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 ДВУХМЕРНОЙ КОМПЬЮТЕРНОЙ ГРАФИКИ

Учебное пособие

Владимир 2015

УДК 004.92
ББК 32.973.26-018
Ж68

Рецензенты:

Доктор технических наук, профессор
зав. кафедрой информатики и защиты информации
Владимирского государственного университета
имени Александра Григорьевича и Николая Григорьевича Столетовых
М. Ю. Монахов

Доктор технических наук, профессор
профессор кафедры инновационного предпринимательства
Московского государственного технического
университета им. Н. Э. Баумана
Д. В. Александров

Печатается по решению редакционно-издательского совета ВлГУ

Ж68 **Жигалов, И. Е.** Программирование двухмерной 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, 2015. – 120 с.
ISBN 978-5-9984-0610-2

Содержит теоретический материал и практические задания по темам, связанным с изучением алгоритмов двухмерной компьютерной графики с использованием среды Visual C# и графической библиотеки OpenGL.

Предназначено для студентов направлений 230400 – Информационные системы и технологии и 231000 – Программная инженерия по дисциплине «Программирование компьютерной графики».

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

Ил. 38. Библиогр.: 10 назв.

УДК 004.92
ББК 32.973.26-018

ISBN 978-5-9984-0610-2

© ВлГУ, 2015

ВВЕДЕНИЕ

Данное издание является логическим продолжением учебного пособия «Программирование компьютерной графики», в котором рассматривались вопросы изучения среды *C#* в *MS VisualStudio* и библиотеки *OpenGL.*

В пособии анализируются важные для изучения компьютерной графики темы по программированию алгоритмов двухмерной графики, обработке растровых изображений, работе с фракталами и сплайнами, применению к графическим объектам геометрических преобразований, приводится большое количество подробно комментированных текстов программ, поясняющих порядок и особенности использования инструментов и команд *C#* при разработке графических приложений. Представленные коды иллюстрируют применение того или иного алгоритма двухмерной графики и поясняют методику использования инструментальных средств для получения желаемого визуального эффекта. Всё это помогает освоить средства разработки графического прикладного программного обеспечения.

Тема 1. ВВЕДЕНИЕ В ПРОГРАММИРОВАНИЕ *2D* ГРАФИКИ

Цель изучения темы. На примере разработки простого растрового редактора рассмотрение принципов программирования *2D* графики в *C#*.

1.1. Разработка растрового редактора

Цель – создание растрового редактора в *OpenGL* с использованием *C#.NET* – небольшого приложения, по своей функциональности напоминающего *Windows Paint*.

Введение

Так как нам придется реализовать довольно большой объем функциональности, разобьем создание приложения на части (функциональность программы будет расширяться от раздела к разделу):

1. Описание объектной модели и принципов работы программы. Визуализацию конечной картинки выполним в *OpenGL*, хотя используются и стандартные классы *C#.NET*.

Далее создадим основу оконного приложения, необходимые меню, панели инструментов и другие элементы.

После создания оболочки программы, разработаем начальный уровень ее работы – рисование одной тестовой кистью с одним слоем без изменения цвета рисования.

2. Добавление в оболочку инструментов рисования и функции выбора цвета.

3. Создание системы слоев и реализация более быстрых методов визуализации в *OpenGL*.

4. Завершение оболочки программы, включая меню и взаимодействие элементов.

5. Оптимизация функции визуализации – добавление дисплейных списков, отрисовка массивов вершин.

Создаваемое приложение – не профессиональный редактор, а демонстрация основ работы с растровой графикой и методов ее реализации с использованием *OpenGL*.

Модель работы программы

Растровое изображение – это структура данных, описывающая сетку пикселей (точек). Как правило, эта структура имеет прямоугольную форму. Создание растрового редактора сводится к разработке программы, которая будет снабжена набором инструментов для редактирования растровой структуры данных. Все эти элементы основаны на математических алгоритмах компьютерной графики.

Работа с программой основывается на визуализированных с помощью оболочки данных: выбор инструментов, рисование в рабочей области, управление слоями. Оболочка программы обрабатывает карты сообщений от операционной системы и вызывает методы класса *anEngine*. Этот управляющий класс отвечает за основные расчеты и визуализацию – он управляет созданием слоев, через него идет обновление информации в слоях, обсчитывается результирующее изображение и т.д. Этот класс будет "движком" нашего графического редактора. Он же управляет кистями. Для хранения кистей используем отдельный класс.

Сейчас мы реализуем не все функции, а лишь самые главные, которые должны обеспечить работоспособность программы – основа оболочки, класс "движка", класс слоев, класс кистей. Сначала работают только один слой, одна кисть и алгоритмы инициализации *OpenGL* (в оболочке), а также визуализации.

Схема работы программы представлена на рис. 1.1.

