Министерство образования и науки Российской Федерации

Федеральное государственное бюджетное образовательное учреждение
высшего профессионального образования
«Владимирский государственный университет
имени Александра Григорьевича и Николая Григорьевича Столетовых»

И. Е. ЖИГАЛОВ
И. А. НОВИКОВ

ПРОГРАММИРОВАНИЕ КОМПЬЮТЕРНОЙ ГРАФИКИ

Учебное пособие

Владимир 2014

УДК 681.3
ББК 32.973-018.2
Ж68

Рецензенты:

Доктор технических наук, профессор
кафедры основ нанотехнологий и теоретической физики
Владимирского государственного университета
имени Александра Григорьевича и Николая Григорьевича Столетовых
М. В. Руфицкий

Кандидат экономических наук, профессор
зав. кафедрой инновационного предпринимательства
Московского государственного технического университета им. Н. Э. Баумана
С. Р. Борисов

Печатается по решению редакционно-издательского совета ВлГУ

Жигалов, И. Е.

Ж68 Программирование компьютерной графики : учеб. пособие / И. Е. Жигалов, И. А. Новиков ; Владим. гос. ун-т им. А. Г. и Н. Г. Столетовых. – Владимир : Изд-во ВлГУ, 2014. − 96 с.

ISBN 978-5-9984-0437-5

Содержит теоретический материал и практические задания по темам, связанным с освоением Visual C# и изучением графической библиотеки OpenGL.

Предназначено для студентов направлений 230400 – Информационные системы и технологии и 231000 – Программная инженерия при изучении дисциплины «Программирование компьютерной графики».

Рекомендовано для формирования профессиональных компетенций в соответствии с ФГОС 3-го поколения.

Ил. 57. Табл. 2. Библиогр.: 8 назв.

УДК 681.3
ББК 32.973-018.2

ISBN 978-5-9984-0437-5

ВВЕДЕНИЕ

Для решения задач отображения сложных графических объектов требуется разработка эффективных методов и алгоритмов обработки графической информации на этапах ввода, кодирования, преобразования и формирования изображений. Все это составляет набор основных задач компьютерной графики.

В пособии рассматриваются типичные для компьютерной графики способы и приемы, ориентированные на расчет узловых точек изображения в пространстве и формирование объектов на их основе. Пособие помогает при изучении конкретных задач, связанных с получением компьютерных изображений на базе графических средств среды программирования *VisualC#* и включает цикл работ по освоению библиотеки *OpenGL* и изучению алгоритмов компьютерной графики.

Каждая тема пособия содержит самостоятельный материал с текстами программ, а весь цикл тем в целом завершается построением одной законченной программы, управляемой графическим меню и формирующей различные изображения графических объектов с использованием изучаемых методов и алгоритмов. Пособие охватывает далеко не все разделы компьютерной графики, однако оно дает основу для дальнейшего углубленного изучения и практического использования этих средств при создании прикладного программного обеспечения.

Тема 1. C#: РАЗРАБОТКА КОНСОЛЬНОГО ПРИЛОЖЕНИЯ

Цель изучения темы. Освоение принципов программирования в среде *Microsoft Visual C#* для операционной системы *Windows* с использованием *Microsoft . NET Framework*, получение практических навыков построения консольных приложений в *Visual C#.*

1.1. О *Microsoft .NET Framework*

Microsoft .NET Framework – это предложенная компанией *Microsoft* программная технология, основной задачей которой является предоставление разработчику набора удобных средств для разработки как простых программ, так и *web*-приложений.

Microsoft . NET Framework вобрал в себя все самое лучшее из современных средств программирования: удобные возможности синтаксиса *C++*, удобства объектной модели *Java*, простота *C++ Builder* в плане построения оконных приложений и методов для доступа к переменным, безопасный код.

Хотя скорости работы идентично реализованных алгоритмов на языках *C#* и *C++* различаются примерно в 2 раза (в пользу *C++*), при использовании *unsafe* (небезопасного) кода, *C#* практически догоняет по скорости выполнения *C++*, недобирая всего несколько процентов.

Подобно технологии *Java* (объектно-ориентированный язык программирования, разработанный компанией *Sun Microsystems*) среда разработки *.NET* из исходного кода программы создаёт байт-код, предназначенный для исполнения виртуальной машиной − программной или аппаратной средой, исполняющей некоторый код (например, байт-код или машинный код реального процессора).

Входной язык этой машины в *.NET* называется *MSIL (Microsoft Intermediate Language)*, или *CIL (Common Intermediate Language),* или просто *IL*. Применение байт-кода предназначено для получения кроссплатформенности на уровне скомпилированного текста программы, а не только на уровне исходного как, например, в языке *C*. С помощью *JIT*-компилятора (*just in time*, компиляция на лету) непосредственно перед запуском сборки

в среде исполнения *CLR* происходит преобразование байт-кода в машинные коды целевого процессора. Для *native*-сборки (компиляция в родной для ОС код) можно выполнить компиляцию с помощью утилиты *NGen.exe*, которая поставляется вместе с *.Net Framework*.

Часто разработчикам приходится писать программы на разных языках программирования. Библиотека классов *.NeT Framerowk* позволяет разработчикам использовать один программный интерфейс для всех функциональных средств *CLR*. Универсальность библиотеки классов *.net* − одна из ее сильнейших сторон.

1.2. Основы синтаксиса языка C#

Синтаксис языка *C#* очень похож на синтаксис языка *C++*. В качестве первого примера создадим традиционное приложение − *Hello World*. Все примеры будут приводиться на *MS Visual Studio 2008*, *SP1* с русскоязычным интерфейсом, который более понятен для начинающих программистов, хотя рекомендуется использовать английскую версию. Откройте редактор *MS Visual Studio*. Заполнение кода в *Visual Studio* стало крайне удобным благодаря отлично реализованной системе подсказок, поэтому набор кода не должен составить никаких проблем.

Запустите *MS Visual Studio*, после чего создайте новый проект, используя меню «Файл», как показано на рис. 1.1 (Файл -> Создать -> Проект…):

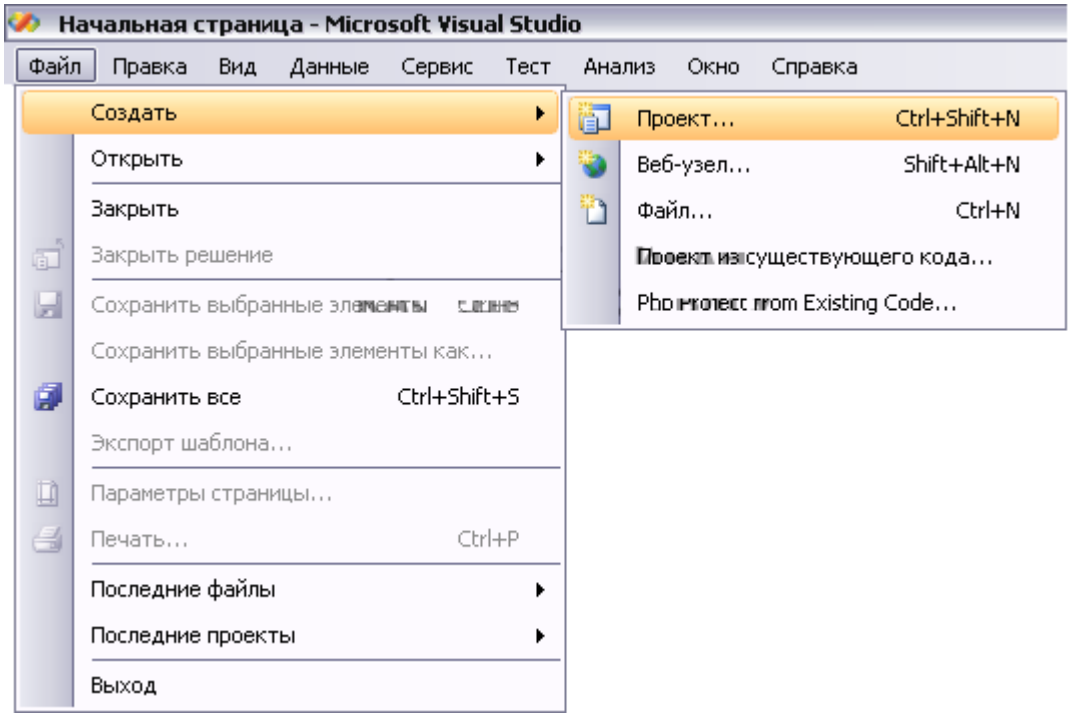


Рис. 1.1

Выберите проекты *Visual C#*. В шаблонах отметьте Консольное приложение и введите имя для нашего проекта: *Hello World* (рис. 1.2). По умолчанию все проекты сохраняются в папке *«Мои документы»* текущего пользователя *\Visual Studio 2008\Projects*.

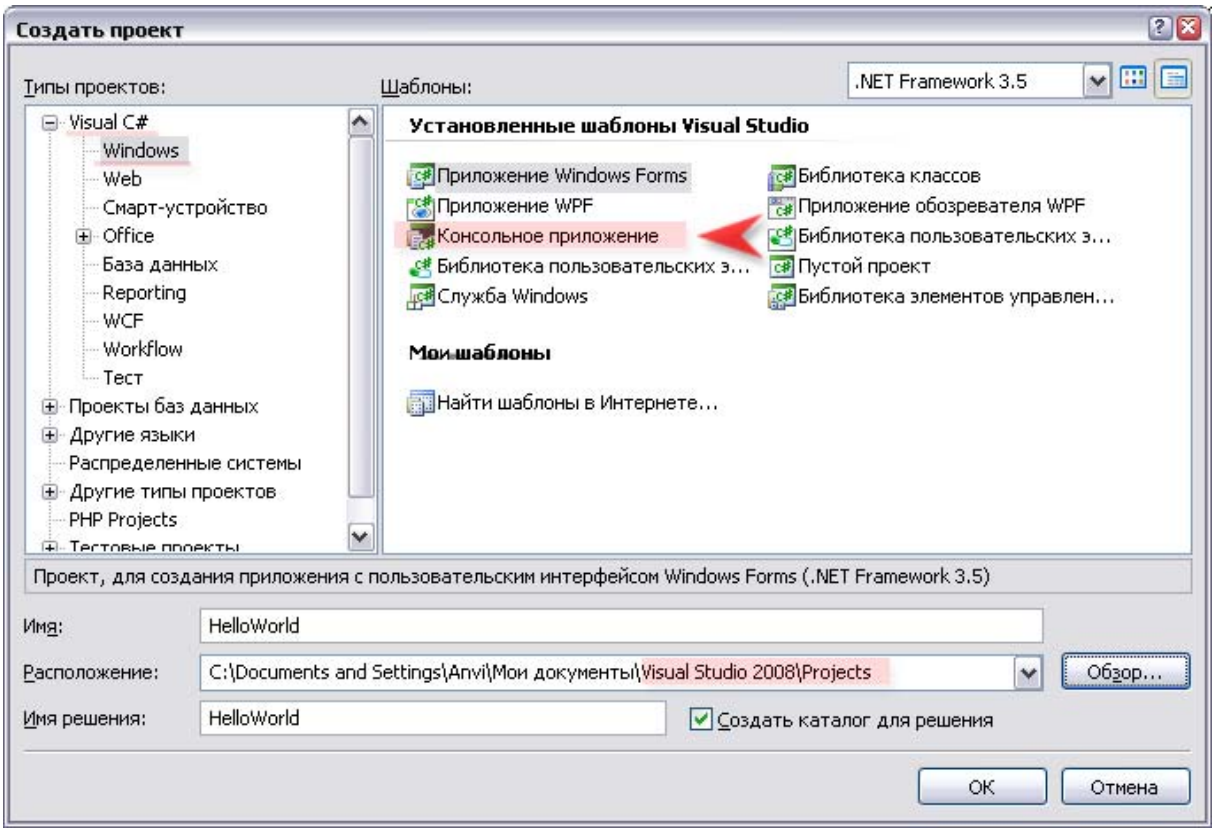


Рис. 1.2

В открывшемся окне редактора будет код шаблона простейшего приложения *.net* на языке *C#*.

В него нам необходимо добавить две строки кода: первая будет выводить на экран сообщение "*Hello World*", вторая ожидать строку, введенную пользователем, после которой программа завершится. Ожидание ввода нам необходимо, чтобы прочитать выведенные программой строки; иначе мы просто не успеем это сделать: программа выполнит код и завершится.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace HelloWorld
{
```

```
class Program
{
static void Main(string[] args)
{
// выводим текст в консоль
System.Console.Write("Hello world again");
// ждем ввода строки от пользователя,
// после которого завершится выполнение программы
String str = System.Console.ReadLine();
}
}
}
```

После набора кода необходимо нажать *F*5 и подтвердить необходимость компиляции.

Программа будет запущена, после чего вы сможете увидеть результат ее работы (рис. 1.3).

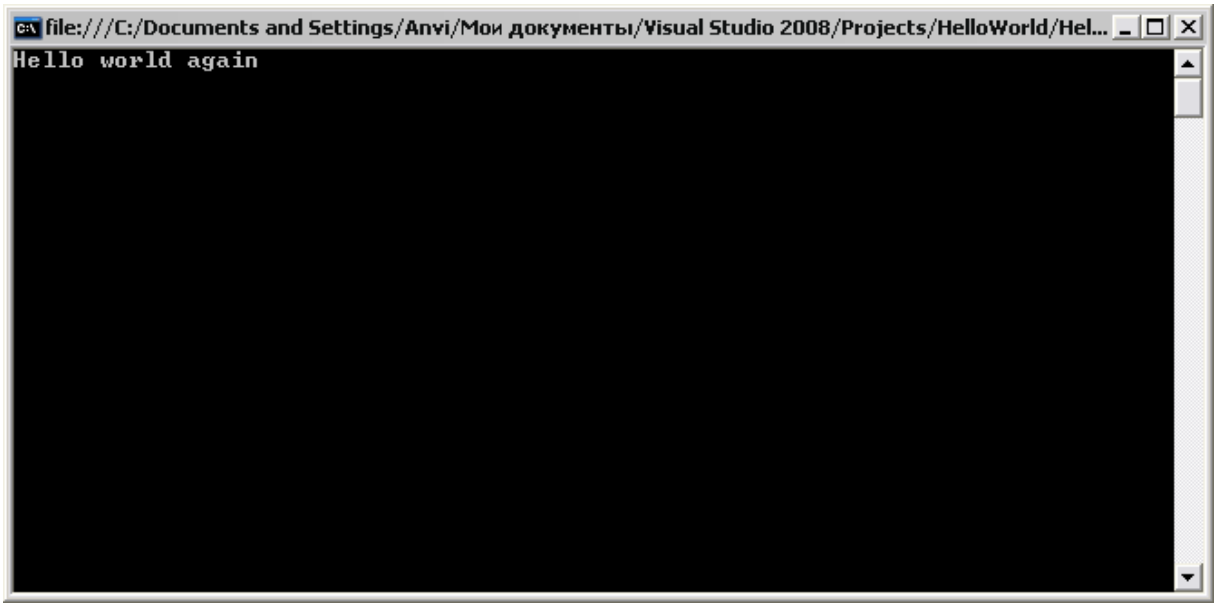


Рис. 1.3

Теперь остановимся подробнее на синтаксисе *C#*.

Объектно-ориентированным языкам свойственно иметь две большие категории типов: типы, которые присущи языку («базовые типы»), и классы, которые программист может создать самостоятельно.

В соответствии с логикой это кажется вполне нормально и удобно, хотя на практике приводит к ряду проблем несовместимости типов между собой. Проблема возникает, например, в том случае, когда вы хотите со-

здать метод, принимающий аргументы любого типа, которые может поддерживать данный язык программирования.

В *.net* и *C#* нет такой проблемы, так как любая сущность *CTS (Common Type System* – уникальная система типов, принятая в *.NET*) является объектом. Причем базовым классом служит *System.Object*.

Все типы в *C# .NET* делятся на РАЗМЕРНЫЕ и ССЫЛОЧНЫЕ.

Размерные типы содержат реальные данные и, следовательно, не могут быть равны *null*.

Именование типов очень напоминает *C++*:

Int, long, char,

за исключением того, что для беззнаковых чисел упрощено именование: теперь для этого используется первый символ "*u*" перед названием типа, вместо слова «*unsigned*», использовавшегося в *C++*.

На практике, написать *uint* намного удобнее, чем *unsigned int.*

Ссылочные типы крайне похожи на ссылки в *C++*. Правда, в отличие от указателя в *C++* ссылочный тип гарантирует, что ссылка указывает на объект заданного типа в памяти либо может быть равна *null*. Например:

string str = "This is a test";

т.е. в данном случае было выделено место в памяти и str содержит ссылку на него.

Массивы и классы в C# являются ссылочными типами.

Типы и псевдонимы (с помощью которых, как правило, происходит объявление) приведены в табл. 1.1.

Таблица 1.1

CTS Тип	Имя псевдонима в C#	Описание
System. Object	object	Класс, базовый для всех типов (CTS)
System. String	string	Строка
System. SByte	sbyt	8-разрядный байт (со знаком)
System. Byte	byte	8-разрядный байт (без знака)
System. S16	short	16-разрядное число (со знаком)
System. UM16	ushort	16-разрядное число (без знака)
System. Int32	int	32-разрядное число (со знаком)
System. UInt32	uint	32-разрядное число (без знака)
System. Int64	long	64-разрядное число (со знаком)
System. UInt64	ulong	64-разрядное число (без знака)
System.Char	char	16-разрядный символ (Unicode)

Окончание табл. 11

CTS Тип	Имя псевдонима в C#	Описание
System. Single	float	32-разрядное число с плавающей точкой (стандарт IEEE)
System. Double	double	64-разрядное число с плавающей точкой (стандарт IEEE)
System. Boolean	bool	Булевское значение (true/false)
System. Decimal	decimal	Данный 128-разрядный тип используется в основном, когда требуется крайне высокая точность (до 28-го знака)

Теперь рассмотрим, как работают операторы выбора и оператор ветвления, как реализуются циклы.

Оператор выбора (условный оператор *IF*)

Как известно, оператор выбора используется для последующего выполнения или невыполнения некоторого оператора или группы операторов в зависимости от условия. Если предлагаемое условие истинно, то вложенный оператор, или блок кода, выполняется. Альтернативная ветвь, которая может присутствовать (а может и нет), выполнится, если условие ложно.

```
if (resault == 777)
{
   System.Console.WriteLine("Congratulations, you win!!!!"); // выведет: «Поздравляем, вы победили»
}
else
{
   // выведет: «Пожалуйста, попробуйте снова»
   System.Console.WriteLine("please, try again \n");
   //выведет: «Мы уверены- вам повезет»
   System.Console.WriteLine("we are confident - you will be lucky");
}
```

Так же, как и в *C++*, можно использовать упрощенный синтаксис с оператором «*?*»:

(resault == 777) ? true : false ;

Оператор ветвления

Оператор ветвления может иметь большое количество ветвей, выбор которых осуществляется с помощью значения управляющего выражения. Это очень удобный способ реализации кода, когда существует некий параметр, в зависимости от которого должны выполняться те или иные ветви кода. В C# он реализуется следующим образом:

```
switch ( value )
{
    default:
    {
        System.Console.WriteLine("для этого варианта действие не определено”);
        break;
    }
    case 1 :
    {
        System.Console.WriteLine("Цифра 1”);
        break;
    }
    case 2 :
    {
        System.Console.WriteLine("Цифра 2”);
        break;
    }
    case 3 :
    {
        System.Console.WriteLine("Цифра 3”);
        break;
    }
}
```

В языке *C#* существуют 4 вида циклов

Циклы реализуются с помощью следующих зарезервированных слов: *while, do while , for , foreach* .

Рассмотрим каждый из них на примере.

Переменные, объявленные в цикле (в том числе в заголовке цикла *for* и *foreach*) не видны снаружи цикла (так же, как в стандартном *C++*).

1. Оператор цикла while

Пока истинно управляющее условие, выполняется оператор (или тело цикла).

Пример: значение переменной *a* инициируется равным 100; затем, пока *a* больше 5, выполняется тело цикла − вывод *a*, затем его значение уменьшается на единицу.

```
int a = 100;
while (a > 5)
{
  System.Console.WriteLine(a);
  a--;
}
```

2. Оператор цикла dowhile

В данном случае тело цикла выполняется до проверки условия:

```
int a = 100;
do
{
  System.Console.WriteLine(a);
 a--;
}while (a > 5)
```

3. Оператор цикла for

Цикл *for* используют, как правило, когда число повторений известно заранее, т.е. в задачах, связанных с перебором. Мы устанавливаем начало отсчета, условие остановки и тип изменения параметра.

Пример: при начальном значении *a* = 100 перебор значений *a* происходит до тех пор, пока *a* больше пяти. После каждого выполнения тела цикла *a* уменьшается на единицу.

```
for (int a = 100; a > 5; a --)
{
  System.Console.WriteLine(a);
}
```

4. Оператор цикла foreach

Этот цикл полезен, когда необходимо перебрать все элементы массива. Рассмотрим на примере: массив *a* состоит из 3 элементов. Цикл *foreach* перебирает все значения, имеющиеся в массиве, присваивая их переменной *x*. В теле цикла выполняется вывод этих значений.

```
int[] a = new int[]{1,2,3}; // наш массив
foreach (int x in a)
{
  System.Console.WriteLine(x);
}
```

Как видно из кода, работа с массивами в *C#* отличается от работы с массивами в *C++*.

Массивы

Массивы претерпели изменения по сравнению с синтаксисом языка *C++*.

Массивы в *.NET* имеют тип *System.Array*. Они не копируются при присваивании (т.е. это ссылки, а не значения). Также их можно использовать в циклах *foreach*. Свойство массива *Length* содержит общее число элементов массива, что очень удобно.

Одномерные массивы

Начнем с одномерных массивов. Данный тип массива включает в себя информацию о базовом типе элементов, которые может содержать массив, а также о количестве элементов в массиве (размерность массива). Нумерация элементов массивов так же, как и в языке *C/C++*, начинается с нуля.

Одномерные массивы определяются следующим образом:

uint[] arr_name = new uint[100];

Здесь *arr_name* − имя массива, 100 − количество элементов в массиве. Так как нумерация начинается с нуля, первый элемент доступен через обращение *arr_name*[0], последний − через *arr_name* [99].

Теперь рассмотрим три способа заполнения массива значениями.

```
int arr_name = new int[2];
arr_name[0] = 1;
arr_name[1] = 2;
int[] arr_name = new int[] {1,2};
int[] arr_name = {1,2};
```

Прямоугольные массивы

Прямоугольный массив − это двумерный массив, например, размерностью 4 × 2:

```
int[,] arr_name_1 = new int[4,2];
int[,] arr_name_2 = {{0, 1, 2, 3}, {0, 1, 2}};
for(i = 0; i < arr_name_1.GetLength(0); i++)
{
  for (j = 0; j < arr_name_1.GetLength(1); j++)
  {
      System.Console.WriteLine(arr_name_1[i,j]);
  }
}
```

В данном коде мы рассмотрели пример вывода значений из двумерного массива.

Классы

Следующим пунктом в освоении *C#* являются классы. Синтаксис крайне прост, очень похож на синтаксис *C++*:

Ключевое слово *class*, затем имя класса и перечисление его членов, заключенное в фигурные скобки, − на первый взгляд, мало что изменилось.

Рассмотрим пример

```
public class MyTestClass
{
public  string SomeStringInformation;
}
```

C# имеет 4 модификатора доступа к элементам и методам класса (табл. 1.2).

Таблица 1.2

Модификатор доступа	Описание
public	Член класса доступен вне определения класса и иерархии производных классов
protected	Член класса невидим за пределами класса, и к нему могут обращаться только производные классы
private	Член класса недоступен за пределами области видимости определяющего его класса, поэтому доступа к этим членам нет даже у производных классов
internal	Член класса видим только в пределах текущей единицы компиляции. Модификатор доступа internals в плане ограничения доступа является гибридом public и protected, который зависит от местоположения кода

Безусловно, классы − очень обширная тема. Мы рассмотрим ее на примере разработки класса и реализации консольной работы с ним.

1.3. Разработка класса и реализация консольного приложения

Приведем пример создания небольшого класса в *C#*. В нем мы рассмотрим создание новых методов и переменных − членов класса, а также работу с классом.

Создадим новый класс, назовем его *Man* (человек).

Наш класс будет схематично описывать персонажа компьютерной игры следующим образом (рис.1.4).

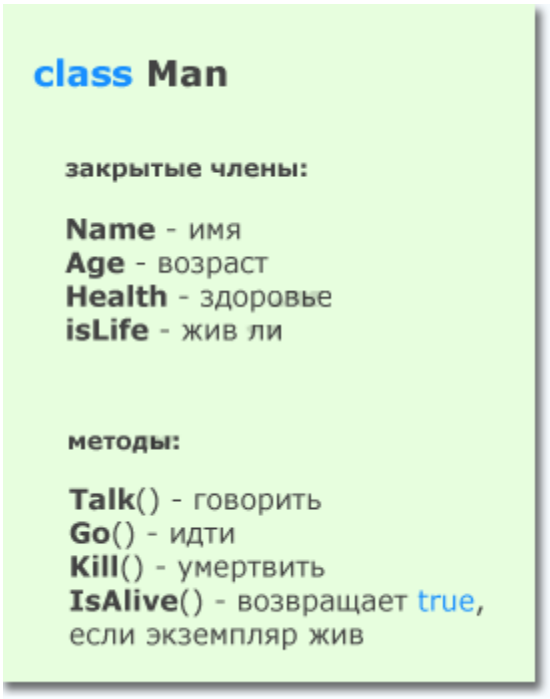


Рис. 1.4

Имя, возраст, здоровье и состояние жив/мертв будут описывать каждый конкретный экземпляр нашего класса − компьютерного человечка. На основе этих параметров будет основываться работа функций *Talk*, *Go*, *Kill* и *IsAlive*.

Имя будет передаваться в класс во время вызова конструктора класса. Возраст и состояние здоровья будут генерироваться в конструкторе автоматически, в дальнейшем они будут влиять на вывод текста в консоль при вызове доступных пользователю методов.

Создадим новое консольное приложение и назовем его *ClassTesting*.

После сгенерированного программой класса *Program* мы создадим свой класс, назвав его *Man* и снабдив необходимыми свойствами и методами:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace ClassTesting
{
    class Program
    {
```

```
        static void Main(string[] args)
        {
        }
    }
    public class Man
    {
        // конструктор класса (данная функция вызывается
        // при создании нового экземпляра класса)
        public Man(string _name)
        {
        Name = _name;
        isLife = true;
        }
        // закрытые члены, которые нельзя изменить
        // извне класса
        // строка, содержащая имя
        private string Name;
        // беззнаковое целое число, содержащее возраст
        private uint Age;
        // беззнаковое целое число, отражающее уровень здоровья
        private uint Health;
        // булево, означающее жив ли данный человечек
        private bool isLife;
        // заготовка функции "говорить"
        public void Talk()
        {
        }
        // заготовка функции "идти"
        public void Go()
        {
        }
        // функция, возвращающая показатель - жив ли данный человечек.
        public bool IsAlive()
        {
        // возвращаем значение, к которому мы не можем
        // обратиться напрямую извне класса,
        // так как оно имеет статус private
        return isLife;
        }
    }
}
```

Это заготовка для нашей программы. Мы добавили в код еще одну функцию, о которой забыли в схеме нашего класса, − *IsAlive*.

Данная функция будет возвращать значение переменной *isLife*, к которой мы не можем получить доступ извне класса, так как она является закрытой (*private*).

Теперь более тщательно займемся реализацией наших классов.

Во-первых, улучшим конструктор класса: в нем будут генерироваться состояние здоровья и возраст нашего человечка случайным образом. Для этого мы будем использовать класс генерации случайных чисел.

Генерация случайного числа в *C#*

Владение языками программирования не обходится без знания того, как выполняется генерация случайных чисел. Сейчас мы рассмотрим, как выполняется генерация случайного числа в *C#.*

Для выполнения данной задачи сначала создадим новый экземпляр класса *Random*

Random rnd = new Random();

Теперь для того чтобы получить случайное число, достаточно использовать созданный экземпляр класса, вызывая метод *Next();* , например

Age = rnd.Next();

При желании можно генерировать число в заданном диапазоне, указав начальное и конечное значения.

Модернизируем конструктор нашего класса так, чтобы он смог генерировать каждому новому экземпляру случайные значения здоровья и возраста.

Теперь конструктор класса выглядит следующим образом:

```
// конструктор класса (данная функция вызывается
// при создании нового экземпляра класса)
public Man(string _name)
{
    // получаем имя человечка из входного параметра
    // конструктора класса
    Name = _name;
    // экземпляр жив
    isLife = true;
    // генерируем возраст от 15 до 50
    Age = (uint)rnd.Next(15,50);
    // и здоровье, от 10 до 100%
    Health = (uint)rnd.Next(10, 100);
}
```

Параметр (*uint*) перед экземпляром класса *rnd* означает, что значение случайного числа, которое вернет функция *Next*, будет приведено к типу *uint* (*unsigned* – т.е. беззнаковое *int*). Просто функция *rnd* возвращает значение с типом *int*, и данное значение не может быть неявно преобразовано, поэтому необходимо сообщить компилятору о преобразовании.

После конструктора класса добавим объявление нашего экземпляра класса *Random*

```
// экземпляр класса Random
// для генерации случайных чисел
        private Random rnd = new Random();
```

Остается только модернизировать функции, предназначенные для перемещения и разговора.

Сначала рассмотрим функцию разговора.

