

МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ
федеральное государственное бюджетное образовательное учреждение
высшего образования
«УЛЬЯНОВСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ»

В. В. Воронина

ПРОГРАММИРОВАНИЕ ИГР: АЛГОРИТМЫ И ТЕХНОЛОГИИ

Учебное пособие

Ульяновск
УлГТУ
2017

УДК 004.92 (075)
ББК 32.973-018.1я7
В75

Рецензенты:

Ведущий инженер-программист ФНЦП АО «НПО «Марс»,
канд. техн. наук Радионова Ю. А.

Заместитель главного конструктора АО «Ульяновское конструкторское бюро приборостроения», канд. техн. наук Черкашин С. В.

*Утверждено редакционно-издательским советом университета
в качестве учебного пособия*

Воронина, Валерия Вадимовна

В75 Программирование игр: алгоритмы и технологии : учебное
пособие / В. В. Воронина. – Ульяновск : УлГТУ, 2017. – 305 с.

ISBN 978-5-9795-1732-2

Учебное пособие рассматривает различные алгоритмы, используемые в игровых приложениях, а также описывает способы их реализации с использованием различных технологий работы с графикой. Пособие предназначено для студентов направлений 09.03.03 «Прикладная информатика», 09.03.04 «Программная инженерия», а также для студентов других направлений, изучающих дисциплины, связанные с разработкой игр и компьютерной графикой.

**УДК 004.92 (075)
ББК 32.973.2-018.2я7**

ISBN 978-5-9795-1732-2

© Воронина В. В., 2017
© Оформление. УлГТУ, 2017

СОДЕРЖАНИЕ

ВВЕДЕНИЕ	6
<i>Часть 1. Первые шаги в программировании игр</i>	9
Тема 1.1. Начало работы.....	11
Ввод-вывод данных	15
Переменные и значения.....	17
Задания к теме 1.1	26
Тема 1.2. Работа с логическим типом данных	27
Задания к теме 1.2	30
Тема 1.3. Разработка игры «Угадай число».....	31
Задания к теме 1.3	34
Тема 1.4. Разделение логики и интерфейса	34
Задание к теме 1.4	43
Тема 1.5. Графический интерфейс. Статика.....	44
Задания к теме 1.5	59
Тема 1.6. Графический интерфейс. Интерактив и уровни.....	61
Задания к теме 1.6	72
Тема 1.7. Разработка игры «Спасти принцессу».....	73
Сценарий игры	73
Подготовка изображений	77
Реализация игры.....	90
Оптимизация кода и сохранение статистики	107
Задания к теме 1.7	110
Тема 1.8. Одномерные массивы.....	110
Цикл for	113
Типовые алгоритмы работы с одномерными массивами	114

Применение массивов в игре «Спасти принцессу».....	118
Задания к теме 1.8	119
Тема 1.9. Двумерные массивы	124
Типовые алгоритмы работы с двумерным массивом.....	124
Задания к теме 1.9	130
Тема 1.10. Разработка игры «морской бой»	134
Начало разработки	134
Расстановка кораблей компьютером	140
Расстановка кораблей человеком	145
Программирование ход человека	150
Ход компьютера	155
Задания к теме 1.10	159
Тема 1.11. Разработка игры «Линии».....	160
Волновой алгоритм	162
Реализация волнового алгоритма	165
Разработка модуля логики игры	172
Задания к теме 1.11	182
Часть 2. Разработка игр на WPF	183
Тема 2.1. Введение в WPF	185
Задание к теме 2.1	196
Тема 2.2. Графика, конверторы и свойства зависимости.....	196
Задания к теме 2.2	215
Тема 2.3. Разработка игры «Линии».....	215
Задания к теме 2.3	238
Часть 3. Работа с технологией OpenGL.....	239
Тема 3.1. Основы работы с OpenGL.....	241
Введение в SharpGL	242

Отрисовка примитивов и преобразования координат	245
Задания к теме 3.1	263
Тема 3.2. Объемные объекты, освещение и трансформации	263
Задания к теме 3.2	275
Тема 3.3. Текстуры и видеоролики.....	276
Работа с текстурами.....	280
Сохранение сцены в bmp-файл.....	286
Видеоролики.....	289
Задания к теме 3.3	303
ЗАКЛЮЧЕНИЕ	304
БИБЛИОГРАФИЧЕСКИЙ СПИСОК.....	305

ВВЕДЕНИЕ

В последние несколько лет игровая индустрия развивается ошеломляющими темпами. По оценкам экспертов по прибыли она уже готова соперничать с киноиндустрией. Причем высокую прибыль приносят не только проекты, выпущенные известными издателями, но и так называемые инди-игры. Согласно Википедии [1]: «Инди-игра (англ. Indie game, от англ. independent video game – «*независимая компьютерная игра*») – компьютерная игра, созданная отдельным разработчиком или небольшим коллективом без финансовой поддержки издателя компьютерных игр. Распространение осуществляется посредством каналов цифровой дистрибуции. Масштаб явлений, связанных с инди-играми, ощутимо возрастает со второй половины 2000-х годов, в основном ввиду развития новых способов онлайн-дистрибуции и средств разработки». Одной из причин этого явления также можно назвать врожденную человеческую тягу к творчеству. Создание любой игры начинается с идеи, которую средства разработки помогают реализовать, а средства дистрибуции – донести до людей. Гениальная мысль может посетить каждого, поэтому задача освоения базовых технологий и алгоритмов создания игр для многих является актуальной. Именно на ее решение направлено данное пособие.

С точки зрения пользователя в любой игре всегда есть две составляющих: сюжет и внешний вид. Причем последнее чаще важнее. С точки зрения программиста – логика и интерфейс взаимодействия с пользователем. Таким образом, успех игры во многом зависит от выбора правильной технологии реализации интерфейса с учетом целевой платформы и назначения игры. Кроме того, без знания хотя бы базовых основ программирования реализация сколько-нибудь сложной игры дело затруднительное. Конечно, существуют программные системы, позволяющие вам конструировать игры без написания кода, но тогда ваши возможности будут ограничены лишь возможностями

этих сред. В связи с этим в данном пособии мы рассмотрим разработку игр с использованием нескольких графических технологий, а также познакомимся с основами программирования, требующимися для реализации тех или иных игр. Ниже приведено краткое описание рассматриваемых технологий:

GDI+ – графическая технология, используемая в проектах WindowsForms. Недостатки: бедная графика, низкая скорость анимации, низкая производительность, единственный способ использования технологии – проекты WindowsForms. Достоинства: простота использования. Возможное использование в разработке игр: реализация учебных игр с целью отработки типовых алгоритмов и обучение разделения логической и графической составляющих игр. Целевая платформа: Windows с установленным .Net Framework.

OpenGL – спецификация, определяющая независимый от языка программирования программный интерфейс для написания приложений, использующих двумерную и трехмерную компьютерную графику [1]. Недостатки: определенная сложность освоения данной технологии, сложность интеграции с некоторыми языками программирования и средствами разработки. Достоинства: кроссплатформенность, высокая производительность, огромные возможности в плане анимации, а также работы с двухмерной и трехмерной графикой. Возможное использование в разработке игр: предназначена для разработки игр на разных платформах. Целевые платформы: разные.

DirectX – набор API, разработанных для программирования графики под Microsoft Windows. Достоинства: высокая производительность, огромные возможности в плане анимации, а также работы с двухмерной и трехмерной графикой. Возможное использование в разработке игр: предназначена для разработки игр на разных платформах. Целевые платформы: основная – Windows, но возможна реализация игр и под другие.

В рамках данной книги мы рассмотрим работу со следующими технологиями. GDI+ мы используем в рамках освоения (или повторения) необходимого теоретического материала по программирова-

нию, который нам понадобится для разработки игр. Технологию OpenGL через работу с его .Net-оберткой SharpGL. Работу же с технологией DirectX мы рассмотрим косвенно: через WPF.

WPF – графическая презентационная подсистема .Net Framework, предназначенная для создания визуально привлекательных пользовательских интерфейсов. Недостатки: определенная сложность освоения данной технологии, единственный язык программирования – C#. Достоинства: высокая производительность, огромные возможности в плане анимации, а также работы с двухмерной и трехмерной графикой. Возможное использование в разработке игр: упрощение освоения технологии через применение ее в разработке игр. Целевая платформа: Windows с установленным .Net Framework.

Пособие разделено на три части. Часть «Первые шаги в программировании игр» полностью ориентирована на работу с GDI+ и преследует цель иллюстрации применения базовых конструкций и алгоритмов в разработке игр различной сложности. Вторая часть – «Разработка игр на WPF» рассматривает возможности WPF для разработки привлекательного графического интерфейса под Windows. Кроме того, в данной части рассматривается применение более сложных языковых конструкций. Третья часть – «Работа с технологией OpenGL» рассматривает работу с указанной технологией, а также с технологией платформенного вызова для создания видеороликов.

Каждая часть разделена на темы, после которых представлен перечень заданий для самостоятельной работы. При работе с книгой рекомендуется воспроизводить примеры, описанные в основных темах, оптимизируя и дорабатывая представленный код, а также выполнять представленные после тем задания.

Часть 1. Первые шаги в программировании игр

*В данной части рассматривается
работа с технологией GDI+
средствами среды Visual Studio,
а также базовые конструкции
и алгоритмы программирования,
необходимые разработчикам игр.
Технология GDI+ была выбрана
как одна из самых простых
в понимании и использовании.*

ТЕМА 1.1. НАЧАЛО РАБОТЫ

Рассмотрим разработку простого развлекательного Windows-приложения, которое не требует никаких структур данных или алгоритмов. Пользователь будет вводить свое имя, а приложение будет с ним здороваться, выводя сообщение: «Привет, {Имя}!».

Для создания приложения Windows Forms в Microsoft Visual Studio необходимо сделать следующее. В меню «File» («Файл») выбрать пункт «New» («Новый») и в нем выбрать подпункт «Project» («Проект»). Далее в диалоговом окне, показанном в разделе Project types необходимо выбрать пункт Visual C# и подпункт Windows. В разделе Templates выбрать Windows Forms Application. После этого у нас на экране появится основная форма, как показано на рисунке 1.1.

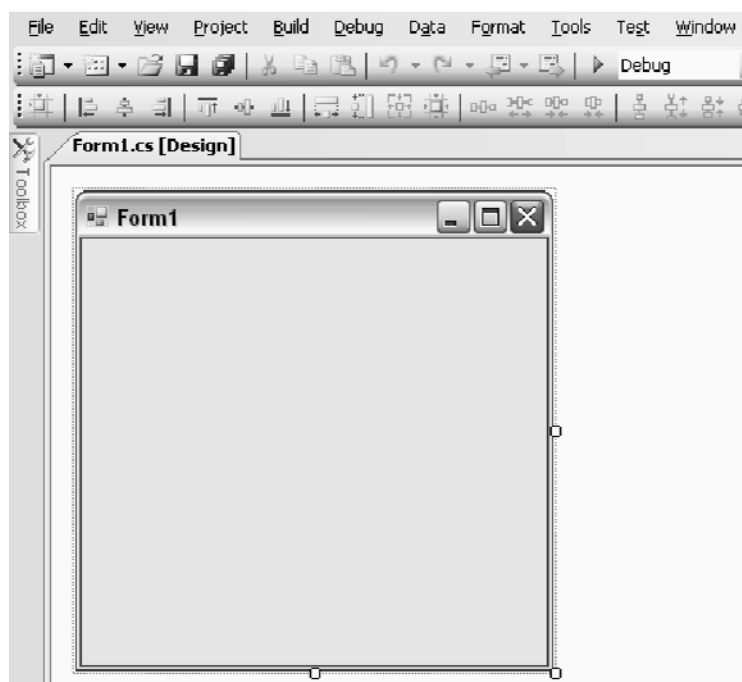
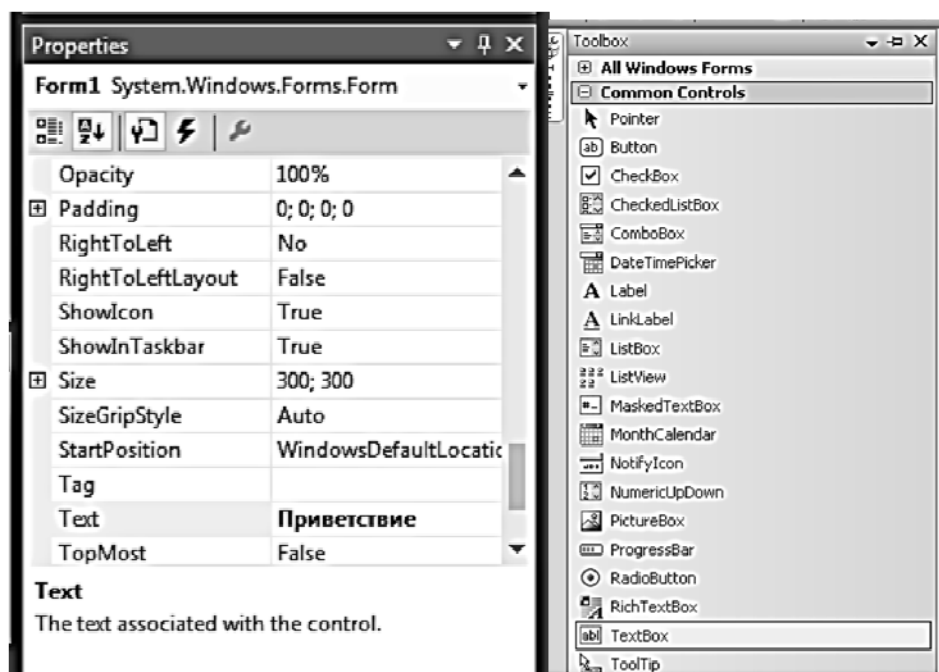


Рисунок 1.1. Основная форма программы

Если мы нажмем правой кнопкой мыши на форму и выберем в контекстном меню Properties, то перейдем к окошку, изображен-

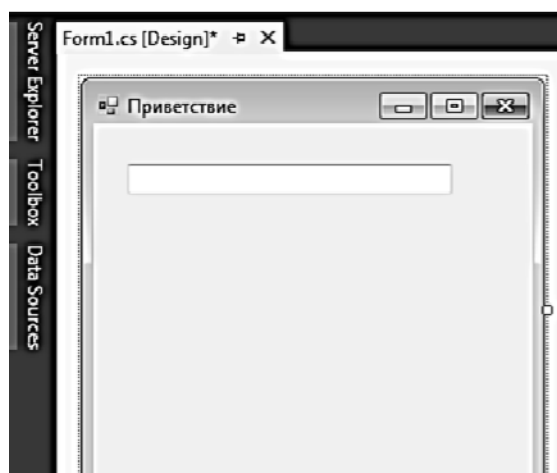
ному на рисунке 1.2(а). В нем отображаются все свойства формы. Зададим в поле Text заголовок формы «Приветствие».

Теперь нам понадобится элемент для ввода имени. Находим слева вкладку Toolbox (рисунок 1.2(б)) и выбираем в разделе CommonControls элемент TextBox. Размещаем его на форме путем двойного щелчка или прямого перетаскивания. После этого форма будет иметь вид, представленный на рисунке 1.2(в).



(а)

(б)



(в)

Рисунок 1.2. Свойства и промежуточный вид формы

Для удобства пользователя, разместим элемент Label на форме. Шрифты у элемента Label так же настраиваются в Properties (Font), текст – там же в поле Text. Как мы видим, окошко свойств содержит разнообразные параметры для каждого элемента. Рекомендуется поэкспериментировать с ними самостоятельно.

Далее разместим на форме кнопку(button1), при нажатии на которую будет выполняться работа программы. В итоге форма приобретет вид, представленный на рисунке 1.3:

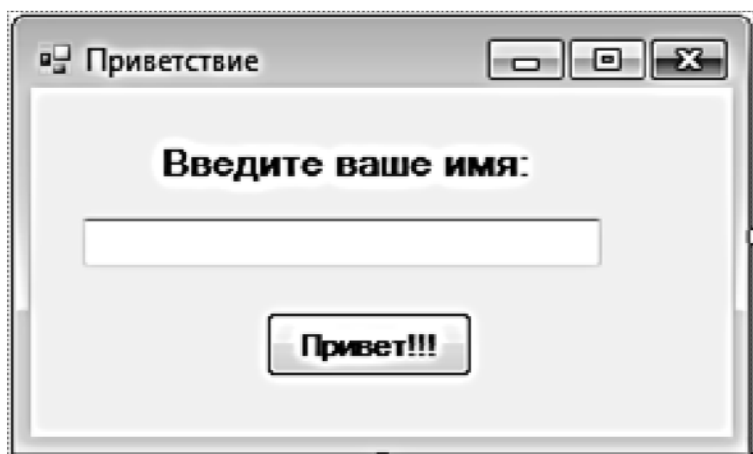


Рисунок 1.3. Итоговый вид формы

Теперь мы можем запустить нашу программу. Запуск выполняется нажатием F5 или через значок-пуск (Start) на панели меню (рисунок 1.4).

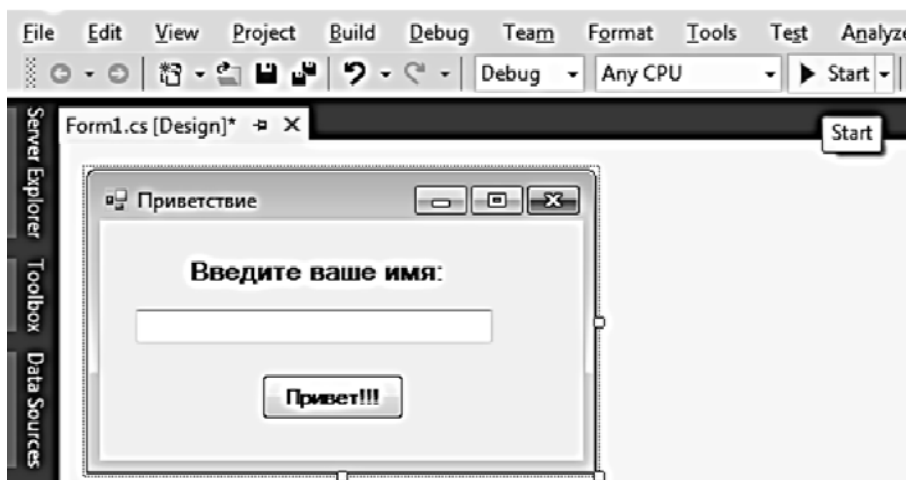


Рисунок 1.4. Запуск программы

Теперь мы можем вводить свое имя в поле ввода, нажимать на кнопку, сворачивать и разворачивать форму, но никаких других действий программа выполнять не будет. Для того, чтобы она смогла что-то делать, нужно написать код на языке программирования C#, который будет выполняться при нажатии на кнопку.

Код, обрабатывающий нажатие кнопки `button1`, размещается в файле `Form1.cs` после двойного нажатия мышкой на эту кнопку. Если вы все сделали верно, то по двойному щелчку вы должны автоматически перейти в окно редактирования кода, внешний вид которого представлен на рисунке 1.5.

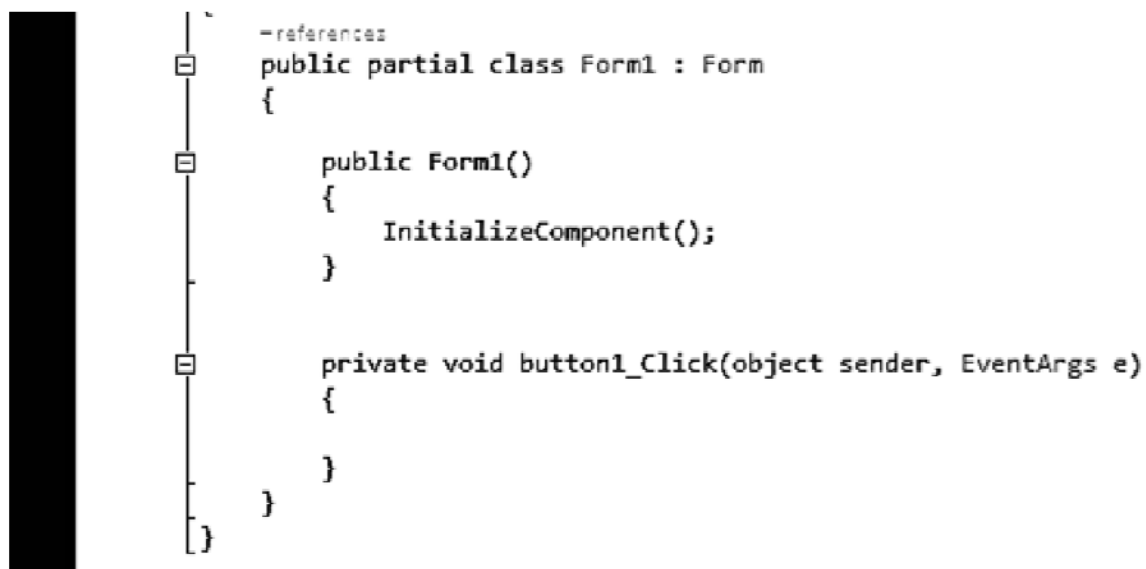


Рисунок 1.5. Окно написания кода

Двойной щелчок на кнопке создал метод с заголовком `button1_Click`. Этот метод называется «методом-обработчиком события нажатия на кнопку». Любой элемент формы, как и она сама, может реагировать на множество событий, то есть выполнять код в ответ на какие-либо действия пользователя. Полный перечень событий элемента представлен в окошке `Properties`, в режиме отображения событий (при нажатой пиктограмме, изображающей молнию). Прикрепление соответствующего метода к событию может быть осуществлено через это окно, как отображено на рисунке 1.6. Если метода обработки у нас нет, то мы просто дважды кликаем левой кнопкой мыши на

пустое поле рядом с именем события, реакцию на которое хотим прописать, и Visual Studio автоматически сгенерирует и привяжет метод. Если же метод уже был ранее написан, мы выбираем его из списка, кликнув по кнопке-стрелочке рядом с именем события.

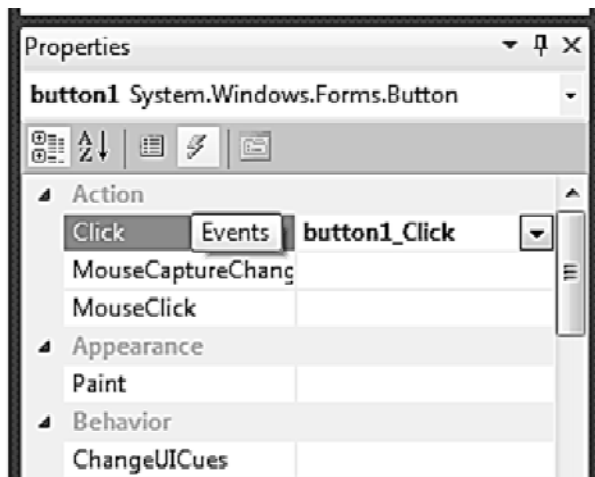


Рисунок 1.6. Привязка методов к событиям

Теперь, чтобы научить нашу программу делать то, что мы планировали (выводить приветствие), нам необходимо разобраться с такими понятиями, как «ввод данных» и «вывод данных».

Ввод-вывод данных

Ввод данных – получение и распознавание программой информации от пользователя. Вывод данных – отображения для пользователя программной информации. Для ввода информации мы будем использовать специальный элемент формы `TextBox`. Информацию, которую внес в него пользователь, мы можем получить, обратившись к его характеристике – `Text`. У каждого размещенного на форме элемента есть имя. Пусть у нас будет `TextBox` с именем `textBox1`. Имя элемента можно посмотреть в окошке `Properties` либо в свойстве `Name`, либо под заголовком `Properties` (выделенный жирным текст). Например, на рисунке 1.6. мы видим имя `button1`. В программном коде к элементам мы всегда обращаемся по имени.

Для вывода данных можно воспользоваться двумя способами: характеристикой `Text` элемента `Label`, размещаемого на форме или же

через окошко сообщений, вызываемого командой `MessageBox.Show`. Например, мы хотим вывести окошко сообщений с текстом «Привет, {Имя}!», где {Имя} вводится пользователем в `textBox`. Код, реализующий это, будет следующим (листинг 1.1).

Листинг 1.1

```
MessageBox.Show("Привет,"+textBox1.Text+"!");
```

Если же мы хотим вывести то же самое в `Label` с именем `label1`, то необходимо написать другой код (листинг 1.2).

Листинг 1.2

```
label1.Text="Привет,"+textBox1.Text+"!";
```

Попробуйте прописать в обработку нажатия на кнопку строчку кода из листинга 1.1. Затем запустите программу, введите свое имя и нажмите на кнопку. Если вы все сделали верно, то на экране должны увидеть сообщение, отображенное на рисунке 1.8.

Обратите внимание!

- При написании кода *Visual Studio* выдает список подсказок, когда вы ставите точку (рисунок 1.7).
- Необходимо точное написание слов: очень важны маленькие и большие буквы, скобочки, запятые и точки с запятыми.
- Если того, что вы хотите написать, нет в списке подсказок, значит вы пишете что-то не то.

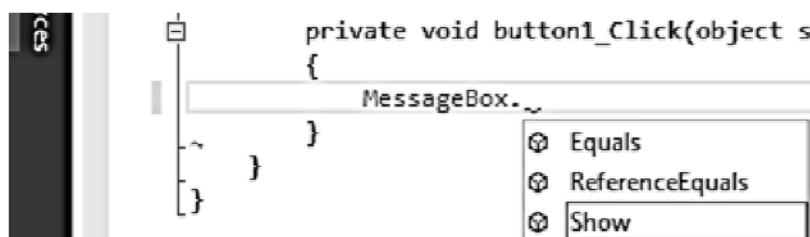


Рисунок 1.7. Подсказки системы

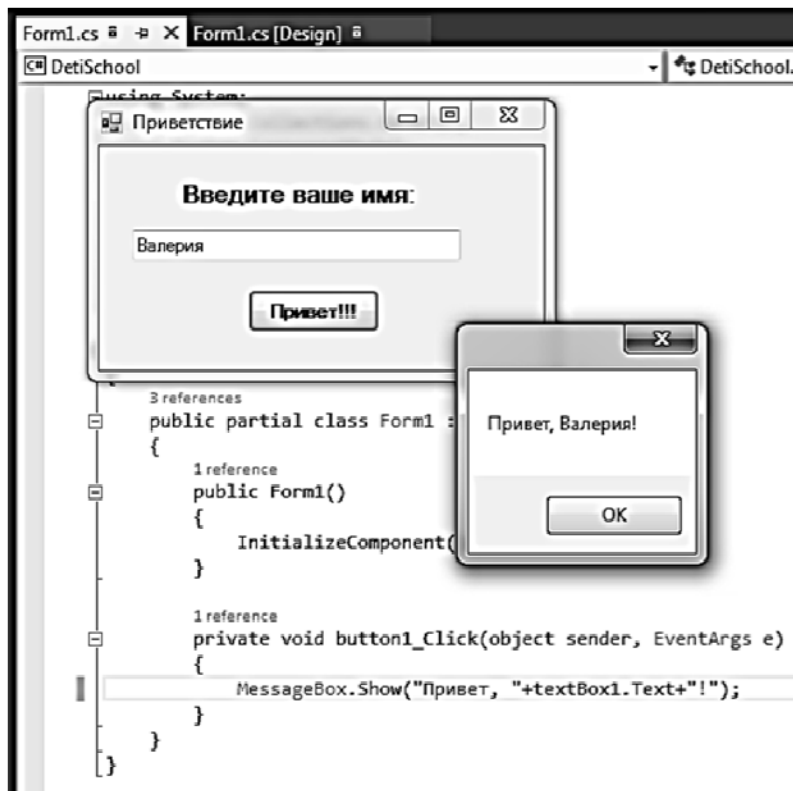


Рисунок 1.8. Работа программы

Теперь давайте разберемся с самой строчкой кода: зачем там кавычки, почему мы используем знаки «плюс» и что такое `textBox1.Text` с точки зрения программного кода. Однако, чтобы разобраться с этими вопросами, нам нужно познакомиться с такими понятиями, как «переменные» и «значения», рассмотрев самую простую структуру данных, используемую в программировании.

Переменные и значения

Переменная – элемент программы, имеющий тип и имя. Предназначен для хранения значений. В зависимости от типа переменной в нее можно записать определенную информацию. Например, в переменной числового типа хранятся только числа, в переменной строкового типа – строки, состоящие из любых символов. Чтобы использовать переменную, ее нужно создать (объявить), указав тип и имя. После этого через знак равно в нее можно занести значения. В C# существует множество различных типов данных, но в настоящем посо-

бии мы рассмотрим только те, которые нам будут необходимы: целый числовой, дробный числовой, строковый и логический.

В листинге 1.3. приведен пример объявления числовой переменной с именем «а» и занесения в нее числа 0.

Листинг 1.3

```
int a=0;
```

Листинг 1.4. – пример объявления строковой переменной с именем «s» и занесения в нее строки «мама».

Листинг 1.4

```
string s="мама";
```

Строки можно складывать. Например, объявим три строковые переменные. В первую запишем слово «мама», во вторую – «папа», а в третью – результат сложения первых двух строк (листинг 1.5). После выполнения этого кода из листинга 1.5 переменная s3 будет иметь значение (будет равна) «мамапапа».

Листинг 1.5

```
string s1="мама"; string s2="папа"; string s3=s1+s2;
```

Если мы хотим, чтобы между словами был вставлен пробел, то операцию сложения нужно переписать так, как показано в листинге 1.6. В результате переменная s3 будет равна «мама папа».

Листинг 1.6

```
string s1="мама"; string s2="папа"; string s3=s1+" "+s2;
```

Разберемся с нашей строчкой: «Привет,”+textBox1.Text+”!”». Здесь мы видим, что «Привет,» и «!» – значения типа string, а textBox1.Text – переменная типа string.

Теперь рассмотрим три важных правила именования переменных.

Во-первых, имена должны быть уникальны, при этом маленькие и большие буквы считаются разными (Sasha и sasha – две разные переменных).

Во-вторых, имена переменных должны начинаться с латинской буквы или знака подчеркивания.

В-третьих, имя переменной должно отражать то, для чего она используется. Например, если вы пишете программу для сложения двух чисел можно назвать переменные `chislo1` и `chislo2`.

Обратите внимание!

- *В выражениях строковые значения всегда записываются в двойных кавычках, а переменные всегда записываются по имени, без указания типа.*
- *Тип переменной указывается только один раз: при ее объявлении, то есть при самом первом появлении ее в тексте программы.*
- *Для иллюстративных примеров допускается использование переменных, чьи имена не несут смысловой нагрузки. Но в реальных программах это не допустимо!*

Теперь рассмотрим использование переменных при вводе-выводе данных. Хотя `textBox1.Text` по сути является переменной, использовать ее не всегда неудобно. Поэтому рассмотрим, как мы можем сохранить введенные данные в другие переменные. Например, считаем введенную пользователем информацию в строковую переменную `name` (листинг 1.7).

Листинг 1.7

```
string name=textBox1.Text;
```

Если же мы предполагаем, что пользователь ввел число, и хотим сохранить его в переменную соответствующего типа, то нам необходимо выполнить преобразование типов, как показано в листинге 1.8. Здесь в только что созданную числовую переменную «а» считывается число, которое ввел пользователь.

Листинг 1.8

```
int a=Convert.ToInt32(textBox1.Text);
```

В случае, если пользователь ввел вместо числа некую последовательность символов, которую нельзя преобразовать к указанному типу, то программа выдаст исключение (рисунок 1.9).

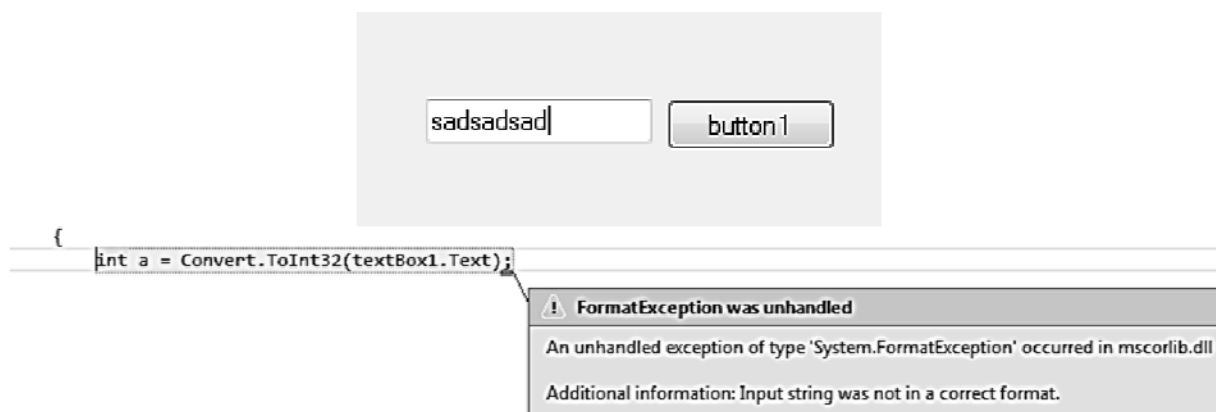


Рисунок 1.9. Пример некорректного ввода данных

Чтобы этого избежать, участки кода, где происходит преобразование типов, необходимо заключать в оператор try-catch, как показано в листинге 1.9. Тогда при возникновении исключительной ситуации программа продолжит работать, просто выдав корректное сообщение об ошибке.

Листинг 1.9

```
try { int a=Convert.ToInt32(textBox1.Text);}
catch(Exception ex) { MessageBox.Show(ex.Message);}
```


Теперь рассмотрим вывод информации на экран. Для этого нам необходимо всегда преобразовывать ее к строковому типу, так как отображаем мы только строки. Допустим, мы хотим вывести в окошко сообщений значение суммы двух считанных с клавиатуры чисел. Реализующий это код представлен в листинге 1.10.

Листинг 1.10

```
int a=Convert.ToInt32(textBox1.Text); int b=Convert.ToInt32(textBox2.Text);  
int c=a+b; MessageBox.Show(Convert.ToString(c));
```

Обратите внимание!

- Пользователь всегда вводит информацию в строковом виде, поэтому напрямую сохранять ее можно только в строковые переменные.
- Чтобы считать с клавиатуры число, мы должны преобразовать его из строкового типа данных в числовой.
- Если мы хотим вывести значение переменной числового типа, мы должны преобразовать ее к строковому типу.
- Преобразование в строку можно выполнить без использования *Convert*, написав: *MessageBox.Show(c.ToString());*
Но преобразование в другие типы необходимо осуществлять через эту конструкцию.

Иногда возникают ситуации, когда нам нужно не запрашивать у пользователя число, а просто записать в переменную какое-то случайное значение. Например, пусть в числовую переменную «а» нам нужно записать случайное число из диапазона от –10 до 10. Решается это с использованием генератора случайных чисел, пример работы с которым приведен в листинге 1.11.

Листинг 1.11

```
Random r=new Random(); int a=r.Next(-10,11);
```

Помимо простого сохранения значений, а также сложения с переменными числового типа мы можем выполнять и иные действия: вычитать, умножать, делить, находить остаток от деления одной на другую. Ниже приведен список этих операций:

- Операция `+` возвращает сумму двух чисел. Например, у пусть нас есть переменные `int a=7; int b=3;int c=0;` Запишем в переменную сумму `a` и `b`: `c=a+b;` значение переменной `c=10`.
- Операция `-` возвращает разность двух чисел. Например, у пусть нас есть переменные `int a=7; int b=3;int c=0;` Запишем в переменную разность `a` и `b`: `c=a-b;` значение переменной `c=4`.
- Операция `*` возвращает произведение двух чисел. Например, у пусть нас есть переменные `int a=7; int b=3;int c=0;` Запишем в переменную произведение `a` и `b`: `c=a*b;` значение переменной `c=21`.
- Операция `/` возвращает результат деления первого числа на второе. Например, у пусть нас есть переменные `int a=6; int b=3;int c=0;` Запишем в переменную результат деления первого числа на второе: `c=a/b;` значение переменной `c=2`. Обратите внимание, здесь говорится только о делении нацело.
- Операция `%` возвращает остаток от деления одного числа на другое. Например, у пусть нас есть переменные `int a=7; int b=3;int c=0;` Запишем в переменную `c` остаток от деления `a` на `b`: `c=a%b;` значение переменной `c=1`. При `a=4` и `b=2` `a%b` даст `0`.
- Операция `++` увеличит переменную на `1`. Например, у пусть нас есть переменная `int a=7;` Код `a++;` запишет в нее `8`.
- Операция `--` уменьшит переменную на `1`. Например, у пусть нас есть переменная `int a=7;` Код `a--;` запишет в нее `6`.

Рассмотренный в примерах тип `int` используется только для хранения целочисленных переменных. Если же нам нужно поработать с дробными значениями, то мы должны использовать типы `double` или `float`. Различие между ними рассмотрим непосредственно при использовании, пока же просто отметим, что работать с ними можно также

как с типом `int`. В листинге 1.12 приведен пример считывания дробного числа.

Обратите внимание!

- Если нам нужно выполнить операцию с переменной и записать результат в нее же, то допускается сокращенная форма приведенных выше операторов. Например, код `a=a+4`; можно упростить, написав `a+=4`;

Листинг 1.12

```
double a=Convert.ToDouble(textBox1.Text);
```

Обратите внимание!

- При считывании числа мы ожидаем, что пользователь введет его, отделив дробную часть запятой. Но в программном коде значения `double` или `float` записываются с использованием точки.

Перейдем к следующему типу данных, а именно, к логическому. Он обозначается ключевым словом `bool`. Переменная этого типа может хранить только два значения `true` (истина, да) и `false` (ложь, нет). Для получения от пользователя логического значения используется элемент `radioButton` (рисунок 1.10).

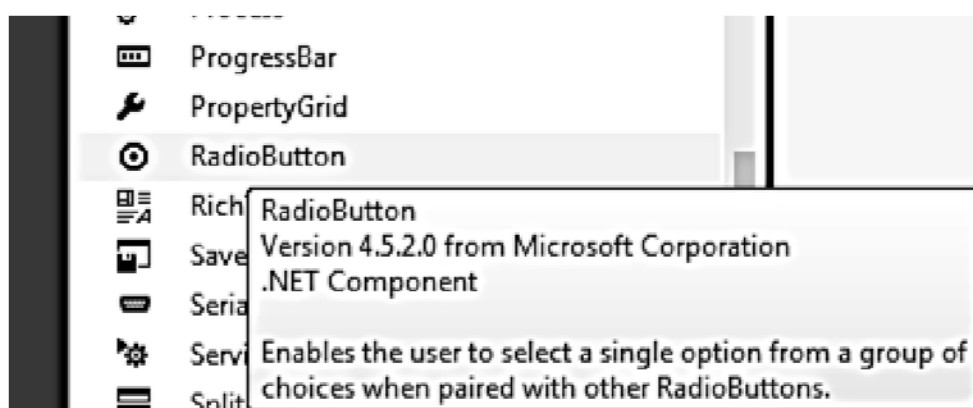


Рисунок 1.10. Расположение `radioButton` в `ToolBox`

Считывание значения в переменную из этого элемента происходит следующим образом (листинг 1.13)

Листинг 1.13

```
bool f=radioButton1.Checked;
```

При этом, если элемент пользователем выбран (в нем стоит «точка», как показано на рисунке 1.11), то в переменную запишется значение true, иначе – false. Для изменения подписи radioButton, отображаемой рядом с точкой мы используем его свойство Text.

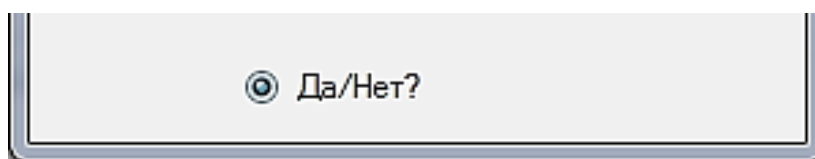


Рисунок 1.11. Пользователь выбрал значение «Да»

Аналогично можно работать с элементом checkBox. Его отличие от предыдущего проявляется лишь при размещении на форме нескольких элементов. Если мы размещаем несколько radioButton, они автоматически связываются в группу таким образом, что выбран может быть только один. Для получения нескольких групп связанных элементов их необходимо размещать не напрямую на форму, а использовать дополнительно Panel или GroupBox. Элементы checkBox такого ограничения не имеют – их можно выделять как угодно.

Логические переменные используются при построении достаточно сложных программных алгоритмов и для их обработки нужны специальные операторы. Поэтому работу с типом данных bool мы рассмотрим в следующей теме. Ни складывать, ни вычитать переменные этого типа нельзя.

Мы рассмотрели основные типы данных, используемые в программировании. Теперь рассмотрим еще несколько моментов, связанных с организацией взаимодействия пользователя и приложения. Иногда возникает необходимость выполнения какого-то действия при

старте программы. Например, перед открытием формы должно вывестись MessageBox с приветствием. Для реализации этого необходимо сделать две вещи. Во-первых, двойным щелчком на форме перейти в окно написания кода, в котором автоматически создастся код для события Form1_Load, представленный на рисунке 1.12. Во-вторых, внутри фигурных скобок у Form1_Load необходимо прописать вызов MessageBox.

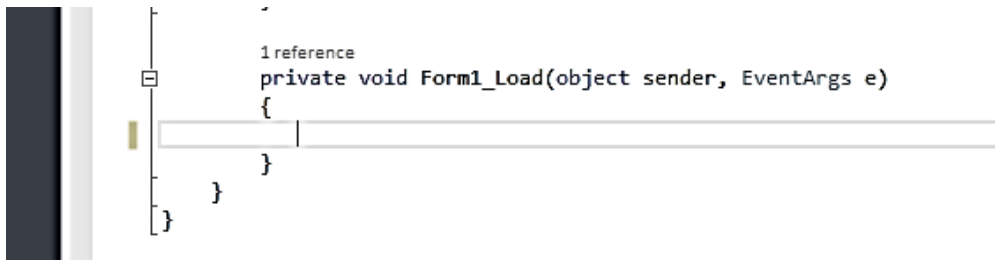


Рисунок 1.12. Код, выполняемый при запуске формы

Разберемся с таким важным вопросом как «область жизни переменных». Под областью жизни созданной переменной понимается участок кода, в котором эту переменную можно использовать. Обычно он ограничивается фигурными скобками. То есть, если вы объявили переменную внутри фигурных скобок Form1_Load, то в коде обработки нажатия на кнопку ее уже не будет видно. Для того чтобы можно было использовать переменную и в Form1_Load, и в button1_Click, ее необходимо объявить за пределами фигурных скобок любого из этих участков кода. Например, как показано на рисунке 1.13.

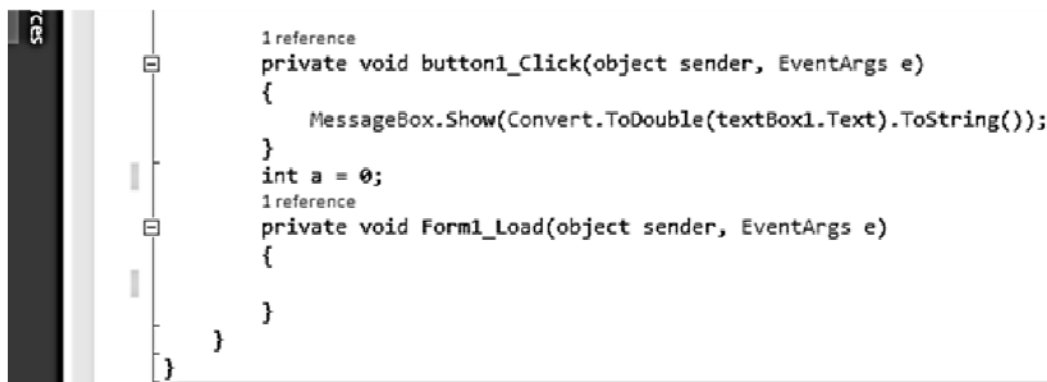


Рисунок 1.13. Объявление глобальной числовой переменной «а»

Такие переменные называются «глобальными». В C# же они называются «полями», принадлежащими классу, в котором объявлены. Например, случай, представленный на рисунке 1.13, – объявления поля «а» у формы.

Мы рассмотрели основы, необходимые для начала написания программ. Для закрепления материала вам предлагается выполнить несколько практических заданий, после чего перейти к изучению следующей темы.

ЗАДАНИЯ К ТЕМЕ 1.1

Реализуйте следующие программы:

1. Пользователь вводит 2 числа. Программа считает разность этих чисел и выводит на экран.
2. Пользователь вводит 3 числа. Программа считает сумму первого и третьего и из этой суммы вычитает второе. Затем выводит результат на экран.
3. Пользователь вводит 4 числа. Вывести в MessageBox остаток от деления суммы первого и второго числа на сумму третьего и четвертого.
4. Пользователь вводит 3 числа. Программа считает сумму первого и третьего и считает остаток от деления этой суммы на второе число. Затем выводит результат на экран.
5. Пользователь вводит 3 числа. Программа считает сумму первого и второго и из этой суммы вычитает третье. Затем выводит результат на экран.
6. При запуске формы генерируется случайное число и выводится в MessageBox.
7. Вывести на экран следующее сообщение: «Меня зовут <Имя пользователя>. Мне <Возраст пользователя> лет. Когда мне будет {Возраст пользователя+10} лет я хочу быть успешным Програм-

мистом.» (Примечание: В угловых скобках данные вводит пользователь, в фигурных – считает сама программа).

8. Сделать простой калькулятор на 4 арифметические действия с участием двух чисел.

9. На форме разместить два элемента `textBox`. Сделать так, чтобы по нажатию кнопки введенные в них значения поменялись местами. То есть, если в первый `textBox` пользователь ввел число 2, а во второй – 3, то после нажатия кнопки первый `textBox` должен содержать 3, а второй – 2.

10. Объявлены две числовые переменные `a` и `b`. Необходимо сделать так, чтобы без объявления других переменных в результате работы алгоритма значения переменных поменялись местами. Например, изначально $a = 2$, $b = 3$, то после обработки переменные стали $a = 3$, $b = 2$. Применять можно только операции присваивания и арифметические операции с числами и значениями переменных.

ТЕМА 1.2. РАБОТА С ЛОГИЧЕСКИМ ТИПОМ ДАННЫХ

Для работы со значениями типа `bool` используется условный оператор. Он реализует выполнение определенных команд при условии, что некоторое логическое выражение или переменная (условие) принимает значение «истина» (`true`). В большинстве языков программирования условный оператор начинается с специального ключевого слова `if`.

Встречаются следующие формы условного оператора: условный оператор с одной ветвью (листинг 1.14) и условный оператор с двумя ветвями (листинг 1.15). При выполнении первого оператора (как в показано листинге 1.14) вычисляется условие, и если оно истинно, то выполняются команды, в противном случае выполнение программы продолжается со следующей за условным оператором команды. Во втором же случае (листинг 1.15) при истинности условия выполняются команды¹ при ложности — команды².

Листинг 1.14

```
if (условие) { команды; }
```

Листинг 1.15

```
if (условие) { команды1;} else { команды2 ;}
```

Обратите внимание!

- *Определить необходимость использования условного оператора очень просто. Как только в формулировке задачи появляется ключевое слово «если», значит необходим будет оператор if. Или когда(если) задачу можно переформулировать, используя слово «если».*

При необходимости проверить последовательно несколько условий, возможно вложение условных операторов друг в друга (листинг 1.16). В этом случае условия будут проверяться последовательно, и как только встретится истинное, будет выполнен соответствующий набор команд и исполнение перейдет к команде, следующей за условным оператором. Если ни одно из условий не окажется истинным, выполняются командыN из ветви else.

Листинг 1.16

```
if (условие1 ) {команды1;}  
else if (условие2){ команды2;}  
...  
else if (условиеN-1){ командыN-1;}  
else {командыN ;}
```

Для записи условий с числовыми переменными обычно используются следующие операции:

- > – проверяет, больше ли первое число второго.
- < – проверяет, меньше ли первое число второго.

== – проверяет, равны ли два числа(или две строки).

!= – проверяет два числа на неравенство друг другу.

Допустим, мы решаем следующую задачу: пользователь вводит два числа, а программа выводит сообщение о том, какое число больше или, если числа равны, то сообщение об этом. Код, реализующий это, представлен в листинге 1.17.

Листинг 1.17

```
int a=Convert.ToInt32(textBox1.Text);
int b=Convert.ToInt32(textBox2.Text);
if (a>b)
{MessageBox.Show(«а больше b»);}
else
{ if (a==b)
  { MessageBox.Show(«а равно b»);}
  else
  {MessageBox.Show(«а меньше b»);}
}
```

Обратите внимание!

- При записи условия сравнения используются два стоящих подряд знака равенства. Например, условие равенства переменных «a» и «b» мы должны записать так: $a==b$.
- Ветки условных операторов всегда должны ограничиваться фигурными скобками. Хотя синтаксис языка допускает написание единичного оператора в ветке без заключения его в фигурные скобки, лучше этого избегать.

Мы рассмотрели работы с логическим типом данных. Для закрепления материала вам предлагается выполнить несколько практических заданий, после чего перейти к изучению следующей темы.

ЗАДАНИЯ К ТЕМЕ 1.2

Реализуйте следующие программы:

1. Пользователь вводит 2 числа. Определить, делится ли первое на второе без остатка. Вывести сообщение об этом в MessageBox.
2. Пользователь вводит два числа. Если первое делится на второе без остатка, вывести в label текст: «Второе число – делитель первого». Если нет – вывести в label текст: «Первое число на второе не делится». Сделать тоже самое для обратной ситуации (делить второе на первое), но результат вывести в MessageBox.
3. Пользователь вводит число, определить, четно оно или нет, выведя об этом сообщение в label.
4. Пользователь вводит число, определить, на какие из чисел 2, 3, 4, 5 делится без остатка это число, выведя об этом сообщение в label.
5. Пользователь вводит три числа. Определить, делится ли сумма первых двух на третье без остатка. Вывести сообщение об этом в MessageBox.
6. Пользователь вводит 2 числа. Программа выводит большее из них.
7. Пользователь вводит 3 числа. Программа выводит меньшее из них.
8. Пользователь вводит 2 числа. Если числа равны, программа выводит надпись «Числа равны». Если числа не равны, программа выводит «Числа не равны».
9. Пользователь вводит 2 числа. Если числа равны, программа выводит надпись «Числа равны». Иначе, если первое число больше второго, выводится надпись «Первое число больше второго». Иначе выводится надпись «Второе число больше первого».
10. Пользователь вводит 4 числа. Если сумма первого и второго числа меньше суммы третьего и четвертого, программа выводит надпись «Сумма первых двух чисел меньше суммы двух вто-

рых». Если они равны, программа выводит «Сумма первых двух чисел равна сумме двух вторых». Иначе выводится «Сумма первых двух чисел больше суммы двух вторых».

ТЕМА 1.3. РАЗРАБОТКА ИГРЫ «УГАДАЙ ЧИСЛО»

Рассмотрим, как освоенный материал можно использовать для разработки игр. Начнем с самой простой игры – «Угадай число». При желании вы можете реализовать эту игру самостоятельно, прочитав лишь блок «Обратите внимание», однако ниже представлены все основные шаги разработки.

Начнем с проектирования интерфейса. Разместим на форме `textBox`, `label` и `button`. Изменим текст формы на «Угадай число», текст `label` – на «Введите число», а текст `button` – на «Проверить». Чтобы игра выглядела более симпатично, мы можем изменить шрифты элементов и загрузить на форму какую-нибудь фоновую картинку. Последнее выполняется через ее свойство `Background` в окошке `Properties`. Вызываем окно импорта картинки и через кнопку `Import` ищем нужный нам файл и загружаем его в программу (рисунок 1.14).

Обратите внимание!

- *Суть игры: компьютер загадывает число, пользователь пытается его угадать.*
- *Алгоритм работы: при старте программы компьютер генерирует случайное число, пользователь вводит свой вариант в `textBox`, нажимает на кнопку и видит одно из трех сообщений: «Вы угадали!», «Ваше число больше», «Ваше число меньше».*
- *Подсказки: нам понадобится кнопка, элемент для ввода данных, две числовые переменные (причем одна из них глобальная), условный оператор, генератор случайных чисел и `MessageBox`.*



Рисунок 1.14. Загрузка фоновой картинки

В результате у вас должно получиться что-то похожее на следующее изображение (рисунок 1.15):

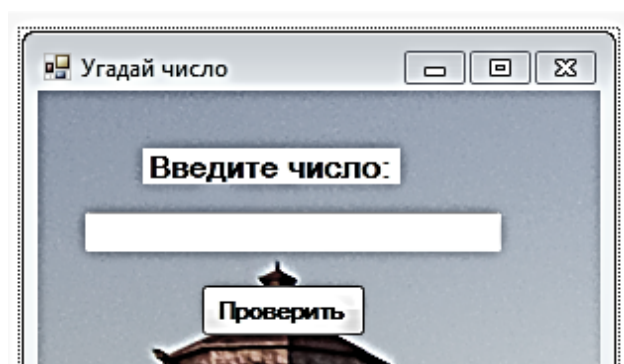


Рисунок 1.15. Внешний вид игры

Теперь займемся самим процессом игры. Так как пока мы пишем однораундную игру, то компьютер должен загадать число при загрузке формы, а проверять его мы должны по нажатию кнопки. Загрузка формы и нажатие кнопки – два разных события, обрабатываемые разными методами, а это значит, что переменная, хранящая число, должна быть объявлена так, чтобы к ней был доступ из обоих

методов. То есть она должна быть глобальной. Поэтому щелкаем два раза левой кнопкой мыши на форме, генерируя метод Form_Load (обработку загрузки формы), и перед заголовком этого метода объявляем целочисленную переменную для хранения числа и генератор случайных чисел. Затем в методе Form_Load присваиваем переменной случайное значение, как показано в листинге 1.18.

Листинг 1.18

```
int Comp_num=0;
Random r=new Random();
private void Form1_Load(object sender, EventArgs e) { Comp_num=r.Next(0,100);}
```

В данном случае числа 0 и 100 определяют диапазон, из которого берется число.

Обратите внимание!

- *При работе с генератором случайных чисел первое указанное в методе Next число включается в диапазон, из которого генерируется число, а второе – нет. То есть код r.Next(0,100) выдает число в диапазоне от 0 до 99.*

Теперь опишем действие программы, происходящее при нажатии на кнопку. Напомним, здесь нам нужно сравнить введенное пользователем число с загаданным. Алгоритм прост: считываем число, пишем условия сравнения, используя вложенные if и выводя соответствующие сообщения. Решение задачи представлено в листинге 1.19.

Листинг 1.19

```
private void button1_Click(object sender, EventArgs e){
int Hum_num=Convert.ToInt32(textBox1.Text);
if(Comp_num==Hum_num) {MessageBox.Show(«Вы угадали!!!»);}
else {if (Comp_num>Hum_num) {MessageBox.Show(«Мое число больше»); }}
else {if (Comp_num<Hum_num) {MessageBox.Show(«Мое число меньше»); }}}}
```

Таким образом, мы написали самый простой вариант компьютерной игры. Для закрепления материала вам предлагается выполнить несколько практических заданий, после чего перейти к изучению следующей темы.

ЗАДАНИЯ К ТЕМЕ 1.3

Усложните игру:

1. Добавьте в игру кнопку «Начать заново», которая должна перезапустить процесс игры. С помощью свойства `Visible` сделайте ее видимой (`button2.Visible=true`), только если игрок угадал число. При загрузке формы свойству присвойте значение `false`.
2. Ограничьте игрока в количестве попыток.
3. Заведите систему баллов.
4. Вместо сообщений о том, что число больше или меньше, поместите на форму `label`. В него выводите промежуток, в котором находится угадываемое число. Варианты, которые вводит игрок, должны сокращать промежуток, устанавливая либо его верхнюю границу, либо нижнюю.
5. Подберите картинку оформления формы, а также цвета кнопок и меток так, чтобы интерфейс программы был красивым.

ТЕМА 1.4. РАЗДЕЛЕНИЕ ЛОГИКИ И ИНТЕРФЕЙСА

В разработанной нами игре есть один существенный минус: ее логика жестко привязана к интерфейсу и, следовательно, если мы захотим реализовать эту же игру, используя другую технологию отрисовки интерфейса, нам придется переписывать и игровой код тоже. Чтобы решить эту проблему, воспользуемся парадигмой объектно-ориентированного программирования и познакомимся с понятием класса.

С точки зрения объектно-ориентированного подхода любая программная система рассматривается состоящей из объектов, выполняющих определенные функции. Например, для нашей игры ее процесс должен был организовываться через взаимодействие двух объектов: объекта «Форма», отвечающего за интерфейс и объекта «Компьютерный игрок», отвечающие за логику игры. Однако прежде чем модифицировать код согласно этому подходу, нам нужно познакомиться с некоторыми теоретическими понятиями:

Класс – это тип, описывающий устройство объектов.

Поля – это переменные, принадлежащие классу.

Методы – это функции (процедуры), принадлежащие классу.

Функция – фрагмент программного кода, оформленный по определенным правилам как полноценная подпрограмма, выполняющая определенные действия. К функции можно обратиться из другого места программы, вызвав ее по имени. В этот момент начинает выполняться описанный в ней код. Затем, после его выполнения, управление возвращается обратно в точку программы, где данная функция была вызвана. Функция может принимать параметры и должна возвращать некоторое значение, возможно пустое.

Процедура – функция, которая возвращает пустое значение.

Объект – это экземпляр класса, сущность в адресном пространстве компьютера.

Можно сказать, что класс является шаблоном для объекта, описывающим его структуру и поведение. Поля класса определяют структуру объекта, методы класса – поведение объекта. С точки зрения практической реализации (в самом тексте программы) класс является типом данных, а объект – переменной этого типа. Именно поэтому парадигма называется объектно-ориентированной: на этапе выполнения программы ее функциональность реализуется объектами, но на этапе написания программы программист описывает классы, которые будут определять, как объекты должны взаимодействовать друг с другом.

Объявление класса состоит из двух частей: объявление заголовка класса и объявление тела класса. Заголовок класса состоит из модификатора доступа, ключевого слова `class` и имени самого класса. Тело класса – есть конструкция, заключенная в фигурные скобки и содержащая объявление полей и методов, принадлежащих классу. Пример объявления класса с целочисленным полем представлен в листинге 1.20.

Листинг 1.20

```
public class MyClass { int a; }
```

Так как в одной программе может быть множество классов, некоторые из которых могут выполнять схожие функции, то для удобства навигации было придумано понятие пространства имен.

Пространство имен (или `namespace`) – средство логической группировки классов программы. Допустим, мы пишем программу для автоматизации деятельности какого-либо магазина. Нам необходимо будет вести учет товара, фиксировать оплаты от клиентов и работать с поставщиками. Логично будет классы, работающие с товарами, объединить в одно пространство имен, с покупателями – в другое, с поставщиками – в третье. Тогда, даже если функции какого-либо типа разобьются по нескольким файлам, мы будем знать, к какому типу они относятся. Или наоборот, если внутри одного файла будут функции нескольких типов, то по пространствам имен мы их разделим. Пространство имен объявляется следующим образом (листинг 1.21).

Листинг 1.21

```
namespace <Имя пространства имен>{ <Объявления классов> }
```

Внутри пространства имен к объявленному в нем классу мы можем обращаться просто по имени. Если же мы вызываем класс, определенный в другом пространстве имен, то обращаться к нему мы должны так: `<Имя пространства имен>.<Имя класса>`. Средством, ко-

торое позволяет сокращать имена классов, является оператор `using`: он «подключает» указанное пространство имен к текущему. Однако, если в разных пространствах имен объявлены одноименные классы, то в этом случае оператор `using` не поможет – придется указывать полное имя класса. Синтаксис `using` показан в листинге 1.22.

Листинг 1.22

```
using <ИмяПространстваИмен>;
```

Обратите внимание!

- *В C# все основные классы находятся в пространстве имен `System`, поэтому в каждом проекте неизбежно встретится строка `using System`;*

Теперь рассмотрим работу с объектами. Объявление объекта (создание объекта как экземпляра класса) состоит из двух частей: создание переменной-ссылки на область памяти, в которой будет располагаться объект, выделение памяти для объекта и заполнение этой памяти начальными значениями, иначе говоря инициализация данной переменной-ссылки. Объявление переменной-ссылки, а иными словами, объекта, подчиняется общему правилу объявления переменных в C#. Напомним, что переменные могут объявляться в любом месте в теле методов. Переменная, объявленная вне тела метода, но внутри тела класса, становится полем. Пример объявления переменной-объекта класса `MyClass` представлен в листинге 1.23.

Листинг 1.23

```
MyClass MyObj;
```

Под созданием объекта будем понимать выделение под него памяти и заполнение ее значениями объявленных в классе полей. Выделение памяти осуществляет оператор `new`, а задачу заполнения памяти начальными значениями решает специальный метод объекта, назы-

ваемый конструктором. Конструктор – метода объекта, объявленный следующим образом: для этого метода всегда используется модификатор доступа `public`, нет типа возвращаемого значения (нет даже `void`), имя метода совпадает с именем класса. Однако компилятор C# не требует обязательного определения конструктора для класса. Если конструктор не объявлен, компилятор вызовет так называемый конструктор по умолчанию, который создаст сам. Пример создания объекта приведен в листинге 1.24.

Листинг 1.24

```
MyClass MyObj=new MyClass();
```

Обратите внимание!

- *При объявлении переменная может быть сразу инициализирована (ей может быть присвоено какое-либо значение, например, `int a=0;`) Для переменной-объекта ее инициализация будет называться созданием объекта.*

Как было сказано выше, у любого класса есть методы. Они могут быть процедурными или функциональными.

Функциональные методы – методы, которые в результате своей работы возвращают какое-либо значение.

Процедурные методы – методы, которые в результате своей работы не возвращают никакого значения.

Синтаксис описания функционального метода представлен в листинге 1.25. Процедурного – в листинге 1.26.

Листинг 1.25

```
<модификатор доступа> <тип возвращаемого значения> <имя метода> (<список параметров>)  
{  
  <.....тело метода .....> return <значение>;  
}
```

Листинг 1.26

```
<модификатор доступа> void <имя метода> (<список параметров>)  
{  
<...тело метода...>  
}
```

В данном случае `void` – специальное ключевое слово, обозначающее пустой тип. Список параметров, если он не пустой, состоит из перечисленных через запятую пар `<тип имя>`. Ключевое слово `return` может использоваться и в процедурном методе, если необходимо по какому-то условию экстренно прервать его работу, но таким образом используется редко.

Обратите внимание!

- *Код метода, расположенный после ключевого слова `return` никогда не будет выполнен!*

Пример объявления класса с процедурным и функциональным методом приведен в листинге 1.27.

Листинг 1.27

```
class MyClass {  
public void ProcMet(int a,int b)  
{  
//тело метода...  
}  
public int FuncMet()  
{  
//тело метода...  
return числовое_значение;}  
}
```

В методе `ProcMet` параметры `a` и `b` по сути являются локальными переменными для его тела. В эти переменные передаются значения из

основной программы, которые указываются в скобках при вызове метода.

Обратите внимание!

- *Текст программы, перед которым стоит двойной слеш – закомментирован. Он виден программисту, но компилятор его игнорирует. Комментарии обычно используются для написания пояснений к коду или для «выключения» каких-либо участков кода.*

Вызов методов класса осуществляется через имя объекта, или через имя класса, если метод был объявлен со специальным ключевым словом `static`. Вызов методов класса, описанного в листинге 1.27, проиллюстрирован листингом 1.28. Второй случай показан в листинге 1.29.

Листинг 1.28

```
MyClass MyObj=new MyClass(); int a=MyObj.FuncMet();  
MyObj.ProcMet(4,5);
```

Листинг 1.29

```
class MyClass  
{  
    public static void ProcMet(int a,int b)  
    {  
        //тело метода...  
    }  
}  
.....  
.....  
//вызов метода:  
MyClass.ProcMet(4,5);
```

В обоих случаях в теле метода `ProcMet` параметр «а» примет значение 4, а параметр «b» – 5.

Единственный метод класса, вызов которого происходит по-другому – это конструктор. Управление ему передается автоматически, используя ключевое слово `new`. Никаким другим образом конструктор вызвать нельзя.

При разработке методов могут возникать неоднозначности обращения к элементам, если поле класса и параметр в методе имеют одно и то же имя, причем в этом случае параметр перекроет видимость поля. Например, пусть класс объявлен следующим образом (листинг 1.30). Ошибки компиляции в данном случае не будет, только предупреждение с текстом: «Assignment made to same variable; did you mean to assign something else?» Компилятор укажет на опасный момент, но предоставит принимать решение программисту: вдруг он на самом деле имел в виду то, что написал и решил присвоить значение параметра самому же параметру.

Листинг 1.30

```
class Class1
{ int a;
public void Method(int a) { a = a; }
}
```

Для разрешения подобных конфликтов используется ключевое слово `this`. То есть если мы хотим полю `a` присвоить значение параметра `a`, то код будет следующим (листинг 1.31). В теле метода ключевое слово `this` хранит указатель на объект, вызвавший этот метод.

Листинг 1.31

```
class Class1
{ int a;
public void Method(int a) { this.a = a; }
}
```

На модификаторах доступа в данном пособии мы подробно останавливаться не будем. Скажем лишь, что ключевое слово `public`

делает поле или метод видимыми за пределами тела класса (то есть их можно вызывать из основной программы). Если же это слово не указано, или стоит модификатор доступа `private`, то вне тела класса использовать эти поля или методы нельзя.

Теперь рассмотрим подробнее написанный нами код игры. В нем мы уже использовали классы, создавали объекты и вызывали методы. Например, строка кода: `Random r=new Random();` реализует создание объекта класса `Random`. А код: `int Hum_num=r.Next(0,10);` записывает в переменную `Hum_num` результат, выдаваемый функциональным методом `Next`, который мы вызвали у объекта `r`.

При выполнении преобразования значения из строки в число, мы пользуемся методом класса `Convert`, объявленным с ключевым `static`. Когда же мы привязываем код к кнопке или к загрузке формы, то по сути мы проектируем методы класса `Form`, которые будут вызываться у объекта этого класса при нажатии на кнопку или при появлении формы на экране. Код же создания объекта класса `Form` находится в файле `Program.cs` в методе `Main` и выглядит так: `Application.Run(new Form1());`

Для добавления собственного класса в проект мы должны перейти в окно `Solution Explorer` (Обозреватель решений) и, щелкнув правой кнопкой мыши на имени проекта, в контекстном меню выбрать пункт `Add -> Class`, как показано на рисунке 1.15*.

В появившемся окне необходимо задать имя класса, а затем в окне кода определить его поля и методы. Для нашей игры в классе должны быть объявлены:

1. Поле, хранящее заданное число.
2. Метод, записывающий в загаданное число случайное значение.
3. Метод, сравнивающий пользовательское число с загаданным.

В коде формы будут следующие изменения:

1. Вместо объявления переменной для хранения числа и создание объекта класса `Random` должен быть код создания объекта нашего класса.

2. В методе `FormLoad` должен быть вызов метода, записывающего в заданное число случайного значения.
3. В методе `button1_Click` должен быть вызов метода, сравнивающего пользовательское число с заданным. При этом в качестве параметра в метод передается введенное в `textBox` значение, преобразованное в числовой тип.

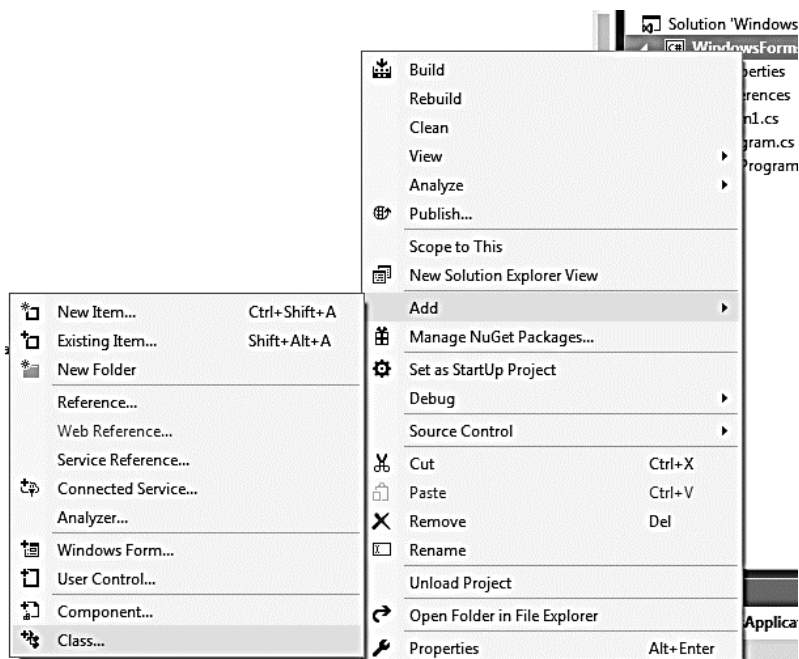


Рисунок 1. 15*. Добавление класса в проект

Теперь код этого разработанного класса мы можем использовать в сочетании с любой технологией реализации интерфейса, работающей с кодом на языке `C#`.

