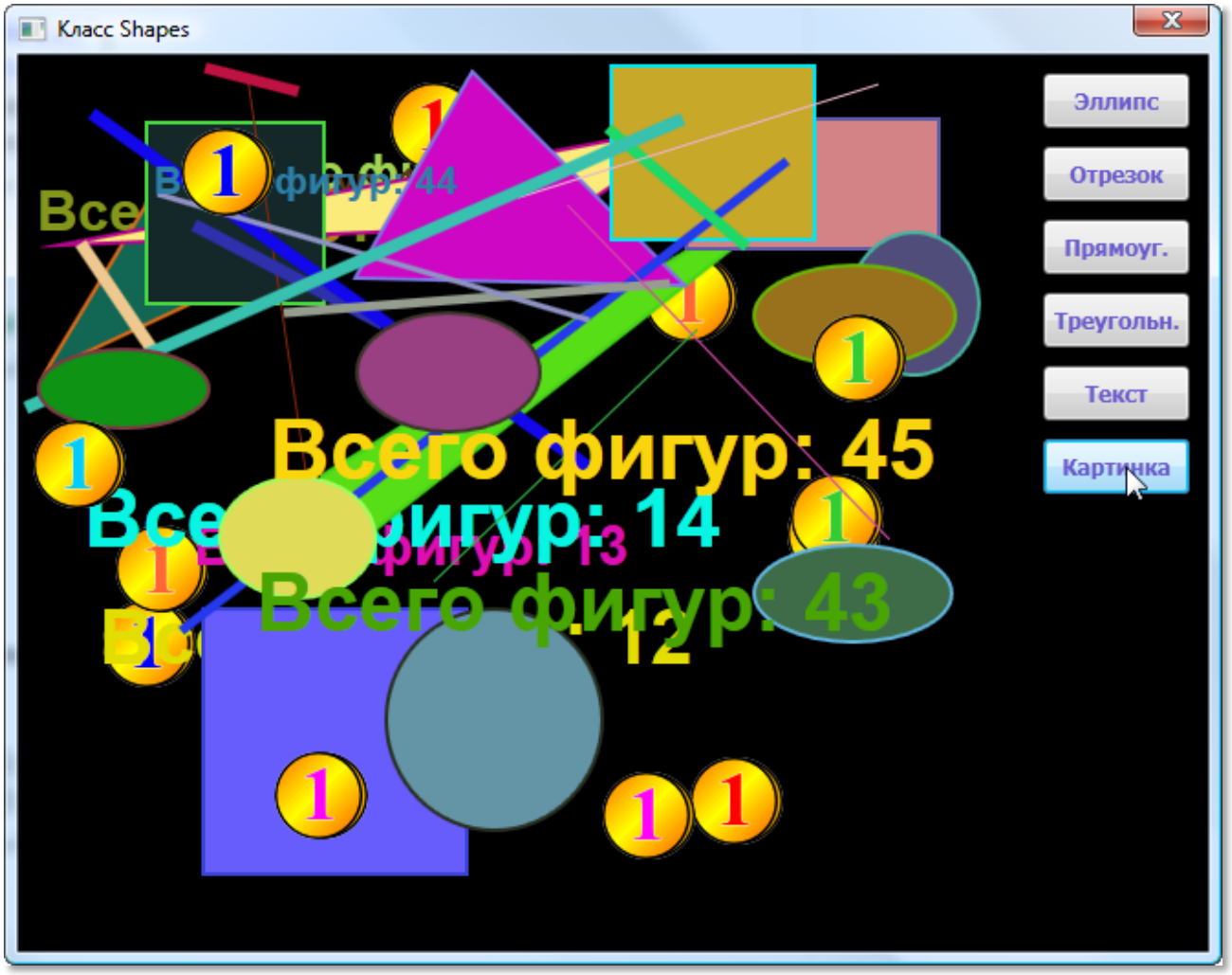
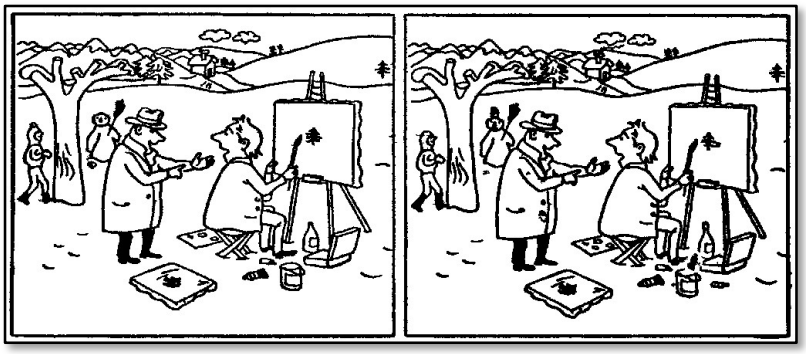


Рис. 18.3. Фигурный подиум

Со статичными фигурами вы бы имели в этом случае головную боль средней тяжести!



Подмигивание фигур трудно передать на картинке. Но если делать снимки через некоторое время, то можно получить вполне интересные задания типа *найдите столько-то отличий на этих картинках*.

Рис. 18.4. Почувствуйте разницу!

Проект *Броуновское движение*

Исходный код программы находится в папке **Броуновское движение**.

Чем ещё могут порадовать и удивить нас фигуры класса **Shapes**? Кроме *мгновенного* перемещения из одной точки в другую, они могут делать это *плавно* и степенно – и даже без нашего участия и таймерного вмешательства.

Метод **Animate** «анимирует» перемещение указанной фигуры *shapeName* в новое положение с координатами *(x,y)* за время *duration*:

```
Animate (shapeName : string;
         x : double; y : double;
         duration : double);
```

С помощью этого метода мы легко смоделируем процесс броуновского, беспорядочного перемещения молекул:

```
unit DrawUnit;

uses
   Microsoft.SmallBasic.Library, System;

type
   Draw = class
   private
      const GWWIDTH = 640;
      const GWHEIGHT = 480;
      // размеры окна:
      width := GWWIDTH;
      height := GWHEIGHT;
      // координаты центра окна:
      CX := width div 2;
      CY := height div 2;

      // число молекул:
      const N_BALL = 100;
      ball := new string[N_BALL];
```

```
      // диаметр молекул:
      const d = 10;

      rand := new Random();
```

В методе **Prepare** разворачивается настоящая молекулярная кухня, в результате чего мы получаем коллекцию из сотни разноцветных молекул:

```
public procedure Prepare();
begin
   GraphicsWindow.Hide();
   GraphicsWindow.Title := 'Броуновское движение';
   GraphicsWindow.Width := GWWIDTH;
   GraphicsWindow.Height := GWHEIGHT;
   GraphicsWindow.Show();
   GraphicsWindow.Left := (Desktop.Width –
                           GraphicsWindow.Width) / 2;
   GraphicsWindow.Top := (Desktop.Height –
                          GraphicsWindow.Height) / 2;
   GraphicsWindow.CanResize := false;
   GraphicsWindow.BackgroundColor := 'Black';

   // создаём молекулы:
   for var i := 0 to N_BALL - 1 do
   begin
      var fill := GraphicsWindow.GetRandomColor();
      GraphicsWindow.BrushColor := fill;
      ball[i] := Shapes.AddEllipse(d, d);
      Shapes.HideShape(ball[i]);
      // координаты:
      var x := rand.Next(d, width - d);
      var y := rand.Next(d, height - d);
      Shapes.Move(ball[i], x, y);
      Shapes.ShowShape(ball[i]);
   end;
   // таймер:
   Timer.Interval := 4000;
   Timer.Tick += OnTick;
end;
```

Все они одного размера, но вы уже хорошо знаете, как можно задать им случайные габариты в заданных пределах.

Таймер в данном случае нужен не для анимации фигур, а для изменения направления и скорости их движения:

```
private procedure OnTick();
begin
   foreach var s in ball do
   begin
      var x := rand.Next(d, width - d);
      var y := rand.Next(d, height - d);
      Shapes.Animate(s, x, y, 4000);
   end
end;
```

Когда молекулы достигнут конечной точки своего очередного перемещения (а это случится через 4 секунды), они получают новый адрес и анимируются в указанном направлении. И этот коловорот и коловращение фигур продолжается вечно и бесконечно...

Итогом нашей незатейливой анимации служит весьма занятная пульсация молекул, сопровождаемая их сбеганием и разбеганием (Рис. 18.5).

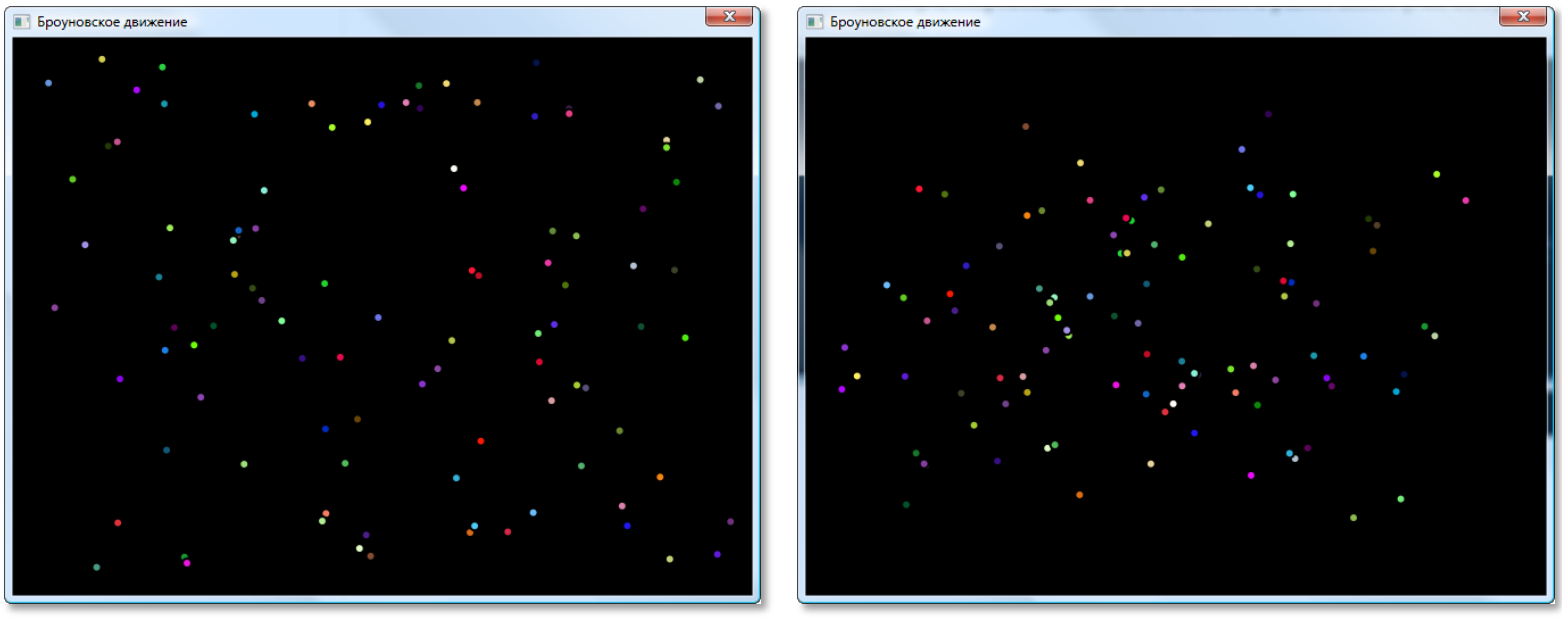


Рис. 18.5. Снимки броуновского процесса

Проект *Трансформации фигур*

Исходный код программы находится в папке **Трансформации фигур**.

Один вид трансформаций мы уже рассмотрели – это **перемещение**, которое осуществляется с помощью метода **Move**. Класс *Shape* имеет также 2 дополнительных метода, которые возвращают текущие **координаты** фигуры (её верхнего левого угла).

Метод **GetLeft** возвращает *горизонтальную* координату, а метод **GetTop** – *вертикальную*:

```
GetLeft(shapeName) : double;
```

```
GetTop(shapeName : string) : double;
```

За **масштабирование** фигур, то есть за изменение их *размера* отвечает метод **Zoom**, а за *поворот* – метод **Rotate**.

```
Zoom(shapeName : string; scaleX, scaleY : double);
```

Коэффициенты масштабирования **scaleX** и **scaleY** изменяются в диапазоне 0.1..20, то есть уменьшить фигуру допускается не более чем в 10 раз относительно исходного размера, а увеличить – не более чем в 20 раз.

Метод **Rotate** поворачивает указанную фигуру **shapeName** вокруг её центра на **angle** градусов относительно исходного положения:

```
Rotate(shapeName : string;  angle : double);
```

Если угол положительный, то фигура поворачивается по часовой стрелке, если отрицательный – против.

Давайте дополним проект *Класс Shapes* новыми возможностями, которые продемонстрируют нам указанные трансформации фигур.