Начало кода функции будет таким:

```
// заготовка функции "говорить"
public void Talk()
{
// генерируем случайное число от 1 до 3
int random_talk = rnd.Next(1, 3);
// объявляем переменную, в которой мы будем хранить строку
        string tmp_str = "";
```

Как видим, здесь объявляется переменная *random_talk*, и сразу при объявлении она получает значение – результат выполнения функции *rnd.Next* в диапазоне от 1 до 3.

В зависимости от этого значения при вызове функции *Talk* случайным образом будет происходить выбор: что же сейчас скажет наш объект?

Далее мы объявляем переменную, в которой будет храниться значение строки, которую мы сгенерируем.

Генерация строки будет выполняться с помощью оператора выбора *switch*:

```
// в зависимости от случайного значения random_talk
switch (random_talk)
{
case 1: // если 1 - называем свое имя
{
// генерируем текст сообщения
tmp_str = "Привет, меня зовут " + Name + ", рад познакомиться";
// завершаем оператор выбора
break;
```

```
}
case 2: // возраст
{
// генерируем текст сообщения
tmp_str = "Мне " + Age + ". А тебе?";
// завершаем оператор выбора
break;
}
case 3: // говорим о своем здоровье
{
// в зависимости от параметра здоровья
if (Health > 50)
tmp_str = "Да я здоров как бык!";
else
tmp_str = "Со здоровьем у меня неважно, дожить бы до " + (Age + 10).ToString();
// завершаем оператор выбора
break;
}
}
```

Как видно из кода, в зависимости от значения переменной *random_talk* выполняется ветвление кода и выбирается одна из веток для значения *random_talk = 1 (case 1)*, *random_talk = 2 (case 2)*, или *random_talk = 3 (case 3)*.

И завершение функции:

```
// выводим на консоль сгенерированное сообщение
System.Console.WriteLine(tmp_str);
}
```

Теперь рассмотрим код функции *Go*.

Здесь все проще: сначала мы определяем, жив ли объект. Если нет, то мы сгенерируем строку с сообщением о том, что данный человечек не жив и не может идти. В противном случае проверим уровень здоровья данного человечка и в зависимости от результата сгенерируем строку.

Код данной функции с подробными комментариями:

```
// заготовка функции "идти"
public void Go()
{
// если объект жив
if (isLife == true)
{
// если показатель более 40
// считаем объект здоровым
if (Health > 40)
```

```
{
// генерируем строку текста
string outString = Name + " мирно прогуливается по городу";
// выводим в консоль
System.Console.WriteLine(outString);
}
else
{
// генерируем строку текста
string outString = Name + " болен и не может гулять по городу";
// выводим в консоль
System.Console.WriteLine(outString);
}
}
else
{
// генерируем строку текста
string outString = Name + " не может идти, он умер";
System.Console.WriteLine(outString);
}
}
```

Напоследок добавим функцию, которая «убивает» нашего человечка. Её можно впоследствии усовершенствовать для исключения возможность «убивать» человечка несколько раз. Код функции будет выглядеть следующим образом:

```
public void Kill()
{
    // устанавливаем значение isLife (жив)
    // в false...
    isLife = false;
}
```

Вот и все. Если вы набрали весь код верно, то при компиляции (клавиша *F5*) не должно возникнуть никаких проблем.

Теперь вернемся к коду главного класса (*Program*).

Нам необходимо сделать так, чтобы консоль ожидала команды пользователя: в зависимости от этих команд будут выполняться различные действия – создание новых экземпляров класса *Man* (генерирование новых человечков) и выполнение функций, которые реализованы в классе *Man*.

Для этого мы создадим бесконечный цикл: в нем программа будет запрашивать пользователя ввести строку. Затем строка будет сверяться с заданным набором команд при помощи оператора выбора (*switch*).

В зависимости от той или иной команды будут выполняться действия над нашим человечком. Команда *exit* будет завершать программу. Сначала посмотрим, как будут происходить ввод строки и выбор команды.

Первым делом объявим переменную *user_command*, в которой мы будем хранить команду, введенную пользователем. Далее мы объявим переменную *Infinity*. До тех пор пока значение *Infinity* будет равно *true*, цикл будет выполняться. Изменение этой переменной произойдет, только если пользователь введет команду *exit*.

```
static void Main(string[] args)
{
    // переменная, которая будет хранить команду пользователя
    string user_command = "";
    // бесконечный цикл
    bool Infinity = true;
    while (Infinity) // пока Infinity равно true
    {
        // приглашение ввести команду
        System.Console.WriteLine("Пожалуйста, введите команду");
        // получение строки (команды) от пользователя
        user_command = System.Console.ReadLine();
    }
}
```

Далее из кода видно, что мы начинаем наш цикл *while*. При старте выполнения кода из тела цикла сначала в консоль выводится строка с приглашением ввести команду. Далее мы ожидаем ввода строки пользователем. Введенная строка после нажатия клавиши *enter* будет записана в переменную *user_command*.

Теперь с помощью операции ветвления мы создадим обработку стандартных команд *exit* и *help*.

Ветвление default будет выполняться в том случае, когда введенная строка не будет соответствовать ни одной существующей команде.

```
// обработка команды с помощью оператора ветвления
switch (user_command)
{
// если user_command содержит строку exit
case "exit":
{
Infinity = false;
        // теперь цикл завершиться, и программа завершит свое выполнение
        break;
    }
```

```
    // если user_command содержит строку help
    case "help":
    {
        System.Console.WriteLine("Список команд:");
        System.Console.WriteLine("---");
        System.Console.WriteLine("create_man : команда создает человечка, (экземпляр класса Man)");
        System.Console.WriteLine("kill_man : команда убивает человечка");
        System.Console.WriteLine("talk : команда заставляет человечка говорить (если создан экземпляр класса)");
        System.Console.WriteLine("go : команда заставляет человечка идти (если создан экземпляр класса)");
        System.Console.WriteLine("---");
        System.Console.WriteLine("---");
        break;
    }
    // если команду определить не удалось
    default:
    {
        System.Console.WriteLine("Ваша команда не определена, пожалуйста, повторите снова");
        System.Console.WriteLine("Для вывода списка команд введите команду help");
        System.Console.WriteLine("Для завершения программы введите команду exit");
        break;
    }
}
```

Не забудьте закрыть все фигурные скобки (}) после операции ветвления.

Если мы теперь откомпилируем проект (клавиша *F5*), то увидим окно консоли с приглашением ввести команду.

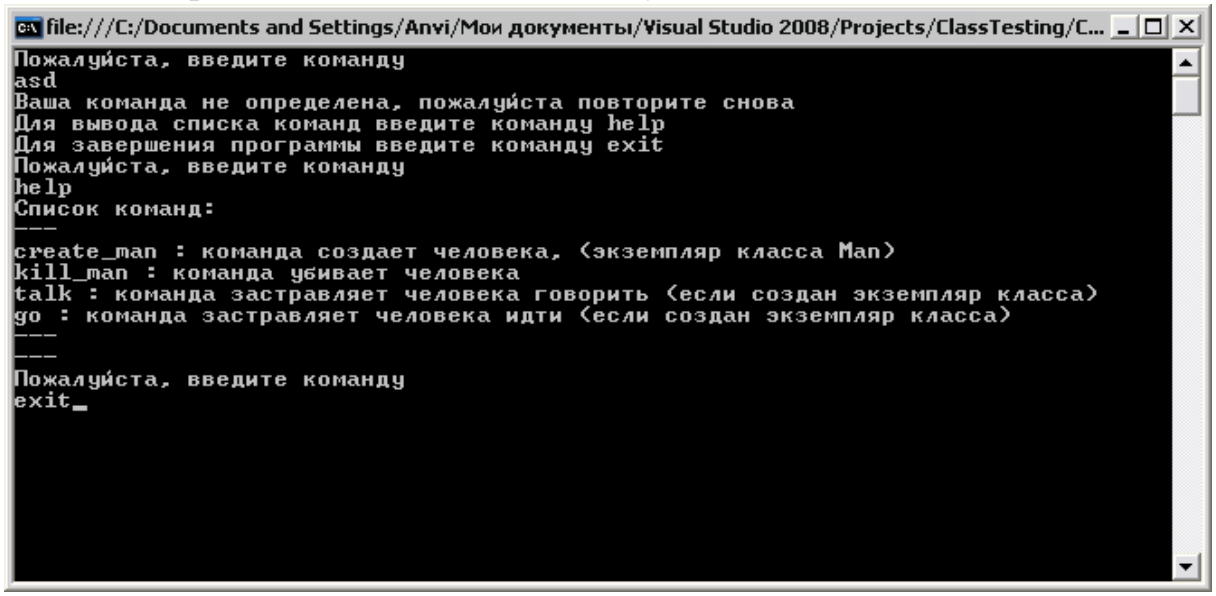


Рис. 1.5

Теперь нам осталось лишь реализовать обработку команд *create_man*, *kill_man*, *talk* и *go*.

После строки объявления переменной и перед началом цикла *while* добавьте объявления экземпляра класса *Man*.

```
// пустой (равный null) экземпляр класса Man
Man SomeMan = null;
```

Теперь рассмотрим код обработки команды *create_man* в теле оператора ветвления (*switch*).

```
case "create_man":
{
// проверяем, создан ли уже какой-либо человечек
if (SomeMan != null)
{
// человечек уже существует - "убиваем" его
SomeMan.Kill();
}
// просим ввести имя человечка
System.Console.WriteLine("Пожалуйста, введите имя создаваемого человечка \n");
// получаем строку, введенную пользователем
user_command = System.Console.ReadLine();
// создаем новый объект в памяти в качестве параметра
// передаем имя человечка
SomeMan = new Man(user_command);
// сообщаем о создании
System.Console.WriteLine("Человечек успешно создан \n");
break;
}
```

Как видно из кода, с помощью оператора new мы создаем новый объект в памяти.

Теперь он имеет имя, параметры, и мы можем выполнять другие команды, осуществляющие работу с классом.

```
case "kill_man":
{
    // проверяем, что объект существует в памяти
    if (SomeMan != null)
    {
        // вызываем функцию "смерти"
        SomeMan.Kill();
    }
    break;
}
```

Это код функции "смерти".

Обратите внимание на то, что перед тем, как воспользоваться вызовом какого-либо метода, реализованного в нашем классе, мы должны быть уверены, что объект, через который вызывается метод, объявлен в памяти.

Мы делаем это, проверяя, не равен ли наш объект (к примеру, *SomeMan*) *null*. Если не выполнить такую проверку, а потом в ходе работы программы окажется, что, например, *SomeMan = null,* и мы попытаемся вызвать любой из его методов (кроме конструктора класса – он как раз и создает объект в памяти), то в программе произойдет ошибка, а потом − обработка исключения.

Для того чтобы, например, выполнить функцию *kill( )*, компилятор сначала получает адрес памяти, где сейчас хранится экземпляр нашего класса. *SomeMan* как раз должен хранить этот адрес, а он будет равен *null*. Следовательно, компилятор получит команду “по адресу *null* определить адрес функции *Kill* и выполнить ее”. У него это, конечно, не получится.

Нам остались команды *talk* и *go*.

Они будут реализованы так же, как и команда *kill*: нам достаточно проверить, что объект существует в памяти. Если он существует – выполнить метод. Если нет – сообщить об этом.

```
case "talk":
{
// проверяем, что объект существует в памяти
if (SomeMan != null)
{
// вызываем функцию разговора
SomeMan.Talk();
}
else // иначе
{
System.Console.WriteLine("Человечек не создан. Команда не может быть выполнена");
}
break;
}
case "go":
{
// проверяем, что объект существует в памяти
if (SomeMan != null)
{
// вызываем функцию передвижения
SomeMan.Go();
```

```
}
else
{
System.Console.WriteLine("Человечек не создан. Команда не может быть выполнена");
}
break;
}
```

Пример работы откомпилированной программы:

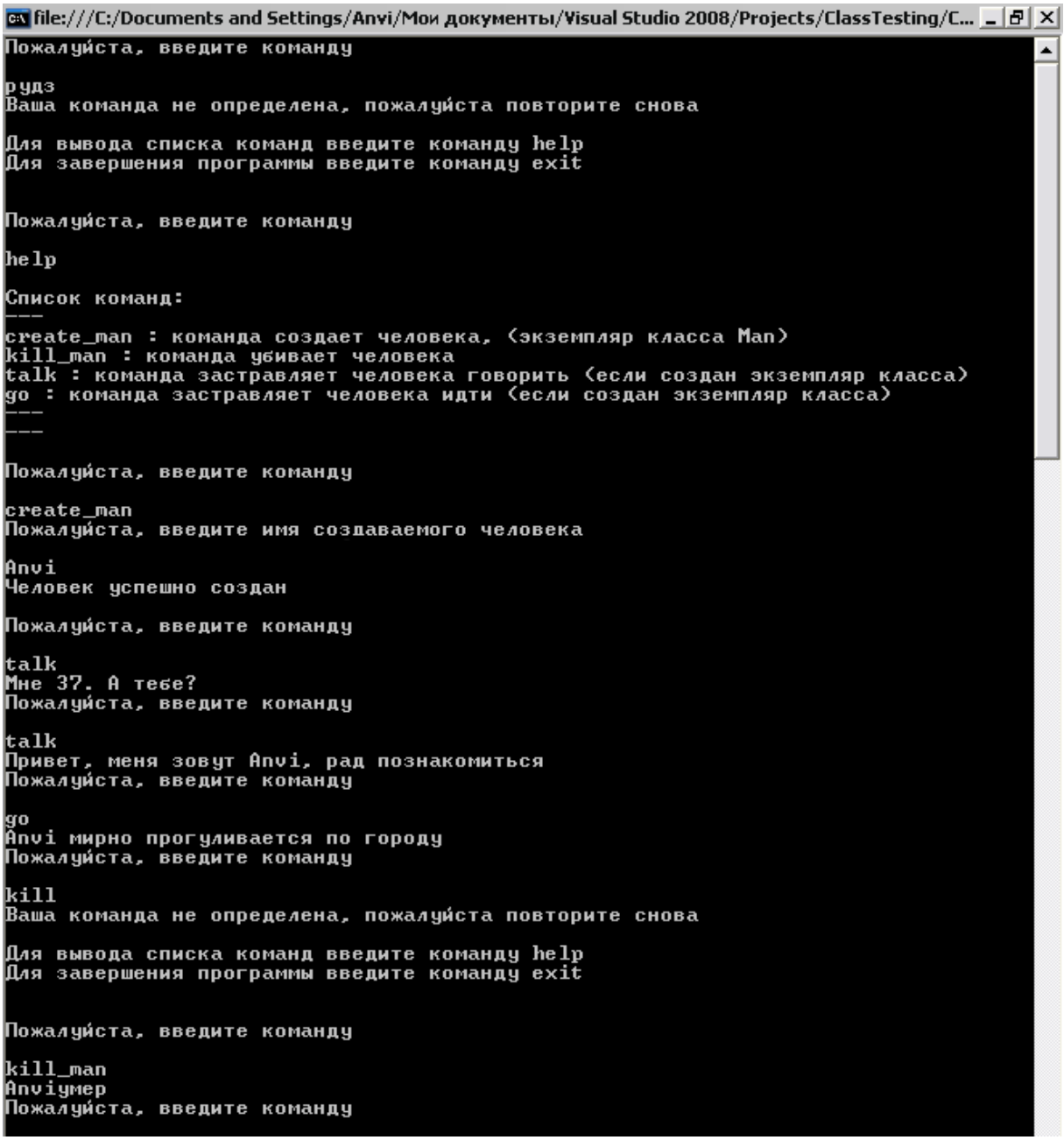


Рис. 1.6

Протестировав программу, обнаружим, что человечек способен разговаривать даже после «смерти». Сочтём это не за ошибку, а за интересную особенность программы.

Контрольные вопросы

1. Классы и объекты в *C#.*
2. Понятия объектно-ориентированного программирования: наследование, перегрузка и переопределение функций, конструктор и деструктор.
3. Структура и интерфейс пользователя рабочей среды *VisualC#.*
4. Основные объекты программы на *VisualC#.*

Тема 2. C#: РАЗРАБОТКА ОКОННОГО ПРИЛОЖЕНИЯ

Цель изучения темы. Освоение работы с оконными элементами в среде *Microsoft Visual C#* для операционной системы *Windows* с использованием *Microsoft .net Framework*. Получение практических навыков построения оконных приложений в *Visual C#.*

2.1. Основы *Windows.Forms*

Windows.Forms используется в *Microsoft .NET* для создания приложений, снабженных графическим интерфейсом. Основывается он на *.NET Framework class library* и имеет намного более совершенную и удобную в работе модель программирования, чем, например, программные интерфейсы *Win32 API* или *MFC*.

Если вам уже приходилось работать с *Win32 API* или *MFC*, то вам понравится то, насколько удобна и быстра разработка с использованием *Windows.Forms*. Написание оконных приложений с использованием *Windows. Forms* как намного удобнее, быстрее, так и на много качественнее, так как *Windows.Forms* устраняют многие ошибки *Windows API*. Код, который вы будете писать, намного проще и компактнее (нет *DDX* переменных и макросов, как в *MFC*).

Windows.Forms − это набор различных управляемых библиотек, с помощью которых вы можете выполнить все необходимые для оконного приложения действия, начиная от обмена сообщениями с операционной системой для отслеживания любых событий клиентского окна и заканчивая диалоговыми системами, связью с другими компьютерами по сети и многими другими возможностями.

В данном случае под формой понимается видимая поверхность окна, включающая информацию для конечного пользователя, а также содержащую в себе набор инструментов (элементов управления) для работы с представленными данными или взаимодействия с пользователем.

Так как *Windows.Forms* должна включать сотни организованных классов, чтобы обеспечивать все необходимые возможности разработчику, *.NET Framework* разбита на ряд иерархических разделов, имеющих свои имена. System является корневым разделом и предназначен для описания фундаментальных типов данных.

2.2. Создание оконного приложения в *.net*

Разрабатываемое приложение будет представлять собой небольшую форму, повторяющую идею известной программы: на форме будет содержаться вопрос «*Вы стремитесь сделать мир лучше?*». При попытке наведения указателя мыши на кнопку «*Да, конечно!*» окно будет убегать от нас. Нажатие на кнопку «*Нет*» не будет ничем ограничено.

Создавая данную программу, мы рассмотрим основные принципы разработки оконных приложений в *C#*, а также методы решения с их помощью каких-либо задач (в нашем случае использование убегающего окна).

Создайте новый проект, в качестве типа шаблона установите приложение *Windows Forms*, как показано на рис. 2.1.

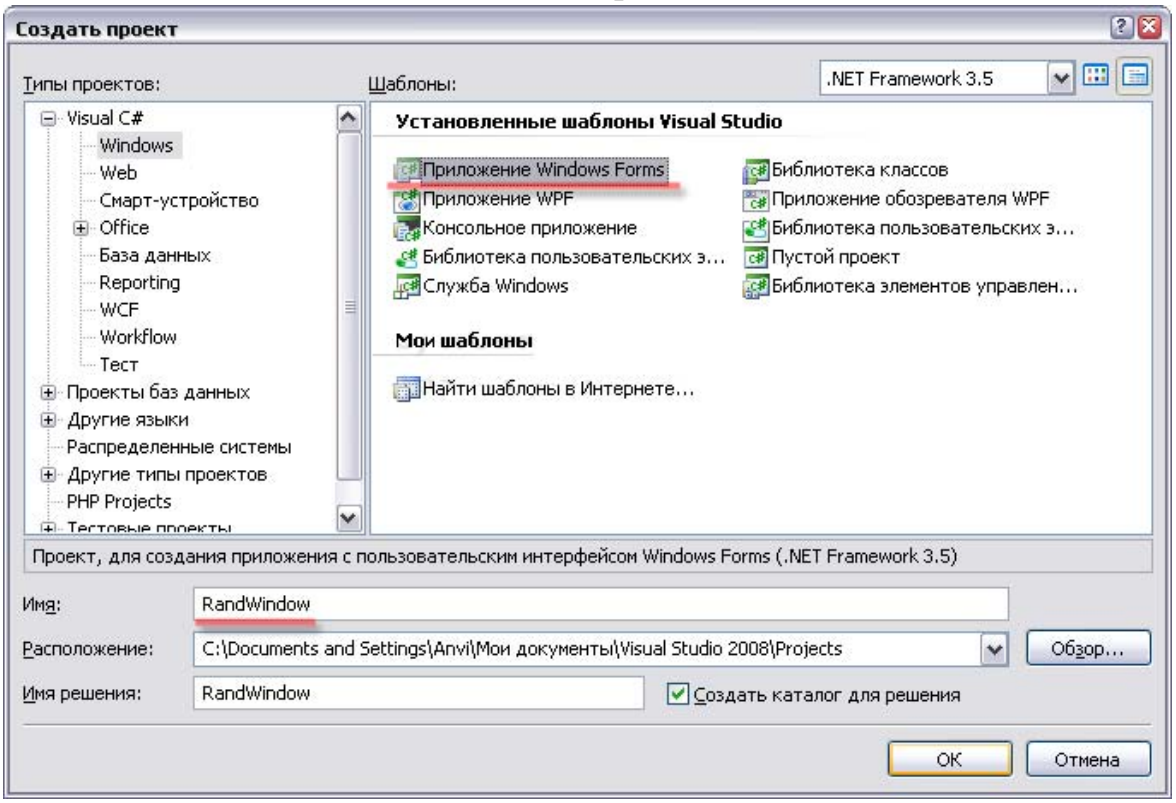


Рис. 2.1

Назовите проект *RandWindow* и нажмите кнопку *OK*.

Рабочее окно *MS Visual Studio* содержит следующие вспомогательные окна (рис. 2.2).

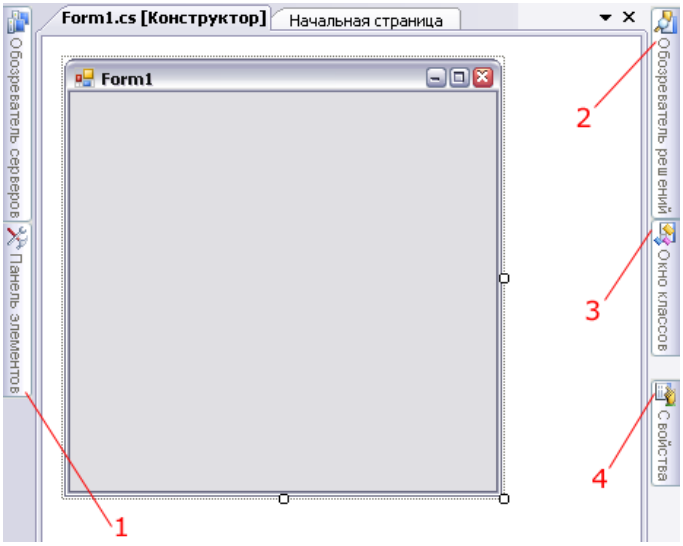


Рис. 2.2

На рисунке цифрами отмечены:

1. Окно *Toolbox* (Панель элементов управления, которые вы можете разместить на создаваемой форме).

2. Окно *Solution Explorer* (Обозреватель решений). Здесь вы сможете увидеть следующие узлы: *Properties* − настройки проекта, *Links* (Ссылки) − подключенные к проекту библиотеки, а также созданные и подключенные к проекту файлы исходных кодов (с расширением *.cs*) и подключенные к проекту формы (например, *Form1*).

3. Окно *Class View* (Окно классов, здесь представлены все созданные в программе классы).

4. Окно *Properties* (Свойства. Выбрав любой элемент управления или даже форму, вы сможете увидеть все параметры данного объекта, а также изменить значения, установленные в них по умолчанию).

Создание оконных приложений сводится к созданию всех необходимых диалоговых окон, а также к размещению на них необходимых элементов. В дальнейшем мы настраиваем обработку событий, создаваемых пользователем, и технические параметры программы. В нашем случае мы сначала разместим все необходимые элементы управления на главной форме, после чего добавим обработчик события перемещения мыши и обработку нажатия кнопок.

Добавление новых элементов управления на форму

После того как мы ввели имя проекта, установили необходимый шаблон и нажали кнопку *OK*, *MS Visual Studio* автоматически создаст каркас оконного приложения, после чего мы сможем добавить на него новые оконные элементы.

Для этого следует перетащить необходимый оконный элемент из окна инструментов (*ToolBox*).

Нашему окну потребуются два элемента поля для ввода, в которые мы будем выводить координаты указателя мыши, что облегчит нам понимание работы программы.

В центре окна будет находиться надпись, которую мы создадим с помощью элемента *Label*.

Снизу будут расположены две кнопки.

Немного растяните заготовку окна. Если вы нажмете на него правой кнопкой, то откроется контекстное меню. В нем нажмите на пункте «Свойства», после чего вы сможете изучить различные параметры окна, которые вы можете изменить. На рис.2. 3 изображены самые необходимые свойства:

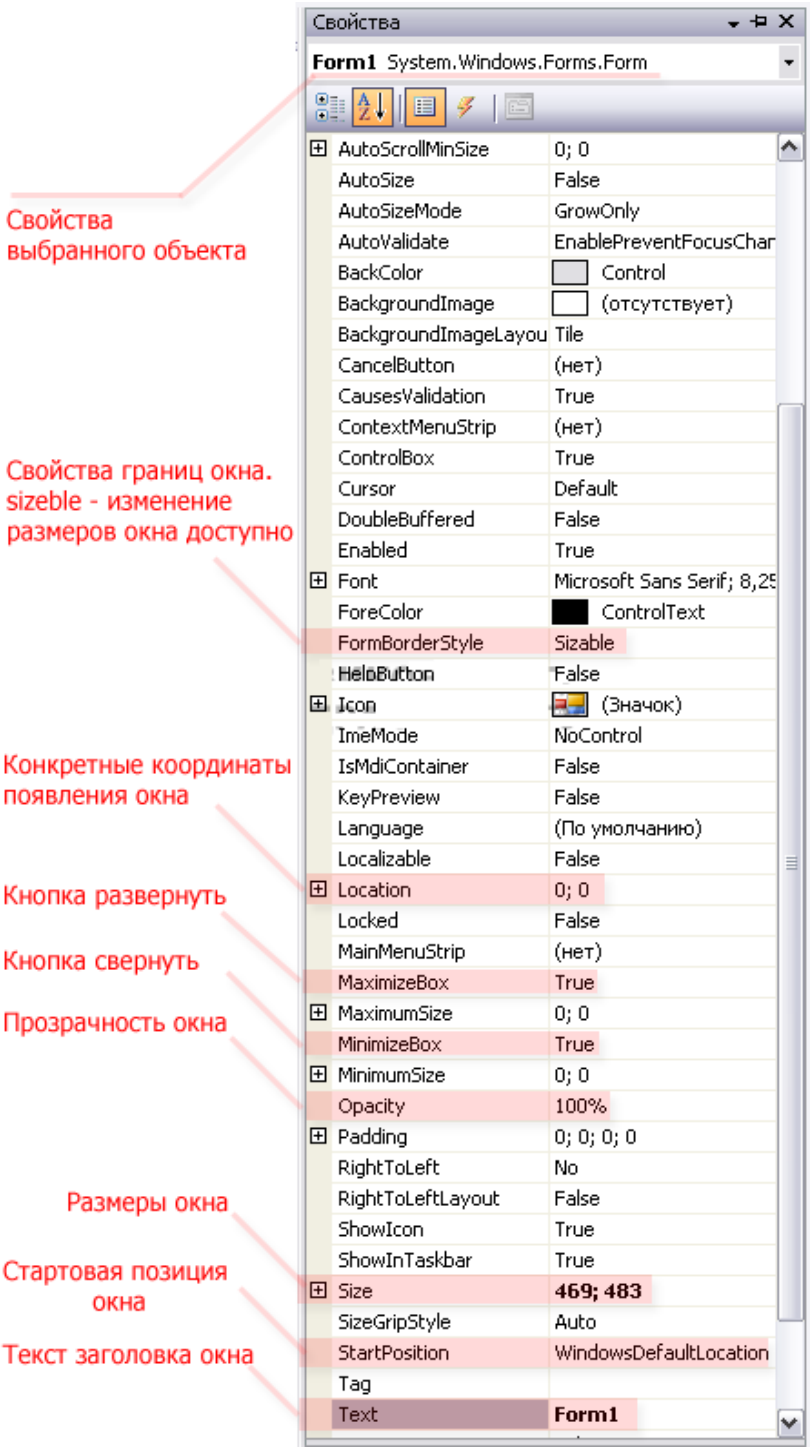


Рис. 2.3

Немного растяните заготовку окна и добавьте все необходимые элементы. На рис. 2.4 вы можете увидеть их в окне *ToolBox*.

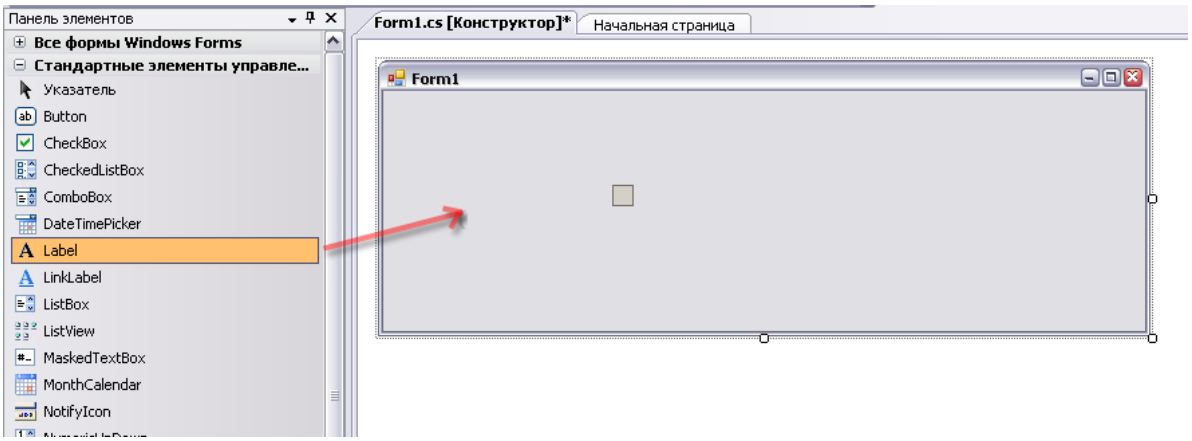


Рис. 2.4

Перейдите в свойства строки *Label1*, где измените текст на «*Вы стремитесь сделать мир лучше?*».