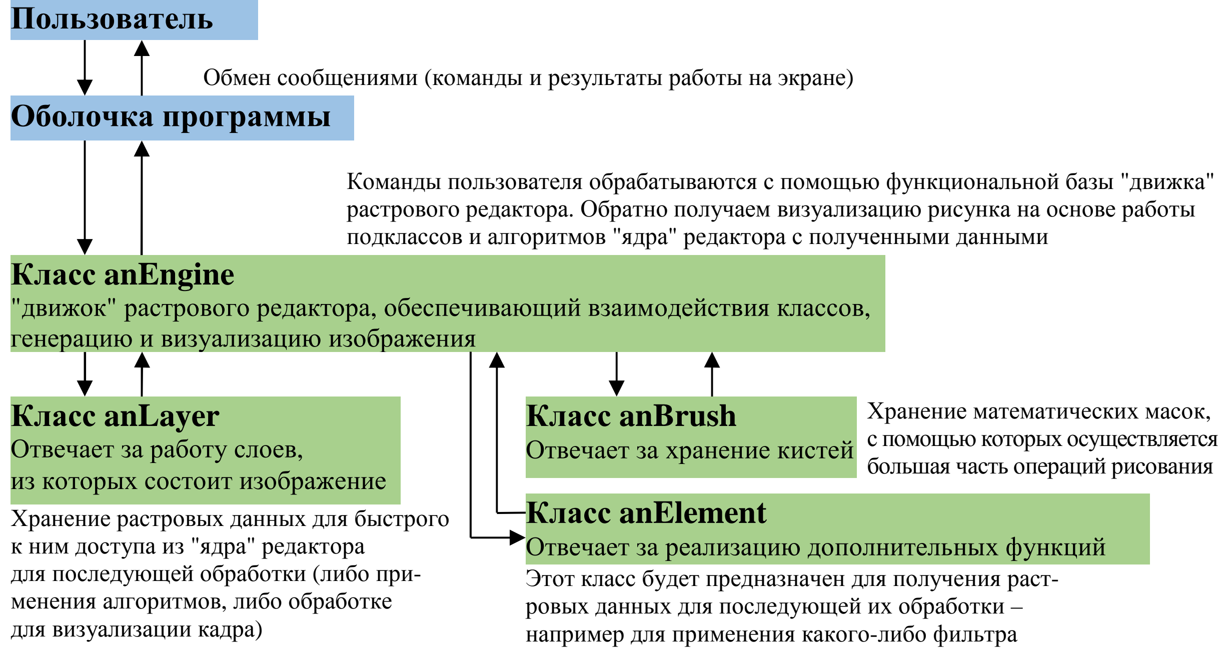


Рис. 1.1

Создание оболочки программы

Создайте новый проект. В качестве шаблона установите приложение *Windows Form*.

Теперь создайте окно на основе примера, представленного на рис. 1.2.

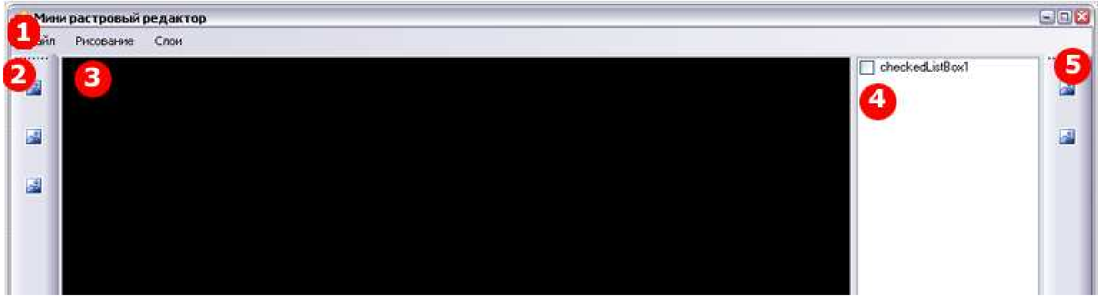


Рис. 1.2

На окне будут расположены следующие элементы (см. рис. 1.2):

1. Меню программы. Создадим три раздела меню: Файл, Рисование и Слои (их подменю можно увидеть на рис. 1.3).

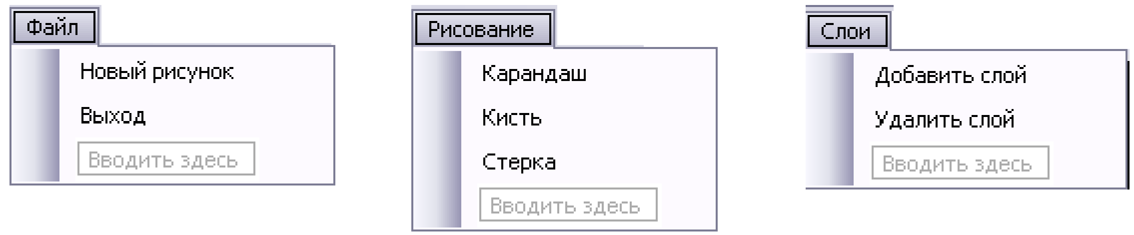


Рис. 1.3

2. Панель инструментов. Ширина панели – 44 пикселя. Будут созданы заготовки трех кнопок. Эта панель далее заполняется кнопками для установки режимов рисования – кистей, геометрических объектов и т.д.

3. Элемент *SimpleOpenGLControl*. Расположите его так, как показано на рисунке, затем свойство *name* установите равным *AnT*. Здесь проходит основной *render*.

4. Здесь находится элемент *CheckedListBox*. Далее реализуем систему слоев, которые будут активно участвовать в визуализации. В этом элементе отобразится список наших слоев. Оболочка также получит функции для редактирования параметров слоев.

5. Здесь находится панель инструментов. Сначала мы расположим на ней две кнопки. В дальнейшем кнопки этой панели будут отвечать за операции над слоями.

При разработке приложения используем код из пп. 2.2 (процесс создания меню и панелей инструментов) и 4.3 (процесс установки элементов для визуализации *OpenGL* и их первоначальной инициализации).

После размещения элементов подключим необходимые пространства имен:

```
…
using Tao.OpenGl;
using Tao.FreeGlut;
using Tao.Platform.Windows;
```

Конструктор формы выглядит следующим образом:

```
public Form1()
{
   InitializeComponent();
   // инициализация элемента SimpleOpenGLControl (AnT)
   AnT.InitializeContexts();
}
```

Далее добавим код объявления экземпляра класса движка растрового редактора:

```
private anEngine ProgrammDrawingEngine;
```

Когда вы добавите этот код, он выделится как содержащий ошибку, так как мы еще не добавили код самого класса *anEngine*. Но сначала надо завершить код оболочки, а затем переходить к функциям дополнительных классов.

Добавьте один таймер и назовите его *RenderTimer*. Затем добавьте обработку события загрузки формы.

Функция, вызываемая событием загрузки формы (чтобы ее добавить, дважды щелкните левой кнопкой мыши по заголовку окна или перейдите через свойства формы приложения к списку доступных событий *event* и добавьте двойным щелчком в пустой области обработку события *Load*), будет содержать код инициализации *OpenGL*:

```
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();
Glu.gluOrtho2D(0.0, AnT.Width, 0.0, AnT.Height);
// переходим к объектно-видовой матрице
Gl. glMatrixMode(Gl. GL_MODELVIEW);
ProgrammDrawingEngine = new anEngine(AnT.Width, AnT.Height, AnT.Width, AnT.Height);
RenderTimer.Start();
}
```

Как видно из кода, мы создаем *2D* ортогональную проекцию, причем в отличие от кода, который был использован ранее, здесь проекция устанавливается таким образом, что размер видимой области в проекции равен размерам элемента *AnT*. То есть координата *X* на элементе *AnT* равна координате *X* видимой части в координатной системе *OpenGL*. Но координатная ось *Y* направлена в противоположную сторону, поэтому координата *Y* в координатной системе *OpenGL* равна *AnT.Height – Y'* (*Y'* – координата *Y* на элементе *AnT*).