Пусть при каждом срабатывании таймера все фигуры **масштабируются** по обеим осям случайным образом:

```
private procedure OnTick();
begin
   // масштабирование:
   foreach var s in name do
   begin
      var zoomX := rand.NextDouble() * 2;
      var zoomY := rand.NextDouble() * 2;
      if (rand.Next(10) = 0) then
         Shapes.Zoom(s, zoomX, zoomY);
   end
end;
```

Вы можете трансформировать фигуры любого типа, а я ограничился только картинками и строками – исключительно для того, чтобы не загромождать экран (Рис. 18.6).

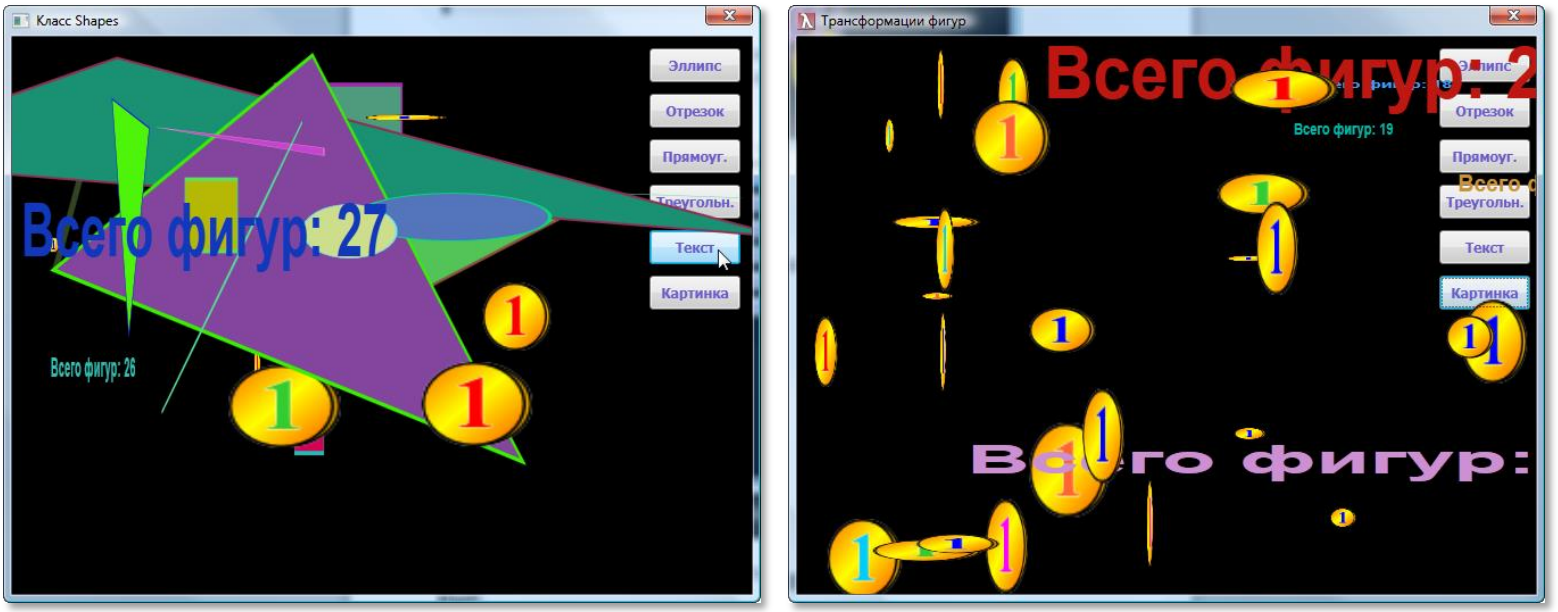


Рис. 18.6. Плющим и растягиваем!

Теперь закомментируйте масштабирующие строки и допишите **поворотные**:

```
private procedure OnTick();
begin
   // масштабирование:
   //         foreach var s in name do
   //         begin
```

```
   //            var zoomX := rand.NextDouble() * 2;
   //            var zoomY := rand.NextDouble() * 2;
    //            if (rand.Next(10) = 0) then
   //              Shapes.Zoom(s, zoomX, zoomY);
   //         end

   // поворот:
   foreach var s in name do
   begin
      var angle := rand.Next(180);
      if (rand.Next(10) = 0) then
         Shapes.Rotate(s, angle);
   end;
end;
```

На Рис. 18.7 хорошо видно кручение и верчение картинок и строк.

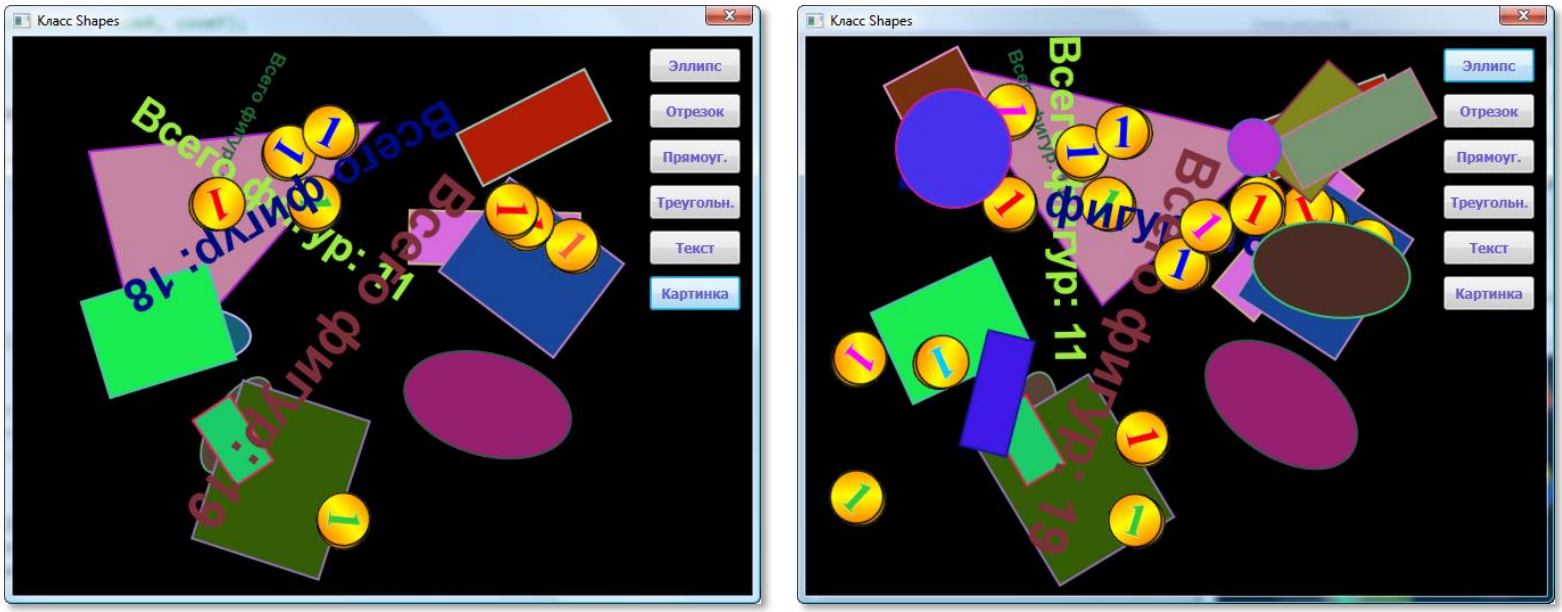


Рис. 18.7. Закрутили!

Метод **SetOpacity** изменяет **прозрачность** фигур. Параметр **level** изменяется от 0 до 100:

```
SetOpacity(shapeName : string; level : double);
```

Значению **0** соответствует полная прозрачность фигуры, а значению **100** – полная её непрозрачность.

Опять в методе **OnTick** случайно изменяем *прозрачность* фигур в указанном диапазоне:

```
private procedure OnTick();
begin
   . . .

   // прозрачность:
   foreach var s in name do
   begin
      var alpha := rand.Next(101);
      if (rand.Next(10) = 0) then
         Shapes.SetOpacity(s, alpha);
  end;
end;
```

Сквозь полупрозрачные фигуры можно увидеть те фигуры, которые находятся «ниже» (Рис. 18.8).

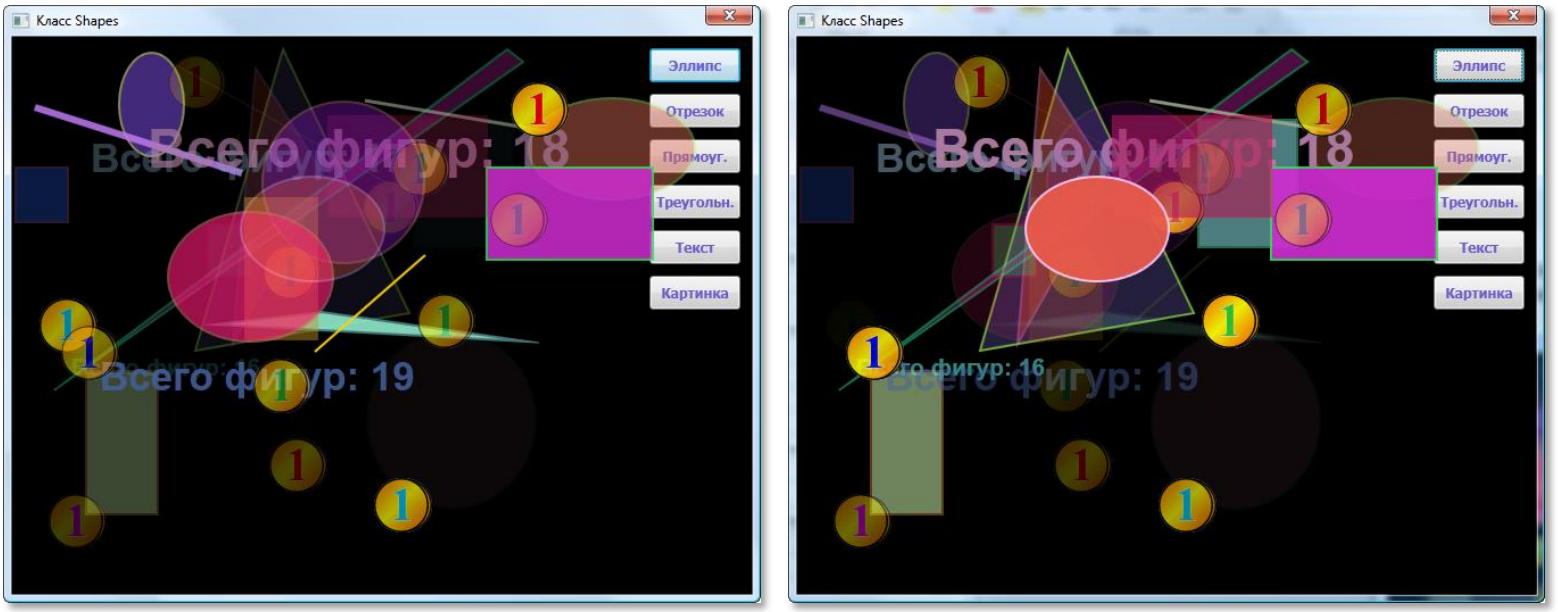


Рис. 18.8. Сквозная видимость

Последний метод класса *Shape* - **GetOpacity** – возвращает текущее значение *прозрачности* (точнее сказать, непрозрачности) фигуры в процентах, то есть от 0 до 100:

```
GetOpacity(shapeName : string) : double;
```

Глава 19. Тараканьи бега по методу Монте-Карло

Вдруг из подворотни
Страшный великан,
Рыжий и усатый
Та-ра-кан!
Таракан, Таракан, Тараканище!

Корней Чуковский, *Тараканище*

Тараканьи бега – любимое развлечение *аристократов и дегенератов*, как выразился бы Лёлик из комедии *Бриллиантовая рука,* - проводятся так.

Берут в нужном количестве больших - длиной до 10 сантиметров - тараканов с острова Мадагаскар (Рис. 19.1). Конечно, тараканам всё равно куда бежать, поэтому их помещают в направляющие желобки, избавляющие их от тяжкого бремени выбора пути. Длина забега 1,5 – 2 метра. Зрители и прочие ротозеи делают ставки на полюбившегося им «спортсмена», после чего все тараканы весело бегут к финишу, подбадриваемые дружеским свистом публики.

Рис. 19.1. И вправду тараканище!

Если вам не довелось воочию увидеть это жуткое зрелище, то вы можете прочитать его художественное описание в пьесе *Бег* Михаила Булгакова (Рис. 19.2) и в романе Алексея Толстого *Похождения Невзорова, или Ибикус*.

Рис. 19.2. Призовые гонки тараканов в фильме *Бег* (1970)

Более изощрённый вариант тараканьих бегов показан в фильме *Китайский сервиз*, где тараканы по условному сигналу подносили шулеру карты, приклеенные к спине (Рис. 19.3).