Также измените тип шрифта; для этого найдите свойство *Font* (рис. 2.5).

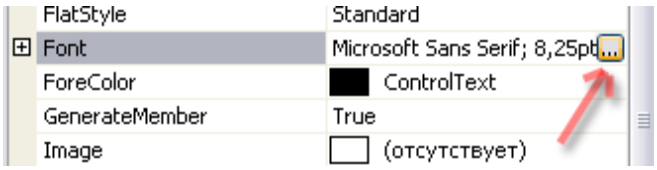


Рис. 2.5

Затем установите тип шрифта *Tahoma*, ширину шрифта *Bold* и размер равный 16 (рис. 2.6).

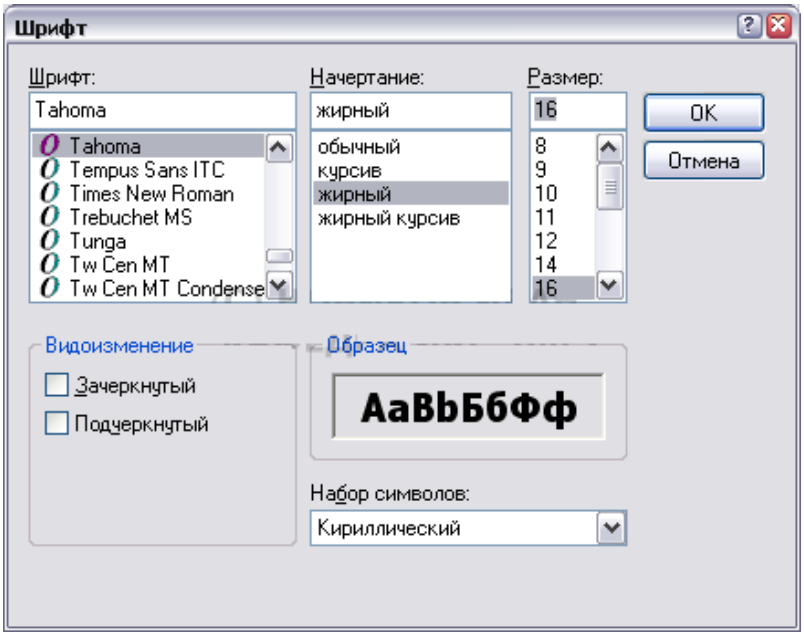


Рис. 2.6

Далее измените текст на кнопках, используя свойство *Text*.

Полученная заготовка окна программы будет выглядеть следующим образом (рис.2. 7).

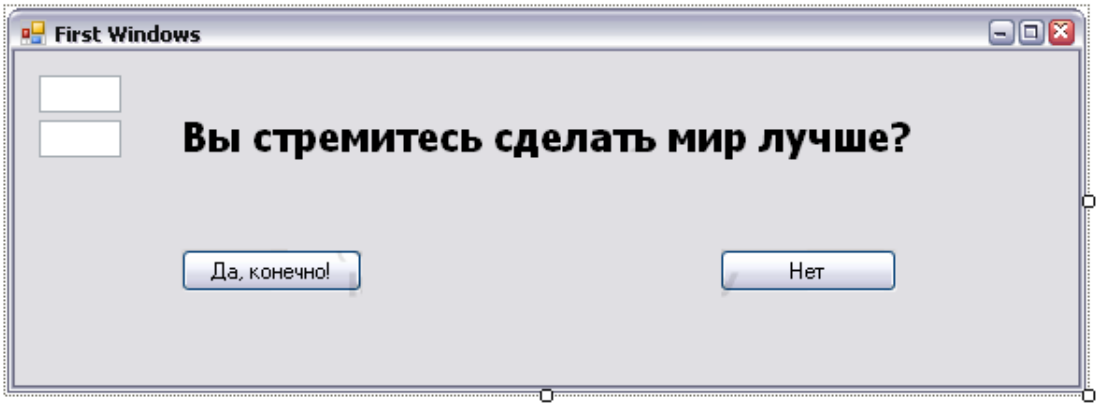


Рис. 2.7

Теперь рассмотрим техническую часть работы нашей программы.

1. Сначала мы добавим обработчик события перемещения мыши и реализуем вывод ее координат *x*, *y* в два созданных поля ввода.

2. Далее мы создадим функции обработчики щелчка по каждой клавише мыши (особенно усердные пользователи все же смогут попасть по кнопке «*Да, конечно!*»).

3. Далее мы добавим код, реализующий случайное перемещение окна в том случае, если курсор приблизится к кнопке «*Да, конечно!*».

Определение перемещения указателя мыши по форме

Щелкните непосредственно на части формы создаваемого приложения (НЕ на одном из элементов).

Теперь перейдите к свойствам формы с помощью щелчка правой кнопки мыши -> контекстное меню свойства.

Теперь необходимо перейти к списку возможных событий, которые может получать данное окно, для этого щелкните по кнопке «*Event*» (события), как показано на рис. 2.8.

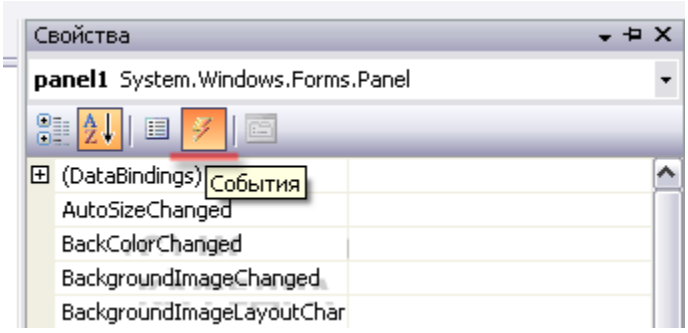


Рис. 2.8

Когда пользователь передвигает указатель мыши по нашему окну, операционная система посылает сообщение программе с текущими координатами указателя. Они-то нам и нужны.

Чтобы назначить обработчик данного события, найдите строку *MouseMove* (рис. 2.9), после чего сделайте двойной щелчок в поле справа от нее – автоматически добавятся обработчик события движения мыши и функция *Form1_MouseMove* в коде нашей программы.

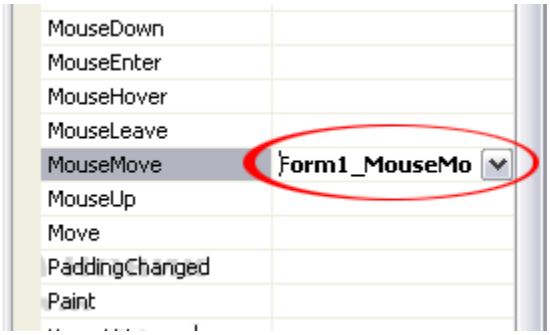


Рис. 2.9

Добавьте в эту функцию две строки, чтобы ее код стал выглядеть следующим образом:

```
private void Form1_MouseMove(object sender, MouseEventArgs e)
{
    // переводит координату X в строку и записывает в поля ввода
    textBox1.Text = e.X.ToString();
    // переводит координату Y в строку и записывает в поля ввода
    textBox2.Text = e.Y.ToString();
}
```

Данная функция, обрабатывающая событие перемещения указателя мыши над формой, получает два параметра – объект-отправитель и экземпляр класса *MouseEventsArgs*, содержащий информацию о координатах указателя мыши и других текущих свойствах.

Объекты *textBox1* и *textBox2* − это экземпляры класса *textbox*, реализующие управление нашими элементами поля для ввода.

Член данных экземпляров *Text* позволяет установить текст в этих полях.

Таким образом, если теперь откомпилировать программу (*F5*), при перемещении указателя мыши по форме окна мы будем видеть координаты указателя (внутри формы), которые будут непрерывно изменяться.

Теперь вернемся к заготовке нашей формы. Для это щелкните на соответствующей закладке (*Form1.cs [Конструктор]*), как показано на рис. 2.10.

Рис. 2.10

Сделайте двойной щелчок по первой кнопке; Visual Studio автоматически добавит код обработки данной кнопки при нажатии.

Добавьте следующие строки кода:

```
private void button1_Click(object sender, EventArgs e)
{
    // Вывести сообщение с текстом "Вы усердны"
    MessageBox.Show( "Вы усердны!!");
    // Завершить приложение
    Application.Exit();
}
```

Теперь снова вернитесь к конструктору и добавьте вторую кнопку также с помощью двойного щелчка по ней.

Она будет содержать следующий код:

```
private void button2_Click(object sender, EventArgs e)
{
    // Вывести сообщение, с текстом "Мы не сомневались в вашем безразличии"
    // второй параметр - заголовок окна сообщения "Внимание"
    // MessageBoxButtons.OK - тип размещаемой кнопки на форме сообщения
    // MessageBoxIcon.Information - тип сообщения - будет иметь иконку "информа-
ция" и соответствующий звуковой сигнал
    MessageBox.Show("Мы не сомневались в вашем безразличии","Внимание",
MessageBoxButtons.OK, MessageBoxIcon.Information);
}
```

Как видим, здесь мы немного усложнили код вызова окна сообщения, чтобы продемонстрировать то, как оно работает. Все параметры, передаваемые в функцию *Show*, закомментированы в исходном коде.

Теперь нам осталось только реализовать перемещение окна в тот момент, когда мышь приближается к кнопке «*Да, конечно*».

Для этого мы добавим код в функцию

```
private void Form1_MouseMove(object sender, MouseEventArgs e)
```

Принцип очень прост: получая координаты движения мыши, мы проверяем, не входят ли они в квадрат, очерчивающий нашу кнопку с неболь-

шим запасом. Если да, то мы генерируем два случайных числа, которые будут использованы для перемещения окна.

Мы могли бы просто отслеживать сообщение о наведении указателя мыши на кнопку, но оно приходит с заметной задержкой, в связи с чем пользователь без особого труда нажмет на кнопку "*Да*", поэтому мы будем просто вычислять попадание курсора в зону вокруг кнопки.

Также нам понадобится объявить несколько "рабочих" переменных, которые мы будем в дальнейшем использовать.

Добавление новых элементов управления на форму

Класс *Random* в *C#* представляет собой генератор псевдослучайных чисел, т.е. данный класс отвечает за выдачу последовательности чисел, соответствующих определенным статистическим критериям случайности.

Random rnd = new Random();

Здесь мы объявили экземпляр класса *Random (rnd)*, с помощью которого будем генерировать случайные числа. В дальнейшем мы будем использовать код вида *rnd.Next(диапазон_генерации)*; или *rnd.Next(от,до)*; для генерации случайного числа.

Объявим еще несколько переменных, часть которых сразу будет инициализирована.

```
Point tmp_location;
int _w = System.Windows.Forms.SystemInformation.PrimaryMonitorSize.Width;
      int _h = System.Windows.Forms.SystemInformation.PrimaryMonitorSize.Height;
```

Переменная *tmp_location* объявляется для того, чтобы в будущем временно хранить текущее положение окна.

Нам следует подумать и о том, что при случайном перемещении наше окно может выйти далеко за пределы экрана.

Определение разрешения экрана в *C#.NET*

Чтобы определить разрешение экрана в *C#.NET* мы будем использовать

```
System.Windows.Forms.SystemInformation.PrimaryMonitorSize
```

Переменные *_h* и *_w* будут хранить в себе размеры экрана пользователя, который определяется при их инициализации.

Теперь код этой функции будет выглядеть следующим образом.

```
private void Form1_MouseMove(object sender, MouseEventArgs e)
{
    // переводим координату X в строку и записываем в поля ввода
    textBox1.Text = e.X.ToString();
    // переводим координату Y в строку и записываем в поля ввода
    textBox2.Text = e.Y.ToString();
    // если координата по оси X и по оси Y лежит в очерчиваемом вокруг кнопки "да,
конечно" квадрате
    if (e.X > 80 && e.X < 195 && e.Y > 100 && e.Y < 135)
    {
        // запоминаем текущее положение окна
        tmp_location = this.Location;
        // генерируем перемещения по осям X и Y и прибавляем их к хранимому значе-
нию текущего положения окна
        // числа генерируются в диапазоне от -100 до 100.
        tmp_location.X += rnd.Next(-100, 100);
        tmp_location.Y += rnd.Next(-100, 100);
        // если окно вышло за пределы экрана по одной из осей
        if (tmp_location.X < 0 || tmp_location.X > (_w - this.Width / 2) || tmp_location.Y < 0
|| tmp_location.Y > (_h - this.Height / 2))
        {
            // новыми координатами станет центр окна
            tmp_location.X = _w / 2;
            tmp_location.Y = _h / 2;
        }
        // обновляем положение окна на новое сгенерированное
        this.Location = tmp_location;
    }
}
```

Вот и все. Откомпилировав приложение, можно попробовать нажать на кнопку "*Да, конечно*". Это будет непросто.

2.3. Создание второго оконного приложения

Целью данного пункта будет закрепление навыков работы с оконными приложениями. Мы рассмотрим на примере, как создаются элементы меню, панели инструментов, увидим, как создаются и вызываются дополнительные диалоговые окна, как работать с окнами выбора файлов. Также мы познакомимся еще с несколькими элементами управления и научимся динамически загружать изображения в форму.

Создайте новый проект, указав в качестве шаблона *Windows Form Application* и назовите *second_application*. Затем нажмите *Ок*.

Добавление меню в приложение *C#.net*

Здесь все просто: чтобы добавить меню в приложение, сначала откройте окно *Toolbox* (Панель элементов).

Здесь нам потребуется вкладка *Menus & Toolbars* (Меню и панели инструментов). В нем нас интересует элемент *MenuStrip* (рис. 2.11).

Зажмите его левой кнопкой мыши и перетащите на форму. На форме отразится элемент меню, как показано на рис. 2.12.

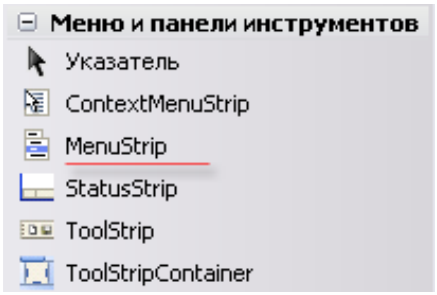


Рис. 2.11

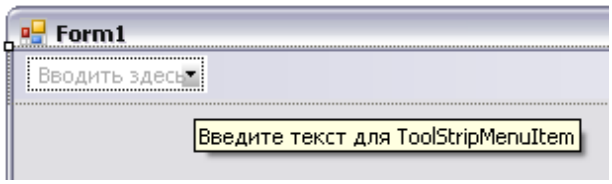


Рис. 2.12

Теперь щелкнув на строке «*введите здесь*», вы можете ввести название меню: назовем его «*File*». При этом добавятся два дополнительных элемента – один снизу, который будет доступен при открытии меню, второй − справа для создания новых разделов меню (рис. 2.13).

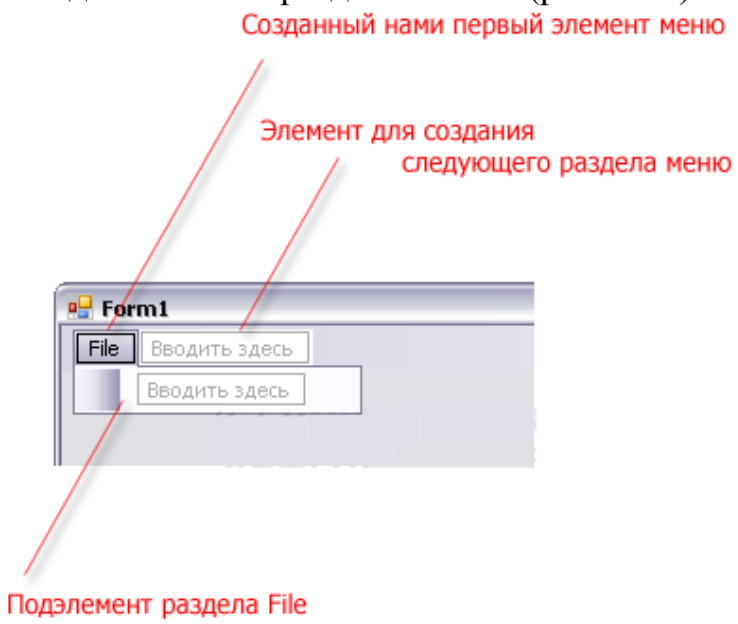


Рис. 2.13

Дайте название подэлементу меню *File* “*Выход*”, как показано на рис. 2.13. После этого назовите еще один раздел меню – «*загрузить*». В меню «*загрузить*» добавьте подэлемент “*загрузить изображение*” и ему назначьте в

раскрывающемся меню два подэлемента : «*В формате JPG*» и «*В формате PNG*». Выглядеть меню будет следующим образом (рис. 2.14).

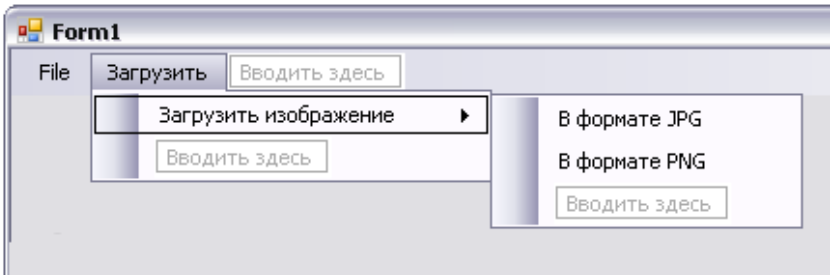


Рис. 2.14

Теперь рассмотрим, как добавить обработчик события меню. Для этого перейдите к меню "*Выход*". Сделайте двойной щелчок левой кнопкой мыши – *MS Visual Studio* автоматически создаст код функции обработчика и настроит событие обработки.

Перед нами откроется код функции обработчика:

```
private void выйтиToolStripMenuItem_Click(object sender, EventArgs e)
{
}
```

Так как этот элемент меню отвечает за выход из приложения, добавим в него код, который будет генерировать *MessageBox* с вопросом о подтверждении выхода из приложения. Если пользователь подтвердит выход, приложение будет завершено.

Новый код функции с комментариями:

```
private void выйтиToolStripMenuItem_Click(object sender, EventArgs e)
{
    // создаем переменную rsl, которая будет хранить результат вывода окна с вопросом
    // (пользователь нажал одну из клавиш на окне - это и есть результат)
    // MessageBox будет создержать вопрос, а также кнопки Yes No и иконку Question (Вопрос)
    DialogResult rsl = MessageBox.Show( "Вы действительно хотите выйти из приложения?", "Внимание!", MessageBoxButtons.YesNo, MessageBoxIcon.Question);
    // если пользователь нажал кнопку "да"
    if (rsl == DialogResult.Yes)
    {
        // выходим из приложения
        Application.Exit();
    }
}
```

Теперь можно откомпилировать приложение и проверить работоспособность кнопки.

Часто панель управления (*Toolbar*) дублирует элементы меню для быстрого к ним доступа.

Сейчас мы рассмотрим, как создается панель управления (*Toolbar*) в *C#.NET*.

Нам снова нужно перейти к окну *Toolbox* (Панель инструментов), вкладке *Menus & Toolbars*. В этот раз мы выберем элемент *ToolStrip* (рис. 2.15).

Рис. 2.15

Перетащите элемент управления на окно и вы увидите, как вдоль его верхней границы разместится панель *ToolBar* (рис. 2.16).

Рис. 2.16

Мы изменим положение привязки нашего *ToolBar*'а. Для этого щелкнем по нему правой кнопкой и в открывшемся контекстном меню выберем пункт "Свойства". Откроется окно свойств: здесь мы изменим привязку на левую часть окна, внеся изменения в параметр *Dock*, как показано на рис. 2.17.

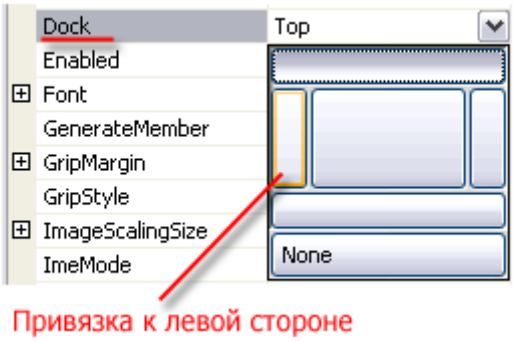


Рис. 2.17

Теперь увеличим размеры кнопок на *toolbar*'е. Для этого сначала необходимо в его свойствах установить параметр *AutoSize* равным *false*. Теперь мы можем изме-

нить размеры самих кнопок: установим параметры *Size - Width* равными 44. Поле станет шире (рис. 2.18).

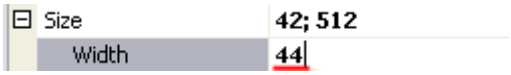


Рис. 2.18

Теперь добавим три кнопки на наш *ToolBar*. Для этого щелкните по нему и в раскрывающемся списке элементов, которые мы можем добавить, выберите элемент *button* (рис. 2.19).

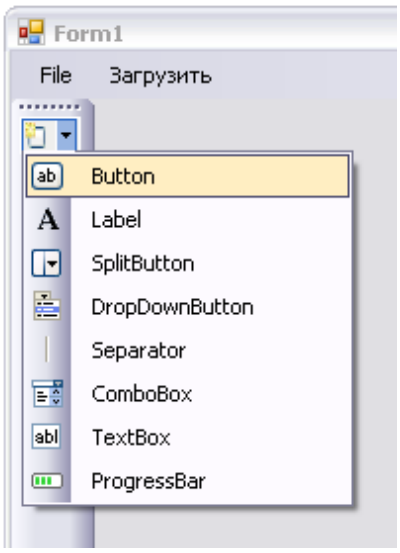


Рис. 2.19

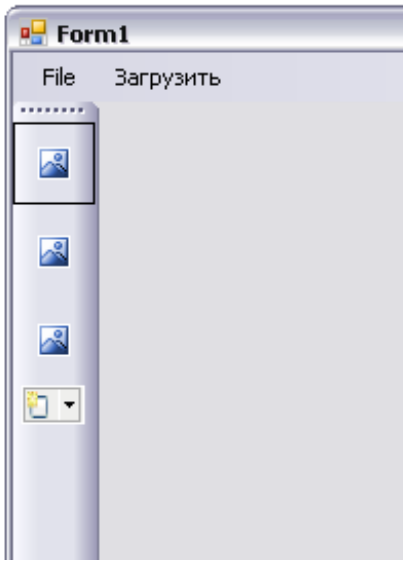


Рис. 2.20

Повторите операцию, чтобы кнопок на панели стало две. Теперь поочередно выберите каждую кнопку и в ее свойствах установите *AutoSize,* равный *false*. После это перейдите к полю *Size* и установите высоту, равную 42. Теперь кнопки примут вид квадрата.

Таким образом, на панели разместятся 3 кнопки, как показано на рис. 2.20.

Теперь назначим изображения для данных картинок. В качестве изображений можно использовать все современные форматы, в том числе и png24 с поддержкой прозрачности.

Мы будем использовать 3 следующих изображения.

Изображение для кнопки 1 (будет назначено кнопке, отвечающей за открытие дополнительного диалогового окна).

Изображение для кнопки 2 (будет назначено кнопке, отвечающей за открытие, загрузку файлов jpg).

Изображение для кнопки 3 (будет назначено кнопке, отвечающей за открытие, загрузку файлов png).

Обратите внимание, что у данных изображений прозрачный фон.

Две кнопки будут дублировать меню с функциями загрузки изображений, 1-я кнопка будет предназначена для вызова окна с отображением картинки, которую мы загрузили.

Теперь для установки изображений необходимо перейти в свойства картинки, после чего мы установим значение параметра *ImageScaling,* равным *none*, чтобы изображение не масштабировалось. Теперь в параметре *Image* мы можем выбрать изображение для загрузки, как показано на рис. 2.21.

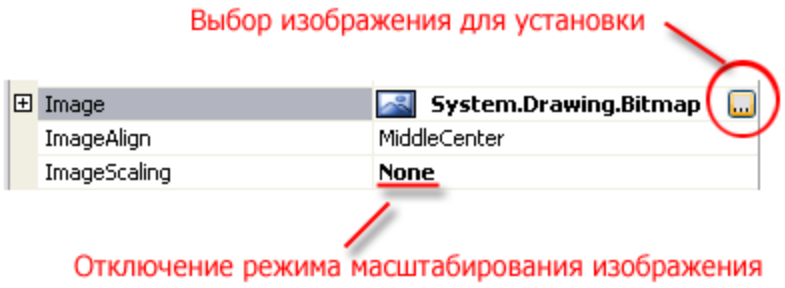


Рис. 2.21

В процессе выбора откроется окно, показанное на рис. 2.22.

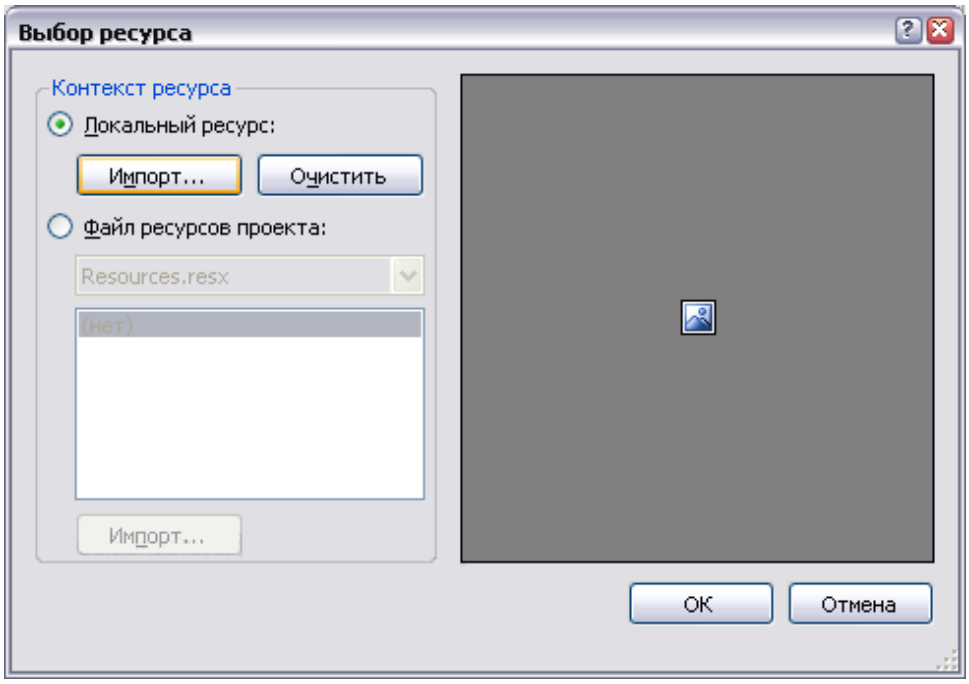


Рис. 2.22

Теперь щелкните на кнопке *Import* и выберите необходимый рисунок. Аналогично повторите с другими рисунками. В результате вы получите 3 красивые кнопки, как показано на рис. 2.23.

Рис. 2.23

Для того чтобы создать обработчики нажатий на кнопки этого *ToolBox*'а, достаточно совершить двойной щелчок мыши на каждом из них – *MS Visual Studio* автоматически сгенерирует код обработчика события и заготовки функций.

В будущем мы добавим вызов необходимых нам функций из этого обработчика.

Теперь мы разместим на форме элемент *PictureBox* и настроим размеры окна, чтобы оно выглядело следующим образом (рис. 2.24).

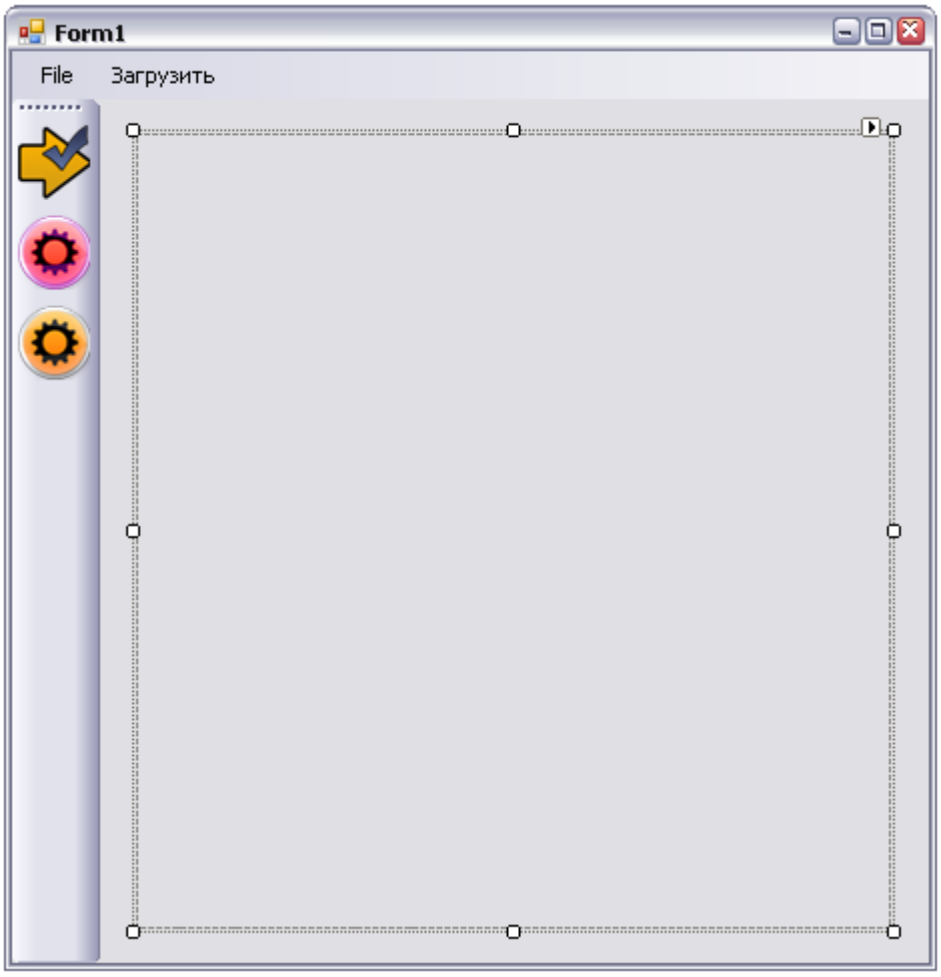


Рис. 2.24

В свойствах добавленного элемента *PictureBox* установите параметр *SizeMode*, равный *StretchImage*. Теперь, когда мы реализуем загрузку изоб-

ражения, оно будет масштабироваться под размеры нашего элемента *PictureBox*.