ЗАДАНИЕ К ТЕМЕ 1.4

Самостоятельно реализуйте разделение логики и интерфейса в игре «Угадай число».

ТЕМА 1.5. ГРАФИЧЕСКИЙ ИНТЕРФЕЙС. СТАТИКА

В разработанной нами игре есть еще один существенный недостаток: ее графический интерфейс крайне беден, статичен и скучен, а это недопустимо. Множество технологий обладает возможностями работы с графикой и применяется для разработки игр, но в рамках первой части книги мы познакомимся с возможностями технологии GDI+. Это самая простая из графических технологий, позволяющая реализовывать элементарные графические операции: отрисовка фигур, текста, заливка, передвижение и трансформация объектов. Ее можно использовать только для реализации самых простых игр под Windows.

GDI (Graphics Device Interface, Graphical Device Interface) является интерфейсом Windows, предназначенным для представления графических объектов и вывода их на монитор или принтер. В первую очередь он отвечает за отрисовку линий, кривых, отображение шрифтов и обработку палитры и обеспечивает унификацию работы с различными устройствами вместо прямого доступа к оборудованию, что позволяет одними и теми же функциями рисовать на различных устройствах, получая при этом практически одинаковые изображения.

В рамках данной части пособия мы освоим технику рисования картинок в оперативной памяти с последующим выводом их на форму. Для рисования мы будем использовать объект класса `Bitmap`, а для вывода на форму – элемент управления `PictureBox`. Рассмотрим написание простейшего приложения, когда по нажатию на кнопку на форме будет отображаться красный кружок. Для этого создаем новый проект типа `WindowsFormsApplication`, размещаем на форме `PictureBox` и `Button`. Затем переходим к методу обработки нажатия на кнопку. В нем нам нужно, во-первых, создать объект класса `Bitmap`, с размерами, совпадающими с нашим `PictureBox`. В данном случае `Bitmap` выполняет роль холста, а `PictureBox` – рамки, в которую этот холст повесят, поэтому так важно совпадение размеров. Затем мы должны

создать объект класса Graphics – основного класса, предоставляющего доступ к возможностям GDI+. Для данного класса не определено ни одного конструктора. Его объект создается в ходе выполнения статического метода, привязывающего объект к конкретному объекту, у которого есть поверхность для рисования. Одним из таких объектов и является Bitmap. Благодаря этой привязке вызовы методов отображения фигур будут обрабатываться на нашей битовой карте.

Класс Graphics содержит множество методов рисования вида Fill* или Draw*, отвечающих за отображение закрашенных или не закрашенных фигур. Первая группа методов в качестве одного из параметров принимает объект типа Brush (кисть), а вторая – объект типа Pen (карандаш). Исключение – метод DrawString, который отображает текст. Этот метод в качестве одного из параметров принимает объект Brush. Для того чтобы отобразить красный круг, как мы задумали ранее, необходимо вызвать метод FillEllipse со следующими параметрами:

- x, y – координаты левого верхнего угла прямоугольника, в который будет вписан эллипс;
- объект-кисточка. Например, Brushes.Red – стандартная красная кисточка;
- высота и ширина эллипса.

Последнее, что необходимо сделать – «вывесить» отрисованную картинку в наш pictureBox, в свойство Image. Код отрисовки эллипса представлен в листинге 1.32.

Листинг 1.32

```
Bitmap bmp = new Bitmap(pictureBox1.Width, pictureBox1.Height);  
Graphics gr = Graphics.FromImage(bmp);  
gr.FillEllipse(Brushes.Red, 10, 10, 40, 40);  
pictureBox1.Image = bmp;
```

В итоге на экране вы должны увидеть примерно следующее (рисунок 1.16), а именно круг. Поясим, что на самом деле это эллипс,

у которого длина равна ширине. При сомнениях измените один из этих параметров и отрисуйте изображение заново.

Теперь рассмотрим способы создания различных кисточек. Например, создадим кисточку случайного цвета и закрасим ей наш круг (листинг 1.33).

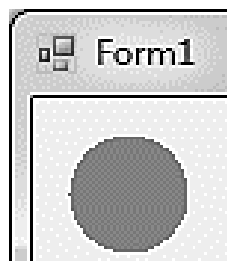


Рисунок 1.16. Полученный круг (эллипс)

Обратите внимание!

- *Рисование в технологии GDI+ осуществляется слоями, которые перекрывают друг друга, поэтому если мы напишем код:*

`gr.FillEllipse(Brushes.Red,10,10,40,40);`

`gr.FillEllipse(Brushes.Green,10,10,40,40);`

то на форме увидим только зеленый круг (эллипс).

Листинг 1.33

```
Bitmap bmp = new Bitmap(pictureBox1.Width, pictureBox1.Height);
Graphics gr = Graphics.FromImage(bmp);
Random r=new Random();
SolidBrush sb=new SolidBrush(Color.FromArgb(r.Next(0,255), r.Next(0,255),
r.Next(0,255), r.Next(0,255)));
gr.FillEllipse(sb,10,10,40,40);
pictureBox1.Image = bmp;
```

В листинге 1.33 метод `FromArgb` принимает в качестве параметров 4 числа, кодирующих цвет в системе ARGB: А (альфа) – прозрачность; R (red) – красный цвет; G (green) – зеленый цвет; B (blue) – синий цвет.

Для создания кисточек с более интересными характеристиками необходимо подключить namespace System.Drawing.Drawing2D (прописать в начале файла проекта using System.Drawing.Drawing2D). Рассмотрим создание градиентной кисти, написав следующий код (листинг 1.34). Как мы знаем, линейная градиентная заливка – это вид заливки, в которой необходимо задать цвет и координаты двух ключевых точек, расположенных на одной прямой, а цвет точек между ними рассчитывается относительно них по определенным математическим алгоритмам. В итоге получают плавные переходы из одного цвета в другой.

Листинг 1.34

```
Bitmap bmp = new Bitmap(pictureBox1.Width, pictureBox1.Height);  
Graphics gr = Graphics.FromImage(bmp);  
Random r=new Random();  
LinearGradientBrush gradientBrush =  
new LinearGradientBrush(new Point(20, 20),new Point(50, 50), Color.White, Col-  
or.Blue);  
gr.FillEllipse(gradientBrush,10,10,40,40);  
pictureBox1.Image = bmp;
```

Результат раскраски эллипса градиентной кистью представлен на рисунке 1.17.

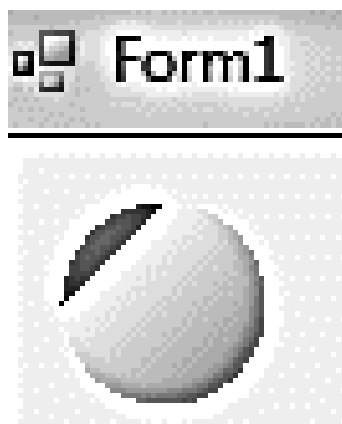


Рисунок 1.17. Градиентная кисть

Следующая кисть, которую мы рассмотрим, – штриховая. Создается и используется она следующим образом (листинг 1.35). Первый параметр конструктора кисти задает вид штриховки. Это системное перечисление с множеством вариантов. Второй параметр отвечает за цвет штрихов, а третий – за цвет фона.

При использовании этой кисти в рисовании нашего круга мы получим следующий результат (рисунок 1.18).

Листинг 1.35

```
Bitmap bmp = new Bitmap(pictureBox1.Width, pictureBox1.Height);  
Graphics gr = Graphics.FromImage(bmp);  
Random r=new Random();  
HatchBrush hatchBrush = new HatchBrush(HatchStyle.Cross,  
Color.Red, Color.Yellow);  
gr.FillEllipse(hatchBrush,10,10,40,40);  
pictureBox1.Image = bmp;
```

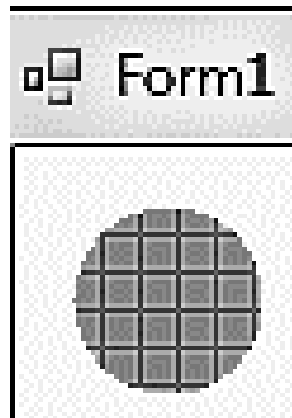


Рисунок 1.18. Штрихованный круг

Кисточки используются для рисования закрашенных фигур. Если же мы хотим изобразить только контурные фигуры, то нам нужны карандаши. Рассмотрим подробнее их создание. Карандаш – объект класса Pen. Его можно создать на основе цвета или на основе ранее созданной кисти. Можно указывать толщину линии, а также форму ее конечной точки (только для незамкнутых контуров) и прочее. Ниже приведен код для создания различных карандашей (лис-

тинг 1.36). Работу с карандашами вам предлагается выполнить самостоятельно.

Листинг 1.36

```
Pen p1=new Pen(Color.Red); // Обычное красное перо
Pen p2=new Pen(Color.Green, 4); // Ширина 4 пиксела
p2.EndCap = LineCap.ArrowAnchor; //Стрелочка на конце
Pen p3=new Pen(gradientBrush, 6); // Градиент
Pen p4=new Pen(hatchBrush, 6); // Штриховка
```

Мы рассмотрели методы вывода различных векторных примитивов. Теперь рассмотрим методы их преобразования:

- перенос;
- поворот;
- масштабирование;
- сдвиг.

Данные методы широко применяются в ситуациях, когда вам необходимо изменить размер, положение или форму фрагмента рисунка, состоящего из большого числа объектов. Перечисленные выше методы применяют преобразования сразу ко всем отображаемым объектам, что экономит вам время.

Первая важная трансформация, которую мы рассмотрим, – перенос. По сути, это линейное перемещение объекта, при котором размер и ориентация не меняются. Напишем код отрисовки красного, обведенного контуром, прямоугольника, затем вызовем метод переноса и повторим код (листинг 1.37).

Листинг 1.37

```
gr.FillRectangle(Brushes.Red, 10, 10, 20, 40);
Pen p2=new Pen(Color.Green, 4); // Ширина 4 пиксела
gr.DrawRectangle(p2, 10, 10, 20, 40);
gr.TranslateTransform(20, 20);
gr.FillRectangle(Brushes.Red, 10, 10, 20, 40);
gr.DrawRectangle(p2, 10, 10, 20, 40);
```

В итоге на экране вы должны увидеть примерно следующее (рисунок 1.19).

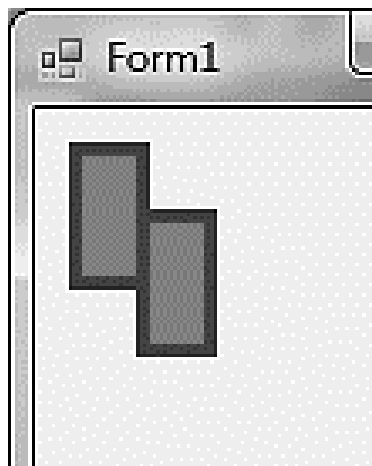


Рисунок 1.19. Трансформация «Перенос»

Метод `TranslateTransform` принимает в качестве параметров величины переноса по обеим осям. То есть он сдвигает начало координат для отрисовки объекта.

Вторая трансформация – поворот. Она представляет собой изменение угла отрисовки объекта относительно начала координат. Величина поворота задается в радианах. Код данной трансформации следующий (листинг 1.38).

Листинг 1.38

```
Pen p2=new Pen(Color.Green, 4); // Ширина 4 пиксела
gr.FillRectangle(Brushes.Red,50, 50, 20, 40);
gr.DrawRectangle(p2, 50, 50, 20, 40);
gr.RotateTransform(20);
gr.FillRectangle(Brushes.Red, 50, 50, 20, 40);
gr.DrawRectangle(p2, 50, 50, 20, 40);
```

Обратите внимание!

- *Поворот выполняется не относительно оси объекта, а относительно начала координат.*

В итоге у вас на форме должно вывестись следующее изображение (рисунок 1.20).

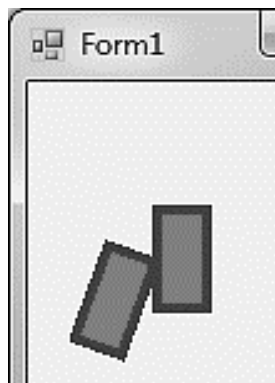


Рисунок 1.20. Трансформация «Поворот»

Третья трансформация – масштабирование. Это изменение размеров объекта по обеим осям. Например, с помощью этой трансформации увеличим наш прямоугольник в два раза в высоту и ширину (листинг 1.39).

Листинг 1.39

```
Pen p2=new Pen(Color.Green, 4); // Ширина 4 пиксела

gr.FillRectangle(Brushes.Red,10, 10, 20, 40);
gr.DrawRectangle(p2, 10, 10, 20, 40);
gr.ScaleTransform(2,2);

gr.FillRectangle(Brushes.Red, 10, 10, 20, 40);
gr.DrawRectangle(p2, 10, 10, 20, 40);
```

В итоге у вас на форме должно появиться следующее (рисунок 1.21).

Последняя, но не менее важная трансформация, которую мы рассмотрим – сдвиг. Он представляет собой практически произвольное искажение объекта. Для него не существует какого-то отдельного метода. В данном случае необходимо напрямую задавать матрицу трансформации (листинг 1.40).

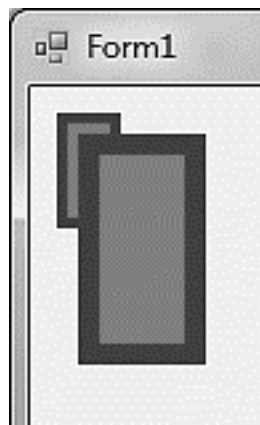


Рисунок 1.21. Трансформация «Масштабирование»

Листинг 1.40

```
Pen p2=new Pen(Color.Green, 4); // Ширина 4 пиксела
gr.FillRectangle(Brushes.Red,10, 10, 20, 40);
gr.DrawRectangle(p2, 10, 10, 20, 40);

Matrix matrix = new Matrix();
matrix.Shear(0.5f, 0.25f);

gr.Transform = matrix;
gr.FillRectangle(Brushes.Red, 10, 10, 20, 40);
gr.DrawRectangle(p2, 10, 10, 20, 40);
```

В итоге на экране должна получиться следующая картинка (рисунок 1.22).



Рисунок 1.22. Трансформация «Сдвиг»

Нужно заметить, что все рассмотренные трансформации основываются на преобразовании матрицы координат. И при вызове нескольких методов подряд, их эффекты будут складываться. Для того чтобы вернуть матрицу координат в исходное состояние, необходимо вызвать метод `ResetTransform`.

Теперь для демонстрации возможностей технологии GDI+ реализуем ее средствами отрисовку изображения, показанного на рисунке 1.23. Оно было создано в графическом редакторе GIMP.



Рисунок 1.23. Тестовое изображение

Для изображения указанной картинке нам потребуется шесть прямых эллипсов, восемь эллипсов с трансформацией «Поворот», пять линий и две дуги. Линии рисуются методом `DrawLine`. В качестве параметров он принимает объект-карандаш, а также четыре координаты: `x-начальное`, `y-начальное`, `x-конечное`, `y-конечное`. Обратите внимание, начало координат – левый верхний угол формы. Дуги рисуются методом `DrawArc`. В качестве параметров метод принимает координаты `x-начала`, `y-начала` отрисовки дуги, ее высоту-ширину, а также начальный и конечный углы.

В результате выполнения кода из листинга 1.41 мы получим изображение, представленное на рисунке 1.24.

Листинг 1.41

```
Bitmap bmp = new Bitmap(pictureBox1.Width, pictureBox1.Height);
Graphics gr = Graphics.FromImage(bmp);

gr.FillEllipse(Brushes.Black, 10, 10, 200, 200);
gr.FillEllipse(Brushes.Pink, 30, 30, 140, 140);
gr.FillEllipse(Brushes.Black, 80, 140, 40, 10);
gr.FillEllipse(Brushes.White, 82, 142, 36, 5);
gr.FillEllipse(Brushes.Black, 80, 110, 45, 25);
gr.FillEllipse(Brushes.White, 82, 122, 10, 5);

gr.RotateTransform(-20);
gr.FillEllipse(Brushes.Black, 40, 70, 30, 60);
gr.FillEllipse(Brushes.White, 43, 73, 25, 55);
gr.FillEllipse(Brushes.Black, 40, 93, 25, 35);
gr.FillEllipse(Brushes.White, 48, 95, 15, 15);

gr.ResetTransform();

gr.RotateTransform(20);
gr.FillEllipse(Brushes.Black, 130, 0, 30, 60);
gr.FillEllipse(Brushes.White, 133, 3, 25, 55);
gr.FillEllipse(Brushes.Black, 133, 23, 25, 35);
gr.FillEllipse(Brushes.White, 135, 25, 15, 15);

gr.ResetTransform();

Pen p = new Pen(Color.Black, 3);
gr.DrawLine(p, 100, 60, 100, 40);
gr.DrawLine(p, 90, 40, 99, 60);
gr.DrawLine(p, 110, 40, 101, 60);
gr.DrawLine(p, 75, 105, 100, 100);
gr.DrawLine(p, 105, 102, 130, 107);
gr.DrawArc(p, 20, 100, 50, 70, -30, -50);
gr.DrawArc(p, 140, 90, 30, 50, -110, -130);

pictureBox1.Image = bmp;
```



Рисунок 1.24. Рисование средствами GDI+

Как мы видим, пропорции относительно оригинала несколько нарушены, следовательно, нужно подправить координаты и размеры отрисовки фигур. Этот шаг вам предлагается выполнить самостоятельно в рамках выполнения одного из заданий к теме для тренировки глазомера и пространственного мышления. Обратите внимание, что с помощью примитивов можно отрисовывать и более сложные объекты. Например, на рисунке 1.25 показано изображение Микки Мауса, созданное с помощью примитивов (в правой части картинки контурами показаны скрытые формы).



Рисунок 1.25. Рисование объекта через примитивы

Как вы могли заметить, GDI+ – это технология отображения векторной графики. Однако использование ее для отрисовки статических векторных изображений оправдано лишь в том случае, если с помощью построения по координатам вы получите нужный вам результат быстрее, нежели используя какой-либо специализированный пакет для создания векторных изображений. Особенно ярко это преимущество видно при создании фрактальной графики. Подробно на данном виде графики в рамках этого пособия мы останавливаться не будем, приведем лишь для примера изображения «Кривой дракона» (рисунок 1.26) и плазменного фрактала (рисунок 1.27), построенные средствами GDI+. Их создание описано в [2], как и описание других фракталов.

Фрактальную графику можно использовать для генерации фоновых изображений, сохраняя созданную картинку в файл. Для выполнения этого действия необходимо у объекта `Bitmap` вызвать метод `Save`, передав в качестве параметра имя файла для сохранения. Например, выполнив код `bmp.Save("1.jpg");` мы сохраним созданное изображение в указанный файл, который будет лежать в папке с именем `bin\Debug` нашего проекта. Добраться до нее можно через контекстное меню, вызываемое щелчком правой кнопкой мыши на имени проекта (рисунок 1.28).

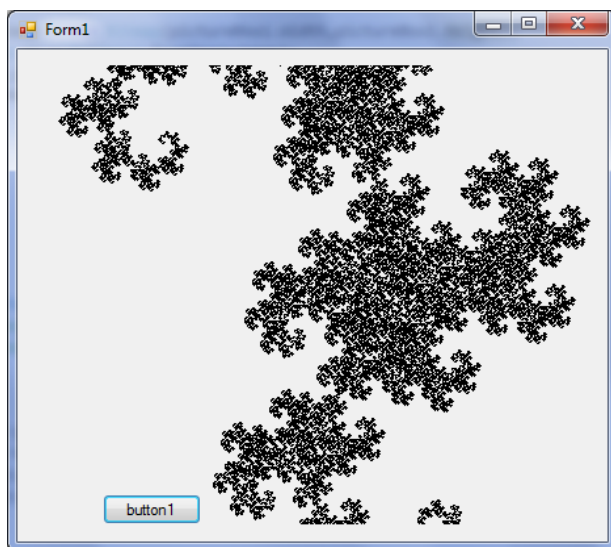


Рисунок 1.26. Кривая дракона

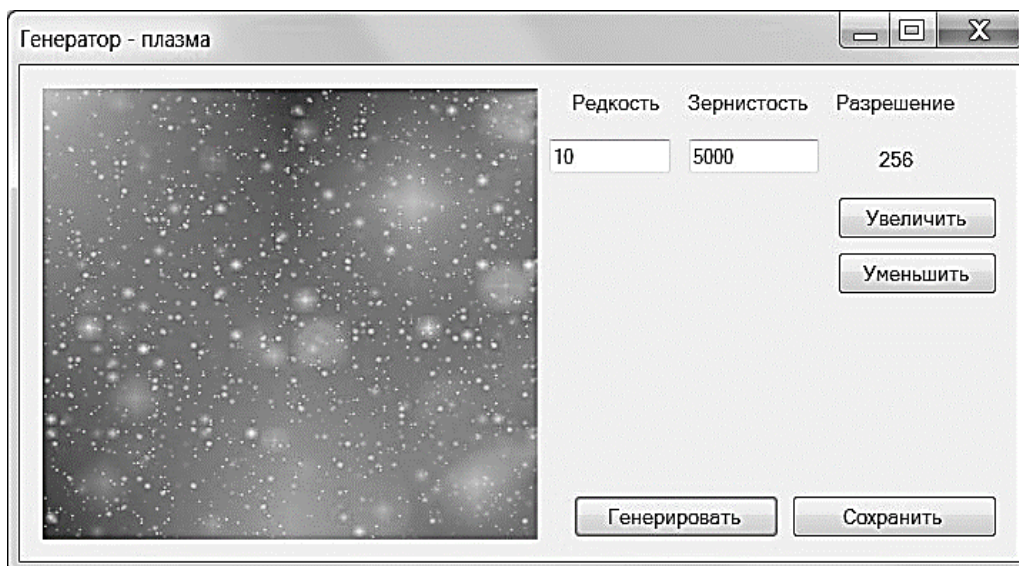


Рисунок 1.27. Плазменный фрактал

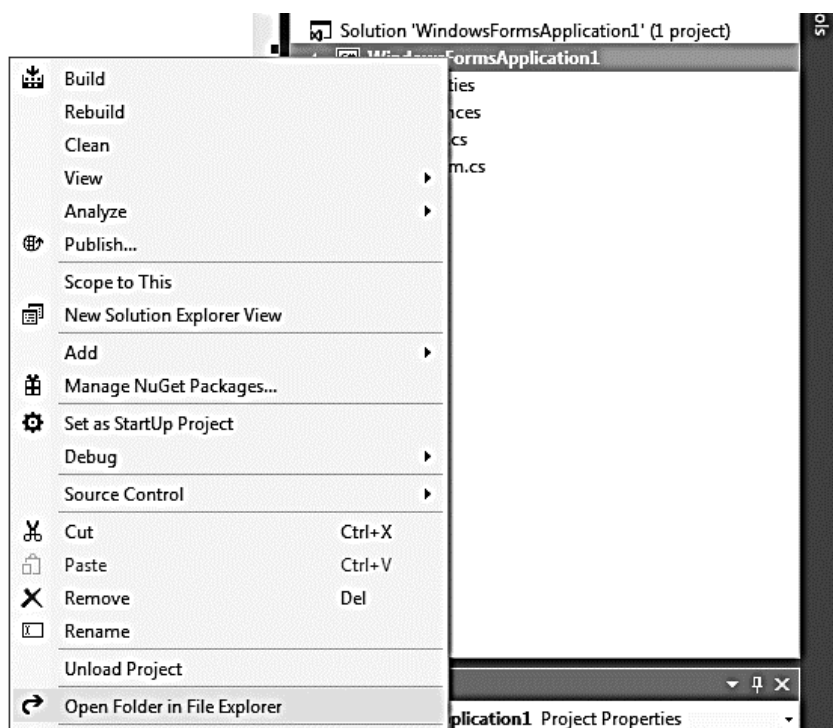


Рисунок 1.28. Открытие папки с проектом

Обратите внимание!

- До шага открытия папки проекта через контекстное меню проект обязательно должен быть сохранен командой *Save All*, иначе он будет лежать во временной папке, что создаст определенные неудобства в работе.

Необходимо отметить, что непосредственная отрисовка графических объектов в играх применяется очень редко. В основном лишь в игровых приложениях типа «Крестики-нолики», «Морской бой» и прочее, где не требуется каких-то сложных графических объектов. Во всех остальных случаях изображения подготавливаются в специализированных графических пакетах, а в игровую программу просто догружаются. В технологии GDI+ для вывода предварительно созданных изображений на форму используется следующий код (листинг 1.42).

Листинг 1.42

```
Image i=Image.FromFile("1.jpg");  
gr.DrawImage(i, 0, 0, 250, 250);
```

При этом предполагается, что файл с изображением лежит в папке bin->Debug нашего проекта. Иначе в методе FromFile необходимо указывать полный путь. При загрузке изображений стоит придерживаться правила «простоты путей»: картинки должны лежать в папке, чье название не содержит кириллицы и недалеко от исполняемого файла проекта. Например, если вы хотите все изображения хранить в отдельной папке Images рядом с exe-файлом проекта, то загрузить из нее изображение можно, указав в методе FromFile путь Application.StartupPath+@"\Images\1.jpg".

Теперь вернемся к загрузке изображений. В программе каждая картинка хранится в объекте Image. По сути, это обычная переменная, работа с которой происходит соответствующим образом. Сам вывод картинки реализуется методом DrawImage у известного нам объекта Graphics. В качестве параметров метод принимает объект-картинку, которую нужно вывести, координаты ее левого верхнего угла, а также отображаемую длину-ширину. Если фактический размер картинки не совпадает с областью вывода, – изображение будет масштабировано.

Кроме работы с изображениями нам необходимо познакомиться с работой с текстом. Для вывода текста используется метод Draw-

String объекта Graphics, принимающий в качестве параметров строку, которую нужно вывести, объект класса Font, кисть и координаты вывода. Конструктор класса Font в свою очередь принимает название и размер шрифта. Приведем пример кода вывода надписи «Hello, world!» (листинг 1.43).

Листинг 1.43

```
Bitmap bmp = new Bitmap(pictureBox1.Width, pictureBox1.Height);  
Graphics gr = Graphics.FromImage(bmp);  
gr.DrawString("Hello, world!", new Font("Arial",15), Brushes.Red,20,20);  
pictureBox1.Image = bmp;
```

Мы рассмотрели основы создания графического интерфейса. Теперь перейдем к обсуждению таких моментов, как интерактив и уровни, но перед этим вам предлагается выполнить ряд заданий по пройденному материалу.

ЗАДАНИЯ К ТЕМЕ 1.5

1. Напишите программу, выводящую следующее изображение (рисунок 1.29):

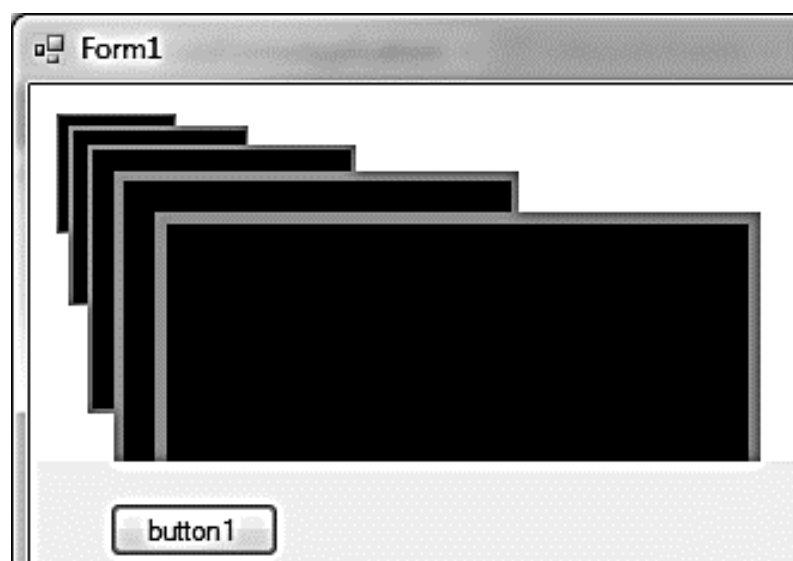


Рисунок 1.29. Изображение для задания 3

2. Доработайте код, выводящий картинку с рисунка 1.24, сделав ее более похожей на изображение с рисунка 1.23.
3. Напишите программу, рисующую Микки Мауса с рисунка 1.25.
4. Напишите программу, реализующую следующее: пользователь вводит свое имя и два числовых значения, программа отрисовывает имя красным цветом в черном прямоугольнике, располагая изображение в указанных координатах.
5. Напишите программу, выводящую следующее изображение (рисунок 1.30):

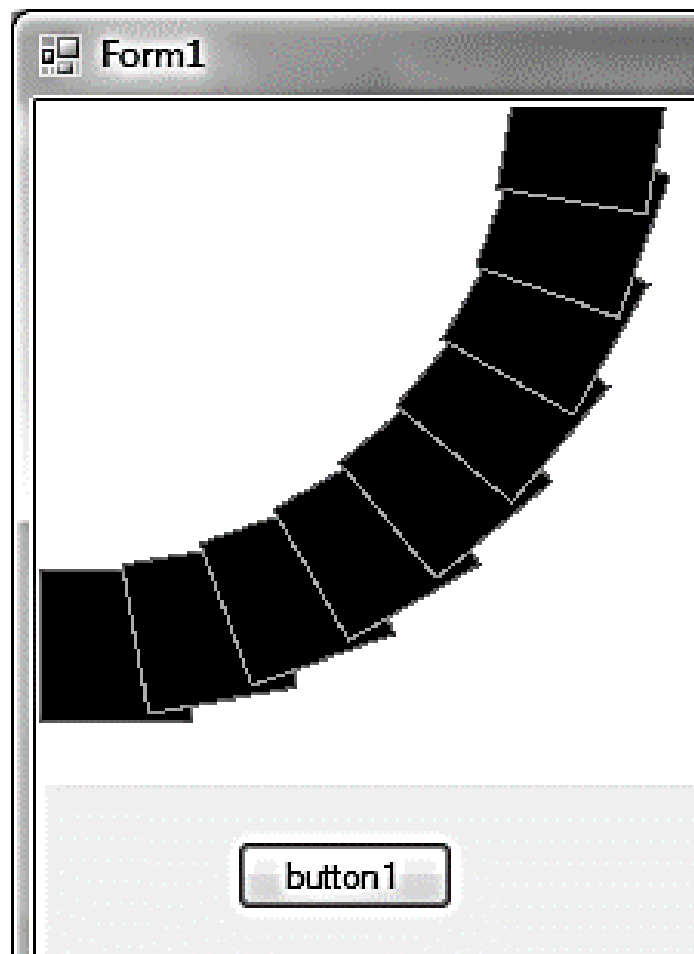


Рисунок 1.30. Изображение для задания 4

ТЕМА 1.6. ГРАФИЧЕСКИЙ ИНТЕРФЕЙС. ИНТЕРАКТИВ И УРОВНИ

Мы рассмотрели методы вывода графики на экран средствами технологии GDI+, однако все описанное статично. В играх же, как правило, присутствует анимация (движение, изменение размеров и прочее), взаимодействие с игроком и прохождение уровней. Рассмотрим реализацию этого средствами GDI+ и начнем с первого. Изучим способ создания анимации. Наш Bitmap мы можем рассмотреть как область видимости некоторой камеры, через которую фиксируется сцена с отображенными на ней объектами. Тогда под анимацией будет пониматься периодическая (n-раз в секунду) перерисовка сцены с обновленными характеристиками объектов. Для примера разработаем приложение, в котором по нажатию на кнопку некоторая картинка будет перемещаться по горизонтали из левой части экрана в правую. Для того чтобы это реализовать, нам потребуется еще один компонент – таймер. Если мы разместим его на форме, то он, в отличие от PictureBox или кнопки, расположится на панели под ней. Это происходит потому, что таймер не имеет (и ему не требуется) интерфейса взаимодействия с пользователем. Он выполняет роль внутренней программной компоненты. Если мы посмотрим на окно его свойств, то увидим Interval и Enabled – это два самых важных. Первое отвечает за отрезки времени, которые отсчитывает таймер, второе – за его запуск. Для лучшего понимания применения свойства Interval в анимации рассмотрим такое понятие, как FPS.

Согласно Википедии [1]: «Кадровая частота, частота кадров (англ. Frames per Second (FPS), Frame rate, Frame frequency) – количество сменяемых кадров за единицу времени в телевидении и кинематографе. В компьютерных играх под кадровой частотой (англ. FPS, Frame Per Second) понимается частота, генерируемая самой игрой в зависимости от ресурсов компьютера и необходимости передачи движений разной интенсивности. Понятие «Фреймрейт» (англ. Frame-

rate) используется как жаргонное обозначение такой частоты кадров. При этом игры можно разделить на два класса: игры с постоянной кадровой частотой и игры с переменной кадровой частотой. Игры с постоянной кадровой частотой выдают на слабых и мощных компьютерах одинаковое количество кадров в секунду. Если ресурсы компьютера невелики, и он не справляется с прорисовкой, то замедляется вся игра. Игры с переменной кадровой частотой на слабых компьютерах начинают пропускать кадры, скорость игрового процесса не меняется».

При использовании технологии GDI+ как такового FPS нет, однако оно моделируется с помощью таймера. Алгоритм работы этого компонента следующий: как только свойство `Enabled` устанавливается в значение `true`, начинается отсчет времени в миллисекундах. Как только он становится равным значению `Interval`, срабатывает программный метод, привязанный к таймеру, и отсчет начинается заново. Для привязки кода к таймеру, как и к кнопке, нужно выполнить на нем двойной клик мышью. В методе таймера должно происходить следующее: во-первых, изменение координат объекта, во-вторых, перерисовка его на новых координатах. Так как первый раз объект отрисовывается по нажатию на кнопку, а затем его координаты будут меняться в методе таймера, значит, их хранение нужно организовать через глобальные переменные. Кроме того, глобальной должна быть объявлена переменная, хранящая отображаемую картинку. С объектами же `Bitmap` и `Graphics` мы можем поступить по-разному: объявить их глобально и использовать и в методе таймера, и в методе кнопки, или в каждом методе объявить свои. Для примера реализуем оба варианта и сравним их достоинства и недостатки.

В первом случае объявим объекты локально и для наглядности сравнения изменим у `pictureBox` цвет в свойстве `BackColor` на белый. Объявим две глобальные переменные `x` и `y`, в кнопке отрисовываем в них объект-картинку, как и в таймере, но во втором случае будем перед отрисовкой изменять на 1 координату `x`. Код, реализующий это, будет следующим (листинг 1.44).

Листинг 1.44

```
int x = 0; int y = 0; Image i = Image.FromFile(Application.StartupPath +
@"\Images\1.jpg");

private void button1_Click(object sender, EventArgs e)
{
    Bitmap bm = new Bitmap(pictureBox1.Width, pictureBox1.Height);
    Graphics gr = Graphics.FromImage(bm);
    gr.DrawImage(i, x, y, 100, 100);
    timer1.Enabled = true;
    pictureBox1.Image = bm;
}

private void timer1_Tick(object sender, EventArgs e)
{
    Bitmap bm = new Bitmap(pictureBox1.Width, pictureBox1.Height);
    Graphics gr = Graphics.FromImage(bm);
    x++;
    gr.DrawImage(i, x, y, 100, 100);
    timer1.Enabled = true;
    pictureBox1.Image = bm;
}
```

Обратите внимание!

- *Выводимое изображение в файле 1.jpg должно лежать в папке Images рядом с exe-файлом проекта.*

Теперь, если мы запустим приложение, то увидим, что наша картинка будет двигаться, но достаточно медленно. Увеличить скорость передвижения мы можем двумя способами. Во-первых, уменьшить значение Interval у таймера. Во-вторых, изменить переменную x на большее значение, или совместить оба варианта.

С визуальной точки зрения наше приложение работает нормально, но, если мы посмотрим на его программную реализацию, то увидим нерациональный расход памяти: каждый раз, когда срабатывает таймер, мы выделяем память под новый объект. В этом как раз и состоит недостаток

использования локальных Bitmap и Graphics. Если мы хотим, чтобы память под них выделялась только один раз, то мы должны их сделать глобальными. Тогда наша программа будет иметь следующий код (листинг 1.45).

Листинг 1.45

```
Bitmap bm; Graphics gr; int x = 0; int y = 0;
Image i = Image.FromFile(Application.StartupPath + @"\Images\1.jpg");
public Form1(){ InitializeComponent();
bm = new Bitmap(pictureBox1.Width, pictureBox1.Height);
gr = Graphics.FromImage(bm);}
private void button1_Click(object sender, EventArgs e)
{ gr.DrawImage(i, x, y, 100, 100); timer1.Enabled = true; pictureBox1.Image = bm;}
private void timer1_Tick(object sender, EventArgs e){
x+=5; gr.DrawImage(i, x, y, 100, 100); timer1.Enabled = true;
pictureBox1.Image = bm;}
```

Результат работы кода из листинга 1.45 приведен на рисунке 1.31.

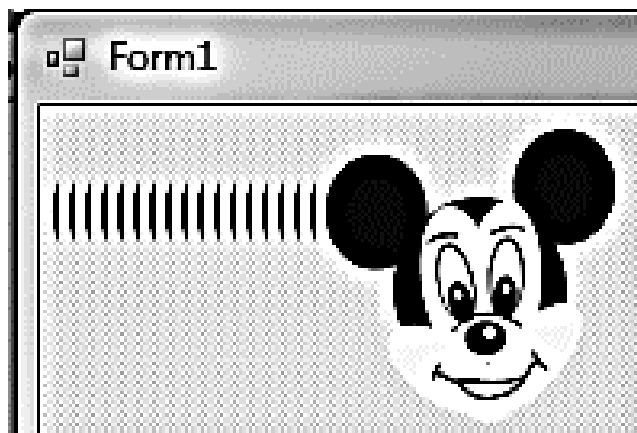


Рисунок 1.31. Результат анимации

Как мы видим, на экране остается часть старого изображения. Именно это является недостатком использования глобального Bitmap – он хранит результаты предыдущих отрисовок. Для того чтобы это ликвидировать, мы должны перед изменением координат объекта отрисовать прямоугольник, равный по размерам Bitmap, зали-

тый цветом фона. Однако, если наш фон – изображение, то мы должны будем рисовать не прямоугольник, а выводить это изображение.

Теперь рассмотрим способ организации взаимодействия игры и пользователя. Как правило, в самом простом случае для управления игровыми объектами игрок использует мышь и клавиатуру. Значит, наша игра должна уметь обрабатывать события нажатия на определенные клавиши и клики мыши. Мы умеем привязывать обработку клика мыши к объекту `button`, но все наши игровые объекты будут располагаться на `pictureBox`, значит, обработку клика нужно привязать к нему. Кроме того, на нашем игровом поле могут присутствовать несколько объектов, различающиеся координатами. Значит, нам нужно будет сравнивать координаты курсора мыши с координатами объектов, определяя, попал ли курсор в область объекта. Программно это реализуется с помощью граничных условий.

На рисунке 1.32 показано иллюстрация к этим условиям для квадрата и курсора, а также объединение их в одно логическое выражение. На картинке x и y – координаты левого верхнего угла прямоугольника, w – его ширина/высота, а x_k и y_k – координаты курсора.

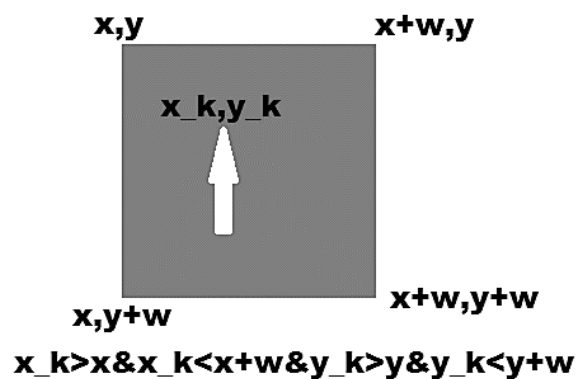


Рисунок 1.32. Попадание в область объекта

То есть мы кликнули мышкой на объект, если для координат курсора и объекта выполняется указанное условие.

Теперь, что касается привязки обработки события: **ее нельзя выполнять двойным щелчком на `pictureBox`**. Так как нам нужны координаты курсора мыши, то нас интересует обработка события

MouseDownClick, при котором наша программа отследит не только факт клика, но и координаты курсора. При двойном щелчке создается обработка простого события Click, которое в данном случае нам не подходит. Привязка события осуществляется через окно Properties: необходимо нажать на значок молнии, найти событие mouseClicked и уже на пустом поле рядом с ним дважды кликнуть мышью (рисунок 1.33).

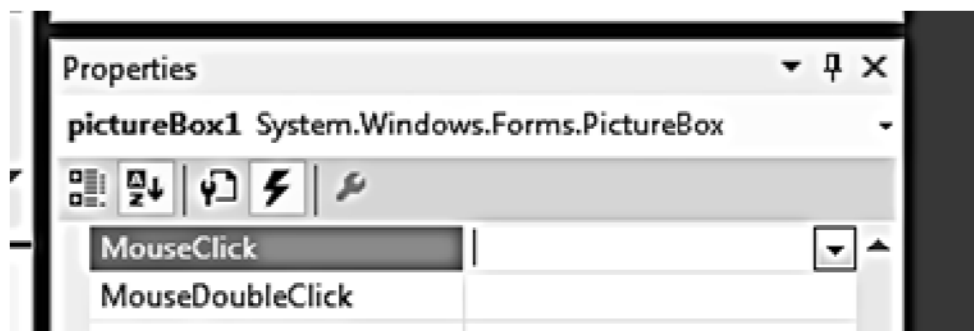


Рисунок 1.33. Привязка события клика мыши

Для примера, разработаем программу, в которой при нажатии на кнопку будет в случайном месте отрисовываться прямоугольник случайного размера, а при клике на pictureBox будет выводиться сообщение о попадании или непадании в него курсором. Так как отрисовка прямоугольника будет происходить по кнопке, а отслеживание координат – в клике на pictureBox, значит, они должны быть объявлены глобально. То же самое должно быть выполнено и с параметром ширины. В методе MouseClick к координатам курсора можно добраться с помощью следующих конструкций: e.X – координата x, e.Y – координата y. Код, реализующий это приложение, приведен в листинге 1.46.

Листинг 1.46

```
Bitmap bm; Graphics gr; int x = 0; int y = 0; int w = 0;
Random r = new Random();
public Form1(){ InitializeComponent();
bm = new Bitmap(pictureBox1.Width, pictureBox1.Height);
gr = Graphics.FromImage(bm);}
```

Окончание листинга 1.46

```
private void button1_Click(object sender, EventArgs e){
w = r.Next(10, 50);
x = r.Next(0, pictureBox1.Width - w);
y = r.Next(0, pictureBox1.Height - w);
gr.FillRectangle(Brushes.Red, x, y, w, w);
pictureBox1.Image = bm;}
private void pictureBox1_MouseClick(object sender, MouseEventArgs e){
if (e.X > x && e.X < x + w & e.Y > y & e.Y < y + w) MessageBox.Show("Попали!");
else MessageBox.Show("Мимо!");}
```

Теперь поговорим о реакции на клавиатуру. С нажатием клавиш связано несколько событий, но проще всего определить, какая клавиша была нажата, через событие `PreviewKeyDown`. Однако в его работе есть один нюанс. Так как нажатие клавиши относится ко всему приложению, то событие перехватывается находящимся в фокусе объектом. Это может быть и `pictureBox`, и кнопка, и форма. Причем в разное время работы программы объект в фокусе может быть разным. Поэтому, если у вас на форме несколько объектов, способных обрабатывать нажатие клавиш, обработку нужно привязывать ко всем. Сделать это можно следующим образом: комбинацией клавиш `Ctrl+A` выделить все объекты, далее перейти в события (нажав в `Properties` молнию) и, как обычно, двойным щелчком привязать метод к `PreviewKeyDown`. Обратите внимание, если у вас на форме есть таймер, то выделив его вместе со всеми объектами, вы не увидите событие `PreviewKeyDown`, так как таймер не умеет на него реагировать. Исключить его из списка выделенных можно, зажав `Ctrl` и кликнув на таймере мышью.

Для примера разработаем приложение, в котором по нажатию на кнопку отрисовывается прямоугольник, по нажатию стрелок на клавиатуре он будет перемещаться по экрану, а по нажатию пробела – менять цвет на случайный. Для получения значения нажатой клавиши воспользуемся конструкцией `e.KeyCode`. Для определения, какая именно была нажата, сравним это значение с интересующими нас. Например,

выражение `e.KeyCode == Keys.Left` – проверка, была ли нажата стрелка влево. Нам будут нужны сравнения со следующими значениями: `Keys.Left`, `Keys.Right`, `Keys.Up`, `Keys.Down`, `Keys.Space`. Код, реализующий программу, приведен в листинге 1.47.

Листинг 1.47

```
Bitmap bm;
Random r=new Random();
Graphics gr;
SolidBrush sb = new SolidBrush(Color.Red);
int x = 0; int y = 0;int w = 0;

public Form1() {
InitializeComponent();
bm = new Bitmap(pictureBox1.Width, pictureBox1.Height);
gr = Graphics.FromImage(bm);
}

private void button1_Click(object sender, EventArgs e)
{w = r.Next(10, 50);
x = r.Next(0, pictureBox1.Width - w);
y = r.Next(0, pictureBox1.Height - w);
gr.FillRectangle(sb, x, y, w, w); }

private void button1_PreviewKeyDown(object sender, PreviewKeyDownEventArgs
e)
{
if (e.KeyCode == Keys.Left) x--;
if (e.KeyCode == Keys.Right) x++;
if (e.KeyCode == Keys.Up) y--;
if (e.KeyCode == Keys.Down) y++;
if (e.KeyCode == Keys.Space)
sb.Color = Color.FromArgb(r.Next(0, 255), r.Next(0, 255), r.Next(0, 255));
gr.FillRectangle(sb, x, y, w, w);
pictureBox1.Image = bm;}
```


Обратите внимание!

- *Так как начало координат находится в левом верхнем углу экрана, то по нажатию стрелки вверх координата у уменьшается, вниз – увеличивается.*

Теперь рассмотрим создание нескольких игровых уровней, а также окон с настройками. При работе с GDI+ это реализуется через добавление в проект дополнительных форм. В приложении форм может быть сколько угодно. Дополнительные могут открываться и закрываться по ходу работы, не влияя на работу самого приложения, но всегда есть одна форма, называемая основной. Она создается при запуске проекта, и приложение работает, только пока она не закрыта. Как правило, основной становится первая созданная в проекте форма, но это можно изменить, открыв файл Program.cs. В нем есть строка кода `Application.Run(new Form1());` Если вместо Form1 прописать другую форму – она станет основной.

Чтобы добавить в проект форму, переходим в окно Solution Explorer (рисунок 1.34), щелкаем правой кнопкой мыши на имени проекта и выбираем в контекстном меню Windows Form.

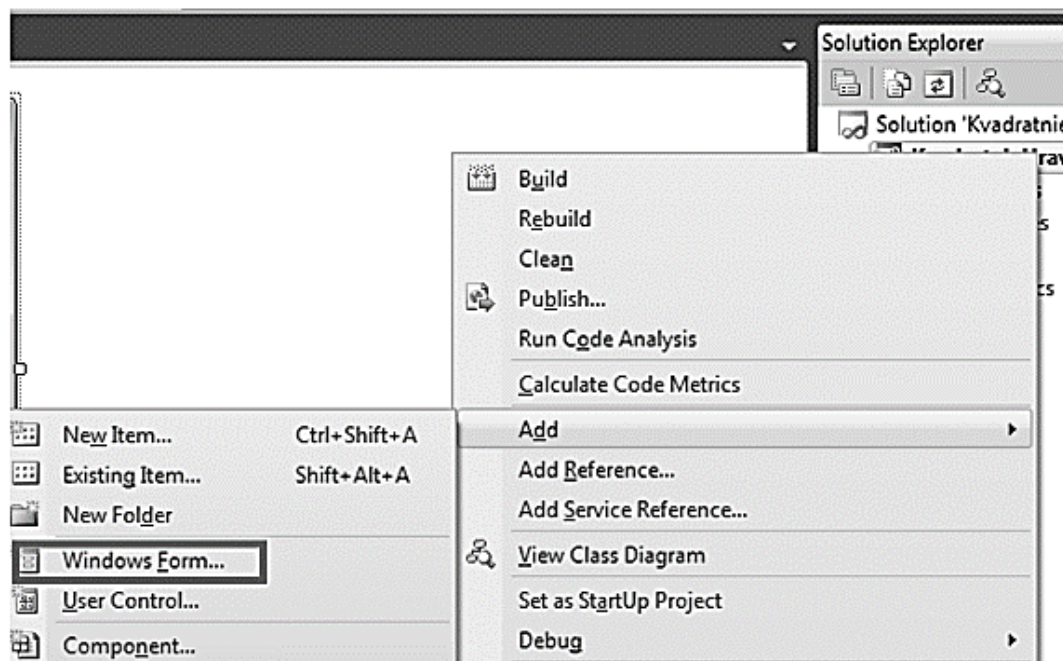


Рисунок 1.34. Добавление формы в проект

Для того чтобы новая форма (Form2) появилась на экране, например по нажатию кнопки, в методе-обработчике соответствующего нажатия необходимо написать следующий код (листинг 1.48).

Листинг 1.48

```
Form2 f=new Form2();  
f.ShowDialog();
```

Дополнительные формы можно закрывать методом Close(), а основную форму – только прятать методом Hide().

Для примера разработаем следующую программу: пользователь на первой форме выбирает, в каких координатах нужно отрисовать круг, затем по нажатию кнопки открывается вторая форма, и этот круг отрисовывается на ней. При этом первая форма исчезает, а при закрытии второй – появляется.

Итак, создаем проект, добавляем в него дополнительную форму и переходим к проектированию их интерфейса. На первой форме нам нужно разместить два textBox-а и кнопку, на второй – pictureBox и кнопку. Теперь мы можем перейти к кодированию, однако здесь нас ожидает проблема: нам нужно передать значения из одной формы в другую. Когда мы передавали значения между методами формы, мы пользовались глобальными переменными, объявленными в теле формы, но вне тела методов. Сейчас нам нужны глобальные переменные более высокого уровня, те, которые будут видны во всех формах проекта. Данную проблему можно решить несколькими способами. Рассмотрим самый простой. Переходим в файл Program.cs и переменные x и y объявляем в нем после метода Main, как отражено в листинге 1.49. Теперь эти переменные, а вернее поля, доступны нам в любом месте кода через обращение: Program.x и Program.y.

Таким образом, код обработки нажатия на кнопку в первой форме будет следующим (листинг 1.50).

Листинг 1.49

```
[STAThread]
static void Main() {
Application.EnableVisualStyles();
Application.SetCompatibleTextRenderingDefault(false);
Application.Run(new Form1());}
public static int x; public static int y;
```

Листинг 1.50

```
private void button1_Click(object sender, EventArgs e){
Program.x = Convert.ToInt32(textBox1.Text);
Program.y = Convert.ToInt32(textBox2.Text);
this.Hide();
Form2 f = new Form2();
f.ShowDialog();
this.Show();}
```

Обратите внимание!

- Строка кода *this.Hide()* прячет первую форму перед открытием второй.
- Строка *this.Show()* отображает первую форму при закрытии второй.

В обработчиках загрузки второй формы и нажатия на кнопку будет написан следующий код (листинг 1.51).

Листинг 1.51

```
private void Form2_Load(object sender, EventArgs e){
Bitmap bm = new Bitmap(pictureBox1.Width, pictureBox1.Height);
Graphics gr = Graphics.FromImage(bm);
gr.FillEllipse(Brushes.Red, Program.x, Program.y, 10, 10);
pictureBox1.Image = bm;
}
private void button1_Click(object sender, EventArgs e){ this.Close(); }
```

Напомним, что метод `Form_Load` выполняется при загрузке формы, и его привязка осуществляется двойным щелчком по форме. Обратите внимание: щелчок должен выполняться именно на поле формы, а не на `pictureBox`-е или любом другом элементе.

Мы рассмотрели основы создания интерактивных интерфейсов средствами технологии GDI+. Теперь рассмотрим процесс разработки тестовой игры, но прежде вам предлагается выполнить несколько заданий по пройденному материалу.

ЗАДАНИЯ К ТЕМЕ 1.6

1. Исправьте код, реализующий иллюстрируемое на рисунке 1.31 так, чтобы при перемещении картинки не осталось следов.

2. Реализуйте предыдущее задание, разделив логику и интерфейс. Для этого определите класс с полями-координатами и полем-именем файла картинки, а также методом перемещения, принимающим в качестве параметра смещение по осям *x* и *y*.

3. Разработайте игровую заставку по следующей схеме (рисунок 1.35). При загрузке формы картинка «вырастает» из точки, текст1 – выезжает слева экрана, текст2 – справа, текст3 – снизу. Цветовую схему, а также содержание картинки и надписей придумать самостоятельно, однако итоговая заставка должна быть осмысленной.



Рисунок 1.35. Иллюстрация к заданию 3

4. Разработайте игру: в случайных координатах появляется круг. При попадании курсором в этот круг, он исчезает и перерисовывается в новых случайных координатах. Игроку за каждое попадание начисляется количество баллов, задаваемое им в окне настройки, которое появляется при запуске игры. После закрытия окна настроек появляется окно с игрой.

5. Разработайте игру: в случайных координатах появляется круг, а на противоположной стороне формы появляется квадрат. Игрок, управляя стрелками, должен довести квадрат до круга, а при достижении этого нажать пробел и получить определенное количество очков. После этого круг появляется в другом месте формы, и игровой процесс повторяется.

ТЕМА 1.7. РАЗРАБОТКА ИГРЫ «СПАСТИ ПРИНЦЕССУ»

В данной теме мы рассмотрим создание полноценной игры средствами GDI+ и программы обработки изображений – GIMP 2.8. Последнее понадобится нам для подготовки графического наполнения нашей игры, ведь, как было сказано выше, графика – это крайне важная составляющая. Но так как GIMP обладает огромными возможностями в плане работы с графикой, то в данной теме мы рассмотрим лишь те его функции, которые понадобятся нам для создания используемых в игре изображений. Более подробно с прочими функциями этого редактора вы можете ознакомиться в следующих ресурсах: [2, 3].

Однако графическое наполнение игры крайне сложно создать, не имея ее концепции, а также хотя бы примерного описания геймплея. Поэтому именно с этого мы и начнем.

Сценарий игры

Сценарий игры можно рассмотреть с двух позиций: концептуальной и геймплейной. Первая сторона и составляющая сценария должна описывать основную идею игры, действующих героев, конф-

ликт, мир и прочую общую канву. Геймплейная составляющая описывает в рамках созданной концепции конкретные сцены игры с действующими в них элементами, а также переходы между сценами.

Начнем с концепции. Для заявленного названия она может быть следующей: «Злой дракон похитил принцессу и спрятал ее в башне. Вам – отважному рыцарю, предстоит спасти похищенную. Для этого необходимо сделать три вещи. Во-первых, проникнуть в башню, открыв кодовый замок на ее двери за ограниченное число попыток. Во-вторых, убить дракона. В-третьих, построить лестницу, чтобы добраться до комнаты заточения принцессы».

Обратите внимание!

- *Обычно название игры придумывается уже после создания ее концепции, как фраза, лучшим образом отражающая ее суть. Однако в нашем случае эти действия поменялись местами, так как название было озвучено заранее. В принципе, такой подход тоже допускается, но используется реже.*

Основная концепция, а также история и конфликты героев в игре могут быть представлены по-разному. Например, вставками мини-роликов, отражающих самые главные моменты. Или же текстовыми описаниями в различных книгах и дневниках, «разбросанных» по игровому миру. Однако для небольших и простых игр основную их концепцию можно представить в виде текстово-иллюстрированных заставок. Именно этот вариант мы и реализуем, заранее подготовив изображения и составив из них анимированные заставки, рассказывающие историю игры и поясняющие действия в сцене.

Согласно концепции, у нас есть три главных героя: принцесса, дракон и рыцарь. Их изображения нам необходимо будет создать. Кроме того, у нас есть замОк, дверь, зал башни, лестница. Однако полностью представить, какие изображения нам понадобятся, будет гораздо проще, если разработать геймплейный сценарий игры. Описывать его можно с разной степенью подробности. Это зависит от

сложности игры, а также от договоренности внутри команды разработчиков. Так как наша игра весьма простая, нам подойдет средняя степень, которую можно реализовать, используя таблицы (см. таблицы 1.1 и 1.2).

Таблица 1.1. Элементы сценария игры

№ и имя сцены	Активные элементы	Переходы (№ сцен)
1. Основная заставка	3 изображения героев, текст	2
2. Меню	Фон, заглавие, кнопки (начать, настройка, выход)	3,6
3. Уровень 1.	Текстовое поле, кнопка, фон	4,8
4. Уровень 2.	Изображение дракона, фон, текст	5,8
5. Уровень 3.	Фон, элементы лестницы	7,8
6. Настройки	3 текстовых поля, 2 переключателя уровня сложности, 2 кнопки	2
7. Заставка победы	Фон	2
8. Заставка проигрыша	Фон	2

Таблица 1.2. Происходящее в сцене

№	Описание
1.	Черный фон, сверху – рамка для изображения дракона, справа – для принцессы, слева – для рыцаря. В центре – текст, постепенно появляющийся на экране. Как только появляется описание героя, его изображение появляется в соответствующей рамке. По щелчку мыши переход на следующую сцену, по Escape – закрытие игры.
2.	На фоновом изображении проявляется заглавие. Как только оно полностью появляется – начинает мигать, а слева и справа экрана выезжают кнопки. По щелчку мыши на кнопку – переход на соответствующую сцену, по Escape – закрытие игры.

№	Описание
3.	На фоновом изображении появляется текстовое поле и кнопка, а также генерируется случайное число. По нажатию на кнопку сравнивается введенное в текстовое поле число и сгенерированное. При совпадении – переход на следующую сцену, при несовпадении – уменьшение количества попыток и вывод результата сравнения чисел. При количестве попыток =0 – переход на 8 сцену. По Escape – закрытие игры.
4.	На фоновом изображении в случайном месте появляется дракон, исчезает и появляется в другом месте. Внизу – надпись с количеством его жизней и жизней игрока. По щелчку проверяется, попал ли курсор в дракона. При попадании – уменьшение количества жизней дракона, при промахе – игрока. Если количество жизней дракона =0, – переход на следующую сцену. Если количество жизней игрока =0, – переход на сцену 8. По Escape – закрытие игры.
5.	На фоновом изображении в разных местах экрана лежат фрагменты лестницы. Задача игрока: выбрать мышью блок и стрелками на клавиатуре довести его до нужного места. Как только все блоки собраны, – нужно нажать пробел. Если все блоки стоят на своем месте, – переход на сцену 7, иначе – на сцену 8. По Escape – закрытие игры.
6.	В первое текстовое поле вводится количество попыток для замка, во второе – количество жизней дракона, в третье – процент урона, наносимого с кликом. Уровень легко: на сцене с замком в углу написано загаданное число, дракон во втором уровне перемещается медленно. Уровень сложно: на сцене с замком в углу написаны первые две цифры загаданного числа, дракон перемещается быстро. Кнопка «сохранить» – настройки применяются, «отменить» – сохраняются по умолчанию. Каждая кнопка – возврат на сцену 2. По Escape – закрытие игры.

№	Описание
7,	Фоновое изображение. По щелчку мыши – переход на сцену 2,
8.	по Escape – закрытие игры.

Теперь, когда мы четко понимаем, что будет происходить в нашей игре, можно перейти к подбору и созданию изображений.

Подготовка изображений

Есть три пути получения изображения для игр. Во-первых, можно обратиться к сайту бесплатных изображений, такому как [4]. Посмотрите на выделенную область справа: именно здесь указано, что данное изображение вы можете использовать в своих проектах, не нарушая ни чьих прав (рисунок 1.36).

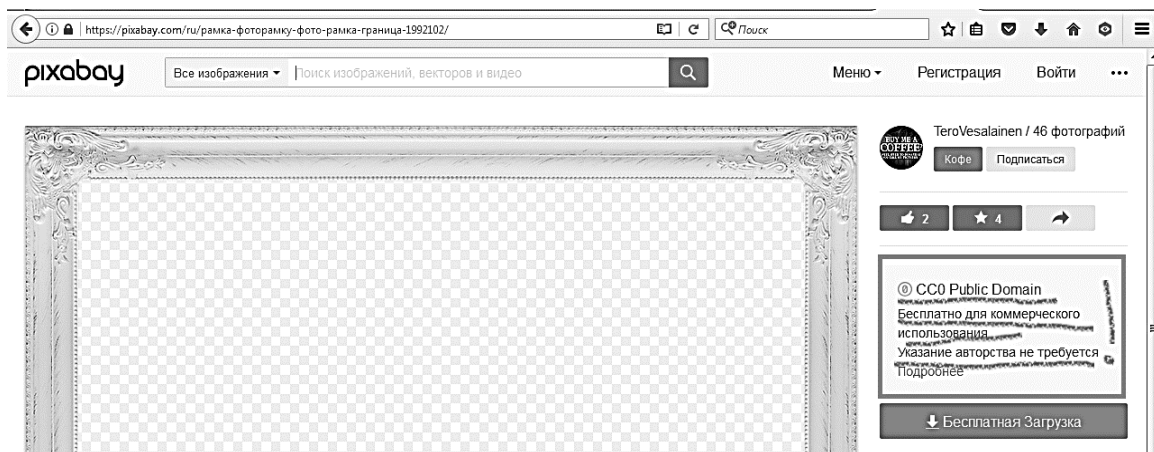


Рисунок 1.36. Сайт pixabay.com

На сайте [4] указано: «Все изображения на Pixabay выпущены по лицензии Creative Commons CC0 (Передача в общественное достояние). Таким образом, вы можете изменять изображения и свободно использовать их как для личных, так и для коммерческих целей, в цифровом или печатном виде. Указание авторства в этом случае не обязательно, но рекомендуется».

ИСКЛЮЧЕНИЯ:

Идентифицируемые люди на фотографиях не должны быть в ситуациях, которые могут их скомпрометировать или оскорбить их достоинство.

Нельзя использовать изображение людей или логотипов каких-либо брендов в качестве рекомендации к использованию вашей продукции. Например, нельзя располагать логотип НАСА рядом с вашим собственным, чтобы выглядело так, что НАСА одобряет ваш продукт.

В дополнение к этому есть еще ряд ограничений, которые описаны ниже... Изображения с идентифицируемыми на них людьми попадают под понятие Релиз Модели (то есть модель должна дать свое разрешение на использование данного изображения)... Это также касается и частной собственности... Использование таких изображений попадает под действие Релиза Собственности. Владелец собственности должен дать согласие на использование изображения его имущества. Под действие этого релиза также попадают некоторые здания, памятники или модели техники, которые охраняются законом. Например, если вы хотите использовать фото какой-нибудь новой модели ноутбука или телефона, изображение здания Chrysler Building в Нью-Йорке или лондонского колеса обозрения, то вам нужно предварительно получить разрешение у собственника. Нельзя также использовать изображения ночной подсветки Эйфелевой башни (причем днем таких ограничений нет).

Кроме того, на некоторых изображениях вы можете встретить чужие логотипы. Например, название марки часов или фотоаппарата, название магазина или кафе. В таком случае без дополнительного разрешения от собственника на коммерческое использование изображения вам не обойтись. Но есть и другой выход — достаточно просто заретушировать логотип или название. Если же речь идет о размещении изображения у себя в блоге, то даже этого делать не придется.

Существует также разница между редакторским и коммерческим использованием изображения. Релиз модели и Релиз собственности особо актуальны при использовании изображений в коммерче-

ских целях. Если вы используете фотографию, например, для своего блога, то это считается не коммерческим, а редакторским использованием. Для этих целей вам не понадобятся дополнительные разрешения. Под коммерческим использованием подразумевается использование изображения для любого вида бизнеса. Особенно важно не нарушать права при использовании фото для больших тиражей.

Итог: нюансы правильного использования изображений могут показаться сложными, но на самом деле все достаточно логично. Просто попробуйте поставить себя на место модели и собственника имущества. Задайте себе вопрос: «Одобрили ли бы вы использование этого изображения третьим лицом без предварительного вашего согласия?» Просто задавайте себе этот вопрос каждый раз, когда сомневаетесь в правильности использования какого-либо снимка.

Выводы: главное понять, что «общественное достояние» — это разрешение на использование авторских прав на само изображение (работу автора). Но ответственность за использование содержания изображения лежит именно на вас».

Основное достоинство данного варианта — простота использования. Основные недостатки: во-первых, сложно реализовать что-то уникальное и единostильное, используя готовые изображения. Во-вторых, в вопросе прав тоже могут быть свои тонкости и нюансы. В-третьих, сложно найти именно то, что нужно.

Второй вариант получения изображений: самостоятельная отрисовка. Основное достоинство этого варианта: вы можете сделать все, что захотите. Основной недостаток: умение рисовать, а также наличие таких технических средств, как сканер и графический планшет.

Третий вариант — заказать у профессионального художника. Основное достоинство: вы можете сделать все, что захотите. Основной недостаток: найти художника, который сделает именно то, что вы хотите, а также существенные финансовые затраты. На 2017-й год ориентировочная цена черно-белого карандашного изображения — от 1000 руб., цветного — от 3000. В зависимости от квалификации художника цифры могут меняться в большую или меньшую сторону.

В рамках примеров для данной книги будут использованы первый и второй варианты.

Начнем с основной заставки. Для ее реализации нам потребуются изображения дракона, рыцаря, принцессы, рамки и текста. Первые три изображения нарисует самостоятельно простым карандашом и раскрасим в программе GIMP 2.8. Мы можем нарисовать только контуры, а цветовую раскраску (со всеми тенями) сделать в программе, или же можем нарисовать карандашом и тени, а в программе наложить только цветовые тона. На рисунке 1.37 представлено изображение, выполненное первым способом, а на рисунке 1.38 – вторым. Как видите, изображения имеют совершенно разный стиль. И необходимо добавить, что первый способ практически нереализуем без графического планшета, тогда как во втором случае он не требуется.



Рисунок 1.37. Карандашный эскиз и итог

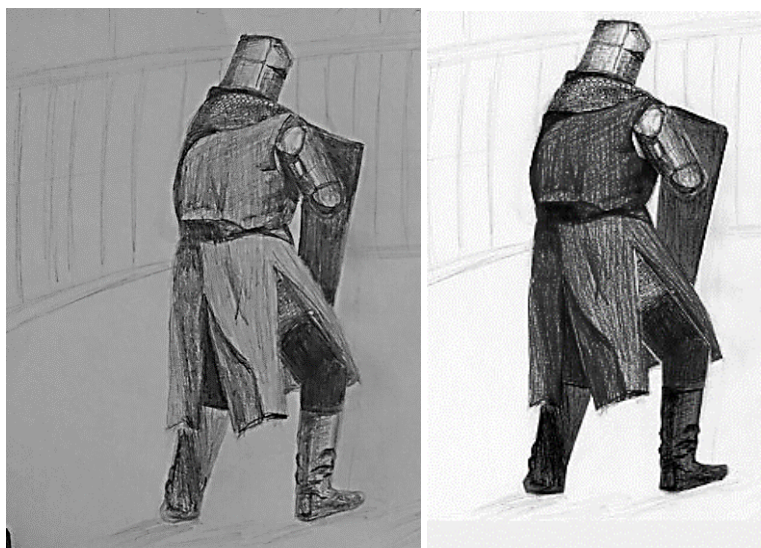


Рисунок 1.38. Карандашный вариант и итог

Мы будем использовать второй вариант. В редакторе GIMP реализовать это крайне просто. Отсканированную картинку, которую необходимо раскрасить, открываем в редакторе. Для выполнения раскраски вам понадобятся выделенные на рисунке 1.39 элементы.