Рис. 19.3. Пляшущие тараканы из фильма *Китайский сервиз*

Наверное, вам встречались и компьютерные игры, симулирующие тараканьи бега (или лошадиные, крысиные, собачьи и поросячьи). Мы сейчас тоже напишем программу, но не игровую, а вполне серьёзную, реализующую простую **компьютерную модель** вообще любых подобных состязаний.

Проект *Тараканьи бега*

Исходный код программы находится в папке **Тараканьи бега**.

Чтобы не отвлекаться на анатомические подробности разных видов «спортсменов», будем считать их *эллипсами*. Эта фигура вполне адекватно заменяет тушки бегунов. Их количество можно принять равным шести, но при желании это число можно изменить в любую сторону.

Перед стартом все участники выстраиваются в одну линию, после чего следует сигнал к началу гонок. Тараканы бегут вперёд, пока первый из них не пересечёт финишную черту. Он объявляется победителем, а все игроки, сделавшие на него ставки, получают призовые деньги.

Если бы в забегах всегда побеждал один и тот же таракан, то состязание потеряло бы всякий смысл, поскольку все зрители ставили бы на него - без всякой надежды получить выигрыш больше той суммы, что они поставили. Значит, в гонках тараканы должны бежать не с одинаковой скоростью, более того, они должны совершать рывки или отставать. На результаты гонок могут влиять и другие факторы, например, физическое состояние участников, питание и настроение.

Поскольку учесть все эти факторы невозможно, то в нашей модели мы будем считать, что скорость бега каждого таракана изменяется по дистанции **случайным** образом. Кроме того, все остальные факторы мы объединим в одну группу и обозначим их как текущее *здоровье* таракана.

Таким образом, наша компьютерная модель будет описывать случайные процессы. Неудивительно, что для этого используют **метод Монте-Карло**. Как вы знаете, в этом городе находится самый известный в мире игорный дом, а рулетку вполне можно считать хорошим генератором случайных чисел. Действительно, шарик в рулетке случайно «выбирает» одно из 37 чисел (от 1 до 36 и 0 - зеро) на колесе.

> Важно, чтобы числа выпадали именно *случайно*, без всякой закономерности, ибо всякая закономерность может быть подмечена внимательным игроком. Такой случай описал Джек Лондон в рассказе *Смок и Малыш*, где героям

удалось выиграть неплохую сумму денег на рулетке, которая стояла рядом с печкой и потому со временем рассохлась. Колесо крутилось неравномерно, и одни номера выпадали чаще других.

На самом деле эта история - не вымысел автора. Английский инженер Джозеф Джаггерс в 1973 году с помощниками изучил поведение рулеток одного из казино в Монте-Карло и определил, что на одном из них числа выпадают неравномерно. За четыре дня он выиграл несколько сотен тысяч долларов, после чего руководство казино догадалось проверить колесо этой рулетки. С тех пор «система Джаггерса» не действует, так как во всех казино строго следят за исправностью рулеток и время от времени опять же случайным образом меняют местами колёса.

В романе Фёдора Михайловича Достоевского *Игрок* описаны все перипетии игры на рулетке.

Метод Монте-Карло разработали в 1948 году американские математики Джон Нейман и Станислав Улам. Кстати говоря, и раньше некоторые задачи решали с помощью случайных выборок, но до появления электронно-вычислительных машин применение этого метода было очень трудоёмким.

Теперь же в любом языке программирования имеется процедура, генерирующая случайные числа (точнее, *псевдослучайные*, поскольку они вычисляются по формуле). Есть такая процедура и в *паскале* – это метод *Next* класса *Random*.

Теперь вы знаете достаточно для того, чтобы приступить к написанию программы.

Как мы и договорились, в забегах будут принимать участие 6 тараканов, которых мы раскрасим в разные ***цвета***:

```
unit DrawUnit;

uses
   Microsoft.SmallBasic.Library, System, System.Threading;

type
   Draw = class
   private
      const GWWIDTH = 640;
      const GWHEIGHT = 480;
```

```
    // размеры окна:
    width := GWWIDTH;
    height := GWHEIGHT;
    // координаты центра окна:
    CX := width div 2;
    CY := height div 2;

    // цвет фона:
    const BackgroundColor = '#2F4F4F';
    // число тараканов:
    const N_TARAKAN = 6;
    // цвета тараканов:
    color := new string[] ('Red',
                           'Yellow',
                           'Blue',
                           'Green',
                           'SandyBrown',
                           'LavenderBlush'
                          );
```

Также нам понадобятся **переменные** для хранения координат тараканов и их здоровья на момент старта:

```
// тараканы:
tar := new string[N_TARAKAN];
// координаты тараканов:
x := new double[N_TARAKAN];
y := new double[N_TARAKAN];
// здоровье тараканов:
zdor := new double[N_TARAKAN];
// время старта:
startTime: double;
// время:
time: double;
// флажки начала и окончания забега:
flgStart: boolean;
flgFinish: boolean;

// координаты стартовой и финишной позиции:
xStart := 10;
yStart := 40;
xFinish: integer;

// расстояние между дорожками по вертикали:
dy := 32;
// размеры тараканов:
wTar := 32;
hTar := 16;
```

```
// ЭУ
txtTime: string;
txtWin: string;
btnStart: string;

rand := new Random();
```

Размеры **окна** вы можете сделать и больше, чтобы продлить удовольствие:

```
public procedure Prepare();
begin
   GraphicsWindow.Hide();
   GraphicsWindow.Title := 'Тараканьи бега';
   GraphicsWindow.Width := GWWIDTH;
   GraphicsWindow.Height := GWHEIGHT;
   GraphicsWindow.Show();
   GraphicsWindow.Left := (Desktop.Width –
                           GraphicsWindow.Width) / 2;
   GraphicsWindow.Top := (Desktop.Height –
                          GraphicsWindow.Height) / 2;
   GraphicsWindow.CanResize := false;
   GraphicsWindow.BackgroundColor := BackgroundColor;
```

Дополним интерфейс программы **элементами управления**. Каждые 10 миллисекунд (0,01 с) будет срабатывать **таймер**:

```
// Таймер:
Timer.Interval := 10;
Timer.Pause();
Timer.Tick += OnTick;
```

Время, прошедшее со старта, мы будем выводить в текстовое поле *txtTime* в методе-обработчике таймера **OnTick**:

```
// Отсчитываем время гонок
private procedure OnTick();
begin
   var t := (double)(Clock.ElapsedMilliseconds) - startTime;
   var s := 'Время: ' + System.Math.Round(t / 1000,
                                          3).ToString();
   Controls.SetTextBoxText(txtTime, s);
```

```
end;
```

Обратите внимание, что для точного подсчёта времени мы используем свойство *ElapsedMilliseconds* класса *Clock*.

Тараканы помчатся вперёд после нажатия на **кнопку** *Начать забег!*:

```
// КНОПКА

// Начать забег:
btnStart := Controls.AddButton('Начать забег!', 10, height - 32);
Controls.ButtonClicked += OnClick;
```

Для вывода информации мы воспользуемся услугами двух **текстовых полей**. Первое, простое будет показывать время, второе – *многострочное* – результаты забега:

```
// ТЕКСТОВЫЕ ПОЛЯ

// Время забега:
txtTime := Controls.AddTextBox(340, 10);
Controls.SetSize(txtTime, 100, 26);
Controls.HideControl(txtTime);
// Победитель:
txtWin := Controls.AddMultiLineTextBox(10, 300);
Controls.SetSize(txtWin, 320, 100);
Controls.HideControl(txtWin);
```

Перед первым стартом мы сбрасываем флажки и расставляем тараканов на стартовой позиции:

```
  flgStart := false;
  flgFinish := false;
  PrepareGame();
end;
```

В методе **PrepareGame** мы создаём новых тараканов заданного цвета и размера. В нашей модели роль тараканов будут исполнять эллипсы. Почему эллипсы? – Во-первых, эллипсы куда приятнее для глаз, чем тараканы. Во-вторых, эллипсы более универсальны: под ними можно понимать и тараканов, и крыс, и жеребцов, что для нашей компьютерной модели тоже

является плюсом. Ну и наконец, вы очень просто можете заменить эллипсы любыми картинками, если загрузите их из файла, и метод *AddEllipse* замените методом *AddImage.*

Здесь же ветврач оценивает боеготовность каждого таракана и делает соответствующие записи в массиве **zdor**.

Для большего правдоподобия проводим *две черты – стартовую* и *финишную*, чтобы ни у одного таракана даже не возникла мысль о фальстарте!

```
// Готовим тараканов к старту
private procedure PrepareGame();
begin
   for var i := 0 to N_TARAKAN - 1 do
   begin
      // цвет таракана:
      var clr := color[i];
      GraphicsWindow.BrushColor := clr;
      GraphicsWindow.PenColor := clr;
      // скрываем старых тараканов:
      Shapes.Remove(tar[i]);
      // создаём новых тараканов:
      tar[i] := Shapes.AddEllipse(wTar, hTar);
      // и расставляем их у стартовой черты:
      x[i] := xStart;
      y[i] := yStart + dy * i;
      Shapes.Move(tar[i], x[i], y[i]);
      // здоровье тараканов:
      zdor[i] := (rand.Next(30) + 1.0) / 100 + 1.2;
   end;
   // чертим линию старта:
   GraphicsWindow.PenColor := 'Red';
   GraphicsWindow.DrawLine(xStart + wTar, yStart, xStart + wTar,
                           y[N_TARAKAN - 1] + hTar);

   // чертим линию финиша:
   GraphicsWindow.PenColor := 'Green';
   xFinish := width - wTar;
   GraphicsWindow.DrawLine(xFinish, yStart, xFinish,
                          y[N_TARAKAN - 1] + hTar);
   // прячем информационное окно:
   Controls.HideControl(txtWin);
end;
```

Тараканы на старте – можно начинать **игровой цикл**:

```
public procedure Game();
begin
   // игровой цикл:
   while (not flgFinish) do
   begin
      // нажата кнопка "Начать забег!":
      if (flgStart) then
      begin
         // вычисляем новые координаты тараканов:
         CalcNewCoords();
         // и перемещаем их туда:
         for var i := 0 to N_TARAKAN - 1 do
            Shapes.Move(tar[i], x[i], y[i]);

         // устанавливаем скорость работы программы:
         Thread.Sleep(TimeSpan.FromMilliseconds(20));
         // проверяем, не финишировал ли таракан:
         TestFinish();
      end
   end
end;
```

Он будет продолжаться до тех пор, пока один из тараканов не достигнет финиша – тогда флаг **flgFinish** будет установлен в *true*.

Однако тараканы будут послушно стоять на старте, если флаг *flgStart* равен *false*. Как вы помните, в начале программы мы задали ему именно это значение. А вот когда пользователь или другой естествоиспытатель нажмёт кнопку **Начать забег!**, ситуация резко изменится:

```
// Начинаем забег
private procedure OnClick();
begin
   Controls.HideControl(btnStart);
   Controls.ShowControl(txtTime);
   // обнуляем время:
   time := 0;
   Controls.SetTextBoxText(txtTime, 'Время: 0');
   // запускаем таймер:
   Timer.Resume();
   // засекаем время старта по системным часам:
   startTime := Clock.ElapsedMilliseconds;
   // стреляем из стартового пистолета:
   flgStart := true;
end;
```

В методе-обработчике **OnClick** мы готовим к бою секундомер и понуждаем тараканов к бегу громким звуком.

Условие *flgStart = true* в игровом цикле выполнено, и тараканы помчались вперёд. В методе **CalcNewCoords** мы определяем текущее положение каждого таракана.

Поскольку тараканы обязаны бежать неравномерно, то каждый раз мы *случайно* задаём каждому таракану то расстояние, которое он должен пробежать в данный момент. Обратите внимание на то, что мы учитываем и физическое состояние каждого таракана. Недомогающие особи будут медленнее шевелить ногами:

```
// Вычисляем новые координаты тараканов
private procedure CalcNewCoords();
begin
   for var i := 0 to N_TARAKAN - 1 do
   begin
      // случайное перемещение:
      var dx := rand.Next(43) / 10.0 + 0.9;
      // новое положение таракана на дистанции:
      x[i] += dx * zdor[i];
   end
end;
```

Переместив всех тараканов в их новое положение, мы должны проверить, не пересёк ли хотя бы один из них финишную черту, иначе один за другим они скроются за пределами окна приложения, так и не выявив победителя. В том, что рано или поздно тараканы доползут до финиша, сомневаться не приходится, поскольку они бегут только вперёд.