Создание окна выбора файла в *C#.net*

Чтобы пользователь мог выбирать файл для загрузки через стандартное в *windows* окно загрузки файлов, мы выполним следующие действия.

Перейдите к окну *ToolBox* (Панель элементов).

Теперь перетащите элемент управления *OpenFileDialog* (рис. 2.25) на форму.

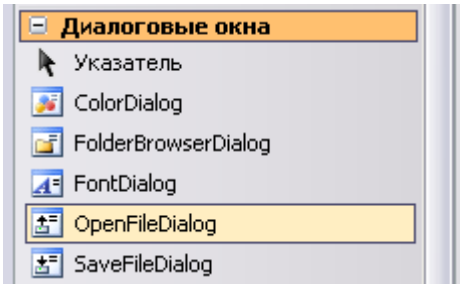


Рис. 2.25

Рис. 2.26

Местоположение, в которое вы перетащите элемент, не важно – он добавится в поле под окном к другим специфическим объектам (рис. 2.26).

Как видно из рис. 2.26, к дополнительным элементам, уже установленным на наше окно (меню и *ToolBox*), добавился еще и элемент *openFileDialog1*.

Теперь мы реализуем код открытия окна выбора файла с необходимыми нам параметрами.

Если вы еще не создали обработчики нажатия на кнопки загрузки и элементы меню, также предназначенные для загрузки, сделайте это с помощью двойных щелчков по ним.

Код сгенерированных функций выглядит следующим образом:

```
// обработка кнопки меню "загрузка - в формате jpg"
private void вФорматеJPGToolStripMenuItem_Click(object sender, EventArgs e)
{
}
// обработка кнопки меню "загрузка - в формате png"
private void вФорматеPNGToolStripMenuItem_Click(object sender, EventArgs e)
{
}
// обработка кнопки № 2 на панели
private void toolStripButton2_Click(object sender, EventArgs e)
{
```

```
}
// обработка кнопки № 3 на панели
private void toolStripButton3_Click(object sender, EventArgs e)
{
}
```

Открытие окна выбора файла и загрузка изображения в *C#.net*

Теперь напишем следующую функцию *LoadImage*.

```
Image MemForImage;
// функция загрузки изображения
private void LoadImage(bool jpg)
{
// директория, которая будет выбрана как начальная в окне для выбора файла
openFileDialog1.InitialDirectory = "c:\\";
// если будем выбирать jpg файлы
if (jpg)
{
// устанавливаем формат файлов для загрузки - jpg
openFileDialog1.Filter = "image (JPEG) files (*.jpg)|*.jpg|All files (*.*)|*.*";
}
else
{
// устанавливаем формат файлов для загрузки - png
openFileDialog1.Filter = "image (PNG) files (*.png)|*.png|All files (*.*)|*.*";
}
// если открытие окна выбора файла завершилось выбором файла и нажатием кнопки
ОК
if(openFileDialog1.ShowDialog() == DialogResult.OK)
{
try // безопасная попытка
{
// пытаемся загрузить файл с именем openFileDialog1.FileName - выбранный пользо-
вателем файл.
MemForImage = Image.FromFile(openFileDialog1.FileName);
// устанавливаем картинку в поле элемента PictureBox
pictureBox1.Image = MemForImage;
}
catch (Exception ex) // если попытка загрузки не удалась
{
// выводим сообщение с причиной ошибки
MessageBox.Show( "Не удалось загрузить файл: " + ex.Message);
}
}
}
```

Функция LoadImage в качестве входного параметра будет получать флаг о том, какой фильтр для выбора файлов необходимо выбрать. Далее вызывается окно выбора файла, и если оно при закрытии возвращает результат DialogResult.OK, то мы пытаемся загрузить и установить выбранную картинку в поле PictureBox. Конструкция try catch необходима нам здесь по следующей причине: если загрузка прошла неудачно, то мы получим сообщение об ошибке, но на выполнение программы это не повлияет, и мы сможем продолжить ее выполнение.

Далее функции обработчики нажатий пунктов меню и кнопок на панели управления реализуют вызов функции загрузки с необходимым флагом.

```
// обработка кнопки меню "загрузка - в формате jpg"
private void вФорматеJPGToolStripMenuItem_Click(object sender, EventArgs e)
{
// загружаем jpg файлы
LoadImage(true);
}
// обработка кнопки меню "загрузка - в формате png"
private void вФорматеPNGToolStripMenuItem_Click(object sender, EventArgs e)
{
// загружаем png файлы
LoadImage(false);
}
// обработка кнопки № 2 на панели
private void toolStripButton2_Click(object sender, EventArgs e)
{
// загружаем jpg файлы
LoadImage(true);
}
// обработка кнопки № 3 на панели
private void toolStripButton3_Click(object sender, EventArgs e)
{
// загружаем png файлы
LoadImage(false);
}
```

Если теперь откомпилировать приложение (*F5*), то можно попробовать загрузить изображение. Обратите внимание, что загрузка png24 с альфа-каналом тоже работает (рис. 2.27).

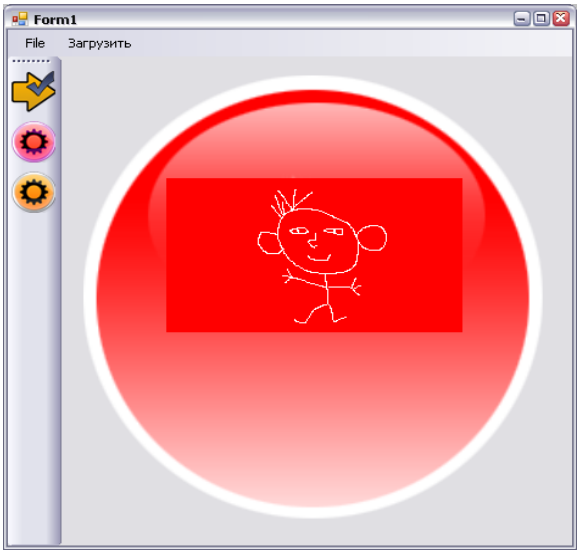


Рис. 2.27

Как видите, наше изображение масштабируется под размер элемента PictureBox. Поэтому сейчас мы добавим в проект еще одну форму, на которой мы будем отображать картинку в ее истинном размере.

Добавление и вызов дополнительного диалогового окна

Для этого перейдите к окну Solution Explorer (Проводник решений), после чего щелкните на названии проекта правой кнопкой мыши и в открывшемся контекстном меню выберите Add->Form, как показано на рис. 2.28.

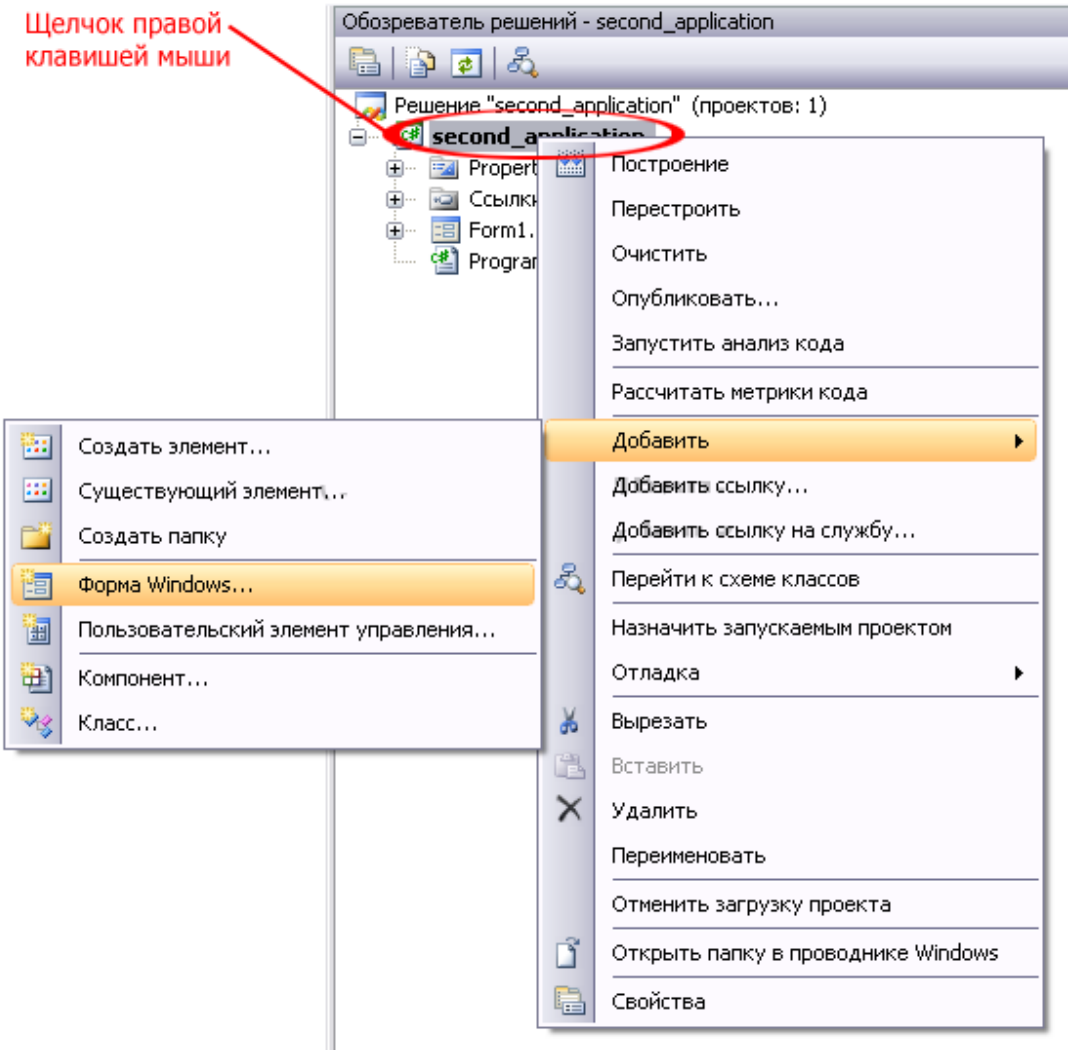


Рис. 2.28

В открывшемся окне введите название класса, который будет отвечать за генерацию нашего вспомогательного окна – Preview.cs (рис. 2.29).

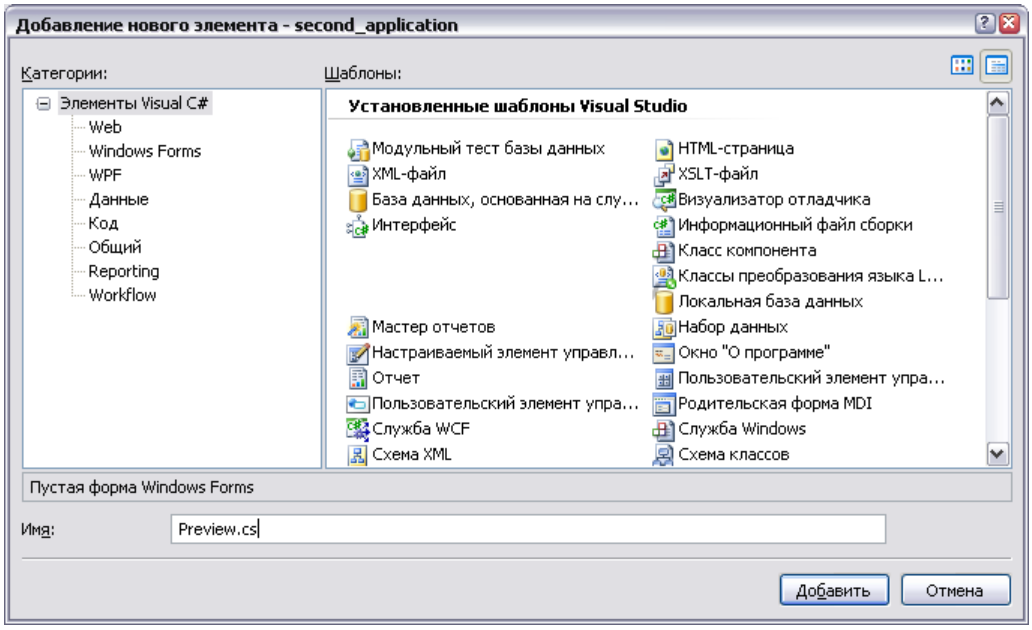


Рис. 2.29

Теперь измените размер окна так, как показано на рис. 2.30. Затем на форме разместите элемент panel. Внутри элемента panel разместите элемент pictureBox (см. рис. 2.30).

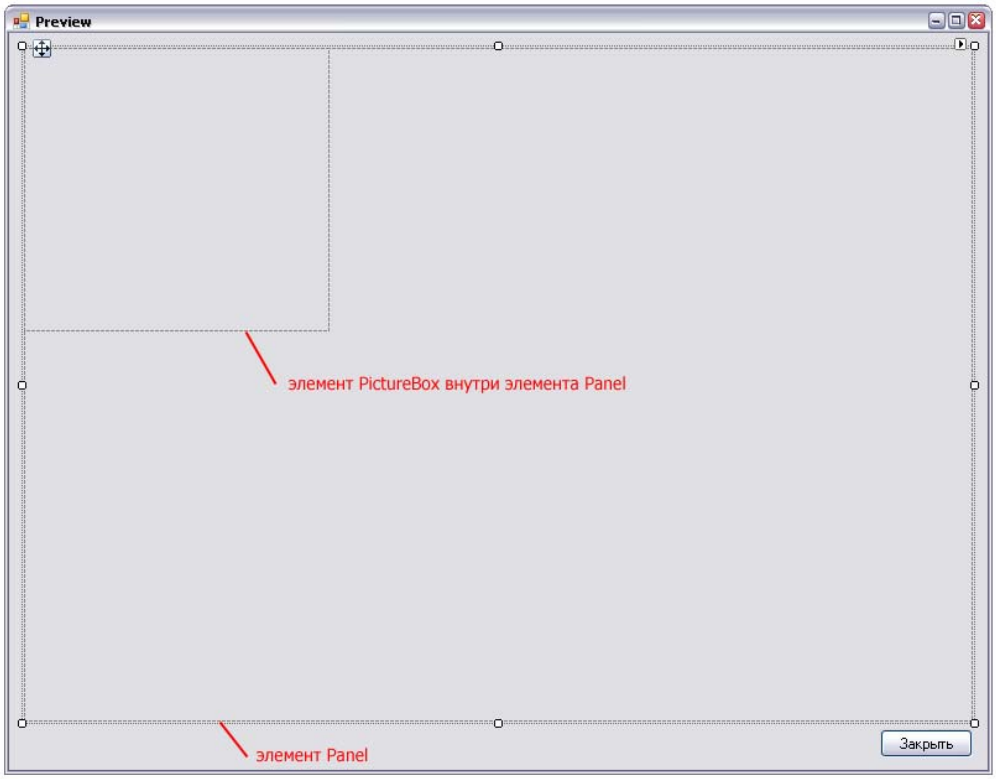


Рис. 2.30

Данное окно будет получать при загрузке в качестве параметра ссылку на наше загруженное изображение. Затем оно будет устанавливать границы элемента pictureBox, равными размерам полученного изображения, после чего устанавливать его в элемент pictureBox.

Перейдите к свойствам элемента panel и установите значение параметра AutoScroll, равное true. Теперь в том случае, если размер изображения будет превышать размер элемента panel1, будут появляться вертикальная и горизонтальная полосы прокрутки, с помощью которых можно будет просмотреть изображение.

Теперь рассмотрим исходный код, необходимый для реализации данных возможностей.

Первым делом назначим обработчик кнопки с изображением стрелки, направленной вправо на нашем главном окне. Щелкните по ней дважды, после чего вы перейдете к редактированию кода функции, которая будет вызвана при щелчке по данной кнопке.

Он будет выглядеть следующим образом:

```
// кнопка активации дополнительного диалогового окна
private void toolStripButton1_Click(object sender, EventArgs e)
{
// создаем новый экземпляр класса Preview,
// отвечающего за работу с нашей дополнительной формой
// в качестве параметра мы передаем наше загруженное изображение
Form PreView = new Preview(MemForImage);
// затем мы вызываем диалоговое окно
PreView.ShowDialog();
}
```

Как видно из исходного кода, класс Preview получает в качестве параметра наше загруженное изображение. Для правильной работы программы мы должны изменить конструктор класса Preview, а также реализовать остальную функциональность программы.

Перейдите к окну Preview, после чего сделайте двойной щелчок левой клавишей мыши на нем (НЕ на размещенных на нем объектах).

Откроется для редактирования функция

```
private void Preview_Load(object sender, EventArgs e)
```

Но мы сначала изменим код конструктора класса, теперь он будет выглядеть следующим образом:

```
public partial class Preview : Form
{
// объект Image для хранения изображения
Image ToView;
// модифицируем конструктор окна таким образом, чтобы он получал
// в качестве параметра изображение для отображения
public Preview(Image view)
{
// получаем изображение
ToView = view;
InitializeComponent();
}
```

Теперь отредактируем код функции Preview_Load. Он будет выглядеть следующим образом:

```
// эта функция выполнится при загрузке окна
private void Preview_Load(object sender, EventArgs e)
{
    // если объект, хранящий изображение неравен null
    if (ToView != null)
    {
        // устанавливаем новые размеры элемента pictureBox1,
        //равные ширине (ToView.Width) и высоте (ToView.Height) изображения
        pictureBox1.Size = new Size(ToView.Width, ToView.Height);
        // устанавливаем изображение для отображения в элементе pictureBox1
        pictureBox1.Image = ToView;
    }
}
```

В заключение самостоятельно добавьте обработчик нажатия кнопки "Закрыть". Функция закрытия будет выглядеть следующим образом:

```
// обработчик кнопки "Закрыть"
private void button1_Click(object sender, EventArgs e)
{
// закрываем диалоговое окно
Close();
}
```

Теперь если будет загружено большое изображение, его отображение в дополнительном окне будет снабжено полосами прокрутки (рис. 2.31).

Рис. 2.31

Откомпилируйте приложение (*F*5), чтобы проверить его.

Контрольные вопросы

1. Назначение *Windows.Forms*.
2. Порядок разработки оконного приложения в *VisualC#*.
3. Создание элементов оконного интерфейса в *VisualC#*.

Тема 3. МНОГОПОТОЧНЫЕ ВЫЧИСЛЕНИЯ

Цель изучения темы. Освоение средств организации многопоточных вычислений в *C#*.

3.1. Многопоточное программирование в *C#*

Понимание задач и методов организации многопоточного программирования важно для создания современных качественных и производительных программ, особенно использующих обработку графической информации.

Одной из очень важных особенностей языка *C#* является то, что он имеет встроенную поддержку многопоточного программирования.

Особенностью многопоточной программы является то, что она может состоять из нескольких блоков, каждый из которых выполняет свою часть по-

ставленной задачи. Таким образом, блоки выполняются параллельно. Такая часть программы называется потоком. Среда *.NET Framework,* в свою очередь, содержит ряд классов, предназначенный для гибкой реализации многопоточных приложений, причем встроенная поддержка многопоточности в *C#* позволяет свести к минимуму проблемы, встречавшиеся в процессе создания многопоточных приложений в некоторых других языках программирования.

Многопоточность может быть ориентирована на потоки и процессы. Основная разница состоит в том, что процесс фактически является отдельно выполняемой программой, т.е. здесь многопоточность основана на том, что выполняются две программы и более.

Поток (по-английски thread, буквально можно перевести как «нить») – это управляемая единица кода, выполняемая в адресном пространстве породившего его потока. Используя многопоточность, мы можем реализовать нашу программу таким образом, чтобы один поток просчитывал графику в сцене, визуализировал ее и обновлял окно, а другой − в это же самое время просчитывал физические законы, которые происходят в сцене.

Другой пример: программа должна заниматься просчетом математических алгоритмов. Если вычисление одного уравнения занимает 10 − 15 с, то окно приложения не будет отвечать на запросы операционной системы («зависнет» на 10 − 15 с), пока вычисление не будет завершено, так как программа выполняет строки кода последовательно.

Таким образом, ни одна современная сложная программа не может обойтись без многопоточности.

Потоки могут как выполняться, так и ожидать выполнения (быть временно приостановленными, заблокированными). Поток может затем возобновиться или просто завершиться. Все это – возможные состояния потока.

Многопоточность в среде *.NET Framework* реализована следующим образом: существуют два типа потоков − высокоприоритетный и низкоприоритетный.

Высокоприоритетный (*foreground*) поток в отличие от низкоприоритетного (или фонового − *background*) назначается как тип потока по умолчанию. Он не будет остановлен, если все высокоприоритетные потоки в его процессе будут остановлены.

Умение написания многопоточных программ сводится к тому, чтобы уметь эффективно разработать объектную модель программы, которая будет использовать в ходе решения задачи несколько отдельных потоков, а также координировать работу этих потоков между собой. Такая координация работы потоков называется синхронизацией потоков. Синхронизация – это специ-

альное средство, оснащенное собственной подсистемой методов и являющееся одной из главных составляющих многопоточного программирования.

Классы *C#*, отвечающие за поддержку многопоточного программирования, определены в пространстве имен *System.Threading*.

3.2. Базовые методы работы с потоками в *C# .net*

Рассмотрим процесс создания потоков на примере разработки консольной программы. При старте программы будут запускаться на выполнение 4 потока, каждый из которых будет выводить свой номер в окно консоли.

Важными здесь являются базовые методы работы с потоками: создание потоков, запуск, ожидание завершения.

При создании потоков мы установим для них различные приоритеты. Тогда некоторым потокам будет выделяться больше квантов времени процессора, и они будут доминировать при выводе своего номера в окно. Мы сможем это наблюдать в ходе работы программы.

Создадим новый проект и назовем его *Thread_Step_1*. В качестве шаблона выберем консольное приложение.

Сгенерированный код первоначально выглядит следующим образом:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace Thread_Step_1
{
    class Program
    {
        static void Main(string[] args)
        {
        }
    }
}
```

Для работы с потоками необходимо подключить пространство имен *System.Threading*.

Добавим строку *using System.Threading* после строки *using System.Text*.

Непосредственно перед функцией *Main* мы добавим функцию *WriteString*, которая будет отвечать за вывод символов, назначенных данному потоку (мы будем назначать номер потока) на экран. В качестве параметра функция будет получать объект *_Data*, который впоследствии будет преобразован в строку в коде функции.

Код функции будет выглядеть следующим образом:

```
static void WriteString(object _Data)
{
    //для получения строки используем преобразование типов:
    // приводим переменную _Data к типу string и записываем
    // в переменную str_for_out
    string str_for_out = (string) _Data;
    // теперь поток 1 тысячу раз выведет полученную строку (свой номер)
    for (int i = 0; i <= 1000; i++)
    Console.Write(str_for_out);
}
```

Теперь перейдем к коду функции *Main*. Здесь мы реализуем следующий код:

Сначала создадим 4 потока, каждому из которых укажем, что они будут выполнять функцию *WriteString*.

Далее мы назначим потокам приоритеты.

После этого мы запустим все четыре потока, передав в качестве параметра их номера.

Далее нам останется только дождаться завершения всех четырех потоков и ожидать ввода пользователем какого-либо символа, чтобы завершить выполнение программы.

Код функции *Main* будет выглядеть следующим образом:

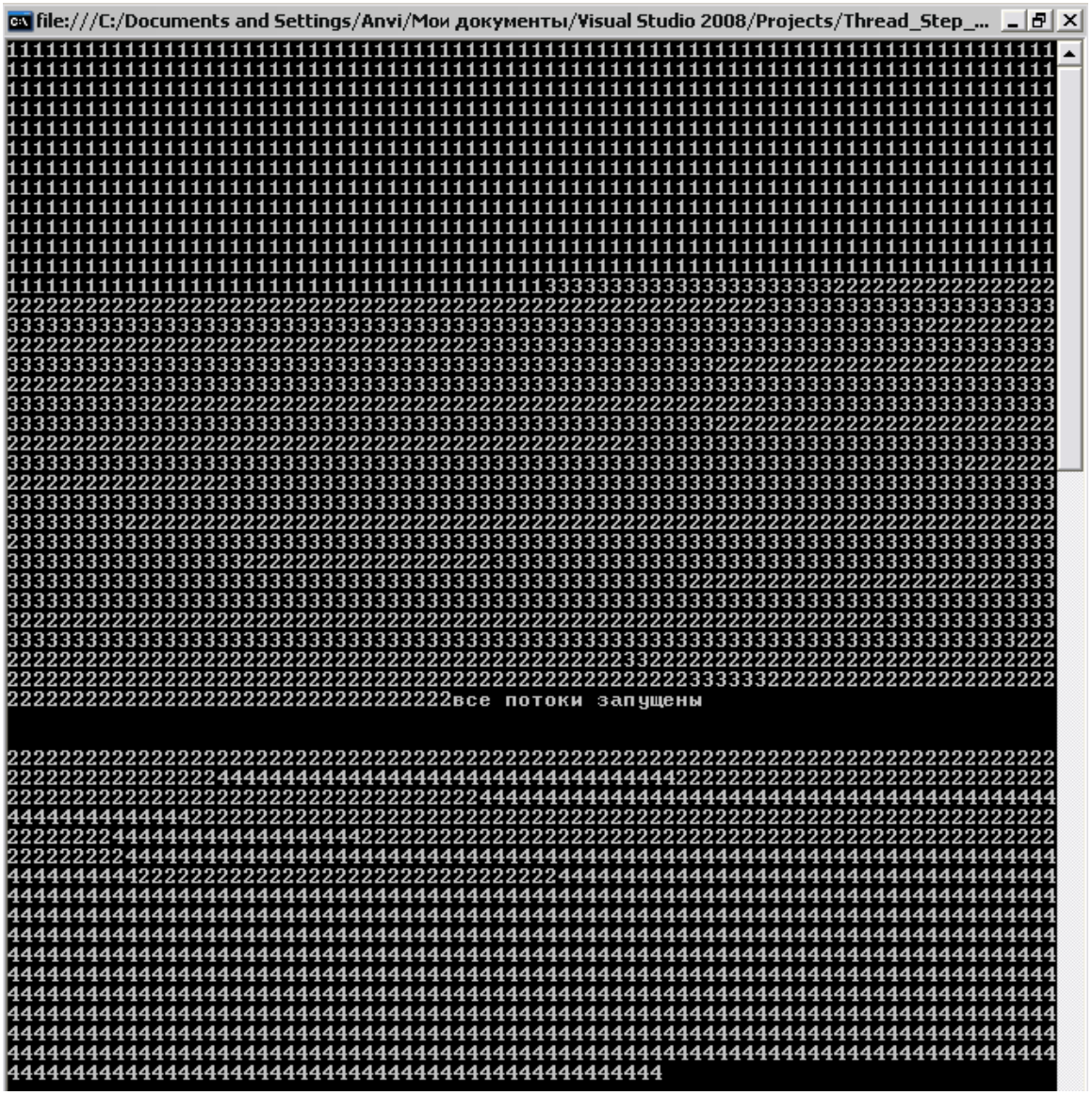
```
static void Main(string[] args) //точка входа в программу
{
//создаем 4 потока, в качестве параметров передаем имя Выполняемой функции
Thread th_1 = new Thread(WriteString);
Thread th_2 = new Thread(WriteString);
Thread th_3 = new Thread(WriteString);
Thread th_4 = new Thread(WriteString);
//расставляем приоритеты для потоков
th_1.Priority = ThreadPriority.Highest; // самый высокий
th_2.Priority = ThreadPriority.BelowNormal; // выше среднего
th_3.Priority = ThreadPriority.Normal; // средний
th_4.Priority = ThreadPriority.Lowest; // низкий
    // запускаем каждый поток, в качестве параметра передаем номер потока
    th_1.Start( "1");
    th_2.Start( "2");
    th_3.Start( "3");
    th_4.Start( "4");
    Console.WriteLine( "все потоки запущены\n\n");
    //Ждем завершения каждого потока
    th_1.Join();
```