После настройки проекции мы инициализируем экземпляр движка растрового редактора и активируем таймер.

В настройках таймера добавьте обработчик события *Tick*. (Либо выделив элемент *таймер*, далее перейдя к его свойствам – меню *Event*, затем двойной щелчок на пустой строке справа от события *event*, либо просто двойной щелчок на элементе *RenderTimer*).

В свойствах таймера установите интервал равным 30 миллисекундам.

Код функции обработчика события таймера и функции визуализации будут выглядеть следующим образом:

```
// событие Tick таймера
private void RenderTimer_Tick(object sender, EventArgs e)
{ // вызываем функцию рисования
```

```
   Drawing();
   }
// функция рисования
private void Drawing()
{ // очистка буфера цвета и буфера глубины
   Gl.glClear(Gl.GL_COLOR_BUFFER_BIT | Gl.GL_DEPTH_BUFFER_BIT);
   // очищение текущей матрицы
   Gl.glLoadIdentity();
   // установка черного цвета
   Gl.glColor3f(0, 0, 0);
   // визуализация изображения из движка
   ProgrammDrawingEngine.SwapImage();
   // дожидаемся завершения визуализации кадра
   Gl.glFlush();
   // сигнал для обновления элемента, реализующего визуализацию
   AnT.Invalidate();
}
```

Когда мы создавали программу, визуализирующую график функции, мы уже назначали объекту *AnT* обработку события движения мыши.

Аналогичные действия необходимо совершить и здесь. Код функции-обработчика будет выглядеть следующим образом:

```
// функция обработчик события движения мыши (событие MouseMove для элемента AnT)
private void AnT_MouseMove(object sender, MouseEventArgs e)
{
   // если нажата левая клавиша мыши
   if(e.Button == MouseButtons.Left)
   ProgrammDrawingEngine.Drawing(e.X, AnT.Height - e.Y);
}
```

Теперь необходимо добавить класс *anEngine* и реализовать в нем методы и конструктор, иначе созданный код останется непригодным для компилирования.

Инициализация *OpenGL* в функции *Form1_Load*, создание счетчиков и назначение обработчиков событий было рассмотрено ранее.

Функция обработки движения мыши проверяет, нажата ли левая клавиша мыши. Если да, то вызывается метод рисования кистью (*Drawing*), который мы далее реализуем в классе *anEngine*. В этот метод в качестве параметров передаются значения координат на элементе *AnT – X* и *AnT.Height – Y*.

Функция *Drawing* выполняет стандартные операции очистки буфера глубины и цвета, а также очистки объектно-видовой матрицы; затем вызывается функция, отвечающая за визуализацию рисунка.

Создание дополнительных классов и их базовых методов

Добавьте в программу новый класс (см. п. 1.3). Назовите класс *anEngine*.

Для него создается заготовка класса и отдельный файл с исходным кодом, называемый *anEngine.cs*. Здесь же разместим еще два дополнительных класса. Сгенерированный код класса выглядит следующим образом:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace My_Paint
{
   class anEngine
   {
    }
}
```

Приступим к его обновлению: сначала добавим все пространства имен, как в файле *Form1.cs*, чтобы в будущем не столкнуться с тем, что какое-либо пространство имен не подключено.

Подключаемые пространства имен:

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Collections;
using System.Windows.Forms;
using using System.IO;
using Tao.OpenGl;
using Tao.FreeGlut;
using Tao.Platform.Windows;
```

Теперь добавим в исходный код еще два класса: *anLayer* и *anBrush*.

Начнем с *anBrush*. Разместите код этого класса перед кодом заготовки класса *anEngine*.

Пока его функциональность сводится к хранению маски кисти. Причем на данном этапе (создания заготовки всех классов и настройки их работоспособности) мы занесем в конструктор стандартную кисть в виде крестика.

Код класса *anBrush*:

```
public class anBrush
{
// объект Bitmap, в котором мы будем хранить битовую карту нашей кисти.
// этот объект выбран из-за удобства, а также из-за того, что в будущем мы добавим возможность загрузки кистей из графических файлов с расширением bmp.
public Bitmap myBrush;
// конструктор класса
public anBrush()
{
// создаем плоскость 5x5 пикселей
myBrush = new Bitmap(5, 5);
// заполняем все пиксели красным цветом (все пиксели красного цвета мы будем
// считать незначимыми черного – значимыми при рисовании кистью)
// для установки пикселя используется функция SetPixel.
for (int ax = 0; ax < 5; ax++)
for (int bx = 0; bx < 5; bx++)
myBrush.SetPixel(ax, bx, Color.Red);
// далее в данном массиве мы рисуем крестик
myBrush.SetPixel(0, 2, Color.Black);
myBrush.SetPixel(1, 2, Color.Black);
myBrush.SetPixel(2, 0, Color.Black);
myBrush.SetPixel(2, 1, Color.Black);
myBrush.SetPixel(2, 2, Color.Black);
myBrush.SetPixel(2, 3, Color.Black);
myBrush.SetPixel(2, 4, Color.Black);
myBrush.SetPixel(3, 2, Color.Black);
myBrush.SetPixel(4, 2, Color.Black);
}
}
```

Как видите, код данного класса пока очень прост. Но он нам необходим, чтобы сразу заложить алгоритм рисования кистью в классе *anLayer*.

Кисть, которую мы задаем по умолчанию кодом:

```
myBrush.SetPixel(0, 2, Color.Black);
myBrush.SetPixel(1, 2, Color.Black);
myBrush.SetPixel(2, 0, Color.Black);
myBrush.SetPixel(2, 1, Color.Black);
myBrush.SetPixel(2, 2, Color.Black);
myBrush.SetPixel(2, 3, Color.Black);
myBrush.SetPixel(2, 4, Color.Black);
myBrush.SetPixel(3, 2, Color.Black);
myBrush.SetPixel(4, 2, Color.Black);
```

Легче понять смысл этого кода, изучив полученный массив чисел (рис. 1.4).

Рис. 1.4

Теперь рассмотрим класс *anLayer*. Его задача – хранение данных закрашенных пикселей одного слоя изображения. Помимо хранения графических данных слой содержит алгоритм рисования на основе выбранной кисти: когда пользователь пытается что-либо нарисовать в

окне, зажав левую клавишу мыши и перемещая курсор по элементу *AnT*, оболочка получает событие мыши, которое обрабатывается (корректируется координата *Y*) и передается в экземпляр класса *anEngine* с помощью функции *Drawing*. Этот код мы уже рассмотрели.