Нам осталось рассмотреть последний метод - **TestFinish**, в котором мы и определяем, закончился ли забег, и если закончился, то какой таракан пришел первым. Здесь важно учесть, что финишировать могут и *несколько* тараканов. Тогда придётся проводить «фотофиниш»: победителем будет признан тот таракан, который убежал дальше за линию финиша на момент проверки:

```
// Проверяем финиш
private procedure TestFinish();
begin
   var xEnd := 0.0;
   var n := 0;
   for var i := 0 to N_TARAKAN - 1 do
```

```
    begin
      if (x[i] >= xFinish - wTar) then
      begin
        flgFinish := true;
        Timer.Pause();
        if (x[i] > xEnd) then
        begin
          // определяем номер победителя:
          n := i + 1;
          xEnd := x[i];
        end
      end
    end;
    if (flgFinish) then
    begin
      var s := ' Победил таракан # ' + n.ToString() + '!' +
               Environment.NewLine;
      var t := (double)(Clock.ElapsedMilliseconds) - startTime;
      t := System.Math.Round(t / 1000, 3);
      Controls.SetTextBoxText(txtTime, 'Время: ' +
                              t.ToString());

      s += ' Его время: ' + t.ToString() + ' сек.';
      flgStart := false;
      Controls.SetTextBoxText(txtWin, s);
      Controls.ShowControl(txtWin);
      Controls.ShowControl(btnStart);
      // ждём нового старта:
      while (not flgStart) do;

      // начинаем новый забег:
      flgFinish := false;
      Controls.SetTextBoxText(txtTime, ' Время 0');
      // готовимся к новой игре:
      PrepareGame();
    end
end;
```

После финиша следует объявление чемпиона, оглашение победного времени (Рис. 19.4) и народные гуляния. Затем все болельщики, пьяные от счастья и шампанского, вновь запускают свежих тараканов по скользкому пути азарта.

Кстати говоря, наша компьютерная модель, несмотря на свою простоту, вполне адекватно отражает процесс тараканьей беготни, за которой наблюдать весьма любопытно. А если добавить к игре ещё и возможность делать ставки, то и на ипподром ходить не надо!

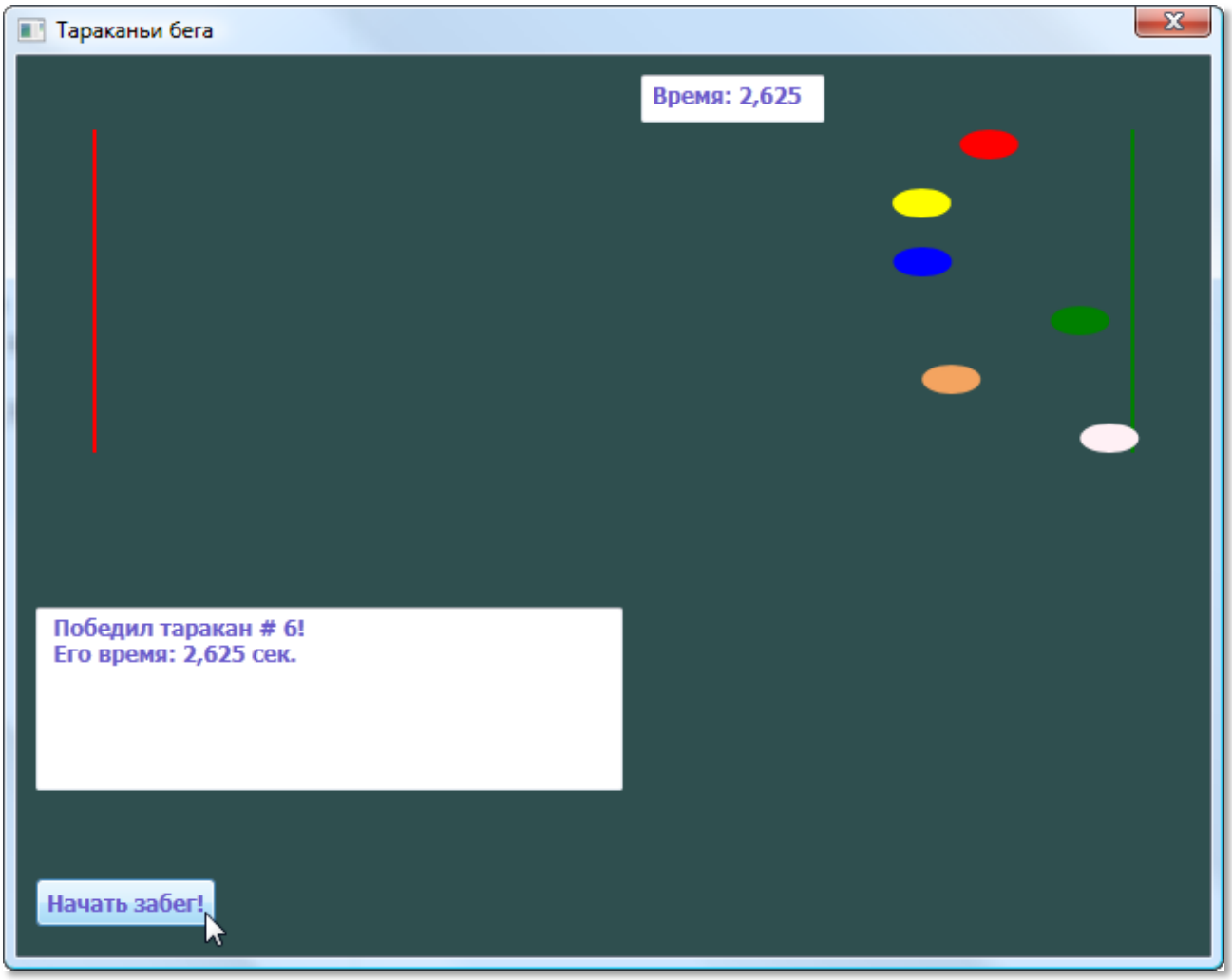


Рис. 19.4. Делайте ставки, господа!

Проект *Электронная гадалка*

Исходный код программы находится в папке **Электронная гадалка**.

Прошедший в 2010 году Чемпионат мира по футболу запомнился многим болельщикам и просто зрителям не только отвратительными африканскими дуделками (вувузелами), но и блестящими предсказаниями результатов матчей, которые делал немецкий **осьминог Пауль** (Рис. 19.5).

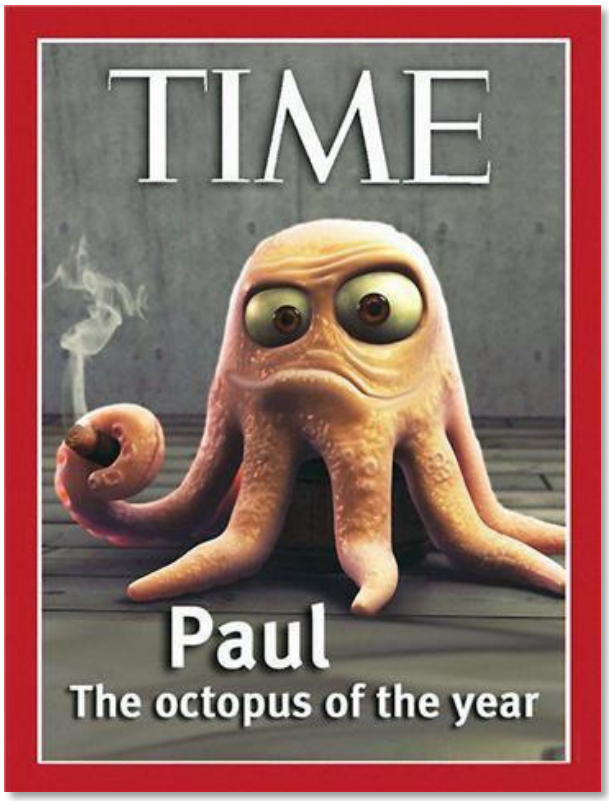


Рис. 19.5. Карикатурная обложка журнала *Тайм*, который признал Пауля лучшим осьминогом года

Он угадал результаты всех семи игр сборной Германии, а также финала. Итого - 8 игр. Для гадания ему предлагали два прозрачных ящичка с флагами стран-участниц очередного матча, из которых он выбирал один – открывал его щупальцами.

> Поскольку встречи в группах могли закончиться и вничью, то само гадание было не совсем корректным. Впрочем, все гадания прошли успешно, потому что сборная Германии ни разу вничью не сыграла.
>
> В Испании, чья футбольная команда завоевала Кубок мира, сделали бронзовую статую осьминога и подарили её океанариуму немецкого города Оберхаузена. Также он стал почётным гражданином испанского города Карбальино, а американец Пэрри Грипп посвятил ему песню *Осьминог Пауль*. В самой Германии, правда, из него хотели сварить суп, но, в конце концов, его простили, поскольку он выступил лучше сборной Германии, и отправили на заслуженный отдых.
>
> Вот так осьминог Пауль *благополучно* завершил свою карьеру предсказателя, что с предсказателями бывает нечасто.

Однако давайте подсчитаем, так ли уж велики заслуги этого головастого головоногого моллюска перед мировым сообществом. Если в каждом матче выигрывает одна из двух команд, то вероятность правильного предсказания результата одной игры равна ½, двух – $(½)^2$ = ¼, а всех восьми - $(½)^8$ = 1/256. То есть из 256 осьминогов один почти наверняка угадал бы все результаты. Так что подвиг осьминога Пауля не столь уж значителен, особенно если учесть, что некоторые люди выигрывают и в *Спортлото* (вероятность угадывания 5 чисел из 36 равна 1/45 239 040, а 6 чисел из 49 - 1/10 068 347 520).

Поскольку результаты футбольных матчей в большой мере случайны, а учесть все факторы невозможно, то мы будем вслед за Паулем полагать, что у каждой команды *равные* шансы на победу.

И нам осталось написать простенькую программу **Осьминог Пауль**, которая моделирует простейшего *электронного оракула*.

Для ввода названий команд нам потребуются два **текстовых поля** и столько же **переменных**.

```
unit DrawUnit;

uses
```

```
   Microsoft.SmallBasic.Library, System;

type
   Draw = class
   private
      const GWWIDTH = 440;
      const GWHEIGHT = 270;
      const BackgroundColor = '2F4F4F';
      // размеры окна:
      width := GWWIDTH;
      height := GWHEIGHT;
      // координаты центра окна:
      CX := width div 2;
      CY := height div 2;


      // названия команд:
      team1: string;
      team2: string;

      // ЭУ
      txtTeam1: string;
      txtTeam2: string;
      txtWin: string;
      btnStart: string;

      rand := new Random();

      public procedure Prepare();
      begin
         GraphicsWindow.Hide();
         GraphicsWindow.Title := 'Осьминог Пауль';
         GraphicsWindow.Width := GWWIDTH;
         GraphicsWindow.Height := GWHEIGHT;
         GraphicsWindow.Show();
         GraphicsWindow.Left := (Desktop.Width -
                                 GraphicsWindow.Width) / 2;
         GraphicsWindow.Top := (Desktop.Height -
                                GraphicsWindow.Height) / 2;
         GraphicsWindow.CanResize := false;

         var path := Environment.CurrentDirectory;
         var background := ImageList.LoadImage(path + '\paul.jpg');
         GraphicsWindow.DrawImage(background, 0, 0);

         // шрифт:
         GraphicsWindow.BrushColor := 'Green';
         GraphicsWindow.FontBold := 'True';
         GraphicsWindow.FontSize := 16;
```

Отдавая должное нашему герою Паулю, мы поместим его фото на задний план окна (Рис. 19.6).

Перейдем к **элементам управления**. Запишите рецепт: возьмите одну *кнопку,* три *текстовых поля* и разместите их в окне по вкусу. Например, так, как показано на Рис. 19.6:

```
   // КНОПКА

   // Предсказать результат:
   btnStart := Controls.AddButton('ПРОГНОЗ!', 10, height - 32);
   Controls.ButtonClicked += OnClick;

   // ТЕКСТОВЫЕ ПОЛЯ

   // Команды:
   txtTeam1 := Controls.AddTextBox(10, 10);
   Controls.SetSize(txtTeam1, 160, 26);
   GraphicsWindow.DrawText(180, 14, '<- Первая команда');

   txtTeam2 := Controls.AddTextBox(10, 48);
   Controls.SetSize(txtTeam2, 160, 26);
   GraphicsWindow.DrawText(180, 52, '<- Вторая команда');

   GraphicsWindow.BrushColor := 'Red';
   // Победитель:
   txtWin := Controls.AddMultiLineTextBox(10, 180);
   Controls.SetSize(txtWin, 300, 50);
end;
```

Нажимаем **кнопку** – и получаем предсказание на интересующую нас игру:

```
private procedure OnClick();
begin
   var s := String.Empty;
   var s1 := Controls.GetTextBoxText(txtTeam1).ToString();
   var s2 := Controls.GetTextBoxText(txtTeam2).ToString();

   if (s1 = String.Empty) or (s2 = String.Empty) then
      s := ' Введите названия команд!'
   else if (team1 = s1) and (team2 = s2) then
      s := ' Повторно не гадаю!'
   else if (s1 = s2) then
      s := ' Одна и та же команда!'
   else
   begin
```

```
        //запоминаем гадание:
      team1 := s1;
      team2 := s2;
      if (rand.Next(2) = 0) then
         s := ' Победит ' + team1
      else
         s := ' Победит ' + team2;
   end;

   GraphicsWindow.BrushColor := 'Blue';
   s += Environment.NewLine + 'Давайте погадаю еще раз!';
   Controls.SetTextBoxText(txtWin, s);
end;
```

Б*о*льшую часть метода занимают проверки и беседа с пользователем, но зато так наш предсказатель выглядит более мудрым. А само предсказание занимает пару строк и основано всё на том же методе *Next* класса *Random*, что и в предыдущем проекте.