```
    th_2.Join();
    th_3.Join();
    th_4.Join();
    // прочитать символ: пока пользователь не нажмет клавишу,
    // программа не завершится (чтобы можно было успеть посмотреть результат)
    Console.ReadKey();
}
```

Откомпилируйте и запустите программу (*F5*).

Пример результата работы программы можно увидеть на рисунке.

При каждом запуске программы выводимый на экран результат будет отличаться от предыдущего. Если запустить созданное приложение не из среды *Visual Studio* (без отладки), а из операционной системы, то до вывода сообщения «Все потоки запущены» по несколько раз может успеть выполниться назначенная функция каждого потока.



Как видно из рисунка, к моменту запуска последнего 4-го потока 1-й поток, имеющий *Highest* приоритет, уже успел полностью выполниться. Последний же 4-й поток с низшим приоритетом, к тому же запущенный последним, выполнится безусловно после всех других потоков.

Контрольные вопросы

1. Сущность многопоточности.
2. Многопоточность в *C# .NET*.
3. Базовые методы работы с потоками в *C# .NET*.

Тема 4. ВВЕДЕНИЕ В OPENGL

Цель изучения темы. Изучение принципов применения библиотеки *OpenGL* при разработке приложений в *C#*.

4.1. *OpenGL*

OpenGL означает *Open Graphics Library*, что переводится как «открытая графическая библиотека».

Другими словами, *OpenGL* − это некая спецификация, включающая в себя несколько сотен функций. Она определяет независимый от языка программирования кросс-платформенный программный интерфейс, с помощью которого программист может создавать приложения, использующие двухмерную и трехмерную компьютерную графику. Первая базовая версия *OpenGL* появилась в 1992 году, она была разработана компанией *Silicon Graphics Inc*, занимающейся разработками в области трехмерной компьютерной графики.

В библиотеку заложен механизм расширений, благодаря которому производители аппаратного обеспечения (например, производители видеокарт) могли выпускать расширения *OpenGL* для поддержки новых специфических возможностей, не включенных в текущую версию библиотеки. Благодаря этому программисты могли сразу использовать эти новые возможности в отличие от библиотеки *Microsoft Direct3D* (в этом случае им бы пришлось ждать выхода новой версии *DirectX*).

Библиотеки *OpenGL* и *DirectX* являются конкурентами на платформе *MS Windows*. *Microsoft* продвигает свою библиотеку *DirectX* и стремится замедлить развитие библиотеки *OpenGL*, что ослабило бы графическую систему конкурирующих ОС, где используется исключительно библиотека *OpenGL* для реализации вывода всей графики.

Мы будем учиться визуализации компьютерной графики именно с применением этой библиотеки. Однако прямой поддержки данной библиотеки в *.NET Framework* нет, поэтому мы будем использовать библиотеку *Tao Framework*.

4.2. *TAO Framework*

Tao Framework – это свободно распространяемая библиотека с открытым исходным кодом, предназначенная для быстрой и удобной разработки кросс-платформенного мультимедийного программного обеспечения в среде *.NET Framework* и *Mono* (см. приложение).

На сегодняшний день *Tao Framework* − оптимальный путь для использования библиотеки *OpenGL* при разработке приложений в среде *.NET* на языке *C#*.

В состав библиотеки на данный момент входят все современные средства, которые могут понадобиться в ходе разработки мультимедиа программного обеспечения: реализация библиотеки *OpenGL*, реализация библиотеки *FreeGlut*, содержащей все самые новые функции этой библиотеки, библиотека *DevIL* (легшая в основу стандарта *OpenIL* − *Open Image Library*) и многие другие.

Самые интересные библиотеки, включенные в *Tao Framework*:

- *OpenGL 2.1.0.12* − свободно распространяемый аппаратно-программный интерфейс для визуализации 2D- и 3D-графики.
- *FreeGLUT 2.4.0.2* – библиотека с открытым исходным кодом, являющаяся альтернативой библиотеке *GLUT* (*OpenGL Utility Toolkit*).
- *DevIL 1.6.8.3* (она же *OpenIL*) – кросс-платформенная библиотека, реализующая программный интерфейс для работы с изображениями. На данный момент библиотека поддерживает работу с изображениями 43 форматов для чтения и 17 форматов для записи.
- *Cg 2.0.0.0* – язык высокого уровня, созданный для программирования текстурных и вершинных шейдеров.
- *OpenAL 1.1.0.1* – свободно распространяемый аппаратно-программный интерфейс для обработки аудиоданных. (В том числе 3D-звука и EAX эффектов).
- *PhysFS 1.0.1.2* – библиотека для работы с вводом/выводом файловой системы, а также различного вида архивами на основе собственного *API*.

• *SDL 1.2.13.0* – кросс-платформенная мультимедийная библиотека, активно использующаяся для написания мультимедийных приложений в операционной системе *GNU/Linux*.

• *ODE 0.9.0.0* − свободно распространяемый физический программный интерфейс, главной особенностью которого является реализация системы динамики абсолютно твёрдого тела и системы обнаружения столкновений.

• *FreeType 2.3.5.0* – библиотека, реализующая растеризацию шрифтов. Данная библиотека используется в *X11* − оконной системе, которая обеспечивает все стандартные инструменты и протоколы для построения *GUI* (графического интерфейса пользователя) в *UNIX* подобных операционных системах.

• *FFmpeg 0.4.9.0* − набор свободно распространяемых библиотек с открытым исходным кодом. Данные мультимедийные библиотеки позволяют работать с аудио- и видеоданными в различных форматах.

Таким образом, библиотека *Tao Framework* является мощным и удобным свободно распространяемым инструментом для решения любых мультимедийных задач, преимущественно кросс-платформенного характера. Работая с данной библиотекой, разработчики могут использовать базу алгоритмов и реализованных за многие годы методов, что сокращает время разработки программных продуктов.

4.3. Инициализация *OpenGL* в *C#*

Проверим работоспособность библиотеки *Tao* с использованием тестового приложения: сначала подключим ее к проекту, затем проведем инициализацию библиотеки и инициализацию *OpenGL* и в качестве проверки визуализируем сферу с помощью библиотеки *FreeGlut*, разместив специальный элемент управления, предназначенный для визуализации сцены в окне программы.

Создание проекта и подключение библиотеки *Tao OpenGL* в *C#*

Сначала создайте новый проект, в качестве шаблона установив приложение *Windows Forms*. Назовите его *Tao-OpenGL-Initialization-Test*.

Дождитесь, пока *MS Visual Studio* закончит генерацию кода шаблона. Теперь перейдите к окну *Solution Explorer* (Проводник решений). Здесь нас интересует узел *Links*, который отображает связи с библиотеками, необходимыми для работы нашего приложения (рис. 4.1).

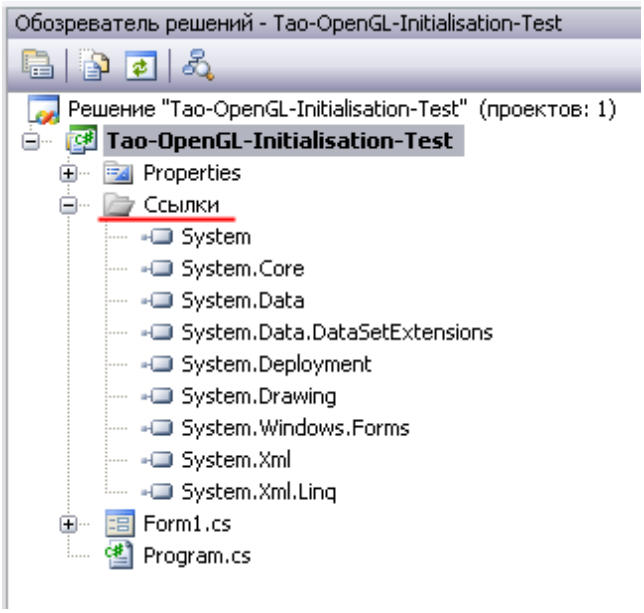


Рис. 4.1

Назовите главное окно «Тестирование инициализации *OpenGL* в *C#.NET*». (Свойства окна, параметр *Text*).

Щелкните по этому узлу правой клавишей мыши, после чего в открывшемся контекстном меню выберите «Добавить ссылку» (“*Add Link*”), как показано на рис. 4.2.

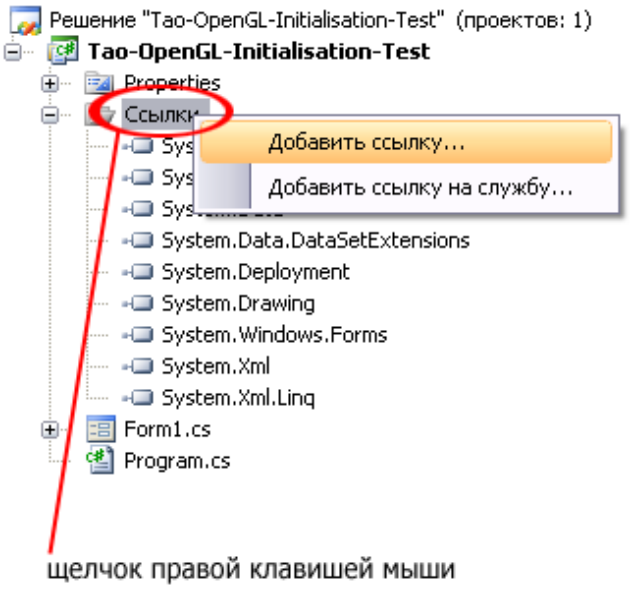


Рис. 4.2

В открывшемся окне «Добавить ссылку» перейдите к закладке "Обзор". После этого перейдите к директории, в которую была установлена библиотека *Tao Framework*. (По умолчанию “*C:\\Program Files\Tao Framework*”).

Нам потребуется папка *bin*, в ней хранятся необходимые нам библиотеки. Перейдите в папку *bin* и выберите три библиотеки, как показано на рис. 4.3.

Tao.OpenGL.dll отвечает за реализацию библиотеки *OpenGL*.

Tao.FreeGlut.dll отвечает за реализацию функций библиотеки *Glut*. Мы будем ее использовать для инициализации рендера, а также для других целей.

Tao.Platform.Windows.dll отвечает за поддержку элементов для визуализации на платформе *Windows*.

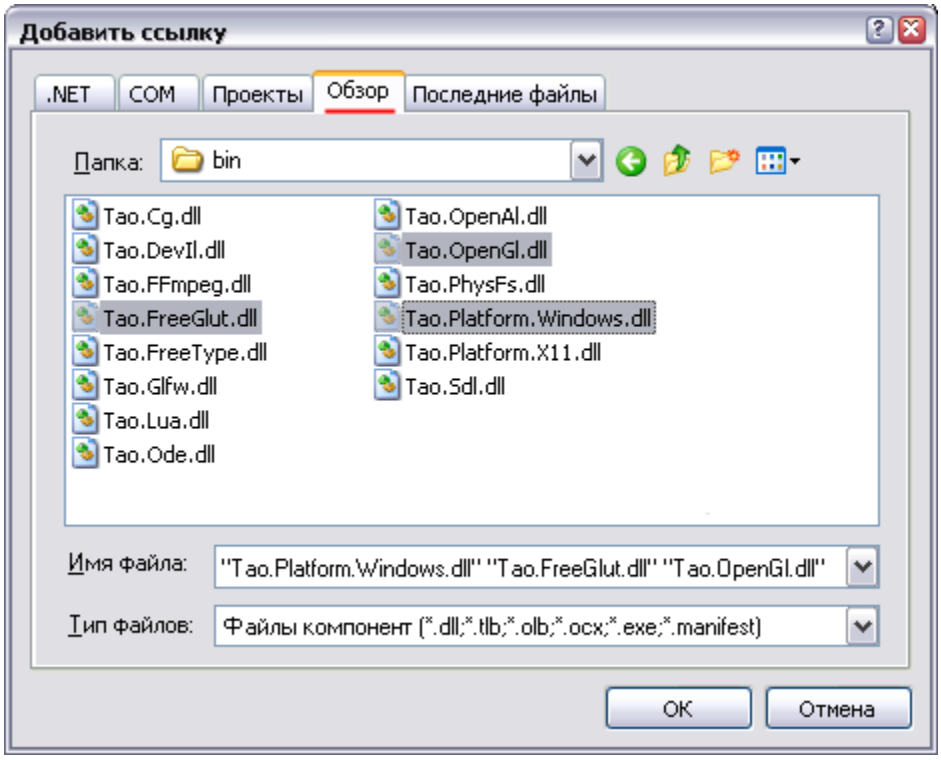


Рис. 4.3

На рис. 4.4 мы видим все добавившиеся библиотеки в узле «Ссылки» (*Links*).

Теперь перейдите к исходному коду окна. Для работы с нашими библиотеками необходимо подключить соответствующие пространства имен:

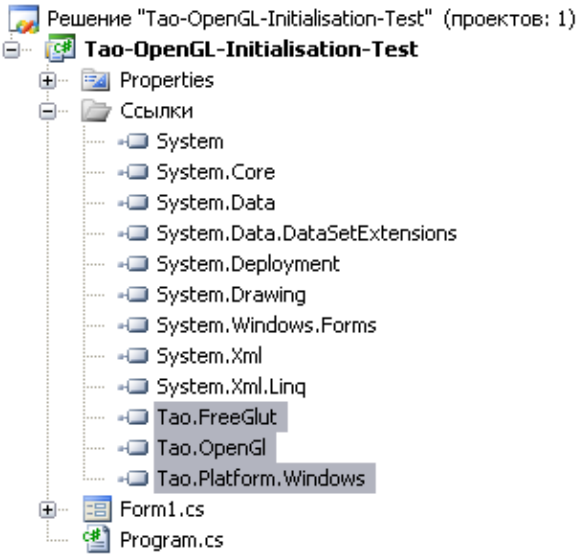


Рис. 4.4

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;
// для работы с библиотекой OpenGL
using Tao.OpenGl;
// для работы с библиотекой FreeGLUT
using Tao.FreeGlut;
// для работы с элементом управления SimpleOpenGLControl
using Tao.Platform.Windows;
```

Теперь вернитесь к конструктору диалогового окна и перейдите к окну *Toolbox* (панель элементов). Щелкните правой кнопкой на вкладке «Общие» и в раскрывшемся контекстном меню выберите пункт «Выбрать элементы» (*Choose Items*), как показано на рис. 4.5.

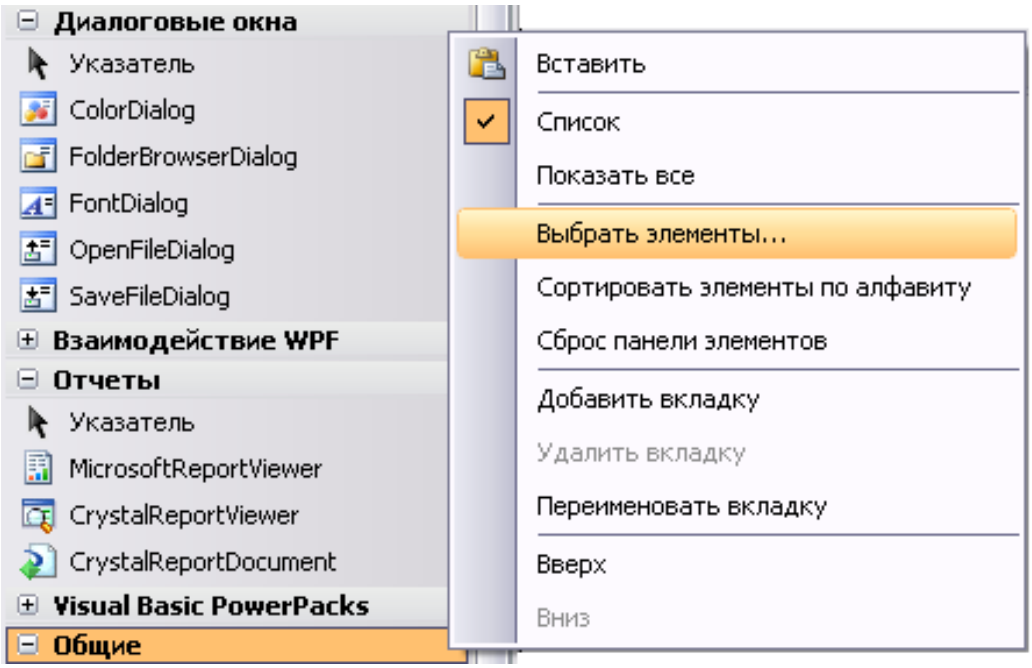


Рис. 4.5

В открывшемся окне найдите элемент S*impleOpenGLControl* и установите возле него галочку, как показано на рис. 4.6. Затем нажмите *OK*.

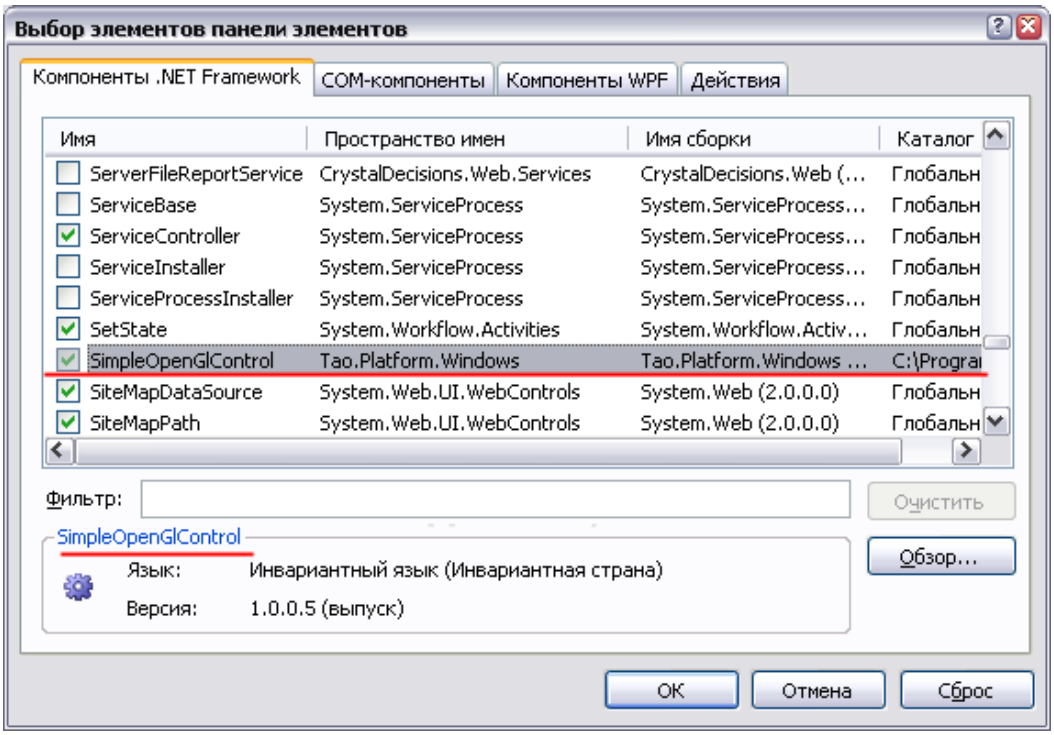


Рис. 4.6

Теперь данный элемент станет доступным для размещения на форме приложения. Перетащите элемент на форму и разместите так, как показано на рис. 4.7. Справа от размещенного элемента установите 2 кнопки – «Визуализировать» и «Выйти».

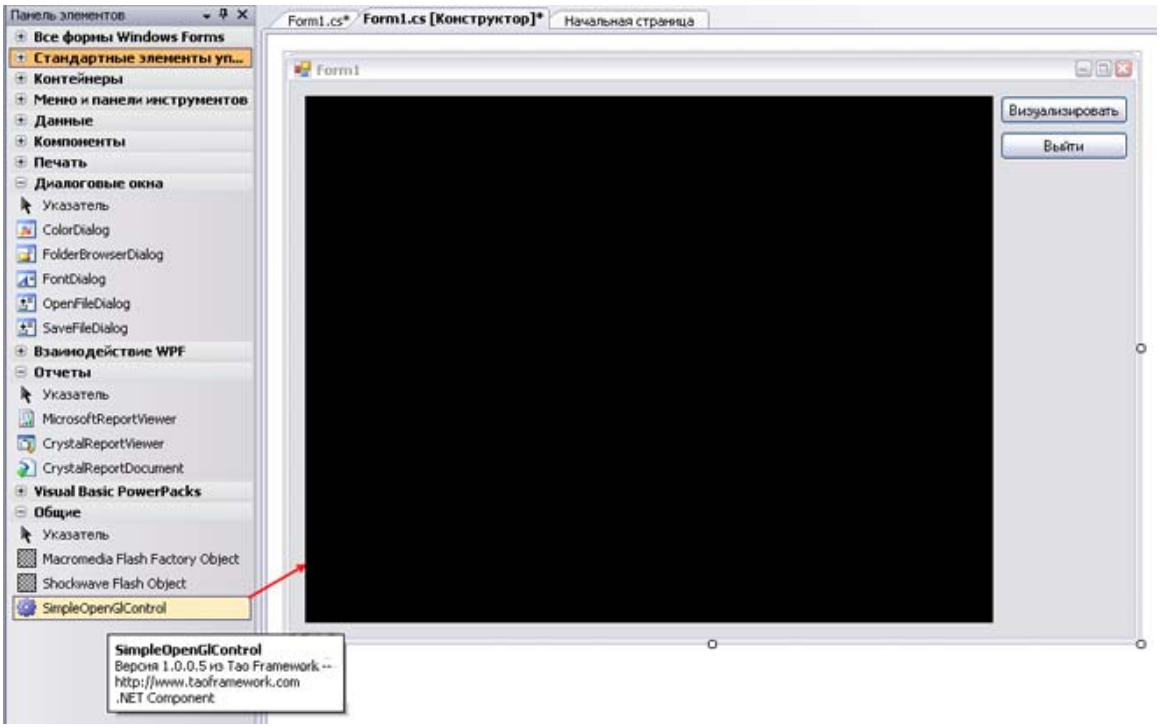


Рис. 4.7

Теперь выделите элемент *simpleOpenGlControl1*, расположенный на форме и перейдите к его свойствам. Измените параметр *name* на значение "*AnT*". Далее элементы *simpleOpenGl Control* мы будем называть *AnT* (рис. 4.8).

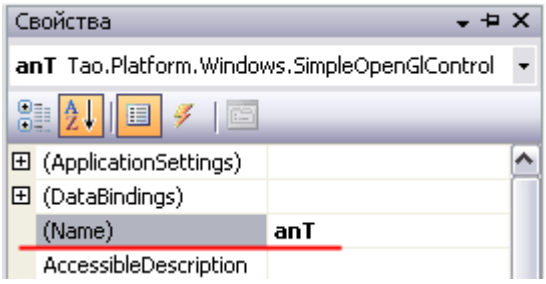


Рис. 4.8

Инициализация *OpenGL* в *C#.NET*

Теперь необходимо инициализировать работу *OpenGl*.

Сначала в конструкторе класса мы должны инициализировать работу элемента *AnT*:

```
public Form1()
{
    InitializeComponent();
    AnT.InitializeContexts();
}
```

Снова перейдите к конструктору и сделайте двойной щелчок левой клавишей мыши на форме – создастся функция обработчик события загрузки формы.

В ней мы поместим код инициализации *OpenGl*. Подробное описание данного кода будет рассмотрено в следующих темах, а сейчас мы только протестируем работу библиотек *OpenGL* и *FreeGLUT*.

```
private void Form1_Load(object sender, EventArgs e)
{
// инициализация Glut
Glut.glutInit();
Glut.glutInitDisplayMode (Glut.GLUT_RGB | Glut.GLUT_DOUBLE |
Glut.GLUT_DEPTH);
// очистка окна
Gl.glClearColor(255, 255, 255, 1);
// установка порта вывода в соответствии с размерами элемента anT
Gl.glViewport(0, 0, AnT.Width, AnT.Height);
// настройка проекции
    Gl.glMatrixMode(Gl.GL_PROJECTION);
    Gl.glLoadIdentity();
    Glu.gluPerspective(45, (float)AnT.Width / (float)AnT.Height, 0.1, 200);
    Gl.glMatrixMode(Gl.GL_MODELVIEW);
    Gl.glLoadIdentity();
```

```
    // настройка параметров OpenGL для визуализации
    Gl.glEnable(Gl.GL_COLOR_MATERIAL);
    Gl.glEnable(Gl.GL_DEPTH_TEST);
    Gl.glEnable(Gl.GL_LIGHTING);
    Gl.glEnable(Gl.GL_LIGHT0);
}
```

На окне мы создали 2 кнопки. Обработчик кнопки "Выйти" будет выглядеть следующим образом:

```
//обработчик кнопки "выйти"
private void button2_Click(object sender, EventArgs e)
{
    Application.Exit();
}
```

Обработчик кнопки «Визуализировать» будет содержать код, реализующий визуализацию сеточного каркаса сферы (за отрисовку трехмерной сферы будет отвечать библиотека *FreeGLUT*). Код, который будет размещен в данной функции, отвечает за разные технические аспекты визуализации, с которыми мы познакомимся в следующих темах.

Код функции:

```
// обработчик кнопки "визуализировать"
private void button1_Click(object sender, EventArgs e)
{
    Gl.glClear(Gl.GL_COLOR_BUFFER_BIT | Gl.GL_DEPTH_BUFFER_BIT);
    Gl.glLoadIdentity();
    Gl.glColor3ub(255, 0, 0);
    Gl.glPushMatrix();
    Gl.glTranslated(0,0,-6);
    Gl.glRotated(45, 1, 1, 0);
    // рисуем сферу с помощью библиотеки FreeGLUT
    Glut.glutWireSphere(2, 32, 32);
    Gl.glPopMatrix();
    Gl.glFlush();
    AnT.Invalidate();
}
```

Откомпилируйте и запустите приложение.

Если при компиляции приложения (по кнопке *F5*) *Microsoft Visual Studio* выдает ошибку с сообщением о том, что не найдена какая-либо библиотека *OpenGL* (например, *freeglut.dll*), необходимо в настройках операционной системы в переменную среды *PATH* добавить каталог *C:\Program*

Files\ TaoFramework\lib. При невозможности это сделать, допустимо из каталога *C:\Program Files\TaoFramework\lib* скопировать все файлы необходимых библиотек (в нашем случае − *freeglut.dll*) в каталог, уже включенный в переменную среды, например *C:\WINDOWS\system32*.

Результат работы приложения показан на рис. 4.9. Если вы правильно набрали исходные коды и выполнили все описанные действия, то после нажатия на кнопке «Визуализировать» вы увидите аналогичную визуализацию сферы.

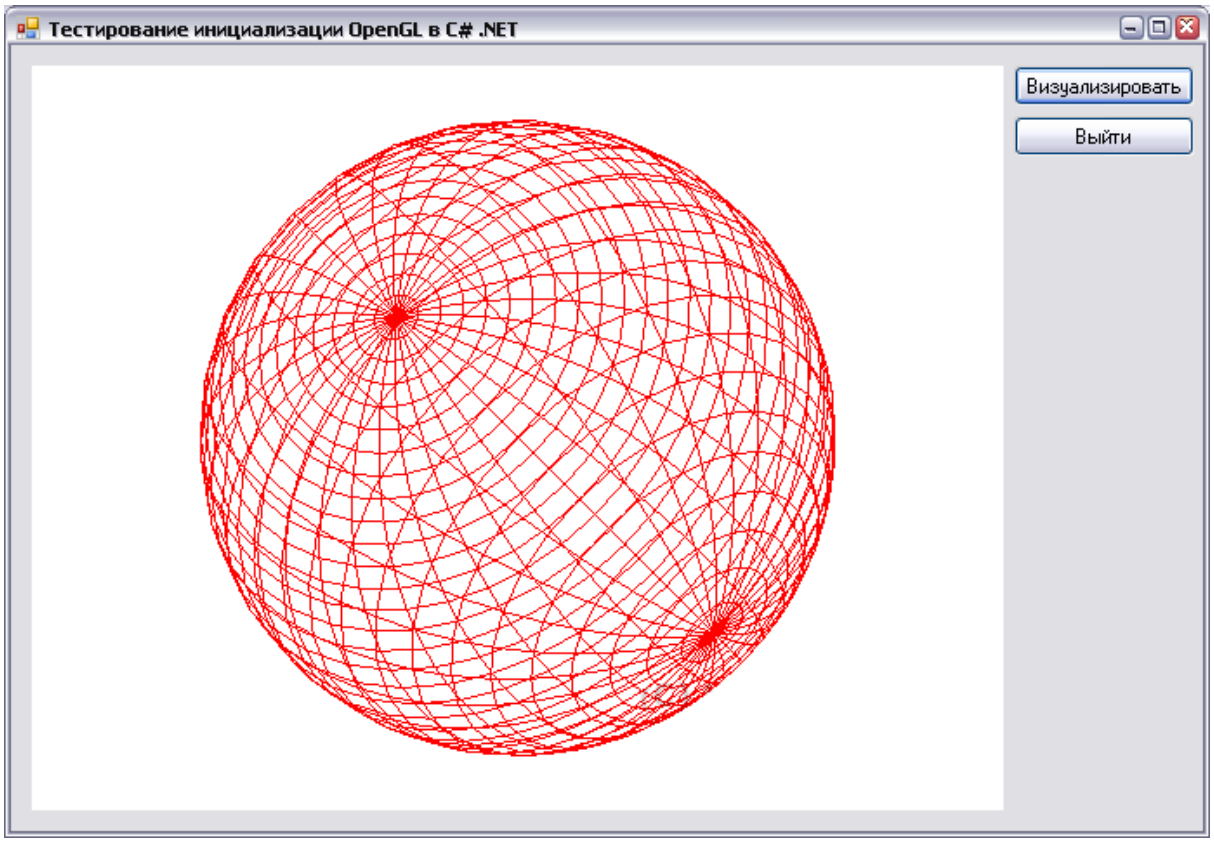


Рис. 4.9

Итак, мы протестировали работоспособность библиотеки *Tao*, инициализировав библиотеку *OpenGL* в *C#.NET*.