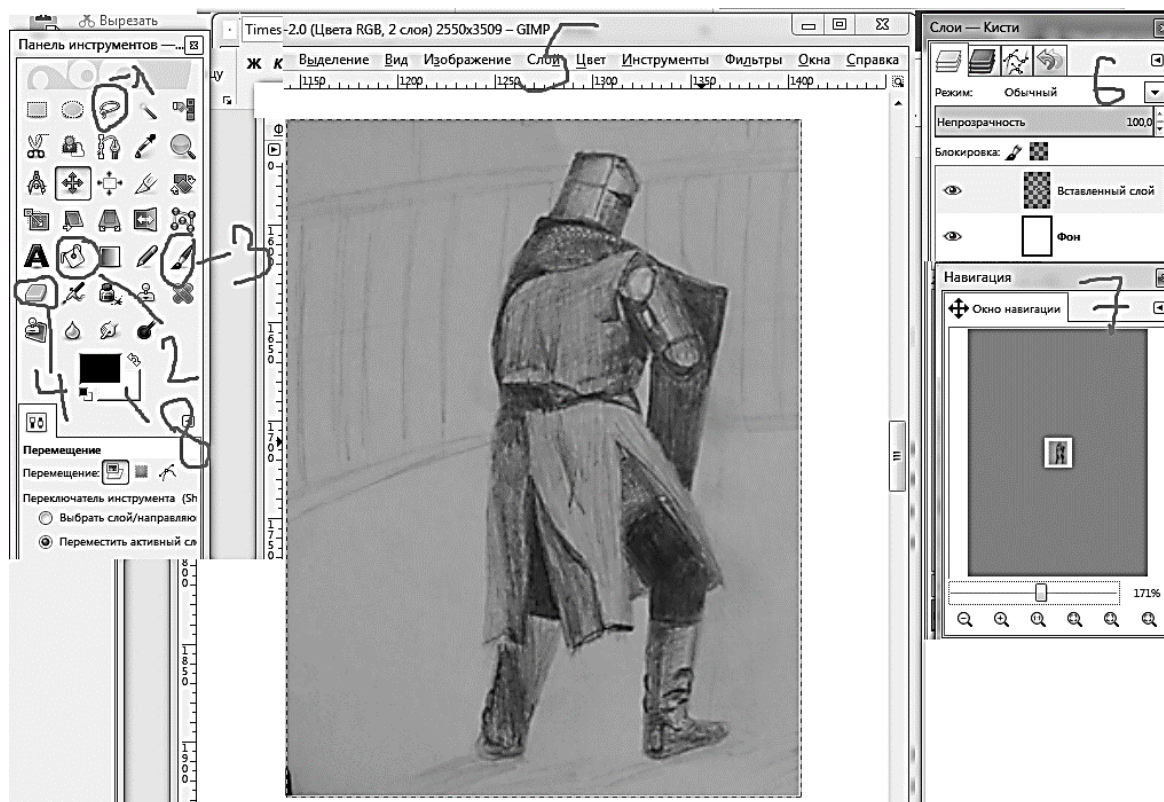


Рисунок 1.39. Основное окно GIMP

Идея метода следующая: с помощью элемента 1 («Произвольное выделение»: щелчком мыши отмечаете начальную точку, а дальше устанавливаете щелчками опорные точки по контуру выделения и возвращаетесь в исходную точку, как показано на рисунке 1.40) выделяется область в окне 5, которую вы хотите закрасить.

Если область очень маленькая, то в окне 7 можно увеличить масштаб так, чтобы в окне 5 была только нужная вам часть картины. Далее в окне 6 создается новый слой (правой кнопкой мыши вызываем контекстное меню и выбираем «Создать слой»). Затем с помощью инструмента 8 выбирается нужный цвет и с помощью инструмента 2 он наносится на область (щелчком на выделенном участке), как проиллюстрировано рисунком 1.41.

Обратите внимание!

- *Каждую область лучше всего закрашивать на отдельном слое, который должен располагаться выше загруженной картинке.*

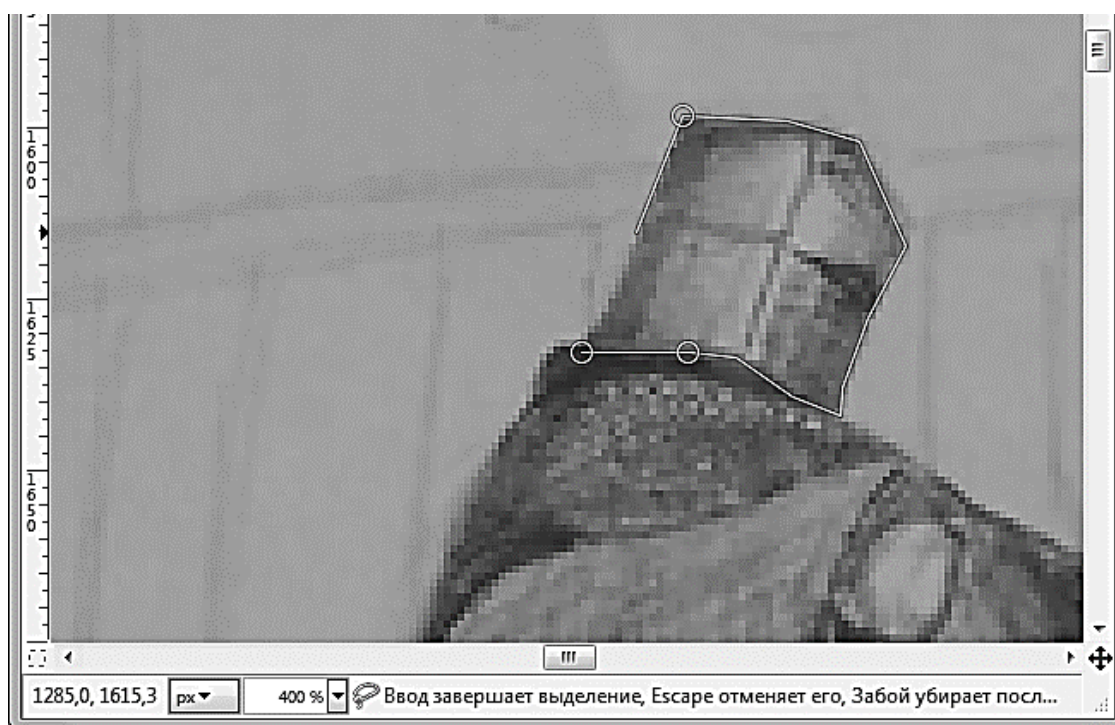


Рисунок 1.40. Выделение области



Рисунок 1.41. Заливка области

Для совсем маленьких участков, которые сложно выделить, используется инструмент 3. Если вдруг вы закрасили лишнее – инструмент 4. Далее, после того как область закрашена, в окне 6 выберите режим отображения «Затемнение». На рисунке 1.42 для примера окрашен шлем рыцаря.

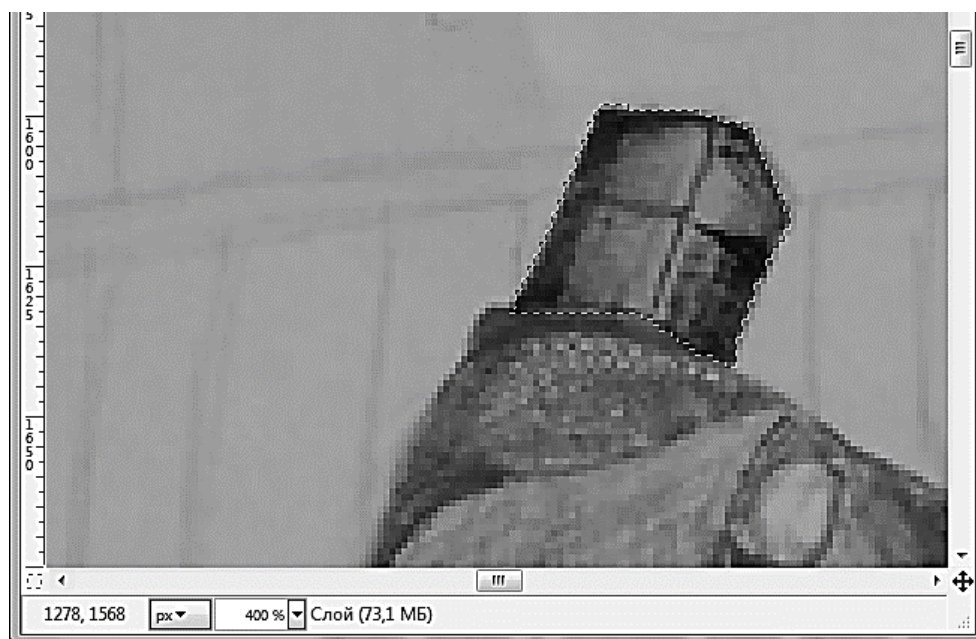


Рисунок 1.42. Применение режима «Затемнение»

Таким образом создадим изображения для нашей основной заставки (рисунки 1.43,1.44,1.45).



Рисунок 1.43. Изображение принцессы



Рисунок 1.44. Дракон и башня



Рисунок 1.45. Рыцарь


Рамочку возьмем с сайта pixabay и с помощью GIMP придадим ей золотистый цвет. Для этого откроем ее изображение в редакторе, добавим новый слой, зальем его нужным нам цветом и применим «затемнение». Кроме этого сделаем макет заставки (рисунок 1.46) и макет сцены с меню (рисунок 1.47). И хоть по стилю изображения все-таки отличаются, так как не рисовались специально для игры, но общая картина нас устраивает.



Рисунок 1.46. Макет заставки



Рисунок 1.47. Меню игры

Для создания текста нам потребуется в GIMP инструмент «Текст» () , а для обводки – кисть определенного шаблона, который можно выбрать в окне, показанном на рисунке 1.48.

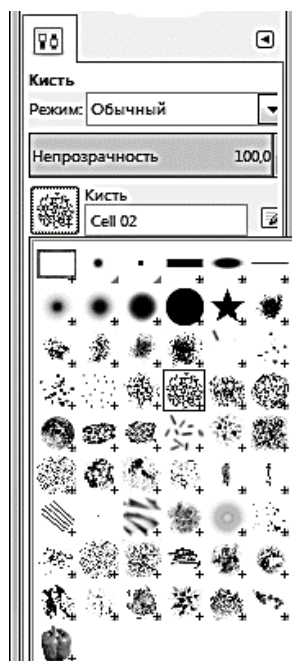


Рисунок 1.48. Выбор кисти

В таблице 1.3 приведен полный перечень необходимых для игры изображений, часть из которых вам предлагается подготовить само-

стоятельно. Приведенные имена файлов в дальнейшем будут использоваться в коде.

Таблица 1.3. Перечень изображений

№ сцены	Имя файла	Описание
1	MainF.png	Картинка с текстом истории
1	Princes.png	Изображение принцессы
1	Knigth.png	Изображение рыцаря
1	DragonZ.png	Изображение дракона
2	Fon.png	Черный прямоугольник с желтой рамкой для фона
2	Yarko.png	Заголовок игры («Спасти принцессу») в яркой цветовой гамме на черном фоне
2	Bleklo.png	Заголовок игры («Спасти принцессу») в приглушенной цветовой гамме на черном фоне
2	Begin.png	Изображение с текстом «Начать»
2	Nastroiki.png	Изображение с текстом «Настройки»
2	Vihod.png	Изображение с текстом выход
3	FonUr1.png	Изображение: дверь на черном фоне
3	Button.png	Изображение для кнопки в виде старинного дверного молотка
4	Dragon.png	Изображение монстра-дракона
4	FonB.png	Изображение залы башни, в которой происходит бой с драконом
5	FonLestn.png	Фоновое изображение последнего уровня
5	Fragm1.png	Первый фрагмент лестницы
5	Fragm2.png	Второй фрагмент лестницы
5	Fragm3.png	Третий фрагмент лестницы
5	Fragm4.png	Четвертый фрагмент лестницы
7	FonVict.png	Изображение для победы
8	GameOver.png	Изображение для поражения

Обратите внимание!

- Так как заставки в нашей игре будут анимироваться, то необходимо каждый из анимируемых изображений сохранить отдельно.
- Так как у некоторых изображений должен быть прозрачный фон (например, у дракона), то файлы сохранены в формате png. Именно он позволяет сохранять прозрачный фоновый слой для картинок.

На рисунках 1.49, 1.50, 1.51 и 1.52, представлены соответствующие макеты сцен-уровней, которые вам предлагается воспроизвести, или вы можете предложить собственные. Для победной и проигрышной заставки макеты предлагается придумать самостоятельно, как и для окна настроек.

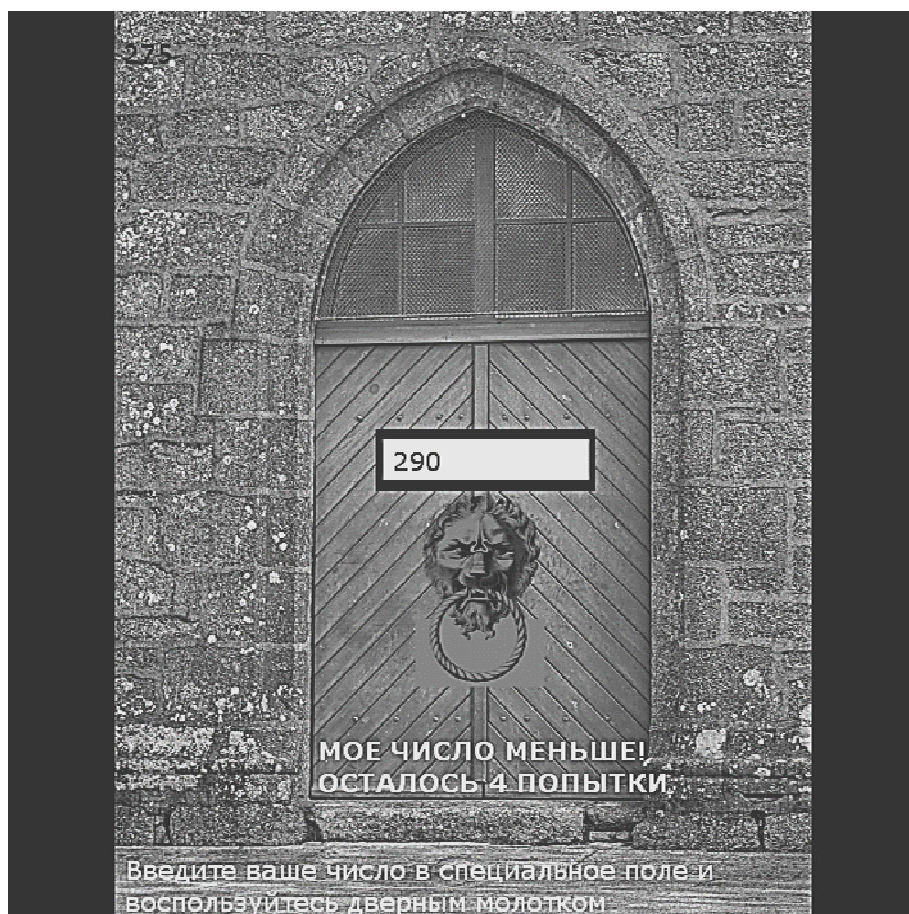


Рисунок 1.49. Макет первого уровня

Посмотрите внимательно на верхний левый угол картинки – без этой подсказки игра становится непроходимой. В зависимости от уровня сложности можно выводить все число или только сотни и десятки. Текст выводится методом DrawString, а изображение молотка можно или отрисовать через DrawImage или поместить обычный элемент Button, у которого в свойстве BackgroundImage выбрать картинку Button.png. В этом случае не придется отслеживать, куда пользователь попал мышкой, а использовать обычную обработку нажатия на кнопку. Сами изображения взяты с сайта pixabay.com.



Рисунок 1.50. Макет второго уровня



Рисунок 1.51. Начальный вид третьего уровня



Рисунок 1.52. Победный вид третьего уровня

Обратите внимание!

- *На рисунках 1.51 и 1.52 приведены макеты, а итоговый вид может отличаться в деталях.*

Теперь, после того как мы подготовили изображения, можно переходить к кодированию.

Реализация игры

Далее будет рассмотрено создание заставки-меню, а также разработка последнего игрового уровня. Все остальное вам предлагается реализовать самостоятельно, используя уже изученный материал. Ниже приведен небольшой список подсказок, которые вам могут понадобиться:

- Выход из игры: `Application.Exit();`
- У вас будет восемь сцен. То есть восемь форм.
- Запуск таймера: `timer1.Enabled=true;`
- Изменение интервала таймера: `timer1.Interval=500;`

- Работа с генератором случайных чисел: глобальное объявление `Random r = new Random();` получение случайного числа осуществляется кодом: `b = r.Next(0, 100);`
- Открытие новой формы и скрывание предыдущей: `this.Hide(); Form2 f = new Form2(); f.ShowDialog();`
- Проверка нажатия мыши: `pictureBox1_MouseClick`. Проверяем `e.X`, `e.Y`.
- Проверка нажатия клавиши клавиатуры: `pictureBox1_PreviewKeyDown`. Сравниваем `e.KeyCode` с `Keys.Up`, `Keys.Down`, `Keys.Left`, `Keys.Right`.
- Отрисовка картинок: `gr.DrawImage(Image.FromFile("1.jpg"), x, y, 20, 20);` При этом файл с именем `1.jpg` должен лежать в папке `bin/Debug` проекта.
- Вывод текста: `gr.DrawString("Выводимый текст", new Font("Arial", 12), Brushes.Yellow, x, y);`

Теперь рассмотрим создание анимированной заставки меню. Для этого создадим новый проект и разместим на форме `pictureBox`. Далее немного изменим свойства формы. Игры, как правило, запускаются в полноэкранном режиме, поэтому, чтобы наша заставка развернулась на весь экран, переходим к свойству `WindowState` формы и ставим его в значение `Maximized`. Далее, свойство `StartPosition` меняем на `CenterScreen`, свойство `ControlBox` – на `false`, а свойство `FormBorderStyle` – на `None`. Кроме того, изменяем цвет фона на черный. В итоге у вас должно получиться нечто похожее на рисунок 1.53.

Обратите внимание!

- *В этом варианте `pictureBox` тоже поменял свой цвет фона на черный, если у вас этого не произошло – не страшно. На `pictureBox` мы все равно будем выводить фоновое изображение, поэтому его цвет на столь важен.*

Теперь нужно сделать еще две важные подготовительные вещи. Во-первых, при запуске программы наша форма растянется на весь экран, значит, и pictureBox должен обрести эти же размеры. Во-вторых, так как команда закрытия приложения нам будет недоступна, то лучше заранее позаботиться о том, чтобы по нажатию клавиши Escape игра завершала свою работу.

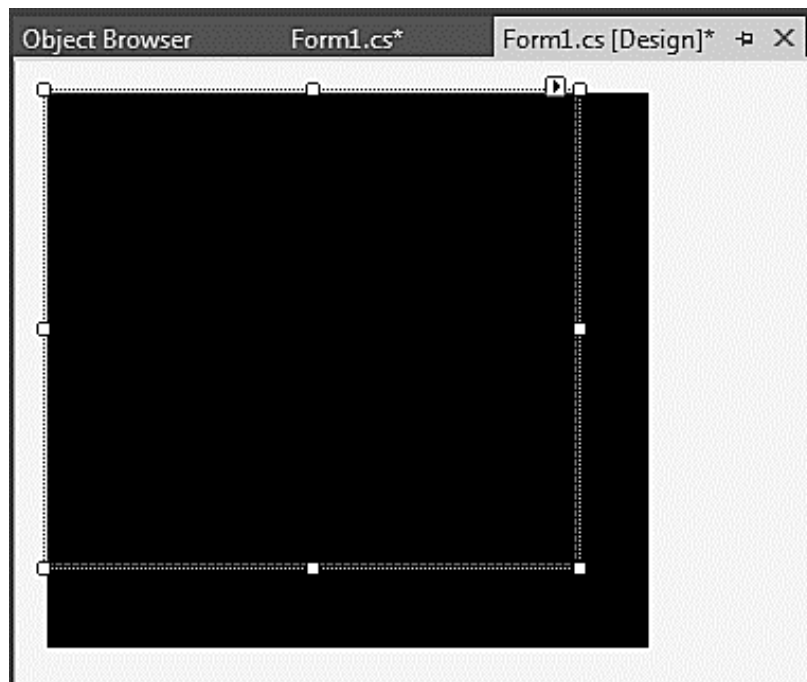


Рисунок 1.53. Форма с измененными свойствами

Для реализации первого пункта переходим в `Form_Load` и пишем там код (листинг 1.52). Таким образом, pictureBox примет размеры формы.

Листинг 1.52

```
private void Form1_Load(object sender, EventArgs e)
{ pictureBox1.Width = this.Width;
  pictureBox1.Height = this.Height;}
```

Для реализации второго пункта через `Ctrl+A` выделяем элементы и переходим к обработке события `PreviewKeyDown`. В нем пишем код (листинг 1.53).

Листинг 1.53

```
if (e.KeyCode == Keys.Escape) Application.Exit();
```

Теперь перейдем непосредственно к созданию заставки. Согласно сценарию, заглавие игры должно постепенно появляться на экране, а после этого кнопки должны выехать слева и справа. Кроме того, название игры должно мигать. Первый эффект мы можем реализовать, если поверх изображения названия отрисуем черный прямоугольник, который с течением времени будет менять свою прозрачность, постепенно исчезая. Для реализации этого нам понадобится таймер, переменная, отвечающая за прозрачность и кисть сплошной заливки. Кроме того, стандартные `Bitmap` и `Graphics`. Причем все переменные – глобальные. После их объявления при загрузке формы мы отрисуем фоновую картинку, название, прямоугольник и запустим таймер, интервал которого в свойствах установлен в 1. Код, реализующий это, представлен в листинге 1.54.

Листинг 1.54

```
Graphics gr; Bitmap bm;
SolidBrush sb = new SolidBrush(Color.Black); int a = 255;
private void Form1_Load(object sender, EventArgs e) {
    pictureBox1.Width = this.Width;
    pictureBox1.Height = this.Height;
    bm = new Bitmap(pictureBox1.Width, pictureBox1.Height);
    gr = Graphics.FromImage(bm);
    gr.DrawImage(Image.FromFile("Fon.png"), 0, 0, pictureBox1.Width,
pictureBox1.Height);
    gr.DrawImage(Image.FromFile("Yarko.png"), pictureBox1.Width / 6,
40, 2 * pictureBox1.Width / 3, pictureBox1.Height / 2);
    gr.FillRectangle(sb, pictureBox1.Width / 6, 40, 2 * pictureBox1.Width / 3,
pictureBox1.Height / 2);
    pictureBox1.Image = bm;
    timer1.Enabled = true; }
```

Теперь переходим к таймеру. В нем нам нужно уменьшить значение прозрачности, изменить характеристики кисти, отрисовать название и прямоугольник и отследить, если прозрачность стала меньше 0, то остановить анимацию. Код, реализующий это, представлен в листинге 1.55.

Обратите внимание!

- 255 – максимальный уровень непрозрачности.
- Максимальные координаты, а также ширина и высота объектов выражены относительно ширины и высоты *pictureBox*. Это сделано для того, чтобы изображение масштабировалось в зависимости от разрешения экрана, на котором запустится игра.

Листинг 1.55

```
private void timer1_Tick(object sender, EventArgs e) {  
    a -=5;  
    if (a < 0) { timer1.Enabled=false; }  
    else {  
sb.Color = Color.FromArgb(a, sb.Color);  
gr.DrawImage(Image.FromFile("Yarko.png"), pictureBox1.Width / 6, 40, 2 *  
pictureBox1.Width / 3, pictureBox1.Height / 2);  
gr.FillRectangle(sb, pictureBox1.Width / 6, 40, 2 * pictureBox1.Width / 3,  
pictureBox1.Height / 2);  
pictureBox1.Image = bm; } }
```

Теперь перейдем к анимации мигания и выезжания кнопок. Для этого в глобальные переменные добавляем еще одну (`int x=0`).

В таймере вместо его остановки изменим координату `x`, если ее значение меньше выбранного нами значения, в котором должны остановиться кнопки. Иначе – остановим анимацию, остановив таймер. Затем отрисуем в ней кнопки: две слева экрана, одну справа, не забыв

перед всей анимацией заново отрисовать фон, чтобы не оставлять следов предыдущего положения кнопок.

Для реализации мигания заголовка проверим значение прозрачности (ведь его изменение мы так и не прекратили). Если оно четно – отрисуем картинку названия в приглушенной цветовой гамме, иначе – в яркой. Для меньшей частоты мигания увеличим интервал таймера до ста. Итоговый метод обработки тика таймера приведен в листинге 1.56.

Листинг 1.56

```
private void timer1_Tick(object sender, EventArgs e){
gr.DrawImage(Image.FromFile("Fon.png"), 0, 0, pictureBox1.Width,
pictureBox1.Height);
a -=5;
if (a < 0) { timer1.Interval = 100;
if (a%2!=0)
    gr.DrawImage(Image.FromFile("Yarko.png"), pictureBox1.Width / 6, 40, 2 *
pictureBox1.Width / 3, pictureBox1.Height / 2);
    else
gr.DrawImage(Image.FromFile("Bleklo.png"), pictureBox1.Width / 6, 40, 2 *
pictureBox1.Width / 3, pictureBox1.Height / 2);
if (x < 2*pictureBox1.Width / 3- pictureBox1.Width / 6) x += 10;
else timer1.Enabled=false;
gr.DrawImage(Image.FromFile("Begin.png"), x, 50 + pictureBox1.Height / 2,
pictureBox1.Width / 6, pictureBox1.Height / 12);
gr.DrawImage(Image.FromFile("Nastroiki.png"), pictureBox1.Width- x, 150 +
pictureBox1.Height / 2, pictureBox1.Width / 6, pictureBox1.Height / 12);
gr.DrawImage(Image.FromFile("Vihod.png"), x, 250 + pictureBox1.Height / 2,
pictureBox1.Width / 6, pictureBox1.Height / 12);
pictureBox1.Image = bm;} else {
sb.Color = Color.FromArgb(a, sb.Color);
gr.DrawImage(Image.FromFile("Yarko.png"), pictureBox1.Width / 6, 40, 2 *
pictureBox1.Width / 3, pictureBox1.Height / 2);
gr.FillRectangle(sb, pictureBox1.Width / 6, 40, 2 * pictureBox1.Width / 3,
pictureBox1.Height / 2);
pictureBox1.Image = bm; }}
```

Теперь реализуем переход на игровой уровень по нажатию кнопки «Начать». Переходим в событие `MouseClicked` у `pictureBox`. В нем проверяем, если анимация закончена и наш курсор попал в область кнопки «Начать», – вызываем вторую форму. Координаты области взяты из метода вывода соответствующей картинке, а также из условия остановки анимации. Код, реализующий описанное, представлен в листинге 1.57.

Листинг 1.57

```
private void pictureBox1_MouseClick(object sender, MouseEventArgs e) {  
    if (timer1.Enabled==false){  
        if (e.X > 2 * pictureBox1.Width / 3 - pictureBox1.Width / 6 && e.X < (2 *  
pictureBox1.Width / 3 - pictureBox1.Width / 6) + pictureBox1.Width / 6 && e.Y > 50 +  
pictureBox1.Height / 2 && e.Y < 50 + pictureBox1.Height / 2 + pictureBox1.Height /  
12)  
        {  
            Form2 f = new Form2();this.Hide();  
            f.ShowDialog();}}}
```

Заставка меню реализована, и мы можем перейти к реализации игрового уровня с перемещением фрагментов лестницы. Во-первых, изменим свойства формы уровня точно также, как это сделано у заставки меню, во-вторых, разместим на ней `pictureBox`, позаботившись о том, чтобы при загрузке формы их размеры сравнялись, и также повесим на `Escape` выход из приложения.

Для разработки данного уровня нам потребуется хранить текущие координаты четырех фрагментов лестницы, координаты их точек назначения, а также имена файлов с картинками. Кроме того, нам нужно будет как-то запоминать, какой из фрагментов в данный момент выбран пользователем. Если мы все это будем реализовывать через переменные, то нам потребуется около ДВАДЦАТИ ЧЕТЫРЕХ переменных, что не очень удобно и оптимально. Более предпочтительным вариантом в данном случае будет объединение информации, описывающей фрагмент, в отдельный класс. Напомним, что класс до-

бавляется в проект точно также, как форма, только в контекстном меню выбирается пункт Class.

В нашем варианте он будет содержать:

- поле для хранения имени файла с картинкой (string FileName);
- поля с текущими координатами x и y (float tek_x; float tek_y);
- поля с желаемыми координатами x и y (float ok_x; float ok_y);
- поле, определяющее выбран фрагмент или нет (bool selected);
- поля с шириной и высотой вывода картинки (float width; float height);
- поле, определяющее, насколько текущие координаты могут отличаться от желаемых (int luft).

Код этого класса представлен в листинге 1.58.

Листинг 1.58

```
class FragnLestn{ public string FileName; public bool selected=false;
public float tek_x; public float tek_y; public float ok_x; public float ok_y; public float
luft;
public float width; public float height;}
```

С каждым фрагментом нам нужно будет осуществлять следующие действия:

- заполнять начальными значениями поля;
- перемещать;
- проверять, достиг ли он своего места на экране;
- проверять, не выбрал ли его пользователь.

Все это тоже целесообразно реализовать в классе, добавив ему четыре метода: конструктор для заполнения начальных значений, метод для передвижения, метод для определения того, что фрагмент достиг своего места на экране, метод, определяющий, что пользова-

тель попал курсором в область фрагмента (выбрал его). Измененный код класса показан в листинге 1.59.

Теперь нам нужно для каждого из фрагментов определить на экране начальные и желаемые положения. Однако, так как координаты должны быть заданы относительно высоты и ширины pictureBox, то для их определения нам придется пойти на небольшую хитрость. Переходим в MouseClick у pictureBox и пишем в его обработчике строку кода: MessageBox.Show(e.X+” “+e.Y);

Благодаря этому при клике мышкой на pictureBox у нас будут выводиться в сообщение координаты курсора. Теперь запустим этот уровень и щелкнем по тем местам, где мы хотим разместить первоначально наши фрагменты (левый верхний угол каждого), записав данные на листок. Затем щелкнем в правый нижний угол экрана, чтобы выяснить текущие размеры pictureBox. В итоге у вас должны получиться следующие данные (таблица 1.4).

Листинг 1.59

```
class FragmLestn{ public string FileName; public bool selected=false;
public float tek_x; public float tek_y; public float ok_x;
public float ok_y; public float luft; public float width; public float height;

public FragmLestn(string fn, float tx, float ty, float okx, float oky, float w, float h, float
luft){
FileName = fn; tek_x = tx; tek_y = ty; ok_x = okx; ok_y = oky;
luft = luft; width = w; height = h;}

public void Move(int dx,int dy) { tek_x += dx; tek_y += dy; }

public bool InPlace(){
return (tek_x > ok_x - luft && tek_x < ok_x + luft & tek_y > ok_y - luft && tek_y < ok_y
+ luft); }

public bool TestKoord(float X,float Y){
selected = (X > tek_x & X < tek_x + width & Y > tek_y & Y < tek_y + height);
return selected;}
```

Таблица 1.4. Полученные данные

Фрагмент	Координата X левого верхнего угла	Координата Y верхнего левого угла
1	967	632
2	60	255
3	82	677
4	1009	296

При этом ширина pictureBox равна 1276, а высота – 1021.

Тогда для каждого фрагмента можно получить коэффициенты пропорциональности между координатами и размерами поля отрисовки. Приведем пример для координаты X. Расчет ведется по формуле

$$K_x = (\text{ширина_pictureBox} / \text{Координата_X}),$$

где Координата_X берется из таблицы 1.4.

Тогда относительная координата X, справедливая для любой ширины области отрисовки, будет рассчитана по формуле

$$\text{Относительный_X} = \text{ширина_pictureBox} / K_x.$$

С координатой Y – аналогично. Итоговый результат отражен в таблице 1.5.

Таблица 1.5. Относительные координаты

Фрагмент	Координата X левого верхнего угла	Координата Y верхнего левого угла
1	pictureBox1.Width/1.32	pictureBox1.Height/1.62
2	pictureBox1.Width / 21	pictureBox1.Height / 4
3	pictureBox1.Width / 16	pictureBox1.Height / 2
4	pictureBox1.Width / 1.3	pictureBox1.Height / 3

Обратите внимание!

- *Некоторые коэффициенты округлены для удобства работы и отображения.*

Теперь нам нужно определить желаемые размеры отрисовки картинок, а также их конечные положения на экране. Для простоты и удобства реализации определим это для первого фрагмента (также путем получения сообщений координат через клик), а для остальных зададим все относительно него. Возможные итоговые результаты отражены в таблице 1.6.

Наконец можем перейти к кодированию. Первым шагом объявим глобальные переменные, затем в методе `Form_Load` уравнием размеры `pictureBox` с формой, создадим объекты `Bitmap` и `Graphics`, отрисуем фон.

Таблица 1.6. Итоговые координаты

Фрагм.	Х левого верхнего угла	У верхнего левого угла	Ширина	Высота
1	<code>pictureBox1. Width / 3</code>	<code>pictureBox1. Height / 2.8</code>	<code>pictureBox1. Width / 6</code>	<code>pictureBox1. Height / 6</code>
2	Координата Х фрагмента1 плюс его ширина	Координата У фрагмента1	Совпадает с размерами фрагмента1	
3	Координата Х фрагмента1	Координата У фрагмента1 плюс его высота		
4	Координата Х фрагмента1 плюс его ширина	Координата У фрагмента1 плюс его высота		

После этого создадим объекты-фрагменты и отрисуем их. Далее отрисуем для пользователя квадраты-подсказки, куда нужно переместить фрагменты, и не забудем вывести отрисованный bitmap в pictureBox.Image. Реализующий описанное код представлен в листинге 1.60.

Листинг 1.60

```
Graphics gr; Bitmap bm; FragmLestn f1; FragmLestn f2;   FragmLestn f3;
FragmLestn f4;
private void Form1_Load(object sender, EventArgs e) {
pictureBox1.Width = this.Width;
pictureBox1.Height = this.Height;
bm = new Bitmap(pictureBox1.Width, pictureBox1.Height);
gr = Graphics.FromImage(bm);
gr.DrawImage(Image.FromFile("FonLestn.png"), 0, 0, pictureBox1.Width,
pictureBox1.Height);
f1 = new FragmLestn("Fragm1.png", (float)(pictureBox1.Width/1.32),
(float)(pictureBox1.Height/1.62), (float)(pictureBox1.Width /
3), (float)(pictureBox1.Height / 2.8), pictureBox1.Width / 6, pictureBox1.Height / 6,
10);
f2 = new FragmLestn("Fragm2.png", (float)(pictureBox1.Width / 21),
(float)(pictureBox1.Height / 4), f1.ok_x+f1.width, f1.ok_y, f1.width, f1.height, 10);
f3 = new FragmLestn("Fragm3.png", (float)(pictureBox1.Width / 16),
(float)(pictureBox1.Height / 2), f1.ok_x, f1.ok_y+f1.height, f1.width, f1.height, 10);
f4 = new FragmLestn("Fragm4.png", (float)(pictureBox1.Width / 1.3),
(float)(pictureBox1.Height / 3), f1.ok_x+f1.width, f1.ok_y + f1.height, f1.width,
f1.height, 10);
gr.DrawImage(Image.FromFile(f1.FileName), f1.tek_x, f1.tek_y, f1.width, f1.height);
gr.DrawImage(Image.FromFile(f2.FileName), f2.tek_x, f2.tek_y, f2.width, f2.height);
gr.DrawImage(Image.FromFile(f3.FileName), f3.tek_x, f3.tek_y, f3.width, f3.height);
gr.DrawImage(Image.FromFile(f4.FileName), f4.tek_x, f4.tek_y, f4.width, f4.height);
gr.DrawRectangle(Pens.Wheat, f1.ok_x, f1.ok_y, f1.width, f1.height);
gr.DrawRectangle(Pens.Wheat, f2.ok_x, f2.ok_y, f2.width, f2.height);
gr.DrawRectangle(Pens.Wheat, f3.ok_x, f3.ok_y, f3.width, f3.height);
gr.DrawRectangle(Pens.Wheat, f4.ok_x, f4.ok_y, f4.width, f4.height);
pictureBox1.Image = bm;
```

Если все сделано верно, то при запуске приложения должно быть нечто похожее на рисунок 1.54.

Теперь перейдем к реализации выделения фрагментов кликом. Визуально это будет отображаться обводкой выделенного красной рамочкой. Однако, так как выделенным может быть только один, то нам придется перерисовывать изображение, чтобы убирать рамочку у ранее выделенных фрагментов. Вновь возвращаемся в метод `MouseClicked`. Сейчас там написан код, выводящий сообщение. Он нам вроде бы не требуется, однако удалять его до финального завершения проекта не разумно: вдруг придется корректировать координаты. Чтобы его не удалять, но и чтобы он не участвовал в программе, мы просто ставим перед ним два слеша. Такие участки кода называются закомментированными. Среда их игнорирует при компиляции программы. Таким образом, код обработки клика примет вид, представленный в листинге 1.61.

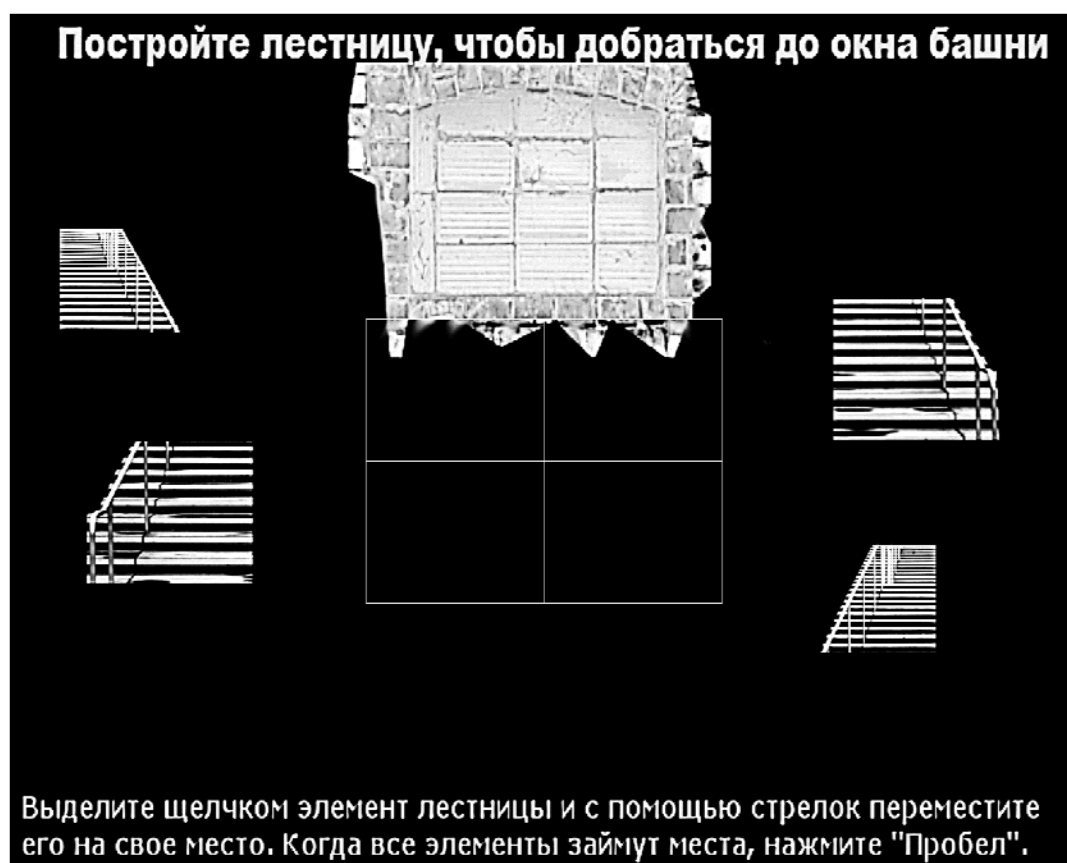


Рисунок 1.54. Начальный вид уровня

Листинг 1.61

```
private void pictureBox1_MouseClick(object sender, MouseEventArgs e){
// MessageBox.Show(e.X + " " + e.Y);

gr.DrawImage(Image.FromFile("FonLestn.png"), 0, 0, pictureBox1.Width,
pictureBox1.Height);
gr.DrawImage(Image.FromFile(f1.FileName), f1.tek_x, f1.tek_y, f1.width, f1.height);
gr.DrawImage(Image.FromFile(f2.FileName), f2.tek_x, f2.tek_y, f2.width, f2.height);
gr.DrawImage(Image.FromFile(f3.FileName), f3.tek_x, f3.tek_y, f3.width, f3.height);
gr.DrawImage(Image.FromFile(f4.FileName), f4.tek_x, f4.tek_y, f4.width, f4.height);

gr.DrawRectangle(Pens.Wheat, f1.ok_x, f1.ok_y, f1.width, f1.height);
gr.DrawRectangle(Pens.Wheat, f2.ok_x, f2.ok_y, f2.width, f2.height);
gr.DrawRectangle(Pens.Wheat, f3.ok_x, f3.ok_y, f3.width, f3.height);
gr.DrawRectangle(Pens.Wheat, f4.ok_x, f4.ok_y, f4.width, f4.height);

if (f1.TestKoord(e.X, e.Y)){
f2.selected = false; f3.selected = false;f4.selected = false;
gr.DrawRectangle(Pens.Red, f1.tek_x, f1.tek_y, f1.width, f1.height);}
else
if (f2.TestKoord(e.X, e.Y)){
f1.selected = false; f3.selected = false; f4.selected = false;
gr.DrawRectangle(Pens.Red, f2.tek_x, f2.tek_y, f2.width, f2.height);}

if (f3.TestKoord(e.X, e.Y)){
f2.selected = false; f1.selected = false; f4.selected = false;
gr.DrawRectangle(Pens.Red, f3.tek_x, f3.tek_y, f3.width, f3.height);}

if (f4.TestKoord(e.X, e.Y))
{
f2.selected = false;
f3.selected = false;
f1.selected = false;
gr.DrawRectangle(Pens.Red, f4.tek_x, f4.tek_y, f4.width, f4.height);
}

pictureBox1.Image = bm;}
```

Теперь, если вы запустите приложение и кликните на втором фрагменте, уровень должен выглядеть примерно следующим образом (рисунок 1.55):



Рисунок 1.55. Выделение фрагмента

Теперь займемся передвижением. Для этого вернемся в событие `PreviewKeyDown`. В нем определим две переменные для хранения текущего смещения (dx и dy). Затем проверим, если была нажата клавиша вверх, то смещению по y присвоим значение -5 . Если вниз, смещению по y присвоим 5 , если влево – смещению по x присвоим -5 , если вправо – смещению по x присвоим 5 . После этого отрисуем фон и квадраты, отмечающие желаемое положение вещей. Далее для выбранного фрагмента осуществим передвижение на указанные смещения, и, если фрагмент достиг нужного места, обведем его желтым, иначе – красным. Затем отрисуем все фрагменты в их текущих коор-

динатах. Код, реализующий указанный алгоритм, приведен в листинге 1.62.

Листинг 1.62

```
private void pictureBox1_PreviewKeyDown(object sender,
PreviewKeyDownEventArgs e)
{
    int dy = 0; int dx = 0;
    if (e.KeyCode == Keys.Escape) Application.Exit();
    if (e.KeyCode == Keys.Up) dy = -5;
    if (e.KeyCode == Keys.Down) dy = 5;
    if (e.KeyCode == Keys.Left) dx = -5;
    if (e.KeyCode == Keys.Right) dx = 5;
    gr.DrawImage(Image.FromFile("FonLestn.png"), 0, 0, pictureBox1.Width,
pictureBox1.Height);
    gr.DrawRectangle(Pens.Wheat, f1.ok_x, f1.ok_y, f1.width, f1.height);
    gr.DrawRectangle(Pens.Wheat, f2.ok_x, f2.ok_y, f2.width, f2.height);
    gr.DrawRectangle(Pens.Wheat, f3.ok_x, f3.ok_y, f3.width, f3.height);
    gr.DrawRectangle(Pens.Wheat, f4.ok_x, f4.ok_y, f4.width, f4.height);

    if (f1.selected)
    {
        f1.Move(dx, dy);
        if (f1.InPlace()) gr.DrawRectangle(Pens.Yellow, f1.tek_x - 1, f1.tek_y - 1,
f1.width + 1, f1.height + 1);
        else
            gr.DrawRectangle(Pens.Red, f1.tek_x - 1, f1.tek_y - 1, f1.width + 1,
f1.height + 1);
    }
    else
    if (f2.selected)
    { f2.Move(dx, dy);
    if (f2.InPlace()) gr.DrawRectangle(Pens.Yellow, f2.tek_x - 1, f2.tek_y - 1, f2.width +
1, f2.height + 1);
        else
            gr.DrawRectangle(Pens.Red, f2.tek_x - 1, f2.tek_y - 1, f2.width + 1, f2.height + 1);
    }
}
```

Окончание листинга 1.62

```
if (f3.selected) {
    f3.Move(dx, dy);
    if (f3.InPlace())
gr.DrawRectangle(Pens.Yellow, f3.tek_x - 1, f3.tek_y - 1, f3.width + 1, f3.height+1);
    else
gr.DrawRectangle(Pens.Red, f3.tek_x - 1, f3.tek_y - 1, f3.width + 1, f3.height+1); }

if (f4.selected) {
    f4.Move(dx, dy);
    if (f4.InPlace())
gr.DrawRectangle(Pens.Yellow, f4.tek_x - 1, f4.tek_y - 1, f4.width + 1, f4.height+1);
    else
gr.DrawRectangle(Pens.Red, f4.tek_x - 1, f4.tek_y - 1, f4.width + 1, f4.height+1);}

gr.DrawImage(Image.FromFile(f1.FileName), f1.tek_x, f1.tek_y, f1.width, f1.height);
gr.DrawImage(Image.FromFile(f2.FileName), f2.tek_x, f2.tek_y, f2.width, f2.height);
gr.DrawImage(Image.FromFile(f3.FileName), f3.tek_x, f3.tek_y, f3.width, f3.height);
gr.DrawImage(Image.FromFile(f4.FileName), f4.tek_x, f4.tek_y, f4.width, f4.height);

pictureBox1.Image = bm;}
```

Обратите внимание!

- *Так как обводка происходит раньше вывода фрагмента, то чтобы она была видна, ее нужно расширить на 1 пиксель.*

Если вы теперь запустите приложение и доведете все фрагменты до их конечных мест, то увидите следующую картину (рисунок 1.56).

Как мы видим, с программной точки зрения все фрагменты стоят на своих местах, однако картинки не совпали. Это произошло потому, что мы отслеживаем положение левого верхнего угла относительно идеала. Если бы наши фрагменты были бы квадратными, то все отработало бы правильно. Но так как размер картинки больше, чем изображение на ней (за счет прозрачного фона), наш алгоритм

работает не совсем корректно. Исправить его вам предлагается самостоятельно, как и реализовать обработку нажатия на пробел.

Заметьте, что передвижение объекта не очень удобное и плавное. Этот недостаток объясняется ограничениями рассматриваемой технологии и объясняет редкость ее применения для разработки игр.



Рисунок 1.56. Итоговый вид уровня

Оптимизация кода и сохранение статистики

Наш код написан крайне неоптимально: в нем много повторяющихся участков. При разработке хороших приложений это недопустимо: все повторяющиеся участки реализуются через методы и описываются только один раз, во всех же прочих местах они должны лишь вызываться.

В нашем коде можно выделить три метода: отрисовка фона, отрисовка фрагментов, передвижение выделенного фрагмента. Приве-

дем пример для последнего (листинг 1.63). Создание методов для отрисовки фона и фрагментов вам предлагается выполнить самостоятельно. Тогда оптимизированный код обработки нажатия на клавишу будет иметь вид, представленный в листинге 1.64. Как можно заметить, сокращение существенно.

Листинг 1.63

```
void Move(FragmLestn f, int dx,int dy){  
    if (f.selected){  
        f.Move(dx, dy);  
    }  
    if (f.InPlace())  
        gr.DrawRectangle(Pens.Yellow, f.tek_x - 1, f.tek_y - 1, f.width + 1, f.height + 1);  
    else  
        gr.DrawRectangle(Pens.Red, f.tek_x - 1, f.tek_y - 1, f.width + 1, f.height + 1);}}
```

Листинг 1.64

```
private void pictureBox1_PreviewKeyDown(object sender,  
PreviewKeyDownEventArgs e){  
    int dy = 0; int dx = 0;  
    if (e.KeyCode == Keys.Escape) Application.Exit();  
    if (e.KeyCode == Keys.Up) dy = -5;  
    if (e.KeyCode == Keys.Down) dy = 5;  
    if (e.KeyCode == Keys.Left) dx = -5;  
    if (e.KeyCode == Keys.Right) dx = 5;  
    DrawFon();  
    Move(f1, dx, dy);  
    Move(f2, dx, dy);  
    Move(f3, dx, dy);  
    Move(f4, dx, dy);  
    DrawFragm();  
    pictureBox1.Image = bm;}
```

Теперь рассмотрим такой момент, как сохранение статистики в файл и вывод ее на экран. Допустим, в переменной Program.Stat мы храним количество передвижений, которое игрок затратил на прохож-

дение этого уровня. Пусть в файле хранится лучший результат – самое меньшее число передвижений фрагментов. В левом нижнем углу экрана будет объект `label`, в который мы выведем этот результат при запуске уровня, если, конечно, этот результат есть. Реализовать описанное вам предлагается самостоятельно. Важные фрагменты кода, необходимые для решения поставленной задачи, представлены в листинге 1.65.

Листинг 1.65

```
//Для работы с файлами нам нужно подключить пространство имен System.IO,
//прописав:
using System.IO;
//При загрузке формы проверим, если файл с результатами, называемый stat.txt
//есть в //папке проекта, то считаем из него информацию и выведем в label. Код
//будет таким:
if (File.Exists("stat.txt")) label1.Text = File.ReadAllText("stat.txt");
//Для сохранения результатов необходимо использовать код:
File.WriteAllText("stat.txt", Program.Stat.ToString());
```

Если же мы хотим реализовать более сложный вариант хранения статистики, например, кроме передвижений еще и количество попыток открытия замка, и количество ударов, нанесенных дракону, то нам необходимо работать с разделителями. В C# у строк есть методы, которые позволяют разбивать их на части, выделяя элементы, заключенные между указанными символами. Например, из строки «Мама#мыла#раму» мы можем, указав разделителем решетку, выделить слова «мама», «мыла», «раму», сохранив каждое из них по отдельности, но для этого нам нужно освоить следующую тему, посвященную специальной структуре данных – массивам. Здесь же скажем, что выполняется это методом `Split`, вызванным через точку у переменной строкового типа.

Мы рассмотрели разработку простой многоуровневой игры, и теперь мы можем познакомиться с более сложными конструкциями, используемыми в программировании игры. Но прежде чем перейти

к новой теме, вам предлагается выполнить несколько заданий для закрепления полученных знаний.

ЗАДАНИЯ К ТЕМЕ 1.7

1. Доработайте изображения для игры и полностью ее реализуйте.
2. Самостоятельно реализуйте окно настроек и организуйте влияние указанных там параметров на игровой процесс.
3. Исправьте нестыковку картинок с лестницей.
4. Доработайте сценарий игры, предусмотрев заставки между уровнями.
5. Анимлируйте финальные заставки игры.
6. Оптимизируйте код игры.
7. Организуйте хранение статистики с выводом в label для каждого уровня в отдельном файле.
8. Усложнение игры: сделать код замка не трехзначным, а трехчисловым. При этом можно переходить на новый уровень, если отгаданы по очереди все три числа.
9. Усложнение игры: с каждым ударом по дракону возрастает скорость его перемещения.
10. Усложнение игры: реализовать перемещение фрагментов лестницы так, чтобы они не выходили за края экрана и не пересекались друг с другом при перемещении.

ТЕМА 1.8. ОДНОМЕРНЫЕ МАССИВЫ

Если мы внимательно посмотрим на код игры «Спасти принцессу», то увидим несколько фрагментов, в рамках которых мы выполняли одни и те же действия с объектами одного типа, однако для каждого используя отдельную переменную и, следовательно, работая с каждым отдельно. Для того чтобы над группой объектов можно бы-

ло выполнять схожие действия, их объединяют в коллекции – специальные структуры данных. Одной из таких структур является массив.

Массив – упорядоченное множество однотипных элементов. Тип может быть любым: и строки, и числа и объекты классов и прочее. В С# определены три различных категории массивов: одномерные, прямоугольные и вложенные. Второй вид представляет собой массив, элементами которого также являются массивы.

В С# массив является ссылочным типом, то есть в программе переменная-массив – ссылка на область памяти, где этот массив фактически хранится. Поэтому при работе с массивами обязательно использовать оператор `new` для выделения памяти при создании нового массива. Однако при создании массива мы всегда указываем его размер. То есть мы должны заранее знать, сколько элементов у нас будет. Примеры объявления ссылок на массив представлены в листинге 1.66, а код создания массивов – в листинге 1.67.

Листинг 1.66

```
int[] arr; // одномерный массив
int[,] arr1; // прямоугольный двумерный массив
int[][] arr2; // одномерный массив одномерных массивов
```

Листинг 1.67

```
int[] arr=new int[7];
int[,] arr1 = new int[2,2];
int[][] arr2= new int[2][];
arr2[0]=new int[2];
arr2[1]=new int[5];
```

На рисунке 1.57 наглядно представлены все объявленные массивы. Обратите внимание, что во вложенном массиве размеры элементов-массивов могут быть разными, но под каждый из них обязательно выделять память.

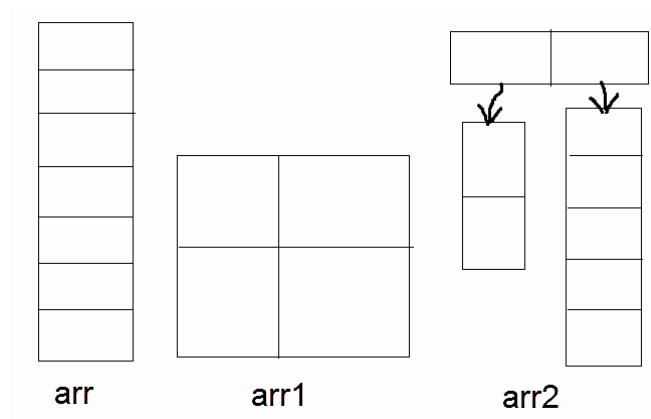


Рисунок 1.57. Графическое представление массивов

При создании одномерного или многомерного массива его можно сразу инициализировать (листинг 1.68).

Листинг 1.68

```
int[] arr = { 1, 3, 4, 5, 3, 5, 6, 4 }; int[,] arr1 = { {1, 3},{ 4, 5}, {3, 5},{ 6, 4 } };
```

Для обращения к элементам массива мы пишем его имя, а затем в квадратных скобках указываем нужный индекс. Приведем пример кода записи числа в первые элементы всех массивов (листинг 1.69).

Листинг 1.69

```
arr[0]=5; arr1[0,0]=5; arr2[0][0]=5;
```

На рисунке 1.58 наглядно показана разница между переменной и одномерным массивом.

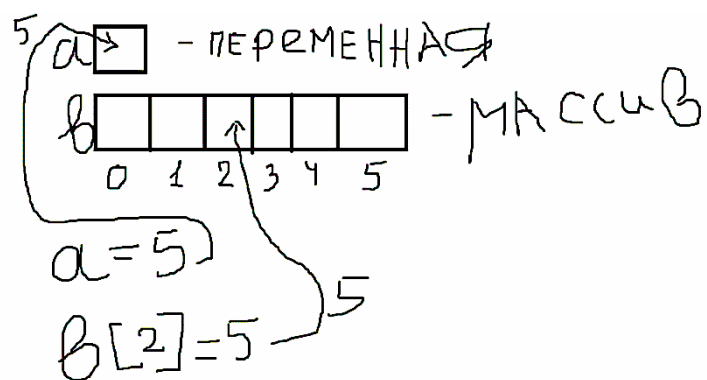


Рисунок 1.58. Отличие переменной от массива

Для массивов ключевым понятием является его длина. В одномерном массиве значение длины можно получить конструкцией: `имя_массива.Length`.

Обратите внимание!

- Для массива с рисунка 1.58 длина будет равна 6, тогда как номер последнего элемента – 5. Это происходит потому, что нумерация начинается с нуля.
- В многомерном массиве конструкция `имя_массива.Length` вернет количество всех элементов массива (то есть для массива 2 на 2 значение будет 4). Для того чтобы получить размер каждого отдельного измерения, нужно использовать конструкцию: `имя_массива.GetLength(номер_измерения)`;

Как мы сказали в самом начале, массивы позволяют организовывать выполнение одних и тех же действий с разными однотипными элементами. Для реализации этого при работе с массивами используются специальные операторы: циклы. В частности, пошаговый цикл `for`, который мы рассмотрим подробнее.

Цикл `for`

Этот оператор используется в задачах, где точно известно количество повторений выполняемых действий. Например, при обработке массивов: в параметрах цикла мы можем определить, какой диапазон элементов нам нужно обработать.

Цикл `for` имеет ряд параметров: `i` – переменная, значение которой будет меняться в цикле. `i<=10` – условие продолжения цикла (то есть пока `i` меньше или равна десяти продолжать). `i++` – оператор изменения значения переменной. (`++` – увеличение, `--` – уменьшение). Увеличение или уменьшение происходит на 1.

Таким образом, синтаксис цикла следующий:

for (<имя переменной>=<начальное значение переменной>;<условие продолжения цикла>;<выражение изменения значения переменной>) { }.

Обратите внимание!

- Действия, которые необходимо выполнить в цикле, записываются внутри фигурных скобок. В цикле *for* одна открывающая и одна закрывающая скобки.
- Пусть нам нужно в *label* вывести числа от 0 до 10. Мы можем написать так:
label1.Text="0 1 2 3 4 5 6 7 8 9 10";
или написать так:
for (int i=0;i<=10;i++) {label1.Text+= i.ToString()+" "; }.
На десяти цифрах разница, конечно, не очень заметна. А если чисел будет 1000?

Типовые алгоритмы работы с одномерными массивами

Рассмотрим несколько алгоритмов, которые вам могут понадобиться при использовании массивов в реализации игр.

- Объявление числового массива размера size (листинг 1.70)

Листинг 1.70

```
int[] mas=new int[size];
```

- Получение количества элементов массива (листинг 1.71)

Листинг 1.71

```
int size= mas.Length;
```

- Считывание одномерного массива с клавиатуры (листинг 1.72)

Мы предполагаем, что пользователь вводит элементы массива в textBox через пробел. Тогда делаем следующее:

- Объявляем строку и считываем в нее данные из textBox-а;
- Объявляем переменную-массив строк и записываем в нее результат работы метода Split, вызванного у переменной;
- Объявляем переменную-массив чисел с размером, равным размеру объявленного ранее строкового массива. Размер можно получить, поставив после имени массива точку и в выпавшем списке найдя Length. Его сохраним в дополнительную переменную size, которая нам в дальнейшем пригодится для различных циклов перебора (условие границы);
- Организуем цикл перебора элементов строкового массива, преобразования их в числа и записи в числовой массив.

Листинг 1.72

```
string s=textBox1.Text;
string[] st = s.Split(new char[] { ' ' }, StringSplitOptions.RemoveEmptyEntries);
//В одинарных кавычках стоит ПРОБЕЛ.
int size= st.Length;
int[] mas=new int[size];
for(int i=0;i<size;i++)
{
mas[i]=Convert.ToInt32(st[i]);
}
```

- Заполнение одномерного массива случайным образом (листинг 1.73)

Мы предполагаем, что пользователь вводит размер массива в textBox. Тогда делаем следующее:

- Объявляем числовую переменную (size) и считываем в нее данные из textBox-а, преобразовав их к числу;
- Объявляем массив длины size;

- Создаем генератор случайных чисел;
- Организуем цикл перебора элементов массива для записи в числовой массив случайного числа.

Листинг 1.73

```
int size=Convert.ToInt32(textBox1.Text);  
int[] mas=new int[size];  
Random r=new Random();  
for(int i=0;i<size;i++) { mas[i]=r.Next(-10,10); }
```

- Вывод одномерного массива на экран (листинг 1.74)

Для решения этой задачи организуем цикл перебора элементов массива и в его теле добавляем элементы в textBox1.Text.

Листинг 1.74

```
for (int i = 0; i < size; i++) textBox1.Text+=mas[i]+" ";
```

- Поиск максимума или минимума в одномерном массиве

Для реализации этого алгоритма на языке C# вам потребуются: заданный одномерный массив, цикл for и условный оператор if, а также дополнительная переменная. Данный алгоритм состоит из следующих шагов:

1. Объявить переменную с типом, совпадающим с типом элементов массива и присвоить ей значение первого подходящего элемента массива.
2. Организовать цикл перебора элементов массива.
3. В цикле сравнивать текущий элемент с объявленной переменной и, если он больше (для минимума) или меньше (для максимума), чем текущий, то сохранять текущий элемент в объявленную переменную.

Примечание: нельзя изначально инициализировать объявленную переменную нулем, так как в массиве могут быть только отрицательные элементы, тогда максимум посчитается некорректно. Или только положительные элементы, тогда минимум посчитается некор-

ректно. Кроме того, задача может заключаться в нахождении минимального четного, тогда нельзя инициализировать даже первым элементом массива, а необходимо искать первый четный элемент и инициализировать его уже им. Для того чтобы определить позицию минимального или максимального элемента, необходимо объявить дополнительную переменную и в нее сохранять индекс тогда, когда элемент с этим индексом записывается в первую объявленную переменную. Или же обойтись только одной переменной, хранящей позицию требуемого элемента.

- Поиск количества определенных элементов в одномерном массиве

Для реализации этого алгоритма на языке C# вам потребуются: заданный одномерный массив, цикл `for` и условный оператор `if`, а также дополнительная переменная.

1. Объявить числовую переменную, присвоить ей 0.
2. Организовать цикл перебора элементов массива.
3. В теле цикла через условный оператор проверять, подходит ли элемент критерию, если подходит, то увеличивать объявленную переменную на 1.

- Поиск суммы и среднего арифметического определенных элементов

Для реализации этого алгоритма на языке C# вам потребуются: заданный одномерный массив, цикл `for` и условный оператор `if`, а также дополнительная переменная.

Сумма:

1. Объявить переменную с типом, совпадающим с типом элементов массива, присвоить ей 0.
2. Организовать цикл перебора элементов массива.
3. В теле цикла через условный оператор проверять, подходит ли элемент критерию, если подходит, то прибавлять к объявленной переменной элемент массива, записывая результат в эту же переменную.

Среднее арифметическое:

По предыдущему алгоритму посчитать количество и разделить на него результат работы предыдущего алгоритма.

Примечание: для поиска суммы всех элементов массива из алгоритма исключается условие.

Применение массивов в игре «Спасти принцессу»

Рассмотрим, как мы можем оптимизировать код игры, используя массивы. В данном случае нам целесообразно осуществить в нем хранение фрагментов. Тогда вместо объявления четырех переменных типа `FragmLestn` мы объявим один массив (листинг 1.75).

Листинг 1.75

```
FragmLestn[] f=new FragmLestn[4];
```

В этом случае код создания фрагментов примет вид, показанный в листинге 1.76.

Листинг 1.76

```
f[0] = new FragmLestn("Fragm1.png", (float)(pictureBox1.Width/1.32),  
(float)(pictureBox1.Height/1.62), (float)(pictureBox1.Width /  
3), (float)(pictureBox1.Height / 2.8), pictureBox1.Width / 6, pictureBox1.Height / 6,  
10);  
FragmLestn f1 = f[0];  
f[1] = new FragmLestn("Fragm2.png", (float)(pictureBox1.Width / 21),  
(float)(pictureBox1.Height / 4), f1.ok_x+f1.width, f1.ok_y,f1.width, f1.height, 10);  
f[2] = new FragmLestn("Fragm3.png", (float)(pictureBox1.Width / 16),  
(float)(pictureBox1.Height / 2), f1.ok_x, f1.ok_y+f1.height, f1.width, f1.height, 10);  
f[3] = new FragmLestn("Fragm4.png", (float)(pictureBox1.Width / 1.3),  
(float)(pictureBox1.Height / 3), f1.ok_x+f1.width, f1.ok_y + f1.height, f1.width,  
f1.height, 10);
```

Обратите внимание!

- *Объявления переменной `f1` необязательно, но удобно, чтобы уменьшить изменение кода создания остальных фрагментов.*

Теперь обработка клика мыши примет вид, показанный в листинге 1.77. Отрисовку фона и фрагментов, а также оптимизацию метода PreviewKeyDown вам предлагается реализовать самостоятельно.

Листинг 1.77

```
private void pictureBox1_MouseClick(object sender, MouseEventArgs e) {
    DrawFon(); DrawFragm();
    // MessageBox.Show(e.X + " " + e.Y);
    for (int i = 0; i < f.Length; i++)
        if (f[i].TestKoord(e.X, e.Y)) {
            for (int j = 0; j < f.Length; j++)
                if (i != j) f[j].selected = false;
            gr.DrawRectangle(Pens.Red, f[i].tek_x, f[i].tek_y, f[i].width, f[i].height);
        }
    pictureBox1.Image = bm;}

```

Мы рассмотрели такие конструкции, как одномерные массивы и циклы. Теперь перейдем к рассмотрению двумерных массивов, однако перед этим вам рекомендуется выполнить несколько заданий по пройденному материалу.

ЗАДАНИЯ К ТЕМЕ 1.8

Здесь вам предлагается разработать дополнительный уровень к игре «Спасти принцессу», а также решить несколько неигровых задач, чтобы в полном объеме освоить работу с циклами и одномерными массивами. Разработку неигровых заданий рекомендуется вести в рамках одного приложения, для каждой задачи выделяя отдельную кнопку.

- Разработка дополнительного уровня к игре «Спасти принцессу»

Суть уровня: На экран выводится цепочка из нескольких зеленых кругов. По таймеру эти круги начинают менять свой цвет на красный. Задача игрока – «стрелять» курсором по красным кругам.

После 5 попаданий уменьшать размер кругов, после 10 попаданий увеличивать скорость мигания. Условие конца уровня – 100 попаданий.

Подсказка: для реализации вам понадобится класс Figure с полями x и y – координатами на экране, а также полем state – хранящим состояние фигуры. Далее, потребуется глобальная переменная: массив фигур случайного размера (от 5 до 10).

- Задачи на работу с циклом for:

1. Пользователь вводит число. Вывести на экран числа от 0 до этого числа.

2. Пользователь вводит число. Вывести на экран числа от этого числа до 0.

3. Пользователь вводит два числа. Если второе число больше первого, вывести на экран числа от первого до второго. Если первое число больше второго, то вывести надпись: «Данные не корректны».

4. Пользователь вводит два числа. Если второе число меньше первого, вывести на экран числа от первого до второго. Если первое число меньше второго, то вывести надпись: «Данные не корректны».

5. Пользователь вводит два числа. Вывести на экран числа от меньшего до большего.

6. Пользователь вводит два числа. Вывести на экран числа от большего до меньшего.

7. Пользователь вводит число. Вывести на экран таблицу умножения для этого числа. Например, пользователь ввел число 2, тогда на экран должно выводиться:

$$2 \times 1 = 2$$

....

$$2 \times 9 = 18$$

$$2 \times 10 = 20.$$

8. Пользователь вводит три числа. Вывести на экран числа от минимального до суммы двух других.

9. Пользователь вводит три числа. Вывести числа от максимального до суммы двух других.

10. Вывести на экран следующую табличку

1 2 3 4 5 6 7 8 9 10

2 4 6 8 10 12 14 16 18 20

....

10 20 30 40 50 60 70 80 90 100

Примечание: использовать циклы for, вложенные друг в друга. Формула вычисления каждого числа: номер строки умножить на номер столбика.