Когда будет вызываться функция *Drawing*, класс *anEngine* определит выбранную на данный момент кисть (пока что она у нас только одна), активный слой (слой пока работает также только один, но далее мы доработаем систему слоев, кистей и функцию оболочки).

Далее у необходимого слоя вызывается функция, отвечающая за рисование, где определяются различные факторы (например, ограничивающие рисунок по краям изображения, для того чтобы в процессе рисования не выйти за границы рисования), и затем происходит изменение цветов пикселей маской кисти.

Код этого класса на данный момент выглядит следующим образом:

```
public class anLayer
{
// размеры экранной области
public int Width, Heigth;
// массив , представляющий область рисунка (координаты пикселя и его цвет)
private int[,,] DrawPlace;
// флаг видимости слоя: true - видимый, false - невидимый
private bool isVisible;
// текущий установленный цвет
private Color ActiveColor;
// конструктор класса, в качестве входных параметров
// мы получаем размеры изображения, чтобы создать в памяти массив,
// который будет хранить растровые данные для этого слоя
public anLayer(int s_W, int s_H)
{
// запоминаем значения размеров рисунка
Width = s_W;
Heigth = s_H;
// создаем в памяти массив, соответствующий размерам рисунка
// каждая точка на плоскости массива будет иметь 3 составляющие цвета
// + 4 ячейка - индикатор того, что данный пиксель пуст (или полностью прозра-
чен)
DrawPlace = new int[Width, Heigth, 4];
// проходим по всей плоскости и устанавливаем всем точкам
// индикатор прозрачности
```

```
for (int ax = 0; ax < Width; ax++)
{
for (int bx = 0; bx < Heigth; bx++)
{
// флаг прозрачности точки в координатах ax,bx.
DrawPlace[ax, bx, 3] = 1;
}
}
// устанавливаем флаг видимости слоя (по умолчанию создаваемый слой всегда
видимый)
isVisible = true;
// текущим активным цветом устанавливаем черный
// в следующей главе мы реализуем функции установки цветов из оболочки
ActiveColor = Color.Black;
}
// функция установки режима видимости слоя
public void SetVisibility(bool visiblityState)
{
isVisible = visiblityState;
}
// функция получения текущего состояния видимости слоя
public bool GetVisibility()
{
return isVisible;
}
// функция рисования
// получает в качестве параметров кисть для рисования и координаты,
// где сейчас необходимо перерисовать пиксели заданной кистью
public void Draw(anBrush BR, int x, int y)
{
// определяем начальную позицию рисования
int real_pos_draw_start_x = x - BR.myBrush.Width / 2;
int real_pos_draw_start_y = y - BR.myBrush.Height / 2;
// корректируем ее для невыхода за границы массива
// проверка на отрицательные значения (граница "слева")
if (real_pos_draw_start_x < 0)
real_pos_draw_start_x = 0;
if (real_pos_draw_start_y < 0)
real_pos_draw_start_y = 0;
// проверки на выход за границу "справа"
int boundary_x = real_pos_draw_start_x + BR.myBrush.Width;
int boundary_y = real_pos_draw_start_y + BR.myBrush.Height;
if(boundary_x > Width)
```

```
boundary_x = Width;
if(boundary_y > Heigth)
boundary_y = Heigth;
// счетчик пройденных строк и столбцов массива, представляющего собой маску
кисти
int count_x = 0, count_y = 0;
// цикл по области с учетом смещения кисти и коррекции для невыхода за грани-
цы массива
for (int ax = real_pos_draw_start_x; ax < boundary_x; ax++, count_x++)
{
count_y = 0;
for (int bx = real_pos_draw_start_y; bx < boundary_y; bx++, count_y++)
{
// получаем текущий цвет пикселя маски
Color ret = BR.myBrush.GetPixel(count_x,count_y);
// цвет не красный
if ( !(ret.R == 255 && ret.G == 0 && ret.B == 0) )
{
// заполняем данный пиксель соответствующим из маски, используя активный
цвет
DrawPlace[ax, bx, 0] = ActiveColor.R;
DrawPlace[ax, bx, 1] = ActiveColor.G;
DrawPlace[ax, bx, 2] = ActiveColor.B;
DrawPlace[ax, bx, 3] = 0;
}
}
}
}
// функция визуализации слоя
public void RenderImage()
{
// данная функция далее будет улучшена для того, чтобы получить более быст-
рую визуализацию
// но пока она выглядит следующим образом
// активируем режим рисования точек
Gl.glBegin(Gl.GL_POINTS);
// проходим по всем точкам рисунка
for (int ax = 0; ax < Width; ax++)
{
for (int bx = 0; bx < Heigth; bx++)
{
// если точка в координатах ax,bx не помечена флагом "прозрачная"
if (DrawPlace[ax, bx, 3] != 1)
```

```
{
// устанавливаем заданный в ней цвет
Gl.glColor3f(DrawPlace[ax, bx, 0], DrawPlace[ax, bx, 1], DrawPlace[ax, bx, 2]);
// и выводим ее на экран
Gl.glVertex2i(ax, bx);
}
}
}
// завершаем режим рисования
Gl.glEnd();
}
}
```

Код подробно прокомментирован и довольно прост. Поясним лишь функцию рисования: в нее передаются в качестве параметров кисть и координаты, в которых должно пройти рисование.

Так как кисть в ширину и высоту может быть больше одного пикселя, нам необходимо определить середину кисти (для этого мы получаем ее ширину и высоту через свойства *Width* и *Heigth* элемента *myBrush*, который является *public* элементом).

Далее мы вычтем из координаты, где должно произойти рисование, половину ширины кисти для оси *X* и половину высоты для оси *Y*. При этом мы можем получить отрицательные значения, если например кисть будет шириной 10 пикселей, и следовательно, нам надо будет сместиться на 5 пикселей влево, чтобы начать алгоритм рисования кистью. Но если при этом координата *X* точки, где должно пройти рисование, будет равна, например 2? Тогда мы получим отрицательный индекс элемента массива, откуда должно начаться рисование, и это приведет к ошибке.

Мы назовем это выходом за границу массива слева. Также можно выйти за границу справа, т.е. индексы получатся превышающими количество объявленных элементов в массиве.

Теперь остается проанализировать код класса *anEngine*.