Контрольные вопросы

1. Сущность и назначение *OpenGL*.
2. Назначение *Tao Framework*.
3. Порядок установки и подключения библиотек *TAO*.
4. Поддержка *OpenGL* в *VisualC#*.
5. Инициализация *OpenGL* в *C#*.

Тема 5. ИНИЦИАЛИЗАЦИЯ OPENGL

Цель изучения темы. Изучение принципов инициализации *OpenGL* и визуализации объектов при разработке приложений в *C#.*

5.1. Инициализация *OpenGL в C#*

Этот пунк посвящен начальной инициализации *OpenGL*, которая предшествует визуализации любой трехмерной сцены: будет рассмотрен код приложения и объяснено, каким образом происходят инициализация *OpenGL* и визуализация объектов.

Инициализация *OpenGl*

После того как объект *SimpleOpenGLControl* прошел инициализацию, стартует загрузка формы. Мы всегда будем отслеживать это событие, так как именно здесь выполняется начальная настройка программы.

Для визуализации трехмерной сцены в предыдущей теме мы использовали следующий код.

```
private void Form1_Load(object sender, EventArgs e)
{
    // инициализация Glut
    Glut.glutInit();
    Glut.glutInitDisplayMode(Glut.GLUT_RGB | Glut.GLUT_DOUBLE |
Glut.GLUT_DEPTH);
    // очистка окна
    Gl.glClearColor(255, 255, 255, 1);
    // установка порта вывода в соответствии с размерами элемента AnT
    Gl.glViewport(0, 0, AnT.Width, AnT.Height);
    // настройка проекции
    Gl.glMatrixMode(Gl.GL_PROJECTION);
    Gl.glLoadIdentity();
    Glu.gluPerspective(45, (float)AnT.Width / (float)AnT.Height, 0.1, 200);
    Gl.glMatrixMode(Gl.GL_MODELVIEW);
    Gl.glLoadIdentity();
    // настройка параметров OpenGL для визуализации
    Gl.glEnable(Gl.GL_DEPTH_TEST);
    Gl.glEnable(Gl.GL_LIGHTING);
    Gl.glEnable(Gl.GL_LIGHT0);
}
```

Здесь в первую очередь выполняется инициализация библиотеки *Glut*.

Как видно из кода, для работы с функциями библиотеки *OpenGL* используется класс *Gl*, находящийся в пространстве имен *Tao.OpenGL*.

Для работы с функциями библиотеки *Glut* используется класс *Glut*.

Таким образом, по сравнению с использованием этих библиотек в *C++* мы всего лишь вызываем методы из классов соответствующих библиотек, где они очень удобно описаны.

Мы обязательно должны вызвать функцию *glutInit()* перед тем, как начнем использовать любые другие функции данной библиотеки, так как эта функция производит инициализацию библиотеки *Glut*.

Далее мы вызываем функцию

Glut.glutInitDisplayMode(Glut.GLUT_RGB | Glut.GLUT_DOUBLE | Glut.GLUT_DEPTH);

Эта функция устанавливает режим отображения. В нашем случае устанавливается режим *RGB* для визуализации (*GLUT_RGB* – это псевдоним *GLUT_RGBA*, он устанавливает режим *RGBA* битовой маски окна). Далее мы устанавливаем двойную буферизацию окна, которая, как правило, используется для устранения мерцания, возникающего в процессе быстрой перерисовки кадров несколько раз подряд. *GLUT_DEPTH* указывает при инициализации окна, будет ли в приложении использоваться буфер глубины.

После инициализации окна мы устанавливаем цвет очистки окна с помощью функции

Gl.glClearColor(255, 255, 255, 1);

Перед дальнейшим изучением кода перечислим этапы создания сцены в *OpenGL*:

1. Позиционирование объема видимости в пространстве (установка координат камеры).
2. Установка в данном пространстве модели (объекта), которая будет попадать в объем видимости нашей камеры.
3. Проецирование, которое определяет форму объема видимости.
4. В рамках порта просмотра мы получаем изображение объекта.

Теперь нам необходимо определить значение порта вывода, устанавливая значения в функции

Gl.glViewport(0, 0, AnT.Width, AnT.Height);

Здесь мы указываем библиотеке *OpenGL* на то, что вывод будет осуществляться во всей области элемента *AnT* (элемент, расположенный на форме для визуализации в него сцены). Определяем тот самый порт просмотра, в который последним шагом будет визуализироваться модель.

После этого происходит настройка проекции. Для этого мы сначала вызываем функцию

Gl.glMatrixMode(Gl.GL_PROJECTION);

Функция *glMatrixMode* предназначена для того, чтобы задавать матричный режим: будет определена матрица, над которой в дальнейшем будут производиться операции. В нашем случае это *GL_PROJECTION* – матрица проекций.

Следующей командой мы очищаем матрицу с помощью функции *glLoadIdentity* (функция заменяет текущую матрицу на единичную). Далее мы устанавливаем тип текущей проекции с помощью функции *gluPerspective*.

```
Gl.glLoadIdentity();
Glu.gluPerspective(45, (float)AnT.Width / (float)AnT.Height, 0.1, 200);
```

Функция *gluPerspective* определена в библиотеке *GLU – OpenGL Utility Library (GLU)*. Эта библиотека является надстройкой над библиотекой *OpenGL*, реализующей ряд более продвинутых функций. Она также является свободно распространяемой и поставляется вместе с библиотекой *OpenGL*.

Данная функция строит пирамиду охвата видимости, основываясь на угле визуального охвата, отношении сторон порта просмотра и установке ближней и дальней плоскости просмотра (рис. 5.1).

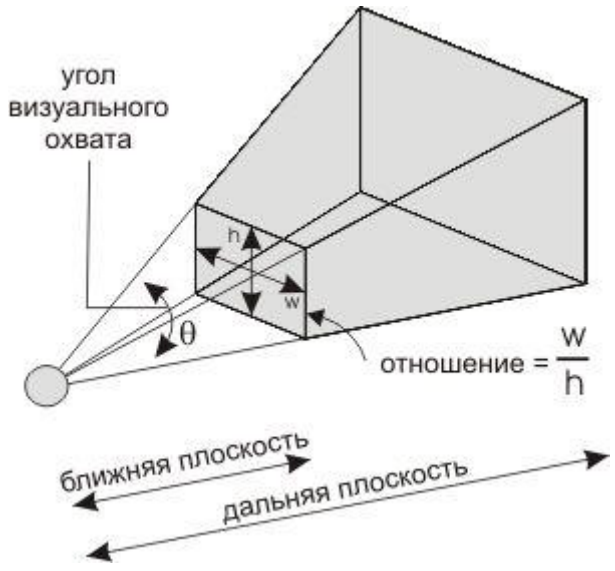


Рис. 5.1

Теперь, когда проекция определена, мы устанавливаем в качестве текущей матрицы объектно-видовую матрицу и очищаем ее.

```
Gl.glMatrixMode(Gl.GL_MODELVIEW);
Gl.glLoadIdentity();
```

Теперь остается только включить некоторые опции, необходимые для корректной визуализации сцены. Это тест глубины, а также отображение цветов материалов.

```
Gl.glEnable(Gl.GL_DEPTH_TEST);
Gl.glEnable(Gl.GL_COLOR_MATERIAL);
```

Визуализация объектов

Теперь рассмотрим функцию, визуализирующую трехмерную сферу.

Код этой функции:

```
// обработчик кнопки "визуализировать"
private void button1_Click(object sender, EventArgs e)
{
   Gl.glClear(Gl.GL_COLOR_BUFFER_BIT |
Gl.GL_DEPTH_BUFFER_BIT);
   Gl.glLoadIdentity();
   Gl.glPushMatrix();
   Gl.glTranslated(0,0,-6);
   Gl.glRotated(45, 1, 1, 0);
   // рисуем сферу с помощью библиотеки FreeGLUT
   Glut.glutWireSphere(2, 32, 32);
   Gl.glPopMatrix();
   Gl.glFlush();
   AnT.Invalidate();
}
```

Когда пользователь нажимает на кнопку и вызывается данная функция, первым делом производится очистка окна (так как до этого уже мог быть реализован какой-либо вывод; очистка экрана перед визуализацией кадра − это стандартный шаг).

Для этого используется функция *glClear*. В качестве параметра функция получает наименования буферов, которые необходимо очистить; в нашем случае это буфер цвета и буфер глубины.

Далее мы очищаем объектно-видовую матрицу – таким образом камера устанавливается в начало координат. Теперь мы можем совершить ее перемещение в пространстве.

Но перед этим мы вызываем функцию

Gl.glColor3f(255, 0, 0);

устанавливающую красный цвет для отрисовки (основывается на *RGB* составляющих).

Займемся перемещением. Функция *glPushMatrix* помещает текущую матрицу в стек матриц, откуда в дальнейшем мы сможем ее вернуть с помощью функции *glPopMatrix*.

Таким способом мы осуществим перемещение отрисовываемого объекта в пространстве, не изменив саму матрицу, отвечающую за положение камеры (наблюдателя).

Если не использовать такой подход, то с каждым визуализированным кадром камера будет перемещаться и очень скоро мы вообще не сможем ее найти.

Сохранив матрицу в стеке, мы производим перемещение объекта на 6 единиц по оси *Z*, после чего выполняем поворот сцены на 45 градусов сразу по двум осям

```
Gl.glTranslated(0,0,-6);
Gl.glRotated(45, 1, 1, 0);
```

После этого мы выполняем рисование объекта в той области, куда мы переместились (сфера радиусом 2 в виде сетки). Сфера будет разбита на 32 меридиана и 32 параллели.

Для визуализации используется библиотека *FreeGlut*.

```
// рисуем сферу с помощью библиотеки FreeGLUT
Glut.glutWireSphere(2, 32, 32);
```

Возвращаем сохраненную в стеке матрицу

Gl.glPopMatrix();

Дожидаемся, пока библиотека *OpenGL* завершит визуализацию этого кадра

Gl.glFlush();

и посылаем нашему элементу *AnT*, в котором происходит визуализация сцены, сигнал о том, что необходимо обновить отображаемый кадр, т.е. вызываем его перерисовку.

AnT.Invalidate();

На этом визуализация объекта заканчивается.

5.2. Визуализация *2D*-примитивов

Процесс рисования всех объектов в компьютерной графике сводится к вычислению координат точек, которые затем соединяются линиями, образуя рисунок. Точками же создаются линии, из которых могут быть постро-

ены простейшие геометрические фигуры – полигоны. Полигоны могут создавать сетку, которая представляет в трехмерном пространстве какую-либо модель.

Полигон – это область в пространстве, которая ограничивается одной замкнутой ломаной линией, причем каждый отрезок данной ломаной линии задается с помощью координат двух точек − начальной и конечной. Описанный таким образом полигон может оказаться очень сложным геометрическим объектом.

Полигон обязательно должен быть выпуклым, поэтому его ребра не должны пересекаться.

При условии соблюдения двух предыдущих условий нет никаких ограничений на то, из скольких точек состоит полигон.

Для рисования примитивов в *OpenGL* мы будем использовать различные режимы визуализации, но сначала рассмотрим, как происходит указание вершин в *OpenGl*.

Для указания вершин используется специальная команда *glVertex*()*

Данная функция имеет следующий формат:

void glVertex{234}{ifd}(TYPE coords);

Одной цифрой (2, 3 или 4) обозначается то количество координат, которыми будет задаваться вершина. В *OpenGL* точки задаются 4 координатами: *x, y, z, r*. Таким образом, положение точки описывается как *x/r*, *y/r*, *z/r*. Когда параметр *r* не указывается, он принимается равным единице. Если указываются точки в двухмерной системе координат, параметр *z* считается равным нулю.

Символ *i*, *f* или *d* обозначает тип принимаемых значений. Например, если написать команду *glVertex3f(,,)* то в скобках должны последовать три переменных типа *float*. Для *d – double*, для *i – int*.

Вызов функции *glVertex* может привести к желаемому результату, только если он сделан между вызовами функций *glBegin()* и *glEnd()*.

Для построения объекта из точек используются разные режимы, которые описываются различными правилами объединения вершин в полигоны.

Такие режимы рисования возможны лишь с помощью перечисления точек между командами *glBegin()* и *glEnd()*, причем при вызове функции *glBegin* указывается, в каком режиме будут соединяться полигоны.

Значения параметров, которые может принимать *glBegin*:

GL_POINTS − на месте каждой указанной с помощью команды *glVertex* вершины рисуется точка.

GL_LINES − каждые 2 вершины объединяются в отрезок.

GL_LINE_STRIP − вершины соединяются последовательно, образуя кривую линию.

GL_LINE_LOOP − то же, что и *GL_LINE_STRIP*, но от последней точки будет идти отрезок к начальной точке.

GL_TRIANGLES − каждые три вершины объединятся в треугольник.

GL_TRIANGLE_STRIP − треугольники рисуются таким образом, что последнее ребро треугольника является начальным ребром следующего.

GL_TRIANGLE_FAN − то же, что и *GL_TRIANGLE_STRIP*, только изменен порядок следования вершин.

GL_QUADS − каждые четыре вершины объединяются в квадрат.

GL_QUAD_STRIP − вершины объединяются в квадраты, причем последнее ребро квадрата является первым ребром следующего.

GL_POLYGON − рисуется полигон на основе вершин; вершин не должно быть менее 3 и должны отсутствовать самопересечения, а также должно соблюдаться условие выпуклости.

Теперь попробуем нарисовать какой-либо рисунок. Для этого создайте приложение, основываясь на программе из п. 4.3 (а именно подключите необходимые ссылки к библиотекам *Tao*, подключите необходимые пространства имен, расположите элементы на окне, а также назначьте необходимые обработчики событий и инициализируйте необходимые элементы).

Эта программа будет отличаться от программы, написанной в п. 4.3, лишь тем, что здесь будут использоваться другие проекции в функции *Form1_Load* и код, отвечающий за визуализацию в функции *button1_Click*.

Перейдя к свойствам формы, измените название окна на «Рисование 2D-примитивов».

Рассмотрим код функции *Form1_Load*. Теперь он будет выглядеть следующим образом.

```
private void Form1_Load(object sender, EventArgs e)
{
// инициализация Glut
Glut.glutInit();
Glut.glutInitDisplayMode(Glut.GLUT_RGB | Glut.GLUT_DOUBLE |
Glut.GLUT_DEPTH);
// очистка окна
Gl.glClearColor(255, 255, 255, 1);
// установка порта вывода в соответствии с размерами элемента AnT
```

```
Gl.glViewport(0, 0, AnT.Width, AnT.Height);
    // настройка проекции
    Gl.glMatrixMode(Gl.GL_PROJECTION);
    Gl.glLoadIdentity();
    // теперь необходимо корректно настроить 2D ортогональную проекцию
    // в зависимости от того, какая сторона больше
    // мы немного варьируем то, как будут сконфигурированы настройки проекции
    if ((float)AnT.Width <= (float)AnT.Height)
    {
        Glu.gluOrtho2D(0.0, 30.0 * (float)AnT.Height / (float)AnT.Width, 0.0, 30.0);
    }
    else
    {
        Glu.gluOrtho2D(0.0, 30.0 * (float)AnT.Width / (float)AnT.Height, 0.0, 30.0);
    }
    Gl.glMatrixMode(Gl.GL_MODELVIEW);
    Gl.glLoadIdentity();
    // настройка параметров OpenGL для визуализации
    Gl.glEnable(Gl.GL_DEPTH_TEST);
    Gl.glEnable(Gl.GL_COLOR_MATERIAL);
}
```

Как видно из кода, за настройку 2D ортогональной проекции отвечает функция *gluOrtho2D*, реализация которой предоставлена библиотекой *Glu*. Эта функция помещает начало координат в самый левый нижний квадрат, а камера (наблюдатель) в таком случае находится на оси *Z*; таким образом визуализируется графика в 2D-режиме.

Мы должны передать в качестве параметров координаты области видимости окна проекции: *left*, *right*, *bottom* и *top*. В нашем случае мы передаем параметры таким образом, чтобы исключить искажения, связанные с тем, что область вывода не квадратная, поэтому параметр *top* рассчитывается как произведение числа 30 и отношения высоты элемента *AnT* (в котором будет идти визуализация) к его ширине.

Если высота больше ширины, используется отношение ширины к высоте.

Теперь рассмотрим код функции *button1_Click*.

В ней будет производиться очистка буфера цвета кадра, после чего будет очищаться текущая объектно-видовая матрица. Затем мы будем в режиме рисования линий задавать координаты для того, чтобы нарисовать объект − букву А.

После этого мы перерисовываем содержимое элемента *AnT*.

Перед тем как нарисовать наш объект (букву А), попробуем просто провести линию из начала координат в противоположенный угол окна.

Код такой функции будет выглядеть следующим образом.

```
private void button1_Click(object sender, EventArgs e)
{
// очищаем буфер цвета и глубины
Gl.glClear(Gl.GL_COLOR_BUFFER_BIT | Gl.GL_DEPTH_BUFFER_BIT);
// очищаем текущую матрицу
Gl.glLoadIdentity();
// устанавливаем текущий цвет - красный
Gl.glColor3f(255, 0, 0);
// активируем режим рисования линий на основе
// последовательного соединения всех вершин в отрезки
Gl.glBegin(Gl.GL_LINE_STRIP);
// первая вершина будет находиться в начале координат
Gl.glVertex2d(0, 0);
// теперь в зависимости от того, как была определена проекция,
if ((float)AnT.Width <= (float)AnT.Height)
{
// рисуем вторую вершину в противоположенном углу
Gl.glVertex2d(30.0f * (float)AnT.Height / (float)AnT.Width, 30);
}
else
{
// рисуем вторую вершину в противоположенном углу
Gl.glVertex2d(30.0f * (float)AnT.Width / (float)AnT.Height, 30);
}
// завершаем режим рисования
Gl.glEnd();
// дожидаемся конца визуализации кадра
Gl.glFlush();
// посылаем сигнал перерисовки элемента AnT.
AnT.Invalidate();
}
```

Теперь если откомпилировать проект и запустить приложение (*F5*), а затем нажать на кнопку «визуализировать», мы увидим следующий результат (рис. 5.2).

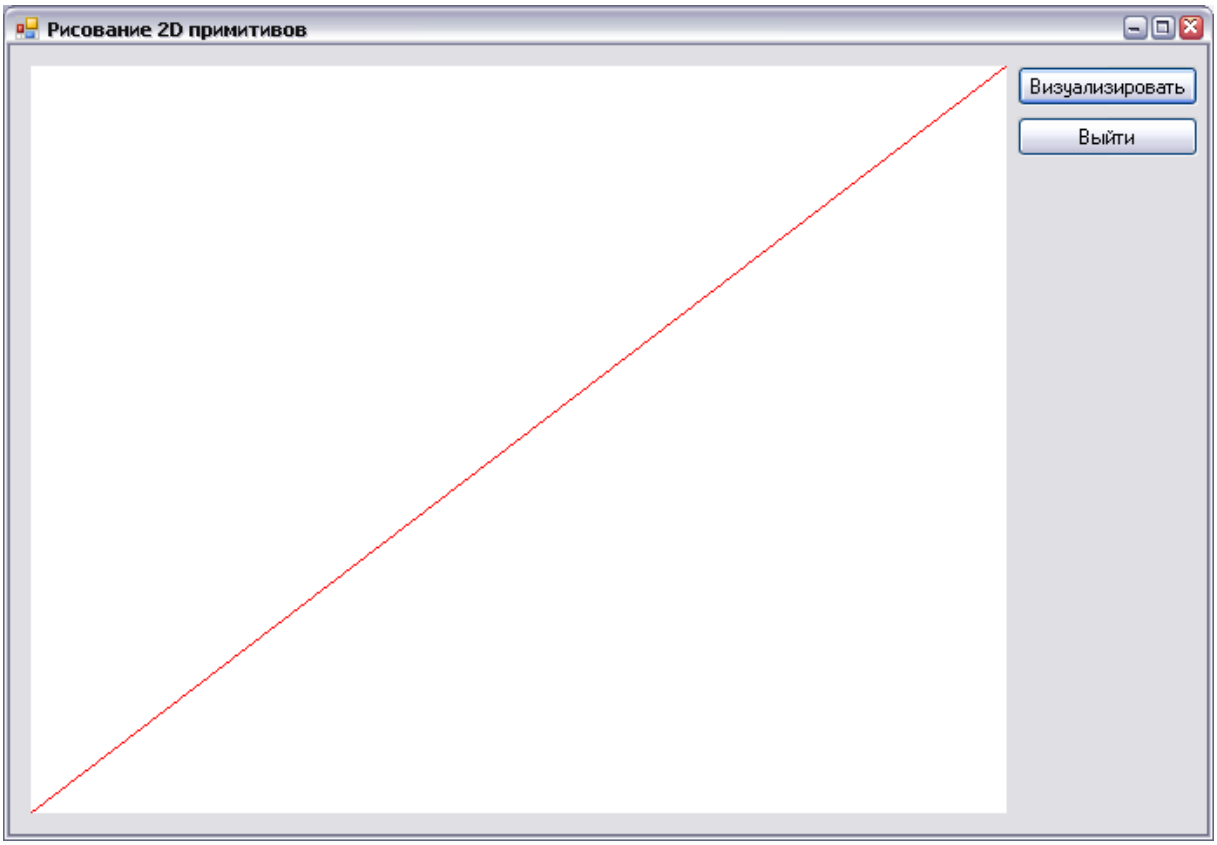


Рис. 5.2

Теперь заменим код

```
Gl.glBegin(Gl.GL_LINE_STRIP);
// первая вершина будет находиться в начале координат
Gl.glVertex2d(0, 0);
// теперь в зависимости от того, как была определена проекция
if ((float)AnT.Width <= (float)AnT.Height)
{
    // рисуем вторую вершину в противоположенном углу
    Gl.glVertex2d(30.0f * (float)AnT.Height / (float)AnT.Width, 30);
}
else
{
    // рисуем вторую вершину в противоположенном углу
    Gl.glVertex2d(30.0f * (float)AnT.Width / (float)AnT.Height, 30);
}
// завершаем режим рисования
Gl.glEnd();
```

на код, который будет рисовать букву А.

Нам необходимо правильно указать координаты вершин. Букву будем рисовать с помощью двух линий.

Форма объекта и координаты соответствующих вершин представлены на рис. 5.3.

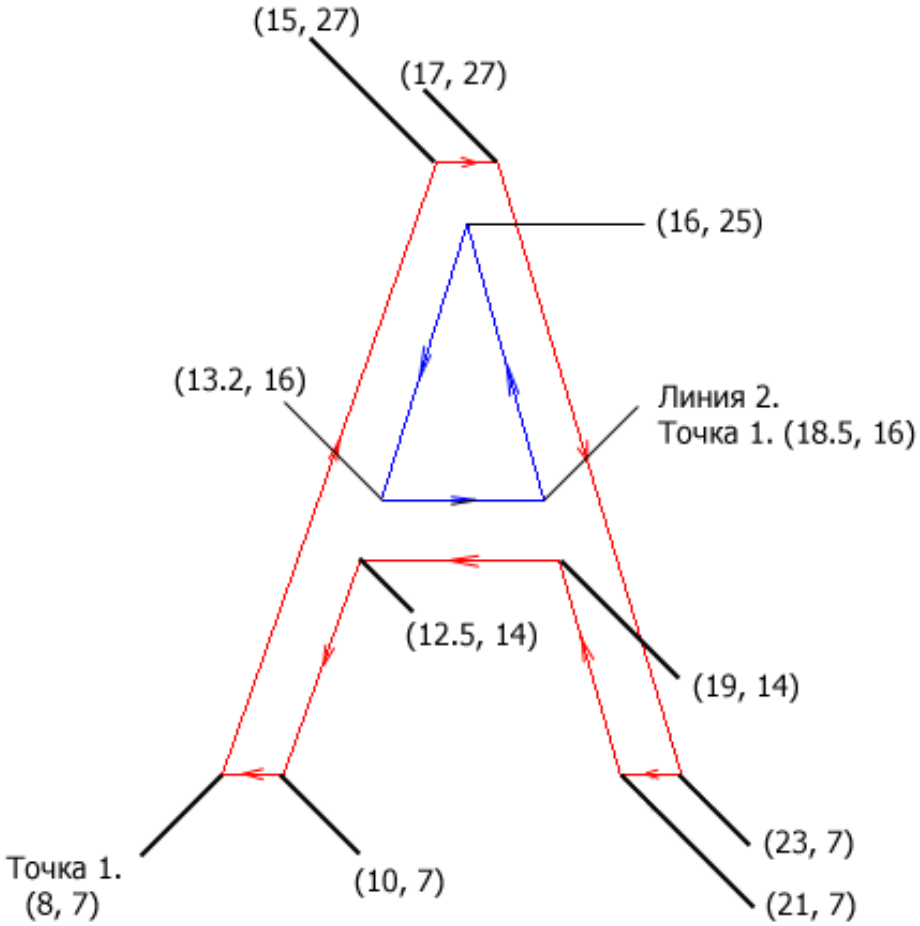


Рис. 5.3

Код рисования буквы А (буква будет рисоваться двумя линиями)

```
// активируем режим рисования линий на основе
// последовательного соединения всех вершин отрезками
Gl.glBegin(Gl.GL_LINE_LOOP);
// первая вершина будет находиться в начале координат
Gl.glVertex2d(8, 7);
Gl.glVertex2d(15, 27);
Gl.glVertex2d(17, 27);
Gl.glVertex2d(23, 7);
Gl.glVertex2d(21, 7);
Gl.glVertex2d(19, 14);
Gl.glVertex2d(12.5, 14);
Gl.glVertex2d(10, 7);
Gl.glEnd();
```

```
// вторая линия
Gl.glBegin(Gl.GL_LINE_LOOP);
Gl.glVertex2d(18.5, 16);
Gl.glVertex2d(16, 25);
Gl.glVertex2d(13.2, 16);
Gl.glEnd();
```

Пример нарисованной в программе буквы представлен на рис. 5.4.

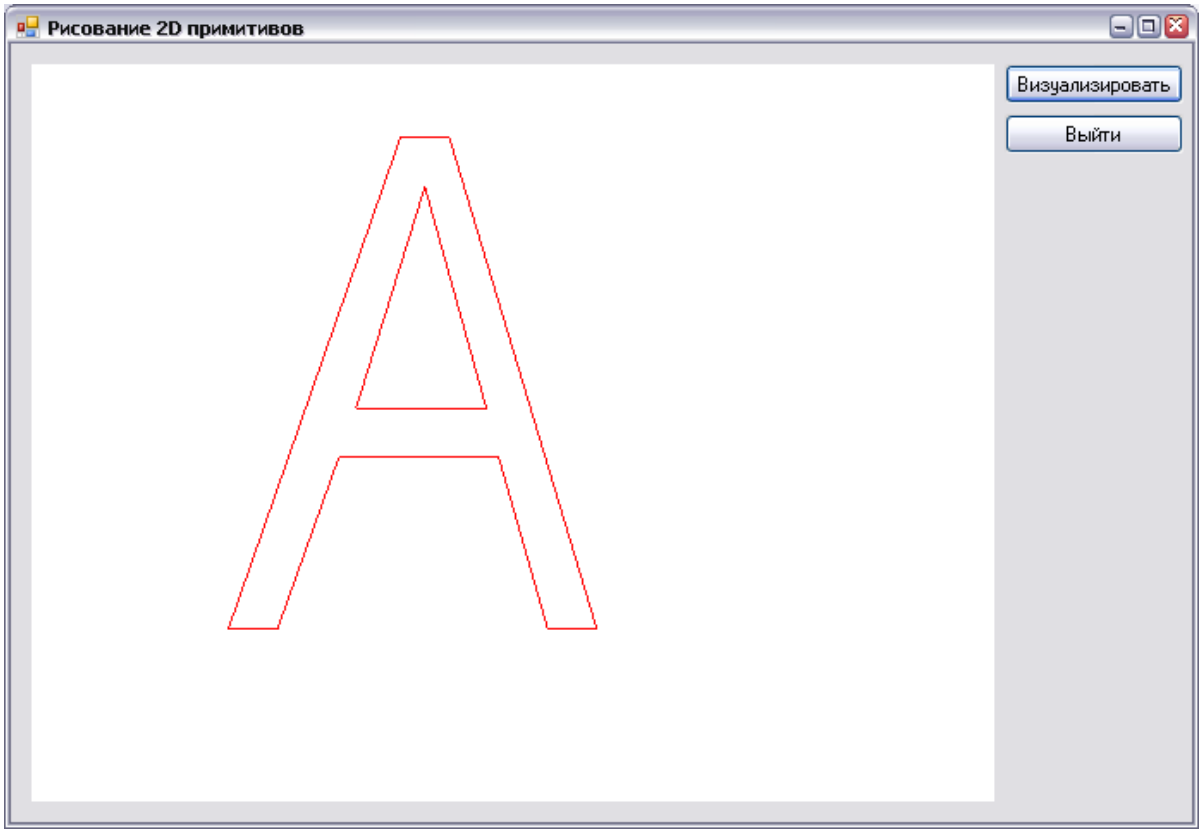


Рис. 5.4

5.3. Визуализация графика функции

Рассмотрим процесс создания программы, задачей которой будет визуализация графика заданной функции с анимационной демонстрацией изменения значения функции на графике.

Это будет реализовано следующим образом: по графику двигается красная точка, принимающая значения y для заданного x в нашем графике (по всей видимой области).

Кроме того, возле курсора будут визуализироваться его координаты (рис. 5.5).