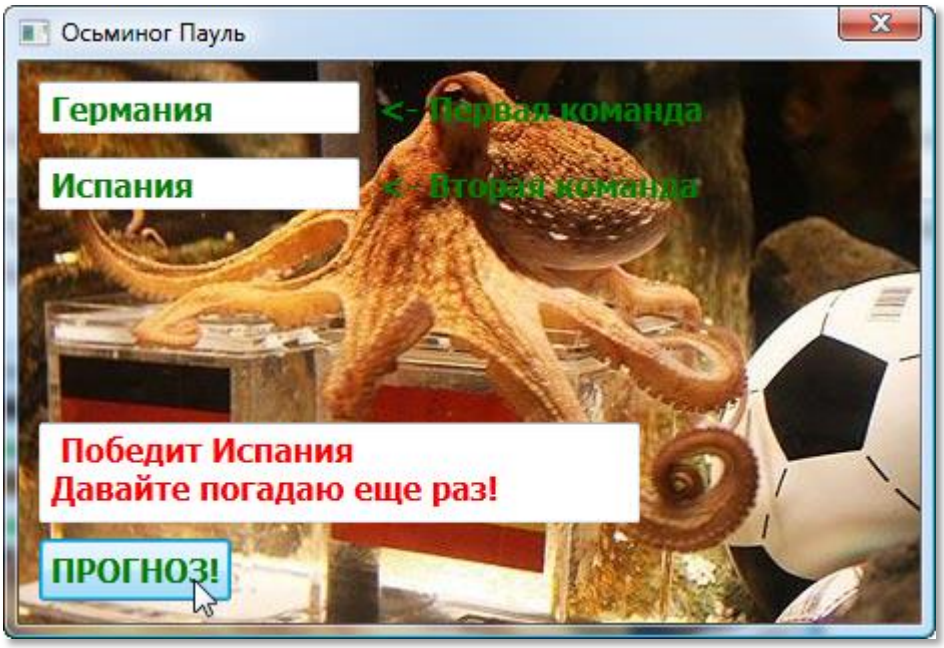
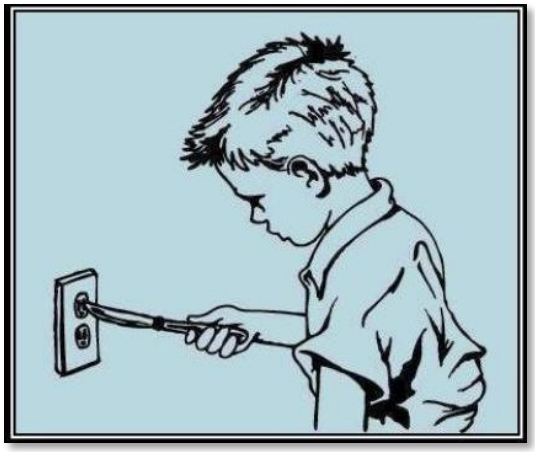


Рис. 19.6. Наш электронный Пауль – парень не промах!

Конечно, наша математическая модель слишком проста, но её можно улучшить, если вычислять вероятность исхода матча с учётом рейтинга ФИФА для сборных команд. Для гаданий на Чемпионат России, например, вполне уместно оценивать положение команд в турнирной таблице и статистику личных встреч. Но сколько бы мы факторов ни учитывали, а всё равно не угадаем!

Проект *Число пи, или Метод научного тыка*

Исходный код программы находится в папке **Число пи**.



А теперь давайте рассмотрим пример более серьёзного применения метода Монте-Карло.

Пусть нам нужно вычислить площадь **S** какой-либо криволинейной фигуры, которая задана графически (нарисована) или аналитически (формула).

Впишем её в квадрат (Рис. 19.7) со сторонами **L** сантиметров (или миллиметров, или метров) и начнём совершенно случайно «тыкать» пальцем так, чтобы все «тыки» приходились в квадрат (можно тыкать и мимо, но тогда неудачные тыки мы просто не учитываем).

Поскольку интересующая нас фигура целиком и полностью лежит *внутри* квадрата, то ей тоже достанется. И вероятность того, что тык попадёт именно в фигуру, прямо пропорциональна её площади.

Если мы тыкнули **N** раз (**красные** и **жёлтые** точки на Рис. 20.7, справа), а в фигуру угодили **P** раз (**красные** точки там же), то мы можем вычислить площадь фигуры так. Площадь квадрата равна L^2, а нашей фигуры – S, значит:

$$L^2 / S = N / P$$

откуда находим **S**:

$$S = L^2 * P / N \qquad (1)$$

Нетрудно понять, что при малом количестве тыков точность вычислений по формуле (1) будет невелика, однако, как показывает практика, при больших N погрешность вычислений уменьшается.

Естественно, тысячи раз тыкать пальцем (а лучше всё-таки карандашом) в чертёж, да ещё при этом надеяться на случайное распределение тыков по площади квадрата, было бы неразумно, поэтому мы поступим правильнее, если научим этому нехитрому занятию компьютер. Он всё сделает быстро и аккуратно.

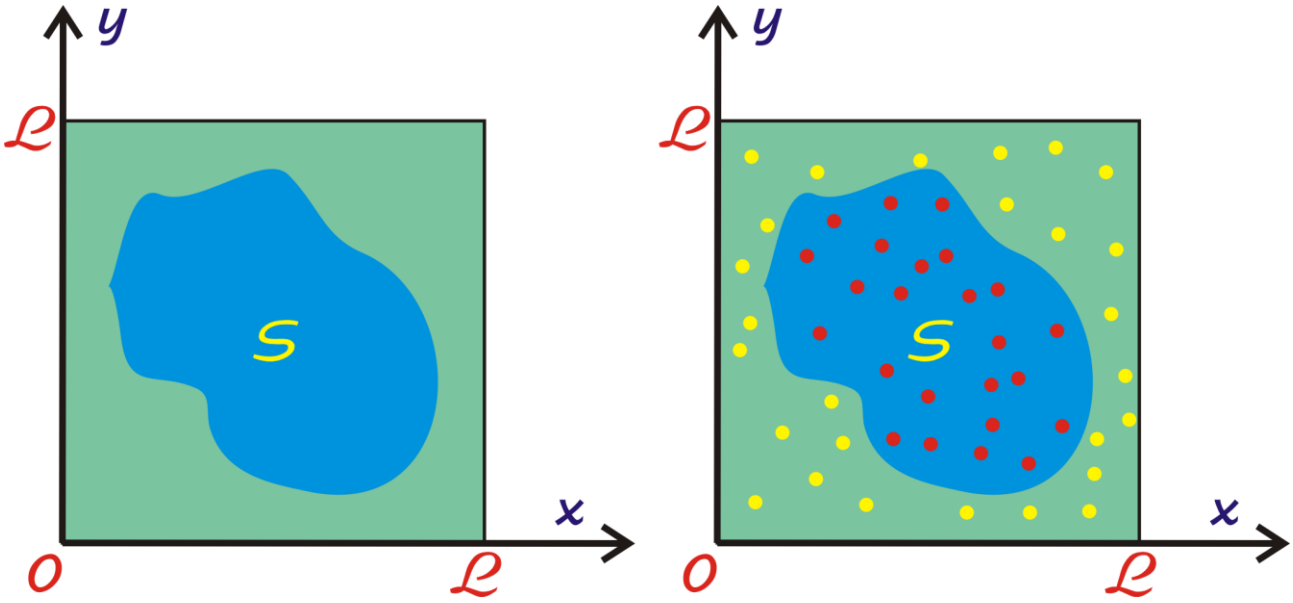


Рис. 19.7. Метод научного тыка в действии

Попробуем этим способом определить число *пи*. Мы знаем, что площадь любой плоской фигуры прямо пропорциональна квадрату её характерного размера. Для собственно квадрата это **длина** стороны, а для круга – **диаметр**. Коэффициент пропорциональности для квадрата равен единице, а для круга – числу *пи,* которого мы пока не знаем.

Нанесём по квадрату и вписанному в него кругу **N** случайных выстрелов, из которых **P** попадут в круг (Рис. 19.8).

Отношение чисел *P* и *N* пропорционально площадям фигур:

$$\frac{P}{N}=\frac{\pi D^2}{4D^2}$$

Из этой формулы мы легко найдем *пи*:

$$\pi=\frac{4P}{N}$$

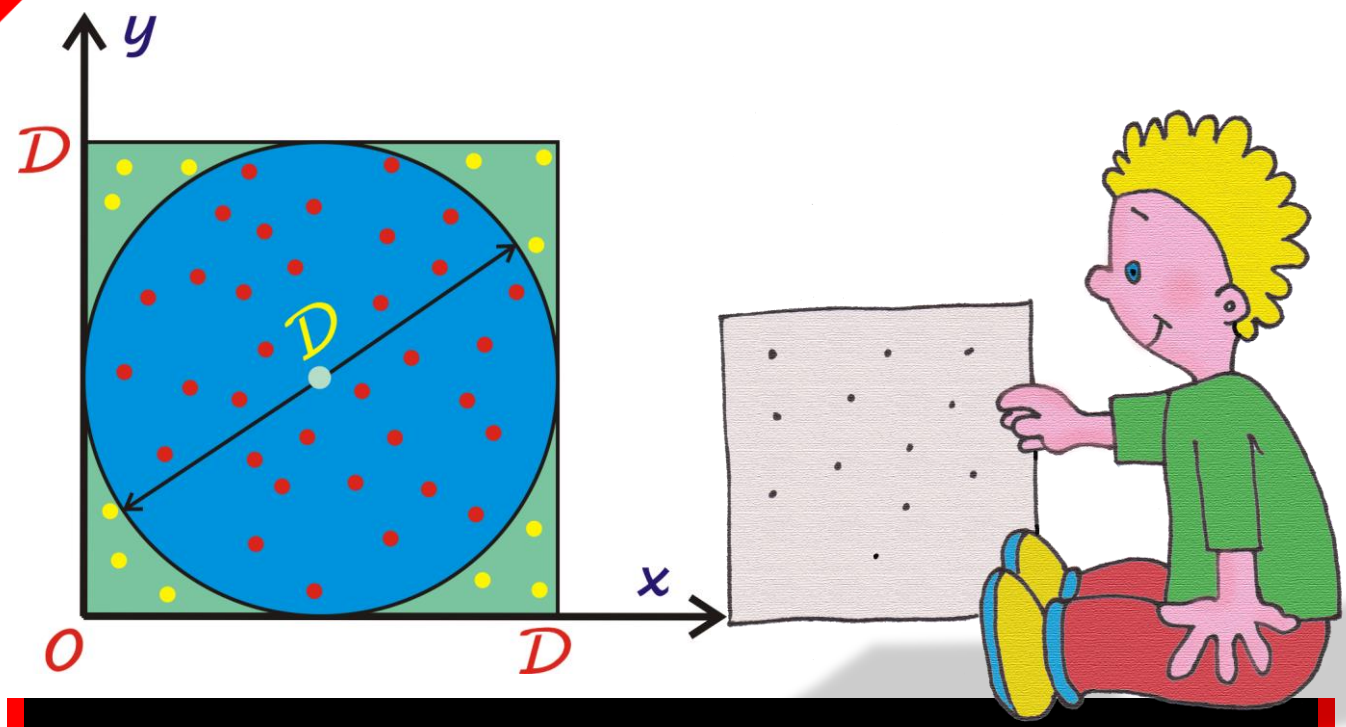


Рис. 19.8. Стреляем по кругу, а попадаем по квадрату

Новую программу **Метод Монте-Карло**, как обычно, мы начнём с объявления *констант* и *переменных*:

```
unit DrawUnit;

uses
   Microsoft.SmallBasic.Library, System;

type
   Draw = class
   private
      const GWWIDTH = 410;
      const GWHEIGHT = 450;
      // цвет фона
      const BackgroundColor = 'MidnightBlue';

      // размеры окна:
      width := GWWIDTH;
      height := GWHEIGHT;
      //координаты центра окна:
      CX := width div 2;
      CY := height div 2;


      // смещение фигур от левого верхнего края окна:
      const dx = 10;
      const dy = 50;
      // диаметр круга:
```

```
      const D = 400.0;
      const D2 = D * D;
      // радиус круга:
      const R = D / 2;

      // общее число точек:
      N := 10000; //160000;//
      // число точек, попавших в круг:
      P: integer;

      //центр круга:
      xc := 0.0;
      yc := 0.0;

      // переменная цикла:
      ii: integer;

      rand := new Random();

      // ЭУ
      txtPi: string;
      btnStart: string;
```