- Задачи на заполнение массивов:

1. Заполнить массив с клавиатуры.
2. Заполнить массив случайными числами.
3. Заполнить массив нулями, кроме первого и последнего элементов, которые должны быть равны единице.
4. Заполнить массив нулями и единицами, при этом данные значения чередуются, начиная с нуля.
5. Заполнить массив последовательными нечетными числами, начиная с единицы.
6. Сформировать возрастающий массив из четных чисел.
7. Сформировать убывающий массив из чисел, которые делятся на 3.
8. Заполнить массив заданной длины различными простыми числами. Натуральное число, большее единицы, называется простым, если оно делится только на себя и на единицу.
9. Создать массив, каждый элемент которого равен квадрату своего номера. Создать массив, на четных местах в котором стоят единицы, а на нечетных местах – числа, равные остатку от деления своего номера на 5.
10. Создать массив, состоящий из троек подряд идущих одинаковых элементов. Создать массив, который одинаково читается как слева направо, так и справа налево.

11. Сформировать массив из случайных чисел, в которых ровно две единицы, стоящие на случайных позициях.

12. Заполните массив случайным образом нулями и единицами так, чтобы количество единиц было больше количества нулей.

13. Сформировать массив из случайных целых чисел от 0 до 9, в котором единиц от 3 до 5 и двоек больше троек.

14. Создайте массив, в котором количество отрицательных чисел равно количеству положительных, и положительные числа расположены на случайных местах в массиве.

15. Заполните массив случайным образом нулями, единицами и двойками так, чтобы первая двойка в массиве встречалась раньше первой единицы, количество единиц было в точности равно суммарному количеству нулей и двоек.

• Задачи на анализ элементов массива. Для следующих задач массив заполняется случайным образом и выводится на экран:

1. Найти сумму элементов на четных местах и сумму элементов на нечетных местах. Определить, какая из них больше.

2. Найти минимум в первой половине массива и максимум во второй. Определить, кто из них больше.

3. Найти количество и сумму элементов, больших среднего арифметического в массиве.

4. Найти количество элементов массива, кратных его первому элементу (первый элемент при этом не рассматривать).

5. Найти наибольшее четное.

6. Найти разность между суммой положительных чисел и суммой отрицательных (по модулю). Модуль переменной `a`: `Math.Abs(a)`.

7. Найти количество элементов массива НЕ кратных его последнему элементу (последний элемент при этом не рассматривать).

8. Найти количество элементов массива, кратных разнице его первого и последнего элементов (элементы и разницу рассматривать по модулю).

9. Найти количество элементов массива, которые НЕ будут делителями для суммы его первого, второго и последнего элементов (элементы рассматривать по модулю).

10. Определить, что больше: сумма отрицательных элементов (по модулю), сумма положительных элементов или количество нулей.

11. Найти сумму положительных элементов на четных местах и сумму отрицательных элементов на нечетных местах. Определить, какая из них по модулю больше.

12. Найти количество четных элементов массива, которые будут делителями для суммы его первого и последнего элементов (элементы рассматривать по модулю).

13. Найти количество отрицательных элементов массива, кратных его первому элементу (первый элемент при этом не рассматривать).

14. Вывести максимальное положительное после каждого нечетного отрицательного.

15. Вывести 0 после элементов, которые НЕ будут делителями для суммы его первого, предпоследнего и последнего элементов (элементы рассматривать по модулю).

16. Вывести максимальный элемент после элементов массива, которые будут делителями для суммы его первого и последнего элементов (элементы рассматривать по модулю).

17. Вывести сумму минимального нечетного отрицательного и максимального четного положительного элементов после каждого элемента.

18. Вывести сумму минимального четного и максимального нечетного элементов после каждого отрицательного.

19. Вывести 0 после элементов, которые будут делителями для суммы его первого, второго и последнего элементов (элементы рассматривать по модулю).

20. Найти количество элементов массива, которые будут делителями для суммы его первого и последнего элементов (элементы рассматривать по модулю).

ТЕМА 1.9. ДВУМЕРНЫЕ МАССИВЫ

Одномерные массивы – хороший способ упорядочить ограниченный набор элементов, с которыми необходимо совершить схожие действия. Но в разработке игр более широкое применение нашли двумерные массивы: с их помощью реализуется множество приложений, таких как крестики-нолики, морской бой, тетрис, «лабиринтные бродилки», линии и прочее.

На рисунке 1.59 представлено изображение двумерного массива. Как мы видим, он действительно напоминает некое игровое поле.

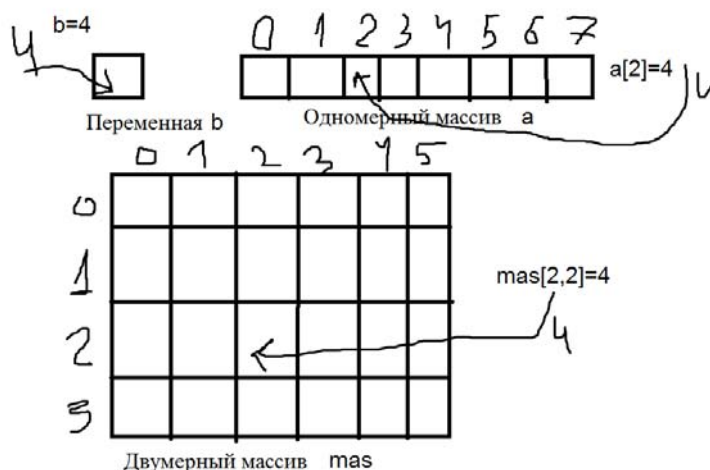


Рисунок 1.59. Двумерный массив

Типовые алгоритмы работы с двумерным массивом

- Объявление числового массива размера size1-строк и size2-столбцов (листинг 1.78)

Листинг 1.78

```
int[,] mas=new int[size1,size2];
```

- Считывание массива с клавиатуры

Мы предполагаем, что пользователь вводит элементы массива в textBox через пробел. Для перехода на новую строку используется

Enter. Обратите внимание, у textBox свойство Multiline надо установить в значение true (рисунок 1.60), а сам textBox растянуть по вертикали вниз, чтобы было место под несколько строчек ввода.

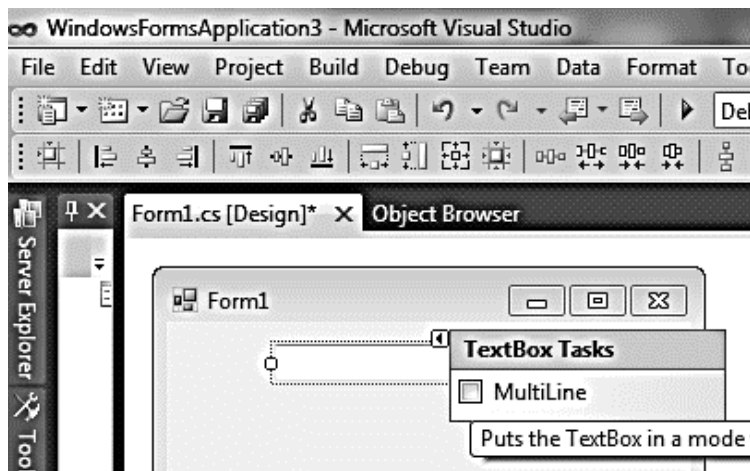


Рисунок 1.60. Многострочный textBox

Ввод пользователя должен выглядеть следующим образом (рисунок 1.61).

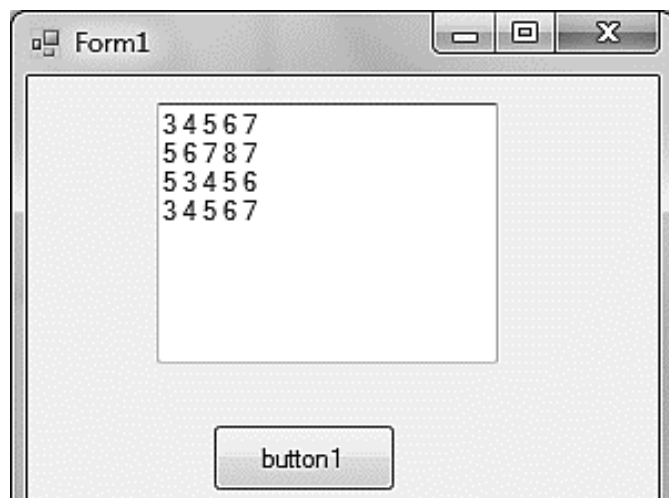


Рисунок 1.61. Введенный массив

Суть алгоритма проста: первым шагом мы методом Split разбиваем все содержимое textBox по Enter и получаем массив строк, в каждой из которой содержатся числа, разделенные пробелами. Следующим шагом мы для каждой строки выполняем разделение по пробелу, получив строковый чисел, затем пробегаемся по этому массиву, преобразуем каждый элемент в число и записываем в основной чис-

ловой массив. В графическом виде алгоритм можно представить следующим образом (рисунок 1.62).

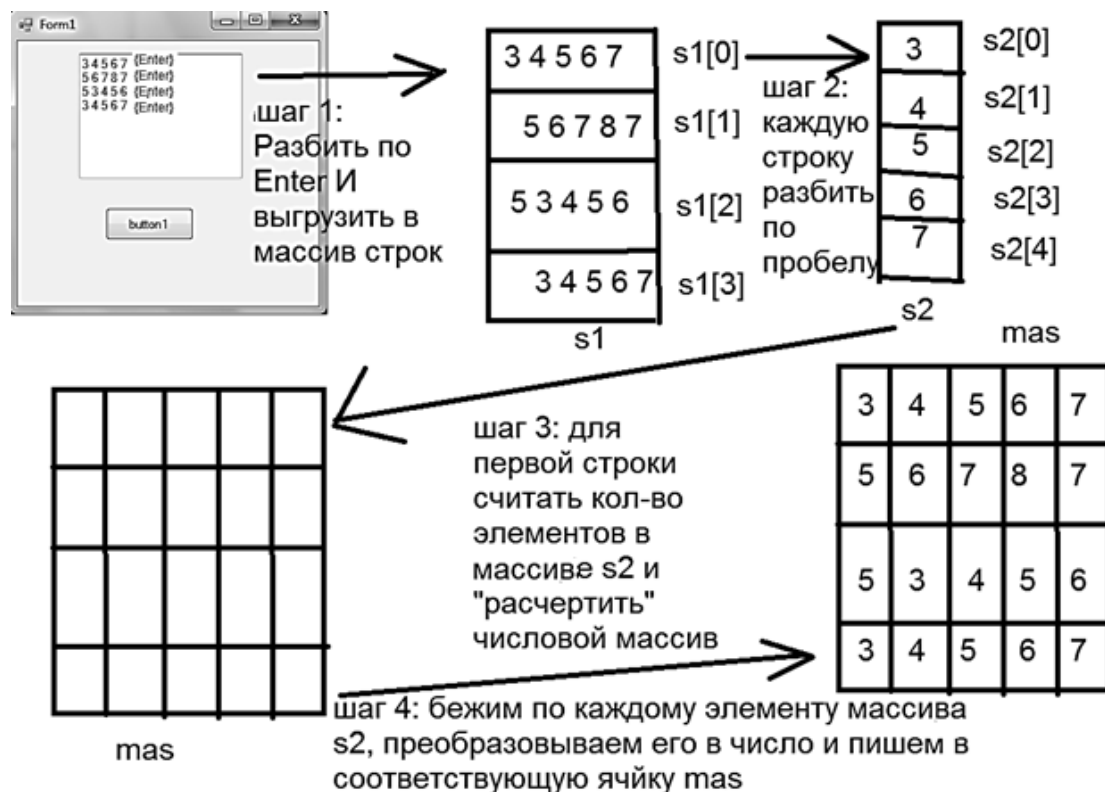


Рисунок 1.62. Преобразование ввода

Для программной реализации алгоритма в коде обработки кнопки делаем следующее:

- Объявляем строку и считываем в нее данные из textBox-а.
- Объявляем переменную-массив строк и записываем в нее результат работы метода Split, вызванного у переменной s, причем разделитель – Environment.NewLine.
- Объявляем переменную-массив чисел и пока присваиваем ей пустое значение.
- Объявляем переменные для числа строк и столбцов массива, причем число строк нам известно – это длина массива s1. Однако количество столбцов мы пока не знаем, поэтому ставим 0.
- Организуем цикл перебора элементов строкового массива, содержащего строки числового. Каждую строку разбиваем методом Split по разделителю «пробел». И если нам на обработку пришла пер-

вая строчка, то расчерчиваем наш числовой массив и организуем цикл перебора элементов конкретной его строки. В теле этого цикла записываем преобразованное в `int` строковое число. Итоговый код представлен в листинге 1.79.

Листинг 1.79

```
string s = textBox1.Text;
string[] s1 = s.Split(new string[] {Environment.NewLine},
StringSplitOptions.RemoveEmptyEntries);
int[,] mas=null;
int countStrok = s1.Length;
int countStolb=0;
for (int i = 0; i < countStrok; i++) {
string[] s2 = s1[i].Split(new char[] { ' ' }, StringSplitOptions.RemoveEmptyEntries);

if (i == 0) {countStolb = s2.Length;
mas = new int[countStrok, countStolb]; }

for (int j = 0; j < countStolb; j++)
mas[i, j] = Convert.ToInt32(s2[j]); }
```

- Вывод массива на экран (листинг 1.80)

Для решения этой задачи организуем циклы перебора элементов в строках и столбцах массива и добавляем элементы в `label1.Text`, переводя на новую строку курсор после каждой строки через `Environment.NewLine`.

Листинг 1.80

```
label1.Text="";
for (int i = 0; i < countStr; i++){
for (int j = 0; j < countStolb; j++) label1.Text+=mas[i,j]+" ";
label1.Text+=Environment.NewLine;}
```

- Заполнение массива случайным образом (листинг 1.81)

Мы предполагаем, что пользователь вводит размер массива в 2 элемента textBox. Тогда делаем следующее:

- Объявляем числовые переменные (countStr и countStolb) и считываем в них данные из элементов textBox, преобразовав их к числу.
- Объявляем массив с количеством строк countStr и количеством столбцов countStolb.
- Создаем генератор случайных чисел.
- Организуем циклы перебора элементов в строках и столбцах массива для записи в числовой массив случайного числа.

Листинг 1.81

```
int countStr =Convert.ToInt32(textBox1.Text);
int countStolb =Convert.ToInt32(textBox2.Text);
int[,] mas=new int[countStr, countStolb];
Random r=new Random();
for (int i = 0; i < countStr; i++)
    for (int j = 0; j < countStolb; j++) {
        mas[i,j]=r.Next(-10,10);    }
```

Если мы внимательно посмотрим на последний алгоритм и сравним его со схожим алгоритмом для одномерного массива, то увидим, что они различаются лишь количеством циклов и индексов. Тоже самое можно сказать и прочих, рассмотренных ранее алгоритмах (поиск суммы, поиск количества и т. д.). Поэтому их словесное описание вам предлагается составить самостоятельно.

- Получение количества строк или столбцов массива (листинг 1.82)

Листинг 1.82

```
int countStr =mas.GetLength(0); // строки
int countStolb =mas.GetLength(1); // столбцы
```

- Работа с отдельными строками и столбцами

Очень часто в двумерных массивах необходимо произвести действия с какими-либо отдельными его частями. Например, со строкой или столбцом. В этом случае алгоритм прост: зафиксировать координату требуемого столбца/строки, а по второй – организовать цикл. Приведем пример кода, вычисляющего сумму элементов первой строки (листинг 1.83). При этом предполагается, что массив `mas` – объявлен и заполнен.

Листинг 1.83

```
int sum=0;
for(int j=0;j<mas.GetLength(1);j++) sum+=mas[0,j];
```

- Работа с диагоналями

Двумерные массивы, у которых количество строк и столбцов совпадает, называют квадратными матрицами и выделяют у них диагонали (рисунок 1.63). Эти цепочки элементов используются в различных алгоритмах. Стрелка 1 на рисунке отмечает главную диагональ, стрелка 2 – побочную.

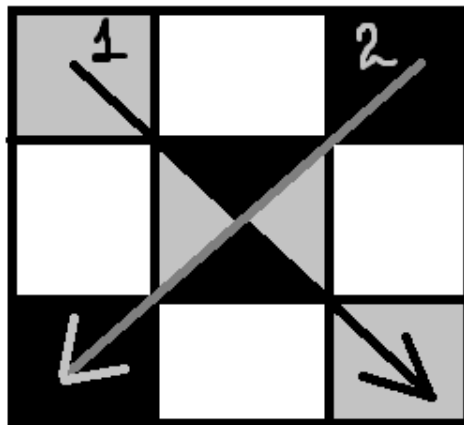


Рисунок 1.63. Диагонали матрицы

Для диагоналей существуют правила:

1. Для элемента на главной диагонали номер строки равен номеру столбца.

2. Для элемента на побочной диагонали номер строки вычисляется по формуле: количество столбцов минус номер столбца и минус 1.

3. Для элементов под главной диагональю номер строки больше номера столбца.

4. Для элементов над главной диагональю номер строки меньше номера столбца.

5. Для элементов над побочной диагональю сумма номеров строки и столбца меньше количества строк (столбцов).

6. Для элементов под побочной диагональю сумма номеров строки и столбца больше количества строк (столбцов).

Приведем пример кода, заполняющего главную диагональ нулями (листинг 1.84).

Листинг 1.84

```
for(int i=0;i<mas.GetLength(0);i++) mas[i,i]=0;
```

Мы рассмотрели основы работы с двумерными массивами и теперь можем перейти к их использованию для реализации игр. Однако перед этим вам рекомендуется выполнить несколько практических заданий для закрепления полученных знаний.

ЗАДАНИЯ К ТЕМЕ 1.9

В заданиях к данной теме вам предлагается решить несколько неигровых задач, чтобы в полном объеме освоить работу с двумерными массивами. Разработку неигровых заданий вести в рамках одного приложения, для каждой задачи выделяя отдельную кнопку. Во всех задачах, кроме первой, пользователь вводит размер массива.

Задачи:

1. Заполнить массив с клавиатуры.
2. Заполнить массив случайными числами.

3. Заполнить каждый четный столбец массива нулями, а каждый нечетный единицами:

0 1 0 1

0 1 0 1

0 1 0 1

4. Заполнить массив нулями, кроме первого и последнего элемента, которые должны быть равны 1:

1 0 0 0

0 0 0 0

0 0 0 1

5. Заполнить массив последовательными нечетными числами, начиная с единицы:

1 3 5 7

9 11 13 15

17 19 21 23

6. Сформировать возрастающий массив из четных чисел:

2 4 6 8

10 12 14 16

18 20 22 24

7. Сформировать убывающий массив из чисел, которые делятся на 3:

36 33 30 27

24 21 18 15

12 9 6 3

8. Создать массив, каждый элемент которого равен произведению номера строки и номера столбца

9. Создать массив, в четных строках которого стоят единицы, а в нечетных – числа, равные остатку от деления суммы своих индексов на 5.

10. Создать массив, каждая строка которого одинаково читается как слева направо, так и справа налево. Строки не должны повторяться.

11. Сформировать массив из случайных чисел, в которых ровно две единицы, стоящие на случайных позициях.
12. Заполните массив случайным образом нулями и единицами так, чтобы количество единиц было больше количества нулей.
13. Определить, содержит ли массив заданное пользователем число.
14. Найти количество четных чисел в массиве.
15. Найти количество чисел в массиве, которые делятся на 3, но не делятся на 7.
16. Определите, каких чисел в массиве больше: которые делятся на первый элемент массива или которые делятся на последний элемент массива.
17. Найдите сумму и произведение элементов массива.
18. Найдите сумму четных чисел массива.
19. Найдите сумму нечетных чисел массива, которые не превосходят 11.
20. Найдите сумму чисел массива, которые расположены до первого четного числа массива.
21. Найти сумму всех чисел за исключением крайних.
22. Найдите сумму чисел массива, которые стоят в четных строках.
23. Найдите сумму чисел массива, которые стоят в нечетных столбцах и при этом превосходят сумму крайних элементов массива.
24. Найти наибольший элемент массива.
25. Найдите сумму наибольшего и наименьшего элементов массива.
26. Найдите количество элементов массива, которые отличны от наибольшего элемента не более чем на 10%.
27. Найдите наименьший четный элемент массива.
28. Среди элементов с нечетными номерами столбцов найдите наибольший элемент массива, который делится на 3.
29. Найти количество отрицательных элементов.

30. Генерируется случайным образом квадратная матрица. Поменять местами первый и последний столбец матрицы.

31. Генерируется случайным образом квадратная матрица. Определить количество положительных, отрицательных и нулевых элементов.

32. Генерируется случайным образом квадратная матрица. Определить количество четных элементов на главной диагонали.

33. Генерируется случайным образом квадратная матрица. Найти количество нечетных элементов на побочной диагонали.

34. Генерируется случайным образом квадратная матрица. Найти количество положительных элементов под главной диагональю.

35. Генерируется случайным образом квадратная матрица. Определить количество нулей среди элементов, которые не находятся на главной диагонали матрицы.

36. Генерируется случайным образом квадратная матрица. Определить сумму элементов над главной диагональю матрицы.

37. Генерируется случайным образом квадратная матрица. Определить произведение элементов побочной диагонали.

38. Генерируется случайным образом квадратная матрица. Определить сумму неотрицательных элементов под побочной диагональю.

39. Генерируется случайным образом квадратная матрица. Найти сумму положительных элементов над главной диагональю матрицы.

40. Генерируется случайным образом квадратная матрица. Найти сумму элементов первого столбца и количество элементов над главной диагональю, чье значение больше, чем найденная сумма.

ТЕМА 1.10. РАЗРАБОТКА ИГРЫ «МОРСКОЙ БОЙ»

Рассмотрим использование двумерных массивов при реализации игры «Морской бой». Приведенный в данном разделе код не оптимизирован, а кроме того, логика игры не отделена от ее интерфейса. Эти недостатки вам предлагается исправить самостоятельно после изучения темы.

Начало разработки

Первое, что нам нужно сделать, – определиться с необходимыми структурами данных, а также интерфейсными элементами, продумав логику игры.

Как мы помним, морской бой – игра, где на поле размером 10×10 расставляются «корабли», и задача игрока – уничтожить «флот» противника, раньше, чем уничтожат его корабли. Как правило, перед глазами у игрока два поля – свое и противника. На первом отмечаются его промахи и попадания, на втором – свои. «Флот» составляют: один четырехпалубник, два трехпалубника, три двухпалубника и четыре однопалубника. Количество палуб соответствует количеству клеток, занимаемых кораблем. При их расстановке необходимо учитывать, что по всем направлениям (и диагональным тоже) между кораблями должна быть как минимум одна свободная клетка. Игровое поле пронумеровано: клетки по горизонтали – цифрами от 1 до 10, по вертикали – буквами от А до К (Е из расчета исключается). Игроки ходят по очереди, называя номер и букву клетки. Если клетка пуста, ход переходит к другому игроку, а стрелявший отмечает у себя промах. Если же клетка занята кораблем – засчитывается попадание, и игрок получает право дополнительного хода.

Таким образом, мы видим, что из структур данных нам потребуется два массива 10 на 10 , а из интерфейсных элементов – два `pictureBox`-а и кнопка начала игры. Причем, так как мы предполагаем, что пользователь будет использовать клик мыши для выбора клетки,

то нумерацию полей на начальном этапе можно не отрисовывать. Следовательно, в массивах нам нужно хранить данные о состоянии полей. Для этого целесообразно назначить им целочисленный тип и ввести следующие правила кодирования: 0 – пустая клетка, 1 – корабль, -1 – промах, 2 – попадание.

После того как мы определились с начальными элементами – размещаем их на форме (рисунок 1.64).

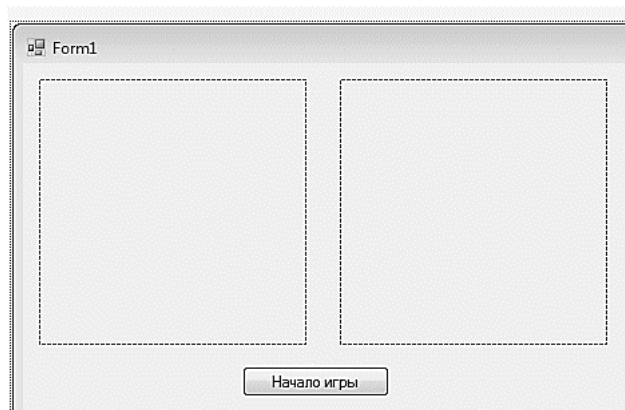


Рисунок 1.64. Начальный вид приложения

Обратите внимание!

- Элементы *pictureBox* квадратные, и их размер кратен 10 (в примере – 200 на 200). Это сделано для упрощения отрисовки.

Теперь нам необходимо объявить структуры данных и «расчертить» поля. Кроме того, нам нужно ввести именованные обозначения для наших элементов кодировки. Поясним последнюю фразу. В зависимости от состояния клетки (пустая, с промахом и т. д.) нам необходимо будет по-разному ее отрисовывать, поэтому в коде неизбежны сравнения со значениями нашей кодировки. Чтобы не приходилось постоянно запоминать, какая цифра что обозначает, разумнее будет присвоить значения осмысленно именованным переменным, и сравнивать уже с ними.

Примечание: если мы используем переменные для хранения неизменяемых смысловых значения, то они превращаются в константы

и объявляются со специальным модификатором `const`. Данный прием улучшает качество вашего кода.

В итоге требуемые нам структуры данных объявляются следующим кодом (листинг 1.85).

Листинг 1.85

```
int[,] gamer = new int[10, 10]; int[,] computer = new int[10, 10];  
const int empty = 0; const int miss = -1; const int ship = 1; const int hit = 2;
```

Примечание: если вы не знаете английского языка, то можете использовать транслитные имена (`pusto`, `igrok`, `korabl` и т.д.).

Теперь займемся кодом отрисовки наших полей. Нам нужно на обоих `pictureBox` отрисовать 10 горизонтальных и 10 вертикальных линий. Так как размеры элементов кратны 10, то формулу для точки, через которую проходит линия, можно записать так: `номер_линии*(сторона_pictureBox/10)`. Если у вас возникли затруднения с пониманием формулы, посмотрите на рисунок 1.65, где приведена иллюстрация к расчету.



Рисунок 1.65. Расчерченное поля

Опираясь на данный расчет, осуществим отрисовку одного из полей при загрузке формы. Код, реализующий это, будет следующим (листинг 1.86).

Листинг 1.86

```
private void Form1_Load(object sender, EventArgs e){
    Bitmap bm = new Bitmap(pictureBox2.Width, pictureBox2.Height);
    Graphics gr = Graphics.FromImage(bm);
    for (int i = 0; i < 10; i++){
        gr.DrawLine(Pens.Black, i * (pictureBox2.Width / 10), 0, i * (pictureBox2.Width / 10),
        pictureBox2.Height);
        gr.DrawLine(Pens.Black, 0, i * (pictureBox2.Height / 10), pictureBox2.Width, i *
        (pictureBox2.Height / 10));}
    pictureBox2.Image = bm;}
```

Теперь займемся отрисовкой состояния поля. Пусть палубы кораблей отрисовываются красными квадратами во всю клетку, промахи – черными кружками по центру клетки, попадания – крестиками во всю клетку.

Если мы представим ход игры, то поймем, что отрисовка состояния поля будет происходить довольно часто. Причем для обоих полей она будет практически одинакова. Единственная разница – корабли противника в процессе игры не должны быть отображены. Однако для отладки работы алгоритма нам потребуется видеть состояние обоих полей. Поэтому целесообразно отрисовку поля сразу вынести в отдельную функцию. Ее параметрами будут: массив, который нужно отрисовать, объект-Graphics, настроенный на определенный Bitmap, и логическая переменная, определяющая, надо ли отрисовывать расстановку кораблей.

Как мы помним, эллипсы в GDI+ отрисовываются от верхнего левого их угла. Поэтому, если нам нужно нарисовать эллипс в клетке (i,j), то, опираясь на рисунок 1.65, мы выведем следующую формулу для его координат:

номер_клетки*(сторона_pictureBox/10)+сторона_pictureBox/40.

Ширина же его будет равна сторона_pictureBox/10-10. Таким образом, мы ожидаем получить небольшой круг в центре клетки. Для квадрата-палубы координаты будут такими же, а ширина – на 2 пикселя меньше клетки. Координаты для крестика-попадания вам предлагается определить самостоятельно и вставить их вместо знаков вопроса в следующий код, реализующий отрисовку, представленный в листинге 1.87.

Листинг 1.87

```
void vivod(int[,] mas,Graphics gr,bool f){
for (int i = 0; i < 10; i++){
    gr.DrawLine(Pens.Black, i * (pictureBox1.Width / 10), 0, i * (pictureBox1.Width / 10), pictureBox1.Height);
    gr.DrawLine(Pens.Black, 0, i * (pictureBox1.Height / 10), pictureBox1.Width, i * (pictureBox1.Height / 10));}
for (int i = 0; i < 10; i++) {
for (int j = 0; j < 10; j++) {
    if (f) if (mas[i, j] == ship) gr.FillRectangle(Brushes.Red, j * (pictureBox1.Width / 10), i * (pictureBox1.Height / 10), (pictureBox1.Width / 10) - 2, (pictureBox1.Height / 10) - 2);
    if (mas[i, j] == hit) {
gr.DrawLine(Pens.Black, ?? * (pictureBox1.Width / 10), ?? * (pictureBox1.Height / 10), (pictureBox1.Width / 10) * ??, (pictureBox1.Height / 10) * ??);
gr.DrawLine(Pens.Black, ?? * (pictureBox1.Width / 10), ?? * (pictureBox1.Height / 10), (pictureBox1.Width / 10) * ??, (pictureBox1.Height / 10) * ??);}
    if (mas[i, j] == miss) gr.FillEllipse(Brushes.Black, j * (pictureBox1.Width / 10)+ (pictureBox1.Width / 40), i * (pictureBox1.Height / 10)+ (pictureBox1.Height / 40), (pictureBox1.Width / 10) - 10, (pictureBox1.Height / 10) - 10); }}}}
```

Для проверки правильности работы нашего алгоритма в кнопке «Начало игры» заполним три клетки массива игрока разными значениями и отрисуем поле. Код обработки нажатия на кнопку представлен в листинге 1.88.

Листинг 1.88

```
private void button1_Click(object sender, EventArgs e) {  
    Bitmap bm = new Bitmap(pictureBox1.Width, pictureBox1.Height);  
    Graphics gr = Graphics.FromImage(bm); gamer[0, 0] = hit; gamer[0, 5] =  
    ship; gamer[4, 3] = miss; vivod(gamer, gr, true); pictureBox1.Image = bm; }
```

Обратите внимание!

- *В массиве мы указываем координаты в порядке i, j , тогда как в отрисовке их порядок обратный: j, i . Это происходит из-за того, что x экрана соответствует столбцам массива, а y – строкам. Будьте внимательны, иначе ваш массив будет отрисовываться в транспонированном виде, и это может привести к ошибкам в алгоритме. Для контроля вы можете разместить на форме `label` и дополнительно выводить массивы в обычном виде, как вы это делали при решении задач. Более того, при начальной разработке игр это настоятельно рекомендуется делать.*

Если все сделано верно, то должно получиться нечто похожее на рисунок 1.66.

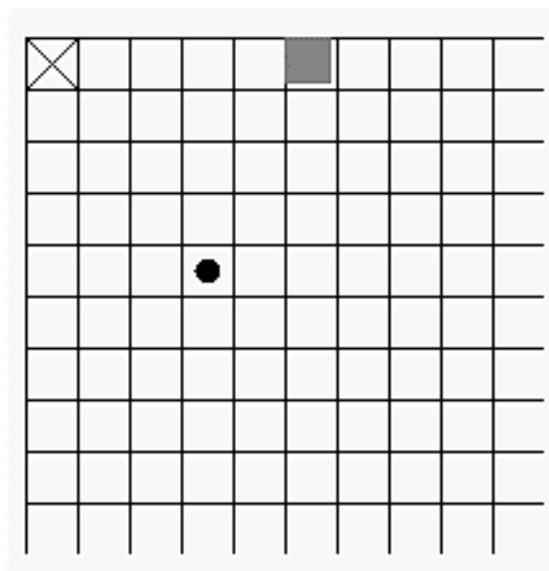


Рисунок 1.66. Пробная отрисовка поля

Как мы видим, палубы корабля отрисовываются не совсем красиво. Этот недостаток вам предлагается исправить самостоятельно, как и сделать тестовое отображение для поля компьютера.

После того как мы убедились, что наша отрисовка работает корректно, можно перейти к реализации логики работы игры, начав с расстановки кораблей компьютером.

Расстановка кораблей компьютером

Нам необходимо реализовать автоматическое заполнение массива-поля четырехпалубником, двумя трехпалубниками, тремя двухпалубниками и четырьмя однопалубниками, причем так, чтобы между кораблями была как минимум одна свободная клетка по всем направлениям.

В первом приближении алгоритм может быть таким:

1. Взять случайную клетку, если она пустая и вокруг нее тоже пусто, то зарезервировать ее для корабля.
2. Выбрать направление и проверить нужное количество клеток в выбранном направлении.
3. Если все клетки направления нас устраивают – поставить корабль.

Если мы собираемся реализовать этот алгоритм, то первое, что нам необходимо разработать – функцию, проверяющую, подходит ли нам клетка. Условие в данном случае: в радиусе одной клетки вокруг нет кораблей, и клетка находится в границах массива. Параметрами функции будут: массив, в котором происходит проверка, а также координаты клетки, которую нужно проверить. Реализующий это код представлен в листинге 1.89.

Листинг 1.89

```
bool func(int x, int y, int[,] mas){
for (int i = -1; i <= 1; i++)
for (int j = -1; j <= 1; j++)
if (x + i < 10 && y + j < 10 && x + i > -1 && y + j > -1){
if (mas[x + i, y + j] == 1) return false;}
else if (x >= 10 || y >= 10 || x < 0 || y < 0) return false;
return true;}
```

Теперь реализуем функцию постановки корабля. Основными его характеристиками будут количество палуб, начальная точка (с которой начинается постановка) и направление, куда разместится корабль относительно этой точки (вниз, влево, вправо, вверх), кроме того, необходимо указать, в какой массив мы размещаем корабль. Таким образом, функция будет иметь 5 параметров.

Внутри функции нам целесообразно использовать дополнительный двумерный массив, в котором количество строк равно количеству палуб, а количество столбцов равно 2. В этом массиве мы будем запоминать координаты каждой палубы и при удачном его заполнении (если все палубы окажутся подходящими) переносить данные в основной массив. Алгоритм работы функции будет следующим:

1. Объявить временный массив и заполнить его -1.
2. Выбрать первое направление: вправо. Организовать цикл с нуля до количества палуб. В цикле проверять каждую клетку массива на возможность установки в нее палубы. При этом счетчик цикла прибавляется к номеру столбца массива.
3. Если клетка подходит, продолжаем движение в выбранном направлении, иначе – меняем направление.
4. Если все клетки нашего корабля подходят, переносим данные в основной массив и возвращаем положительный результат, говорящий о том, что корабль поставлен. Иначе – возвращаем отрицательный результат.

Данный алгоритм реализуется следующим кодом (листинг 1.90).

Листинг 1.90

```
bool func2(int x, int y, int z, int[,] mas, int n){ int[,] temp = new int[z, 2];
    for (int i = 0; i < z; i++) for (int j = 0; j < 2; j++) temp[i,j] = -1;
if (n == 0)for (int t = 0; t < z; t++)
if (func(x, y + t, mas)) { temp[t, 0] = x; temp[t, 1] = y + t;}
if (n==1)for (int t = 0; t < z; t++)
if (func(x, y - t, mas)) { temp[t, 0] = x; temp[t, 1] = y - t;}
if (n == 2)for (int t = 0; t < z; t++)
if (func(x+t, y, mas)) { temp[t, 0] = x+t; temp[t, 1] = y; }
if (n == 3) for (int t = 0; t < z; t++)
if (func(x - t, y, mas)) { temp[t, 0] = x - t; temp[t, 1] = y;}
int k = 0;
    for (int i = 0; i < z; i++)
        for (int j = 0; j < 2; j++)
            if (temp[i, j] == -1) k++;
if (k == 0){
    for (int i = 0; i < z; i++)
        mas[temp[i, 0], temp[i, 1]] = 1;}
return k==0;}
```

Как можно заметить, в данном коде вместо именованных констант, указывающих направление, используются числа, что затрудняет его понимание. Данный недостаток вам предлагается исправить самостоятельно. Кроме того, рекомендуется произвести оптимизацию кода с целью уменьшения его объема. Разработайте вариант определения направления таким образом, чтобы для проверки направления можно было использовать только один цикл.

После того как мы реализовали вспомогательные проверочные функции, можно реализовать полноценный код расстановки кораблей компьютером. Алгоритм будет следующим:

1. Начинаем с четырехпалубника.
2. Выбираем случайную точку и направление. Пробуем поставить корабль.
3. Если удачно – переходим к следующему кораблю, если нет – повторяем шаг два.

Если мы внимательно посмотрим на взаимосвязь количества палуб и количество кораблей данного типа на поле, то выведем следующую формулу: количество_кораблей=5-количество_палуб. Используя эту взаимосвязь, мы сможем описать расстановку кораблей, используя два цикла: первый от 4 до 1 – по количеству палуб, второй – от 5-количество_палуб до 1. Таким образом, мы охватим весь пул кораблей. Алгоритм же приобретет следующий вид:

1. Организуем цикл по палубам: с 4 до 1.
2. Организуем цикл с 5-количество_палуб до 1.
3. Выбираем случайную точку и направление. Пробуем поставить корабль.
4. Если удачно – переходим к следующему кораблю, если нет – повторяем шаг три.
5. Если корабли данного типа расставлены, уменьшаем количество палуб (оператор цикла это сделает за нас) и переходим к шагу 2.

Остановимся подробнее на шагах 2-4. С одной стороны, они описывают обычный цикл, в котором переменная меняется в фиксированном диапазоне. Однако изменение данной переменной может происходить не на каждом шаге. Например, если мы трижды попытаемся неудачно поставить корабль, то с 2 до 1 наша переменная изменится не за один шаг, а за пять. В данном случае цикл `for` использовать нельзя, потому что этот оператор применяется только в случаях, когда переменная проходит точное число шагов одинакового размера. Для нашей же задачи подойдет один из двух других операторов организации цикла: `while` или `do-while`.

Первый – цикл с предусловием, как и `for` (то есть перед выполнением операторов цикла проверяется условия выхода). Однако условие выхода программист определяет сам. В общем случае это может быть любое логическое выражение, никак не связанное с изменением переменной. Оператор имеет следующий синтаксис:

```
while (УсловиеПродолжения) {Операторы; }
```

Пример: допустим, нам нужно повторять генерацию случайного числа, пока не выпадет значение большее 10. Код, реализующий это, будет следующим (листинг 1.91).

Листинг 1.91

```
int i=0; Random r=new Random(); while(i<=10) i=r.Next(0,20);
```

Как мы видим, работа оператора организуется по следующему правилу: сначала проверяется условие продолжения оператора и в случае, если значение условного выражения равно true, соответствующий оператор (блок операторов) выполняется.

Схожим с while является оператор do ... while. Это цикл с пост-условием. Синтаксис:

do {Операторы;} while (УсловиеПродолжения)

Реализация того же примера с циклом do-while представлена в листинге 1.92.

Листинг 1.92

```
int i=0; Random r=new Random(); do i=r.Next(0,20); while(i<=10);
```

Разница с ранее рассмотренным оператором цикла состоит в том, что здесь сначала выполняется оператор (блок операторов), а затем проверяется условие продолжения. С более практической точки зрения: цикл do-while используется, если блок операторов первый раз в любом случае должен выполняться.

В нашем случае один раз код постановки корабля точно должен выполняться, поэтому мы используем цикл do-while. Хотя и while здесь подошел бы с тем же успехом. Тем не менее финальный код расстановки кораблей компьютером имеет вид, представленный в листинге 1.93. Результат работы кода из листинга 1.93 представлен на рисунке 1.67.

Листинг 1.93

```
for (int i = 4; i > 0; i--){int k = 5 - i;  
do{ if (func2(r.Next(0, 10), r.Next(0, 10), i, computer, r.Next(0, 4))) k--;}  
while (k > 0);}
```

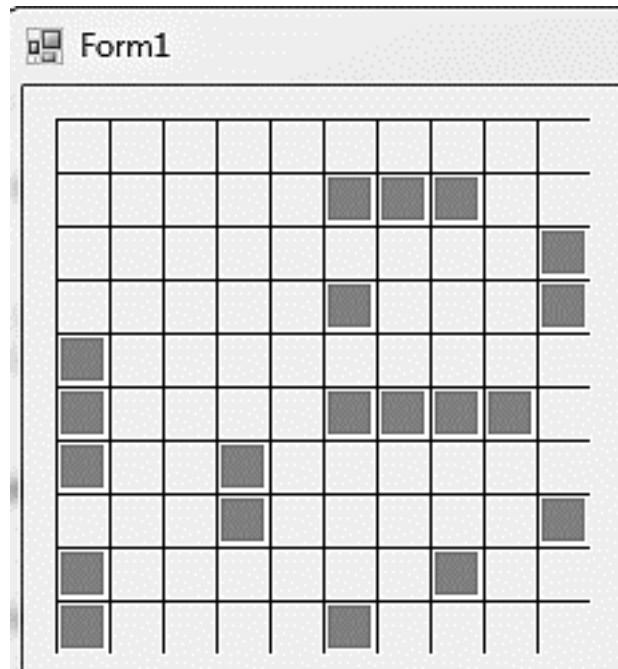


Рисунок 1.67. Расставленные компьютером корабли

Обратите внимание!

- *В данном коде не объявлен генератор случайных чисел, так как предполагается, что его мы объявим вместе с массивами.*

Теперь мы можем реализовать расстановку кораблей и для человека.

Расстановка кораблей человеком

Основная идея алгоритма: мы предполагаем, что человек щелкает мышкой 1 раз по полю, указывая начальную точку, а затем щелкает второй раз для указания направления. Компьютер пытается поставить корабль по указанным координатам и, в случае неудачи, выдает сообщение об ошибке и просит повторить задание условий.

Для организации хода человека нам понадобятся: четыре radioButton-а для указания того, какой тип корабля в данный момент ставится. Кроме того, нам понадобятся три глобальных числовых переменных `hod` – счетчик ходов, `x_n` – координата X начальной точки, `y_n` – координата Y конечной точки. В элементах radioButton свойство `Text` будет содержать количество палуб корабля, а свойство `Tag` – количество кораблей, которые мы должны поставить. Свойство `Checked` будет отвечать за выбор корабля для расстановки. Как только корабль будет установлен, его свойство `Tag` уменьшится на 1, а при достижении 0 radioButton заблокируется. На рисунке 1.68 показано, что для radioButton1, отвечающего за четырехпалубники? свойство `Text` установлено в 4, а свойство `Tag` – в 1.

Для более удобной реализации напомним несколько дополнительных функций. Во-первых, функцию **получения числа палуб** в зависимости от того, какой radioButton выбран. Работать она будет просто: для каждого radioButton мы проверяем, выбран он или нет. Если выбран, то сверяемся на всякий случай с его свойством `Tag`: если записанное там число больше 0, то возвращаем преобразованное к числу значение свойства `Text`. В приведенном ниже коде (листинг 1.94) – начало данной функции для первого radioButton. Код для всех остальных вам предлагается дописать самостоятельно.

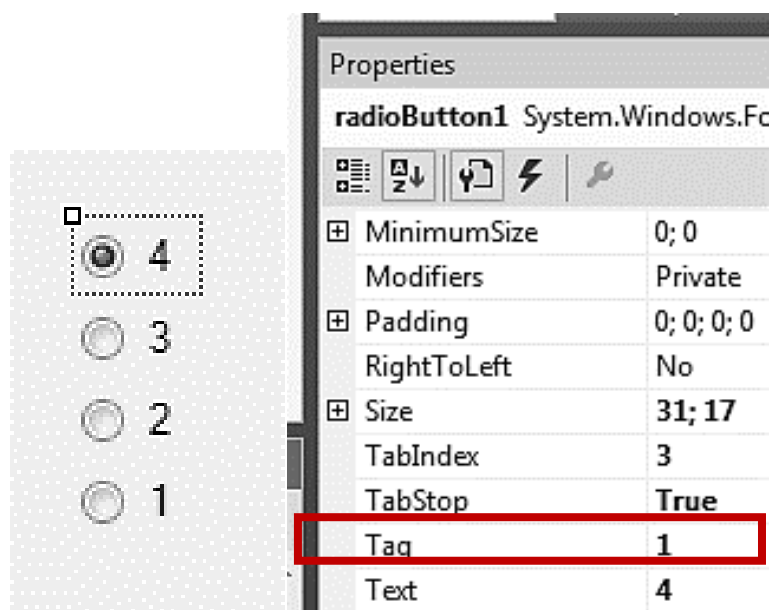


Рисунок 1.68. Установка свойств RadioButton

Листинг 1.94

```
public int recive_i(){ if (radioButton1.Checked)
{ if (Convert.ToInt32(radioButton1.Tag.ToString()) > 0)
    return Convert.ToInt32(radioButton1.Text.ToString());
//самостоятельно дописать для остальных RadioButton
    return -1;}
```

Во-вторых, напомним функцию **обновления свойства Tag**, которая будет вызываться при успешной установке корабля, а также блокировки radioButton при достижении свойством Tag 0. Приведенный ниже код (листинг 1.95) также описывает работу лишь с первым radioButton.

Листинг 1.95

```
public void set_i(){ if (radioButton1.Checked){
radioButton1.Tag= (Convert.ToInt32(radioButton1.Tag.ToString())-1).ToString();
radioButton1.Enabled = Convert.ToInt32(radioButton1.Tag.ToString()) > 0;
radioButton1.Checked = radioButton1.Enabled;}}
```

В-третьих, нам нужна функция **перевода экранных координат в строки и столбцы массива**. Причем, так как в нашем случае ширина поля равна его высоте, то мы можем реализовать одну функцию для обеих координат. Для получения требуемой величины нам нужно поделить экранный x или y на десятую часть стороны поля и округлить результат до целого. Если мы объявим функцию, получающую и возвращающую значение типа int, то округление до целого среда выполнит за нас самостоятельно. Код функции будет следующим (листинг 1.96).

Листинг 1.96

```
public int mas_kur(int x){ return x/(pictureBox1.Height/10);}
```

В-четвертых, нам нужна функция **определения выбранного пользователем направления**. Работать она будет следующим образом: в функцию будут передаваться координаты клеток массива, в ко-

торые пользователь кликнул первый и второй раз (то есть начальные и конечные точки). Мы посчитаем, где больше отклонение: по горизонтали или вертикали, и в зависимости от этого будем определять, куда нам попробовать поставить корабль относительно исходной точки: влево, вправо, вниз или вверх. По сути, эта функция определяет нам параметр *n* (от 0 до 3) для *func2*. Код функции представлен в листинге 1.97.

Обратите внимание!

- *В приведенном ниже коде также отсутствуют именованные константы направлений.*

Листинг 1.97

```
public int get_n(int xn,int yn,int xk,int yk){
int x = (xk - xn); int y = (yk - yn);
bool gor = Math.Abs(x)>Math.Abs(y); //горизонталь или нет
if (gor){if (x<0) return 3; if (x>0) return 2;}
else {if (y<0) return 1;if (y>0) return 0;}
return 0;}
```

Теперь напомним с помощью наших функций расстановку кораблей человеком. Будем считать, что его поле отображается в *pictureBox2*. Тогда переходим в события элемента *pictureBox2*, находим в списке *MouseClicked* и щелкаем рядом с ним 2 раза (рисунок 1.69).

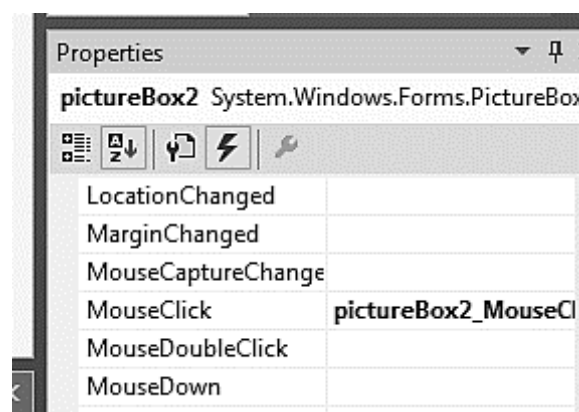


Рисунок 1.69. Привязка реакции на клик мыши

В коде будет выполняться следующее.

Во-первых, мы проверяем, отмечен ли какой-нибудь из radioButton, и если условие истинно, то увеличиваем счетчик ходов на 1 (листинг 1.98).

Листинг 1.98

```
if (radioButton1.Checked || radioButton2.Checked || radioButton4.Checked ||  
radioButton3.Checked) hod++;
```

Затем проверяем, если отмечен radiobutton, отвечающий за однопалубники, то переводим координаты курсора в координаты массива, запоминаем их в соответствующих переменных и увеличиваем счетчик ходов. Последнее сделано потому, что для однопалубника не надо задавать направление. Реализующий это код представлен в листинге 1.99.

Листинг 1.99

```
if (radioButton4.Checked)  
{ y_n = mas_kur(e.X, -1);  
  x_n = mas_kur(-1, e.Y); //stroka  
  hod++;  
}
```

Далее мы проверяем, если счетчик ходов нечетный (то есть пользователь кликнул 1, 3 или 5 раз), то мы всего лишь переводим координаты курсора в координаты массива, запоминаем их в соответствующих переменных и ждем следующего клика (повторяется код, приведенный для однопалубника, но без увеличения счетчика шагов).

Если же количество ходов (кликов) четно, то с помощью описанных нами функций мы пытаемся поставить корабль. Если нам это не удастся, мы просим пользователя выбрать новую начальную точку и направление, иначе мы меняем свойство Tag соответствующего radioButton (листинг 1.100).

Листинг 1.100

```
if (func2(x_n, y_n, recive_i(), gamer, get_n(x_n, y_n, mas_kur(e.Y), mas_kur(e.X)))) {  
    set_i(); }  
else { MessageBox.Show("Выберите новую начальную точку и направление!");}
```

После мы выполняем отрисовку, как представлено в листинге 1.101.

Листинг 1.101

```
Bitmap bm = new Bitmap(pictureBox2.Width, pictureBox2.Height);  
Graphics gr = Graphics.FromImage(bm);  
vivod(gamer, gr, true);  
pictureBox2.Image = bm;
```

Таким образом, мы реализовали заполнение кораблями обоих полей, и теперь можем переходить к реализации игрового процесса, начав с более простого: с хода человека.

Программирование ход человека

Суть алгоритма для этой задачи будет следующей: человек кликает по клетке на поле (то есть код обработки вешаем на `MouseClicked` первого `pictureBox`). Если клик попадает в корабль, то в массив компьютера заносится 2, а в месте клика рисуется крестик. Если не попадает – заносится -1 и рисуется черный кружок. Как только корабль разбивается полностью, он окружается черными точками. Если это последний корабль на поле, то все пустые клетки также заполняются черными точками и выводится сообщение о победе.

Далее нам нужно сделать следующее:

1. Реализовать обработку клика с определением попадания-непопадания.
2. Написать функцию отыскания разбитых кораблей и окружение их промахами.
3. Написать функцию определения победы.

С первым все просто: с помощью разработанной ранее функции `mas_kur` находим координаты клика, определяем, что находится в клетке по указанным координатам, и в зависимости от этого меняем значение текущего элемента массива. Затем вызываем функцию поиска разбитых кораблей, потом функцию проверки победы и функцию отрисовки.

Второй пункт гораздо сложнее. Сначала определимся с понятием «разбитый корабль». Для нашего массива разбитый корабль это непрерывная горизонтальная или вертикальная последовательность двоек, ограниченная нулями или -1 или границами поля. То есть разбитый корабль (например, вертикальный двухпалубник) может быть изображен так (рисунок 1.70).

-1	0	-1	0
2	2	2	2
2	2	2	2
0	0	-1	-1

Рисунок 1.70. Четыре случая кодировки разбитого корабля

Приняв это понятие за основу, мы можем реализовать нашу функцию для всех не однопалубников следующим образом. Бежим по массиву, находим 2. Как только мы ее нашли, объявляем 4 переменных, равные -1. Они нам понадобятся для запоминания ограничивающих двойки ячеек. Далее от ячейки с двойкой идем вправо до первой НЕ двойки. Если нам встретилась 1, то прекращаем просмотр этого корабля и идем к следующей двойке. Если нам встретился 0 или -1, то запоминаем эту позицию. То же самое делаем для хода влево. Затем аналогично для вертикали и горизонтали. В результате, для каждого убитого корабля у нас будут по две переменные (k и $k2$ для горизонтали и $k3$ и $k4$ по вертикали). Если мы внимательно посмотрим на наши корабли, то увидим, что разбитый корабль (не однопа-

лубный) вписывается в правило: разница между граничными точками больше двух клеток ($k-k_2 > 2$ или $k_4-k_3 > 2$). Значит, мы можем отработать по алгоритму: Если расстояние между краями больше двух, то заполняем минус единицами область вокруг корабля от границы -1 до границы $+1$.

Исключения из этого алгоритма: однопалубники и двухпалубники, стоящие на краю поля указанным образом (рисунок 1.71).

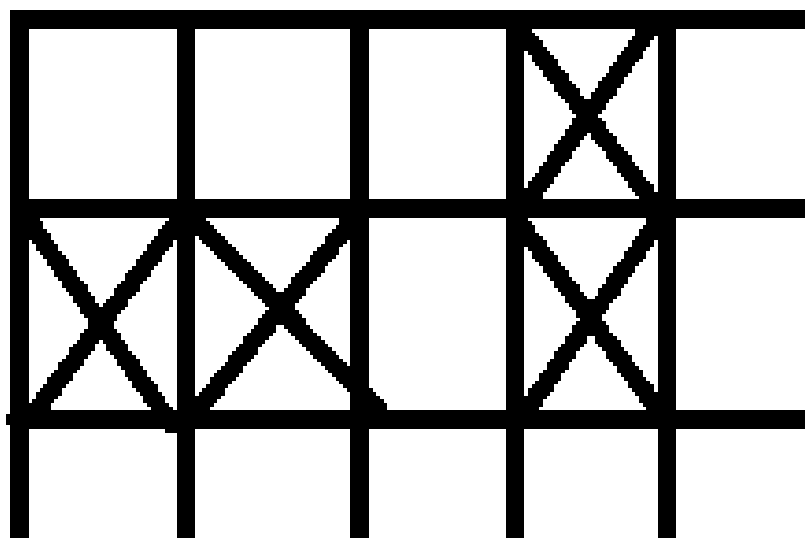


Рисунок 1.71. Исключительная ситуация для двухпалубников

Значит, однопалубники мы должны обработать отдельно. То есть как только мы нашли в массиве двойку, мы проверяем, является ли она однопалубником. Реализовать это очень просто: мы проверяем клетку под ней, над ней, слева и справа на отсутствие 1. Если это так, то заполняем окружность клетки -1 , не забывая отслеживать границы. Как только наш индекс выходит за границы, мы пропускаем его обработку ключевым словом `continue`. Этот оператор автоматически возвращает исполняемый код к заголовку цикла, игнорируя идущие далее операторы. Цикл для проверки заполнения окружности клетки можно взять из функции `func`.

Для двухпалубников мы просто отслеживаем, не вышла ли какая-то граница за края поля. Как только это произошло, мы выстав-

ляем в истину дополнительную логическую переменную, и при проверке расстояния в две клетки дополняем условие, что расстояние может быть равно двум клеткам, при условии, что дополнительная переменная стоит в значении истина.

Приведем ниже ключевые фрагменты кода.

Как только мы нашли в массиве двойку, необходимо объявить дополнительные переменные, как показано в листинге 1.102.

Листинг 1.102

```
int x = 0;
bool fv = false; // проверка для двухпалубников,
// расположенных на краю
bool flag = true; // проверка прерывания последовательности
int k = -1; // позиция не двойки внизу
int k2 = -1; // позиция не двойки вверху
int k3 = -1; // позиция не двойки слева
int k4 = -1; // позиция не двойки справа
```

Пример кода для отслеживания последовательности по вертикали, вниз от текущей двойки (считается, что `mas[i,j]==2`) приведен в листинге 1.103.

Листинг 1.103

```
x = 0; flag = true;
while (flag) { x++;
    if (i + x < 10){ if (mas[i + x, j] != 2) {
if (mas[i + x, j] == 1) { k = -1; break; }
else if (mas[i + x, j] == 0 || mas[i + x, j] == -1) k = i + x; } flag = false; }
    else { k = 9; fv = true; flag = false; }}
if (flag) continue; // нам встретилась 1, смотреть корабль нет смысла.
```

Код окружения корабля по вертикали приведен в листинге 1.104.

Листинг 1.104

```

if (k!=-1&&k2!=-1) { if ((k - k2) > 2||(k-k2==2&(fv)) )
    {for (int t = -1; t <= 1; t++) {
        if (j + t < 0 || j + t > 9) continue;
        if (mas[k, j + t] ==0) mas[k, j + t] = -1;
        if(mas[k2, j + t] ==0) mas[k2, j + t] = -1; }
    for (int t = k2; t <= k; t++) {
if (mas[t, Math.Max(j - 1, 0)]==0)
mas[t, Math.Max(j - 1, 0)] = -1;
if (mas[t, Math.Min(j + 1, 9)]==0)
mas[t, Math.Min(j +1, 9)] = -1;} } }

```

Функция определения победы будет принимать массив, в котором организуется поиск 1. Как только нашли 1, возвращаем false. Если же мы благополучно вышли из цикла, пройдя по всему массиву и не встретив 1, то после цикла возвращаем true. В ходе человека, если функция определения победы вернула true, бежим по массиву, меняя все нули на -1, и выводим сообщение о победе. Реализовать ее вам предлагается самостоятельно. Код же обработки клика мыши будет следующим (листинг 1.105).

Листинг 1.105

```

private void pictureBox1_MouseClick(object sender, MouseEventArgs e){
    Bitmap bm = null;
    Graphics gr = null;
    x_n = mas_kur(e.Y); //stroka
    y_n = mas_kur(e.X);
    if (computer[x_n, y_n] == 1) { gamer[x_n, y_n] = 2;}
    if (computer [x_n, y_n] == 0) { gamer[x_n, y_n] = -1;}
    //поиск разбитых кораблей
    find_crash_ship(computer);
    //проверка, не выиграл ли человек
    if (findEnd(computer)){
        for (int i = 0; i < 10; i++)
            for (int j = 0; j < 10; j++)
                if (computer [i, j] == 0) computer [i,j]=-1;
    }
}

```

Окончание листинга 1.105

```
    MessageBox.Show("Вы выиграли!");}
bm = new Bitmap(pictureBox1.Width, pictureBox1.Height);
gr = Graphics.FromImage(bm);
vivid(computer, gr,true);
pictureBox1.Image = bm; }
```

На этом работа над ходом человека завершена. Теперь мы можем перейти к финальной части реализации нашей игры, разработав самое трудное – ход компьютера. Основная сложность здесь будет заключаться в том, чтобы запрограммировать компьютер «добивать» корабль после попадания.

Ход компьютера

Суть алгоритма: компьютер стреляет в случайную точку. Если он промахнулся – отмечается промах, ход переходит к человеку. Если компьютер попал, мы запоминаем координаты точки и выбираем направление «добивания». Далее компьютер снова стреляет. Если попал – продолжаем просмотр вдоль выбранного направления, если промах – запоминаем, что нужно сменить направление и передаем ход игроку.

Для реализации алгоритма введем несколько дополнительных глобальных переменных (листинг 1.106).

Листинг 1.106

```
int x_tek = -1; //для хранения столбца точки попадания
int y_tek = -1; //для хранения строки точки попадания
bool est_hod = true; //для отслеживания права хода компьютера
//(надо ли передавать ход игроку или компьютер еще стреляет)
//Константы направлений:
const int nikuda = -1;
const int niz = 0;
const int verh = 1;
const int levo = 2;
const int pravo = 3;
int napr_tek = nikuda; // для хранения направления добивания
```

Теперь напишем функцию, которая будет выполнять «стрельбу» от текущей точки в заданном направлении. В качестве параметров она будет принимать массив, а также смещение по x и y. Возвращаться будет логическое значение, говорящее об успешности стрельбы в выбранном направлении.

В теле функции мы на всякий случай проверим, есть ли ход у компьютера. Если нет, то вернем значение ложь. Следующим шагом проверяем границы для текущей точки плюс смещение. Если вышли за рамки массива – меняем направление на противоположное. Если не вышли за рамки массива, то продолжаем проверку.

Если в указанных координатах – корабль, то меняем в массиве 1 на 2 и запоминаем направление в зависимости от значения смещения. Далее проверяем, не убили мы корабль, вызвав функцию поиска убитых кораблей. Затем – запоминаем текущие точки, как исходные плюс смещение, и возвращаем истину.

Теперь опишем действия, которые будут происходить, если в указанных координатах нет корабля. Это может случиться, если мы попали в пустую клетку или в промах. Тогда в случае пустой клетки меняем ее на промах и забираем у компьютера право хода, а также меняем направление на противоположное и возвращаем ложь. Если же мы попали в клетку с подбитым кораблем, то продолжаем идти по выбранному направлению, меняя координаты текущих точек, пока не кончатся двойки или пока не упремся в границу. Затем возвращаем ложь. Код описанной функции представлен в листинге 1.107.

Листинг 1.107

```
bool Strelba(int[,] mas,int dy,int dx)
{
    if (!est_hod) return false;
    if (x_tek + dx < 10 && x_tek + dx > -1 && y_tek + dy < 10 && y_tek + dy > -1){
        if (mas[y_tek + dy, x_tek + dx] == 1){
            mas[y_tek + dy, x_tek + dx] = 2;
        }
        if (napr_tek == nikuda){
            if (dx > 0 && dy == 0) napr_tek = pravo; else
```


Окончание листинга 1.107

```
if (dx < 0 && dy == 0) napr_tek = levo; else
if (dy < 0 && dx == 0) napr_tek = verh; else
if (dy > 0 && dx == 0) napr_tek = niz;}
find_crash_ship(mas);
if (x_tek != -1) x_tek += dx;
if (y_tek != -1) y_tek += dy;
return true;}
else
if (mas[y_tek + dy, x_tek + dx] == 0 || mas[y_tek + dy, x_tek + dx] == -1){
if (mas[y_tek + dy, x_tek + dx] == 0){ mas[y_tek + dy, x_tek + dx] = -1;
est_hod = false;}
if (napr_tek!=nikuda){
if (napr_tek == levo) napr_tek = pravo; else
if (napr_tek == pravo) napr_tek = levo; else
if (napr_tek == niz) napr_tek = verh; else
if (napr_tek == verh) napr_tek = niz;
return false;}
return false;}
else if ( mas[y_tek + dy, x_tek + dx] == 2){
do{ x_tek += dx; y_tek += dy;
if (x_tek+dx < 0 || y_tek+dy < 0 || x_tek+dx > 9 || y_tek+dy > 9) break;} while
(mas[y_tek+dy, x_tek+dx] == 2) ;
return false; }}
else{
if (napr_tek == levo ) napr_tek = pravo; else
if (napr_tek == pravo ) napr_tek = levo; else
if (napr_tek == niz ) napr_tek = verh; else
if (napr_tek == verh ) napr_tek = niz;
return false;}return false;}
```

Теперь реализуем процедуру «Ход компьютера». Вызываться она будет при промахе человека, а в качестве параметра принимать массив. Первым шагом в теле функции мы «дадим компьютеру право хода» и организуем цикл, работающий, пока это право хода будет действовать. Внутри цикла мы проверим, если у нас есть текущая

точка для стрельбы (ее координаты не равны -1), то есть прошлым ходом мы куда-то попали, то проверим, есть ли у нас направление. Если направления пока нет, то пробуем выстрелить вправо, вызвав функцию `Strelba`. Если попали – запоминаем направление. Если нет – пробуем выстрелить в другом. Если же направление у нас уже есть, то стреляем в имеющемся у нас направлении.

Если мы стреляем первый раз, то выбираем случайную свободную точку (не -1 и не 2). Проверяем – если мы попали, то меняем значение на 2 и проверяем, не добились ли мы корабль. Если промахнулись – меняем значение на -1 , присваиваем текущим координатам -1 , «отбираем у компьютера право хода», меняем направление на «никуда».

Последним шагом проверяем, не выиграл ли компьютер. Если выиграл – пишем в оставшиеся пустые клетки -1 , выводим соответствующее сообщение, и отключаем пользователю возможность ходить, меняя свойство `Enable` у `pictureBox1` в `false`. Код описанной функции представлен в листинге 1.108.

Листинг 1.108

```
void hod_kompa(int[,] mas){
    est_hod = true;
    while (est_hod){
        if (x_tek != -1 && y_tek != -1)
        {
            if (napr_tek==nikuda) //ищем следующую клетку
            {if (Strelba(mas, 0, 1))
                { if (napr_tek == nikuda) napr_tek = pravo;}
                else
                if (Strelba(mas, 1, 0))
                {if (napr_tek == nikuda) napr_tek = niz;}
                else
                if (Strelba(mas, 0, -1))
                {if (napr_tek == nikuda) napr_tek = levo;}
                else
                if (Strelba(mas, -1, 0))
```

Окончание листинга 1.108

```
        { if (napr_tek == nikuda) napr_tek = verh; }  
    }  
    else //добить  
    { if (napr_tek == pravo) Strelba(mas, 0, 1);  
      if (napr_tek == levo) Strelba(mas, 0, -1);  
      if (napr_tek == niz) Strelba(mas, 1, 0);  
      if (napr_tek == verh) Strelba(mas, -1, 0);}}  
    else { do{x_tek = r.Next(0, 10); y_tek = r.Next(0, 10); }  
          while (mas[y_tek, x_tek] == 2 || mas[y_tek, x_tek] == -1);  
      if (mas[y_tek, x_tek] == 1){  
          mas[y_tek, x_tek] = 2; find_crash_ship(mas);}  
      else if (mas[y_tek, x_tek] == 0){  
          mas[y_tek, x_tek] = -1;  
          napr_tek = nikuda;  
          x_tek = -1; y_tek = -1; est_hod = false;}}}  
    if (findEnd(mas)){  
        for (int i = 0; i < 10; i++)  
            for (int j = 0; j < 10; j++)  
                if (mas[i, j] == 0) mas[i, j] = -1;  
        MessageBox.Show("Я выиграл!");  
        pictureBox1.Enabled = false;}}
```

Мы рассмотрели реализацию одной из классических игр, где используются двумерные массивы. Как можно заметить, на этой структуре данных достаточно легко реализовывать игры с большой логической составляющей. Более того, на двумерных массивах разработаны специальные алгоритмы, применяемые в играх. С одним из них мы познакомимся в следующей теме, сейчас же вам рекомендуется выполнить несколько заданий по пройденному материалу.

ЗАДАНИЯ К ТЕМЕ 1.10

1. Реализуйте полностью игру «Морской бой».
2. Оптимизируйте код игры «Морской бой».

3. Разделите логику и интерфейс игры «Морской бой».
4. Улучшите интерфейс игры, заменив графические примитивы картинками и добавив анимацию.
5. Введите уровень сложности «новичок»: компьютер не добывает корабли, а «стреляет» по случайным клеткам. А также введите уровень сложности «эксперт», где направление добывания первый раз выбирается случайно.
6. Введите подсчет очков и организуйте вывод и хранение таблицы рекордов.
7. Разработайте игру «Крестики-нолики».
8. Разработайте игру «Тетрис».
9. Разработайте игру «Сапер».
10. Разработайте игру «Хожение в лабиринте»: задача игрока собрать как можно больше призов за отведенное время, перемещая своего героя по лабиринту. Сделать два уровня сложности. «Новичок» – герой может разбивать стены и «Эксперт» – стены придется обходить.

ТЕМА 1.11. РАЗРАБОТКА ИГРЫ «ЛИНИИ»

В данной теме мы рассмотрим еще одну игру, разрабатываемую с использованием двумерных массивов, а именно – «Линии». Суть данной игры простая: на поле размером десять на десять появляются шарики пяти цветов по три за раз. Задача игрока – составлять линии из пяти и больше шариков одного цвета в любом из четырех направлений, перемещая шарики по полю. Такие линии исчезают, а в зависимости от их длины начисляются очки. Если перемещенный шарик не привел к исчезновению линии – появляются три новых. Если привел – не появляются. Игра заканчивается, когда на поле не остается пустых клеток. Перемещение шарика организуется следующим образом: игрок кликает на шарик, а затем на клетку, куда его хочет переместить. Если от исходной точки до конечной можно добраться по

горизонталям и вертикалям, не столкнувшись с другими шарами, то шарик «пробегает» указанный путь. На рисунке 1.72 показаны варианты открытого пути (левая доска) и закрытого (правая).