В нем мы заранее объявим некоторые методы, которые пока не задействованы, но они пригодятся позже.

Код класса *anEngine*:

```
// класс, реализующий "ядро" нашего растрового редактора
public class anEngine
{
// размеры изображения
private int picture_size_x , picture_size_y;
```

```
// положение полос прокрутки будет использовано в будущем
private int scroll_x, scroll_y;
// размер оконной части (объекта AnT)
private int screen_width, screen_height;
// номер активного слоя
private int ActiveLayerNom;
// массив слоев
private ArrayList Layers = new ArrayList();
// стандартная кисть
private anBrush standartBrush;
// конструктор класса
public anEngine(int size_x, int size_y, int screen_w, int screen_h)
{
// при инициализации экземпляра класса сохраним настройки
// размеров элементов и изображения в локальных переменных
picture_size_x = size_x;
picture_size_y = size_y;
screen_width = screen_w;
screen_height = screen_h;
// полосы прокрутки у нас пока отсутствуют, поэтому просто обнулим значение
переменных
scroll_x = 0;
scroll_y = 0;
// добавим новый слой для работы, пока что он будет единственным
Layers.Add( new anLayer(picture_size_x, picture_size_y) );
// номер активного слоя - 0
ActiveLayerNom = 0;
// и создадим стандартную кисть
standartBrush = new anBrush(1,false);
}
// функция для установки номера активного слоя
public void SetActiveLayerNom(int nom)
{
ActiveLayerNom = nom;
}
// установка видимости / невидимости слоя
public void SetWisibilityLayerNom(int nom, bool visible)
{
// вернемся к этой функции позже
}
// рисование текущей кистью
public void Drawing(int x, int y)
```

```
{
// транслируем координаты, в которых проходит рисование стандартной кистью
((anLayer)Layers[0]).Draw(standartBrush, x, y);
}
// визуализация
public void SwapImage()
{
// вызываем функцию визуализации в нашем слое
((anLayer)Layers[0]).RenderImage();
}
}
```

Как видим, некоторые функции объявлены, но не несут никакой функциональной нагрузки.

Визуализация реализована так же, как вызов визуализации одного единственного слоя.

В дальнейшем мы реализуем алгоритм объединения всех слоев в одном для функции *SwapImage*. Этот алгоритм генерирует единственный итоговый слой с учетом иерархии и настроек видимости, и затем уже будет визуализирован.

Но даже такой код уже создает минимальный объем объектной модели программы, который заставит ее работать. На рис. 1.5 представлен пример работы программы.

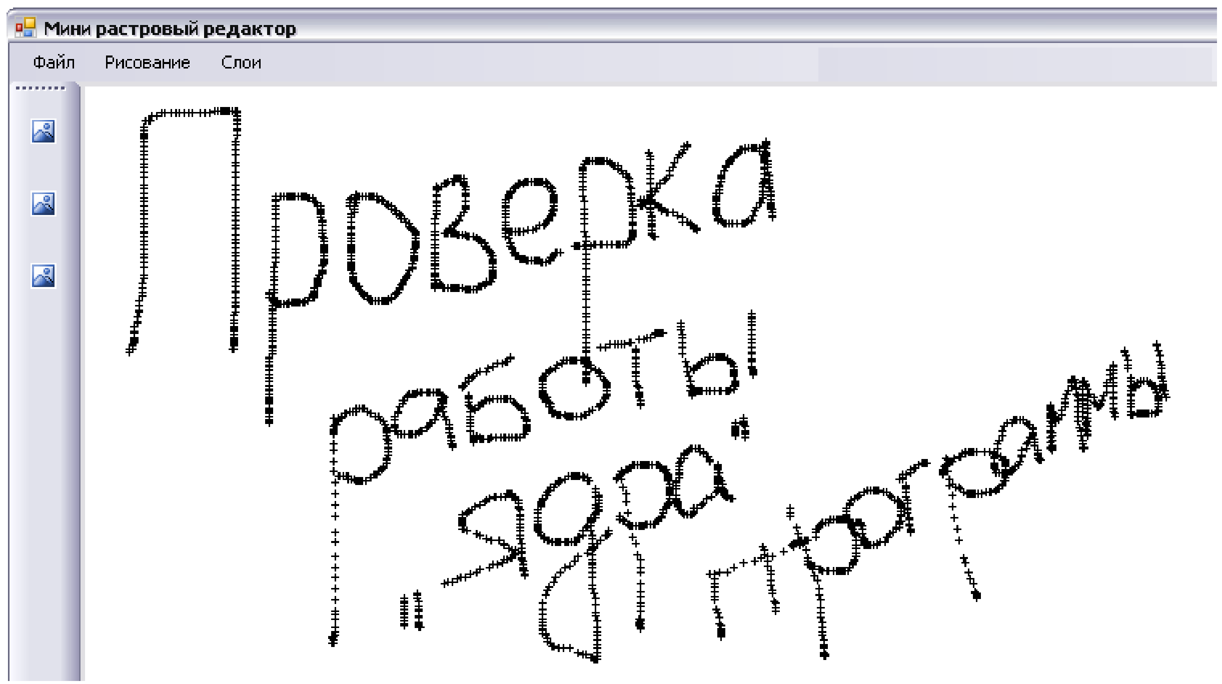


Рис. 1.5

Откомпилировав приложение, можно проверить работу функции рисования.

Теперь можно перейти к более точной реализации системы классов, кистей и визуализации, а также подключить все созданные элементы управления в оболочке программы.

1.2. Растровый редактор: инструменты

Здесь на основе разработанной оболочки будет реализовано несколько элементов рисования и создание возможности выбора рисования необходимым цветом, а также внесены некоторые косметические изменения программы.

Доработка класса, управляющего кистями

Начнем с доработки класса *anBrush* и реализации функций работы с кистями.

Мы полностью переработаем конструктор этого класса. Начнем с того, что конструкторов теперь два.

Первый конструктор public anBrush(int Value, bool Special) отвечает за создание кистей двух классов.

Если параметр *Special* указан как *false*, то создается стандартная кисть размером *Value* х *Value*. Все ее пиксели устанавливаются черными.

В противоположном случае используются специальные кисти, маски которых задаются вручную и хранятся в программе.

В зависимости от параметра *Value* выполняется ветвление кода с помощью конструкции *switch – case*. Кистью по умолчанию назначим кисть, созданную нами вручную в пункте "Создание дополнительных классов и их базовых методов".