Задаём разумные **размеры** окна и фигур, хотя в нашем случае размер имеет значение – чем больше, тем лучше!

```
public procedure Prepare();
begin
   GraphicsWindow.Hide();
   GraphicsWindow.Title := 'Метод Монте-Карло';
   GraphicsWindow.Width := GWWIDTH;
   GraphicsWindow.Height := GWHEIGHT;
   GraphicsWindow.Show();
   GraphicsWindow.Left := (Desktop.Width –
                           GraphicsWindow.Width) / 2;
   GraphicsWindow.Top := (Desktop.Height –
                          GraphicsWindow.Height) / 2;
   GraphicsWindow.CanResize := false;
   GraphicsWindow.BackgroundColor := BackgroundColor;
```

Из **элементов управления** нам понадобится *кнопка*, чтобы начать процесс. *Текстовое окно* для вывода текущего значения числа *пи* и *таймер*, который и обеспечит нас актуальной информацией о ходе эксперимента:

```
 // КНОПКА
```

```
btnStart := Controls.AddButton('Начать!', 10, 10);
Controls.ButtonClicked += OnClick;

// ТЕКСТОВОЕ ПОЛЕ

// Пи:
txtPi := Controls.AddTextBox(100, 10);
Controls.SetSize(txtPi, 200, 24);


// ТАЙМЕР
Timer.Interval := 100;
Timer.Pause();
Timer.Tick += OnTick;
```

Рисуем на канве героев нашего эксперимента – **квадрат** и вписанный в него **круг**:

```
   // рисуем квадрат:
   var clr := 'Green';
   GraphicsWindow.BrushColor := clr;
   GraphicsWindow.PenColor := 'Black';
   GraphicsWindow.FillRectangle(dx, dy, D, D);
   // рисуем круг:
   clr := 'Blue';
   GraphicsWindow.BrushColor := clr;
   GraphicsWindow.FillEllipse(dx, dy, D, D);
   // вычисляем координаты центра круга:
   xc := D / 2 + dx;
   yc := D / 2 + dy;
end;
```

Для эксперимента всё готово. Без тени сомнения нажимаем кнопку **Начать!** – и *процесс пошёл*, как говаривал первый Президент СССР.

> Правда, со словом *начать* у него были орфоэпические проблемы.

```
private procedure OnClick();
begin
   // запускаем таймер:
   Timer.Resume();
   // ставим точки:
   P := 0;
   SetPoints();
```

```
end;

// Вычисляем пи:
private procedure  OnTick();
begin
   var s := 'Пи = ' + (4.0 * P / ii).ToString();
   Controls.SetTextBoxText(txtPi, s);
end;
```

Главные события сей научной драмы разворачиваются в методе **SetPoints**:

```
// Ставим точки
private procedure SetPoints();
begin
   var clr := String.Empty;
   for var i := 1 to N do
   begin
      ii := i;
      // выбираем случайные координаты  точки:
      var x := rand.NextDouble() * D + dx;
      var y := rand.NextDouble() * D + dy;
      var xx := xc - x;
      var yy := yc - y;
      // попали в круг?
      if (xx * xx + yy * yy <= R * R) then //попали
      begin
         P += 1;
         clr := 'Red';
      end
      else // не попали
         clr := 'Yellow';

      // устанавливаем цвет точки:
      GraphicsWindow.BrushColor := clr;
      // рисуем точку:
      GraphicsWindow.FillEllipse(x, y, 2, 2);
   end;
```

Чтобы проверить, попадает ли точка с координатами *(xx, yy)* внутрь круга или нет, достаточно по теореме Пифагора вычислить длину гипотенузы *r* прямоугольного треугольника (Рис. 19.9):

$$r = \sqrt{xx^2 + yy^2}$$

Поскольку *r >= 0*, то обе части равенства можно возвести в квадрат:

$$r^2 = xx^2 + yy^2$$

Теперь достаточно сравнить r^2 с R^2. Если r^2 <= R^2, то точка находится *внутри* круга, иначе – *снаружи*.

> В *компьютерной* модели проверку можно проводить и иначе. Мы знаем, что круг окрашен в **синий** цвет, поэтому достаточно проверить цвет пикселя в точке с координатами *(xx,yy)*. Если он **синий**, то точка находится внутри круга, в противном случае – снаружи.
>
> Однако после того как в круге появятся точки **красного** цвета, нужно будет учитывать и их тоже. Поскольку мы окрашиваем точки вне круга в **жёлтый** цвет, то проверка будет очень простой.

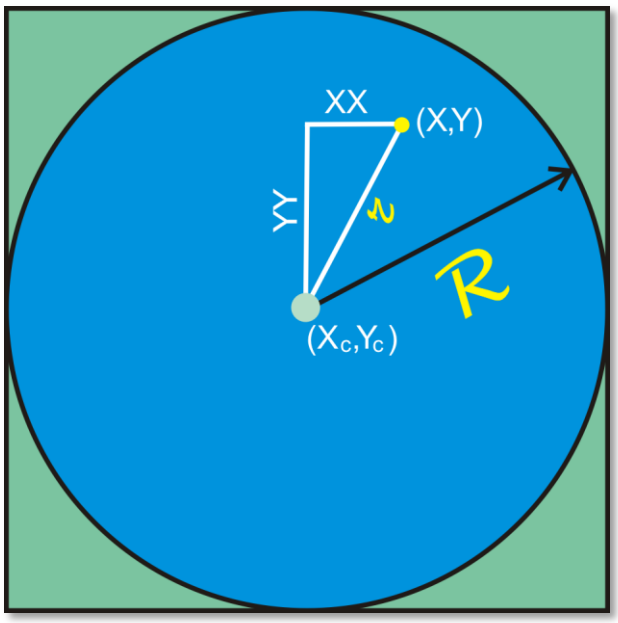


Рис. 19.9. Геометрия – это наука!

Вопреки теоретическим изысканиям, наша модель не уточняет значения *пи* с увеличением числа испытаний, что хорошо видно на Рис. 20.10. Зато генератор случайных чисел работает совсем неплохо, поскольку на правом рисунке почти все точки закрашены (при его идеальной работе были бы закрашены все 160 000 точек, а это не так).

И хотя в *текстовом поле* мы видим много цифр после запятой, точность вычислений ограничивается уже *вторым* знаком. Несколько улучшить ситуацию можно, если провести **серию** испытаний, а затем вычислить среднее значение.

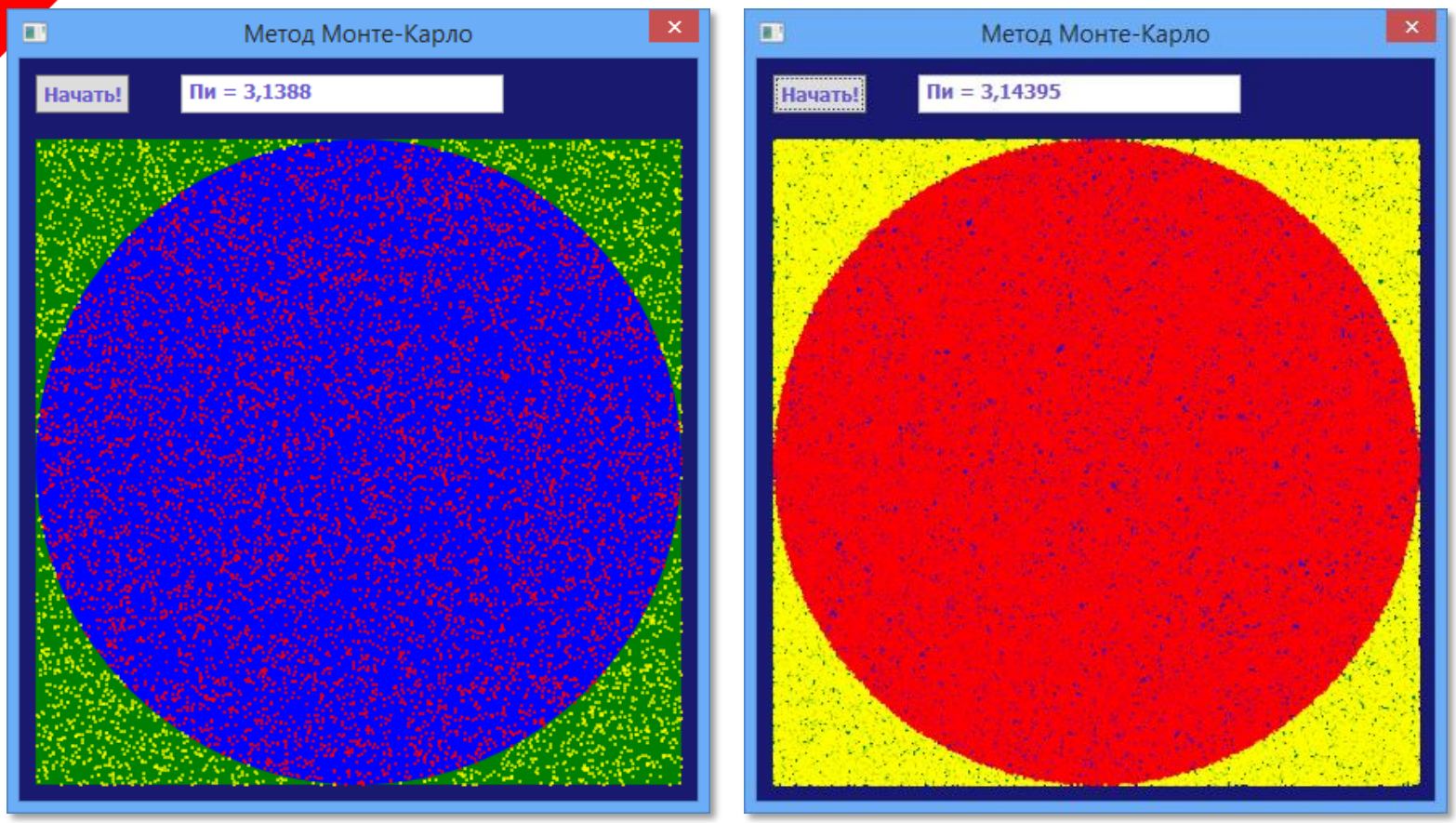


Рис. 19.10. Результаты эксперимента при N= 10 000 и N= 160 000

Задания для самостоятельного решения

Метод Монте-Карло

1. Напишите программу для бросания **монеты**. Для этого достаточно слегка изменить проект *Осьминог Пауль*.

2. Немного труднее научить программу бросать кубик, вытаскивать карту из колоды или бочонок из мешка в игре лото.

3. Экстремальным вариантом рулетки можно считать **русскую рулетку** (иначе она называется *гусарской рулеткой*). «Играют» в неё так. В барабан шестизарядного револьвера вставляется один патрон (из соображений гуманности и здравомыслия будем считать, что *холостой*). Затем

барабан прокручивается несколько раз так, чтобы «игроки» не знали его положения относительно ствола пистолета. Мы, не обсуждая все разновидности этой игры, будем считать, что после каждого спуска крючка барабан прокручивается вновь. Таким образом, вероятность выстрела будет постоянной на протяжении всей игры и равна 1/6. Соответственно, вероятность благоприятного исхода после первого выстрела равна 1 - 1/6 = 5/6 = 0,833. После второго (5/6)2 = 0,694, после шестого - (5/6)6 = 0,335, и так далее, приближаясь к нулю (вот почему так важно учить теорию вероятностей!). Математическая модель этой игры ничем не отличается от моделирования бросания игрального кубика, у которого в точности шесть граней.

4. Слот-машина, больше известная под названием *однорукий бандит*, представляет собой три или пять барабанов, которые вращаются независимо друг от друга, наподобие колеса рулетки. На боковую поверхность барабанов нанесены различные символы, комбинация которых после остановки колёс и определяет сумму выигрыша или проигрыша игрока, сделавшего ставку (Рис. 19.11). Справедливое прозвище этих автоматов *однорукий бандит* восходит к тому времени, когда колёса раскручивались рычагом на боковой поверхности автомата. Этот рычаг и придавал ему такой запоминающийся облик.

Сейчас больше распространены видео-слоты, которые обходятся без механических барабанов, поэтому рычаг им не нужен. Поскольку в новых машинах комбинация символов задаётся генератором псевдослучайных чисел, то достаточно просто написать программу, имитирующую их бандитскую деятельность.