Как и в случае с морским боем, определимся с требуемыми структурами данных. Так как действие игры происходит, как и в морском бое, на поле фиксированного размера, то мы можем использовать двумерный массив. А так как нам необходимо хранить информацию о пяти типах шариков, то мы можем использовать для этого цифры от 1 до 5.

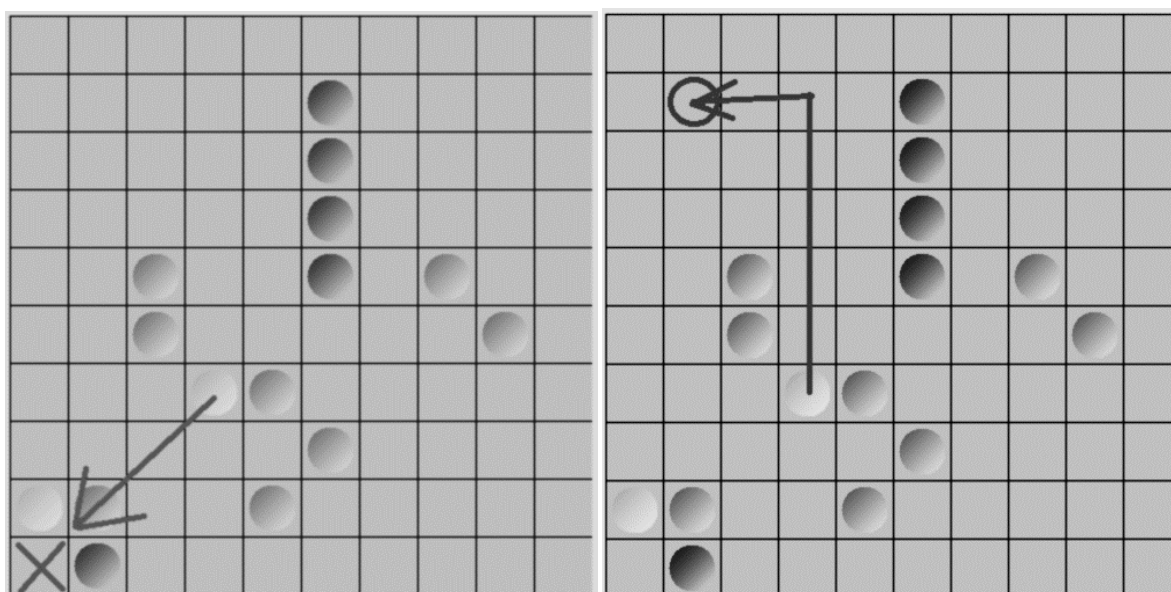


Рисунок 1.72. Игра «Линии»

Основная сложность данной игры – заставить шарик «бежать» от исходной до конечной точки по кратчайшему пути. То есть построить для него этот путь. Однако данная задача является по сути частным случаем задачи поиска кратчайшего пути в лабиринте (в нашем варианте стенами будут другие шары). Поэтому мы можем использовать один из классических алгоритмов ее решения: волновой алгоритм.

Волновой алгоритм

Согласно Википедии [1]: «Алгоритм волновой трассировки (волновой алгоритм, алгоритм Ли) – алгоритм поиска кратчайшего пути на планарном графе. Принадлежит к алгоритмам, основанным на методах поиска в ширину. В основном используется при компьютерной трассировке (разводке) печатных плат, соединительных проводников на поверхности микросхем. Другое применение волнового алгоритма – поиск кратчайшего расстояния на карте в компьютерных стратегических играх. Волновой алгоритм в контексте поиска пути в лабиринте был предложен Э. Ф. Муром. Ли независимо открыл этот же алгоритм при формализации алгоритмов трассировки печатных плат в 1961 году».

По другой классификации волновой алгоритм можно отнести к эвристическим алгоритмам. Под эвристическими алгоритмами понимаются такие алгоритмы, в которых на определенном этапе используется интуиция разработчика. От правильности принятого интуитивного решения зависит скорость работы алгоритма.

Задача, решаемая данным алгоритмом, может формулироваться следующим образом: дано поле размером n на n клеток, содержащее значения: 0 – пустое поле; -1 – поле непроходимо; -2 – точка отправления (А); -3 – точка назначения (В). Необходимо по проходимым полям построить кратчайший путь от точки А до точки В.

Алгоритм состоит из двух фаз: проход волны и построения пути. Фаза прохода волны содержит следующие шаги:

1. Установить фронт волны, равный номеру шага.
2. От точки А по четырем направлениям заполнить пустые клетки фронтом волны.
3. Если ни по одному из направлений не обнаружилось конечной точки, и волна не уперлась в границу поля или преграду, то увеличить фронт волны на 1.
4. Повторить шаги 2-3 для всех клеток текущего фронта.

Фаза построения пути содержит следующие шаги:

1. В округе точки В найти точку А. Если она не найдена – найти меньший фронт волны.

2. Принять найденную точку за исходную и повторить шаг 1.

Адаптируем данный алгоритм под нашу задачу. Так как шарики в игре кодируются цифрами от одного до пяти, то для простоты реализации в конечную точку пути запишем значение 10, а для фронта волны будем использовать отрицательные числа, начав с -2 . В этом случае номер шарика никогда не совпадет со значением фронта волны. Тогда при нахождении кратчайшего пути мы будем искать наибольшее отрицательное в округе точки.

На рисунке 1.73 показаны начальные и конечные точки (а), а также распространение первых трех фронтов (б, в, г). На рисунке 1.74 – окончание распространения волны и кратчайший путь в виде массива.

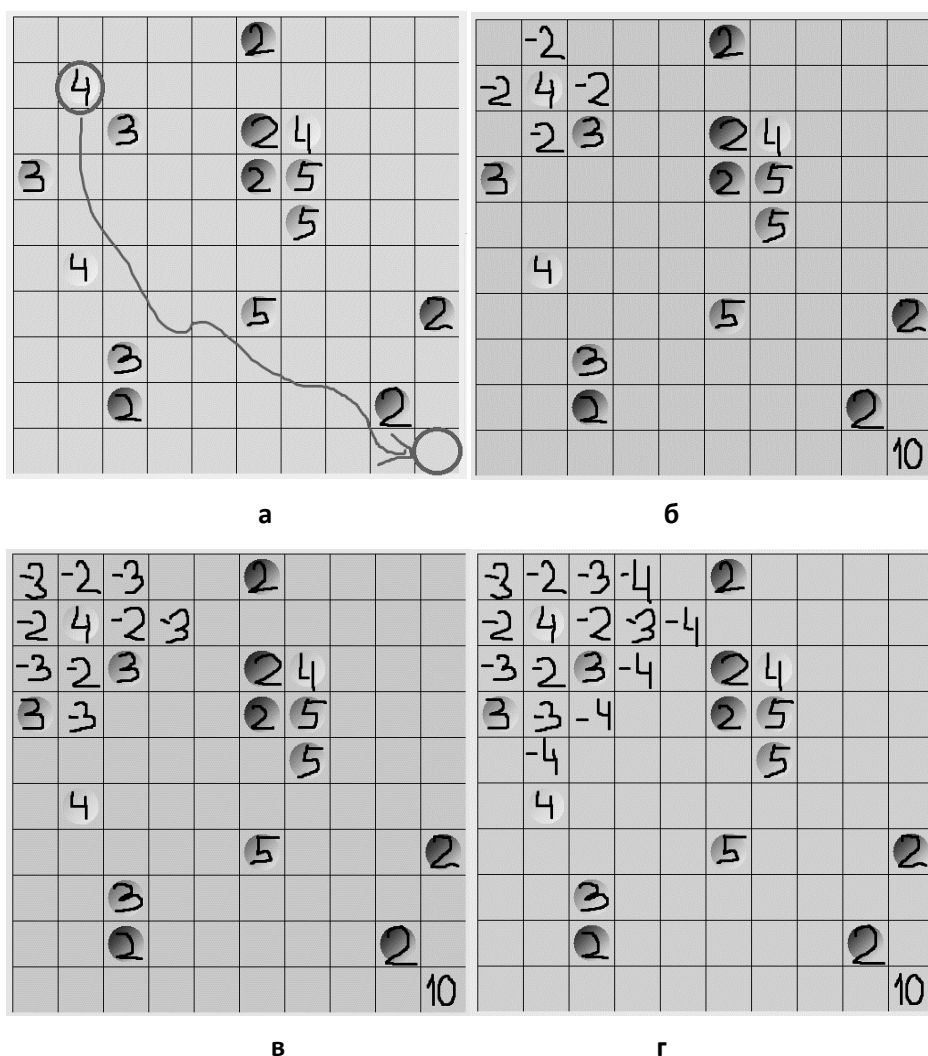


Рисунок 1.73. Распространение волны

Как вы можете заметить, на самом деле кратчайших путей несколько, однако все они приводят к одному и тому же результату. От выбора пути будет зависеть лишь анимация: по какой конкретно траектории шарик будет двигаться от начальной точки к конечной. Именно тут и включается эвристика: при поиске следующей точки нам придется выбирать направление просмотра (влево, вверх, вправо, вниз). Этот порядок в итоге и определит траекторию.

Примечание: на рисунке 1.74 отображена просто одна из возможных траекторий.

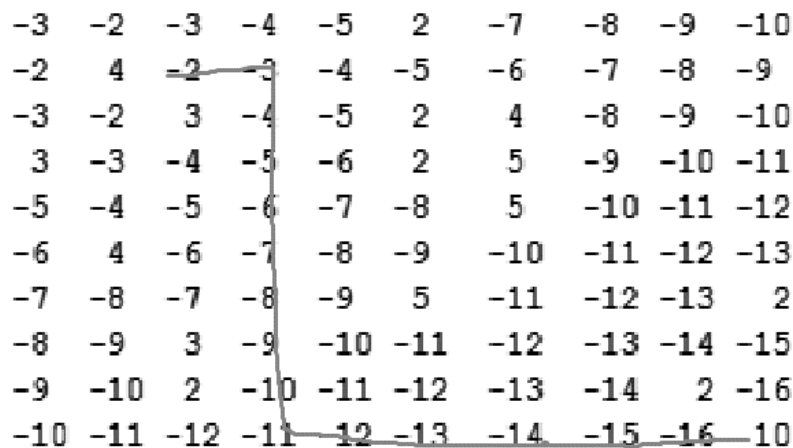


Рисунок 1.74. Кратчайший путь

На рисунке 1.75 показано построение волны для несуществующего пути. Начальные и конечные точки обведены, преграды – подчеркнуты.

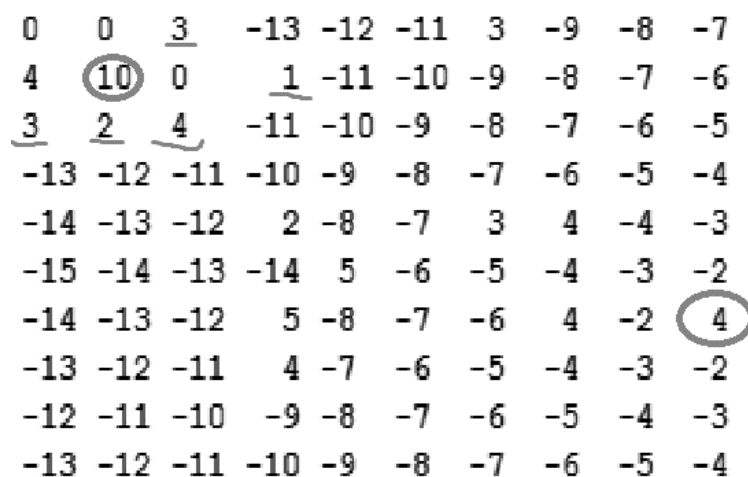


Рисунок 1.75. Несуществующий путь

Данный алгоритм можно реализовать разными способами, и мы рассмотрим его реализацию с использованием рекурсии.

Реализация волнового алгоритма

Во-первых, рассмотрим определение рекурсии в области программирования. Согласно Википедии [1]: «Рекурсия – вызов функции (процедуры) из нее же самой, непосредственно (простая рекурсия) или через другие функции (сложная или косвенная рекурсия), например, функция А вызывает функцию В, а функция В – функцию А. Количество вложенных вызовов функции или процедуры называется глубиной рекурсии. Рекурсивная программа позволяет описать повторяющееся или даже потенциально бесконечное вычисление, причем без явных повторений частей программы и использования циклов.»

В нашем случае на основе рекурсии можно построить следующую реализацию волнового алгоритма:

1. Заранее присвоим конечной точке значение 10.
2. Объявим функцию `search`, принимающую в качестве параметров массив, координаты текущей точки и фронт волны (на первом шаге – начальной).
3. В теле функции объявим вспомогательную переменную для хранения результата, который вернет цепочка рекурсивных вызовов.
4. По каждому направлению проверим: если следующая точка не выходит за границы поля, то проверим, не является ли она точкой назначения. Если является – вернем текущий фронт волны.
5. Иначе проверим, если клетка пустая или ее значение меньше текущего фронта волны, то присвоим ей значение следующего фронта волны и вызовем функцию `search` для нашей новой точки и следующего фронта волны.
6. Проверим результат цепочки рекурсивных вызовов. Если он больше нуля и значение переменной, хранящей результат це-

почки рекурсивных вызовов больше него или это значение нулевое, обновим переменную, хранящую результат цепочки рекурсивных вызовов.

7. В конце метода вернем переменную, хранящую результат цепочки рекурсивных вызовов.

Ниже приведен код для проверки точки в следующем столбце (листинг 1.109). Обход остальных направлений вам предлагается реализовать самостоятельно.

Листинг 1.109

```
int search(int[,] pole,int x1, int y1, int i) {int k = 0;
if (y1 + 1 < pole.GetLength(1)){ if (pole[x1, y1 + 1] == 10) { return i; } else
if (pole[x1, y1 + 1] == 0 || (pole[x1, y1 + 1] < -i)){ pole[x1, y1 + 1] = -(i + 1); int result
= search2(x1, y1 + 1, 1 + i);
if ((result > 0) && (k > result || k == 0)) k = result;}}
//код для остальных направлений.
return k;}
```

Для отладки алгоритма создайте тестовой приложение с элементом textbox и кнопкой. У textbox свойство MultiLine установите в значение true и растяните элемент по горизонтали и вертикали. Кроме того, в свойстве Font выберите значение Courier New для более удобного отображения информации. В этом тестовом приложении разработайте метод search, а обработку нажатия кнопки реализуйте, как показано в листинге 1.110.

Листинг 1.110

```
private void button1_Click(object sender, EventArgs e) { int[,] mas = new int[10, 10];
mas[0, 0] = 10;//конечная точка
mas[1, 3] = 4; mas[3, 3] = 4; mas[4, 3] = 4; mas[1, 5] = 4; mas[1, 4] = 4;mas[7, 6] = 4;
mas[8, 8] = 100;//начальная точка
search(mas, 8, 8, 1); for (int i = 0; i < 10; i++) { for (int j = 0; j < 10; j++)
textBox1.Text += mas[i, j].ToString().PadLeft(4) + " ";
textBox1.Text += Environment.NewLine;}}
```

Если вы все сделаете верно, то приложение должно выдать следующий результат для указанных исходных данных (рисунок 1.76). Меняя исходные данные, протестируйте работу вашего алгоритма в разных условиях, убедившись, что он отрабатывает во всех ситуациях.

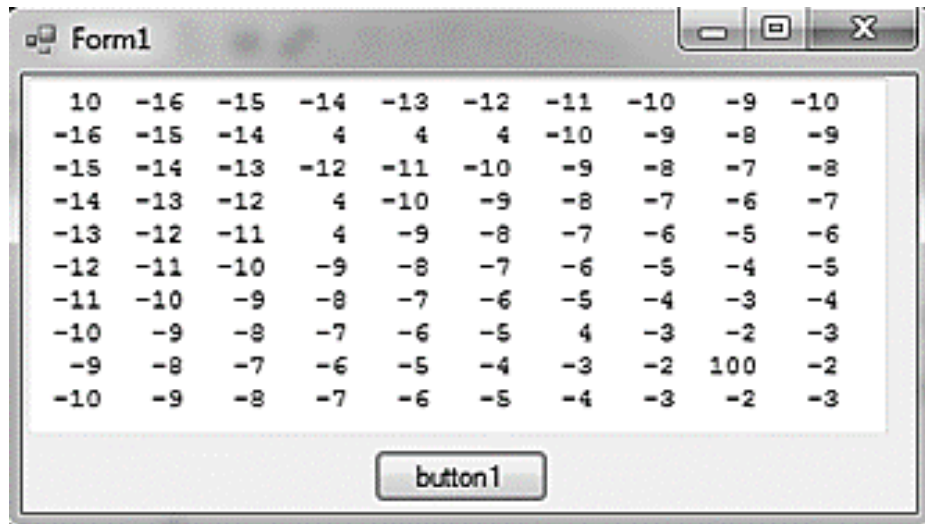


Рисунок 1.76. Тестирование построения волны

Обратите внимание!

- При выводе числа выстроились ровными столбиками. Это достигнуто за счет настройки свойства *Font* и применение метода *PadLeft*. Дело в том, что *Courier New* относится к так называемым «моноширинным» шрифтам – символы в нем имеют одну и ту же ширину (что 0, что 1, что тире, что буква, что пробел). Метод *PadLeft* дополняет строку пробелами слева таким образом, что ее длина становится равной указанному в скобках параметру.

Теперь перейдем к части построения пути. Для реализации анимации, когда шарик перемещается последовательно от клетки к клетке, мы сохраним все точки пути в строку, сформировав путь.

Общий алгоритм будет следующим:

1. Записать в конечную точку 10.

2. Запустить search для начальной точки и фронта волны равного 1.

3. Для начальной точки найти соседнюю максимальную точку фронта волны. Если ее не существует (нам некуда идти), сообщить о невозможности построения пути. Если она является конечной, добавить ее в путь, используя в качестве разделителя пробел и восклицательный знак. Обнулить все отрицательные элементы массива, а также конечную точку. Вернуть путь.

4. Если в шаге 3 мы не вышли из функции, то повторить этот шаг для найденной точки.

Определение максимальной точки вынесем в отдельную функцию. В качестве параметров она будет принимать координаты точки назначения и координаты текущей точки. Возвращать же функция будет координаты новой точки, а также информацию об окончании поиска. Для реализации этого объявим класс, который позволит нам объединить в себе значения разных типов данных (листинг 1.111).

Листинг 1.111

```
class Max{ public int i; public int j;public bool result;  
public Max(int i_,int j_,bool result_){i = i_;j = j_; result = result_;}}
```

Код функции представлен в листинге 1.112. Алгоритм же ее работы следующий:

1. Проверить, не является соседняя точка конечной. Если да – вернуть ее координаты и установить результат поиска в значение true.

2. Проверить, что от текущей точки мы можем куда-либо двинуться в принципе. Для этого объявляем логическую переменную со значением false. Затем проверяем каждую из соседних точек на то, что ее значение отрицательно. Если это так – меняем значение логической переменной на true. Если значение логической переменной так и осталось ложно, возвращаем Max с отрицательными координатами (-1, -1).

3. Найти первую отрицательную точку в окрестностях текущей. Для этого рассмотреть точку в предыдущей строке. Если предыдущая строка за границами массива, то рассмотреть следующую строку. Если данная точка не отрицательна, рассмотреть другое направление. Запомнить координаты найденной точки в дополнительных переменных *imax* и *jmax*.

4. Сравнить соседнюю к исходной точку с точкой в запомненных координатах. Если ее значение больше, запомнить координаты этой точки.

5. Прodelать шаг 4 для оставшихся направлений.

6. В конце метода вернуть запомненные координаты и отрицательный результат окончания поиска.

Листинг 1.112

```
Max findMax(int[,] pole,int i_beg, int j_beg, int i_kon, int j_kon) {  
    //конец пути в окрестной точке?  
    if (i_beg - 1 >= 0) if (i_beg - 1 == i_kon && j_beg == j_kon)  
return new Max(i_beg - 1, j_beg, true);  
//аналогично для i_beg + 1, j_beg + 1, j_beg - 1  
    //нам есть куда идти в принципе?  
    bool cm = false;  
    if (i_beg - 1 >= 0) if (pole[i_beg - 1, j_beg] < 0) cm = true;  
    //аналогично для i_beg + 1, j_beg + 1, j_beg - 1  
    if (!cm) return new Max(-1, -1, false);  
//ищем следующую точку  
    int imax = i_beg - 1;  
    int jmax = j_beg;  
    if (imax < 0) imax = i_beg + 1;  
    if (pole[imax, jmax] >= 0)  
    { imax = i_beg + 1; jmax = j_beg;  
      if (imax > 9) imax = i_beg - 1;  
      if (pole[imax, jmax] >= 0)  
      { imax = i_beg;  
        jmax = j_beg - 1;  
      }  
    }
```

Окончание листинга 1.112

```

        if (jmax < 0) jmax = j_beg + 1;
        if (pole[imax, jmax] >= 0)
        {
            imax = i_beg;
            jmax = j_beg + 1;
            if (jmax > 9) jmax = j_beg - 1; } }
    if (i_beg + 1 < 10) {
        if (pole[imax, jmax] < pole[i_beg + 1, j_beg]) {
            if (pole[i_beg + 1, j_beg] < 0)
            {
                imax = i_beg + 1; jmax = j_beg;
            }
        }
        else if (i_beg + 1 == i_kon && j_beg == j_kon) return new Max(i_beg + 1,
j_beg, true);
    } }
//аналогично для j_beg + 1, j_beg - 1
    if (imax == i_kon && jmax == j_kon) return new Max(imax, jmax, true);
    return new Max(imax, jmax, false); }

```

Вспомогательные функции вам предлагается реализовать самостоятельно, здесь же в листинге 1.113 приведем код функции построения кратчайшего пути от начальной до конечной точки, оптимизировать который вам предлагается самостоятельно.

Листинг 1.113

```

public string Volna(int[,] pole, int i_beg, int j_beg, int i_kon, int j_kon){ bool brk = false;
pole[i_kon, j_kon] = 10; search(pole, i_beg, j_beg, 1);
Max s = findMax(pole, i_kon, j_kon, i_beg, j_beg);
if (s.i == -1){
for (int k1 = 0; k1 < pole.GetLength(0); k1++)
for (int k2 = 0; k2 < pole.GetLength(1); k2++){
if (pole[k1, k2] < 0 || pole[k1, k2] == 10) pole[k1, k2] = 0;        } return null; }
brk = false;
string s1 = i_kon + " " + j_kon + "!" + s.i + " " + s.j + "!";
if (s.result){ for (int k1 = 0; k1 < pole.GetLength(0); k1++)
for (int k2 = 0; k2 < pole.GetLength(1); k2++){

```

Окончание листинга 1.113

```
if (pole[k1, k2] < 0 || pole[k1, k2] == 10) pole[k1, k2] = 0;}
return s1; }
while (!brk){ s = findMax(pole,s.i, s.j, i_beg, j_beg);
    if (s.i == -1){ for (int k1 = 0; k1 < pole.GetLength(0); k1++)
        for (int k2 = 0; k2 < pole.GetLength(1); k2++){
            if (pole[k1, k2] < 0 || pole[k1, k2] == 10) pole[k1, k2] = 0;
        }return null; }
    s1 += s.i + " " + s.j + "!";
    brk = s.result;}
for (int k1 = 0; k1 < pole.GetLength(0); k1++)
    for (int k2 = 0; k2 < pole.GetLength(1); k2++){
        if (pole[k1, k2] < 0||pole[k1,k2]==10) pole[k1, k2] = 0;}
return s1;}
```

Внесем изменения в тестовое приложение, как показано в листинге 1.114, и запустим приложение. Если все сделано верно, то на начальных данных вы должны увидеть нечто похожее на левую форму, изображенную на рисунке 1.77. Правая форма на данном рисунке отображает случай отсутствия пути.

Листинг 1.114

```
private void button1_Click(object sender, EventArgs e){
    int[,] mas = new int[10, 10]; mas[0, 0] = 10; mas[1, 3] = 4;
    mas[3, 3] = 4; mas[4, 3] = 4; mas[1, 5] = 4; mas[1, 4] = 4;
    mas[7, 6] = 4; mas[8, 8] = 100;
    string s= Volna(mas, 8, 8, 0, 0);
    for (int i = 0; i < 10; i++) {
        for (int j = 0; j < 10; j++)
            textBox1.Text += mas[i, j].ToString().PadLeft(4) + " ";
        textBox1.Text += Environment.NewLine; }
    textBox1.Text += Environment.NewLine;
    if (s!=null) textBox1.Text += "Путь: "+s;
    else textBox1.Text += "Пути нет "};
```

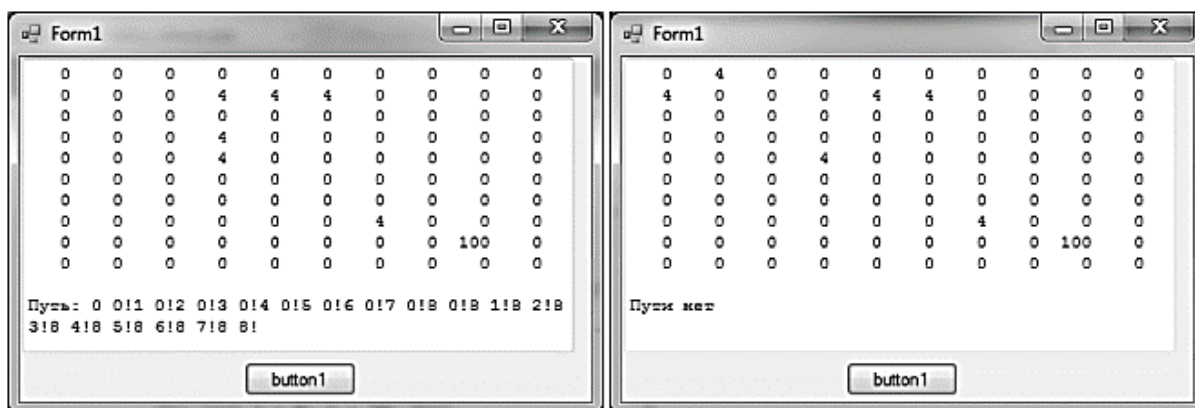


Рисунок 1.77. Тестирование построения пути

После того как мы отладили наш основной алгоритм, можно переходить к реализации остальных игровых функций, объединив их в отдельный модуль логики игры.

Разработка модуля логики игры

Как было сказано выше, отделение логики игры от ее графической части является обязательным, если мы хотим реализовать одну и ту же игру на разных графических технологиях. Однако простая разработка класса не решит проблему окончательно, ведь класс останется привязанным к проекту. Чтобы реализовать модуль, который можно будет использовать в разных решениях, познакомимся с понятием библиотеки.

Если мы посмотрим в папку bin-Debug любой из наших игр, то найдем там файл с расширением exe. Это исполняемый файл, который сразу можно запустить. То есть готовая функционирующая программа. Такой файл называется сборкой. Библиотека – проект типа ClassLibrary, который при компиляции выдает сборку с расширением dll. Ее нельзя запустить напрямую, но можно подключить к любому проекту, разрабатываемому в среде Visual Studio.

Примечание: если мы перейдем в окно свойств проекта через меню Project -> Имя_проекта Properties (рисунок 1.78), то увидим такое свойство, как целевая платформа (Target framework). У библио-

теки и у проекта, к которому она подключается, эти свойства должны быть одинаковыми.

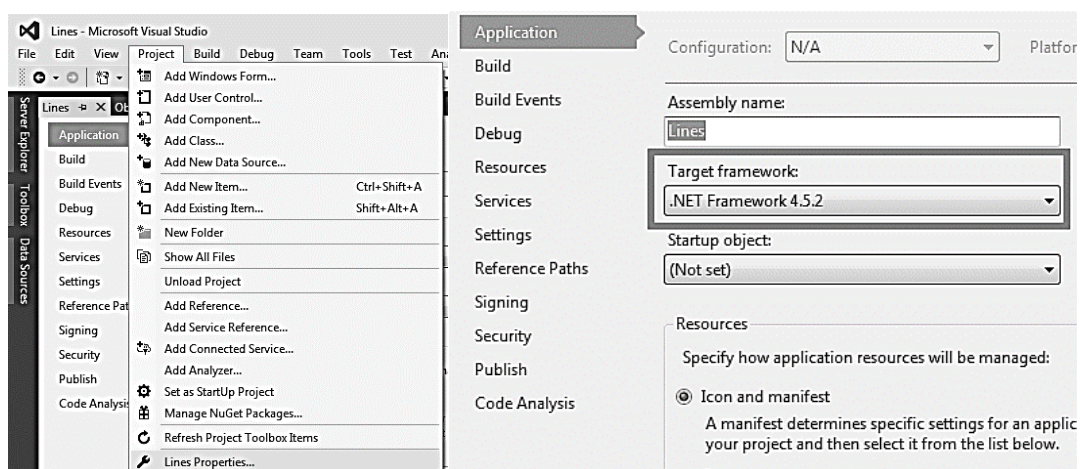


Рисунок 1.78. Переход к свойствам проекта

Библиотека создается также, как обычный exe-файл: как только мы написали и скомпилировали проект типа ClassLibrary, в папке с именем bin\Debug появился dll-файл. Для того чтобы подключить созданную библиотеку к проекту, необходимо в окне Solution Explorer щелкнуть на имени проекта правой кнопкой мыши и выбрать Add -> Reference (рисунок 1.79).

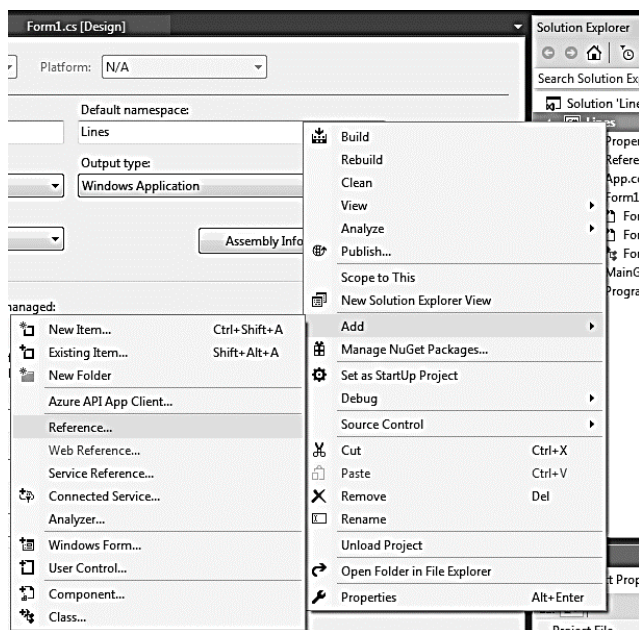


Рисунок 1.79. Вызов окна подключения библиотеки

Далее в открывшемся окне перейти на вкладку Browse, вызвать диалог выбора файла, нажав на кнопку Browse, затем найти нужную библиотеку и, отметив ее галочкой, нажать кнопку ОК (рисунок 1.80).

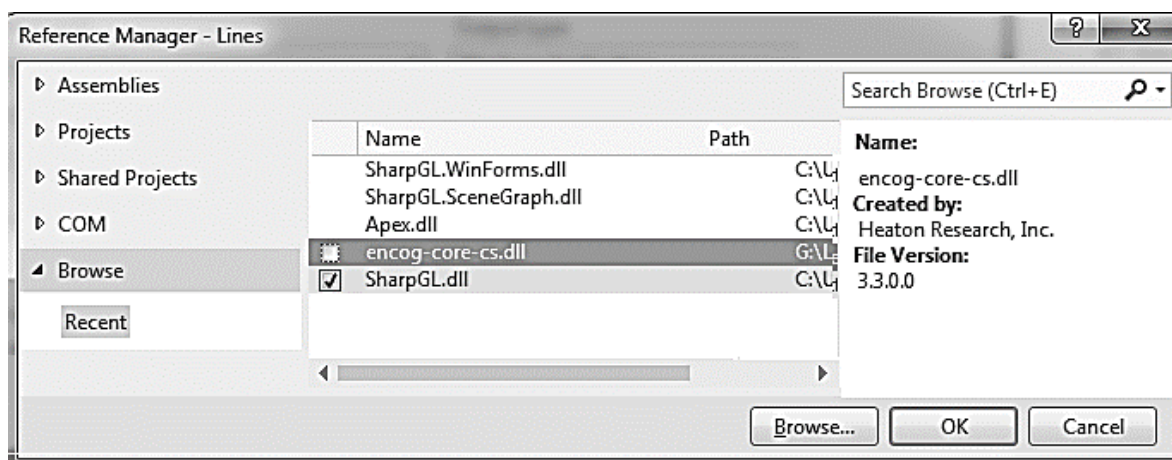


Рисунок 1.80. Подключение библиотеки

После этого необходимо в проекте, к которому подключена библиотека, прописать в самом верху файла с кодом using <Имя namespace из библиотеки>. Например, если в библиотеке у нас namespace Lines, то пишем using Lines; После этого с классами из библиотеки можно работать также, как с классами проекта, если, конечно, они объявлены с модификатором public.

Теперь перейдем непосредственно к разработке библиотеки. Создаем новый проект типа ClassLibrary, назвав его, например, LinesDll. В появившемся файле заменяем class Class1 на public class MainGame. Сюда же перенесем из нашего проекта тестирование волны класс Max (за пределы MainGame), а методы, относящиеся к волновому алгоритму, – в тело MainGame. Здесь же объявим двумерный массив с именем role. Обратите внимание, так как и массив, и методы волны находятся в одном классе, то из заголовков методов параметр-массив можно убрать: в нашем контексте ни с каким другим массивом они работать не будут.

Помимо массива нам нужно объявить генератор случайных чисел и целочисленное поле Score (для хранения очков), а также привя-

занное к нему свойство. Остановимся на последнем подробнее. Свойства класса – нечто среднее между полем и методом, представляет собой конструкцию вида (листинг 1.115).

Листинг 1.115

```
<Модификатор доступа> <Тип свойства> <Имя свойства>
{ get{return <значение>} set{<поле>=value} }
```

Обычно свойства связываются с закрытыми полями класса и помогают осуществить доступ к этим полям из внешних (относительно класса) частей программы. Свойства вместе с модификаторами доступа реализуют механизм защиты данных от несанкционированного доступа. Как мы видим, свойство имеет заголовок и тело. В заголовке указывается модификатор доступа (обычно public), тип возвращаемого свойством значения и имя свойства. В теле объявлено два метода get и set. Больше ничего в теле свойства объявлять нельзя. Метод get имеет ключевое слово return и возвращает какое-либо значение (обычно значение какого-либо поля, хотя не обязательно). Метод set имеет ключевое слово value и присваивает(устанавливает) это значение полю объекта.

Пример объявления свойства в классе MyClass и использования его в программе показаны в листинге 1.116.

Листинг 1.116

```
public class MyClass
{ int a; //закрытое поле
  public int A// свойство
  { get { return a;}
    set { a=value;}}}

//Пример использования описанного свойства в программе:
MyClass MyObj=new MyClass();
MyObj.A=6; // полю a объекта MyObj присвоится значение 6.
int b=MyObj.A; // переменной b присвоится значение поля a объекта MyObj.
```

В нашем же случае мы воспользуемся такой особенностью свойств, что в методе `set`, кроме обычного присвоения полю значения, можно использовать различные арифметические операции. Благодаря этому мы сможем реализовать различные алгоритмы подсчета очков. Например, мы хотим, чтобы за уничтожение линии из 5 шаров игроку начислялось 5 баллов, а каждый шарик сверх 5 приносил 2 очка. Тогда, если мы будем передавать в свойство количество шариков в удаляемой линии, то его код может быть следующим (листинг 1.117).

Листинг 1.117

```
public int _Scores{
    get { return Scores; }
    set { Scores += (value - 5) * 2 + 5; }}
```

Обратите внимание!

- *Свойства, как правило, называются так же, как и поля, только с каким-либо отличием. В нашем случае – перед `Score` стоит нижнее подчеркивание.*

Теперь объявим конструктор, в котором будет выделяться память под массив, а также в три случайные позиции помещаться шарики случайного типа. Реализовать это можно так, как показано в листинге 1.118.

Листинг 1.118

```
public MainGame()
{
    pole = new int[10, 10];
    for (int k = 0; k < 3; k++){ int i = r.Next(0, 10);
    int j = r.Next(0, 10);
    if (pole[i, j] == 0) pole[i, j] = r.Next(1, 6); else k--;
    }
}
```

Примечание: если мы захотим изменять размеры поля, то конструктор должен будет принимать целочисленный параметр, определяющий размер массива.

На основе конструктора объявим метод, помещающий на поле три новых шарика на каждом шаге. В его теле сначала посчитаем количество пустых клеток на поле, сохранив результат в переменную *c*. Если оно равно нулю – вернем значение ложь, говорящее, что шарики ставить некуда. Иначе повторим цикл из конструктора, только вместо 3 в условии *for* поставив выражение `Math.Min(3, c)`. Данная модификация сделана на случай, если пустых клеток останется меньше трех.

Теперь объявим метод удаления одноцветных линий. По логике игры он будет вызываться для каждого шарика, изменившего свое положение на доске (перемещенного игроком или сгенерированного компьютером). Поэтому в качестве параметров метод будет принимать координаты шарика. Далее необходимо проверить, не образует ли этот шарик вместе со стоящими по какому-либо направлению (горизонталь, вертикаль, диагональ \ , диагональ /) линию из 5 шариков такого же, как он цвета. Если образует – удаляем линию и начисляем очки. Отследить линию мы можем следующим образом: выбираем первое направление, например, горизонталь. От текущего шарика двигаемся вправо, пока не упремся в границу или не найдем шарик отличного от текущего цвета. Тогда запоминаем координаты последней подходящей точки как максимальное значение по горизонтали (самую правую подходящую точку непрерывной линии), возвращаемся в исходную точку и двигаемся от нее влево, по такому же принципу найдя минимальное значение по горизонтали (самую левую подходящую точку непрерывной линии). В процессе поиска, при нахождении каждого подходящего значения, увеличиваем длину линии на 1, а после нахождения самой левой и самой правой точки этой линии проверяем: если длина линии больше либо равна 5, то очищаем линию от самой левой до самой правой точки.

Ниже, в листинге 1.119, приведен пример кода, реализующего описанный алгоритм для горизонтали. Для прочих направлений вам предлагается написать его самостоятельно.

Листинг 1.119

```
public bool DelLine(int i,int j)
{   bool gor = true;   int c = pole[i, j];   int tmpi = i;   int count_line=1;
    while(gor) {
        tmpi++;
    if (tmpi >= 10) { gor = false; tmpi--; }      else
    if (pole[tmpi, j] == c) { count_line++; }
    else { gor = false; tmpi--; }}
    int maxgor = tmpi; gor = true;tmpi = i;
    while (gor){   tmpi--;
        if (tmpi < 0) { gor = false; tmpi++; }   else
        if (pole[tmpi, j] == c) { count_line++; }
    else { gor = false; tmpi++; }}
    int mingor = tmpi;
    if (count_line >= 5){
    for (int k = Math.Max(0, mingor); k <= Math.Min(9, maxgor); k++)
        pole[k, j] = 0;
    _Scores = count_line;
    return true;} else count_line = 1; }
```

Теперь объявим несколько вспомогательных конструкций, которые упростят нам работу при привязке логики к пользовательскому интерфейсу. Во-первых, метод, возвращающий количество шаров на доске (листинг 1.120).

Листинг 1.120

```
public int CountShar(){
int c = 0;
for (int k1 = 0; k1 < pole.GetLength(0); k1++)
    for (int k2 = 0; k2 < pole.GetLength(1); k2++)
        { if (pole[k1, k2] != 0) c++; }
return c;}
```

Во-вторых, метод, возвращающий количество строк или количество столбцов массива-поля (листинг 1.121).

Листинг 1.121

```
public int GetLength(int i){return pole.GetLength(i);}
```

В-третьих, специальную конструкцию, упрощающую доступ к массиву класса из основной программы: индексатор. Это аналог свойства, но для переменных-массивов. Поговорим о нем подробнее.

Индексатор объявляется как свойство за исключением двух вещей. Во-первых, вместо имени он содержит специальное ключевое слово `this`. Во-вторых, после слова `this` в квадратных скобках перечисляются параметры-индексы, используемые для доступа к массиву. В программе работа с индексаторами осуществляется через имя объекта и квадратных скобочек, в которых указывается индекс. Приведем пример объявления индексаторов и пример их использования (листинг 1.122).

Листинг 1.122

```
class A{ int[] arr = new int[2]; int[,] arr2 = new int[2,2];
public int this[int ind1]{
    get { return arr[ind1]; }
    set { arr[ind1] = value; } }
public int this[int ind1,int ind2]
{ get { return arr2[ind1,ind2]; }
  set { arr2[ind1,ind2] = value; } } }

// использование:
A a=new A();
a[0]=0; //работает метод set первого индексатора
string s=a[0,0].ToString(); //работает метод get второго //индексатора.
```

Если бы у нас были просто открытые массивы (объявленные с модификатором `public`), то доступ к ним без индексатора осуществлялся бы как показано в листинге 1.123.

Листинг 1.123

```
A a=new A(); a.arr[0]=0;
```

Как видите, индексаторы позволяют сократить код. Для нашего класса индексатор будет иметь вид, показанный в листинге 1.124.

Листинг 1.124

```
public int this[int i, int j]{get { return pole[i, j]; }}
```

В итоге ваша библиотека должна содержать следующее (рисунок 1.81).

```
namespace LinesDll
{
    18 references
    class Max[...]
    1 reference
    public class MainGame
    {
        int[,] pole;
        int Scores = 0;
        4 references
        public int _Scores[...]
        Random r = new Random();
        0 references
        public int this[int i, int j][...]
        0 references
        public int GetLength(int i)[...]
        0 references
        public MainGame()[...]
        2 references
        Max findMax(int i_beg, int j_beg, int i_kon, int j_kon)[...]
        5 references
        int search2(int x1, int y1, int i)[...]
        0 references
        public string Volna(int i_beg, int j_beg, int i_kon, int j_kon)[...]
        0 references
        public int CountShar()[...]
        0 references
        public bool NextStep(int count)[...]
        1 reference
        public bool Deline(int i, int j)[...]
    }
}
```

Рисунок 1.81. Краткое представление библиотеки

Разработать интерфейсную часть и связать ее с логикой вам предлагается самостоятельно. В качестве подсказок приведем вид итоговой игры (рисунок 1.82) и код реализации анимации (листинг 1.125).

Листинг 1.125

```
private void timer1_Tick(object sender, EventArgs e){
string[] s1 = s.Split(new char[] { '|' }, StringSplitOptions.RemoveEmptyEntries);
if (cc == -1) cc = s1.Length - 1; else { cc--; } if (cc >= 0){ string[] s2 = s1[cc].Split(' ');
if (cc == s1.Length - 1) { i_beg = Convert.ToInt32(s2[0]); j_beg = Convert.ToInt32(s2[1]);}
else { int i_tek = Convert.ToInt32(s2[0]); int j_tek = Convert.ToInt32(s2[1]);
m[i_tek, j_tek] = m[i_beg, j_beg]; m[i_beg, j_beg] = 0; i_beg = i_tek; j_beg = j_tek;
Draw();} if (cc== -1) { timer1.Enabled = false; if (!m.DelLine(i_beg, j_beg))
m.NextStep(3); Draw(); } } else { timer1.Enabled = false; if (!m.DelLine(i_beg,
j_beg)) m.NextStep(3); Draw();}}
```

Итоговый вид игры представлен на рисунке 1.82. Как можно видеть из этого рисунка, на форме вам необходимо разместить button, pictureBox, два элемента label, а также для организации анимации вам потребуется таймер.

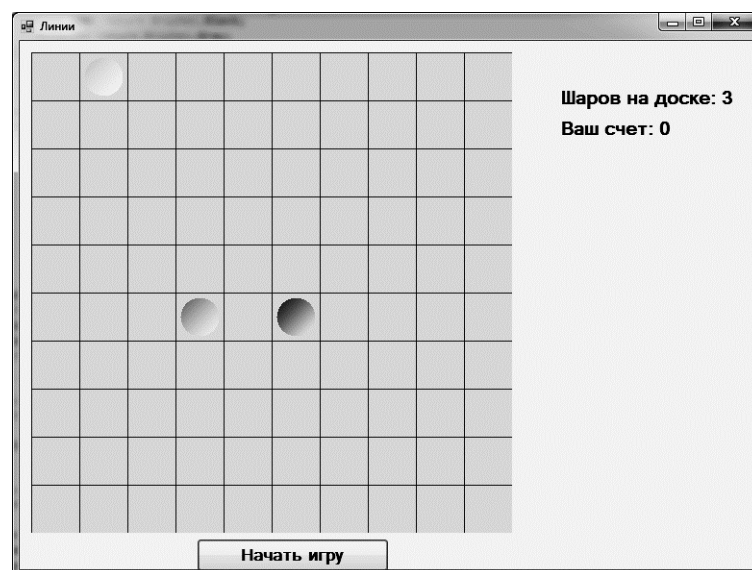


Рисунок 1.82. Игра «Линии»

В коде листинга 1.125 s – глобальная переменная, хранящая возвращенный методом Volna путь. Глобальная переменная cc хранит номер текущих координат в пути, которые в данный момент обрабатываются методом реализации анимации (куда перемещается шарик).

Обратите внимание!

- *Метод `Volna` возвращает путь от пустой клетки к шарiku, поэтому строку нужно обрабатывать с конца.*

Мы рассмотрели основные конструкции, методы и алгоритмы, необходимые для реализации игр, а также научились разрабатывать их, используя технологию GDI+. Теперь можно перейти к изучению различных инструментов и методов реализации графического интерфейса для игровых приложений. Однако перед этим вам предлагается выполнить несколько заданий.

ЗАДАНИЯ К ТЕМЕ 1.11

1. Используя технологию GDI+, реализуйте игру «Линии», поместив ее логику в библиотеку.
2. Оптимизируйте код игры «Линии» и опишите в словесной форме алгоритм реализации анимации.
3. Разработайте уровни сложности в игре «Линии». Например, на простом уровне сложности показываются следующие шары и места, куда они будут выведены. На среднем – просто показываются следующие шары без указания мест вывода. На сложном – следующие шары не показываются и есть ограничение по времени. Кроме того, модифицируйте код так, чтобы можно было выбирать размеры поля и количество типов шариков.
4. Реализуйте разные алгоритмы подсчета очков, а также хранение статистики в игре «Линии».
5. Придумайте игру, где бы потребовался волновой алгоритм. Реализуйте ее, используя технологию GDI+.

Часть 2. Разработка игр на WPF

В данной части рассматривается работа с технологией WPF с привязкой к разработке игр, реализованных в первой части. Необходимо отметить, что для разработки игр WPF практически не применяется, однако, как мы убедимся, ее возможности вполне позволяют это делать.

ТЕМА 2.1. ВВЕДЕНИЕ В WPF

В предыдущей части мы рассмотрели возможности технологии GDI+ для создания игровых интерфейсов. Однако, как вы могли бы заметить, созданные приложения в плане графики выглядят весьма бедно. Поэтому в этой и последующих частях книги мы рассмотрим возможности других технологий работы с графикой и построения интерфейсов. Начнем с переноса разработанной нами игры «Линии» с WindowsForms и GDI+ на WPF, предварительно ознакомившись с базовыми возможностями этой технологии.

Согласно Википедии [1]: «Windows Presentation Foundation (WPF) — это графическая (презентационная) подсистема платформы .NET Framework, предназначенная для разработки приложений Windows. Данная технология обладает обширными возможностями для создания визуально привлекательных интерфейсов. С помощью WPF можно создавать как настольные, так и запускаемые в браузере приложения.

В основе WPF лежит векторная система визуализации, созданная с учетом возможностей современного графического оборудования и не зависящая от разрешения устройства вывода. Графической технологией, лежащей в основе WPF, является DirectX, что существенно повышает производительность за счет использования аппаратного ускорения графики».

Можно сказать, что приложение, созданное в WPF, состоит из двух частей: визуальная часть, описанная на XAML, и функциональная часть, выполненная на любом .NET-совместимом языке. Это может быть C#, VB, C++, Ruby и многие другие. С помощью языка XAML описывается внешний вид приложения, а также осуществляются привязки программного кода. Он представляет собой язык декларативного описания интерфейса, основанный на XML. Напомним некоторые моменты синтаксиса языков этого вида.

Так же как и в языке XML, в языке XAML основной элемент – это тэг. Тэги бывают открывающие и закрывающие. Пример открывающего тэга и закрывающего к нему приведен в листинге 2.1.

Листинг 2.1

```
<Grid> </Grid>
```

Между открывающим и закрывающим тэгами обычно помещается содержимое элемента. Если же содержимого нет, то закрывающий тэг можно не писать, поставив слеш перед закрывающей скобкой открывающего тэга. У открывающего тэга могут быть параметры со значениями. Параметры тэга пишутся через пробел после его имени, а значения каждого параметра – в кавычках через знак равно. Пример тэга без содержимого, но с параметрами приведен в листинге 2.2.

Листинг 2.2

```
<Button Content="Button" Height="23" HorizontalAlignment="Left" Margin="12,23,0,0" Name="button1" VerticalAlignment="Top" Width="75" Click="button1_Click" />
```

Для комментирования в языке XAML используется следующая комбинация (листинг 2.3).

Листинг 2.3

```
<!-- Закомментированный участок текста -->
```

Таким образом, при использовании языка C#, WPF-приложение состоит из файла с расширением *.xaml, описывающего интерфейс (на языке XAML), и файла с расширением *.cs, описывающего собственно работу приложения.

Рассмотрим для начала реализацию жизненно важных моментов, таких как:

- структуру и взаимосвязь файлов в WPF-приложении;
- определение главного окна приложения;

- добавление новых окон в проект;
- добавление элементов в саму форму;
- изменение параметров формы и элементов;
- привязку реакций на события.

Для создания WPF-приложения мы выбираем тип проекта WPFApplication, как показано на рисунке 2.1.

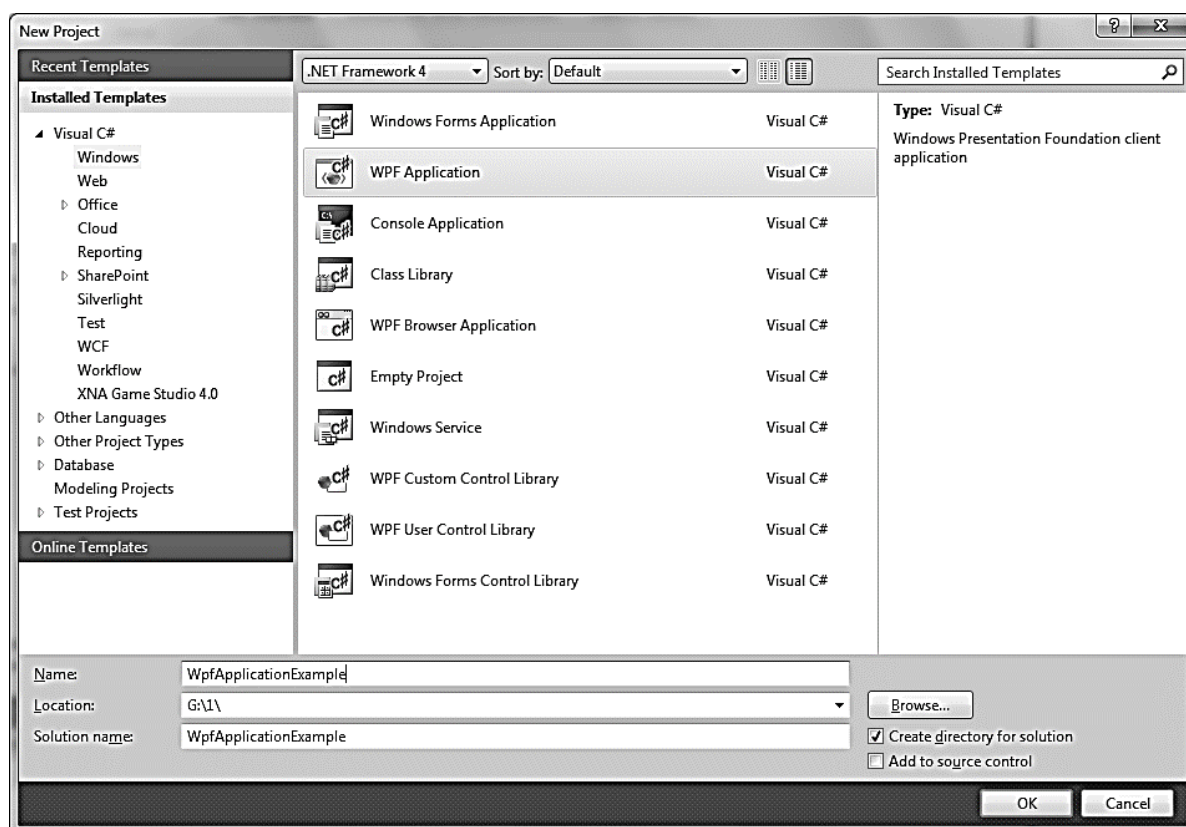


Рисунок 2.1 Создание WPF-приложения

В итоге мы получили следующие файлы: MainWindow.xaml и MainWindow.xaml.cs, а также App.xaml и App.xaml.cs. В первом содержится код, представленный в листинге 2.4.

Листинг 2.4

```
<Window x:Class="WpfApplicationExample.MainWindow"
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/
presentation"
```

Окончание листинга 2.4

```
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
Title="MainWindow" Height="350" Width="525">
<Grid>
</Grid>
</Window>
```

Строка `x:Class="WpfApplicationExample.MainWindow"` содержит полное имя класса, описывающего основное окно приложения. То, что это окно основное, мы видим из файла `App.xaml`, описывающего класс, управляющий нашим приложением, и содержащего код, приведенный в листинге 2.5.

Листинг 2.5

```
<Application x:Class="WpfApplicationExample.App"
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/
presentation"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
StartupUri="MainWindow.xaml">

<Application.Resources>

</Application.Resources>

</Application>
```

Указание имени основного окна содержится в параметре `StartupUri`. Если мы хотим сделать основным другое окно, то выполняем следующие действия. Во-первых, добавляем новое окно в проект. Для этого в окне `Solution Explorer` правой кнопкой мыши нажимаем на имя проекта, выбираем в контекстном меню пункт `Add New Item` и в появившемся окне (рисунок 2.2) выбираем `Window (WPF)`. Теперь, если в файле `App.xaml` мы изменим код следующим образом (листинг 2.6), то при запуске приложения запустится именно это окно.

Листинг 2.6

```
<Application x:Class="WpfApplicationExample.App"
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/
presentation"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
StartupUri=" Window2.xaml">

<Application.Resources>
</Application.Resources>

</Application>
```

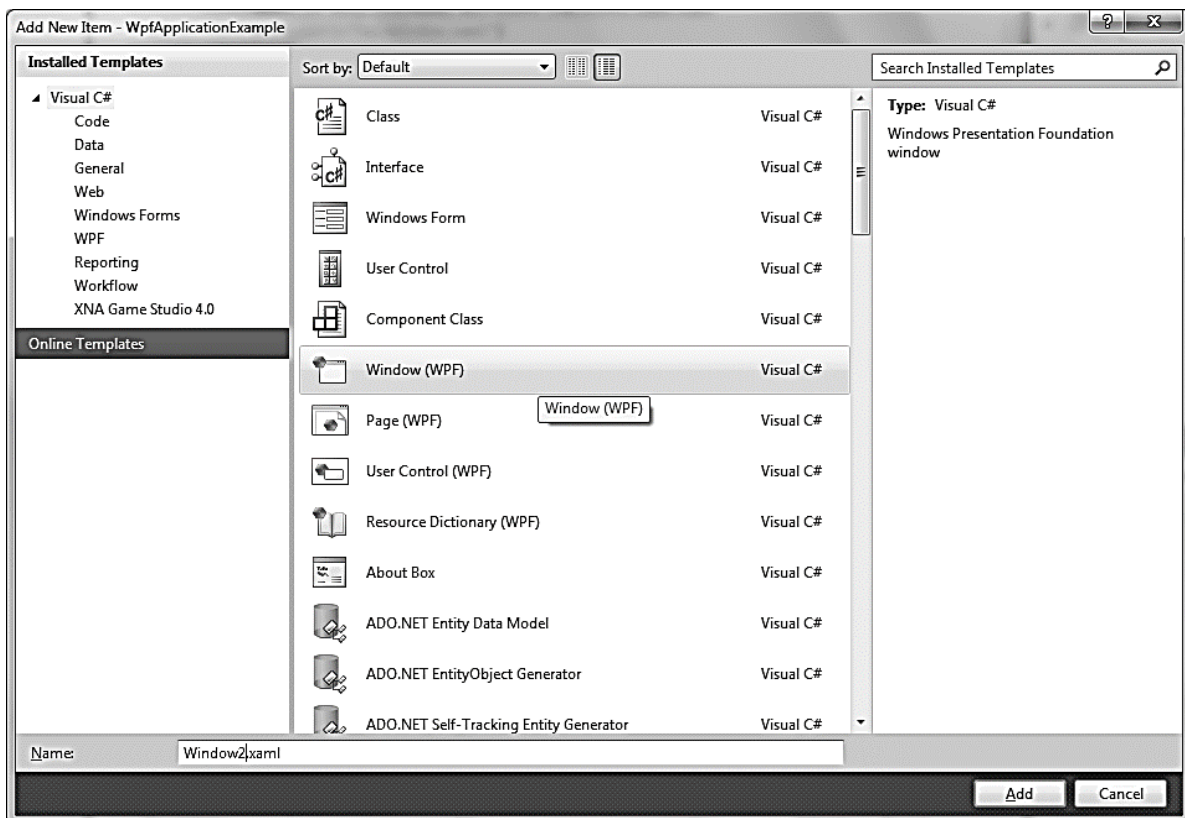


Рисунок 2.2 Добавление нового окна в проект

При работе с приложениями WPF проектирование интерфейса может выполняться не только с помощью стандартных методов (используя панель ToolBox и окошко Properties), но и путем прямого изменения xaml-файла. Например, для изменения заголовка окна и его

размеров достаточно изменить значение соответствующих параметров (листинг 2.7).

Листинг 2.7

```
<Window x:Class="WpfApplicationExample.MainWindow"
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/
presentation"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
Title="Линии" Height="650" Width="650" MinHeight="200" MinWidth="250"
MaxHeight="800" MaxWidth="800">
<Grid> </Grid>
</Window>
```

Привязка методов-обработчиков событий может выполняться также, как в обычном приложении WinForms: либо через вкладку Events в окошке Properties, либо двойным щелчком на требуемом элементе. Сам метод в таком случае появляется в соответствующем файле с расширением *.cs. Кроме того, подписку на события мы можем реализовать через изменение xaml документа. При добавлении кнопки через ToolBox и привязки события через Properties в xaml-файле появился код, представленный в листинге 2.8.

Листинг 2.8

```
<Button Content="Button" Height="23" HorizontalAlignment="Left" Margin="12,23,0,0" Name="button1" VerticalAlignment="Top" Width="75" Click="button1_Click" />
```

Если мы в файле с расширением *.cs определим метод button2_Click и добавим в xaml-файл код из листинга 2.9, то новый метод (button2_Click) автоматически привяжется ко второй кнопке.

Листинг 2.9

```
<Button Content="Button two" Height="23" HorizontalAlignment="Left" Margin="102,23,0,0" Name="button2" VerticalAlignment="Top" Width="75" Click="button2_Click" />
```

Если мы хотим, чтобы при нажатии на первую кнопку открывалась вторая форма, то код обработки события должен быть следующим (листинг 2.10).

Листинг 2.10

```
private void button1_Click(object sender, RoutedEventArgs e){  
    Window2 w = new Window2(); w.ShowDialog();}
```

Обратите внимание!

- *При привязке события через xaml-код, метод-обработчик должен быть определен заранее, так как автоматически не генерируется.*

Помимо жесткого отделения кода от определения интерфейса, в приложениях WPF очень легко реализуется идея разделения внешнего вида элементов от их функциональности, а также унификации оформления. Например, если мы хотим, чтобы все кнопки нашего приложения имели одинаковый размер, выравнивание, шрифт и фон, то это реализуется написанием следующего кода (листинг 2.11).

Листинг 2.11

```
<Window.Resources>  
<Style TargetType="{x:Type Button}">  
    <Setter Property="FontSize" Value="14.0"/>  
    <Setter Property="Background" Value="#FFEF1818"/>  
    <Setter Property="Height" Value="23"/>  
    <Setter Property="HorizontalAlignment" Value="Left"/>  
    <Setter Property="VerticalAlignment" Value="Top" />  
    <Setter Property="Width" Value="75"/>  
</Style>  
</Window.Resources>
```

Как мы видим, тэг Setter имеет два параметра – собственно свойство и устанавливаемое значение.

Что касается различных визуальных эффектов, то WPF предоставляет для этого множество возможностей. Например, с использованием триггеров можно привязать эффекты к определенным действиям пользователя.

В следующем примере (листинг 2.12) вид элемента Button меняется, когда пользователь наводит на него указатель мыши. Вид ожидаемого действия пользователя задает значение «IsMouseOver».

Листинг 2.12

```
<Style TargetType="{x:Type Button}">

<Setter Property="FontSize" Value="14.0"/>
<Setter Property="Background" Value="#FFEF1818"/>
<Setter Property="Height" Value="23"/>
<Setter Property="HorizontalAlignment" Value="Left"/>
<Setter Property="VerticalAlignment" Value="Top" />
<Setter Property="Width" Value="45"/>

<Style.Triggers>

<Trigger Property="IsMouseOver" Value="True">
  <Setter Property="Button.FontWeight" Value="Bold" />
  <Setter Property="Button.FontSize" Value="14.0" />
  <Setter Property="Button.Background" Value="LightBlue" />
</Trigger>

</Style.Triggers >

</Style>
```

Обычно структура xaml-документа следующая: корневой элемент документа описывает основное окно. В его параметрах, кроме таких полезных свойств как заголовок окна, размещается имя класса, описывающего это окно, а также ссылка на текущее пространство имен. Далее идет элемент, который содержит описания ресурсов окна. К таким ресурсам относятся, например, стили для каждого типа

элементов. Внутри описания стилей содержится описание установки значений свойств, а также описание триггеров, отвечающих за визуальные эффекты. После ресурсов окна располагается блок привязки команд, а за ним блоки, группирующие элементы на форме. Остановимся на последнем подробнее.

Один из компоновочных элементов, облегчающих проектирование интерфейса, локализацию приложения и обработку динамического содержимого, – Grid. Кроме уже упомянутого в WPF используются DockPanel, Uniform Grid, StackPanel, WrapPanel и Canvas. Все они различаются между собой способами компоновки элементов. Например, Grid по сути представляет собой табличку с ячейками под каждый элемент. Причем иногда элемент может занимать несколько смежных ячеек. И если здесь ячейки могут быть разного размера, то в Uniform Grid все ячейки имеют одинаковый размер. В DockPanel элементы могут прикрепляться к одной из четырех его сторон, а StackPanel упорядочивает элементы друг за другом либо по горизонтали, либо по вертикали. WrapPanel работает также, как StackPanel, но при достижении конца ряда или колонки выполняется циклический переход в начало. Элемент Canvas же дает полный простор действий, не имея собственных средств компоновки. Указанные элементы могут вкладываться друг в друга.

К примеру, рассмотрим начало создания интерфейса для игры «Линии». Допустим, мы хотим, чтобы слева располагалась кнопка, под ней – две надписи, а справа было игровое поле. Реализующий это код будет следующим (листинг 2.13).

Листинг 2.13

```
<Window x:Class="WpfLines.MainWindow"
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/
presentation"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
```

Окончание листинга 2.13

```
xmlns:local="clr-namespace:WpfLines"
xmlns:sys="clr-namespace:System;assembly=mscorlib"
mc:Ignorable="d"
Title="Линии" Height="650" Width="650" MinHeight="200" MinWidth="250"
MaxHeight="800" MaxWidth="800">
<Window.Resources>
<Style x:Key="MyTextStyle">
<Setter Property="Control.FontFamily" Value="Calibri"></Setter>
<Setter Property="Control.FontSize" Value="18"></Setter>
<Setter Property="Control.FontWeight" Value="Bold"></Setter>
<Setter Property="Control.Padding" Value="5"></Setter>
<Setter Property="Control.Margin" Value="5"></Setter>
</Style> </Window.Resources>
<Grid> <Grid.ColumnDefinitions>
<ColumnDefinition Width="200"/>
<ColumnDefinition Width="*/>
</Grid.ColumnDefinitions>
<StackPanel Background="AliceBlue">
<Button Click="Button_Click" Style="{StaticResource
MyTextStyle}">Начать</Button>
<Label Name="Scores" Style="{StaticResource MyTextStyle}">Очки:</Label>
<Label Name="Count" Style="{StaticResource MyTextStyle}">Количество
шаров:</Label>
</StackPanel>
</Grid> </Window>
```

В итоге визуальное отображение формы будет следующим (рисунок 2.3).

Посмотрите внимательно на тэг `<Grid.ColumnDefinitions>`, а также на параметры высоты-ширины окна. Благодаря параметрам, мы определили максимальный и минимальный размеры окна, а благодаря `ColumnDefinition Width`, указали, что под кнопку и текст при любом размере окна будет выделяться размер в 200 пикселей. В итоге минимальный размер формы будет иметь вид, как показано ниже на рисунке 2.4.

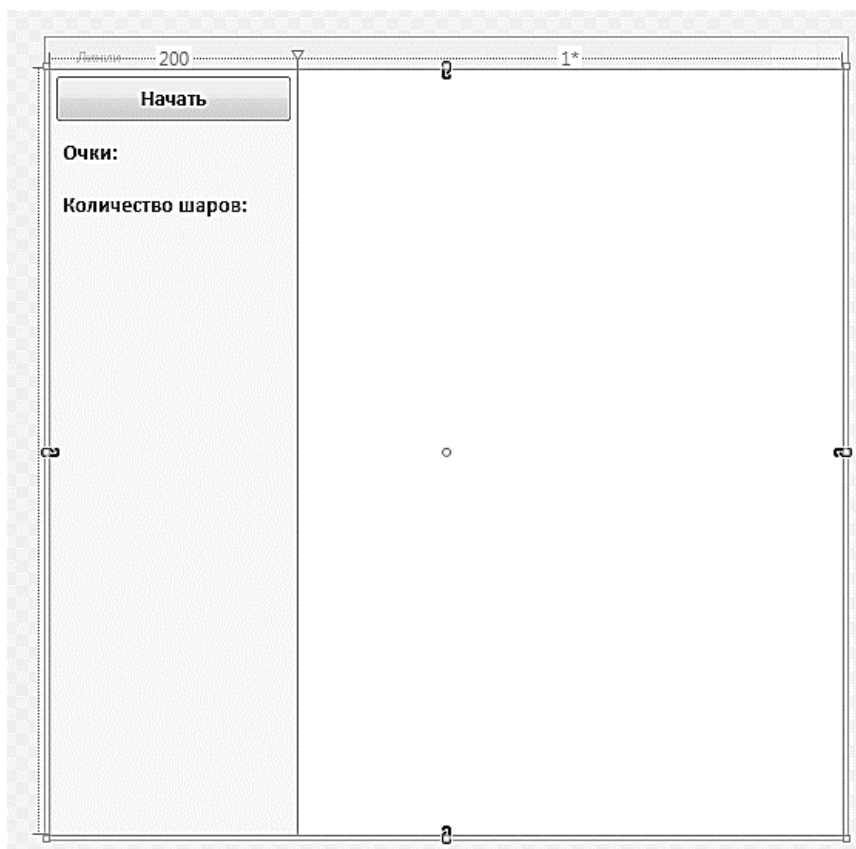


Рисунок 2.3. Основное окно

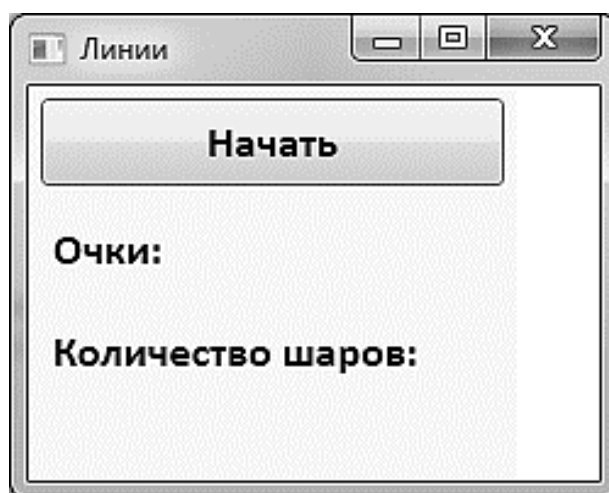


Рисунок 2.4. Минимальный размер окна приложения

Как вы могли заметить, в WPF используются знакомые нам визуальные элементы: Label, Textbox, Button. Помимо прямого объявления их в XAML-коде вы можете также перетаскивать их на форму с ToolBox, находясь в режиме дизайнера.