Второй конструктор public anBrush(string FromFile) в качестве параметра примет имя bmp-файла, из которого загружаются растровые данные. Этот файл можно создать в любом графическом редакторе. Единственное требование – пиксели, невидимые впоследствии, должны быть заданы красным цветом: RGB(255,0,0).

Обновленный код этого класса:

```
public class anBrush
{
public Bitmap myBrush;
// стандартная (квадратная) кисть, с указанием масштаба
// и флагом закраски узлов
public anBrush(int Value, bool Special)
{
if (!Special)
{
myBrush = new Bitmap(Value, Value);
for (int ax = 0; ax < Value; ax++)
for (int bx = 0; bx < Value; bx++)
myBrush.SetPixel(0, 0, Color.Black);
}
else
{
// здесь мы размещаем предустановленные кисти
// созданная нами ранее кисть в виде перекрестия двух линий – кисть по умолчанию
// на тот случай, если задан неописанный номер кисти
switch (Value)
{
default:
{
myBrush = new Bitmap(5, 5);
for (int ax = 0; ax < 5; ax++)
for (int bx = 0; bx < 5; bx++)
myBrush.SetPixel(ax, bx, Color.Red);
myBrush.SetPixel(0, 2, Color.Black);
myBrush.SetPixel(1, 2, Color.Black);
myBrush.SetPixel(2, 0, Color.Black);
myBrush.SetPixel(2, 1, Color.Black);
myBrush.SetPixel(2, 2, Color.Black);
myBrush.SetPixel(2, 3, Color.Black);
myBrush.SetPixel(2, 4, Color.Black);
myBrush.SetPixel(3, 2, Color.Black);
myBrush.SetPixel(4, 2, Color.Black);
break;
}
}
}
}
```

```
// второй конструктор позволяет загружать кисть из стороннего файла
public anBrush(string FromFile)
{
string path = Directory.GetCurrentDirectory();
path += "\\" + FromFile;
myBrush = new Bitmap(path);
}
}
```

Создание возможности выбора цвета и рисования заданным цветом

Теперь обратимся к классу *anEngine*.

Вначале добавим переменную, которая хранит последний установленный цвет. Это необходимо для того, чтобы при установке цвета он назначался текущему активному слою. Назначать ее сразу всем слоям (циклическим перебором слоев) неудобно.

Мы просто храним последний установленный цвет, и когда далее добавим функции смены слоев, новому установленному или созданному слою устанавливается данный цвет в качестве активного.

```
// последний установленный цвет
private Color LastColorInUse;
```

Здесь теперь в конструкторе класса создается начальная кисть, заданная следующим образом:

```
// создание кисти по умолчанию в конструкторе класса anEngine
standartBrush = new anBrush(3,false);
```

Также мы добавим три функции, которые вызываются нажатиями кнопок на панели инструментов в левой части окна создаваемой программы для установки кистей:

```
// функция установки стандартной кисти, передается только размер
public void SetStandartBrush(int SizeB)
{
   standartBrush = new anBrush(SizeB, false);
}
// функция установки специальной кисти
public void SetSpecialBrush(int Nom)
{
   standartBrush = new anBrush(Nom, true);
}
// установка кисти из файла
public void SetBrushFromFile(string FileName)
```

```
{
   standartBrush = new anBrush(FileName);
}
```

Добавим функции для установки активного цвета:

```
// функция установки активного цвета
public void SetColor(Color NewColor)
{
   ((anLayer)Layers[ActiveLayerNom]).SetColor(NewColor);
   LastColorInUse = NewColor;
}
```

То есть мы вызываем функцию SetColor для активного в данный момент (ActiveLayerNom) слоя. Класс anLayer ранее не содержал такой функции, так что нашим следующим действием будет ее добавление.

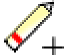
Перейдем к классу anLayer и добавим реализацию данной функции:

```
// установка текущего цвета для рисования в слое
public void SetColor(Color NewColor)
{
   ActiveColor = NewColor;
}
```

Обновление дизайна и функциональности оболочки растрового редактора

Теперь перейдем к редактированию оболочки. Сначала обновим кнопки для установки кистей.

Используйте данные изображения для размещения на кнопках.

Кнопка 1: Кнопка 2:  Кнопка 3:

Теперь панель выглядит следующим образом (рис. 1.6).

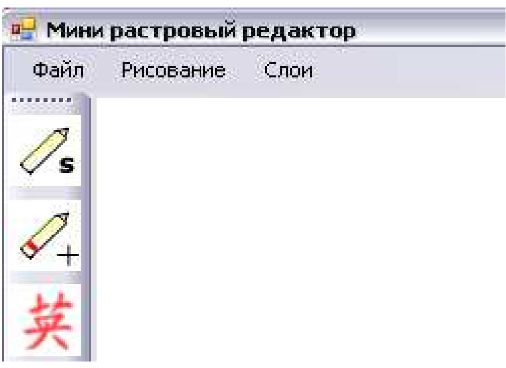


Рис. 1.6

Код для обработки события нажатия на эти кнопки выглядит следующим образом:

```
private void toolStripButton1_Click(object sender, EventArgs e)
{
   // устанавливаем стандартную кисть 4x4
   ProgrammDrawingEngine.SetStandartBrush(4);
}
private void toolStripButton2_Click(object sender, EventArgs e)
{
   // устанавливаем специальную кисть
   ProgrammDrawingEngine.SetSpecialBrush(0);
}
private void toolStripButton3_Click(object sender, EventArgs e)
{
   // установить кисть из файла
   ProgrammDrawingEngine.SetBrushFromFile( "brush-1.bmp");
}
```

Для того чтобы кисть из файла успешно установилась, необходимо добавить данное изображение в папку *bin -> debug* текущего проекта.

Добавление функций установки текущего цвета

Теперь добавим на нашу форму дополнительные элементы:

1. Элемент *colorDialog* из панели инструментов (рис. 1.7). После добавления переименуйте данный элемент в "*changeColor*" (свойство *name*).

2. Два элемента *panel*. Переименуйте их в *color1* и *color2*. Расположение этих элементов показано на рис. 1.8.

3. Элемент *LinkLabel*. Параметр *text* в его свойствах установите равным "поменять". Положение элемента также представлено на рис. 1.9.