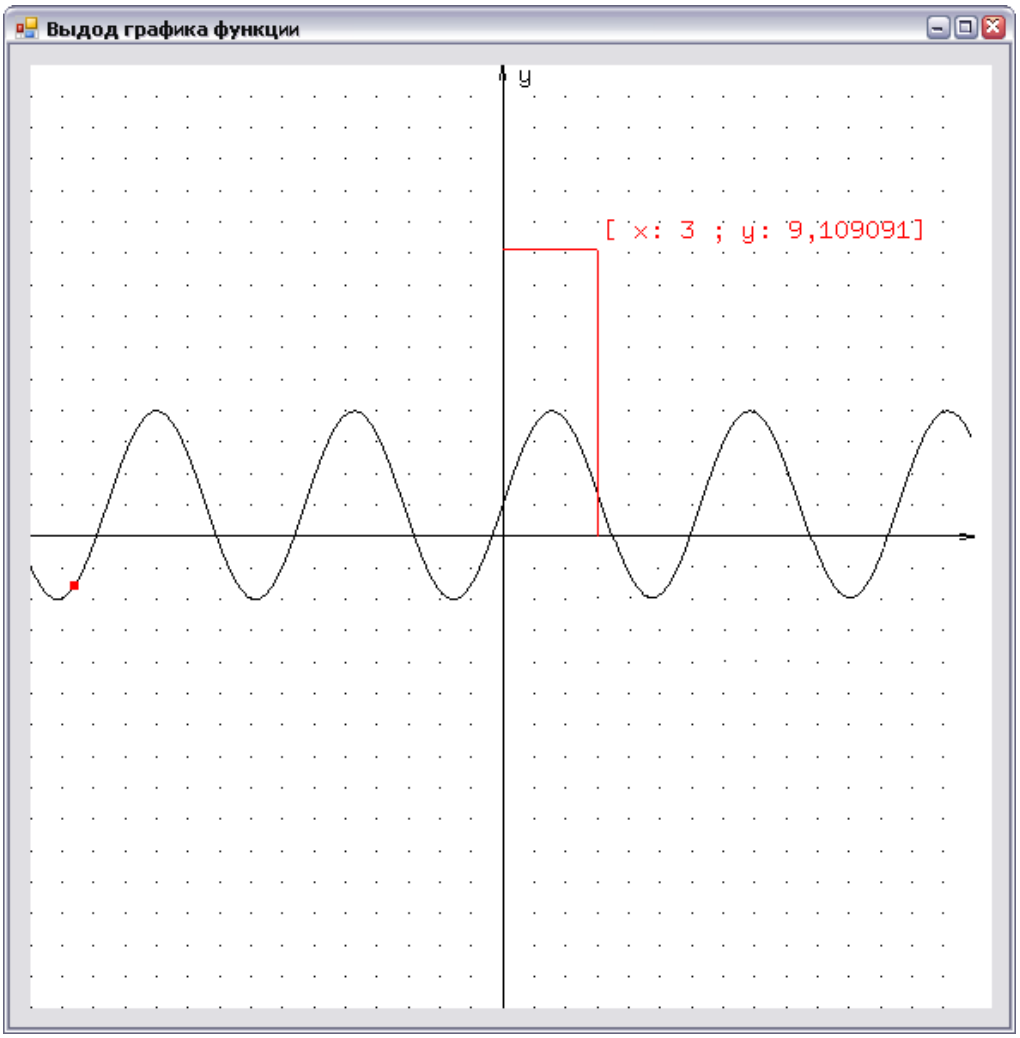


Рис. 5.5

Создайте основу приложения, как это было описано в п. 4.3, но не добавляйте кнопки «Визуализировать» и «Закрыть», ограничьтесь элементом *SimpleOpenGLControl*.

Окно должно иметь форму, представленную на рис. 5.5.

Добавьте в проект один таймер. Для того чтобы добавить этот объект, перейдите к окну *ToolBox*, выберите в свитке «Компоненты» элемент *Timer* (рис. 5.6) и перетащите его на форму.

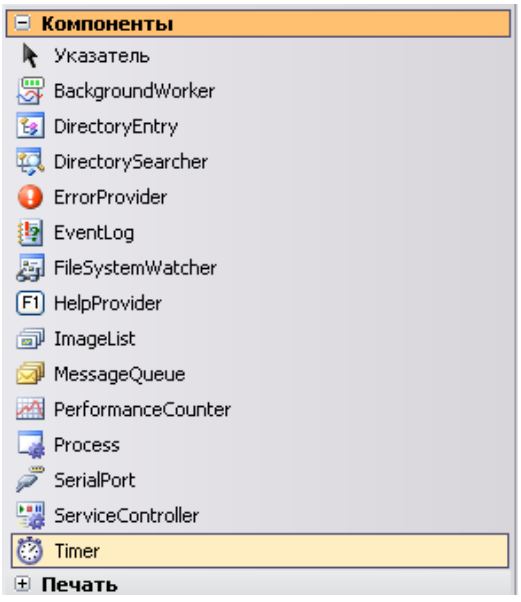


Рис. 5.6

Место, куда вы перетащите этот элемент, не важно, поскольку последний не занимает его на форме и отобразится в полоске таких же («не занимающих место») объектов под формой.

Щелкните по элементу и перейдите в его свойства.

Смените параметр *name* в свойствах таймера на значение *PointInGrap*, а параметр *Interval* − на значение 30 (раз в 30 мс будет вызываться событие *OnTimer*).

Теперь щелкните по значку таймера на форме двойным щелчком левой клавиши мыши. Создастся функция *PointInGrap_Tick*, отвечающая за обработку события *OnTimer*. Отсюда мы будем затем вызывать функцию *Draw*.

Теперь, когда основа приложения создана, мы перейдем к исходному коду.

Он будет основан на семи функциях, которые мы сейчас рассмотрим, но сначала перед кодом функции-конструктора класса добавьте инициализацию следующих переменных:

```
// размеры окна
double ScreenW, ScreenH;
// отношения сторон окна визуализации
// для корректного перевода координат мыши в координаты,
// принятые в программе
private float devX;
private float devY;
// массив, который будет хранить значения x,y точек графика
private float[,] GrapValuesArray;
// количество элементов в массиве
private int elements_count = 0;
// флаг, означающий, что массив со значениями координат графика пока еще не заполнен
private bool not_calculate = true;
// номер ячейки массива, из которой будут взяты координаты для красной точки
// для визуализации текущего кадра
private int pointPosition = 0;
// вспомогательные переменные для построения линий от курсора мыши к координатным осям
float lineX, lineY;
// текущие координаты курсора мыши
float Mcoord_X = 0, Mcoord_Y = 0;
```

Перед каждой переменной в комментарии приведено ее назначение.

Теперь вернемся к нашим семи функциям.

Первая функция – это обработчик события загрузки формы − *Form1_Load*. Здесь при загрузке приложения будет произведена инициализация *OpenGL* для последующей визуализации.

Инициализацию *OpenGL* с двухмерной проекцией мы рассматривали ранее. Код этой функции с более подробными комментариями:

```
private void Form1_Load(object sender, EventArgs e)
{
    // инициализация библиотеки glut
    Glut.glutInit();
    // инициализация режима экрана
    Glut.glutInitDisplayMode(Glut.GLUT_RGB | Glut.GLUT_DOUBLE);
    // установка цвета очистки экрана (RGBA)
    Gl.glClearColor(255, 255, 255, 1);
    // установка порта вывода
    Gl.glViewport(0, 0, AnT.Width, AnT.Height);
    // активация проекционной матрицы
    Gl.glMatrixMode(Gl.GL_PROJECTION);
    // очистка матрицы
    Gl.glLoadIdentity();
    // определение параметров настройки проекции в зависимости от размеров сторон
элемента AnT.
    if ((float)AnT.Width <= (float)AnT.Height)
    {
        ScreenW = 30.0;
        ScreenH = 30.0 * (float)AnT.Height / (float)AnT.Width;
        Glu.gluOrtho2D(0.0, ScreenW, 0.0, ScreenH);
    }
    else
    {
        ScreenW = 30.0 * (float)AnT.Width / (float)AnT.Height;
        ScreenH = 30.0;
        Glu.gluOrtho2D(0.0, 30.0 * (float)AnT.Width / (float)AnT.Height, 0.0, 30.0);
    }
    // сохранение коэффициентов, которые нам необходимы для перевода координат
указателя в оконной системе, в координаты
    // принятые в нашей OpenGL сцене
    devX = (float)ScreenW / (float)AnT.Width;
    devY = (float)ScreenH / (float)AnT.Height;
    // установка объектно-видовой матрицы
    Gl.glMatrixMode(Gl.GL_MODELVIEW);
    // старт счетчика, отвечающего за вызов функции визуализации сцены
    PointInGrap.Start();
}
```

Теперь обратимся к функции *PointInGrap_Tick*. Эта функция вызывается с задержкой в 30 мс.

В ней определяется, из какого элемента массива с координатами графика мы возьмем координаты, которые используем для рисования красной точки.

Отсюда также вызывается функция *Draw*, отвечающая за визуализацию.

Код этой функции:

```
// функция обработчик события таймера
private void PointInGrap_Tick(object sender, EventArgs e)
{
    // если мы дошли до последнего элемента массива
    if (pointPosition == elements_count-1)
    pointPosition = 0; // переходим к начальному элементу
    // функция визуализации
    Draw();
    // переход к следующему элементу массива
    pointPosition++;
}
```

Перед тем как перейти к функциям, отвечающим за визуализацию, рассмотрим несколько небольших вспомогательных функций.

Начнем с функции *AnT_MouseMove*.

Эта функция добавляется созданием события *MouseMove* для элемента *SimpleOpnGLControl (AnT)*. Событие создается аналогично тому, как мы его создавали в п. 2.2. Только в данном случае мы переходим к свойствам элемента *AnT* и уже в них переходим во вкладку *Event,* добавляем событие *MouseMove*.

В данной функции мы производим сохранение текущих координат мыши, чтобы в будущем использовать их при визуализации графика, а также производим вычисление размеров двух красных линий, которые будут по нормалям соединять координаты указателя мыши с координатными осями (рис. 5.5).

Код этой функции выглядит следующим образом:

```
// обработка движения мыши над элементом AnT
private void AnT_MouseMove(object sender, MouseEventArgs e)
{
    // сохраняем координаты мыши
    Mcoord_X = e.X;
    Mcoord_Y = e.Y;
    // вычисляем параметры для будущей дорисовки линий от указателя мыши к координатным осям
    lineX = devX * e.X;
    lineY = (float)(ScreenH - devY * e.Y);
}
```

Теперь рассмотрим функцию, которая будет осуществлять визуализацию текстовых строк.

Эта функция устанавливает координаты вывода растровых символов в соответствии с координатами, переданными в параметрах *x* и *y*, а затем в цикле перебирает все символы из указанной в параметре строки текста. Каждый символ визуализируется с помощью функции *glutBitmapCharacter*. В этой функции указывается шрифт для вывода и переменная типа *char* для визуализации.

Код функции выглядит следующим образом:

```
// функция визуализации текста
private void PrintText2D(float x, float y, string text)
{
    // устанавливаем позицию вывода растровых символов
    // в переданных координатах x и y
    Gl.glRasterPos2f(x, y);
    // в цикле foreach перебираем значения из массива text,
    // который содержит значение строки для визуализации
    foreach (char char_for_draw in text)
    {
        // визуализируем символ с помощью функции glutBitmapCharacter,
        // используя шрифт GLUT_BITMAP_9_BY_15
        Glut.glutBitmapCharacter(Glut.GLUT_BITMAP_9_BY_15, char_for_draw);
    }
}
```

Следующая функция, которая нам понадобится, − это функция, вычисляющая координаты для построения графика. В ней инициализируется массив координат и производится вычисление всех координат графика в зависимости от указанного диапазона значений *x* и шага приращения этих значений.

Обратите внимание на то, что при инициализации массива для хранения координат должно быть указано такое количество элементов массива, чтобы в дальнейшем их хватило для размещения всех координат, иначе произойдет исключение, так как программа в процессе работы попытается обратиться к области памяти, которая ей не принадлежит.

Код этой функции с комментариями:

```
// функция, производящая вычисления координат графика
// и заносящая их в массив GrapValuesArray
private void functionCalculation()
{
```

```
    // определение локальных переменных X и Y
    float x = 0, y = 0;
    // инициализация массива, который будет хранить значение 300 точек
    // из которых будет состоять график
    GrapValuesArray = new float[300, 2];
    // счетчик элементов массива
    elements_count = 0;
    // вычисления всех значений y для x, принадлежащего промежутку от -15 до 15, с
шагом в 0.01f
    for (x = -15; x < 15; x += 0.1f)
    {
        // вычисление y для текущего x
        // по формуле y = (float)Math.Sin(x)*3 + 1;
        // эта строка задает формулу, описывающую график функции для нашего урав-
нения y = f(x).
        y = (float)Math.Sin(x)*3 + 1;
        // запись координаты x
        GrapValuesArray[elements_count, 0] = x;
        // запись координаты y
        GrapValuesArray[elements_count, 1] = y;
        // подсчет элементов
        elements_count++;
    }
    // изменяем флаг, сигнализировавший о том, что координаты графика не вычисле-
ны
    not_calculate = false;
}
```

Рассмотрим функцию, выполняющую визуализацию графика.

Так как визуализацию графика можно отнести к конкретной подзадаче функции визуализации сцены, вынесем визуализацию графика в отдельную функцию (чтобы не загромождать функцию визуализации сцены).

В этой функции сначала будет проверен флаг, сигнализирующий о том, что координаты графика вычислены и занесены в массив (переменная *not_calculate*). В том случае если флаг указывает, что просчета значений еще не было, вызывается функция, которая посчитает значения координат точек графика и заполнит ими массив.

Далее реализуется проход циклом *for* по массиву значений координат точек графика и их визуализация, причем мы визуализируем не точки, а соединяем эти точки линией, основываясь на значении координат точек (вершины линии).

По завершению отрисовки графика производится рисование красной точки в тех координатах, до которых мы дошли, последовательно перебирая значения элементов в массиве координат.

Исходный код данной функции:

```
// визуализация графика
private void DrawDiagram()
{
    // проверка флага, сигнализирующего о том, что координаты графика вычислены
    if (not_calculate)
    {
        // если нет, то вызываем функцию вычисления координат графика
        functionCalculation();
    }
    // стартуем отрисовку в режиме визуализации точек
    // объединяемых в линии (GL_LINE_STRIP)
    Gl.glBegin(Gl.GL_LINE_STRIP);
    // рисуем начальную точку
    Gl.glVertex2d(GrapValuesArray[0, 0], GrapValuesArray[0, 1]);
    // проходим по массиву с координатами вычисленных точек
    for (int ax = 1; ax < elements_count; ax+=2)
    {
        // передаем в OpenGL информацию о вершине, участвующей в построении ли-
ний
        Gl.glVertex2d(GrapValuesArray[ax, 0], GrapValuesArray[ax, 1]);
    }
    // завершаем режим рисования
    Gl.glEnd();
    // устанавливаем размер точек, равный 5 пикселям
    Gl.glPointSize(5);
    // устанавливаем текущий цвет - красный
    Gl.glColor3f(255, 0, 0);
    // активируем режим вывода точек (GL_POINTS)
    Gl.glBegin(Gl.GL_POINTS);
    // выводим красную точку, используя ту ячейку массива, до которой мы дошли
(вычисляется в функии-обработчике событий таймера)
    Gl.glVertex2d(GrapValuesArray[pointPosition, 0], GrapValuesArray[pointPosition, 1]);
    // завершаем режим рисования
    Gl.glEnd();
    // устанавливаем размер точек, равный единице
    Gl.glPointSize(1);
}
```

Теперь нам осталось рассмотреть последнюю функцию – *Draw*.

В ней визуализируется координатная сетка под графиком, координатные оси и буквы для их обозначений, а также вызывается функция рисова-

ния графика и выводятся координаты мыши с линиями, соединяющими указатель мыши и оси координат.

Код этой функции выглядит следующим образом.

```
// функция, управляющая визуализацией сцены
private void Draw()
{
// очистка буфера цвета и буфера глубины
Gl.glClear(Gl.GL_COLOR_BUFFER_BIT | Gl.GL_DEPTH_BUFFER_BIT);
// очистка текущей матрицы
Gl.glLoadIdentity();
// установка черного цвета
Gl.glColor3f(0, 0, 0);
// помещаем состояние матрицы в стек матриц
Gl.glPushMatrix();
// выполняем перемещение в пространстве по осям X и Y
Gl.glTranslated(15, 15, 0);
// активируем режим рисования (указанные далее точки будут выводиться как точки
GL_POINTS)
Gl.glBegin(Gl.GL_POINTS);
// с помощью прохода двумя циклами создаем сетку из точек
for (int ax = -15; ax < 15; ax++)
{
for (int bx = -15; bx < 15; bx++)
{
// вывод точки
Gl.glVertex2d(ax, bx);
}
}
// завершение режима рисования примитивов
Gl.glEnd();

// активируем режим рисования, каждые две точки, последовательно вызванные ко-
мандой glVertex, объединяются в линию
Gl.glBegin(Gl.GL_LINES);
// рисуем координатные оси и стрелки на них
Gl.glVertex2d(0, -15);
Gl.glVertex2d(0, 15);
Gl.glVertex2d(-15, 0);
Gl.glVertex2d(15, 0);
// вертикальная стрелка
Gl.glVertex2d(0, 15);
Gl.glVertex2d(0.1, 14.5);
```

```
Gl.glVertex2d(0, 15);
Gl.glVertex2d(-0.1, 14.5);
// горизонтальная стрелка
Gl.glVertex2d(15, 0);
Gl.glVertex2d(14.5, 0.1);
Gl.glVertex2d(15, 0);
Gl.glVertex2d(14.5, -0.1);
// завершаем режим рисования
Gl.glEnd();

// выводим подписи осей "x" и "y"
PrintText2D(15.5f, 0, "x");
PrintText2D(0.5f, 14.5f, "y");
// вызываем функцию рисования графика
DrawDiagram();
// возвращаем матрицу из стека
Gl.glPopMatrix();
// выводим текст со значением координат возле курсора
PrintText2D(devX * Mcoord_X + 0.2f, (float)ScreenH - devY * Mcoord_Y + 0.4f, "[ x: "
+ (devX * Mcoord_X - 15).ToString() + " ; y: " + ((float)ScreenH - devY * Mcoord_Y -
15).ToString() + "]");
// устанавливаем красный цвет
Gl.glColor3f(255, 0, 0);
// включаем режим рисования линий, для того чтобы нарисовать
// линии от курсора мыши к координатным осям
Gl.glBegin(Gl.GL_LINES);
Gl.glVertex2d(lineX, 15);
Gl.glVertex2d(lineX, lineY);
Gl.glVertex2d(15, lineY);
Gl.glVertex2d(lineX, lineY);
Gl.glEnd();
// дожидаемся завершения визуализации кадра
Gl.glFlush();
// сигнал для обновления элемента, реализующего визуализацию
AnT.Invalidate();
}
```

В этой функции также нет ничего сложного, необходимо только разобраться с координатным методом построения примитивов. Если какая-либо часть кода остается непонятной, измените параметры в коде и посмотрите на результат этих изменений, чтобы сделать вывод о работе программы.

5.4. Вывод *2D* цветового треугольника

Рассмотрим программу отрисовки двухмерного треугольника, каждой вершине которого сопоставлен отдельный цвет, умножаемый на некоторую переменную. Цвет каждой точки внутри треугольника будет интерполироваться между цветами вершин. Таким образом мы получим плавный спектр цветов в треугольнике.

Мы внесем в код управляемые переменные, которые смогут влиять на установку цвета в определенной вершине и тем самым дадут возможность сформировать различные варианты цветового спектра.

Создадим новый проект и реализуем в нем систему визуализации, основанную на объекте *SimpleOpenGLControl* и подключении ссылок к библиотекам *Tao*, как это было рассмотрено в п. 4.3. Настройки инициализации *OpenGL* (код функции *Form1_Load* будут соответствовать коду, представленному в предыдущем пункте).

Приведем еще раз код функции-обработчика события *Form1_Load*:

```
private void Form1_Load(object sender, EventArgs e)
{
    // инициализация библиотеки GLUT
    Glut.glutInit();
    // инициализация режима окна
    Glut.glutInitDisplayMode(Glut.GLUT_RGB | Glut.GLUT_DOUBLE |
Glut.GLUT_DEPTH);
    // устанавливаем цвет очистки окна
    Gl.glClearColor(255, 255, 255, 1);
    // устанавливаем порт вывода, основываясь на размерах элемента управления AnT
    Gl.glViewport(0, 0, AnT.Width, AnT.Height);
    // устанавливаем проекционную матрицу
    Gl.glMatrixMode(Gl.GL_PROJECTION);
    // очищаем ее
    Gl.glLoadIdentity();
    // теперь необходимо корректно настроить 2D ортогональную проекцию
    // в зависимости от того, какая сторона больше,
    // мы немного варьируем конфигурацией настройки проекции
    if (AnT.Width <= AnT.Height)
    Glu.gluOrtho2D (0.0, 30.0, 0.0, 30.0 * (float)AnT.Height / (float)AnT.Width);
    else
    Glu.gluOrtho2D(0.0, 30.0 * (float)AnT.Width / (float)AnT.Height, 0.0, 30.0);
    // переходим к объектно-видовой матрице
    Gl.glMatrixMode(Gl.GL_MODELVIEW);
}
```

Помимо этого нам понадобится добавить на окно программы дополнительные элементы управления: 3 элемента *TrackBar* (вы можете найти

их в Панели элементов (*ToolBox*) в конце самого первого свитка («Все формы *Windows Forms*»)) и три элемента *Label*: они будут находиться над элементами TrackBar и содержать названия коэффициентов, которыми будут управлять данные *TrackBar*'ы.

Под каждым элементом *TrackBar* будет находиться по одному дополнительному элементу *label* для вывода текущего значения параметра, которым управляет данный ползунок.

(Сверху будут находиться элементы *label1*, *label2*, *label3*, а снизу *label4*, *label5*, *label6*).

Разместите все элементы, как показано на рис. 5.7. Для того чтобы элемент *TrackBar* стал вертикальным, необходимо в его свойствах указать параметр *Orientation* равным *Vertical*.

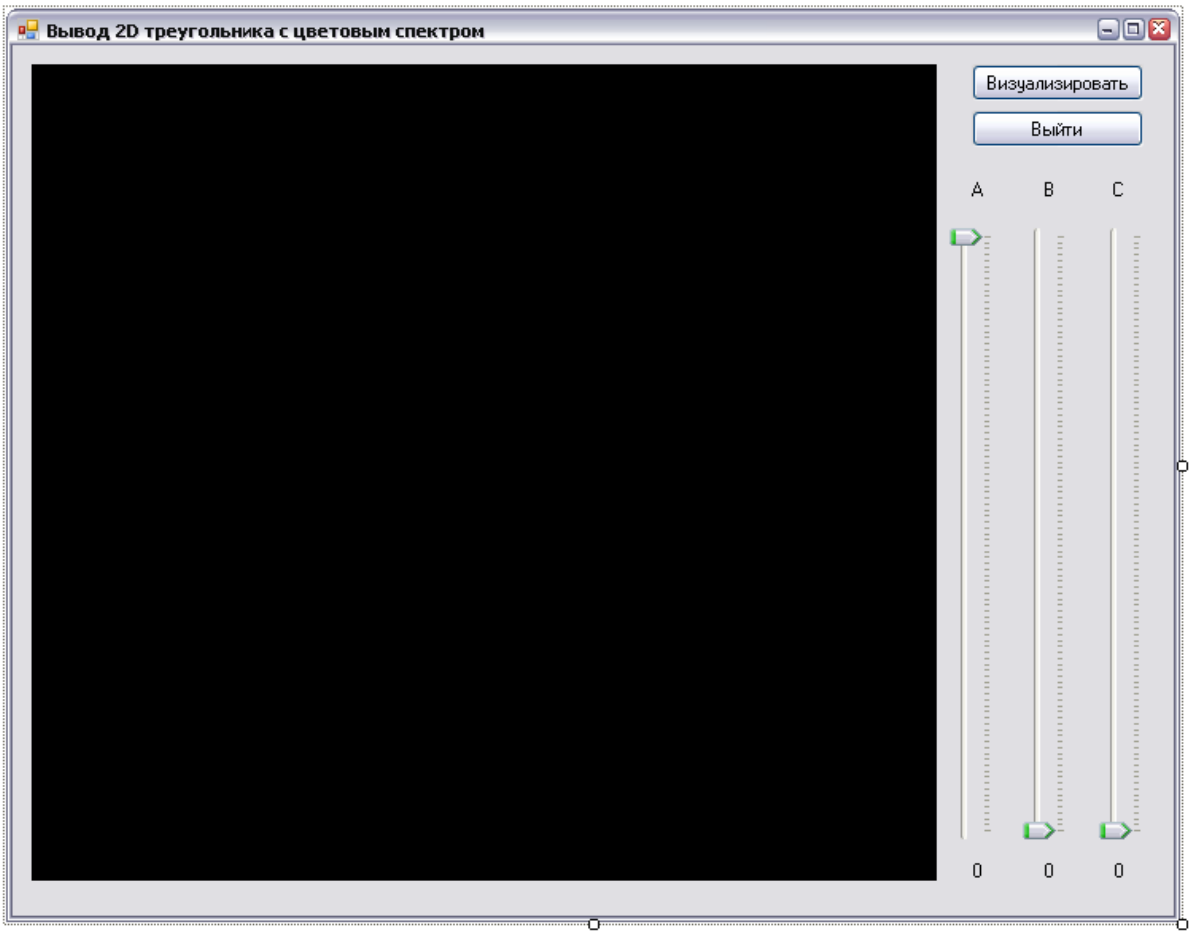


Рис. 5.7

В свойствах *TrackBar*'ов необходимо указать диапазон значений:
Minimum 0
Maximum 1000
TickFrequensy 10 (количество позиций между отметками).

Для первого *TrackBar*'а установите значение (*Value*), равное 1000, для остальных – нуль.

Выполните двойной щелчок левой клавишей мыши по каждому *TrackBa*r'у, чтобы создать функции-обработчики события перемещения ползунка пользователем.

В начале класса, отвечающего за нашу форму, добавьте следующую строку (для исключения путаницы в коде разместите эту строку перед функцией *Form1_Load*).

double a = 1, b = 0, c = 0;

Теперь реализуем функцию, отвечающую за рисование. Назовем ее *Draw*. Эта функция будет вызываться при нажатии на кнопку «Визуализировать» и при перемещении управляющих ползунков на *TrackBar*'ах, так как при этом будут изменяться коэффициенты *a, b, c* и наша сцена будет нуждаться в повторной визуализации.

Код функции с комментариями:

```
// функия Draw
private void Draw()
{
    // очищаем буфер цвета
    Gl.glClear(Gl.GL_COLOR_BUFFER_BIT);
    // активируем рисование в режиме GL_TRIANGLES, при котором три заданные
    // с помощью функции glVertex2d или glVertex3d вершины
    // будут объединяться в полигон (треугольник)
    Gl.glBegin(Gl.GL_TRIANGLES);
    // устанавливаем параметр цвета, основанный на параметрах a b c
    Gl.glColor3d(0.0 + a, 0.0 + b, 0.0 + c);
    // рисуем вершину в координатах 5,5
    Gl.glVertex2d(5.0, 5.0);
    // устанавливаем параметр цвета, основанный на параметрах c a b
    Gl.glColor3d(0.0 + c, 0.0 + a, 0.0 + b);
    // рисуем вершину в координатах 25,5
    Gl.glVertex2d(25.0, 5.0);
    // устанавливаем параметр цвета, основанный на параметрах b c a
    Gl.glColor3d(0.0 + b, 0.0 + c, 0.0 + a);
    // рисуем вершину в координатах 25,5
    Gl.glVertex2d(5.0, 25.0);
    // завершаем режим рисования примитивов
    Gl.glEnd();
    // дожидаемся завершения визуализации кадра
    Gl.glFlush();
    // обновляем изображение в элементе AnT
    AnT.Invalidate();
}
```

Функции-обработчики перемещения ползунка в *TrackBar*'ах будут выглядеть следующим образом:

```
private void trackBar1_Scroll(object sender, EventArgs e)
{
    // генерация коэффициента
    a = (double)trackBar1.Value / 1000.0;
    // вывод значения коэффициента, управляемого данным ползунком.
    // (под TrackBar'ом)
    label4.Text = a.ToString();
}
private void trackBar2_Scroll(object sender, EventArgs e)
{
    // генерация коэффициента
    b = (double)trackBar2.Value / 1000.0;
    // вывод значения коэффициента, управляемого данным ползунком.
    // (под TrackBar'ом)
    label5.Text = b.ToString();
}
private void trackBar3_Scroll(object sender, EventArgs e)
{
    // генерация коэффициента
    c = (double)trackBar3.Value / 1000.0;
    // вывод значения коэффициента, управляемого данным ползунком.
    // (под TrackBar'ом)
    label6.Text = c.ToString();
}
```

Для корректной визуализации (например, в том случае, когда наше окно перекроет какое-либо другое) мы добавим в нашу форму объект − таймер. Он будет отсчитывать промежутки времени таким образом, чтобы примерно 40 раз в секунду перерисовывать содержимое нашего окна.

Для того чтобы добавить объект, перейдите к окну *ToolBox*, выберите в свитке «Компоненты» элемент *Timer* (рис. 5.8) и перетащите его на форму.