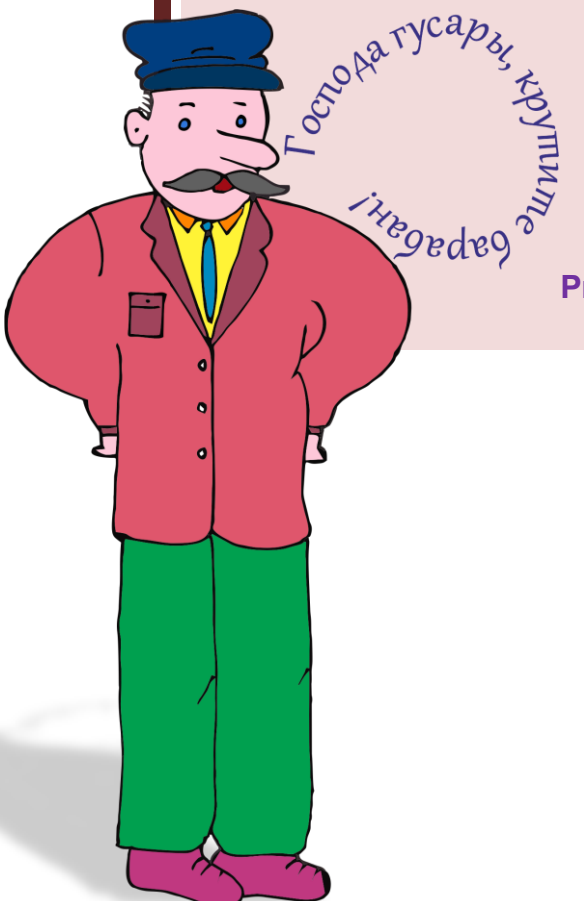


Рис. 19.11. Удачная комбинация!

Справочник

В следующих таблицах представлена информация о самых важных классах библиотеки **SmallBasicLibrary** и некоторых **дополнительных библиотек**.

Таблица <Класс *Clock*>

Класс **Clock** предоставляет свойства для работы с датами и временем.

Класс находится в пространстве имён *Microsoft.SmallBasic.Library*.

Свойства	Описание
Date ← Primitive	Возвращает текущую **дату:** `3.12.2014`
Year ← integer	Возвращает текущий **год:** `2014`
Month ← integer	Возвращает текущий **месяц** 1..12: `12`
Day ← integer	Возвращает текущий **день месяца:** `3`
WeekDay ← string	Возвращает **название** текущего **дня недели:** `среда`
Hour ← integer	Возвращает текущий **час 0..23:** `18`
Minute ← integer	Возвращает число **минут** текущего времени: `15`
Second ← integer	Возвращает число **секунд** текущего времени 0..59: `51`
Millisecond	Возвращает **миллисекунды** текущего вре-

← integer	мени: 781
Time ← Primitive	Возвращает текущее **время:** 18:15:51
ElapsedMilliseconds ← double	Возвращает общее **число миллисекунд**, прошедших с 1 января 1900 года: 3591612491020,88

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

Таблица <Класс *Controls*>

Класс **Controls** предоставляет свойства и методы для работы с элементами управления.

{...} Класс находится в пространстве имён *Microsoft.SmallBasic.Library*.

Свойства	Описание
LastClickedButton ← string	**Возвращает** имя последней нажатой **кнопки**.
LastTypedTextBox ← string	**Возвращает** имя последнего **текстового поля**, в котором набирался текст.
Методы	**Описание**
AddButton() → string Caption → integer left → integer top ← string	**Создаёт** новую **кнопку** с заданной надписью *Caption* и устанавливает её в точку с координатами *(left, top)*. **Возвращает** имя кнопки.
AddMultiLineTextBox() → int eger left	**Создаёт** новое **многострочное текстовое поле** и устанавливает его в точку с коорди-

→ int eger top ← string	натами *(left, top).* **Возвращает** имя многострочного текстового поля.
AddTextBox() → integer left → integer top ← string	**Создаёт** новое **текстовое поле** и устанавливает его в точку с координатами *(left, top).* **Возвращает** имя текстового поля.
GetButtonCaption() → string buttonName ← string	**Возвращает** надпись на **кнопке** *buttonName*.
GetTextBoxText() → string textBoxName ← string	**Возвращает** текст, набранный в **текстовом поле** *textBoxName*.
HideControl() → string controlName	**Скрывает** указанный **элемент управления** *controlName*.
ShowControl() → string controlName	**Показывает** указанный **элемент управления** *controlName.*
RemoveControl() → string controlName	**Удаляет** указанный **элемент управления** *controlName.*
Move() → string control → integer x → integer y	**Перемещает** указанный **элемент управления** *control* в точку с координатами *(x, y).*
SetButtonCaption() → string buttonName → string Caption	Задаёт новую **надпись** *Caption* на **кнопке** *buttonName*.
SetSize() → string control → integer width → integer height	Задаёт новые **размеры** *width* и *height* заданному **элементу управления** *control*.
SetTextBoxText() → string textBoxName	**Печатает** текст *text* в **текстовом поле** *textBoxName*.

→ string text	

События	Описание
ButtonClicked	**Возникает** при нажатии на **кнопку**.
TextTyped	**Возникает** при **вводе текста** в текстовое поле.

Таблица <Класс *Desktop*>

Класс **Desktop** предоставляет методы для определения размеров *Рабочего стола*.

Класс находится в пространстве имён *Microsoft.SmallBasic.Library*.

Методы	Описание
Width() ← integer	Возвращает **ширину** *Рабочего стола* в пикселях.
Height() ← integer	Возвращает **высоту** *Рабочего стола* в пикселях.
SetWallPaper() → string fileOrUrl	Заменяет текущие **обои** картинкой, путь к которой задан строкой *fileOrUrl*.

Таблица <Класс *Flikr*>

Класс **Flikr** предоставляет методы для загрузки файлов изображений с сайта *flickr.com*. **Не работает!**

{...} Класс находится в пространстве имён *Microsoft.SmallBasic.Library*.

Методы	Описание
GetPictureOfMoment() ← string	**Возвращает путь** к случайному файлу изображения на сайте *flickr.com*.
GetRandomPicture() → string tag ← string	Как предыдущий метод, но строковый параметр *tag* позволяет указать **ключевое слово**, или тему изображения.

Таблица <Класс *GraphicsWindow*>

Класс **GraphicsWindow** предоставляет свойства и методы для работы с *Графическим окном*.

{...} Класс находится в пространстве имён *Microsoft.SmallBasic.Library*.

Свойства	Описание
BackgroundColor → string ← string	Устанавливает или возвращает **цвет фона** *Графического окна*.
BrushColor → string ← string	Устанавливает или возвращает **цвет заливки** фигур и текста.
PenColor → string ← string	Устанавливает или возвращает **цвет контура** фигур и линий.
PenWidth	Устанавливает или возвращает **толщину**

→ double ← double	**контура** фигур и линий.
CanResize → bool ← bool	Если значение свойства равно *true*, то **размеры окна приложения** допускается изменять. Если значение свойства равно *false*, то размеры окна приложения изменять нельзя.
FontBold → bool ← bool	Если значение свойства равно *true*, то **надписи** выводятся **жирным** шрифтом. Если значение свойства равно *false*, то обычным шрифтом.
FontItalic → bool ← bool	Если значение свойства равно *true*, то **надписи** выводятся ***наклонным*** шрифтом. Если значение свойства равно *false*, то прямым шрифтом.
FontName → string ← string	Устанавливает или возвращает **название** текущего **шрифта**.
FontSize → integer ← integer	Устанавливает или возвращает **размер** текущего **шрифта**.
Height → integer ← integer	Устанавливает или возвращает **высоту** *Графического окна*.
Width → integer ← integer	Устанавливает или возвращает **ширину** *Графического окна*.
Left → integer ← integer	Устанавливает или возвращает ***x*-координату** левого верхнего угла *Графического окна*.
Top → integer ← integer	Устанавливает или возвращает ***y*-координату** левого верхнего угла *Графического окна*.
Title → string	Устанавливает или возвращает **заголовок** *Графического окна*.

← string	
LastKey ← string	Возвращает **название** последней нажатой **клавиши**.
LastText ← string	Возвращает **символ** последней нажатой **клавиши**.
MouseX ← integer	Возвращает ***x*-координату курсора** мышки в *Графическом окне*.
MouseY ← integer	Возвращает **y-координату курсора** мышки в *Графическом окне*.
Методы	**Описание**
Clear()	**Стирает** *Графическое окно*, заливая его цветом фона.
DrawText() → double x → double y → string text	**Печатает** текст *text* текущим шрифтом, начиная с указанных координат *(x,y)* верхнего левого угла описанного прямоугольника. **Цвет** символов определяется текущим значением свойства *BrushColor*.
DrawBoundText() → double x → double y → double width → string text	**Печатает** текст *text* текущим шрифтом, начиная с указанных координат *(x,y)* верхнего левого угла описанного прямоугольника, ширина которого равна *width.* **Цвет** символов определяется текущим значением свойства *BrushColor*.
DrawEllipse() → double x → double y → double width → double height	Чертит **эллипс** с осями *width* и *height*. Координаты верхнего левого угла описанного прямоугольника находятся в заданной точке *(x,y).* **Цвет контура** определяется текущим значением свойства *PenColor*. **Толщина контура** определяется текущим значением свойства *PenWidth*.

DrawImage() → string imageName → double x → double y	**Печатает заданную картинку** *imageName* так, что координаты её верхнего левого угла находятся в заданной точке *(x,y)*, а размеры не изменяются.
DrawResizedImage() → string imageName → double x → double y → double width → double height	**Печатает заданную картинку** *imageName* так, что координаты её верхнего левого угла находятся в заданной точке *(x,y)*, а **размеры** - *width x height*
DrawLine() → double x1 → double y1 → double x2 → double y2	Чертит **отрезок прямой** с началом в точке *(x1,y1)* и концом в точке *(x2,y2)* . **Цвет линии** определяется текущим значением свойства *PenColor*. **Толщина линии** определяется текущим значением свойства *PenWidth*.
DrawRectangle() → double x → double y → double width → double height	Чертит **прямоугольник** размером *width x height*. Координаты верхнего левого угла прямоугольника находятся в точке *(x,y).* **Цвет контура** определяется текущим значением свойства *PenColor*. **Толщина контура** определяется текущим значением свойства *PenWidth*.
DrawTriangle() → double x1 → double y1 → double x2 → double y2 → double x3 → double y3	Чертит **треугольник** с заданными координатами вершин *(x1,y1)*, *(x2,y2)* и *(x3,y3)*. **Цвет контура** определяется текущим значением свойства *PenColor*. **Толщина контура** определяется текущим значением свойства *PenWidth*.
FillEllipse() → double x → double y → double width → double height	Рисует **эллипс** с осями *width* и *height*. Координаты верхнего левого угла описанного прямоугольника находятся в заданной точке *(x,y).*

	Цвет заливки определяется текущим значением свойства *BrushColor*.
FillRectangle() → double x → double y → double width → double height	Рисует **прямоугольник** размером *width x height*. Координаты верхнего левого угла прямоугольника находятся в точке *(x,y).* **Цвет** заливки определяется текущим значением свойства *BrushColor*.
FillTriangle() → double x1 → double y1 → double x2 → double y2 → double x3 → double y3	Рисует **треугольник** с заданными координатами вершин *(x1,y1)*, *(x2,y2)* и *(x3,y3)*. **Цвет** заливки определяется текущим значением свойства *BrushColor*.
GetColorFromRGB() → double red → double green → double blue ← string	Возвращает **цвет**, заданный цветовыми составляющими *red*, *green*, *blue*.
GetRandomColor() ← string	Возвращает **случайный цвет**.
GetPixel() → integer x → integer y ← string	Возвращает **цвет пикселя** с координатами *(x,y).*
SetPixel() → integer x → integer y → string color	Окрашивает **пиксель** с координатами *(x,y)* в цвет *color*.
Hide()	**Скрывает** *Графическое окно*.
Show()	**Показывает** *Графическое окно*.
ShowMessage() → string text	Показывает **диалоговое окно** с кнопкой *ОК* и *сообщением text*.

→ string title	**Заголовок** окна определяется значением параметра *title*.

События	**Описание**
KeyDown	Возникает при **нажатии на клавишу**.
KeyUp	Возникает при **отпускании клавиши**.
MouseDown	Возникает при **нажатии на кнопку мышки**.
MouseUp	Возникает при **отпускании кнопки мышки**.
MouseUp	Возникает при **перемещении мышки**.
TextInput	Возникает при **вводе текста с клавиатуры**.

Таблица <Класс *ImageList*>

Класс **ImageList** предоставляет методы для загрузки файлов изображений с диска компьютера или из Интернета в запущенное приложение.

{...} Класс находится в пространстве имён *Microsoft.SmallBasic.Library*.

Методы	**Описание**
LoadImage() → string fileNameOrUrl ← string	**Загружает картинку** с диска или из Интернета. *fileNameOrUrl* – полный путь к файлу. **Возвращает** имя картинки в списке.
GetWidthOfImage() → string imageName ← integer	**Возвращает ширину** в пикселях указанной картинки *imageName* в списке.
GetHeightOfImage()	**Возвращает высоту** в пикселях указанной

→ string imageName ← integer	картинки *imageName* в списке.