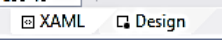
Мы рассмотрели основы WPF и теперь можем переходить к более сложным конструкциям данной технологии. Однако перед этим вам предлагается выполнить простое задание для закрепления полученных знаний.


ЗАДАНИЕ К ТЕМЕ 2.1

Реализуйте игру «Угадай число» на WPF.

ТЕМА 2.2. ГРАФИКА, КОНВЕРТОРЫ И СВОЙСТВА ЗАВИСИМОСТИ

Прежде чем перейти к разработке игры «Линии», нам необходимо познакомиться с двумя важными понятиями: конверторы типов и свойства зависимости, а также рассмотреть работу с графикой в WPF как минимум на уровне отрисовки нужных нам примитивов.

Но для начала организуем более удобное расположение окон на экране. По умолчанию, когда вы создаете новый WPF-проект, окно дизайнера и окно кода делят ваш экран пополам, что не очень удобно для больших проектов, поэтому лучше, чтобы окна разворачивались полностью. Реализовать это можно, нажав кнопку «Expand Pane» в нижней части окна (самая правая): . Тогда переключаться между кодом и внешним видом окна можно будет с помощью ярлыков-вкладок, расположенных в нижней части: . Первые две кнопки (рядом с «Expand») отвечают за способ деления экрана окнами дизайнера и кода. По умолчанию он горизонтальный, поэтому вкладки XAML и Design будут располагаться внизу слева. Если же способ будет вертикальным, то вкладки расположатся справа

сверху по вертикали и будут выглядеть так:  .

Начнем с простого. Рассмотрим, как, используя WPF, можно отрисовать эллипсы и прямоугольники. Для рисования будем использовать компоновочный элемент Canvas. С помощью него отобразить эллипсы можно разными способами. Первый – напрямую через XAML. В этом случае для вывода эллипса можно использовать тэг `Ellipse`, а для заливки – `Ellipse.Fill`. В качестве содержимого последнего тэга указываем кисть, определяющую тип и цвет заливки. Код приложения, выводящего эллипс – листинг 2.14.

Листинг 2.14

```
<Window x:Class="Grafika.MainWindow"
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/
presentation"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
xmlns:local="clr-namespace:Grafika"
mc:Ignorable="d"
Title="Отрисовка" Height="350" Width="525">
<Canvas x:Name="BaseCanvas" Grid.Column="1" Background="WhiteSmoke" >
<Ellipse StrokeThickness="3" Width="50" Height="50">
<Ellipse.Fill>
<SolidColorBrush Color="Red"/>
</Ellipse.Fill> <Ellipse.Stroke>
<SolidColorBrush Color="LightCyan"/>
</Ellipse.Stroke> </Ellipse> </Canvas>
</Window>
```

Результат работы этого кода представлен на рисунке 2.5а. Если же мы хотим отрисовать прямоугольник, то вместо тэга `Ellipse` используем тэг `Rectangle`. Кроме того, в содержимом тэга `Stroke` изменим цвет с `LightCyan` на `Black`, изменив тем самым обводку фигуры. Результат – на рисунке 2.5б.

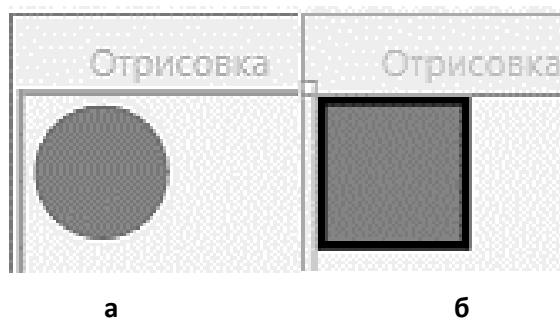


Рисунок 2.5. Эллипс и квадрат

Кроме SolidBrush мы можем использовать RadialGradientBrush. Код, реализующий это, приведен в листинге 2.15, а результат его работы – на рисунке 2.6.

Листинг 2.15

```
<Ellipse StrokeThickness="3" Width="50" Height="50">
<Ellipse.Fill>
<RadialGradientBrush GradientOrigin="0.2,0.2">
<GradientStop Color="White" Offset="0" />
<GradientStop Offset="0.8" Color="Red"/>
</RadialGradientBrush>
</Ellipse.Fill>
<Ellipse.Stroke>
<SolidColorBrush Color="Red"/>
</Ellipse.Stroke>
</Ellipse>
```

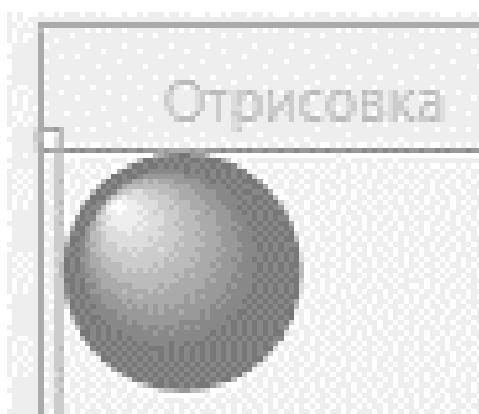


Рисунок 2.6. Градиентная заливка эллипса

Или же мы можем взять LinearGradientBrush, как показано в листинге 2.16. Результат работы – рисунок 2.7.

Листинг 2.16

```
<LinearGradientBrush>  
<GradientStop Color="White" Offset="0" />  
<GradientStop Offset="2" Color="Red"/>  
</LinearGradientBrush>
```

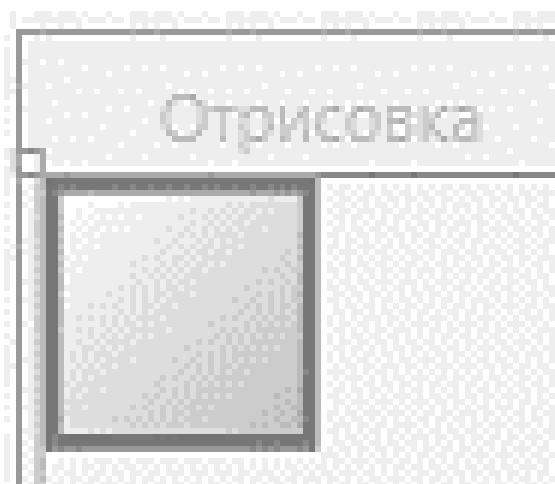


Рисунок 2.7. Градиентная заливка квадрата

Для рисования фигуры в определенных координатах указываем у тэга, ее определяющего, параметры Canvas.Left и Canvas.Top. Например, отобразим рядом эллипс и квадрат (рисунок 2.8). Реализующий это код приведен в листинге 2.17.

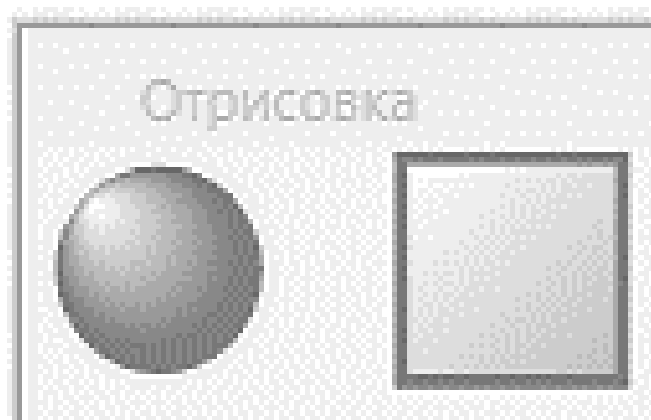


Рисунок 2.8. Эллипс и квадрат

Листинг 2.17

```
<Canvas x:Name="BaseCanvas" Grid.Column="1" Background="WhiteSmoke" >
<Rectangle StrokeThickness="3" Width="50" Height="50" Canvas.Left="75">
<Rectangle.Fill>
<LinearGradientBrush>
  <GradientStop Color="White" Offset="0" />
  <GradientStop Offset="2" Color="Red"/>
</LinearGradientBrush>
</Rectangle.Fill>
<Rectangle.Stroke>
  <SolidColorBrush Color="Red"/>
</Rectangle.Stroke>
</Rectangle>
<Ellipse StrokeThickness="3" Width="50" Height="50">
  <Ellipse.Fill>
    <RadialGradientBrush GradientOrigin="0.2,0.2">
      <GradientStop Color="White" Offset="0" />
      <GradientStop Offset="0.8" Color="Red"/>
    </RadialGradientBrush>
  </Ellipse.Fill>
  <Ellipse.Stroke>
    <SolidColorBrush Color="LightCyan"/>
  </Ellipse.Stroke>
</Ellipse>
</Canvas>
```

Так как у эллипса мы не задали параметры `Canvas.Left` и `Canvas.Top`, то по умолчанию он рисуется в левом верхнем углу. Обратите внимание и на сам способ задания положения объектов: так как `Canvas` является средством группировки и компоновки объектов, то их координаты определяются через его левый верхний угол. То есть по сути это отступы от него.

Второй способ работы с графикой, который мы рассмотрим, – написание собственного класса-наследника `Canvas`. Напомним, что наследование – это процесс, благодаря которому один объект может наследовать основные свойства другого объекта и добавлять к ним черты, характерные только для него. Наследование является важным, поскольку оно позволяет поддерживать концепцию иерархии классов.

Применение иерархии классов делает управляемыми большие потоки информации. Без использования иерархии классов для каждого объекта пришлось бы задать все характеристики, которые бы полностью его определяли. Однако при использовании наследования можно описать объект путем отделения общей части в так называемый базовый класс, а внутри нового класса (класса-наследника) описать лишь черты, делающие объект уникальным. Синтаксис наследования следующий: при описании класса-потомка его класс-предок указывается через двоеточие. Пример определения класса-предка А и класса-потомка В приведен в листинге 2.18.

Листинг 2.18

```
public class A {}  
public class B: A {}
```

В нашем случае мы определим класс MyBoardCanvas для того, чтобы внести изменения в метод OnPaint, вызывающийся при появлении элемента на экране. Допустим, мы хотим, чтобы в окне отобразилась сеточка. Тогда код будет таким, как показано в листинге 2.19. Результат работы этого кода представлен на рисунке 2.9.

Листинг 2.19

```
class MyBoardCanvas : Canvas{  
protected override void OnRender(DrawingContext dc){  
base.OnRender(dc);  
  
Pen pen = new Pen(new SolidColorBrush(Colors.Gray), 1);  
  
for (double y = 0; y <= this.ActualHeight; y += 10){  
dc.DrawLine(pen, new Point(0d, y), new Point(this.ActualWidth, y));  
}  
for (double x = 0; x <= this.ActualWidth; x += 10) {  
dc.DrawLine(pen, new Point(x, 0d), new Point(x, this.ActualHeight));  
}}}
```

Обратите внимание!

- Слово *override* указывает, что данный метод расширяется в классе потомке.
- Строка кода *base.OnRender(dc)* заставляет отработать общую для *Canvas* часть метода, а далее идет код, специфичный для нашего потомка.



Рисунок 2.9. Сеточка на форме

Если же мы хотим отрисовать, к примеру, несколько эллипсов, расположенных по диагонали, код может быть следующим (листинг 2.20). Результат его работы представлен на рисунке 2.10.

Листинг 2.20

```
Pen pen = new Pen(new SolidColorBrush(Colors.Gray), 1);  
for (double y = 0; y <= this.ActualHeight; y += 10) {  
    dc.DrawEllipse(Brushes.Black, pen, new Point(y, y), 10, 10));}
```

Как видите, здесь довольно много похожего с GDI+. Единственно, в данном случае точка `new Point(y, y)` указывает координаты центра эллипса, а не координаты его левого верхнего угла.

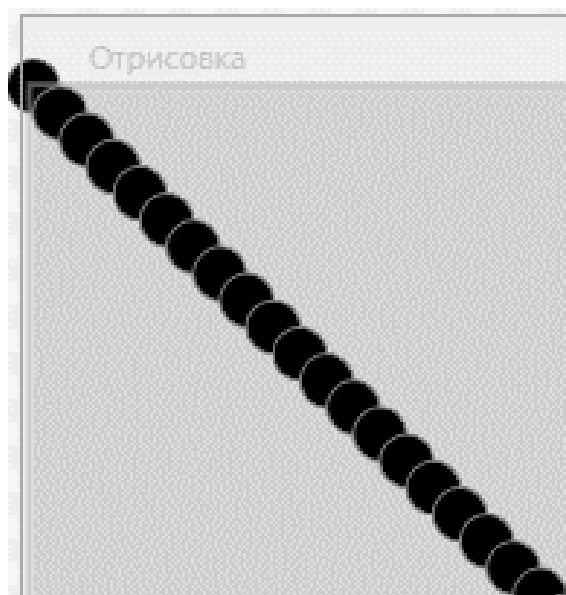


Рисунок 2.10. Эллипсы

Однако простая работа с графикой здесь не столь важна. Гораздо интереснее рассмотреть возможности WPF в плане зависимости отрисовки объектов от свойств, определенных в программном коде и изменяемых пользователем. «Изюминкой» WPF является расширение функциональности свойств объектов, вычисление значений свойств на основе разнообразных входных данных и автоматическую реакцию интерфейса на изменение свойств. Реализуется это за счет привязок данных и так называемых свойств зависимости.

Для простоты понимания рассмотрим их применение для решения конкретной задачи. Допустим, мы хотим, чтобы наше поле всегда было квадратным и расчерченным сеточкой размером десять на десять клеток, независимо от изменения размера окна. На рисунке 2.11 показано одно и то же окно разных размеров. Поле же в обоих случаях остается квадратным.

Алгоритм реализации этого довольно прост: как только изменился размер окна, нужно изменить размеры нашего Canvas, установив в качестве и высоты, и ширины минимальное значение размера стороны окна (без заголовка). То есть, по сути, нам нужно связать размеры окна и размеры Canvas, выполнив преобразование в виде установления минимального размера.

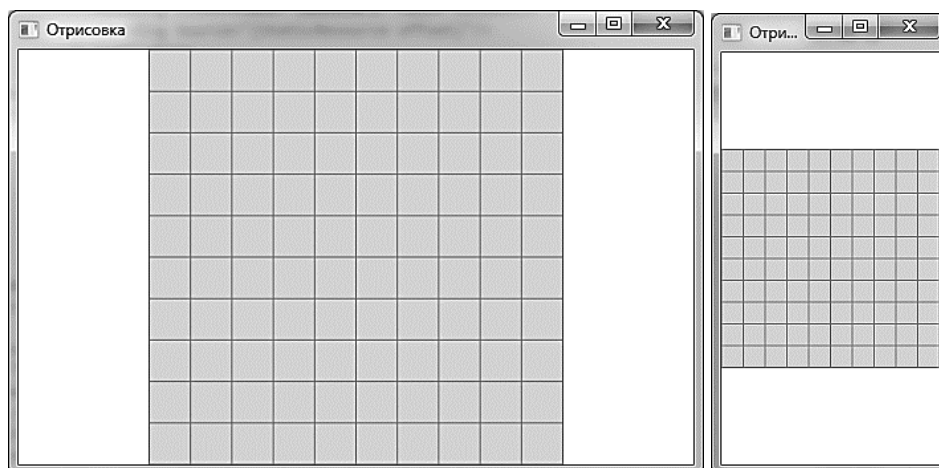


Рисунок 2.11. Два состояния окна

Привязки данных к интерфейсу осуществляются в XAML-файле с помощью тэгов Binding. Согласно официальной документации MSDN [5]: «Привязка данных является процессом, который устанавливает связь между пользовательским интерфейсом (UI) приложения и его бизнес-логикой. Если привязка имеет правильные параметры и данные предоставляют правильные уведомления, то при изменении значений данных в элементах, которые привязаны к данным, автоматически отражаются изменения. Привязка к данным может также означать, что, если внешнее представление данных в элементе изменяется, то базовые данные могут автоматически обновляться для отражения изменений. Например, если пользователь изменяет значение в элементе TextBox, базовое значение данных автоматически обновляется, чтобы отразить это изменение».

Синтаксис привязки показан в листинге 2.21.

Листинг 2.21

```
<тэг_свойства> <Binding ElementName="ИмяЭлементаСКоторымСвязываемся"
Path="СвойствоЭлементаСКоторымСвязываемся"/></тэг_свойства>
```

Если для привязки требуется выполнить некое преобразование значения, то определяется конвертер: класс, наследник интерфейса `IValueConverter`, в переопределенном методе `Convert` которого описы-

вается код преобразования значения. В тэг Binding добавляется параметр Converter, значение которого содержит ссылку на нужный класс.

Рассмотрим пример решения следующей задачи: мы хотим привязать размеры нашего Canvas к размерам окна так, чтобы высота Canvas была равна половине высоты окна.

Первое, что мы делаем, – присваиваем нашему окну имя, добавляя параметр Name в его свойства следующим образом, как показано в листинге 2.22.

Листинг 2.22

```
Name="BaseCanvas"
```

Затем добавляем в проект новый класс, назвав его, к примеру, MyBoardCoordConverter, а вверху прописываем подключение пространства имен System.Windows.Data.

Затем через двоеточие после имени класса пишем имя интерфейса, от которого будем его наследовать, а именно, IValueConverter. В отличие от классов, интерфейсы при наследовании требуют переопределения всех описанных в них методах, даже если они вам не нужны. Для того чтобы не запоминать все их названия и определения, можно воспользоваться помощью среды. Как только вы написали двоеточие и имя интерфейса, вы можете навести курсор на первую букву его имени и вызвать контекстное меню, нажав на стрелочку, как показано на рисунке 2.12.

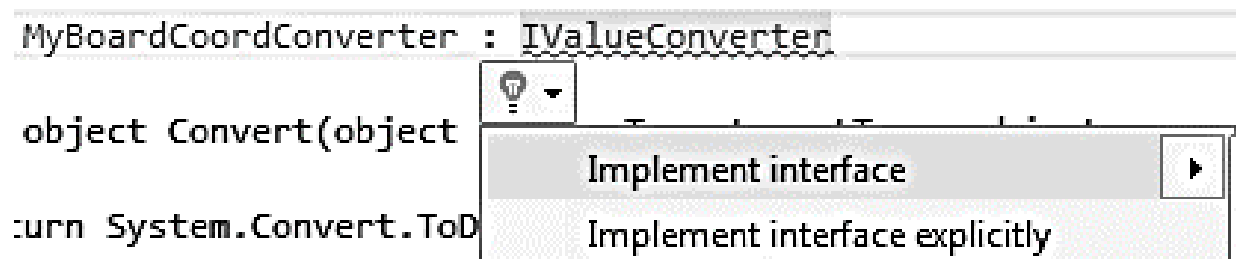


Рисунок 2.12. Реализация интерфейса

Visual Studio автоматически сгенерирует вам «заглушки» для всех методов интерфейса, позволив описать функциональность лишь требуемых. Нам нужен метод Convert, поэтому класс будет иметь вид, представленный в листинге 2.23.

Листинг 2.23

```
public class MyBoardCoordConverter: IValueConverter{
public object Convert(object value, Type targetType, object parameter, CultureInfo culture){
return System.Convert.ToDouble(value) / 2;
}
public object ConvertBack(object value, Type targetType, object parameter, CultureInfo culture) {throw new NotImplementedException();
}}
```

Обратите внимание!

- *Тип object – это универсальный тип C#. К нему можно преобразовать любой типа данных. Таким образом разрабатываются унифицированные методы, способные работать с любыми типами.*

Для использования конвертера в XAML-файле для привязок, его необходимо объявить в блоке ресурсов окна следующим образом (листинг 2.24)

Листинг 2.24

```
<Window.Resources>
  <local:MyBoardCoordConverter x:Key="MyBoardCoordConverter"/>
</Window.Resources>
```

Теперь код для Canvas с привязкой свойств будет иметь вид, показанный в листинге 2.25. Результат работы кода представлен на рисунке 2.13.

Листинг 2.25

```
<local:MyBoardCanvas x:Name="GameBoard" Background="LightGray"> <Canvas.Height>  
  <Binding ElementName="BaseCanvas" Path="Height" Converter="{StaticResource  
MyBoardCoordConverter}"/> </Canvas.Height> </local:MyBoardCanvas>
```

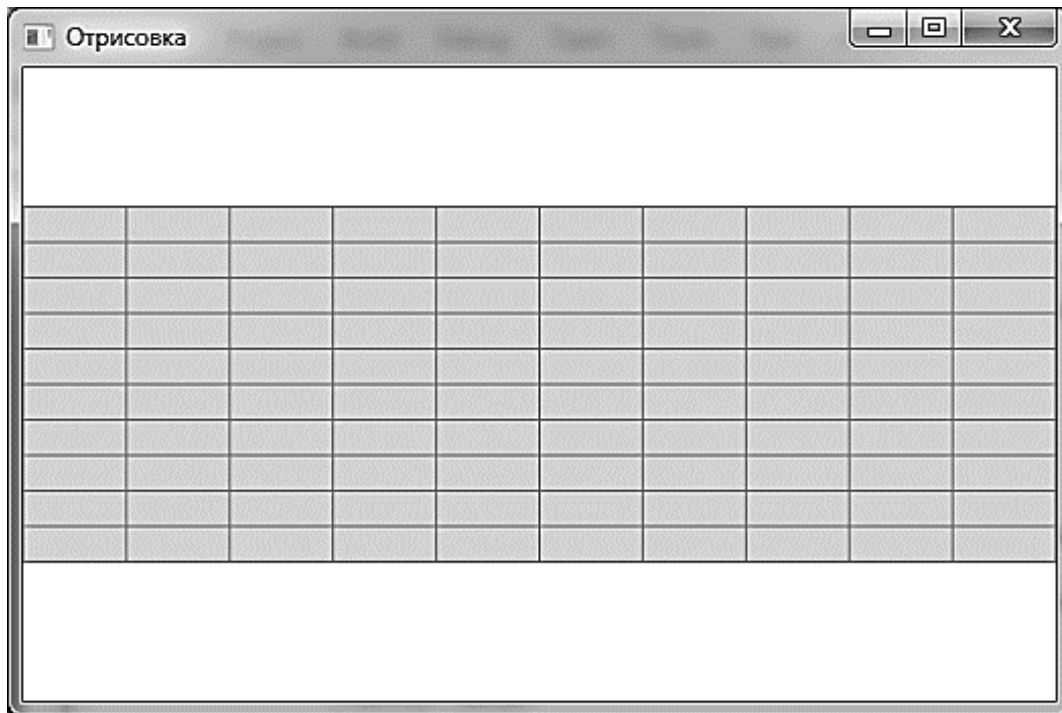


Рисунок 2.13. Результат привязки свойства

Если же нам необходимо к одному свойству привязать значение, вычисленное из значений нескольких свойств, то используется тэг `Multibinding`, в параметре которого указывается конвертер, а в содержимом находятся тэги `Binding` для каждого из требуемых свойств. При этом класс-конвертер должен быть унаследован от интерфейса `IMultiValueConverter`.

Теперь перейдем к решению исходной задачи – выводу квадратного поля, независимо от размеров окна.

Первое, что нам необходимо сделать, – определить свой класс, наследник `Canvas`, отрисовывающий поле. Например, как показано в листинге 2.26.

Листинг 2.26

```
class MyBoardCanvas : Canvas
{
    public double cellwidth
    {
        get { return this.ActualWidth / 10; }
    }
    public double cellheight
    {
        get { return this.ActualHeight / 10; }
    }
    protected override void OnRender(DrawingContext dc) {
        base.OnRender(dc);
        Pen pen = new Pen(new SolidColorBrush(Colors.Gray), 1);
        for (double y = 0; y <= this.ActualHeight+cellheight/2;
            y += cellheight){dc.DrawLine(pen, new Point(0d, y), new Point(this.ActualWidth, y));
        }
        for (double x = 0; x <= this.ActualWidth+cellwidth/2;
            x += cellwidth){dc.DrawLine(pen, new Point(x, 0d), new Point(x, this.ActualHeight)); }
    }
}
```

ActualWidth и ActualHeight в данном случае возвращают реальные размеры элемента. Это расчетное значение, основанное на параметрах среды, в которой работает приложение.

Так как мы будем привязывать наше значение к высоте и ширине, то нам необходим класс-конвертер нескольких значений. Его объявление представлено в листинге 2.27.

Листинг 2.27

```
public class SizeForSquareConverter : IMultiValueConverter
{
    public object Convert(object[] values, Type targetType, object parameter, System.Globalization.CultureInfo culture)
    {
        double res = Math.Min((double)values[0], (double)values[1]);
        if (res < 1) res = 1;
        return res;
    }
    public object[] ConvertBack(object value, Type[] targetTypes, object parameter, System.Globalization.CultureInfo culture)
    {
        throw new NotImplementedException();
    }
}
```

Для возможности использования мы должны описать этот конвертер в ресурсах окна, как показано в листинге 2.28. Затем мы должны присвоить имя элементу, свойство которого будем связывать с нашим Canvas. Однако если мы используем размеры окна, то получим не совсем корректный результат: учтется размер заголовка. Нам же нужны размеры только «рабочей» части окна. Для решения этой проблемы поместим наш Canvas в элемент Grid, привязавшись к его размерам. В листинге 2.29 приведен код для параметра высоты. Ширину вам предлагается привязать самостоятельно.

Листинг 2.28

```
<local:SizeForSquareConverter x:Key="SizeForSquareConverter"/>
```

Листинг 2.29

```
<Grid Name="BaseCanvas">
<local:MyBoardCanvas x:Name="GameBoard" Background="LightGray">
  <Canvas.Height>
    <MultiBinding Converter="{StaticResource SizeForSquareConverter}">
<Binding ElementName="BaseCanvas" Path="ActualHeight"/>
<Binding ElementName="BaseCanvas" Path="ActualWidth"/>
    </MultiBinding>
  </Canvas.Height>
</local:MyBoardCanvas>
</Grid>
```

Теперь рассмотрим следующий пример: пусть на поле отрисовывается эллипс, который будет менять свой цвет в зависимости от числа, введенного пользователем в TextBox.

Первым шагом определим требуемый конвертер. Его метод будет принимать посылаемое ему значение типа object, пытаться преобразовать его в число и в зависимости от результата возвращать нужный цвет. Код разрабатываемого нами конвертера показан в листинге 2.30.

Листинг 2.30

```
public object Convert(object value, Type targetType, object parameter, CultureInfo culture){
    int type = 0;
    try { type = System.Convert.ToInt32(value); }
    catch(Exception ex) { type = 3; }
    switch (type) {
        case 1: return Colors.Red;
        case 2: return Colors.Green;
        case 3: return Colors.Blue;
        case 4: return Colors.Violet;
        case 5: return Colors.Orange;
        default: return Colors.Black; }}

```

После объявления конвертера поместим на форму textbox для ввода значения и Label для его подписи. Для этого определим у Grid два столбца, в первый поместим StackPanel, а в оставшийся – наш Canvas. Ниже, в листинге 2.31 приведен код, реализующий описанное.

Листинг 2.31

```
<Grid Name="BaseCanvas">
<Grid.ColumnDefinitions>
    <ColumnDefinition Width="200"/>
    <ColumnDefinition Width="*/>
</Grid.ColumnDefinitions>
<StackPanel Background="AliceBlue">
<Label Name="text" Style="{StaticResource MyTextStyle}">
Введите тип шарика:</Label>
<TextBox x:Name="typeSh" Style="{StaticResource MyTextStyle}"
Text="1"/></StackPanel>
<local:MyBoardCanvas x:Name="GameBoard" Background="LightGray"
Grid.Column="1">
</local:MyBoardCanvas></Grid>

```

Найдите свойство Name textbox: оно будет использоваться в листинге 2.32, когда мы с помощью конвертера свяжем содержимое

свойства text и цвет градиентной кисти заливки эллипса. Результат работы данного кода представлен на рисунке 2.14.

Листинг 2.32

```
<Ellipse StrokeThickness="3" Width="30" Height="30" Canvas.Left="30">
<Ellipse.Fill>
<RadialGradientBrush GradientOrigin="0.2,0.2">
<GradientStop Color="White" Offset="0" />
<GradientStop Offset="0.8" Color="{Binding ElementName=typeSh, Path=Text, Con-
verter={StaticResource TypeToColorConverter}}"/>
</RadialGradientBrush>
</Ellipse.Fill>
<Ellipse.Stroke>
<SolidColorBrush Color="LightCyan"/>
</Ellipse.Stroke>
</Ellipse>
```

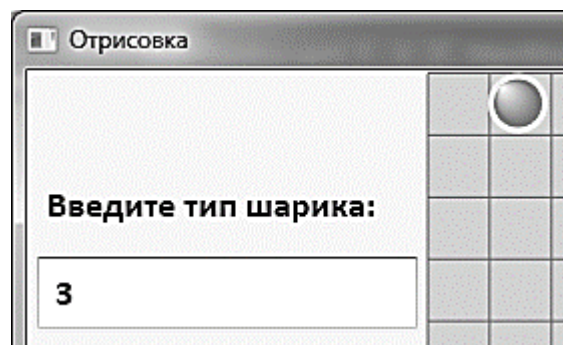


Рисунок 2.14. Отрисовка эллипса

Все свойства, для которых мы определяли привязку, относятся к типу `DependencyProperties`, как и все свойства стандартных элементов управления. Если же мы хотим у своего класса (например, у нашего наследника `Canvas`) определить свойство, для которого можно создать привязку, нам нужно зарегистрировать его как `DependencyProperty`.

Рассмотрим решение следующей задачи: пусть пользователь с помощью элемента `checkbox` управляет отображением сетки (есть галочка – есть сетка, нет – нет).

Первое, что мы делаем, – в нашем классе объявляем публичное статическое поле типа `DependencyProperty`, присвоив ему значение, возвращенное статическим методом `Register` класса `DependencyProperty`. Данный метод принимает следующие параметры:

1. Имя свойства.
2. Тип значения.
3. Тип, содержащий это свойство.
4. Объект типа `FrameworkPropertyMetadata`.

Конструктор последнего принимает параметры:

1. Значение по умолчанию.
2. Флаг, определяющий, что делать при изменении свойства (чаще всего используют значение `FrameworkPropertyMetadataOptions.AffectsRender` – перерисовать).

Для нашей задачи код будет следующим (листинг 2.33).

Листинг 2.33

```
public static readonly DependencyProperty backgroundGridVisebleProperty =
DependencyProperty.Register("BackgroundGridVisible", typeof(bool),
typeof(MyBoardCanvas), new FrameworkPropertyMetadata(true,
FrameworkPropertyMetadataOptions.AffectsRender));
```

После этого нам необходимо объявить само свойство, как показано в листинге 2.34.

Листинг 2.34

```
public bool BackgroundGridVisible {
get{return (bool)this.GetValue(backgroundGridVisebleProperty);}
set{this.SetValue(backgroundGridVisebleProperty,value); }}
```

Мы видим, что его методы `get` и `set` связываются с определенным нами полем.

Теперь внесем изменения в метод `OnRender`, завязав отрисовку сетки на значение нашего свойства, согласно листингу 2.35.

Листинг 2.35

```
if (BackgroundGridVisible){  
for (double y = 0; y <= this.ActualHeight + cellheight / 2; y += cellheight)  
{  
dc.DrawLine(pen, new Point(0d, y), new Point(this.ActualWidth, y));  
}  
for (double x = 0; x <= this.ActualWidth + cellwidth / 2; x += cellwidth)  
{  
dc.DrawLine(pen, new Point(x, 0d), new Point(x, this.ActualHeight));  
}}  

```

После этого обязательно скомпилируем проект, чтобы XAML-файл «подхватил» изменения. Теперь перейдем в него и после строки, объявляющей элемент `textbox`, вставим строку, объявляющую `checkbox` (листинг 2.36) или же перетащим его на форму с `ToolBox`, настроив свойства в окне `Properties`.

Листинг 2.36

```
<CheckBox x:Name="checkBox" Content="Сетка" IsChecked="True"/>
```

После этого пропишем код привязки для нашего свойства в тэге, объявляющем поле (листинг 2.37).

Листинг 2.37

```
<local:MyBoardCanvas x:Name="GameBoard" Background="LightGray"  
Grid.Column="1" BackgroundGridVisible="{Binding ElementName=checkBox,  
Path=IsChecked}">
```

Результат работы кода представлен на рисунке 2.15: слева – сетка есть, справа – нет.

Теперь рассмотрим решение следующей задачи: пользователь кликнул на поле мышкой, нам нужно вывести сообщение с координатами курсора. Одним из способов ее решения будет – обработка нашим полем события `MouseLeftButtonDown`.

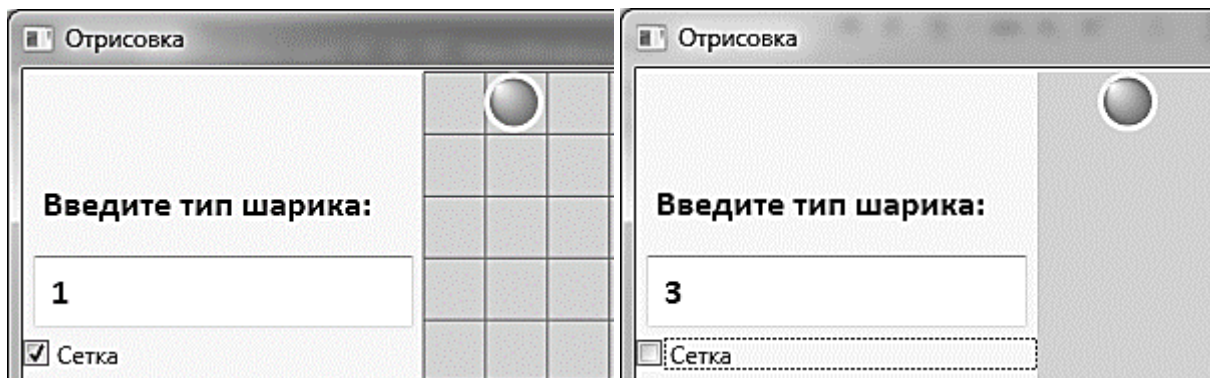


Рисунок 2.15. Управление сеткой

В XAML-файле меняем код для нашего поля, как показано в листинге 2.38. Самый последний параметр здесь – подписка на событие.

Листинг 2.38

```
<local:MyBoardCanvas x:Name="GameBoard" Background="LightGray"
Grid.Column="1" BackgroundGridVisible="{Binding ElementName=checkBox,
Path=IsChecked}" MouseLeftButtonDown="GameBoard_MouseLeftButtonDown">
```

В классе MainWindow, описывающем наше основное окно, пишем метод, который будет вызываться при возникновении события (листинг 2.39).

Листинг 2.39

```
public partial class MainWindow : Window
{
    public MainWindow(){InitializeComponent();}
    private void GameBoard_MouseLeftButtonDown(object sender,
    MouseButtonEventArgs e){
    MessageBox.Show(e.GetPosition((IInputElement)sender).X+ " " +
    e.GetPosition((IInputElement)sender).Y);}
}
```

Итак, мы рассмотрели основы работы с графикой в WPF, познакомились с конверторами типов, а также свойствами зависимости. Теперь можно перейти к еще более сложным элементам и реализо-

вать, наконец, игру «Линии». Однако перед этим вам рекомендуется выполнить несколько заданий для закрепления пройденного материала.

ЗАДАНИЯ К ТЕМЕ 2.2

1. Разработайте на WPF игру «Стрельба по мишеням».
2. Разработайте на WPF игру «Крестики-нолики».
3. Разработайте на WPF игру «Сапер».
4. Разработайте на WPF игру «Морской бой».
5. Разработайте на WPF игру «Пятнашки».

ТЕМА 2.3. РАЗРАБОТКА ИГРЫ «ЛИНИИ»

При разработке игры на WPF используем все возможности объектно-ориентированного программирования. В частности, представим шарик на доске отдельной сущностью, способной реагировать на действия пользователя, а также научим наше основное окно получать от сообщения класса-игры о произошедших изменениях. Для этого нам потребуется вспомнить механизм делегатов и событий, а также познакомиться с классом `ObservableCollection`, а также с элементом `ItemsControl`.

Начнем с первого. В C# классы могут обмениваться между собой сообщениями. Это реализовано через механизм делегатов и событий. Делегат – это класс, а его объект содержит в себе указатели на методы. Механизм событий в C# построен следующим образом. Есть класс, который выполняет какое-либо действие и есть другой класс, который должен при выполнении действия первым классом выполнить свое. Тогда в первом классе определяется событие, а все «заинтересованные» классы на него подписываются. Примером из реальной жизни могут служить оповещения социальной сети «ВКонтакте».

Как только у кого-то происходит события, все «заинтересованные» получают об этом уведомления и могут как-то отреагировать.

Объявление делегата очень похоже на объявление методов. Более того, его внешний вид определяет вид методов, которыми классы могут подписаться на события. Синтаксис объявления делегата показан в листинг 2.40.

Листинг 2.40

<code>delegate <тип возвращаемого значения> Имя(<параметры>);</code>
--

Переменная типа делегат, объявленная в классе – событие. Для их явного отличия от прочих полей используется ключевое слово `event`. Объявленная вне тела класса переменная типа делегат – это просто объект класса делегат.

Когда вы работает с делегатами, вам необходимо выполнить следующие шаги:

1. Определить класс, который будет инициировать событие. Создать у него метод, в котором событие будет происходить.
2. Определить вид метода, которым заинтересованные классы будут реагировать на событие. Объявить делегат, соответствующий виду методов-реакций.
3. Создать поле-событие у основного класса и метод, который будет осуществлять подписку на это событие. Описать вызов события в теле метода основного класса.
4. Описать классы, заинтересованные в событии, и описать их методы-реакции.
5. В основной программе создать объекты требуемых классов, вызвать метод подписки на событие и основной метод, «запускающий» событие.

С программной точки зрения подписка на событие – добавление ссылки на метод к переменной типа делегат, то есть обычное сложение. Синтаксис ее следующий (листинг 2.41).

Листинг 2.41

```
событие+=метод;  
//или  
событие+=переменная-делегат;  
//или  
событие=new ИмяДелегата(метод);  
// Последняя операция выполняется при первой подписке.
```

Рассмотрим пример. Пусть есть класс А, который выполняет некоторое действие, и класс В, который после выполнения действия классом А должен выдать на экран сообщение. Пошаговое решение задачи представлено в листинге 2.42.

Листинг 2.42

```
//Выполняем первый шаг, описанного ранее алгоритма:  
class A { public void Method() { } }  
//На втором шаге определяем вид метода – процедурный, без параметров, и  
//объявляем соответствующий делегат:  
delegate void MyDelegate();  
//Переходим к третьему шагу, расширяем наш класс А:  
class A { public void Method(){ ev(); } public event MyDelegate ev;  
public void Sign(MyDelegate e)  
{ if (ev == null) ev = new MyDelegate(e); else ev += e; } }  
//Четвертый шаг:  
class F { public void fff() { MessageBox.Show("Событие");} }  
//Последний шаг:  
A p = new A(); F f = new F(); p.Sign(f.fff); p.Method();
```

Обратите внимание!

- *При подписке на событие не обязательно иметь переменную-ссылку на объект реагирующего класса. То есть второй оператор четвертого шага из листинга 2.42 можно опустить, а третий тогда перепишется так:
`p.Sign((new F()).fff);`*

С механизмом делегатов и событий вам уже приходилось работать: на нем основана организация взаимодействия с пользователем в приложениях Windows. Основной делегат, описывающий структуру методов-обработчиков событий, имеет вид, показанный в листинге 2.43.

Листинг 2.43

```
public delegate void EventHandler(Object sender, EventArgs e);
```

Параметр `sender` хранит ссылку на объект-источник события, а параметр `e` содержит аргументы события. `EventArgs` – базовый класс для иерархии классов, описывающих разные события. Например, для информации о событиях, связанных с нажатием мыши, объявлен класс `MouseEventArgs`, через объект которого можно получить доступ к координатам курсора. Для событий, описывающих нажатие клавиш на клавиатуре, описан класс `PreviewKeyDownEventArgs` и т. д.

В контексте нашей задачи механизм делегатов и событий должен реализовывать следующее: как только изменилось состояние игрового поля, основное окно должно на это отреагировать отрисовкой. Для этого доработаем наш класс `MainGame`, в котором описана логика игры: добавим делегат, определим событие, опишем в конструкторе подписку на событие, убрав из конструктора появление шаров. Кроме того, определим метод `Move` и добавим вызов события везде, где происходит изменение состояния поля. Помимо этого сделаем еще одну полезную вещь: так как на разные изменения наша программа должна реагировать по-разному (где-то переместить шарик, где-то удалить, где-то нарисовать новый), то для идентификации того, какое именно действие нужно отработать, объявим еще и перечисление. Оно определяется с помощью ключевого слова `enum` и представляет собой непустой список неизменяемых именованных значений. Каждое значение имеет свой порядковый номер, отличающийся от предыдущего на 1, если не указано иное.

Пример объявления перечисления и иллюстрация работы с ним показаны в листинге 2.44. Данное перечисление используется для случая, когда в метод приходит некое значение (параметр f), в зависимости от которого нужно выполнить действие с файлом.

Листинг 2.44

```
enum FileMode {Open,Close,Delete,Remove};  
//где-то в методе:  
if (f==FileMode.Open) File.OpenText("1.txt");
```

Как вы помните, схожую задачу мы решали через константы. Однако перечисление для этой цели использовать удобнее. В нашем случае конструкция будет содержать четыре элемента: перемещение, удаление, добавление, обновление очков. Тогда файл с нашим классом MainGame будет иметь вид, представленный в листинге 2.45.

Примечание: в коде отражены только внесенные изменения.

Листинг 2.45

```
namespace Wpflines {  
public enum Events {path, del, add, scor};  
public delegate void StateChange(object[] n);  
class MainGame{  
public event StateChange stCange;  
public MainGame(int t,StateChange ch){  
if (stCange == null) stCange = new StateChange(ch);  
else stCange += ch;  
pole = new int[t, t];}  
  
public void moveShar(int oldx,int oldy,int newx,int newy){  
stCange(new object[] { Events.path, oldx,oldy,newx,newy }));}  
public bool NextStep(int count){  
//....Код метода...  
for (int k = 0; k < count; k++){  
int i = r.Next(0, 10);  
int j = r.Next(0, 10);
```

Окончание листинга 2.45

```
if (pole[i, j] <= 0){
pole[i, j] = r.Next(1, 6);
stCange(new object[] { Events.add, pole[i, j], i, j });
DelLine(i, j); } else k--;} return true;}
public bool DelLine(int i, int j){
//....Код метода...
if (count_line >= 5){
for (int k = Math.Max(0, mingor); k <= Math.Min(9, maxgor); k++)
{ pole[k, j] = 0;
stCange(new object[] { Events.del, k, j });
}
_Scores = count_line;
stCange(new object[] { Events.scor});
return true;} else count_line = 1;
//....Код метода...
}}
```

Обратите внимание!

- *В нашем делегате описано, что реагирующий на событие метод в качестве параметров должен принимать список объектов. Это сделано потому, что в разных ситуациях мы вызываем событие с разным числом и типом параметров (сравните вызовы в методе DelLine: при удалении мы передаем 3 параметра, а для обновления очков – один).*

Теперь перейдем к реализации интерфейса. Нам необходимо сделать так, чтобы на нашем поле располагались шарики, с которыми пользователь мог бы взаимодействовать. То есть, по сути, у нас есть группа однотипных с точки зрения представления элементов, которые необходимо расположить на нашем Canvas. Для решения этой задачи мы можем использовать ItemsControl. Он берет коллекцию и располагает ее элементы на нужной панели, используя шаблон. Панель, на которой происходит расположение задается тэгом ItemsPanelTemplate,

шаблон представления – `DataTemplate`. Привязка коллекции – параметром `ItemsSource` тэга `ItemsControl`. Шаблон XAML файла, выполняющий описанное, будет таким, как показано ниже в листинге 2.46.

Листинг 2.46

```
<ItemsControl x:Name="GameBoardItemControl" ItemsSource="{Binding ??????}">
  <ItemsControl.ItemsPanel>
    <ItemsPanelTemplate>
      <local:GameBoardCanvas x:Name="GameBoard" Background="LightGray"/>
    </ItemsPanelTemplate>
  </ItemsControl.ItemsPanel>
  <ItemsControl.ItemTemplate>
    <DataTemplate>
      <Ellipse StrokeThickness="3">
        </Ellipse>
      </DataTemplate>
    </ItemsControl.ItemTemplate>
  </ItemsControl>
</Canvas>
```

Посмотрите на строку `ItemsSource="{Binding ??????}"`. Сюда мы должны поместить ссылку на коллекцию элементов, которую будет отображать наш `ItemsControl`. Но ее у нас пока нет, а раз так, то прежде чем двигаться дальше, нам необходимо описать в коде эту коллекцию. Поэтому возвращаемся в файлы `*.cs`.

Так как элементы в нашей коллекции могут добавляться, удаляться или изменять значения своих параметров (менять координаты `X` и `Y`), то нам нужна специальная коллекция, которая имеет возможность генерироваться сообщения при своем изменении, на которые будет реагировать интерфейс. Для этого мы используем `ObservableCollection`. Напомним, что коллекция – это совокупность однотипных элементов произвольного размера. Объявление коллекции следующее (листинг 2.47).

Листинг 2.47

```
ObservableCollection A=new ObservableCollection<Тип_данных>();
```

В случае коллекции указанного типа для использования ее возможности оповещения о событиях, хранящиеся в ней элементы должны наследоваться от встроенного интерфейса `INotifyPropertyChanged`. В контексте нашей задачи коллекция содержит шарики, у которых есть свойство тип, координаты X, Y, а также свойство, говорящее о том, выбран этот шарик игроком или нет. Все эти свойства должны быть связаны с соответствующими полями, а при их изменении должно возникать типовое событие. Опишем класс `Sharik`, содержащий все требуемые поля и реализующий указанное для свойства «выбран» (`Selected`). Для прочих свойств опишем только каркасы, а их функциональный код вам предлагается добавить самостоятельно. Код класса `Sharik` же представлен в листинге 2.48.

Листинг 2.48

```
public class Sharik : INotifyPropertyChanged {

    public event PropertyChangedEventHandler PropertyChanged;
    private void OnPropertyChanged(string PropertyName){
        if (PropertyChanged != null)
            PropertyChanged(this, new PropertyChangedEventArgs(PropertyName)); }
    int _type;
    double _x; double _y;
    double _newX; double _newY;
    bool _selected;

    public bool Selected{
        get { return _selected; }
        set { if (_selected != value)
            {
                _selected = value;
                OnPropertyChanged("Selected");
            }
        }
    }
    public double X {get; set;}
    public double Y {get; set;}
    public double NewX {get; set;}
```

Окончание листинга 2.48

```
public double NewY {get; set;}  
public double Type {get; set;}  
}
```

Обратите внимание!

- *В метод `OnPropertyChanged` передается точное имя свойства. Именно по этому имени будет происходить привязка логики к интерфейсу (позже мы увидим, что все эти свойства участвуют в `Binding` в `XAML`).*

В прошлой теме мы разработали расчерченное сеткой поле, которое всегда остается квадратным. Используем этот класс в нашей задаче, только с небольшими изменениями: назовем его `GameBoard` и добавим функциональность для оповещения об изменении своих свойств, как у класса `Sharik`. Последнее нам необходимо для того, чтобы размеры и положения наших шариков изменялись в зависимости от размера ячейки поля. Обновленный код класса представлен в листинге 2.49.

Листинг 2.49

```
class GameBoardCanvas : Canvas, INotifyPropertyChanged  
{ public double cellwidth { get { return this.ActualWidth / 10; } }  
  public double cellheight  
  { get { return this.ActualHeight / 10; } }  
  public event PropertyChangedEventHandler PropertyChanged;  
  private void OnPropertyChanged(string PropertyName){  
    if (PropertyChanged != null) PropertyChanged(this, new  
    PropertyChangedEventArgs(PropertyName));}  
  protected override void  
  OnPropertyChanged(DependencyPropertyChangedEventArgs e){  
    base.OnPropertyChanged(e);  
    if (e.Property.Name == "ActualWidth" ){  
      OnPropertyChanged("cellwidth");}  
  }
```

Окончание листинга 2.49

```
if ( e.Property.Name == "ActualHeight")
{   OnPropertyChanged("cellheight");}

protected override void OnRender(DrawingContext dc){
    base.OnRender(dc);
    Pen pen = new Pen(new SolidColorBrush(Colors.Gray), 1);
    for (double y = 0; y <= this.ActualHeight+ cellheight/2; y += cellheight){
        dc.DrawLine(pen, new Point(0d, y), new Point(this.ActualWidth, y));}
    for (double x = 0; x <= this.ActualWidth + cellwidth / 2; x += cellwidth){
        dc.DrawLine(pen, new Point(x, 0d), new Point(x, this.ActualHeight));}}
```

Обратите внимание!

- *Наши свойства cellwidth и cellheight являются производными от ActualWidth и ActualHeight, поэтому в методе OnPropertyChanged мы отслеживаем их изменение, а сообщаем об изменении производных.*

Теперь из логики нам осталось описать функциональность самого класса MainWindow, а также требуемые конверторы для Binding. Начнем с первого, так как его логика очень похожа на уже разработанное нами приложение WindowsForms, а конверторы вам будет предложено разработать самостоятельно.

Как мы помним, для реализации игры «Линии» нам нужно обработать нажатие на кнопку (запуск игры), тик таймера (анимация) и клик мыши на поле (перемещение шаров). Кроме того, нам необходимо определить метод, которым наше окно будет реагировать на событие смены состояния поле, определенного у класса MainGame.

Далее определяем вспомогательные глобальные переменные (поля нашего окна). Кроме того, добавляем новые, необходимые в текущем приложении – таймер, коллекцию наших шариков, выбранный шарик и логическую переменную, говорящую о наличии выбранного шарика. Последние два поля вводятся для упрощения обработки клика мыши. Их роль поясним позже. Помимо этого необходимо допи-

сать метод, запускающий игру (обработка нажатия на кнопку). По сути, начало игры – появление шариков на доске. Так как это действие из конструктора класса `MainGame` мы убрали, то начало игры – просто следующий шаг. Тогда код нашего класса `MainWindow` имеет вид, представленный в листинге 2.50.

Листинг 2.50

```
public partial class MainWindow : Window {
    MainGame m; //Логика игры

    // для построения пути в анимации
    int cc = -1; string s; int i_beg; int j_beg;

    // для построения пути в анимации
    DispatcherTimer t;//таймер

    //выбранный шарик
    bool sharicselected = false;
    Sharik selsh;
    //выбранный шарик
    //коллекция шариков
    public ObservableCollection<Sharik> sharikCollection
        {get; private set;}

    public MainWindow()//конструктор
    { InitializeComponent(); }

    //начало игры
    private void Button_Click(object sender, RoutedEventArgs e) {m.NextStep(3); } }
```

Теперь перейдем к коду анимации. В нем будут два изменения: вместо прямой манипуляции с элементами массива-поля – вызов метода `moveShar`, а остановка таймера – методом `Stop()`. Итоговый код будет иметь вид, представленный в листинге 2.51.

Листинг 2.51

```
private void timer_Tick(object sender, EventArgs e) {
    string[] s1 = s.Split(new char[] { '|' }, StringSplitOptions.RemoveEmptyEntries);
    if (cc == -1) cc = s1.Length - 1; else { cc--; }
    if (cc >= 0) { string[] s2 = s1[cc].Split(' ');
    if (cc == s1.Length - 1) {
        i_beg = Convert.ToInt32(s2[0]); j_beg = Convert.ToInt32(s2[1]); }
        else {
            int i_tek = Convert.ToInt32(s2[0]); int j_tek = Convert.ToInt32(s2[1]);
            m.moveShar(i_beg, j_beg, i_tek, j_tek);
            i_beg = i_tek; j_beg = j_tek; }
        if (cc == -1) { t.Stop(); if (!m.DelLine(i_beg, j_beg)) m.NextStep(3); } }
        else { t.Stop(); if (!m.DelLine(i_beg, j_beg)) m.NextStep(3); } }
```

Теперь займемся обработкой клика мыши. В предыдущей теме мы организовывали ее через отслеживание нажатия левой кнопки мыши. В данной теме используем событие `MouseUp`. Прежде чем перейти к коду, необходимо прояснить один момент, касаемый диспетчеризации событий. Так как элементы могут располагаться слоями друг на друге, то все они могут перехватывать событие. Поэтому в код нашего метода мы на всякий случай вставим проверку, что нажатие было выполнено именно на элементе `GameBoard`. В метод `MouseUp` приходят два параметра: `sender` типа `object` и `e` типа `MouseButtonEventArgs`. С помощью первого мы осуществим указанную проверку, а у второго объекта нам потребуется метод `GetPosition` для получения координат курсора и свойство `e.Source`. Оно будет хранить ссылку на объект, на который кликнули. Используя его метод `GetType`, мы проверим: если клик произошел на поле и у нас есть выбранный шарик, то запустим волну и, если она найдет путь, – снимем выделение с шарика и запустим анимацию. Если же под кликом окажется эллипс, то мы снимем выделение с текущего шарика (если таковой есть), запомним координаты нового выбранного шарика и в переменную `selsh` запишем ссылку на него. Код метода представлен в листинге 2.52.

Листинг 2.52

```
private void GameBoard_MouseUp(object sender, MouseButtonEventArgs e){
//проверка на тип объекта
if (sender.GetType() != typeof(GameBoardCanvas)) return;
//сохранение ссылки на поле
GameBoardCanvas GameBoard = (GameBoardCanvas)sender;
//получение координат курсора на поле
Point pos = e.GetPosition(GameBoard);
//перевод координат в строку и столбец массива
Point ourPoint = new Point(Math.Truncate(pos.Y / GameBoard.cellheight),
Math.Truncate(pos.X / GameBoard.cellwidth));
//Если кликнули на пустой клетке
if (e.Source.GetType() == typeof(GameBoardCanvas)){
//Если у нас есть выбранный шарик
if (sharicselected){
//Запуск волны
s = m.Volna(i_beg, j_beg, (int)ourPoint.X, (int)ourPoint.Y);
//Если пути нет
if (s == null) MessageBox.Show("Error!");
//Если путь есть
else {
//снимаем выделение с шарика
sharicselected = false;
selsh.Selected = false;
//сбрасываем текущую точку пути в начало
ss = -1;
//запускаем анимацию (таймер)
t.Start(); }}}
//Если кликнули на эллипсе
if (e.Source.GetType() == typeof(Ellipse))
{ //Если у нас есть выбранный шарик, снимаем выделение с него
if (selsh!=null) selsh.Selected = false;
//Отмечаем, что есть новый выбранный шарик
sharicselected = true;
//Запоминаем координаты клика (в массиве)
i_beg = (int)ourPoint.X;
j_beg = (int)ourPoint.Y;
```

Окончание листинга 2.52

```
//В нашей коллекции находим шарик, стоящий на указанных
//координатах и сохраняем ссылку на него в переменную
selsh = sharikCollection.FirstOrDefault(p => (p.X == j_beg && p.Y == i_beg));
//Меняем свойство «выбран» у найденного шарика.
selsh.Selected = true;}}
```

Поясним некоторые моменты кода. Например, условие `e.Source.GetType() == typeof(Ellipse)`. В данном случае метод `GetType` возвращает тип `Type`, поэтому чтобы сравнить его с типом какого-либо класса (в нашем случае `Ellipse`), нам необходимо применить к имени класса оператор `typeof`, который вернет значение типа `Type`.

Теперь обратимся к следующей строке кода: `sharikCollection.FirstOrDefault (p => (p.X == j_beg && p.Y == i_beg))`. Метод `FirstOrDefault` организует поиск по коллекции, выдав ссылку на элемент, соответствующий условию. Условие в данном случае задается лямбда-выражением.

Согласно официальной документации MSDN [5]: «Лямбда-выражение — это анонимная функция, с помощью которой можно создавать типы делегатов или деревьев выражений. С помощью лямбда-выражений можно писать локальные функции, которые можно передавать в качестве аргументов или возвращать в качестве значений из вызовов функций. Лямбда-выражения особенно полезны при написании выражений запросов LINQ. Чтобы создать лямбда-выражение, необходимо указать входные параметры (если они есть) с левой стороны лямбда-оператора `=>`, и поместить блок выражений или операторов с другой стороны».

В нашем варианте лямбда выражение вернет ссылку на объект в переменную `p`, если этот объект будет соответствовать записанному справа от оператора `=>` условию (координаты объекта совпадают с координатами клика).

Следующим шагом нам необходимо описать реакцию на событие смены состояния поля. В нем мы будем проверять, если прои-

зошло событие добавления шарика, то добавим в коллекцию новый объект с соответствующими параметрами. Если событие удаления – удалим из коллекции наш объект. Если событие перемещения – изменим координаты нужного шарика. Как мы помним, наш метод должен принимать массив объектов, в котором элемент с индексом 0 содержит значение типа перечисление, определяющее тип действия, а остальные параметры – координаты. Наш метод будет иметь вид, представленный в листинге 2.53.

Листинг 2.53

```
private void draw(object[] n)
{
    if ((Events)n[0] == Events.add) sharikCollection.Add(new Sharik { Type = (int)n[1], X
    = (int)n[3], Y = (int)n[2] });
    if ((Events)n[0] == Events.del){
        sharikCollection.Remove(sharikCollection.FirstOrDefault(x => (x.X == (int)n[2] && x.Y
        == (int)n[1])));}
    if ((Events)n[0] == Events.path){
        int xk = (int)n[2];
        int yk = (int)n[1];
        int x = (int)n[4];
        int y = (int)n[3];
        Sharik tmp = sharikCollection.FirstOrDefault(p => (p.X == xk && p.Y == yk));
        if (tmp != null){
            tmp.X = x;tmp.Y = y;
            m[y, x] = tmp.Type;
            m[yk, xk] = 0;}
    }
}
```

Теперь нам осталось лишь дописать в конструкторе MainWindow код, выделяющий память под соответствующие объекты, как показано в листинге 2.54.

Листинг 2.54

```
public MainWindow()
{
    InitializeComponent();
    //Для работы привязок устанавливаем контекст данных
    this.DataContext = this;
    //выделяем память под коллекцию
    sharikCollection = new ObservableCollection<Sharik>();
    //создаем объект-игру
    m = new MainGame(10, draw);
    //создаем таймер
    t = new DispatcherTimer();
    //привязываем метод к событию
    t.Tick += new EventHandler(timer_Tick);
    //устанавливаем скорость анимации
    t.Interval = new TimeSpan(15000);
}
```

Поясним строчку `this.DataContext = this`. Она указывает, что все данные для привязки будут расположены именно в данном и только в данном приложении. Без этого `ObservableCollection` не свяжется с XAML-кодом.

Теперь нам осталось лишь собрать все вместе в XAML-файле с помощью привязок, и наша игра заработает. Для начала давайте поймем, как будет выглядеть наше приложение. Мы хотим, чтобы слева располагались кнопка и два элемента `label`, а справа было игровое поле, которое всегда должно быть квадратным и находиться в центре окна, независимо от изменения его размеров. На поле должны находиться шарики, причем всегда в центре ячейки, и также масштабироваться в зависимости от размеров окна. Каждый эллипс окрашивается определенным цветом, в зависимости от его типа, а выбранные шарики обводятся контуром.

Тогда в XAML-файле нам нужно описать элемент `Grid` с двумя столбцами. В первом – `StackPanel` с кнопкой и обоими `label`, во вто-

ром – типовой Canvas (с именем BaseCanvas), внутри которого содержится ItemsControl. Для того чтобы наше поле всегда было квадратным, нам нужно объявить конвертер (SizeForSquareConverter) и с помощью него связать параметры ItemsControl.Height и ItemsControl.Width со свойствами ActualHeight и ActualWidth элемента BaseCanvas.

Для того чтобы ItemsControl всегда находился в центре, нам необходимо его параметр Canvas.Left связать с ActualWidth элемента BaseCanvas и тем же параметром нашего ItemsControl, но через другой конвертер. Назовем его CoordForCenterLocationConverter. Суть его работы будет в вычислении половинной разницы между размером BaseCanvas и ItemsControl, а реализующий его код представлен в листинге 2.55.

Листинг 2.55

```
public object Convert(object[] values, Type targetType, object parameter, System.Globalization.CultureInfo culture)
{
    double side = (double)values[0];
    double innerSide = (double)values[1];
    double res = (side - innerSide) / 2;
    if (res < 0.2 || double.IsNaN(res)) return 0.2;
    return res;
}
```

Посмотрите на порядок привязок: первым будет размер BaseCanvas. Для параметра Canvas.Top все выполняется аналогично, только с использованием свойства ActualHeight.

Для того чтобы эллипс масштабировался в зависимости от размеров ячейки, нам необходимо его свойства Width и Height связать со свойствами cellwidth и cellheight нашего GameBoard. Причем, так как эллипс должен отставать от краев, то сделать это необходимо с помощью конвертера. Назовем его, к примеру, EllipseSizeOffsetConverter. В результате своей работы он должен вернуть размер ячей-

ки, уменьшенный на некую фиксированную величину. Для простоты зададим эту величину в ресурсах окна следующей строчкой (листинг 2.56).

Листинг 2.56

```
<sys:Double x:Key="sizeoffset">-6</sys:Double>
```

А затем используем в привязке, как показано в листинге 2.57.

Листинг 2.57

```
<Ellipse.Width>  
  
<MultiBinding Converter="{StaticResource EllipseSizeOffsetConverter}">  
<Binding ElementName="GameBoard" Path="cellwidth"/>  
<Binding Source="{StaticResource sizeoffset}"/>  
</MultiBinding>  
  
</Ellipse.Width>
```

Для использования тэга `sys:Double` не забудьте прописать в параметрах окна соответствующее объявление пространства имен, как показано в листинге 2.58.

Листинг 2.58

```
xmlns:sys="clr-namespace:System;assembly=mscorlib"
```

Цвет эллипса привязываем также, как мы делали в предыдущей теме. Код привязки показан в листинге 2.59.

Листинг 2.59

```
<GradientStop Offset="0.8" Color="{Binding Path=Type, Converter={StaticResource  
TypeToColorConverter}}"/>
```

Обратите внимание!

- В привязке не указано имя элемента, поэтому по умолчанию свойство *Type* берется у объекта-содержимого *ObservableCollection*.

Для цвета контура определяем конвертер *SelfToColorConverter*, который связывает свойство *Selected* и цвета следующим образом: при значении *true* возвращается *Colors.GhostWhite*, иначе – цвет фона (*Colors.LightGray*). Основной его метод приведен в листинге 2.60.

Листинг 2.60

```
public object Convert(object value, Type targetType, object parameter, CultureInfo culture){ bool type = System.Convert.ToBoolean(value);
    switch (type) { case true: return Colors.GhostWhite;
        case false: return Colors.LightGray; } return null; }
```

Схожим образом задаются связки координат эллипса и свойств *X*, *Y* шариков относительно ячеек поля.

Ниже, в листинге 2.61, приведен полный код XAML-файла, однако вам рекомендуется написать его самостоятельно, обращаясь к приведенному ниже только в случае затруднений.

Листинг 2.61

```
<Window x:Class="WpfLines.MainWindow"
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
xmlns:local="clr-namespace:WpfLines"
xmlns:sys="clr-namespace:System;assembly=mscorlib"
mc:Ignorable="d"
```

Продолжение листинга 2.61

```
Title="Линии" Height="650" Width="650" MinHeight="200" MinWidth="250"
MaxHeight="800" MaxWidth="800">
<Window.Resources>
<sys:Double x:Key="offset">3</sys:Double>
  <sys:Double x:Key="sizeoffset">-6</sys:Double>
  <local:SizeForSquareConverter x:Key="SizeForSquareConverter"/>
<local:CoordForCenterLocationConverter
x:Key="CoordForCenterLocationConverter"/>

<local:TypeToColorConverter x:Key="TypeToColorConverter"/>
<local:SelToColorConverter x:Key="SelToColorConverter"/>
<local:GameBoardCoordConverter x:Key="GameBoardCoordConverter"/>
<local:EllipseSizeOffsetConverter x:Key="EllipseSizeOffsetConverter"/>

<Style x:Key="MyTextStyle">
<Setter Property="Control.FontFamily" Value="Calibri"></Setter>
<Setter Property="Control.FontSize" Value="18"></Setter>
<Setter Property="Control.FontWeight" Value="Bold"></Setter>
<Setter Property="Control.Padding" Value="5"></Setter>
<Setter Property="Control.Margin" Value="5"></Setter>
</Style>
</Window.Resources>

<Grid> <Grid.ColumnDefinitions>
  <ColumnDefinition Width="200"/>
  <ColumnDefinition Width="**"/>
</Grid.ColumnDefinitions>

<StackPanel Background="AliceBlue">
  <Button Click="Button_Click" Style="{StaticResource
MyTextStyle}">Начать</Button>
  <Label Name="Scores" Style="{StaticResource MyTextStyle}">Очки:</Label>
  <Label Name="Count" Style="{StaticResource MyTextStyle}">Количество
шаров:</Label>
</StackPanel>
```

```
<Canvas x:Name="BaseCanvas" Grid.Column="1" Background="LightBlue">
<ItemsControl x:Name="GameBoardItemControl" ItemsSource="{Binding
sharikCollection}">

<ItemsControl.Height>

<MultiBinding Converter="{StaticResource SizeForSquareConverter}">
<Binding ElementName="BaseCanvas" Path="ActualHeight"/>
<Binding ElementName="BaseCanvas" Path="ActualWidth"/>
</MultiBinding>

</ItemsControl.Height>

<ItemsControl.Width>
<MultiBinding Converter="{StaticResource SizeForSquareConverter}">
<Binding ElementName="BaseCanvas" Path="ActualHeight"/>
<Binding ElementName="BaseCanvas" Path="ActualWidth"/>
</MultiBinding>

</ItemsControl.Width>

<Canvas.Left>
<MultiBinding Converter="{StaticResource CoordForCenterLocationConverter}">
<Binding ElementName="BaseCanvas" Path="ActualWidth"/>
<Binding ElementName="GameBoardItemControl" Path="ActualWidth"/>
</MultiBinding>
</Canvas.Left>

<Canvas.Top>
<MultiBinding Converter="{StaticResource CoordForCenterLocationConverter}">
<Binding ElementName="BaseCanvas" Path="ActualHeight"/>
<Binding ElementName="GameBoardItemControl" Path="ActualHeight"/>
</MultiBinding>
</Canvas.Top>
```

```

<ItemsControl.ItemsPanel>

<ItemsPanelTemplate>
<local:GameBoardCanvas x:Name="GameBoard" Background="LightGray"
MouseUp="GameBoard_MouseUp"/>
</ItemsPanelTemplate>
</ItemsControl.ItemsPanel>

<ItemsControl.ItemTemplate>
<DataTemplate>
<Ellipse StrokeThickness="3">

  <Ellipse.Width>
<MultiBinding Converter="{StaticResource EllipceSizeOffsetConverter}">
<Binding ElementName="GameBoard" Path="cellwidth"/>
<Binding Source="{StaticResource sizeoffset}"/>
</MultiBinding>
</Ellipse.Width>

  <Ellipse.Height>
<MultiBinding Converter="{StaticResource EllipceSizeOffsetConverter}">
<Binding ElementName="GameBoard" Path="cellheight"/>
<Binding Source="{StaticResource sizeoffset}"/>
</MultiBinding>
</Ellipse.Height>

  <Ellipse.Fill>
<RadialGradientBrush GradientOrigin="0.2,0.2">
<GradientStop Color="White" Offset="0" />
<GradientStop Offset="0.8" Color="{Binding Path=Type, Converter={StaticResource
TypeToColorConverter}}"/>
</RadialGradientBrush>
</Ellipse.Fill>

```


Окончание листинга 2.61

```
<Ellipse.Stroke>
<SolidColorBrush Color="{Binding Path=Selected, Converter={StaticResource
SelToColorConverter}}"/> </Ellipse.Stroke>
</Ellipse> </DataTemplate>
</ItemsControl.ItemTemplate>

<ItemsControl.ItemContainerStyle>
<Style TargetType="ContentPresenter">
<Setter Property="Canvas.Left">

<Setter.Value>
<MultiBinding Converter="{StaticResource GameBoardCoordConverter}">
<Binding Path="X"/> <Binding ElementName="GameBoard" Path="cellwidth"/>
<Binding Source="{StaticResource offset}"/>
</MultiBinding>
</Setter.Value>
</Setter>

<Setter Property="Canvas.Top">
<Setter.Value>
<MultiBinding Converter="{StaticResource GameBoardCoordConverter}">
<Binding Path="Y"/>
<Binding ElementName="GameBoard" Path="cellheight"/>
<Binding Source="{StaticResource offset}"/>
</MultiBinding></Setter.Value>
</Setter>
</Style>
</ItemsControl.ItemContainerStyle></ItemsControl>

<Label x:Name="myl" Visibility="Hidden" Style="{StaticResource MyTextStyle}">Игра
закончена</Label>
</Canvas>
</Grid>
</Window>
```

В итоге наше приложение будет иметь вид, представленный на рисунке 2.16.