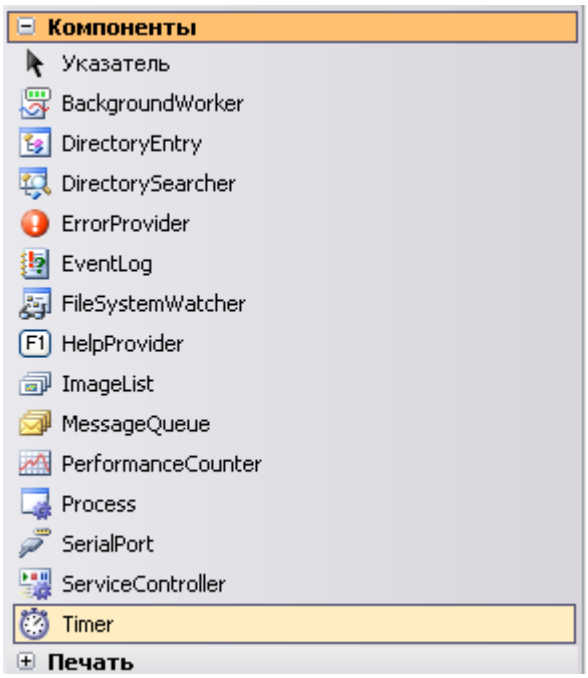


Рис. 5.8

Место, куда вы перетащите этот элемент, не важно: он не занимает его на форме и отобразится в полоске таких же («не занимающих место») объектов под формой.

Щелкните по нему и перейдите в его свойства.

Смените параметр *name* на значение *RenderTimer*, а параметр *Interval* − на значение 25 (раз в 25 мс будет вызываться событие *OnTimer*).

Теперь щелкните по нему двойным щелчком левой клавиши мыши. Создастся функция-обработчик события *OnTimer*. Отсюда мы будем вызывать функцию *Draw*:

```
private void RenderTimer_Tick(object sender, EventArgs e)
{
    // функция визуализации
    Draw();
}
```

Кнопка «Визуализировать» будет отвечать за активацию этого таймера. Для этого будет использоваться функция *Start*:

```
// обработчик кнопки "Визуализировать"
private void button1_Click(object sender, EventArgs e)
{
    // старт таймера, отвечающего за вызов функции,
    // визуализирующей кадр
    RenderTimer.Start();
}
```

И в заключение назначьте обработчик кнопке «Выйти»

```
// обработчик кнопки "Выйти"
private void button2_Click(object sender, EventArgs e)
{
    // выход из приложения
    Application.Exit();
}
```

После компиляции и запуска программы мы сможем управлять разложением спектра в треугольнике (рис. 5.9, 5.10).

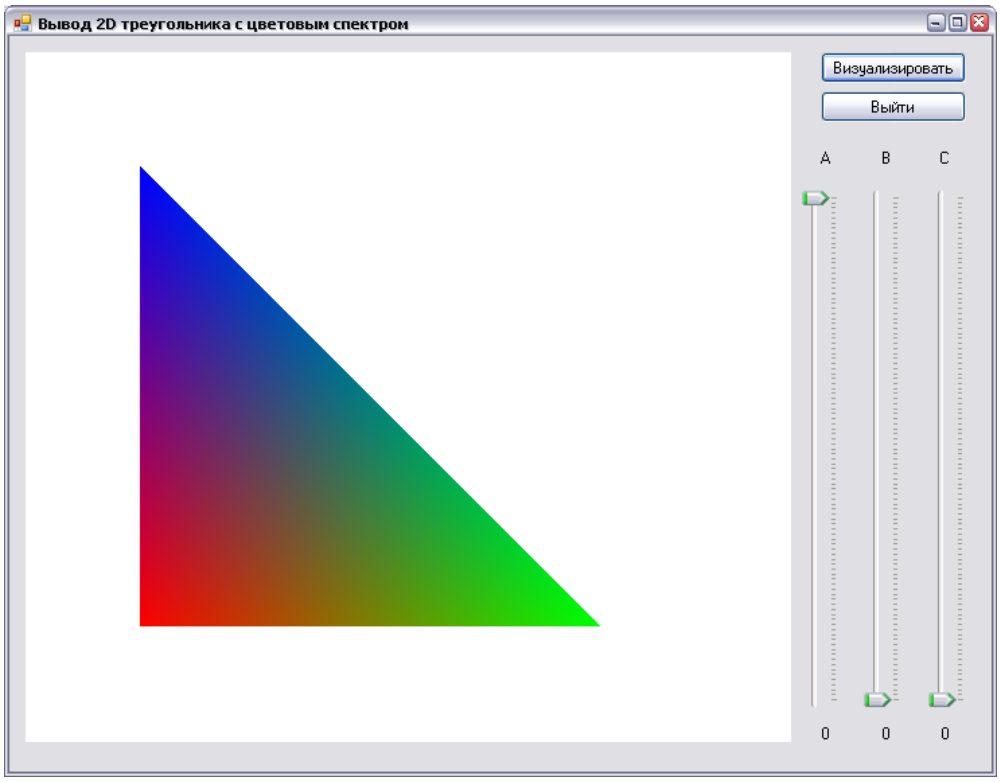


Рис. 5.9

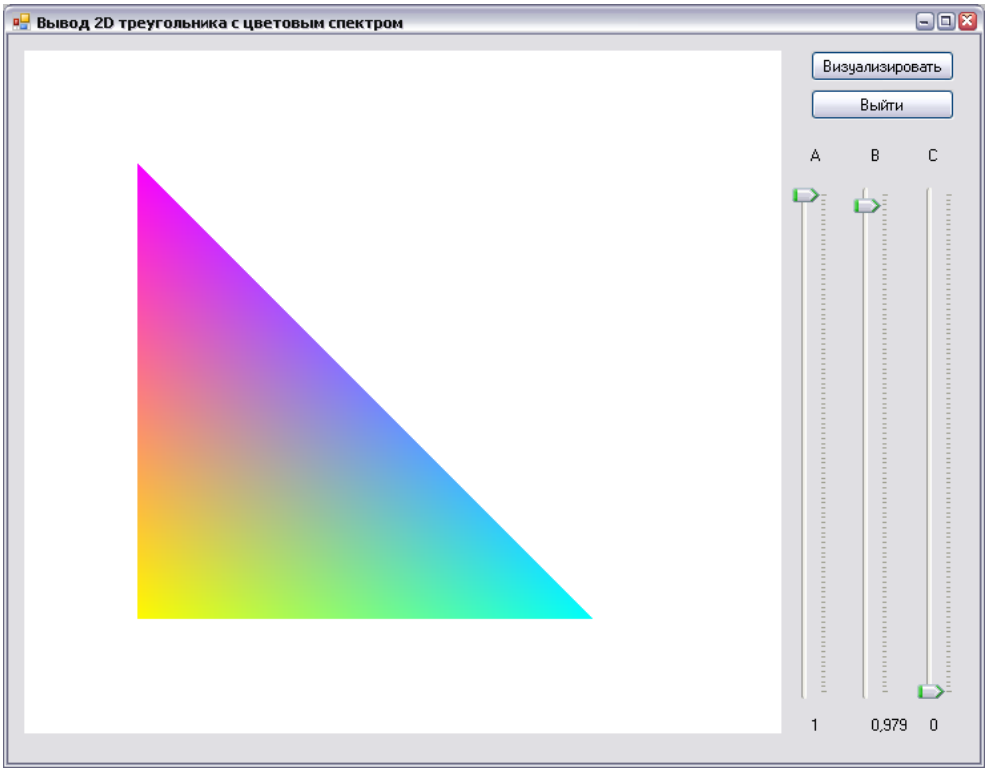


Рис. 5.10

Контрольные вопросы

1. Порядок инициализации *OpenGL* в *C#*.
2. Способы визуализации *2D*-примитивов в *VisualC#*.

ПРИЛОЖЕНИЕ

Установка и подключение библиотек TAO

Рассмотрим установку *Tao Framework* на примере версии 2.1.0.

Установка может не начаться по причине отсутствия . *NET Framework 2.0*. Однако если уже установлена *MS Visual Studio 2008*, то такой проблемы возникнуть не должно, так как она уже содержит *.NET Framework 3.0*, который устанавливается в ходе общей установки *MS Visual Studio*.

После запуска дистрибутива появится окно подтверждения установки (рис. П1). Подтвердите установку и следуйте вперед, нажимая кнопку "Далее", чтобы подтвердить все стандартно выставленные программой опции (рис. П2 – П6). На шестом шаге установки (рис. П7) обязательно оставьте указанные галочки.

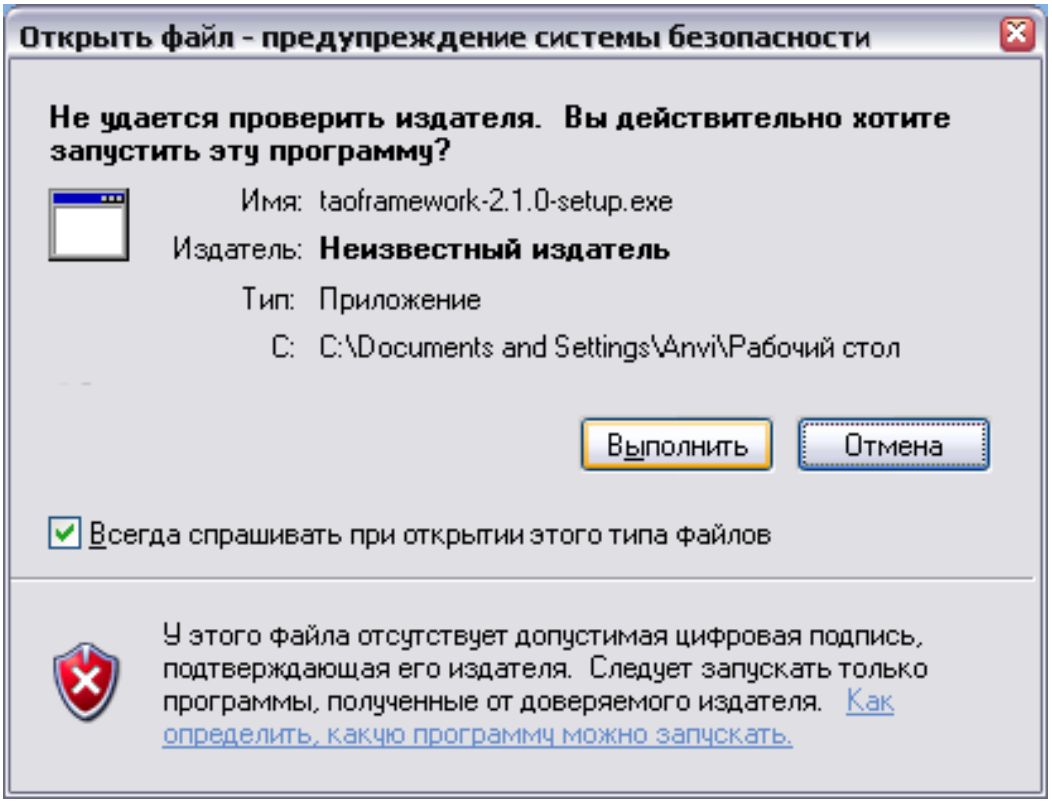


Рис. П1

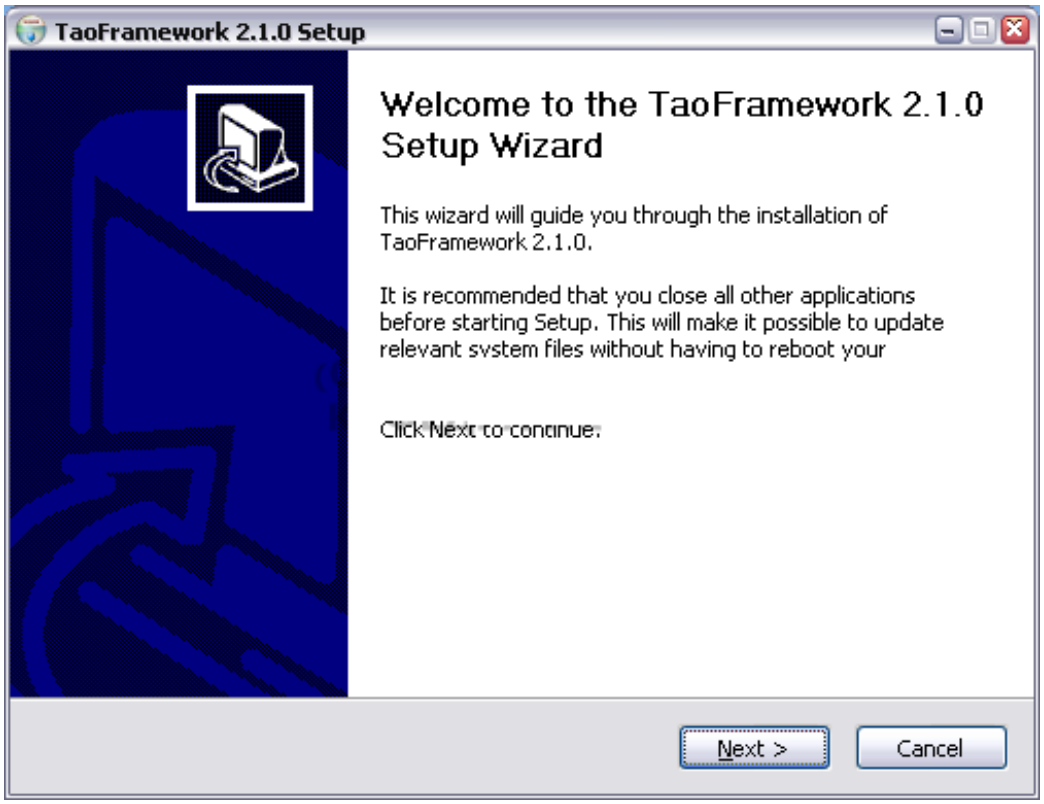

TaoFramework 2.1.0 Setup
Welcome to the TaoFramework 2.1.0 Setup Wizard
This wizard will guide you through the installation of TaoFramework 2.1.0.
It is recommended that you close all other applications before starting Setup. This will make it possible to update relevant system files without having to reboot your
Click Next to continue.
Next >
Cancel

Рис. П2

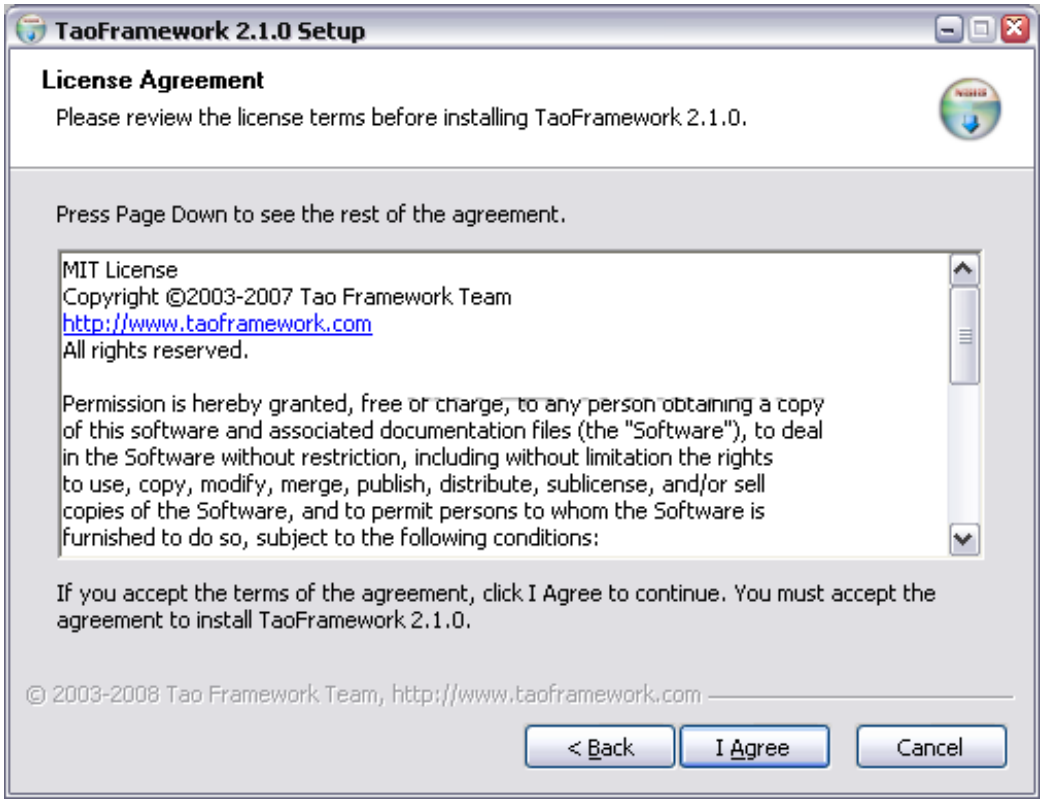

TaoFramework 2.1.0 Setup
License Agreement
Please review the license terms before installing TaoFramework 2.1.0.
Press Page Down to see the rest of the agreement.
MIT License
Copyright ©2003-2007 Tao Framework Team
http://www.taoframework.com
All rights reserved.
Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:
If you accept the terms of the agreement, click I Agree to continue. You must accept the agreement to install TaoFramework 2.1.0.
© 2003-2008 Tao Framework Team, http://www.taoframework.com
< Back
I Agree
Cancel

Рис. П3

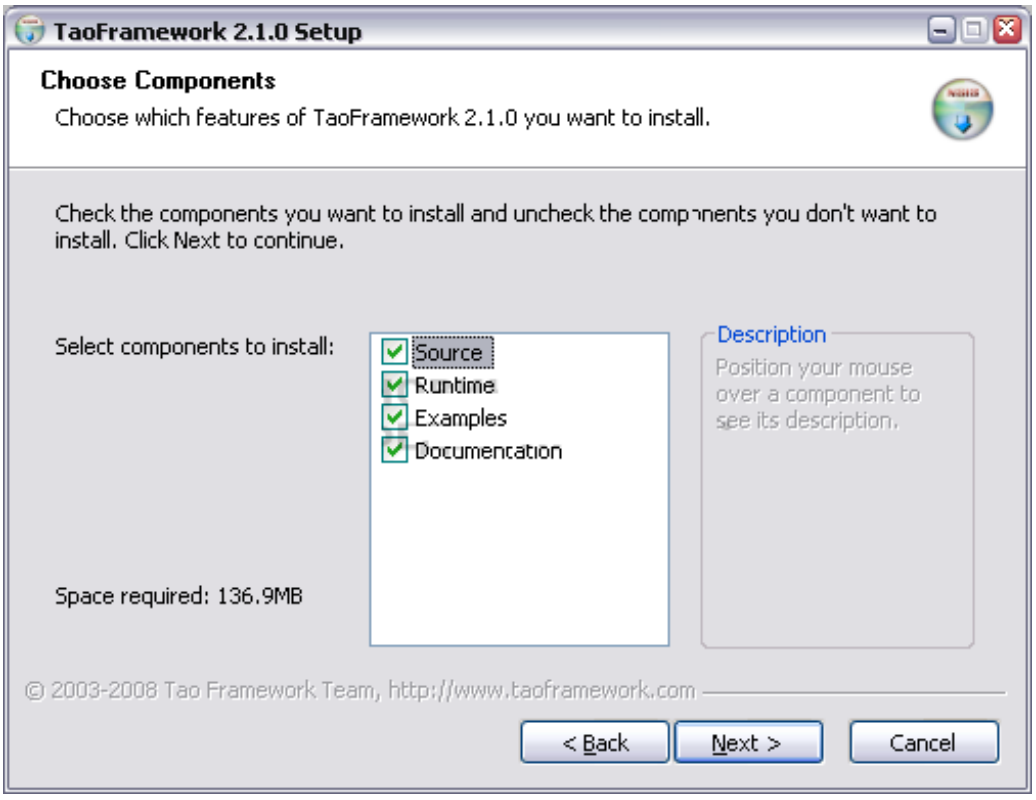

TaoFramework 2.1.0 Setup
Choose Components
Choose which features of TaoFramework 2.1.0 you want to install.
Check the components you want to install and uncheck the components you don't want to install. Click Next to continue.
Select components to install:
Source
Runtime
Examples
Documentation
Description
Position your mouse over a component to see its description.
Space required: 136.9MB
© 2003-2008 Tao Framework Team, http://www.taoframework.com
< Back
Next >
Cancel

Рис. П4

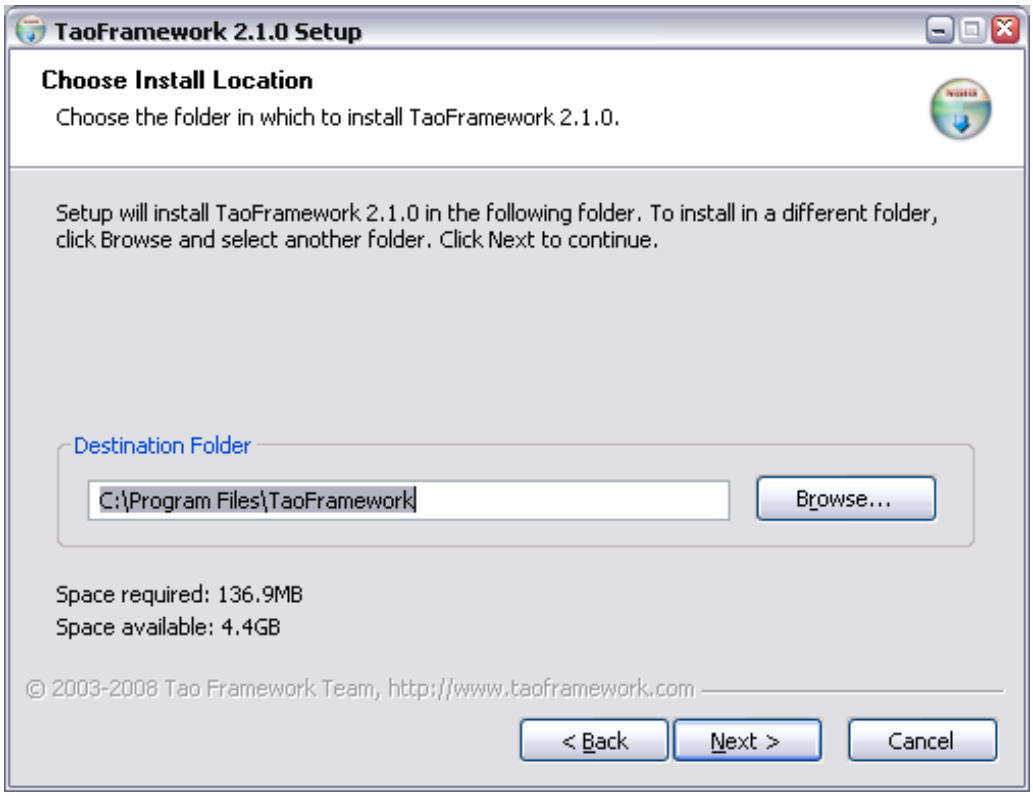

TaoFramework 2.1.0 Setup
Choose Install Location
Choose the folder in which to install TaoFramework 2.1.0.
Setup will install TaoFramework 2.1.0 in the following folder. To install in a different folder, click Browse and select another folder. Click Next to continue.
Destination Folder
C:\Program Files\TaoFramework
Browse...
Space required: 136.9MB
Space available: 4.4GB
© 2003-2008 Tao Framework Team, http://www.taoframework.com
< Back
Next >
Cancel

Рис. П5

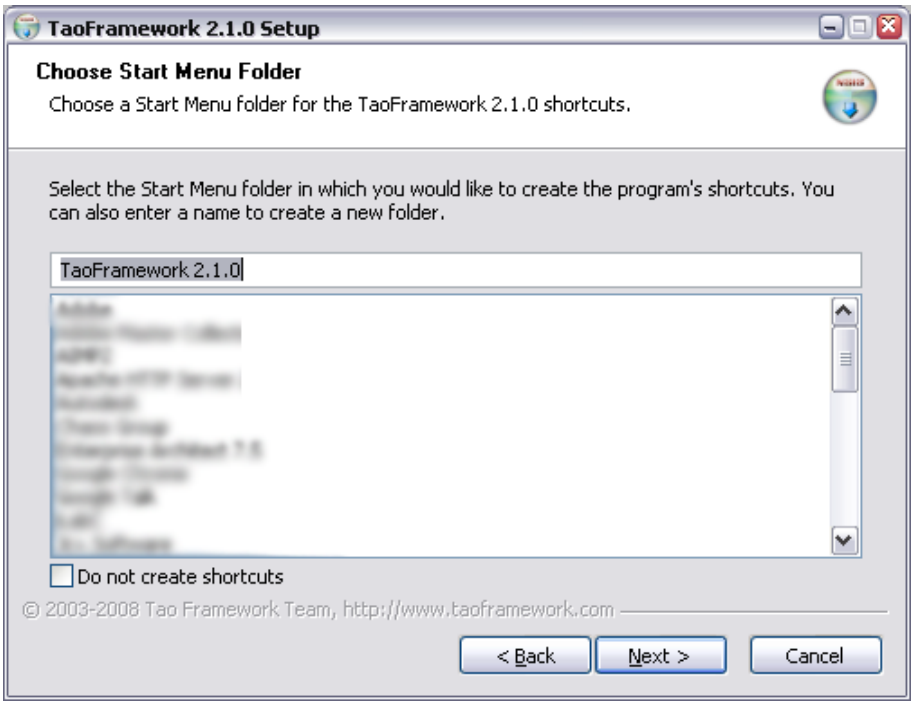


Рис. П6

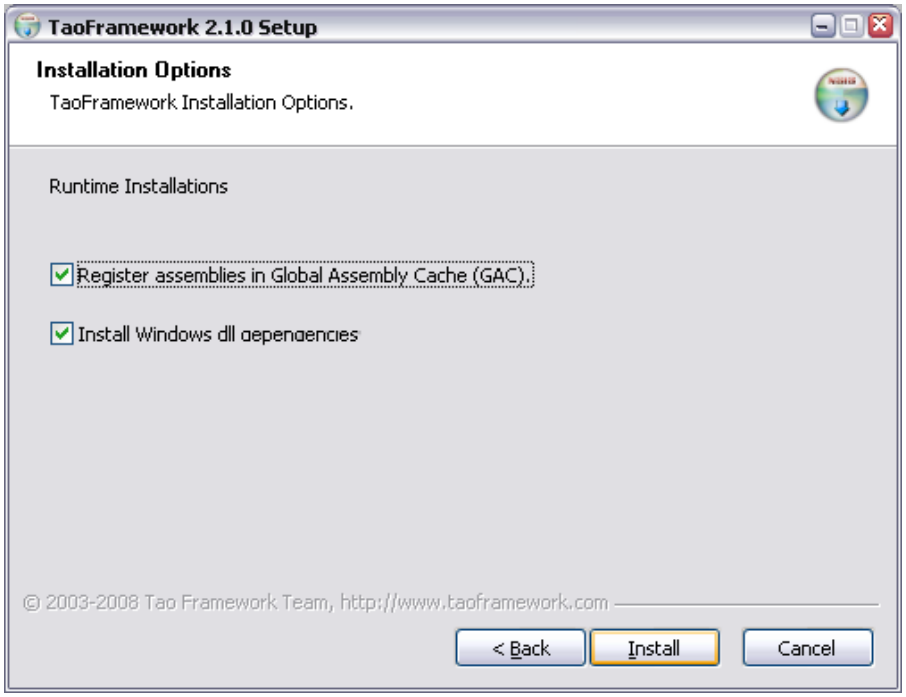


Рис. П7

После этого шага пройдет инсталляция файлов и *Tao Framework* готова к работе.

Для использования библиотеки может понадобиться в настройках операционной системы в переменную среды *PATH* добавить каталог *C:\Program Files\TaoFramework\lib*.

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. *Жигалов, И. Е*. Компьютерная графика : курс лекций / И. Е. Жигалов. – Владимир : Ред.-издат. комплекс ВлГУ, 2004. − 124 с.

2. *Жигалов, И. Е*. Программирование компьютерной графики : практикум / И. Е. Жигалов ; Владим. гос. ун-т. − Владимир, 2002. − 100 с.

3. *Никулин, Е. А*. Компьютерная геометрия и алгоритмы машинной графики / Е. А. Никулин. − СПб. : БХВ, 2003. – 560 с. − ISBN 5-94157-264-6.

4. *Поляков, А. Ю.* Методы и алгоритмы компьютерной графики в примерах на Visual C++ / А. Ю. Поляков. – СПб. : БХВ-Петербург, 2002. – 416 с. − ISBN 5-941571-36-4.

5. *Порев, В. Н*. Компьютерная графика / В. Н. Порев. – СПб. : БХВ-Петербург, 2002. − 432 с. − ISBN 5-94157-139-9.

6. *Тарасов, И. А*. Основы программирования в OpenGL / И. А. Тарасов. − М. : Телеком, 2000. − 188 с. − ISBN 5-935170-16-7.

7. *Эйнджел, Э*. Интерактивная компьютерная графика. Вводный курс на базе OpenGL / Э. Эйнджел. – М. : Вильямс, 2001. – 592 с. − ISBN 5-8459-0209-6.

8. *Павловская, Т. А.* C#. Программирование на языке высокого уровня : учеб. для вузов / Т. А. Павловская. − СПб. : Питер, 2009. − 432 с. − ISBN 978-5-91180-174-8.

Оглавление

Учебное издание

ЖИГАЛОВ Илья Евгеньевич
НОВИКОВ Иван Андреевич

ПРОГРАММИРОВАНИЕ
КОМПЬЮТЕРНОЙ ГРАФИКИ

Учебное пособие

Подписано в печать 14.03.14.
Формат 60x84/16. Усл. печ. л. 5,58. Тираж 75 экз.
Заказ
Издательство
Владимирского государственного университета
имени Александра Григорьевича и Николая Григорьевича Столетовых.
600000, Владимир, ул. Горького, 87.