Таблица <Класс *Mouse*>

Класс **Mouse** предоставляет свойства и методы для работы с мышкой.

{...} Класс находится в пространстве имён *Microsoft.SmallBasic.Library*.

Свойства	Описание
IsLeftButtonDown ← bool	Возвращает *true*, если нажата **левая** кнопка мышки. В противном случае возвращает *false*.
IsRightButtonDown ← bool	Возвращает *true*, если нажата **правая** кнопка мышки. В противном случае возвращает *false*.
MouseX → integer PositionX ← integer	**Устанавливает курсор** мышки в точку, x-координата которой равна *PositionX*. **Возвращает** *x-координату* курсора мышки. Координаты мышки **экранные**, а не оконные!
MouseY → integer PositionY ← integer	**Устанавливает курсор** мышки в точку, y-координата которой равна *PositionY*. **Возвращает** *y-координату* курсора мышки.
Методы	**Описание**
HideCursor()	**Скрывает** курсор мышки.
ShowCursor()	**Показывает** курсор мышки.

Таблица <Класс *Network*>

Класс **Network** предоставляет методы для загрузки файлов любых типов из Интернета.

{...} Класс находится в пространстве имён *Microsoft.SmallBasic.Library*.

Методы	Описание
DownloadFile() → string url ← string	**Загружает файл** любого типа с заданной веб-страницы *url* во временную папку на диске компьютера и **возвращает** полный путь к нему.
GetWebPageContents() → string url ← string	**Возвращает** *HTML*-код заданной веб-страницы *url*.

Таблица <Класс *Shapes*>

Класс **Shapes** предоставляет собой коллекцию фигур разного типа и предоставляет методы для управления ими.

{...} Класс находится в пространстве имён *Microsoft.SmallBasic.Library*.

Методы	Описание
AddLine() → double **x1** → double **y1** → double **x2** → double **y2** ← string	**Добавляет** к коллекции фигур **отрезок** прямой с заданными координатами начальной и конечной точек (*x1,y1*) и (*x2,y2*). **Возвращает** имя фигуры.
AddTriangle () → double **x1** → double **y1** → double **x2** → double **y2** → double **x3** → double **y3**	**Добавляет** к коллекции фигур **треугольник** с заданными координатами вершин (*x1,y1*), (*x2,y2*) и (*x3,y3*). **Возвращает** имя фигуры.

← string	
AddEllipse() → double **width** → double **height** ← string	**Добавляет** к коллекции фигур **эллипс** с заданной шириной *width* и высотой *height*. Координаты верхнего левого угла описанного прямоугольника равны (*0,0*). **Возвращает** имя фигуры.
AddRectangle() → double **width** → double **height** ← string	**Добавляет** к коллекции фигур **прямоугольник** с заданной шириной *width* и высотой *height*. Координаты верхнего левого угла прямоугольника равны (*0,0*). **Возвращает** имя фигуры.
AddText() → string **text** ← string	**Добавляет** к коллекции фигур **строку** *text*. Координаты верхнего левого угла описанного прямоугольника равны (*0,0*). **Возвращает** имя фигуры.
SetText() → string **shapeName** → string **text**	Задаёт новый **текст** *text* для текстовой фигуры с именем *shapeName*.
AddImage() → string **imageName** ← string	**Добавляет** к коллекции фигур **картинку** по заданному пути *imageName.* Координаты верхнего левого угла картинки на экране равны (*0,0*). **Возвращает** имя фигуры.
Move() → string **shapeName** → double **x** → double **y**	**Перемещает** фигуру *shapeName* в точку с заданными координатами (*x,y*).

GetLeft() → string **shapeName** ← double	Возвращает **x-координату** верхнего левого угла фигуры.
GetTop() → string **shapeName** ← double	Возвращает **y-координату** верхнего левого угла фигуры.
HideShape() → string **shapeName**	Делает **невидимой** указанную фигуру *shapeName*.
ShowShape() → string **shapeName**	Делает **видимой** указанную фигуру *shapeName*.
RemoveShape() → string **shapeName**	**Удаляет** указанную фигуру *shapeName* из коллекции.
Animate() → string **shapeName** → double **x** → double **y** → double **duration**	**Плавно перемещает** указанную фигуру *shapeName* в новое положение, заданное координатами (*x,y*), за время *duration* (в миллисекундах).
Rotate() → string **shapeName** → double **angle**	**Поворачивает** указанную фигуру *shapeName* на заданный угол *angle* в градусах вокруг её центра относительно исходного положения.
SetOpacity() → string **shapeName** → double **level**	**Устанавливает прозрачность** указанной фигуры *shapeName* согласно значению *level* = 0..100.
GetOpacity() → string **shapeName** ← double	**Возвращает прозрачность** указанной фигуры *shapeName* (0..100).
Zoom() → string **shapeName** → double **scaleX** → double **scaleY**	**Масштабирует** указанную фигуру *shapeName*, изменяя её ширину и высоту в *scaleX* и *scaleY* раз, соответственно. Уменьшить фигуру можно не более чем в 10 раз, а увеличить – не более чем в 20.

Таблица <Класс *Timer*>

Класс **Timer** предоставляет свойства и методы для работы с системными часами.

Класс находится в пространстве имён *Microsoft.SmallBasic.Library*.

Свойства	Описание
Interval → integer interval ← integer	**Устанавливает** заданный интервал *interval* в миллисекундах между срабатываниями (тиками) таймера. **Возвращает** интервал в миллисекундах между срабатываниями таймера.
Методы	**Описание**
Pause()	**Приостанавливает** работу таймера.
Resume()	**Возобновляет** работу таймера.
События	**Описание**
Tick	**Возникает** при каждом срабатывании (тике) таймера.

Таблица <Класс *Turtle*>

Статический класс **Turtle** предоставляет свойства и методы для работы с *Черепашкой*.

{...} Класс находится в пространстве имён *Microsoft.SmallBasic.Library*.

Свойства	Описание
Angle → double angle ← double	**Устанавливает** заданный угол *angle* в градусах. Положительные углы отсчитываются против часовой стрелки, отрицательные – по часовой стрелке. При значении свойства *Angle = 0 Черепашка* смотрит вверх. **Возвращает** угол поворота *Черепашки* в градусах.
X → double x ← double	**Устанавливает** *Черепашку* в точку канвы, с *x*-координатой, равной *x*. **Возвращает** x-координату *Черепашки* в пикселях.
Y → double y ← double	**Устанавливает** *Черепашку* в точку канвы, с *y*-координатой, равной *y*. **Возвращает** *y*-координату *Черепашки* в пикселях.
Speed → integer speed ← integer	**Устанавливает** относительную **скорость** перемещения *Черепашки* в диапазоне от 1 (очень медленно) до 10 (очень быстро). **Возвращает** относительную скорость перемещения *Черепашки*.
Методы	**Описание**
Hide()	**Скрывает** *Черепашку*.
Show()	**Показывает** *Черепашку*.
Move() → double distance	**Перемещает** *Черепашку* на заданное расстояние *distance* в пикселях в направлении текущего поворота *Черепашки* (вперёд).

MoveTo() → double x → double y	**Перемещает** *Черепашку* в точку канвы, заданную координатами *(x,y).*
TurnLeft()	**Поворачивает** *Черепашку* на 90 градусов против часовой стрелки (влево) относительно текущего положения.
TurnRight()	**Поворачивает** *Черепашку* на 90 градусов по часовой стрелке (вправо) относительно текущего положения.
Turn() → double angle	**Поворачивает** *Черепашку* на заданный угол *angle* в градусах относительно текущего положения. Если угол *angle* положительный, то *Черепашка* поворачивается вокруг своей оси против часовой стрелки, если отрицательный – по часовой стрелке.
PenDown()	**Опускает карандаш** – *Черепашка* при перемещении чертит линии.
PenUp()	**Поднимает карандаш** – *Черепашка* при перемещении не чертит линии.

Таблица <Класс *Tortoise*>

Статический класс **Tortoise** предназначен для управления *Черепашкой*.

{...} Класс находится в библиотеке *SmallBasicFun*.

Методы	Описание
GetAngle() ←double	**Возвращает** текущий **угол поворота** *Черепашки.*
GetPenColor() ←string	**Возвращает цвет линий**.
GetPenWidth() ←double	**Возвращает толщину линий** в пикселях.
GetSpeed() ←int	**Возвращает** текущую **скорость** перемещения *Черепашки*.
GetX() ←double	**Возвращает *x*-координату** *Черепашки*.
GetY() ←double	**Возвращает *y*-координату** *Черепашки*.
Hide()	**Скрывает** *Черепашку*.
InstantSpeed() → bool goFast	Устанавливает **максимально возможную скорость** *Черепашки*, если значение параметра *goFast* равно *true*.
Show()	**Показывает** *Черепашку* на экране.
Move() → double distance	**Перемещает** *Черепашку* на заданное расстояние *distance* в пикселях в направлении текущего поворота *Черепашки* (вперёд).
MoveTo() → double x → double y	**Перемещает** *Черепашку* в точку канвы, заданную координатами *(x,y).*
Reset()	**Создаёт** *Черепашку* с параметрами по умолчанию.

SetAngle() → double angle	**Поворачивает** *Черепашку* в заданное направление. Если *angle = 0*, то *Черепашка* смотрит вверх. Если *angle = 180*, то *Черепашка* смотрит вниз.
SetOrientation() → integer x → integer y → double angle	Одновременно **перемещает** *Черепашку* в точку *(x,y)* и **поворачивает** её в заданное направление.
SetPenColor() → string color	**Устанавливает цвет линий**.
SetPenWidth() → double width	**Устанавливает толщину линий** в пикселях.
SetPosition() → integer x → integer y	**Переносит** *Черепашку* в указанную точку *(x,y).*
SetSpeed() → integer speed	**Устанавливает** относительную **скорость** перемещения *Черепашки* в диапазоне от 1 (очень медленно) до 10 (очень быстро).
SetX() → double x	**Задаёт** *Черепашке* ***x*-координату.**
SetY() → double y	**Задаёт** *Черепашке* ***y*-координату**.
Turn() → double angle	**Поворачивает** *Черепашку* на заданный угол *angle* в градусах относительно текущего положения. Если угол *angle* положительный, то *Черепашка* поворачивается вокруг своей оси против часовой стрелки, если отрицательный – по часовой стрелке.
PenDown()	**Опускает карандаш** – *Черепашка* при перемещении чертит линии.
PenUp()	**Поднимает карандаш** – *Черепашка* при перемещении не чертит линии.

Литература

[ГМ93]

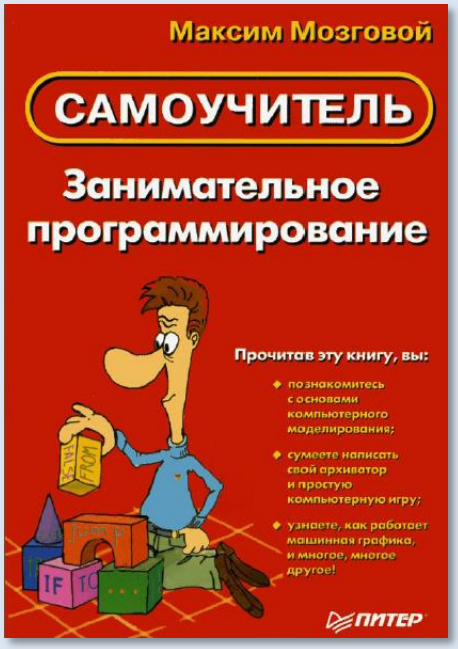
Гарднер Мартин

От мозаик Пенроуза к надёжным шифрам

М.: Мир, 1993. – 417 с.

ISBN 5-03-001991-X

[ММ05]



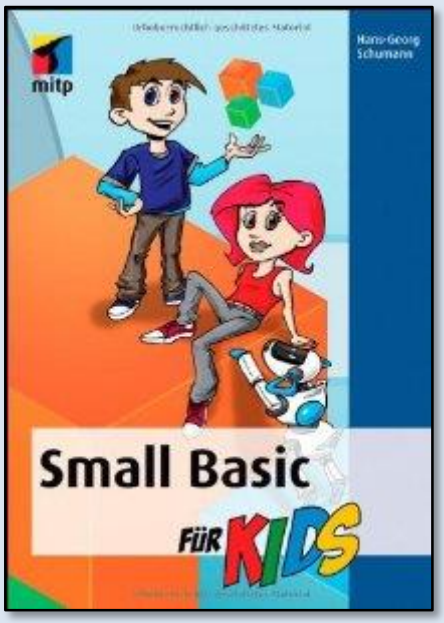
Максим Мозговой

Занимательное программирование

Питер, 2005. – 208 с.

Броуновское движение – стр. 20-23.
Черепашья графика – стр. 45 -48.

[SHG11]



Hans-Georg Schumann

Small Basic für Kids

mitp, 2011. – 272 c.

ISBN: 978-3826681882