Мы рассмотрели применение технологии WPF для реализации игровых приложений. Как можно заметить, она вполне подходит для разработки казуальных логических игр, однако для более серьезных проектов требуются специальные технологии, одну из которых мы рассмотрим в следующей части книги. Сейчас же вам вновь предлагается выполнить несколько заданий.

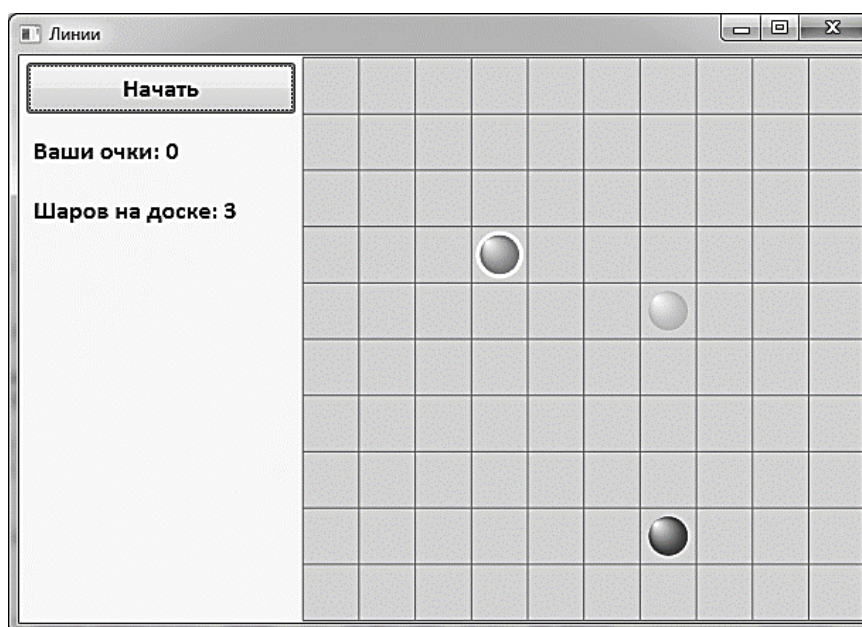


Рисунок 2.16. «Линии» на WPF

ЗАДАНИЯ К ТЕМЕ 2.3

1. Реализуйте игру «Линии» на WPF.
2. Сделайте так, чтобы по завершении игры на экране возникал текст «Игра закончена».
3. Повторите на WPF выполнение заданий 3-5 из темы 1.11.

Часть 3. Работа с технологией OpenGL

В данной части рассматриваются основы работы с графической технологией OpenGL через .Net-библиотеку SharpGL. Кроме того, представлен способ создания видеороликов из OpenGL-анимации путем применения возможностей библиотеки avifil32.dll и технологии платформенного вызова.

ТЕМА 3.1. ОСНОВЫ РАБОТЫ С OPENGL

В данной части мы рассмотрим работу с графикой, используя одну из основных технологий – OpenGL. Согласно Википедии [1]: «OpenGL (Open Graphics Library) – спецификация, определяющая платформонезависимый (независимый от языка программирования) программный интерфейс для написания приложений, использующих двумерную и трехмерную компьютерную графику. Включает более 300 функций для рисования сложных трехмерных сцен из простых примитивов. OpenGL ориентируется на следующие две задачи:

Скрыть сложности адаптации различных 3D-ускорителей, предоставляя разработчику единый API.

Скрыть различия в возможностях аппаратных платформ, требуя реализации недостающей функциональности с помощью программной эмуляции.

Основным принципом работы OpenGL является получение наборов векторных графических примитивов в виде точек, линий и треугольников с последующей математической обработкой полученных данных и построением растровой картинки на экране и/или в памяти. Векторные трансформации и растеризация выполняются графическим конвейером (graphics pipeline), который по сути представляет собой дискретный автомат. Абсолютное большинство команд OpenGL попадает в одну из двух групп: либо они добавляют графические примитивы на вход в конвейер, либо конфигурируют конвейер на различное исполнение трансформаций.

OpenGL является низкоуровневым процедурным API, что вынуждает программиста диктовать точную последовательность шагов, чтобы построить результирующую растровую графику (императивный подход). Это является основным отличием от дескрипторных подходов, когда вся сцена передается в виде структуры данных (чаще всего дерева), которое обрабатывается и строится на экране. С одной

стороны, императивный подход требует от программиста глубокого знания законов трехмерной графики и математических моделей, с другой стороны – дает свободу внедрения различных инноваций».

Работа над OpenGL началась в начале 1990-х. С тех пор данная спецификация прошла несколько этапов развития от OpenGL 2.0 до 4.5, каждая из которых расширяла возможности предыдущей версии, однако ядро данной технологии остается практически неизменным. Понятно, что за такой срок технология приобрела массу разнообразных возможностей, описание которых крайне сложно уместить в рамках одной главы. Поэтому в данной книге мы рассмотрим лишь основные моменты: работа с камерой и вывод примитивов; трансформации и матрицы преобразования; освещение и текстуры; обработка событий от мыши и клавиатуры. Таким образом, цель этого раздела книги – дать начальное представление об OpenGL, необходимые знания для старта работы с этой технологией, а также базу для ее дальнейшего, более углубленного изучения (при необходимости).

Так как базовой средой разработки нами выбрана Visual Studio, то работа с OpenGL будет рассматриваться с использованием именно этого инструмента. Нужно отметить, что выбранная среда разработки создает приложения под определенный .Net Framework. Для OpenGL интеграция с ним напрямую невозможна, поэтому для разработки .Net-приложений необходимо использовать различные .Net-совместимые «обертки», такие как OpenTK или SharpGL. Это библиотеки, которые предоставляют классы для работы с OpenGL, причем имена их методов практически идентичны аналогичным функциям OpenGL. Мы выберем для работы SharpGL, так как она является более новой разработкой.

Введение в SharpGL

SharpGL – бесплатная библиотека для работы с OpenGL. Загрузить ее можно с официального сайта: <https://sharpgl.codeplex.com/releases>. Там же можно найти расширение для шаблонов-типов проекта

для версий Visual Studio до 2013 включительно. Если же вы работаете с VS-2015, то шаблона проекта у вас не будет, однако разработке это не мешает. В зависимости от того, какой тип проекта вы хотите создать (WinForms или WPF), вы скачиваете соответствующее решение. Так как в примерах будет использован WinForms, то с сайта скачан архив SharpGL.WinForms.zip. Внутри архива содержатся библиотеки: SharpGL.dll, SharpGL.WinForms.dll, SharpGL.SceneGraph.dll. Для того чтобы работать с OpenGL, нам необходимо подключить эти библиотеки к проекту и разместить на форме openGLControl. Последнее выполняется следующим образом: переходим в режим проектирования формы, правой кнопкой мыши щелкаем на панели Toolbox и выбираем пункт меню Choose Items (Выбрать элементы), как показано на рисунке 3.1.

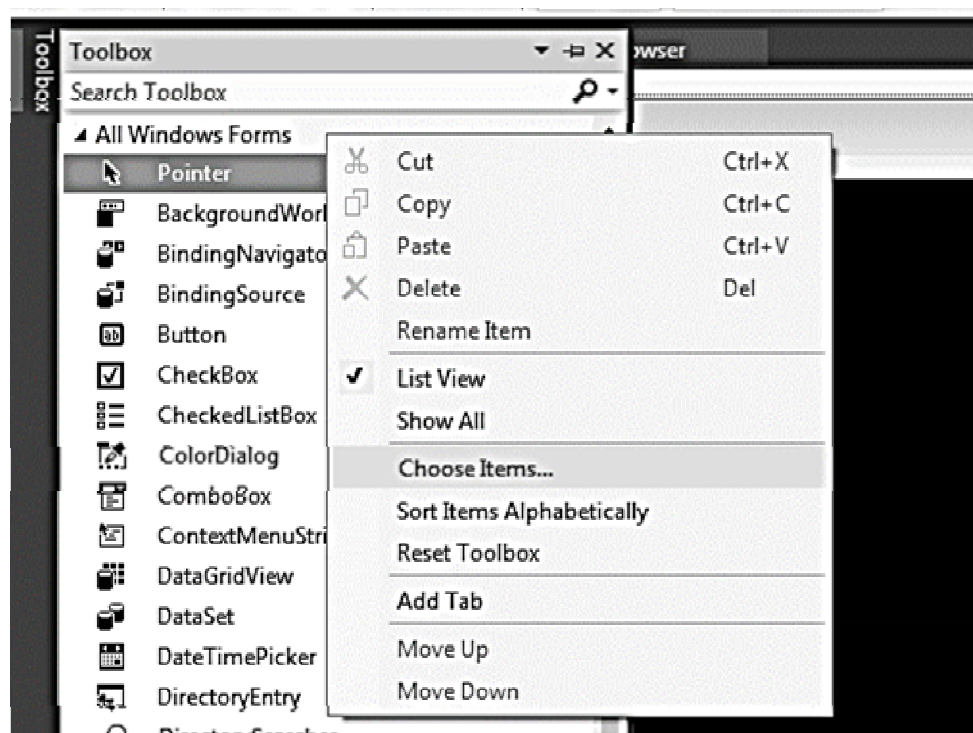


Рисунок 3.1. Выбор элементов Toolbox

В открывшемся окне нажимаем кнопку Browse и ищем библиотеку SharpGL.WinForms.dll. Как только она загрузится, появится окошко с отмеченным галочкой элементом OpenGLControl (рису-

нок 3.2*). Убедитесь, что это так, и закройте окно, нажав кнопку ОК. После этого найдите его на Toolbox и перенесите на форму.

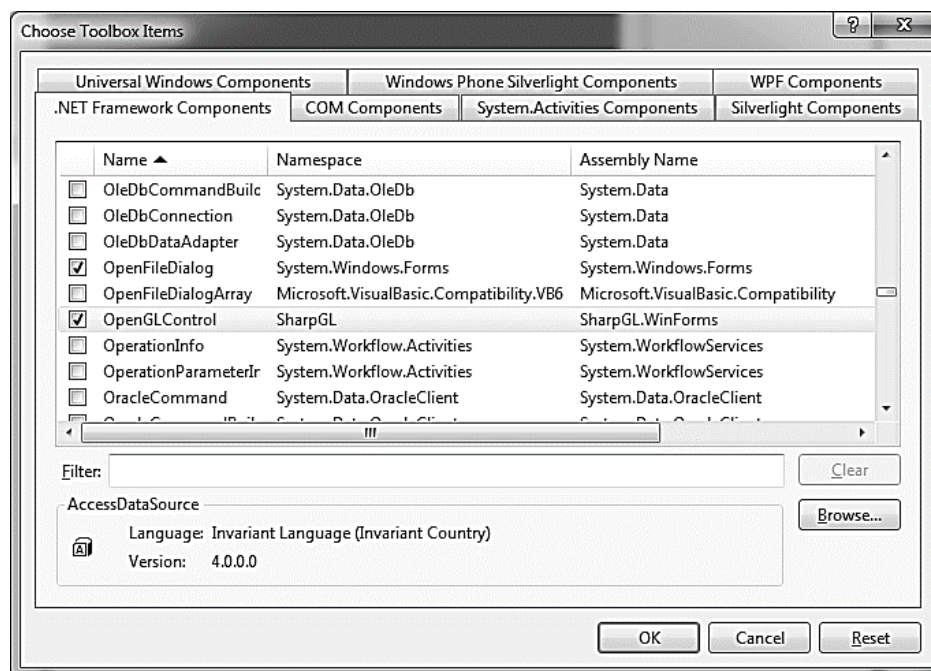


Рисунок 3.2*. Подключение контрола

Для работы с контролом нам понадобятся пока два основных события: `openGLControl1_Load` и `openGLControl1_OpenGLDraw`. Первое – аналог `FormLoad`: возникает при загрузке контрола и появлении его на экране. Второе – метод, отвечающий за перерисовку сцены. Скорость перерисовки задается в свойстве `FrameRate`, измеряемом в герцах. Метод `OpenGLDraw` может вызываться двумя способами: с привязкой к системному таймеру или в ручном режиме. Определяется это свойством `RenderTrigger`: если его значение `TimerBased`, то перерисовка будет вызываться каждую секунду. Если – `Manual`, то для перерисовки необходимо у контрола вызвать метод `DoRender()` (например, по нажатию кнопки).

Рассмотрим обработку событий от мыши и клавиатуры. Второе происходит также в методе `PreviewKeyDown`, как и для технологии GDI+. Первое же – в методе `openGLControl.MouseClick`. Координаты курсора хранятся в свойствах `e.X` и `e.Y`. Однако при работе с OpenGL эти значения нельзя использовать без дополнительных преобразова-

ний. Подробнее с ними мы познакомимся в одной из следующих тем. Пока же рассмотрим вопросы, связанные с отрисовкой простых примитивов.

Отрисовка примитивов и преобразования координат

Прежде чем перейти непосредственно к отрисовке, нам необходимо разобраться с системами координат, используемыми в OpenGL. С точки зрения математики OpenGL оперирует однородными координатами, то есть значениями, сохраненными в матрицах. Если смотреть с точки зрения построения изображения, то систем две: ортогональная и перспективная. Если же смотреть с точки зрения пространственных осей и их расположения, то по сути систем три: мировая, видовая и экранная (рисунок 3.2. а, б и в соответственно). Все они взаимосвязаны. Перспективная или ортогональная система будет определять каким именно образом мы будем переходить из одной координатной плоскости в другую. Во всех же этих переходах мы будем использовать однородную систему координат, то есть работать с матрицами. Поясним сразу, что так как в играх чаще всего используется перспективное отображение, то работать мы будем с ним, а не с ортогональным вариантом.

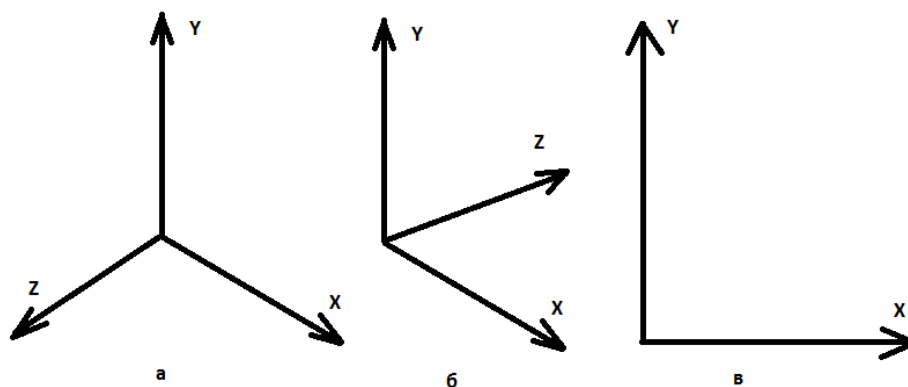


Рисунок 3.2. Системы координат OpenGL

Пояснение к рисунку 3.2.в: координаты левой нижней точки $\{-1, -1\}$, а максимумы по обеим осям равны 1 (верхняя правая точка

имеет координаты $\{1,1\}$). В предыдущих же случаях начало координат – $\{0,0\}$, а максимумы не заданы. Все три системы, изображенные на рисунке 3.2, взаимосвязаны следующим образом: в OpenGL предполагается, что на мир мы смотрим через объектив камеры и отображаем на экране полученную картинку. Таким образом, в координатах первой системы задается объект, который мы хотим отрисовать. Вторая система используется для построения его вида относительно камеры, третья – для отображения на экране. То есть, если нам нужно отобразить нарисованный линиями треугольник, заданный координатами в мировом пространстве, то мы устанавливаем в мировом пространстве камеру и пересчитываем координаты относительно ее области видимости: на экране будет отображаться все, что попало в зону действия камеры (рисунок 3.3).

В трехмерном виде область видимости камеры можно изобразить следующим образом (рисунок 3.4). Тогда на экран попадет все, что заключено между плоскостями А-В, и прочими плоскостями, ограниченными проходящими через вершины плоскостей А-В видовыми лучами. То есть куб С отрисуется на экране, а куб Е – нет.

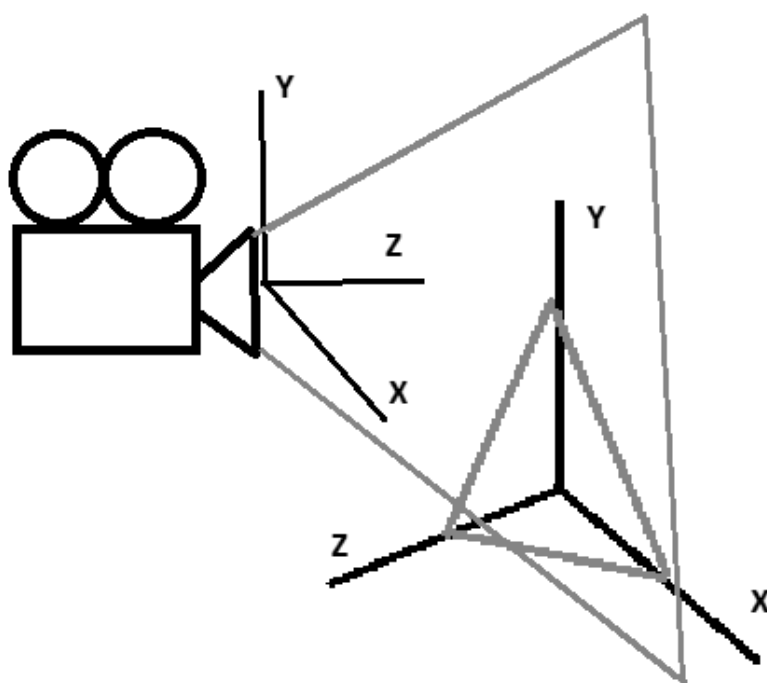


Рисунок 3.3. Взаимосвязь систем координат

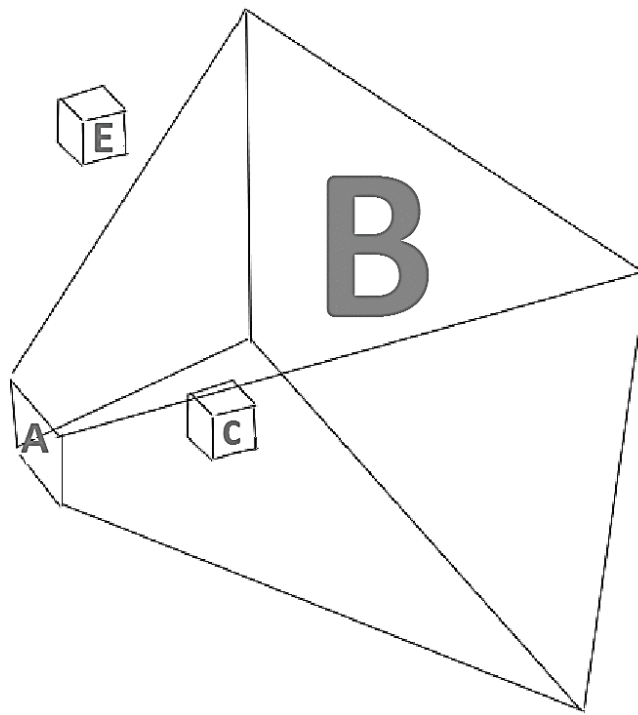


Рисунок 3.4. Трехмерный вид области видимости камеры

Для отображения на экране область видимости камеры превращается в куб, а все объекты, попавшие в него, трансформируются таким образом, что ближайшие к камере – больше, а дальние – меньше (рисунок 3.5).

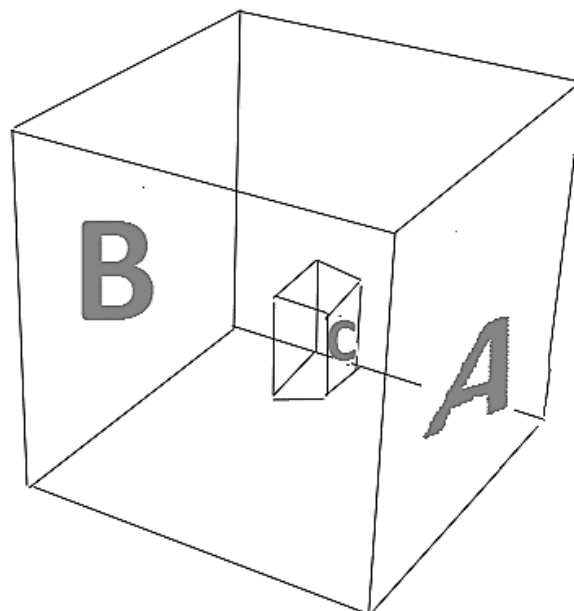


Рисунок 3.5. Преобразование области камеры

То есть строится обычная перспектива. Рассмотрим, как это схематично будет выглядеть на экране. Допустим, в область действия камеры попали полтора куба, тогда на экране они отобразятся так, как показано на рисунке 3.6. Куб С расположен ближе к камере, Е – дальше. При этом мы видим, что куб С не поместился полностью в область видимости: его правая сторона срезана.

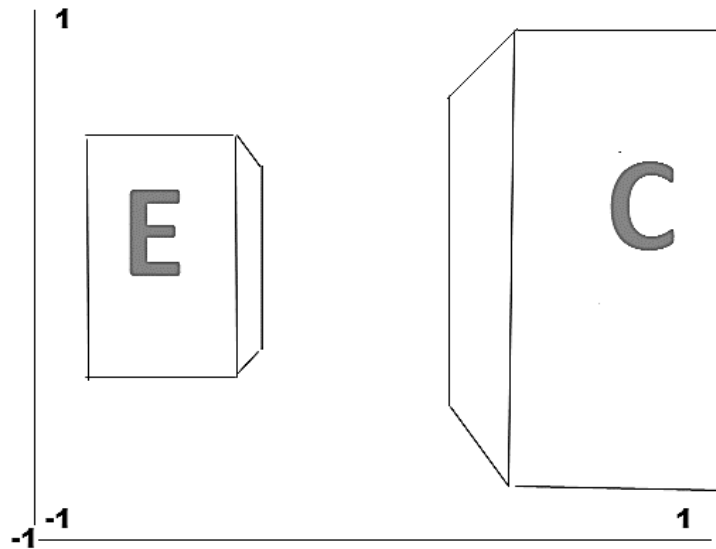


Рисунок 3.6. Схематичное изображение на экране

Реализация перехода из одной системы координат в другую происходит с помощью матриц трансформации. Рассмотрим подробнее этот момент. Для хранения координат точек объектов в OpenGL используются кортежи, в которых сохраняется позиция (x,y,z), заданная в соответствующей системе координат, а также четвертый параметр, указывающий, что мы храним: точку или направление. То есть каждый объект – множество точек в пространстве. Тогда общая схема трансформации следующая:

[Матрица модели] × [Координаты, заданные относительно центра изображаемого объекта] = [Мировые координаты]

[Матрица вида] × [Мировые координаты] = [Видовые координаты]

[Матрица проекции] × [Видовые координаты] = [Экранные координаты]

По сути, переход из одной системы координат в другую – комбинация выполняемых над объектом операций сдвига, поворота и масштабирования. То есть матрицы преобразования координат – это комбинация матриц, выполняющих указанные манипуляции с координатами объекта.

Для лучшего понимания, что такое матричные преобразования, рассмотрим пример операции сдвига для некоторого объекта. Допустим, у нас есть точка с координатами (5,5,5), мы хотим сдвинуть ее на 5 единиц вправо по оси X. Тогда берем матрицу переноса. Она имеет вид

$$\begin{matrix} 1 & 0 & 0 & dx \\ 0 & 1 & 0 & dy \\ 0 & 0 & 1 & dz \\ 0 & 0 & 0 & 1 \end{matrix}$$

В этой матрице dx, dy и dz – смещения по соответствующим осям. Запись нашей операции переноса в матричном виде будет следующей:

$$\begin{matrix} 1 & 0 & 0 & 5 & 5 & 1 \times 5 + 0 \times 5 + 0 \times 5 + 5 \times 1 & 10 \\ 0 & 1 & 0 & 0 & 5 & 0 \times 5 + 1 \times 5 + 0 \times 5 + 0 \times 1 & 5 \\ 0 & 0 & 1 & 0 & 5 & 0 \times 5 + 0 \times 5 + 1 \times 5 + 0 \times 1 & 5 \\ 0 & 0 & 0 & 1 & 1 & 0 \times 5 + 0 \times 5 + 0 \times 5 + 1 \times 1 & 1 \end{matrix}$$

То есть новые координаты нашей точки после преобразования сдвига будут: (10,5,5).

Однако, выполнять матричные операции вручную приходится крайне редко: большинство действий реализуется с помощью соответствующих операторов. В процессе работы программы все необходимые для преобразования матрицы хранятся в стеке в оперативной памяти. Для их задания, помещения в стек или извлечения из него также используются специальные функции. Например, для задания матрицы проекции используется метод Perspective. В качестве параметров он принимает:

1. Значение угла обзора в градусах. Обычно его диапазон – от 30 до 90.
2. Соотношение сторон экрана.
3. Расстояние до ближней плоскости отсечения (A).
4. Расстояние до дальней плоскости отсечения (B).

Матрица вида задается методом LookAt. В качестве параметров принимается:

1. Координаты x , y , z – положение камеры в мировом пространстве.
2. Координаты $x1$, $y1$, $z1$ – положение точки, в которую направлена камера (куда она смотрит).
3. Координаты $x2$, $y2$, $z2$ – поворот «головы» камеры по соответствующим осям.

Начальное задание необходимых матриц происходит при загрузке сцены, однако их изменение может происходить в любых местах кода.

Теперь перейдем к отрисовке примитивов. Общая схема здесь следующая:

1. Вызвать метод Begin, указав, какой тип объекта будет отрисован (линия, треугольник, полигон).
2. Задать координаты точек, задающих объект. Причем их порядок должен быть последовательным: как будто ведем линию, очерчивающую объект от одной точки к другой. Точки задаются с помощью статического метода Vertex, вызываемого у свойства OpenGL нашего контрола. В качестве параметров в метод передаются координаты x , y и z .
3. Вызвать метод End, говорящий, что отрисовка закончена.

При желании перед началом вывода точки можно задать ей цвет. Причем стоит помнить, что для рисования текущего объекта берется последний заданный цвет.

В качестве примера рассмотрим отрисовку картинки с изображением координатных осей и разноцветного домика, которые показаны на рисунке 3.7.

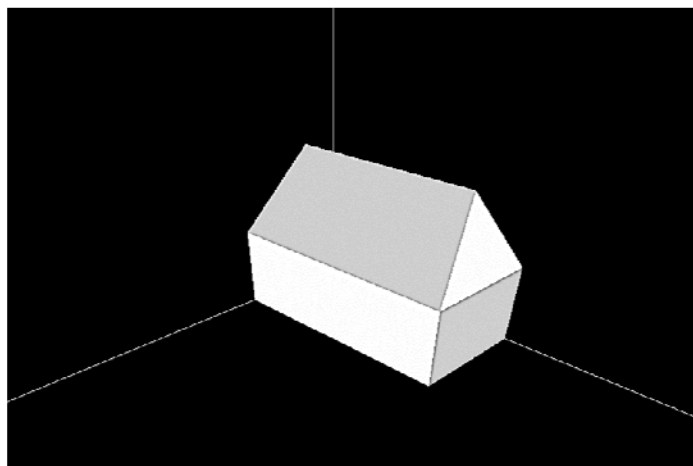


Рисунок 3.7. Тестовая сцена

Первое, что нам необходимо сделать, – задать необходимые матрицы при загрузке контрола (листинг 3.1).

Листинг 3.1

```
//для сокращения кода объявляем ссылку на объект OpenGL
OpenGL GL;

public Form1() {
InitializeComponent();

//получаем ссылку на объект OpenGL нашего контрола
GL = openGLControl1.OpenGL;}
private void openGLControl1_Load(object sender, EventArgs e){
//ставим размеры контрола практически равными размерам формы
openGLControl1.Width = this.Width-30;
openGLControl1.Height = this.Height-100;
// Задаем матрицу
GL.MatrixMode(OpenGL.GL_PROJECTION);
//загружаем матрицу в стек
GL.LoadIdentity();
//устанавливаем параметры матриц
GL.Perspective(80, 4 / 3, 0.1, 200);
GL.LookAt(10, 10,10, 0, 1, 0, 0, 1, 0);
GL.MatrixMode(OpenGL.GL_MODELVIEW);
GL.LoadIdentity();
```

Окончание листинга 3.1

```
//устанавливаем цвет очистки сцены (цвет фона). Первые три //параметра определяют цвет, последний – прозрачность цвета.  
GL.ClearColor(0.1f,1f,0.3f,1);  
// подготавливаем сцену для вывода изображений(очищаем ее)  
GL.Clear(OpenGL.GL_COLOR_BUFFER_BIT |  
OpenGL.GL_DEPTH_BUFFER_BIT);}
```

Теперь переходим в метод OpenGLDraw и рисуем оси, как показано в листинге 3.2.

Листинг 3.2

```
private void openGLControl1_OpenGLDraw(object sender, RenderEventArgs args){  
//Задали красный цвет  
GL.Color(Color.Red);  
//Начали отрисовку линий  
GL.Begin(OpenGL.GL_LINES);  
//Задали координаты начальной и конечной точки для оси X  
GL.Vertex(0, 0, 0);  
GL.Vertex(10, 0, 0);  
//сменили цвет и нарисовали Y  
GL.Color(Color.Yellow);  
GL.Vertex(0, 0, 0);  
GL.Vertex(0, 10, 0);  
//сменили цвет и нарисовали Z  
GL.Color(Color.Blue);  
GL.Vertex(0, 0, 0);  
GL.Vertex(0, 0, 10);  
//Закончили отрисовку линий  
GL.End();}
```

Теперь займемся домом. Определимся, что размер его длинной стороны будет 5, а короткой – 2.5. Реализовать рисование дома вам предлагается самостоятельно, воспользовавшись подсказками: для рисования треугольника крыши используется код из листинга 3.3, а для рисования стен – из листинга 3.4.

Листинг 3.3

```
GL.Begin(OpenGL.GL_TRIANGLES);

GL.Color(Color.Violet);
GL.Vertex(0, 2.5, 0);
GL.Vertex(0, 2.5, 2.5);
GL.Vertex(0.5, 5, 1.25);

GL.End();
```

Листинг 3.4

```
GL.Begin(OpenGL.GL_POLYGON);

GL.Color(Color.Violet);
GL.Vertex(0, 0, 2.5);
GL.Vertex(5, 0, 2.5);
GL.Vertex(5, 2.5, 2.5);
GL.Vertex(0, 2.5, 2.5);

GL.End();
```

Теперь реализуем следующую задачу: пусть по щелчку мыши, сделанному левее оси Y, изображается противоположная сторона дома.

Первое, что нам необходимо сделать – выяснить экранную координату x для оси Y. Так как линия отрисована без искажений, то нам подойдет простая координата X курсора. Для ее выяснения пишем код из листинга 3.5.

Листинг 3.5

```
private void openGLControl1_MouseClick(object sender, MouseEventArgs e)
{ MessageBox.Show(e.X.ToString());}
```

Теперь, запустив приложение и щелкнув на оси, получаем ее координату (рисунок 3.8).

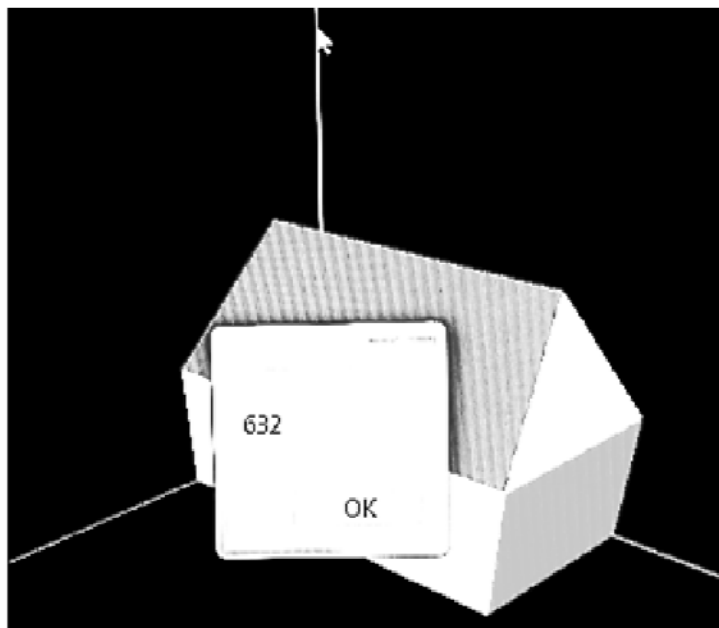


Рисунок 3.8. Получение координаты линии

Запомнив эту координату, объявляем вне тела методов переменную *z*, присваиваем ей значение 10, а затем пишем код в методе *MouseClicked* (листинг 3.6).

Листинг 3.5

```
if (e.X < 632) z *= -1;
GL.MatrixMode(OpenGL.GL_PROJECTION);
GL.LoadIdentity();
GL.Perspective(80, 4 / 3, 0.1, 200);
GL.LookAt(10, 10, z, 0, 1, 0, 0, 1, 0);
GL.MatrixMode(OpenGL.GL_MODELVIEW);
GL.Clear(OpenGL.GL_COLOR_BUFFER_BIT |
OpenGL.GL_DEPTH_BUFFER_BIT);
GL.LoadIdentity();
```

Теперь, если мы щелкнем мышью левее оси *Y*, то получим изображение, представленное на рисунке 3.9.

Примечание: если для покраски соответствующих граней вы выбрали другие цвета, – ваш результат может отличаться по цветовой гамме.

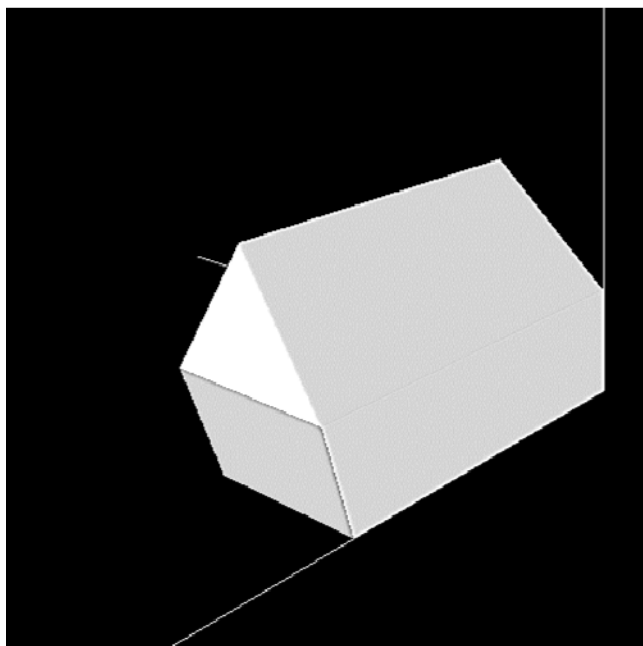


Рисунок 3.9. Смена вида объекта

Далее рассмотрим пример работы с преобразованием координат для организации взаимодействия с объектом. Например, перед нами стоит задача: определить, в какую из ста клеток поля кликнул игрок. Причем поле отображается под наклоном (рисунок 3.10).

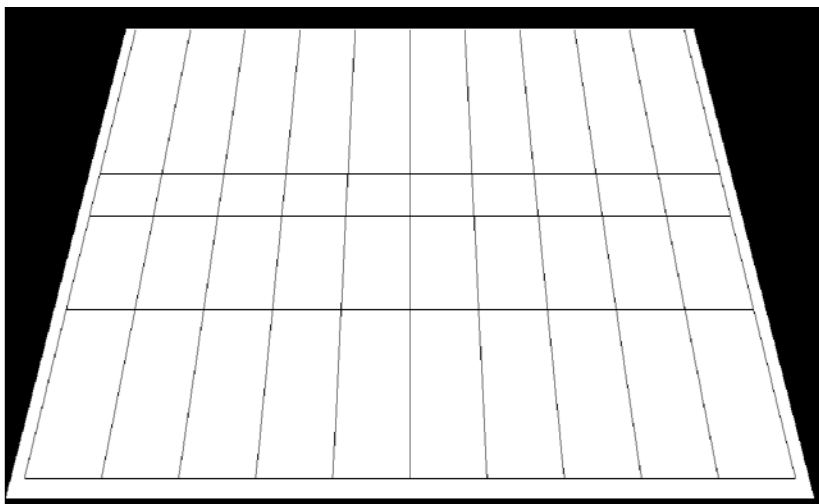


Рисунок 3.10. Отображение поля

Так как из-за перспективы происходит искажение ячеек, то метод простого пересчета экранных координат, который мы использовали ранее, не подойдет. Нам придется определять, между какими

линиями находится курсор мыши, и из этого делать вывод, какой клетке он принадлежит. Для того чтобы реализовать описанное, необходимо привести координаты курсора и координаты линий поля в одну систему координат: либо в экранную, либо в мировую. Первый вариант проще в реализации, поэтому рассмотрим его.

В OpenGL есть функция `Project`. В качестве параметров она принимает вектор координат (объект типа `Vertex`), а возвращает вектор, содержащий их проекцию на экране. Причем выходные координаты выдаются уже не в диапазоне $(-1,1)$, а соответствуют обычным, в системе, где точка $(0,0)$ располагается в левом нижнем углу экрана. Для работы функция использует текущие матрицы проекции и вида.

Сначала рассмотрим построение самого поля на экране. Для этого зададим необходимые матрицы, как показано в листинге 3.6.

Листинг 3.6

```
GL.Perspective(80, 4 / 3, 10, 200);  
GL.LookAt(0, 17, -5, 0, 0, 0, 0, 1,0);
```

Для отрисовки самого поля используем код из листинга 3.7.

Листинг 3.7

```
GL.Begin(OpenGL.GL_POLYGON); GL.Color(Color.White);  
GL.Vertex(11, 0, 11);GL.Vertex(11, 0, -11);GL.Vertex(-11, 0, -11); GL.Vertex(-11, 0,  
11);  
GL.End();  
for (int j = 0; j <= 10; j += 2){ GL.Begin(OpenGL.GL_LINES);  
GL.Color(Color.Black); GL.Vertex(j, 1, 10);GL.Vertex(j, 1, -10);GL.Vertex(10, 1, j);  
GL.Vertex(-10, 1, j); GL.Vertex(-j, 1, 10); GL.Vertex(-j, 1, -10); GL.Vertex(10, 1, -j);  
GL.Vertex(-10, 1, -j); GL.End();}
```

Обратите внимание!

- *Наша камера расположена таким образом, что поле лежит в плоскостях x и z , однако на экране мы все равно будем работать через точки x и y .*

Теперь перейдем к обработке клика мыши. Но до написания кода нам необходимо определиться с алгоритмикой. В самом простом варианте от точки клика нужно найти две ближайшие горизонтальные и вертикальные линии поля. Поэтому первым этапом нам придется запрограммировать решение задачи нахождения минимального расстояния от точки до прямой, то есть построение перпендикуляра. Тогда определение двух ближайших линий сведется к нахождению двух минимальных по длине перпендикуляров. Геометрически данная задача может быть представлена следующей схемой (рисунок 3.11).

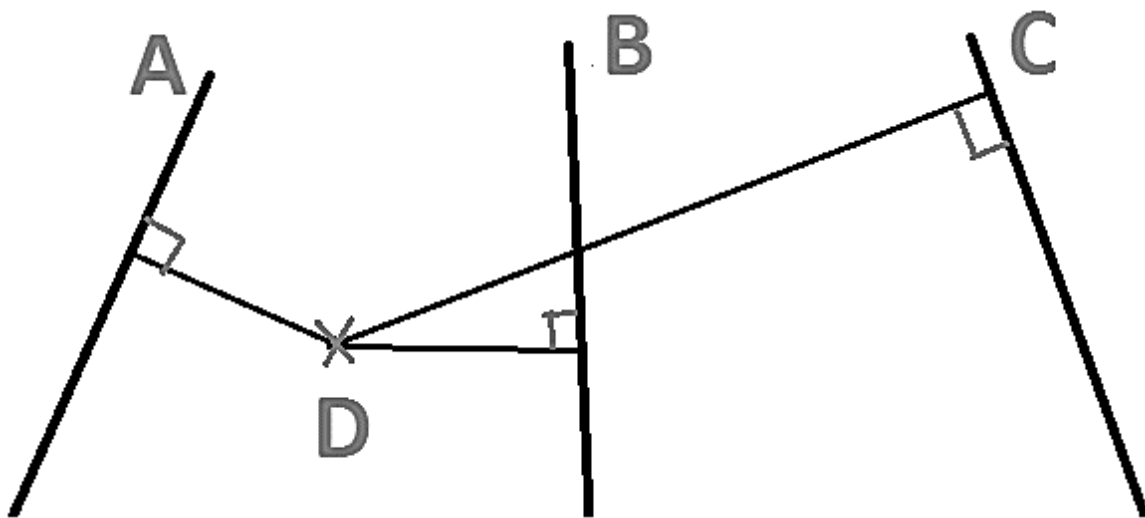


Рисунок 3.11. Геометрический вид задачи

Как мы видим, перпендикуляр от точки D до прямой B меньше, чем от D до C. К линии B точка ближе, чем к линии C. Расстояние же от линий A и B до точки одинаковое.

Разработаем метод, вычисляющий точку пересечения перпендикуляра и линии. В качестве параметров он будет принимать координаты начала и конца линии, а также координаты точки, из которой опускается перпендикуляр. Возвращаться будут координаты точки пересечения. Для простоты заключим координаты в тип `Vertex`. Тогда код метода будет следующим (листинг 3.8). Суть же его работы – из школьного курса геометрии, поэтому подробно останавливаться на ней не будем.

Листинг 3.8

```
/// <summary>
/// Возвращает точку на отрезке AB, через которую из точки C проходит прямая,
/// перпендикулярная отрезку AB
/// </summary>
/// <param name="A">Точка A отрезка AB</param>
/// <param name="B">Точка B отрезка AB</param>
/// <param name="C">Точка C, из которой будет проведен перпендику-
/// ляр</param>
/// <returns>Точка на отрезке AB</returns>
public Vertex Perpendicular(Vertex A, Vertex B, Vertex C){
double x, y;
if (B.X == A.X){ x = B.X; y = C.Y;}
else
if (B.Y == A.Y){ x = C.X; y = B.Y;}
else
{ x = ((B.X - A.X) * (B.Y - A.Y) * (C.Y - A.Y) + A.X * Math.Pow(B.Y - A.Y, 2) + C.X *
Math.Pow(B.X - A.X, 2)) / (Math.Pow(B.Y - A.Y, 2) + Math.Pow(B.X - A.X, 2));
y = (B.Y - A.Y) * (x - A.X) / (B.X - A.X) + A.Y;}
return new Vertex((float)x, (float)y,A.Z);}
```

Обратите внимание!

- *В листинге 3.8 есть комментарии, заданные с помощью тройного слеши. Благодаря ним описание методов и параметров будет отображено в подсказках среды Visual Studio при написании кода вызова метода.*

Теперь для каждой линии поля нам необходимо:

- посчитать проекционные координаты ее начала и конца;
- построить перпендикуляр от точки клика;
- найти его длину;
- запомнить два минимальных по длине перпендикуляра;
- выдать соответствующие номера линий по x и y.

Для того чтобы упростить себе задачу, поручим поиск двух самых коротких перпендикуляра C#: поместим все данные в список и с помощью `linq`-запроса отсортируем и отфильтруем его так, как нам необходимо. Ранее мы реализовывали списки объектов, используя массив. Сейчас же рассмотрим другой способ решения этой задачи: использование класса `List<T>`, где `T` – любой тип данных. Основное отличие от массива – не нужно при определении сразу задавать его длину. Коллекция `List` динамически расширяется или уменьшается при необходимости. Новый элемент добавляется в список методом `Add`. Для хранения наших данных объявим класс с полями `l` – длина перпендикуляра, `px` – номер линии по `x`, `py` – номер линии по `y`. Примем правило: если в поле `px` записывается номер линии, то поле `py` устанавливается в значение `-100`. Если в `py` – номер линии, то в `px` записывается `-100`. Код класса представлен в листинге 3.9.

Листинг 3.9

```
class Sort {public double l;public double px; public double py;}
```

Теперь перейдем к обработке клика мыши. Первое, что сделаем, – объявим список наших объектов и сохраним в объект типа `Vertex` координаты клика, причем координату по оси `y` инвертируем относительно высоты экрана, как показано в листинге 3.10.

Листинг 3.10

```
List<Sort> c = new List<Sort>();  
Vertex mouseLoc = new Vertex(e.X,openGLControl1.Height - e.Y,0);
```

Затем организуем такой же цикл, как мы делали для рисования линий (листинг 3.11). В нем для каждой точки будем высчитывать перпендикуляр и сохранять данные о ней в список. Обратите внимание, что точки с координатами `-j` необходимо обрабатывать только для ненулевого `j`. Ниже приведен пример для точек по оси `x`. Остальные вам предлагается обработать самостоятельно, как и оптимизировать приведенный в листинге 3.11 код.

Листинг 3.11

```
for (int j = 0; j <= 10; j += 2) { double length = 0;
s = Perpendicular(GL.Project(new Vertex(j, 1, 10)), GL.Project(new Vertex(j, 1, -10)),
mouseLoc);
length = Math.Sqrt(Math.Pow(s.X - mouseLoc.X, 2) + Math.Pow(s.Y - mouseLoc.Y,
2));
c.Add(new Sort() { l = length, px = j, py = -100});
if (j != 0) {s = Perpendicular(GL.Project(new Vertex(-j, 1, 10)), GL.Project(new Ver-
tex(-j, 1, -10)), mouseLoc);
length = Math.Sqrt(Math.Pow(s.X - mouseLoc.X, 2) + Math.Pow(s.Y - mouseLoc.Y,
2));
c.Add(new Sort() { l = length, px = -j, py = -100 });}}
```

Теперь займемся сортировкой. В С# есть весьма полезный инструмент – linq-запрос. Данная аббревиатура расшифровывается как «language integrated query». Перевод – запрос, интегрированный в язык. Он позволяет строить запросы наподобие запросов к базам данных. Как и в любом языке, в linq есть свой синтаксис, однако во все его нюансы мы вдаваться не будем. Рассмотрим лишь необходимое. Синтаксис, который мы будем использовать, приведен в листинге 3.12.

Листинг 3.12

```
var <имя_для_результатирующего_списка> =
from <идентификатор_элемента> in <имя_коллекции>
where <условие_фильтрации>
orderby <поле_элемента> ascending
select <идентификатор_элемента>;
```

Пример запроса для вывода списка у-линий, отсортированных по длине перпендикуляра, показан в листинге 3.13.

Листинг 3.13

```
var h = from w in c
where w.px==-100
orderby w.l ascending
select w;
```


Для доступа к элементам списка-результата мы используем цикл `foreach`, прервав его после второго элемента. Данный цикл – разработан специально для вывода элементов коллекций. Его синтаксис показан в листинге 3.14, а использование в нашей задаче – в листинге 3.15.

Листинг 3.14

```
foreach(<тип_элемента> <идентификатор_элемента> in <коллекция>)
```

Листинг 3.15

```
string rez = ""; int i1 = 0;
foreach(Sort cl in h){
    rez += "Длина перпендикуляра: "+cl.l+" Линия x: "+ (5 - cl.px/2)+" Линия y: "+ (5 -
    (cl.py / 2)) + Environment.NewLine;
    i1++; if (i1 == 2)
        break;}
MessageBox.Show(rez);
```

Тогда при клике в левый верхний квадрат появится следующее сообщение (рисунок 3.12).

Обратите внимание!

- *Преобразование $(5 - (cl.py / 2))$ необходимо, чтобы номера линий отображались не в диапазоне от -10 до 10, как они рисуются в цикле, а в диапазоне от 0 до 10.*

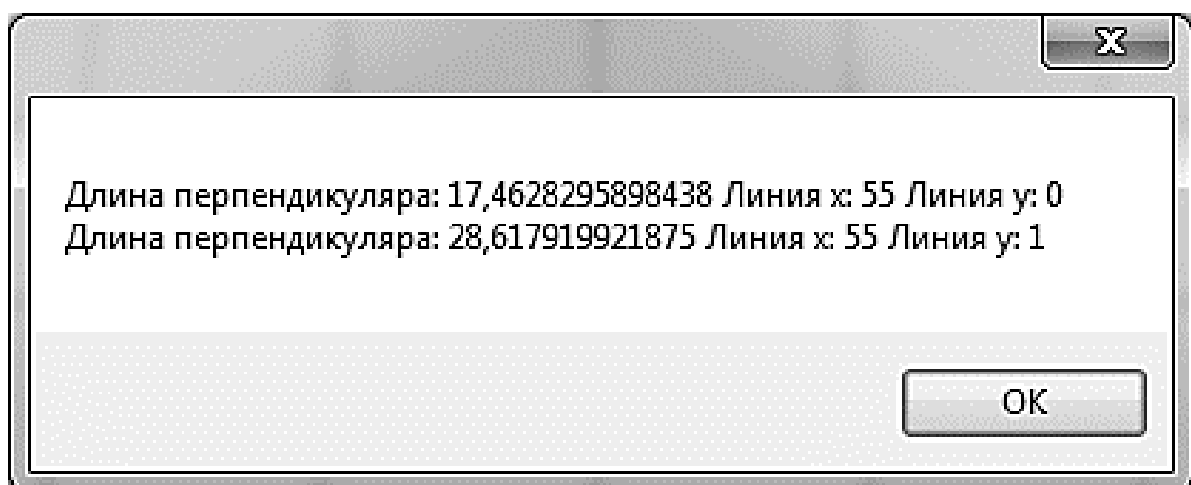


Рисунок 3.12. Результат работы кода

Как было сказано выше, метод Project выдает координаты, используя заданные матрицы. При разработке алгоритмов это необходимо помнить и учитывать. Например, если мы для текущей задачи зададим положение камеры, как показано в листинге 3.16, то клик в левом верхнем прямоугольнике выдаст уже другое сообщение (рисунок 3.13), хотя внешний вид поля не изменится.

Листинг 3.16

```
GL.LookAt(-5, 17,0, 0, 0, 0, 0, 1,0);
```

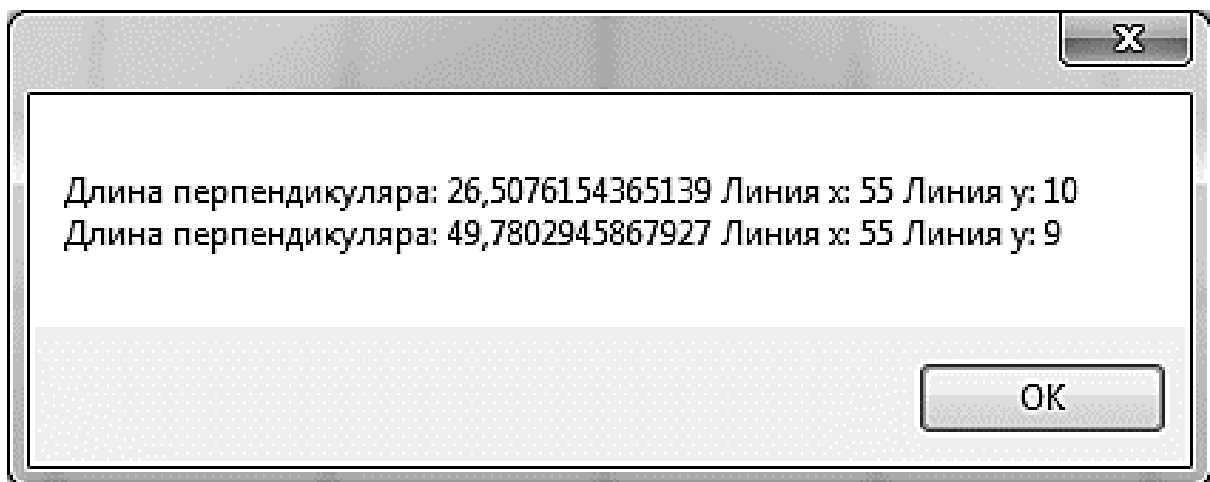


Рисунок 3.13. Результат для другого ракурса

Обратите внимание!

- *У нашего алгоритма есть существенный недостаток: он не проверяет, что курсор находится внутри поля, и если мы кликнем за пределами доски, то текущий алгоритм выдаст, что мы находимся между первой и второй (или последней и предпоследней) линиями, так как к точке клика они будут ближайшими. Исправить это вам предлагается самостоятельно.*

Мы рассмотрели основы работы с OpenGL, используя SharpGL. Теперь перейдем к более сложным элементам, но перед этим вам предлагается выполнить несколько заданий для закрепления материала.

ЗАДАНИЯ К ТЕМЕ 3.1

1. Реализуйте движение домика по любому заданному маршруту.
2. Реализуйте перемещение домика, управляемое клавиатурой.
3. Разработайте игру «Крестики-нолики», используя OpenGL.
4. Разработайте игру «Морской бой», используя OpenGL.
5. Реализуйте вывод изображений (рисунок 3.14).

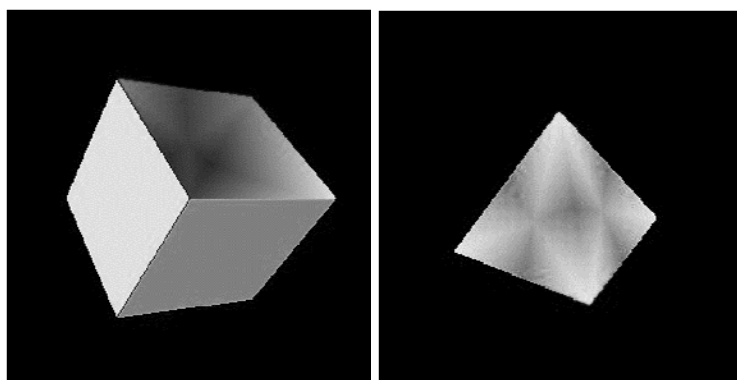


Рисунок 3.14. Изображения для задания 5

ТЕМА 3.2. ОБЪЕМНЫЕ ОБЪЕКТЫ, ОСВЕЩЕНИЕ И ТРАНСФОРМАЦИИ

В прошлой теме мы познакомились с отрисовкой примитивов. Теперь рассмотрим отрисовку более сложных объектов: сферы, цилиндра, тора. Для построения первых двух фигур в SharpGL есть специальные методы, последнюю же придется строить из примитивов, используя ее геометрическое уравнение. Нужно отметить, что и сфера, и цилиндр тоже строятся из примитивов, однако их отрисовка инкапсулирована в методах.

Но начнем со сферы. В чистом OpenGL функции отрисовки данного объекта нет, однако в надстройках для него есть. В SharpGL имеется метод Sphere. В качестве первого параметра он принимает

ссылку на `QuadricObject`. Рассмотрим этот момент подробнее. Для отрисовки сложных объектов в OpenGL используется метод квадрати-рования: каждый объект строится из полигонов. Метод `Sphere` полу-чает ссылку на пустой квадратичный объект и «перестраивает» его так, как необходимо. Остальными параметрами метода являются ра-диус, а также количество «строительных квадратов» по вертикали и горизонтали (чем больше, тем более гладким будет объект). Чтобы отрисовать черную сферу радиусом 10 на белом фоне, нам необходим код, представленный в листинге 3.17.

Листинг 3.17

```
GL.ClearColor(1f, 1, 1f, 1);
GL.Clear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);

var i = GL.NewQuadric();
GL.Color(Color.Black);
GL.QuadricDrawStyle(i, GLU_FILL);

GL.Sphere(i, 10, 25, 25);
```

Посмотрите на метод `QuadricDrawStyle`: он определяет тип отрисовки фигуры. `GLU_FILL` – сплошная заливка, `GLU_LINES` – «сеточное» отображение. На рисунке 3.15 представлено изображе-ние сферы в двух вариантах отрисовки. Слева – сплошная заливка, справа – «сетчатая».

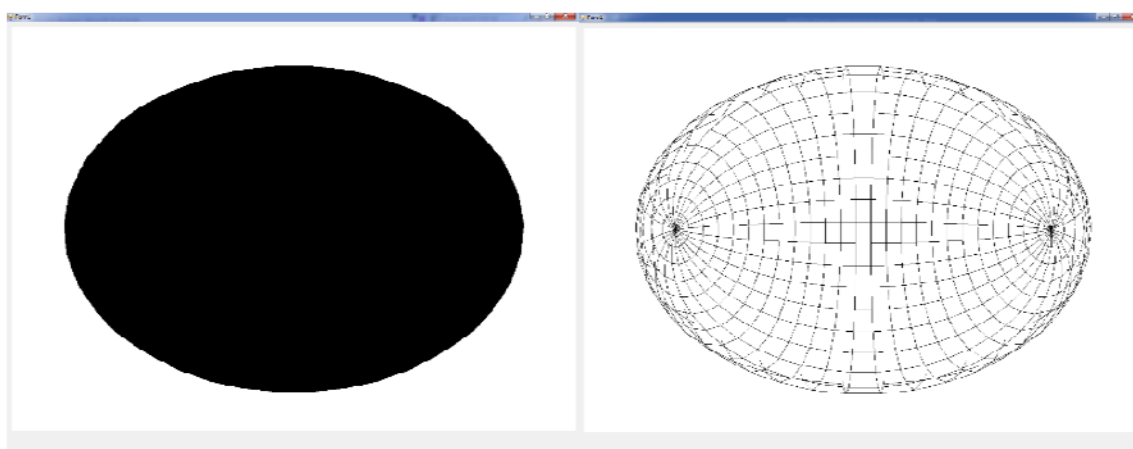


Рисунок 3.15. Две сферы

Диапазон значений цвета для метода ClearColor: от (0,0,0) – черный до (1,1,1) – белый.

Обратите внимание!

- *Ни координатами сферы, ни углом ее поворота мы управлять не можем при отрисовке. Эти параметры должны настраиваться до ее вывода на экран с помощью трансформаций поворота и вращения. Но данные операции не изменяют параметров самого объекта, они изменяют матрицы, используемые для отображения всей сцены.*

Для применения трансформаций к отдельно взятому объекту необходимо выполнять следующие шаги:

1. Сохранить текущее состояние матриц.
2. Применить трансформации.
3. Отрисовать объект.
4. Восстановить сохраненное состояние матриц.

Например, напомним программный код, представленный в листинге 3.18. В итоге на экране отобразится изображение, показанное на рисунке 3.16.

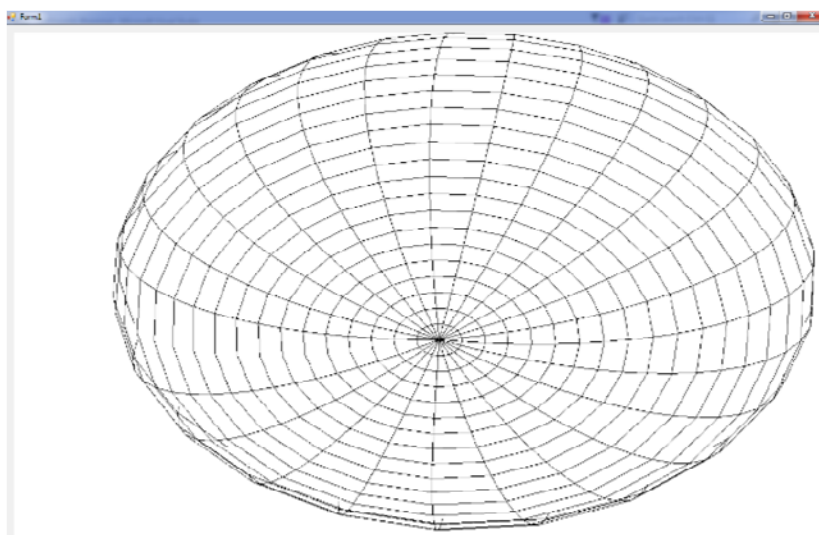


Рисунок 3.16. Сфера с трансформацией

Листинг 3.18

```
var i = GL.NewQuadric();  
//сохранить матрицы  
GL.PushMatrix();  
//трансформация перемещения. Параметры: сдвиг по x,y,z  
GL.Translate(1, 1, 1);  
//трансформация поворота. Параметры: угол относительно x,y,z  
GL.Rotate(-90, 0, 0);  
GL.Color(Color.Black); GL.QuadricDrawStyle(i, OpenGL.GLU_LINE);  
GL.Sphere(i, 10, 25, 25);  
//восстановление состояний матриц до трансформаций  
GL.PopMatrix();
```

Если теперь мы применим метод `Project` к координатам сферы, то получим некорректный результат, так как метод `Project` в том виде, в котором мы его использовали, берет текущие матрицы проекции и вида, а для отрисовки использовались измененные. Для устранения этого в метод `Project` необходимо передавать кроме координат еще и матрицы. Чтобы выгрузить их из стека в массивы, нужен код, представленный в листинге 3.19.

Листинг 3.19

```
int[] viewport = new int[4];  
double[] projection = new double[16];  
double[] modelview = new double[16];  
// выгружаем параметры viewport-a.  
GL.GetInteger(OpenGL.GL_VIEWPORT, viewport);  
// выгружаем матрицу проекции.  
GL.GetDouble(OpenGL.GL_PROJECTION_MATRIX, projection);  
// выгружаем видовую матрицу.  
GL.GetDouble(OpenGL.GL_MODELVIEW_MATRIX, modelview);
```

Для отображения цилиндра используется метод `Cylinder`, который также принимает ссылку на пустой квадратичный объект, а также размеры верхнего и нижнего радиусов, высоту и количество «строительных квадратов». Пример кода приведен в листинге 3.20, а результат его работы – на рисунке 3.17.

```

var i = GL.NewQuadric();
GL.PushMatrix(); GL.Translate(1, 1, 1); GL.Rotate(0, 90, 0);
GL.Color(Color.Black); GL.QuadricDrawStyle(i, OpenGL.GLU_LINE);
GL.Cylinder(i, 5, 2, 10, 25, 25); GL.PopMatrix();

```

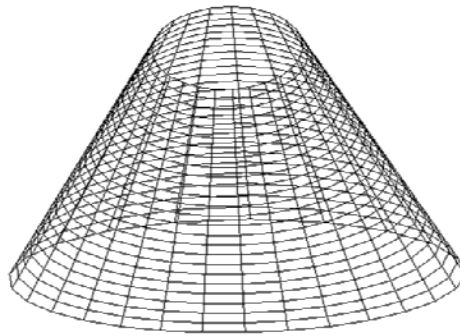
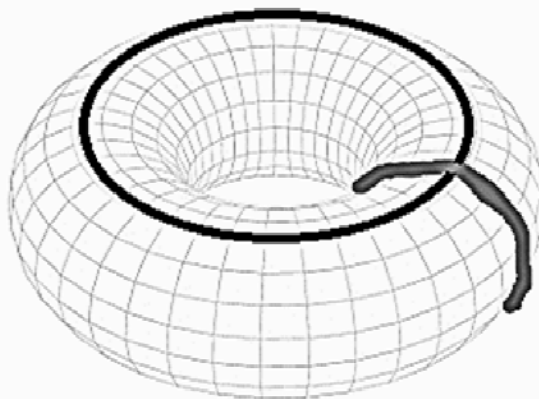


Рисунок 3.17. Цилиндр

Теперь перейдем к тору. В SharpGL специального метода для его построения нет, поэтому необходимо обратиться к геометрическому уравнению, описывающему данную фигуру. Согласно Википедии [1], тор описывается следующим уравнением (рисунок 3.18). В данном случае R – радиус образующей окружности (на рисунке – выделена черным), а r – радиус второй окружности (на рисунке – серым).



$$\begin{cases} x(\varphi, \psi) = (R + r \cos \varphi) \cos \psi \\ y(\varphi, \psi) = (R + r \cos \varphi) \sin \psi \\ z(\varphi, \psi) = r \sin \varphi \end{cases} \quad \varphi \in [0; 2\pi), \psi \in [-\pi; \pi)$$

Рисунок 3.18. Уравнение тора

Зная это, мы можем написать код рисования тора примитивом «линия», как показано в листинге 3.21. Результат работы кода представлен на рисунке 3.19.

Листинг 3.21

```
void DrawThor()  
{  
float x = (float)((5 + 3 * Math.Cos(0)) * Math.Cos(-Math.PI));  
float y = (float)((5 + 3 * Math.Cos(0)) * Math.Sin(-Math.PI));  
float z = (float)(3 * Math.Sin(0));  
GL.Begin(OpenGL.GL_LINES); GL.Color(Color.Red);
```

Окончание листинга 3.21

```
for (float j = (float)(-Math.PI); j <= Math.PI+ Math.PI / 100; j += (float)(Math.PI / 100))  
for (float i = 0; i <= 2*Math.PI+ Math.PI / 100; i+=(float)(Math.PI/100)){  
GL.Vertex(x, y, z);  
x = (float)((5 + 3 * Math.Cos(i)) * Math.Cos(j));  
y = (float)((5 + 3 * Math.Cos(i)) * Math.Sin(j));  
z = (float)(3 * Math.Sin(i));  
GL.Vertex(x, y, z);}  
GL.End();  
}
```

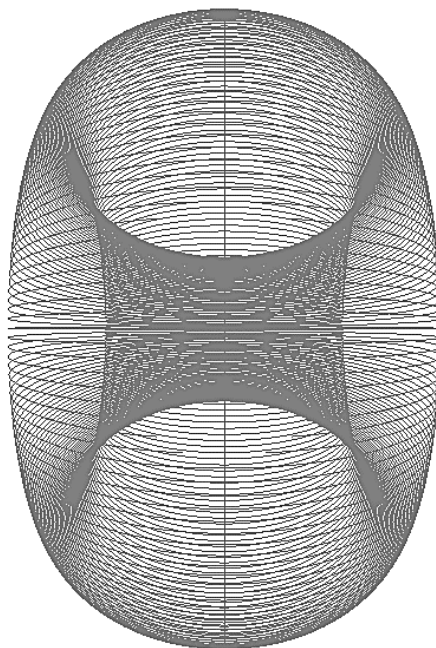


Рисунок 3.19. Тор – линиями

Если же мы хотим сделать вращающийся тор, то необходимо объявить глобальную переменную `rotation`, а в методе отрисовки сцены написать код из листинга 3.22.

Листинг 3.22

```
GL.ClearColor(1f, 1, 1f, 1);  
GL.Clear(OpenGL.GL_COLOR_BUFFER_BIT|OpenGL.GL_DEPTH_BUFFER_BIT);  
rotation += 3.0f;  
GL.Rotate(rotation, 0, 0);  
DrawThor();
```

Если же задать углы поворота случайными по всем осям, а также задать случайные цвета отрисовки и отключить очищение сцены при перерисовке кадра, то можно получить следующее изображение (рисунок 3.20).