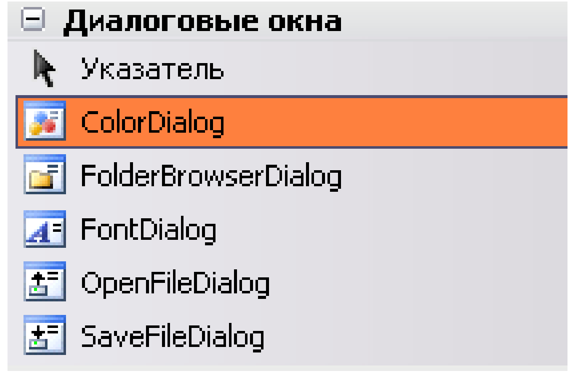


Рис. 1.7

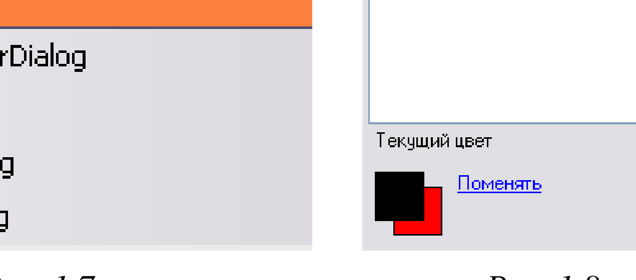


Рис. 1.8

Свойство *BackColor* для элемента *color1* установите равным *black*. Для элемента *color2* – любое. Также установите границы обоих элементов: свойство *BorderStyle*, равное *FixedSingle*. Данная комбинация элементов отвечает за установку текущего цвета.

Добавьте обработчик нажатия клавиши мыши на элементе *color1* (*MouseClick*) и на элементе *linklabel*. При щелчке на элементе *color1* открыто диалоговое окно выбора цвета, и если цвет успешно выбран, его значение будет установлено переменной *BackColor* элемента *color1*. Так он виден пользователю. После установки этот цвет передается в класс *anEngine* для дальнейшей установки активным цветом текущего слоя.

В случае щелчка мыши на элементе *LinkLabe* (текст "поменять") производится обмен цветов у элементов *color1* и *color2*. Далее цвет, установленный для *color1*, передается в класс *anEngine*.

Код обработки данных событий

```
// обмен значений цветов
private void linkLabel1_LinkClicked(object sender, LinkLabelLinkClickedEventArgs e)
{
   // временное хранение цвета элемента color1
   Color tmp = color1.BackColor;
   // замена:
   color1.BackColor = color2.BackColor;
   color2.BackColor = tmp;
   // передача нового цвета в ядро растрового редактора
   ProgrammDrawingEngine.SetColor(color1.BackColor);
}
// функция установки нового цвета с помощью диалогового окна выбора цвета
private void color1_MouseClick(object sender, MouseEventArgs e)
{
   // если цвет успешно выбран
   if (changeColor.ShowDialog() == DialogResult.OK)
   {
      // установить данный цвет
      color1.BackColor = changeColor.Color;
      // и передать его в класс anEngine для установки активным цветом текущего
слоя
      ProgrammDrawingEngine.SetColor(color1.BackColor);
   }
}
```

Пример работы программы (рис. 1.9).

Рис. 1.9

1.3. Растровый редактор: система слоев

Цель – завершение реализации системы слоев и обновление системы их визуализации, повышающее быстродействие и учитывающее алгоритм их взаимодействия при визуализации.

Система слоев

Внесем в программу некоторые изменения, направленные на то, чтобы при запуске программы происходила именно инициализация нулевого слоя (на данный момент она жестко определяется нулевым индексом, что затрудняет обновление системы слоев).

Следующим шагом станет добавление обработчиков событий в окне формы, направленных на работу со слоями, а также создание соответствующих функций в классе движка.

На данном этапе взаимодействие слоев не используется, но в дальнейшем на нем основывается применение цветовых фильтров.

Сначала обратимся к форме окна.

Переименуем элемент *checkedListBox1* в *LayersControl*.

Теперь перейдем к функции *Form1_Load*. Перед ней объявим три переменные

```
// текущий активный слой
private int ActiveLayer = 0;
```

```
// счетчик слоев
private int LayersCount = 1;
// счетчик всех создаваемых слоев для генерации имен
private int AllLayrsCount = 1;
```

В коде самой функции добавится строка

```
// добавление элемента, отвечающего за управления главным слоем в объект
LayersControl
LayersControl.Items.Add("Главный слой", true );
```

Теперь мы рассмотрим добавление еще трех функций в оболочку программы, после чего перейдем к их реализации в коде движка программы.

Добавим обработчики функций "добавить слой" и "удалить слой".

В функции "добавить слой" создадим элемент с именем слоя, содержащий его порядковый номер; выделим слой и пометим его как активный; вызовем функцию движка *AddLayer()*, где будет выполняться код, дублирующий создания слоя, но уже не в оболочке программы для управления им, а в движке – для рисования.

Функцию *AddLayer* в движок программы мы добавим позже.

```
// функция добавления слоя
private void добавитьСлойToolStripMenuItem_Click( object sender, EventArgs e)
{
   // счетчик созданных слоев
   LayersCount ++;
   // вызываем функцию добавления слоя в движке графического редактора
   ProgrammDrawingEngine.AddLayer();
   // добавляем слой, генерируем имя "Слой №" в объекте LayersControl
   // обязательно после функции ProgrammDrawingEngine.AddLayer();,
   // иначе произойдет попытка установки активного цвета для еще не суще-
ствующего цвета
   int AddingLayerNom = LayersControl.Items.Add("Слой" + LayersCount.ToString(),
false );
   // выделяем его
   LayersControl.SelectedIndex = AddingLayerNom;
   // устанавливаем его как активный
   ActiveLayer = AddingLayerNom;
}
```

В функции «удалить слой» сначала запрашиваем подтверждение на удаление слоя с помощью *MessageBox*’а. Затем (если удаляемый