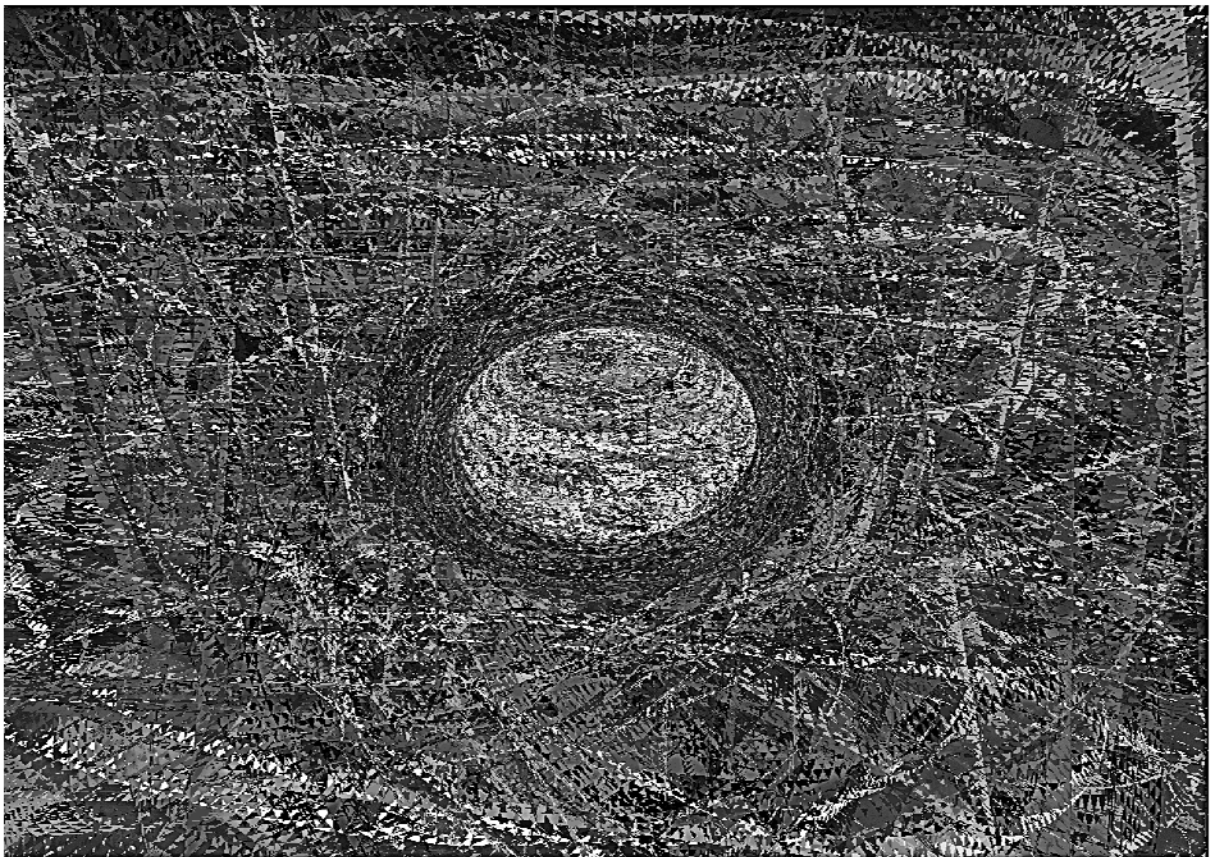


Рисунок 3.20. Случайная картина

Теперь вернемся к цилиндру. Рассмотрим вариант его отрисовки сплошной заливкой. Например, выполняемой кодом из листинга 3.23. В результате на экране получится следующее изображение (рисунок 3.21). Никакого объема у фигуры не видно, хотя та же самая отрисовка линиями покажет, что объем есть.

Листинг 3.23

```
var i = GL.NewQuadric();  
  
GL.PushMatrix();  
  
GL.Translate(1, -10, 1);  
GL.Rotate(50, 60, 0);  
GL.Color(Color.Black);  
GL.QuadricDrawStyle(i, OpenGL.GLU_FILL);  
GL.Cylinder(i, 5, 2, 10, 25, 25);  
  
GL.PopMatrix();
```

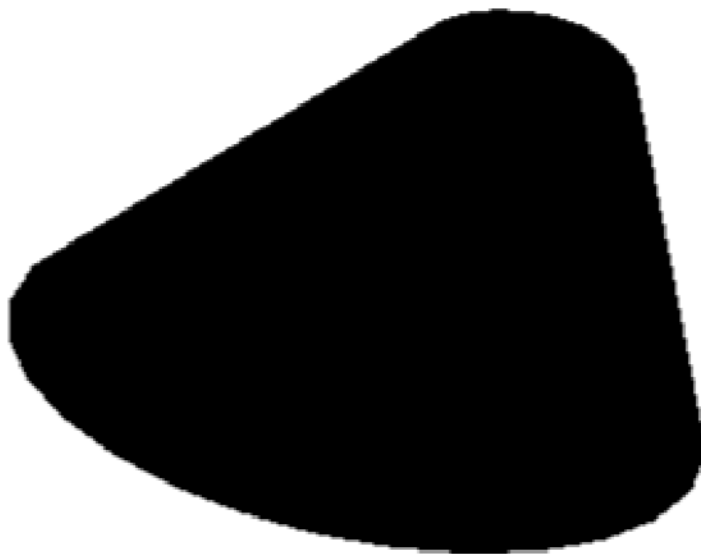


Рисунок 3.21. Цилиндр – сплошной заливкой

Для того чтобы появился объем, нам необходимо включить освещение. Режим освещения включается строчкой кода `GL.Enable(OpenGL.GL_LIGHTING)`, а также активизацией одной из лампочек,

например: `GL.Enable(OpenGL.GL_LIGHT0)`. Написание указанных строчек перед отрисовкой нашего цилиндра даст следующий эффект (рисунок 3.22).

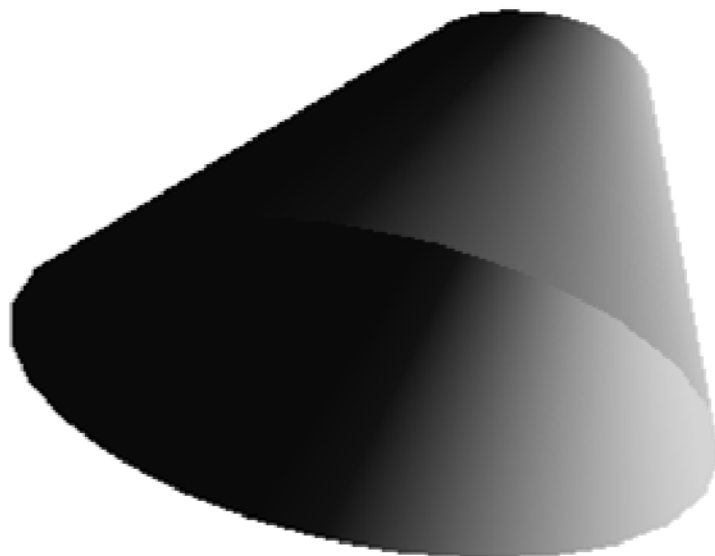


Рисунок 3.22. Освещенный цилиндр

Помимо этого, OpenGL допускает настройку свойств лампочки и комбинацию освещения от нескольких источников. Например, рассмотрим код из листинга 3.24.

Листинг 3.24

```
float[] light0_diffuse3 = { 0f, 1f, 1f };  
float[] light0_direction3 = { 4.0f, 1f, 3.0f, 1.0f };  
  
GL.Enable(OpenGL.GL_LIGHTING);  
GL.Enable(OpenGL.GL_LIGHT5);  
  
GL.Light(OpenGL.GL_LIGHT5, OpenGL.GL_DIFFUSE, light0_diffuse3);  
GL.Light(OpenGL.GL_LIGHT5, OpenGL.GL_POSITION, light0_direction3);
```

В данном случае мы настраиваем для лампы LIGHT5 параметры «цвета» освещения и ее позицию. Результат применения данного освещения к цилиндру представлен на рисунке 3.23.

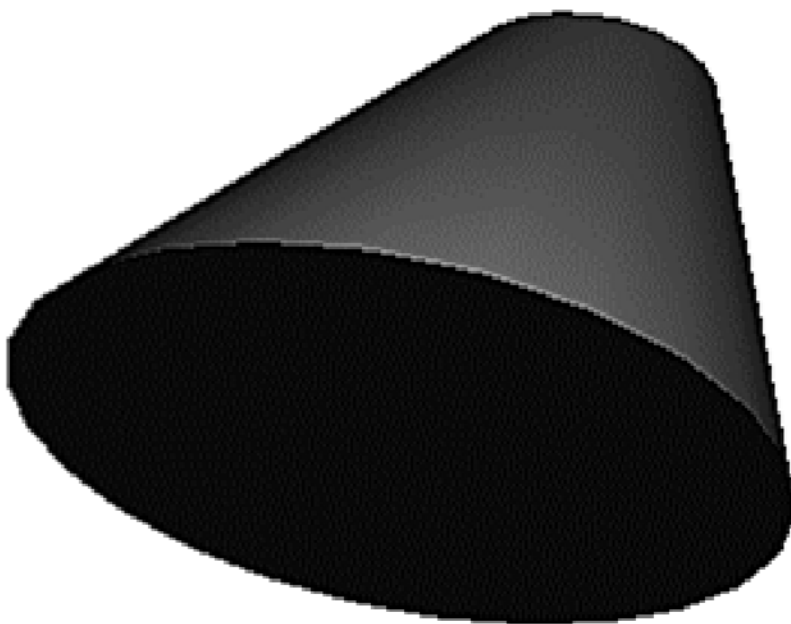


Рисунок 3.23. Другое освещение цилиндра

Если же к лампе LIGHT5 мы добавим LIGHT0, дописав строчку `GL.Enable(GL.GL_LIGHT0)`, то получим следующий результат (рисунок 3.24).

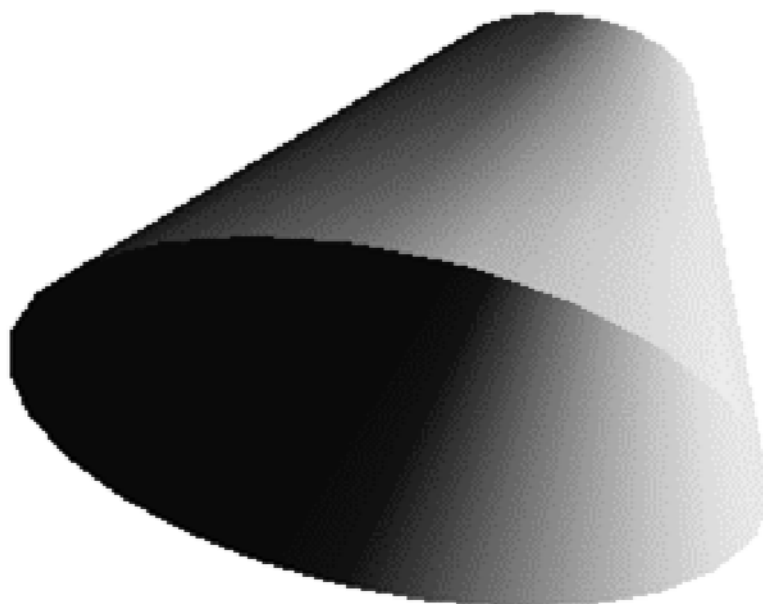


Рисунок 3.24. Комбинированное освещение

Настройка свойств лампы происходит в методе `Light`. При этом последние два параметра указывают, какое именно свойство у лампы будет настраиваться и с применением каких значений. Свойство задается с помощью константы. В нашем примере мы настроили цвет (константа `GL_DIFFUSE`), задав цвет в формате RGBA (все компоненты меняются от 0 до 1) через массив чисел, а также позицию лампы (константа `GL_POSITION`), задав в массиве координаты `x`, `y`, `z` и указатель, что лампа находится в точке (последний параметр равен 1). Если бы он был равен 0, то лампа считалась бы бесконечно удаленной от объекта, а ее свет направлялся бы в указанную точку. Помимо указанных констант, в настройке освещения могут использоваться:

- `GL_SPOT_EXPONENT` – распределение интенсивности цвета. Для этой константы используется один параметр из диапазона от 0 до 128, описывающее интенсивность света и его уровень фокусировки. При значении, равном 0, – рассеянный свет.
- `GL_SPOT_CUTOFF` – угол разброса света. Диапазон значений параметра: от 0 до 90 или 180.
- `GL_AMBIENT` – цвет фоновое освещение. Параметры аналогичны параметрам для константы `GL_DIFFUSE`.
- `GL_SPECULAR` – цвет зеркального отражения. Параметры аналогичны параметрам для предыдущей константы.
- `GL_SPOT_DIRECTION` – направление цвета. Параметры аналогичны параметрам для константы `GL_POSITION`.

Помимо освещения для объектов можно задавать материалы. Рассмотрим отрисовку цилиндра с использованием материала, а не цвета. Реализуется это кодом из листинга 3.25, результат работы которого показан на рисунке 3.25.

Листинг 3.25

```
float[] light0_diffuse3 = { 0f, 1f, 1f };  
float[] light0_direction3 = { 4.0f, 1f, 3.0f, 1.0f };  
float[] light0_diffuse = { 0.5f, 0.1f, 1f };
```

Окончание листинга 3.25

```
GL.Enable(GL.GL_LIGHTING);
GL.Enable(GL.GL_LIGHT5);
GL.Enable(GL.GL_LIGHT0);
GL.Light(GL.GL_LIGHT5, GL.GL_DIFFUSE, light0_diffuse3);
GL.Light(GL.GL_LIGHT5, GL.GL_POSITION, light0_direction3);
var i = GL.NewQuadric();
GL.PushMatrix();
GL.Translate(1, -10, 1);
GL.Rotate(50, 60, 0);
GL.Material(GL.GL_FRONT_AND_BACK, GL.GL_AMBIENT,
light0_diffuse);
GL.QuadricDrawStyle(i, GL.GLU_FILL);
GL.Cylinder(i, 5, 2, 10, 25, 25);
GL.PopMatrix();
```

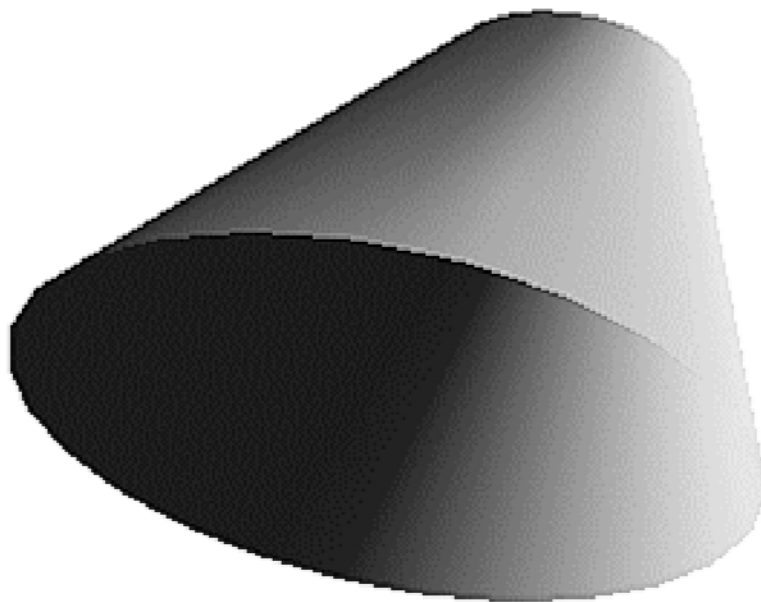


Рисунок 3.25. Материал плюс комбинированное освещение

Команда `GL.Material` принимает параметр, определяющий, к какой грани объекта применяется указанное свойство материала, также параметр-идентификатор свойства и параметр – значение для свой-

ства. Как и в случае с освещением, существует определенный набор констант – идентификаторов свойств:

- `GL_AMBIENT` – цвет материала в тени. Значение аналогично такому же свойству для света.
- `GL_DIFFUSE` – цвет диффузного отражения материала. Значение аналогично такому же свойству для света.
- `GL_SPECULAR` – цвет зеркального отражения. Значение аналогично такому же свойству для света.
- `GL_SHININESS` – степень зеркального отражения. Значение – число из диапазона от 0 до 128.
- `GL_EMISSION` – интенсивность излучаемого материалом света. Значение – цвет в формате RGBA (также, как для константы `GL_AMBIENT`).

Мы рассмотрели работу с объемными объектами и трансформацией. Теперь перейдем к последней теме – работа с текстурами и осуществление выбора объектов. Но перед этим вам рекомендуется выполнить приведенные ниже задания.

ЗАДАНИЯ К ТЕМЕ 3.2

1. Реализуйте отрисовку тора через примитив «треугольник».
2. Реализуйте вывод тора через примитив «полигон».
3. С помощью OpenGL отрисуйте натюрморт: настольная лампа и лежащее рядом яблоко.
4. Воспроизведите изображения, представленные на рисунке 3.26, используя код отрисовки тора.
5. Разработайте анимацию, используя код отрисовки цилиндра, сферы и тора, в конце которой на экране появится следующее изображение (рисунок 3.27).

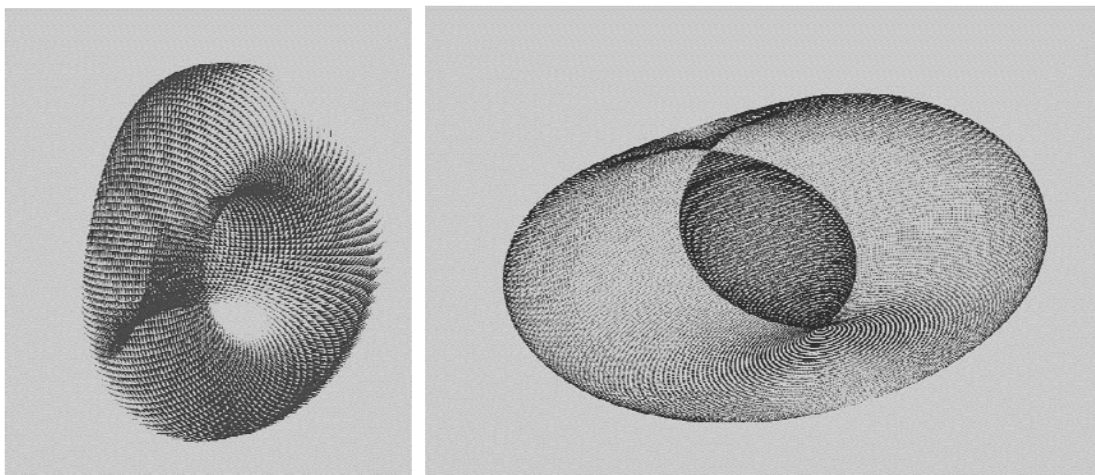


Рисунок 3.26. Изображения для задания 4

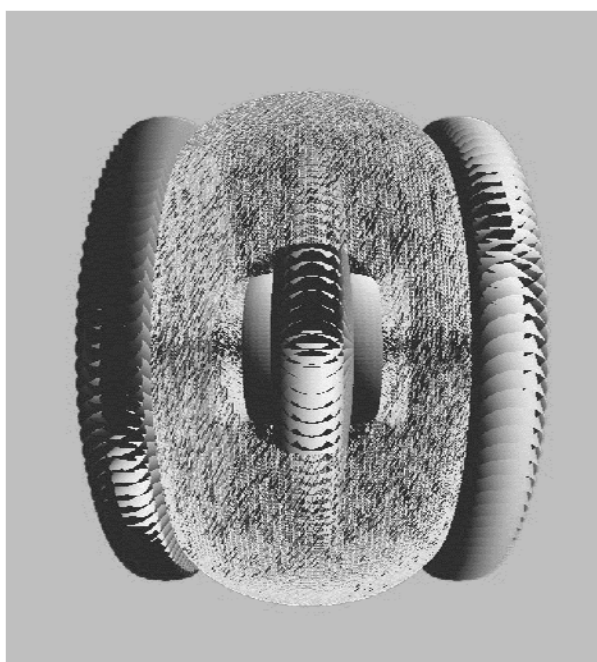


Рисунок 3.27. Изображения для задания 5

ТЕМА 3.3. ТЕКСТУРЫ И ВИДЕОРОЛИКИ

Для знакомства с текстурами приведем вырезку из статьи Википедии [1]: «Текстура – изображение, воспроизводящее разнообразные визуальные свойства каких-либо поверхностей или объектов. В компьютерной графике текстурами часто называют растровые цифровые изображения, содержащие текстурные элементы.

Текстуру часто ошибочно называют фоном. Понятие фона относится к перспективному месту на изображении, заднему плану. Текстура же в этом смысле — это изображение, визуально отображающее совокупность свойств поверхности какого-либо объекта (реального или вымышленного).

Понятия «текстура» и «фактура» применительно к свойствам какой-либо поверхности используются синонимично. Иногда словом «фактура» называют совокупность тактильных свойств, а текстурой — совокупность свойств визуальных. Однако за цифровым графическим изображением таких свойств закрепилось слово «текстура».

Для цифрового художника предпочтительнее использовать текстуры большого размера, даже если результат его работы по размеру меньше, чем сами текстуры. Это связано с эффектом антиалиасинга, возникающем, если пытаться увеличивать текстурное изображение в графическом редакторе. При увеличении с антиалиасингом образуется размытость изображения — результат работы программных алгоритмов вычисления и усреднения цвета, что может существенно снизить качество текстуры на создаваемом изображении. В противоположность этому, при уменьшении большого изображения такие эффекты незаметны. Поэтому текстуры большого размера (англ. *high resolution*) более ценны как инструмент художника. Размер текстур точно так же, как и размеры обычных растровых изображений, измеряют в пикселях.

По эффектам при замощении выделяют обычные (или шовные) и бесшовные текстуры (англ. *Seamless patterns*). Бесшовные текстуры при сочленении не образуют видимого шва, то есть нарушения текстурного рисунка, поэтому ими можно безболезненно замостить холст сколь угодно большого размера. Бесшовные текстуры часто называют паттернами, что является калькой с англ. *Pattern* — *узор*. Например, в растровом графическом редакторе Adobe Photoshop во многих последних версиях имеются предустановленные бесшовные текстуры для замощения холста.

По типу изображаемой текстуры можно выделить:

- текстуры природных объектов (древесная кора, листья, небо и т. д.);
- текстуры поверхностей из различных материалов (деревянные, металлические, глиняные, каменные, бумажные поверхности и т. д.);
- текстуры шума, ветра, царапин, выщербленности, иных повреждений;
- абстрактные текстуры, на которых не изображены объекты, но имеется более-менее однородный фон и так далее.

К способам получения текстур относятся:

- фотографирование объекта, содержащего текстуру, на цифровую фотокамеру с опциональной пост-обработкой в графическом редакторе;
- сканирование объекта, содержащего текстуру. Недостаток этого способа в том, что при преобладающей распространенности сканеров планшетного типа объемные элементы (к примеру, габаритный деревянный щит) отсканировать трудно или вовсе невозможно;
- отрисовка текстуры «с нуля» в графическом редакторе. Таким образом создаются, как правило, абстрактные текстуры, которым нет аналога в окружающем мире.

В трехмерной графике текстура – растровое изображение, накладываемое на поверхность полигональной модели для придания ей цвета, окраски или иллюзии рельефа. Приблизительно использование текстур можно легко представить как рисунок на поверхности скульптурного изображения. Использование текстур позволяет воспроизвести малые объекты поверхности, создание которых полигонами оказалось бы чрезмерно ресурсоемким. Например, шрамы на коже, складки на одежде, мелкие камни и прочие предметы на поверхности стен и почвы.

Качество текстурированной поверхности определяется *текселями*, то есть количеством пикселей на минимальную единицу текстуры. Так как сама по себе текстура является изображением, разрешение текстуры и ее формат играют большую роль, которая впоследствии сказывается на общем впечатлении от качества графики в 3D-приложении.

Метод проецирования текстур для использования в трехмерной графике впервые был предложен Эдвином Катмуллом в 1974 году.

Существует технология *parallax mapping* для создания трехмерного описания текстурированной поверхности с использованием карт смещения.

Parallax mapping («параллактическое отображение», также известен как *offset mapping*, *per-pixel displacement mapping* или *virtual displacement mapping*) – программная техника (методика) в трехмерной компьютерной графике, усовершенствованный вариант техник *bump mapping* или *normal mapping*. *Parallax mapping* используется для процедурного создания трехмерного описания текстурированной поверхности с использованием карт смещения (не путать с *Displacement mapping*) вместо непосредственного генерирования новой геометрии. Методику «*Parallax mapping*» условно можно назвать «2.5D», так как она позволяет добавлять трехмерную сложность в текстуры, не создавая реальные трехмерные графические структуры. Например, текстура каменной стены будет иметь визуальную объемность, хотя на самом деле геометрически она будет плоской. *Parallax mapping* был представлен Томомити Канэко (англ. *Tomomichi Kaneko*) в 2001 году. *Parallax mapping* полностью выполняется на графических процессорах видеокарты как пиксельный шейдер. От английского *shader* – *замещающая программа* – компьютерная программа, предназначенная для исполнения процессорами видеокарты (GPU). Шейдеры состояются на одном из специализированных языков программирования и компилируются в инструкции для GPU.

Parallax mapping осуществляется смещением текстурных координат так, чтобы поверхность казалась объемной. Главное отличие

parallax mapping от displacement mapping в том, что в нем все расчеты попиксельные, а не повершинные. Идея метода состоит в том, чтобы возвращать текстурные координаты той точки, где видовой вектор пересекает поверхность. Это требует просчета лучей (рейтрейсинг) для карты высот, но если она не имеет слишком резко изменяющихся значений («гладкая» или «плавная»), то можно обойтись аппроксимацией без использования рейтрейсинга. Если же в parallax mapping используется рейтрейсинг, то такой вариант называется «Parallax occlusion mapping».

Таким образом, parallax mapping хорош для поверхностей с плавно изменяющимися высотами, без просчета пересечений и больших значений смещения. Подобный простой алгоритм отличается от normal mapping всего тремя инструкциями пиксельного шейдера: две математические инструкции и одна дополнительная выборка из текстуры. После того как вычислена новая текстурная координата, она используется дальше для чтения других текстурных слоев: базовой текстуры, карты нормалей и т. п. Такой метод parallax mapping на современных графических процессорах почти так же эффективен, как обычное наложение текстур, а его результатом является более реалистичное отображение поверхности по сравнению с простым normal mapping».

Работа с текстурами

Для работы с текстурами в OpenGL используется следующий обобщенный алгоритм:

1. Загрузить битовое изображение в память, привязав к нему указатель.
2. Включить режим отображения текстур.
3. Выполнить отрисовку в режиме GL_QUADS, соотнеся координаты текстуры с координатами сцены.

Нужно отметить, что текстовая информация также отрисовывается с помощью текстур, то есть в OpenGL нет специальной функции

для вывода текста. Поэтому, если нам необходимо отобразить какое-либо текстовое сообщение, то в SharpGL это осуществляется комбинацией возможностей GDI+ и OpenGL. Рассмотрим для примера вывод красной надписи «Hello, world!» на черном фоне.

Первое, что нам необходимо сделать, – сгенерировать саму картинку-текстуру, сохранив ее в bmp-файл. Выполнить это можно кодом, представленным в листинге 3.26.

Листинг 3.26

```
Bitmap BmpImage = new Bitmap(256, 256);
Graphics gr = Graphics.FromImage(BmpImage);
gr.FillRectangle(Brushes.Black, 0, 0, 256, 256);
gr.DrawString("Hello, world!!!", new Font("Arial", 20), Brushes.Red, 0, 0);
BmpImage.Save("Hello.bmp");
```

Обратите внимание!

- *При работе с текстурами в OpenGL существует правило: их размеры должны быть равными степени двойки, то есть допустимы 16, 32, 64, 128, 256 и т. д.*

Теперь перейдем к работе с OpenGL. Так как для каждой текстуры нам необходимо хранить указатель, а также соотношение ее координат с координатами сцены, то для удобства работы определим следующую структуру данных, приведенную в листинге 3.27.

Листинг 3.27

```
public struct TextureStruct
{
    public uint TextureValue;
    public double[,] X_YText;
    public double[,] CoordX_YText;
}
```

Здесь TextureValue – число-указатель, по которому мы связываем координаты с картинкой в оперативной памяти. X_Ytext – координаты углов прямоугольника-текстуры ({0,0} – левый верхний, {0,1} – правый верхний, {1,1} – правый нижний, {1,0} – левый нижний). CoordX_YText – координаты углов прямоугольника-текстуры на сцене.

Следующим шагом нам нужно заполнить эти параметры. Пусть камера и перспектива у нас заданы следующим образом (листинг 3.28). Тогда для отображения надписи мы можем использовать координаты, задаваемые кодом листинга 3.29. Загрузка текстуры в память осуществляется кодом из листинга 3.30.

Листинг 3.28

```
GL.Perspective(90, 4 / 3, 10, 200);  
GL.LookAt(10, 10, 10, 0, 1, 0, 0, 1, 0);
```

Листинг 3.29

```
TextureStruct Textures = new TextureStruct();  
Textures.CoordX_YText = new double[4, 3]  
{ { 0, -0.1, 0 }, { 8, -0.1, 0 }, { 8, -0.1, 8 }, { 0, -0.1, 8 } };  
Textures.X_YText = new double[4, 2] { { 0, 0 }, { 1, 0 }, { 1, 1 }, { 0, 1 } };
```

Листинг 3.30

```
Bitmap BmpImage = new Bitmap(new Bitmap(«Hello.bmp»), 256, 256);  
Rectangle rect = new Rectangle(0, 0, BmpImage.Width, BmpImage.Height);  
BitmapData BmpData = BmpImage.LockBits(rect, ImageLockMode.ReadOnly,  
BmpImage.PixelFormat);  
GL.TexImage2D(OpenGL.GL_TEXTURE_2D, 0, (int)OpenGL.GL_RGBA,  
BmpImage.Width, BmpImage.Height, 0, OpenGL.GL_BGRA,  
OpenGL.GL_UNSIGNED_BYTE, BmpData.Scan0);  
BmpImage.UnlockBits(BmpData);
```

В данном случае (листинг 3.30) объект `BitmapData` хранит атрибуты растрового изображения, полученные методом `LockBits`. Этот метод переносит изображение в системную память и «замораживает» его там для выполнения дальнейших манипуляций. Параметр `rect` определяет размер рисунка, `ImageLockMode` указывает, какие операции с этой областью памяти допустимы, `PixelFormat` показывает, как следует интерпретировать записанные в память биты (в каком формате они записаны). В приведенном примере мы используем параметр, напрямую считанный у изображения. Как только все нужные манипуляции сделаны, область памяти «размораживается» методом `UnlockBits`. OpenGL начинает свою работу с метода `TexImage2D`. Его параметры говорят следующее. `OpenGL.GL_TEXTURE_2D` – загружается 2D-текстура. 0 – уровень отображения. `OpenGL.GL_RGBA` – внутренний формат записи. `BmpImage.Width`, `BmpImage.Height`, 0 – размер текстуры и границы. `OpenGL.GL_BGRA` – формат вывода, `OpenGL.GL_UNSIGNED_BYTE` – тип, в котором картинка хранится в памяти, `BmpData.Scan0` – указатель на область памяти, где «заморожена» картинка.

После загрузки текстуры мы должны «привязать» ее к координатам, назначить идентификатор и установить параметры. Привязка осуществляется кодом из листинга 3.31.

Листинг 3.31

```
GL.BindTexture(OpenGL.GL_TEXTURE_2D, Textures.TextureValue);
```

Обратите внимание!

- *Идентификатор будет назначен последней загруженной в память текстуре.*

Параметры текстуры устанавливаются специальным методом `TexParameter`. В качестве идентификаторов параметров допустимы следующие константы:

1. `GL_TEXTURE_MIN_FILTER` – определяет функцию минимизации, используемую, когда пиксели текстуры отображаются в область, большую, чем один элемент текстуры. Для данного идентификатора допустимы следующие параметры:

- `GL_NEAREST` – возвращает значение ближайшего элемента текстуры до центра текстурного пикселя.
- `GL_LINEAR` – возвращает средневзвешенное значение четырех текстурных элементов, которые находятся ближе всего к центру текстурного пикселя.
- `GL_NEAREST_MIPMAP_NEAREST` – выбирает *mip*-карту, которая наиболее точно соответствует размеру текстурного пикселя и использует критерий `GL_NEAREST`, чтобы получить значение. Согласно Википедии [1]: «MIP-текстурирование (англ. MIP mapping) – метод текстурирования, использующий несколько копий одной текстуры с разной детализацией».
- `GL_LINEAR_MIPMAP_NEAREST` – аналогично, но использует `GL_LINEAR` критерий.
- `GL_NEAREST_MIPMAP_LINEAR` – аналогично `GL_NEAREST_MIPMAP_NEAREST`, но возвращает две *mip*-карты. Окончательное значение – взвешенное из полученных двух.
- `GL_LINEAR_MIPMAP_LINEAR` – аналогично предыдущему, но используется критерий `GL_LINEAR`.

2. `GL_TEXTURE_MAG_FILTER` – определяет функцию увеличения текстуры, которая используется, когда пиксели отображаются в область меньшую или равную одному текстурному элементу. Допустимые параметры: `GL_NEAREST` и `GL_LINEAR`.

3. `GL_TEXTURE_WRAP_S` – устанавливает параметр разномножения для горизонтальной координаты текстуры. Может принимать следующие значения: `GL_CLAMP_TO_EDGE`, `GL_MIRRORED_REPEAT`, `GL_REPEAT`.

4. `GL_TEXTURE_WRAP_T` – аналогично для вертикальной координаты.

Для наших целей нам достаточно установить параметры, как показано в листинге 3.32.

Листинг 3.32

```
GL TexParameter(OpenGL.GL_TEXTURE_2D,  
OpenGL.GL_TEXTURE_MIN_FILTER, OpenGL.GL_LINEAR);  
GL TexParameter(OpenGL.GL_TEXTURE_2D,  
OpenGL.GL_TEXTURE_MAG_FILTER, OpenGL.GL_LINEAR);
```

Отрисовка текстуры выполняется так же, как для всех примитивов OpenGL. Код, выполняющий указанное, приведен в листинге 3.33. В результате на экране мы должны увидеть изображение с рисунка 3.28.

Листинг 3.33

```
GL.Enable(OpenGL.GL_TEXTURE_2D); GL.Begin(OpenGL.GL_QUADS);  
for (int i = 0; i < 4; i++){  
GL.TexCoord(Textures.X_YText[i, 0], Textures.X_YText[i, 1]);  
GL.Vertex(Textures.CoordX_YText[i, 0], Textures.CoordX_YText[i,  
1], Textures.CoordX_YText[i, 2]);}  
GL.End(); GL.Disable(OpenGL.GL_TEXTURE_2D);
```



Рисунок 3.28. Вывод текста

Обратите внимание!

- Если мы зададим массив координат текстуры следующим образом: `Textures.X_YText = new double[4, 2] { { 1, 1 }, { 1, 0 }, { 0, 0 }, { 0, 1 } }`, то на экране появится изображение, представленное на рисунке 3.29. Это говорит об ошибочном соотношении координат.

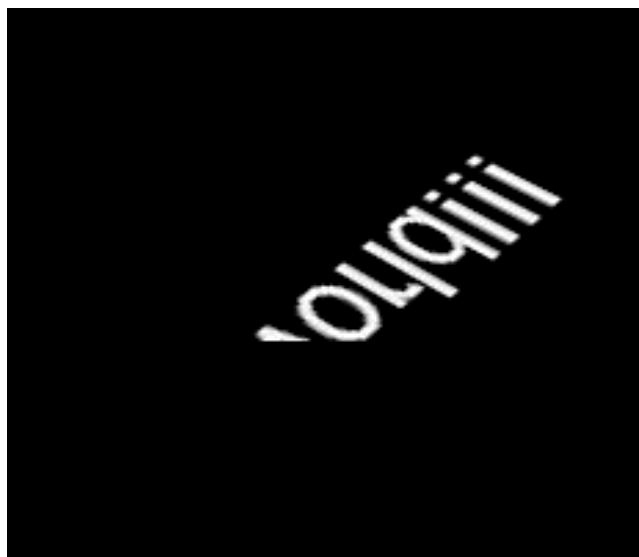


Рисунок 3.29. Ошибочное соотнесение координат

Сохранение сцены в bmp-файл

Мы рассмотрели способ отображения текстур в OpenGL. Теперь перейдем к решению обратной задачи, а именно к сохранению созданных с помощью OpenGL сцен в картинки формата bmp. Для реализации описанного мы используем метод чтения пикселей из буфера вывода. Обобщенный алгоритм может быть описан следующим образом:

1. Создать пустой Bitmap, по размеру равный области вывода сцены.
2. «Заморозить» под него некоторую область памяти с правами на запись.
3. Считать из буфера вывода пиксели и записать их в «замороженную» область.
4. Сохранить полученную картинку в файл.

Реализация третьего пункта будет содержать вызовы следующих методов.

Во-первых, `PushClientAttrib`. Этот метод укажет, какие группы переменных состояния клиента следует сохранить в стеке атрибутов клиента. В нашем случае метод примет параметр с именем `GL_CLI-`

ENT_PIXEL_STORE_BIT. Это будет означать, что мы будем работать с пикселями.

Во-вторых, PixelStore. Он устанавливает режим хранения пикселей, который будет влиять на работу метода чтения пикселей. Допустимы следующие константы-параметры:

- GL_PACK_SWAP_BYTES;
- GL_PACK_LSB_FIRST;
- GL_PACK_ROW_LENGTH;
- GL_PACK_IMAGE_HEIGHT;
- GL_PACK_SKIP_PIXELS;
- GL_PACK_SKIP_ROWS;
- GL_PACK_SKIP_IMAGES;
- GL_PACK_ALIGNMENT;
- GL_UNPACK_SWAP_BYTES;
- GL_UNPACK_LSB_FIRST;
- GL_UNPACK_ROW_LENGTH;
- GL_UNPACK_IMAGE_HEIGHT;
- GL_UNPACK_SKIP_PIXELS;
- GL_UNPACK_SKIP_ROWS;
- GL_UNPACK_SKIP_IMAGES;
- GL_UNPACK_ALIGNMENT.

Константы вида * PACK* отвечают за режим упаковки пикселей в память. * UNPACK* – за распаковку.

В-третьих, метод ReadBuffer. Он указывает, из какого буфера необходимо считывать пиксели.

В-четвертых, метод ReadPixels – считывает пиксели.

И, в-пятых, метод PopClientAttrib – извлекает сохраненные в стеке атрибуты клиента.

Таким образом, код сохранения сцены в bmp-файл будет иметь вид, представленный в листинге 3.34.

Листинг 3.34

```
Bitmap image = new Bitmap(openGLControl1.Width, openGLControl1.Height);
BitmapData imgData = image.LockBits(new Rectangle(0, 0, image.Width, im-
age.Height), ImageLockMode.WriteOnly, PixelFormat.Format32bppArgb);
GL.PushClientAttrib(OpenGL.GL_CLIENT_PIXEL_STORE_BIT);
GL.PixelStore(OpenGL.GL_PACK_ALIGNMENT, 4);
GL.ReadBuffer(OpenGL.GL_FRONT); GL.ReadPixels(0, 0, image.Width, im-
age.Height, OpenGL.GL_BGRA, OpenGL.GL_UNSIGNED_BYTE,
imgData.Scan0);
GL.PopClientAttrib(); image.UnlockBits(imgData); im-
age.RotateFlip(RotateFlipType.Rotate180FlipX); image.Save("save.bmp");
```

Обратите внимание!

- Строка кода *image.RotateFlip(RotateFlipType.Rotate180FlipX)* выполняет поворот изображения на 180 градусов относительно оси X. Без этой манипуляции в файл оно сохранится перевернутым.
- При работе с областями памяти могут возникать исключения, но даже в режиме отладки среда может их «не отловить», и приложение продолжит функционирование. Вместо картинки появится изображение с рисунка 3.30.

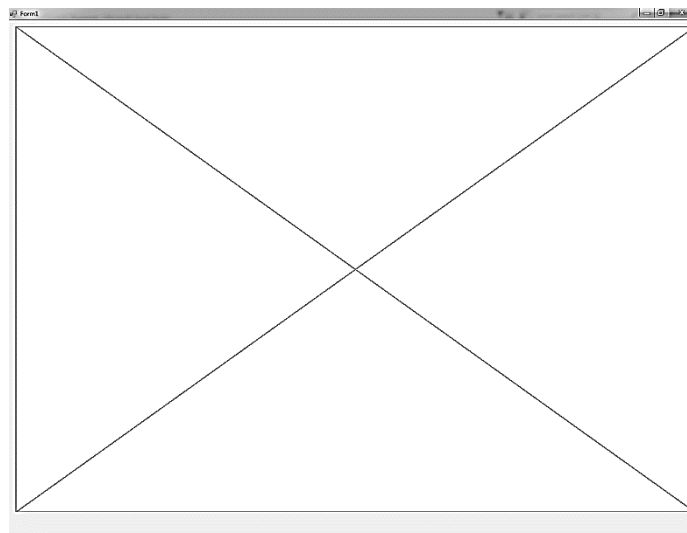


Рисунок 3.30. Некорректная работа программы

Видеоролики

Теперь рассмотрим способ сохранения созданных анимаций в файл формата avi, используя возможности библиотеки avifil32.dll. Материал данного пункта подготовлен по результатам изучения исходного кода проекта [6]. Указанный проект – полноценная библиотека работы с аудио- и видеопотоками. Здесь же рассмотрена только часть, касающаяся решения описанной задачи.

Библиотека avifil32.dll содержит необходимые функции для работы с видеопотоками и поставляется вместе с операционной системой Windows. Ее функционала вполне хватит для решения поставленной нами задачи, однако при работе с VisualStudio есть одна сложность: указанная библиотека разработана не для платформы .Net. Поэтому напрямую подключить ее к проекту нельзя и придется использовать технологию платформенного вызова, которая позволит импортировать из библиотеки необходимые нам функции и структуры данных. Поясним подробнее данную необходимость.

Платформа .Net поддерживает многоязыковую разработку приложений благодаря универсальному межъязыковому интерфейсу Common Language Infrastructure, или CLI, а также благодаря среде выполнения программ – Common Language Runtime (CLR). Первый поддерживает разработку программных компонентов на различных языках программирования, а вторая отвечает за управление памятью, управление типами данных, развертывание приложений и организацию межъязыкового взаимодействия. Если код выполняется под управлением CLR, то он называется управляемым. Иначе – неуправляемым. Для объединения управляемого и неуправляемого кодов используется технология платформенного вызова. То есть, если вам необходимо в .Net-проекте использовать функции Win32 API, компоненты COM или какие-либо библиотеки, не поддерживаемые платформой, – это она.

Платформенный вызов – служба, позволяющая коду, реализованному на платформе .Net, вызывать функции в библиотеках, разра-

ботанных не для этой платформы. Например, функции WinAPI, или необходимой нам avifil32.dll. Алгоритм работы с этой службой следующий:

1. Найти вызываемую функцию в библиотеке;
2. Создать класс для сохранения вызываемой функции;
3. Объявить прототип этой функции в управляемом коде;
4. Вызвать функцию.

Когда мы выполняем пункты 2 и 3, то фактически мы создаем .Net-обертку для неуправляемого кода. Например, используемый нами SharpGL – обертка для функций OpenGL, написанная с использованием технологии платформенного вызова.

Эта служба выполняет следующие действия:

1. Находит библиотеку, содержащую функцию;
2. Загружает ее в память;
3. Находит адрес функции в памяти и заносит в стек ее параметры;
4. Управление передается неуправляемой функции.

Таким образом, чтобы использовать функции из неподключенной к проекту библиотеки, мы определяем пустой класс, в котором перечисляем заголовки импортируемых функций, объявленных с атрибутом DllImport (атрибут указывается в квадратных скобках перед импортируемой функцией). Затем в основном коде вызываем их как обычно.

Обратите внимание!

- *Для использования технологии платформенного вызова необходимо подключить пространство имен `System.Runtime.InteropServices`.*

Тогда, общий алгоритм решения задачи будет следующим:

1. При загрузке OpenGL-контроля открываем видеопоток;
2. Сохраняем каждый кадр в файл формата bmp со случайным временным именем и добавляем его в созданный видеопоток;

3. При закрытии формы экспортируем созданный видеопоток в файл, используя стандартные кодеки.

Для создания видеопотока нам необходимо импортировать из библиотеки следующие функции: AVIFileInit, AVIFileOpen, AVIFileCreateStream, а также структуру AVISTREAMINFO. Для добавления кадра в поток нам понадобится функция AVIStreamWrite. Для сохранения файла мы будем использовать функции AVISaveOptions, AVISaveOptionsFree, AVISaveV, а также структуру RECT, AVISTREAMINFO, AVICOMPRESSOPTIONS и аналогичный ей класс. Кроме того, мы объявим несколько констант, используемых в импортируемых функциях.

Обратите внимание!

- *Для импорта структур мы будем использовать атрибут StructLayout. Он управляет расположением данных в памяти. В нашем варианте структуры всегда будут располагаться в ячейках памяти последовательно.*

Для примера разработаем тестовое приложение, где по кнопке будет открываться диалог выбора bmp-файлов, которые затем запишутся в avi-файл.

Первое, что нам необходимо сделать, – импортировать функции и структуры данных из avifil32.dll. Код импорта будет следующим (листинг 3.35). Здесь и далее предполагается, что код импорта прописывается в теле класса Form. Вынести его в новый класс предлагается самостоятельно.

Листинг 3.35

```
public static readonly int streamtypeVIDEO = mmioFOURCC('v', 'i', 'd', 's');
    public static int RGBQUAD_SIZE = 4;
    // show KeyFrame Every box
    public const UInt32 ICMF_CHOOSE_KEYFRAME = 0x0001;
    public const UInt32 ICMF_CHOOSE_DATARATE = 0x0002; // show DataRate box
```

Продолжение листинга 3.35

```
[StructLayout(LayoutKind.Sequential, Pack = 1)]
public struct BITMAPINFO {
    public BITMAPINFOHEADER bmiHeader;
    [MarshalAs(UnmanagedType.ByValArray, SizeConst = 256)]
    public RGBQUAD[] bmiColors; }

[StructLayout(LayoutKind.Sequential, Pack = 1)]
public struct BITMAPINFOHEADER {
    public Int32 biSize;
    public Int32 biWidth;
    public Int32 biHeight;
    public Int16 biPlanes;
    public Int16 biBitCount;
    public Int32 biCompression;
    public Int32 biSizeImage;
    public Int32 biXPelsPerMeter;
    public Int32 biYPelsPerMeter;
    public Int32 biClrUsed;
    public Int32 biClrImportant; }

public static Int32 mmioFOURCC(char ch0, char ch1, char ch2, char ch3)
{
    return ((Int32)(byte)(ch0) | ((byte)(ch1) << 8) |
        ((byte)(ch2) << 16) | ((byte)(ch3) << 24)); }

[DllImport("winmm.dll", EntryPoint = "mmioStringToFOURCCA")]
public static extern int mmioStringToFOURCC(String sz, int uFlags);

[DllImport("avifil32.dll")]
public static extern void AVIFileInit();

[DllImport("avifil32.dll", PreserveSig = true)]
public static extern int AVIFileOpen(
    ref int ppfile, String szFile,
    int uMode,
    int pclsidHandler);
```


Продолжение листинга 3.35

```
[DllImport("avifil32.dll")]
    public static extern int AVIFileCreateStream(
        int pfile,
        out IntPtr ppavi,
        ref AVISTREAMINFO ptr_streaminfo);

[DllImport("avifil32.dll")]
    public static extern int AVIStreamWrite(
        IntPtr aviStream, Int32 lStart, Int32 lSamples,
        IntPtr lpBuffer, Int32 cbBuffer, Int32 dwFlags,
        Int32 dummy1, Int32 dummy2);

[DllImport("avifil32.dll")]
    public static extern bool AVISaveOptions(
        IntPtr hwnd,
        UInt32 uiFlags,
        Int32 nStreams,
        ref IntPtr ppavi,
        ref AVICOMPRESSOPTIONS_CLASS plpOptions);

[DllImport("avifil32.dll")]
    public static extern long AVISaveOptionsFree( int nStreams,
        ref AVICOMPRESSOPTIONS_CLASS plpOptions);

[DllImport("avifil32.dll")]
    public static extern int AVISaveV(String szFile, Int16 empty, Int16
lpfnCallback,
    Int16 nStreams, ref IntPtr ppavi,
    ref AVICOMPRESSOPTIONS_CLASS plpOptions );

[StructLayout(LayoutKind.Sequential, Pack = 1)]
    public struct RECT
    {
        public UInt32 left;  public UInt32 top;  public UInt32 right;
        public UInt32 bottom;
    }
```

```
[StructLayout(LayoutKind.Sequential, Pack = 1)]
public struct RGBQUAD    {
    public byte rgbBlue; public byte rgbGreen; public byte rgbRed;
    public byte rgbReserved; }
```

```
[StructLayout(LayoutKind.Sequential, Pack = 1)]
public struct AVISTREAMINFO {
    public Int32 fccType; public Int32 fccHandler;
    public Int32 dwFlags; public Int32 dwCaps;
    public Int16 wPriority; public Int16 wLanguage;
    public Int32 dwScale; public Int32 dwRate;
    public Int32 dwStart;
    public Int32 dwLength;
    public Int32 dwInitialFrames;
    public Int32 dwSuggestedBufferSize;
    public Int32 dwQuality;
    public Int32 dwSampleSize;
    public RECT rcFrame;
    public Int32 dwEditCount;
    public Int32 dwFormatChangeCount;
```

```
[MarshalAs(UnmanagedType.ByValArray, SizeConst = 64)]
public UInt16[] szName; }
```

```
[StructLayout(LayoutKind.Sequential, Pack = 1)]
public class AVICOMPRESSOPTIONS_CLASS
{
    public UInt32 fccType;
    public UInt32 fccHandler;
    public UInt32 dwKeyFrameEvery;
    public UInt32 dwQuality;
    public UInt32 dwBytesPerSecond;
    public UInt32 dwFlags;
    public IntPtr lpFormat;
    public UInt32 cbFormat;
    public IntPtr lpParms;
```

```

    public UInt32 cbParms;
    public UInt32 dwInterleaveEvery;

    public AVICOMPRESSOPTIONS ToStruct()
    {
        AVICOMPRESSOPTIONS returnVar = new AVICOMPRESSOPTIONS();
        returnVar.fccType = this.fccType;
        returnVar.fccHandler = this.fccHandler;
        returnVar.dwKeyFrameEvery = this.dwKeyFrameEvery;
        returnVar.dwQuality = this.dwQuality;
        returnVar.dwBytesPerSecond = this.dwBytesPerSecond;
        returnVar.dwFlags = this.dwFlags;
        returnVar.lpFormat = this.lpFormat;
        returnVar.cbFormat = this.cbFormat;
        returnVar.lpParms = this.lpParms;
        returnVar.cbParms = this.cbParms;
        returnVar.dwInterleaveEvery = this.dwInterleaveEvery;
        return returnVar;
    }
}

[DllImport("avifil32.dll")]
public static extern int AVIStreamSetFormat(
    IntPtr aviStream, Int32 IPos,
    //ref BITMAPINFOHEADER lpFormat,
    ref BITMAPINFO lpFormat,
    Int32 cbFormat);

[StructLayout(LayoutKind.Sequential, Pack = 1)]
public struct AVICOMPRESSOPTIONS
{
    public UInt32 fccType;
    public UInt32 fccHandler;
    public UInt32 dwKeyFrameEvery;
    public UInt32 dwQuality;
    public UInt32 dwBytesPerSecond;

```

Окончание листинга 3.35

```
    public UInt32 dwFlags;
    public IntPtr lpFormat;
    public UInt32 cbFormat;
    public IntPtr lpParms;
    public UInt32 cbParms;
    public UInt32 dwInterleaveEvery;
}
public const int OF_WRITE = 1;
public const int OF_CREATE = 4096;
```

Теперь перейдем к коду кнопки, в котором будем использовать импортированные функции. Первое, что нам необходимо сделать, – открыть файл для временной записи видеопотока. Дело в том, что итоговый файл у нас должен быть сохранен с использованием сжатия, чтобы он мог воспроизводиться в плеерах. Однако добавлять картинки-кадры можно только в несжатый поток. Поэтому нам придется использовать временный файл. Создается он следующим кодом (листинг 3.36).

Листинг 3.36

```
//Инициализация библиотеки
AVIFileInit();
//переменная для хранения результата, выдаваемого импортируемой функцией
int result;
//указатель на файл
int aviFile = 0;
//открытие файла с параметрами «создать или записать»
result = AVIFileOpen(ref aviFile, "1.avi", OF_WRITE | OF_CREATE, 0);
//если функция вернула результат отличный от 0, значит произошла какая-то
ошибка. //Возвращаем ее шестнадцатеричный код, чтобы можно было сориен-
тироваться по //документации.
if (result != 0) { throw new Exception("Error in CreateFile: " + result.ToString("X"));}
```

После того как мы создали файл, нам необходимо создать привязанный к нему видеопоток, причем с определенными настройками.

Для определения части настроек мы заранее создадим картинку, которая будет первым кадром («0.bmp»), и считаем с нее некоторые данные. Выполняется это следующим кодом, как показано в листинге 3.37.

Листинг 3.37

```
//определение параметра «скорость кадров» для видео
double frameRate = 60;
//загрузка опорной картинки
Bitmap FirstFrame = new Bitmap(Image.FromFile("0.bmp"));
//считывание цветовой палитры опорной картинки
ColorPalette template = FirstFrame.Palette;
//конвертация палитры в структуру данных, требуемую для функции
//из библиотеки avifil32.dll

RGBQUAD[] palette = new RGBQUAD[template.Entries.Length];
for (int i = 0; i < palette.Length; i++) {
    if (i < template.Entries.Length) {
        // R – красная компонента, G – зеленая, B – синяя
        palette[i].rgbRed = template.Entries[i].R;
        palette[i].rgbGreen = template.Entries[i].G;
        palette[i].rgbBlue = template.Entries[i].B; }
    else {
        palette[i].rgbRed = 0;
        palette[i].rgbGreen = 0;
        palette[i].rgbBlue = 0; }}

//получение информации об опорной картинке
BitmapData bmpData = FirstFrame.LockBits(new Rectangle(
0, 0, FirstFrame.Width, FirstFrame.Height),
ImageLockMode.ReadOnly, FirstFrame.PixelFormat);
//задание размера кадра
int frameSize = bmpData.Stride * bmpData.Height;
//задание размера изображения
int width = FirstFrame.Width;
int height = FirstFrame.Height;
```

Продолжение листинга 3.37

```
//определение формата кодировки изображения
String formatName = FirstFrame.PixelFormat.ToString();
if (formatName.Substring(0, 6) != "Format"){
throw new Exception("Unknown pixel format: " + formatName); }
formatName = formatName.Substring(6, 2);

//установка параметра «бит на пиксель» исходя из кодировки
Int16 countBitsPerPixel = 0;
if (Char.IsNumber(formatName[1])){
//16, 32, 48
countBitsPerPixel = Int16.Parse(formatName);}
else {
//4, 8
countBitsPerPixel = Int16.Parse(formatName[0].ToString());}
FirstFrame.UnlockBits(bmpData);

//указатель на видеопоток
IntPtr aviStream;

//установка параметров масштаба и frameRate
int scale = 1;
double rate = frameRate;

//корректировка параметров. Условие остановки – framerate должно
//занимать определенное число байтов в памяти
while (frameRate != (long)frameRate)
{
frameRate = frameRate * 10;
scale *= 10;
}
rate = frameRate;
//определение счетчика кадров (указатель текущего кадра)
int countFrames = 0;
//определение структуры данных для импортируемой функции
AVISTREAMINFO strhdr = new AVISTREAMINFO();
```

Продолжение листинга 3.37

```
strhdr.fccType = mmioStringToFOURCC("vids", 0);
strhdr.fccHandler = mmioStringToFOURCC("CVID", 0);
strhdr.dwFlags = 0;
strhdr.dwCaps = 0;
strhdr.wPriority = 0;
strhdr.wLanguage = 0;
strhdr.dwScale = (int)scale;
strhdr.dwRate = (int)rate; // Frames per Second
strhdr.dwStart = 0; strhdr.dwLength = 0;
strhdr.dwInitialFrames = 0;
strhdr.dwSuggestedBufferSize = frameSize; //height_ * stride_;
strhdr.dwQuality = -1; //default
strhdr.dwSampleSize = 0; strhdr.rcFrame.top = 0; strhdr.rcFrame.left = 0;
strhdr.rcFrame.bottom = (uint)height; strhdr.rcFrame.right = (uint)width;
strhdr.dwEditCount = 0; strhdr.dwFormatChangeCount = 0; strhdr.szName = new
UInt16[64];

// создание потока
result = AVIFileCreateStream(aviFile, out aviStream, ref strhdr);

if (result != 0)
{
    throw new Exception("Error in CreateStream: " + result.ToString("X"));
}

//выравнивание исходной картинки, чтобы она не оказалась перевернутой
FirstFrame.RotateFlip(RotateFlipType.RotateNoneFlipY);

//создание структуры для установки формата видеопотока
BITMAPINFO bi = new BITMAPINFO();
bi.bmiHeader.biWidth = width;
bi.bmiHeader.biHeight = height;
    bi.bmiHeader.biPlanes = 1;
bi.bmiHeader.biBitCount = countBitsPerPixel; bi.bmiHeader.biSizeImage =
frameSize;
```

Окончание листинга 3.37

```
//получение размера структуры данных в байтах
bi.bmiHeader.biSize = Marshal.SizeOf(bi.bmiHeader);
//корректировка параметров при определенной кодировке изображения
if (countBitsPerPixel < 24){ bi.bmiHeader.biClrUsed = palette.Length;
bi.bmiHeader.biClrImportant = palette.Length;
bi.bmiColors = new RGBQUAD[palette.Length];
palette.CopyTo(bi.bmiColors, 0);
bi.bmiHeader.biSize += bi.bmiColors.Length * RGBQUAD_SIZE;}
//установка формата для видеопотока
result = AVIStreamSetFormat(aviStream, countFrames, ref bi, bi.bmiHeader.biSize);
if (result != 0) { throw new Exception("Error in SetFormat: " + result.ToString("X")); }
```

После того как мы создали видеопоток с правильными параметрами, мы можем переходить к добавлению кадров. Для начала добавим наш первый кадр, записав следующий код, как показано в листинге 3.38.

Листинг 3.38

```
// «заморозка» картинки в памяти и получение указателя на нее
BitmapData bmpDat = FirstFrame.LockBits(
new Rectangle(
0, 0, FirstFrame.Width, FirstFrame.Height),
ImageLockMode.ReadOnly, FirstFrame.PixelFormat);
//запись битов картинки в поток через полученный указатель
//в позицию текущего кадра
result = AVIStreamWrite(aviStream, countFrames, 1, bmpDat.Scan0,
(Int32)(bmpDat.Stride * bmpDat.Height), 0, 0, 0);
if (result != 0)
{ throw new Exception("Exception in Write: " + result.ToString());}
FirstFrame.UnlockBits(bmpDat);
//сдвиг указателя на текущий кадр на следующую позицию
countFrames++;
```

Теперь нам необходимо добавить в поток и все остальные кадры. Мы предполагаем, что они лежат в той же папке, что и exe-файл

нашей программы. Тогда создаем объект типа OpenFileDialog, назначаем ему возможность многофайлового выбора, начальную директорию – директорию запуска приложения, а также фильтр на расширение файла. После выбора файлов пробегаем в цикле по массиву FileNames, добавляя каждую картинку в поток, также как мы добавляли первый кадр, и не забывая корректировать ее поворот методом RotateFlip. Добавление картинок в поток вам предлагается реализовать самостоятельно, установка же параметров объекта OpenFileDialog и его открытие выполняется следующим кодом (листинг 3.39).

Листинг 3.39

```
OpenFileDialog o = new OpenFileDialog();  
o.InitialDirectory = Application.StartupPath;  
o.Filter = "Картинки|*.bmp";  
o.Multiselect = true;  
o.ShowDialog();
```

Теперь нам осталось лишь сохранить поток в файл. Выполняется это кодом из листинга 3.40.

Листинг 3.40

```
AVICOMPRESSOPTIONS_CLASS opts = new AVICOMPRESSOPTIONS_CLASS();  
  
opts.fccType = (uint)streamtypeVIDEO;  
opts.lpParms = IntPtr.Zero;  
opts.lpFormat = IntPtr.Zero;  
  
AVISaveOptions(IntPtr.Zero, ICMF_CHOOSE_KEYFRAME |  
ICMF_CHOOSE_DATARATE, 1, ref aviStream, ref opts);  
AVISaveOptionsFree(1, ref opts);  
  
AVISaveV("2.avi", 0, 0, 1, ref aviStream, ref opts);
```

Обратите внимание!

- *Файлы для видеопотока должны каждый раз создаваться заново, иначе возможна некорректная запись.*

Когда вызывается метод `SeveOptions`, на экране появляется диалоговое окно, представленное на рисунке 3.31, в котором можно выбрать кодек для сжатия, а также установить такие параметры, как скорость передачи, качество сжатия и наличие опорных кадров.

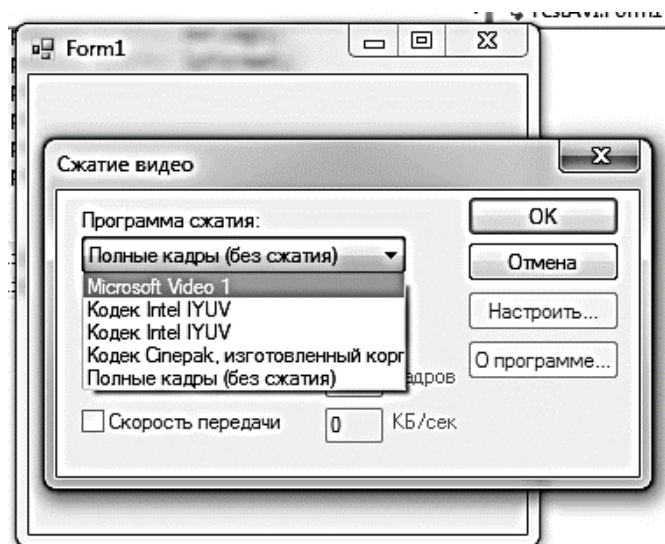


Рисунок 3.31. Параметры сохранения видео

В итоге в папке с вашим проектом появятся два файла (рисунок 3.32).

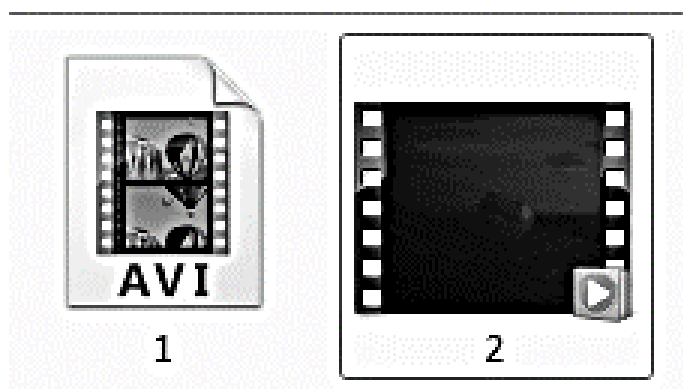


Рисунок 3.32. Результат работы кода

Как мы видим, файл 2.avi распознается как воспроизводимое видео, а 1.avi – как некий видеофайл, который воспроизвести штатными программами нельзя.

Мы рассмотрели основы работы с технологией OpenGL через .Net-обертку SharpGL. На этом наше знакомство с основными алгоритмами и технологиями, используемыми в программировании игр, заканчивается. Предлагаем выполнить несколько заданий для закрепления пройденного материала.

ЗАДАНИЯ К ТЕМЕ 3.3

1. Оптимизируйте код для работы с видеопотоком, отделите логику от интерфейса.
2. Создайте сцену: кирпичный домик с черепичной крышей, стоящий на траве на фоне закатного неба. Сохраните созданную сцену в bmp-файл. Добавьте к полученной сцене случайное освещение. Сохраните полученную анимацию в видеофайл.
3. Реализуйте анимированную заставку к игре «Морской бой».
4. Реализуйте анимацию: грузовик едет по дороге. Сохраните полученную анимацию в видеофайл.
5. Реализуйте игру «Линии» с анимированной заставкой и выводом текстовых сообщений.

ЗАКЛЮЧЕНИЕ

Мы рассмотрели основные структуры данных, алгоритмы и технологии, используемые в разработке игр. Представленный в книге материал преследовал цель познакомить читателя с «начинкой» игровых приложений, а также осветить основной трек изучения программирования через разработку игр и познакомить с основными технологиями работы с графикой, совместимыми с платформой .Net и языком C#.

Приведенные в книге игры предоставляют широкое поле для тренировки навыка разработки алгоритмов средней степени сложности. Если же читателю интересно продвинуться дальше в этом направлении, то можно разработать игры «Тетрис» и «Убеги от врагов в лабиринте». Сложность последней обуславливается динамическим запуском волнового алгоритма: «враги» должны пересчитывать свои пути при изменении положения героя на карте.

Как можно заметить, возможности рассмотренных нами технологий весьма велики, однако их удобство использования в разработке сложных игр далеко от идеала. Поэтому все чаще для создания игр используются различные движки. Согласно Википедии [1]: «Игровой движок – центральный программный компонент компьютерных и видеоигр или других интерактивных приложений с графикой, обрабатываемой в реальном времени. Он обеспечивает основные технологии, упрощает разработку и часто дает игре возможность запускаться на нескольких платформах». Одним из самых популярных движков для создания 2D- и 3D-игр в настоящее время является Unity 3D. Однако многообразие его возможностей требует объемного и детального описания, поэтому в данной книге он не представлен – для него необходимо отдельное полноценное издание, работа над которым автором планируется в будущем.

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. [Электронный ресурс]: Материалы свободной энциклопедии «Википедия». – URL: <http://ru.wikipedia.org/wiki/>, (режим доступа – свободный), (дата обращения: 28.03.2017).
2. Воронина, В. В. Компьютерная графика : методические указания к лабораторным работам / В. В. Воронина. – Ульяновск : УлГТУ, 2014. – 37 с.
3. Шишкин, В. В. Графический растровый редактор Gimp : учебное пособие / В. В. Шишкин, О. Ю. Шишкина, З. В. Степчева. – Ульяновск : УлГТУ, 2010. – 119 с.
4. [Электронный ресурс]: Материалы сайта pixabay.com. – URL: <http://pixabay.com>, (режим доступа – свободный), (дата обращения: 28.03.2017).
5. [Электронный ресурс]: Материалы MSDN. – URL: <https://msdn.microsoft.com>, (режим доступа – свободный), (дата обращения: 28.03.2017).
6. [Электронный ресурс]: Материалы сайта CodeProject: “Wrapper for AviFile-library”. – URL: <https://www.codeproject.com/Articles/7388/A-Simple-C-Wrapper-for-the-AviFile-Library>, (режим доступа – свободный), (дата обращения: 28.03.2017).
7. Ватсон, Б. C# 4.0 на примерах / Б. Ватсон. – СПб. : БХВ-Петербург, 2011. – 608 с. : ил.
8. [Электронный ресурс]: Материалы Сайта khronos.org: «OpenGL Reference pages». – URL: <https://www.khronos.org/registry/OpenGL-Refpages/>, (режим доступа – свободный), (дата обращения: 28.03.2017).
9. [Электронный ресурс]: Материалы Сайта professorweb.ru. – URL: <https://professorweb.ru>, (режим доступа – свободный), (дата обращения: 28.03.2017).

Учебное электронное издание

ВОРОНИНА Валерия Вадимовна

**ПРОГРАММИРОВАНИЕ ИГР:
АЛГОРИТМЫ И ТЕХНОЛОГИИ**

Учебное пособие

Редактор Н. А. Евдокимова

ЛР №020640 от 22.10.97.

ЭИ № 1027. Объем данных 8,8 Мб.

Печатное издание

Подписано в печать 31.10.2017.

Усл. печ. л. 17,90. Формат 60×84/16.

Тираж 120 экз. Заказ 5.

Ульяновский государственный технический университет,
432027, г. Ульяновск, ул. Сев. Венец, д. 32.
ИПК «Венец», УлГТУ, 432027, г. Ульяновск, ул. Сев. Венец, д. 32.
Тел.: (8422) 778-113
E-mail: venec@ulstu.ru
venec.ulstu.ru