слой не нулевой) выполняем удаление выделенного слоя и вызываем функцию *RemoveLayer* движка графического редактора (ее код мы добавим в программу позже).

```
// функция удаления слоя
private void удалитьСлойToolStripMenuItem_Click( object sender, EventArgs e)
{
   // запрашиваем подтверждение действия с помощью messageBox
   DialogResult res = MessageBox.Show("Будет удален текущий активный слой.
   Продолжить?", "Внимание!", MessageBoxButtons.YesNo,
   MessageBoxIcon.Warning);
   // если пользователь нажал кнопку "ДА" в окне подтверждения
   if( res == DialogResult.Yes)
   {
      // если удаляемый слой - начальный
      if (ActiveLayer == 0)
      {
         // сообщаем о невозможности удаления
         MessageBox.Show("Вы не можете удалить нулевой слой.", "Внимание!",
         MessageBoxButtons.OK, MessageBoxIcon.Stop);
      }
      else // иначе
      { // уменьшаем значение счетчика слоев
         LayersCount--;
         // сохраняем номер удаляемого слоя,
         // т.к. SelectedIndex изменится после операций в LayersControl
         int LayerNomForDel = LayersControl.SelectedIndex;
         // удаляем запись в элементе LayersControl
         // (с индексом LayersControl.SelectedIndex - текущим выделенным слоем)
         LayersControl.Items.RemoveAt(LayerNomForDel);
         // устанавливаем выделенный слоем - нулевой (главный слой)
         LayersControl.SelectedIndex = 0;
         // помечаем активный слой - нулевой
         ActiveLayer = 0;
         // помечаем галочкой нулевой слой
         LayersControl.SetItemCheckState(0, CheckState.Checked);
         // вызываем функцию удаления слоя в движке программы
         ProgrammDrawingEngine.RemoveLayer(LayerNomForDel);
      }
   }
}
```

Теперь выделим элемент *LayersControl*, после чего добавим ей обработчик события *SelectedValueChanged*.

Код этой функции:

```
// данная функция обрабатывает изменения значения элементов LayersControl
private void LayersControl_SelectedValueChanged( object sender, EventArgs e)
{ // если отметили новый слой, необходимо снять галочку выделения со старого
   if (LayersControl.SelectedIndex != ActiveLayer)
   { // если выделенный индекс является корректным
      // (больше либо равен нулю и входит в диапазон элементов)
      if (LayersControl.SelectedIndex != -1 && ActiveLayer < LayersCon-
trol.Items.Count)
      { // снимаем галочку с предыдущего активного слоя
         LayersControl.SetItemCheckState(ActiveLayer, CheckState.Unchecked);
         // сохраняем новый индекс выделенного элемента
         ActiveLayer = LayersControl.SelectedIndex;
         // помечаем галочкой новый активный слой
         LayersControl.SetItemCheckState(LayersControl.SelectedIndex, Check-
State.Checked);
         // посылаем движку программы сигнал об изменении активного слоя
         ProgrammDrawingEngine.SetActiveLayerNom(ActiveLayer);
} } }
```

Теперь внесем изменения в код движка графического редактора (класс *anEngine*).

Сначала добавим функции, отвечающие за добавление и удаление слоев:

```
// функция добавления слоя
public void AddLayer()
{
// добавляем слой в массив слоев ArrayList
int AddingLayer = Layers.Add( new anLayer(picture_size_x, picture_size_y));
// устанавливаем его активным
SetActiveLayerNom(AddingLayer);
}
// функция удаления слоев
public void RemoveLayer( int nom)
{
// если номер корректен (в диапазоне добавленных в ArrayList)
if (nom < Layers.Count && nom >= 0)
```

```
{
// удаляем запись о слое
Layers.RemoveAt(nom);
// делаем активным слой 0
SetActiveLayerNom(0);
}
}
```

Теперь внесем изменения в функции *Drawing*, *SetColor* и *SwapImage*.

Здесь вместо жестко определенного нами ранее «нулевого» слоя теперь применяется активный слой, поэтому в коде этих функций используется переменная *ActiveLayerNom*, вместо «0»:

```
// рисование текущей кистью
public void Drawing( int x, int y)
{
// транслируем координаты, в которых проходит рисование стандартной кистью
((anLayer)Layers[ActiveLayerNom]).Draw(standartBrush, x, y);
}
// функция установки активного цвета
public void SetColor(Color NewColor)
{
((anLayer)Layers[ActiveLayerNom]).SetColor(NewColor);
LastColorInUse = NewColor;
}
// визуализация
public void SwapImage()
{
// вызываем функцию визуализации в нашем слое
((anLayer)Layers[ActiveLayerNom]).RenderImage();
}
```

Далее внесём небольшие изменения в функцию *SetActiveLayer Nom*, так как теперь при изменении активного слоя необходимо изменять его текущий активный цвет.

Код этой функции выглядит следующим образом:

```
// функция для установки номера активного слоя
public void SetActiveLayerNom( int nom)
{
   // новый активный слой получает установленный активный цвет для предыдущего активного слоя
   ((anLayer)Layers[nom]).SetColor( ((anLayer)Layers[ActiveLayerNom]).GetColor() );
```

```
    // установка номера активного слоя
    ActiveLayerNom = nom;
}
```

Также необходимо добавить реализацию функции *GetColor* для класса *anLayer*:

```
// получение текущего активного цвета
public Color GetColor()
{
    // возвращаем цвет
    return ActiveColor;
}
```

Система слоев готова. Теперь программа сможет создавать слои, и рисование будет происходить в устанавливаемых слоях.

Алгоритм визуализации

Далее изменим алгоритм визуализации. Ранее визуализировался активный слой, теперь визуализация выполняется для всех существующих слоев.

```
// визуализация
public void SwapImage()
{
    // вызываем функцию визуализации в нашем слое для всех существующих
слоев
    for( int ax = 0; ax < Layers.Count; ax++)
    ((anLayer)Layers[ax]).RenderImage();
}
```

Имея в 2D-редакторе систему слоев, можно добавлять ему различную функциональность – отключение какого-либо слоя, реализацию прозрачности и т.д.

Добавим две дублирующие кнопки на правой панели инструментов: первая будет вызывать функцию добавления слоя, вторая – функцию удаления слоя.

Для этого достаточно сделать двойной щелчок на данных кнопках, и обработчики события создадутся автоматически. Остается только добавить вызовы функций, которые мы использовали для добавления и удаления слоев.

```
// дублирование создания слоя
private void toolStripButton4_Click( object sender, EventArgs e)
```

```
{
   добавитьСлойToolStripMenuItem_Click(sender, e);
}
// дублирование удаления слоя
private void toolStripButton5_Click( object sender, EventArgs e)
{
   удалитьСлойToolStripMenuItem_Click(sender, e);
}
```

Не забывайте при установке изображений указывать параметр *ImageScaling* равным *none*.

Добавление элемента "ластик"

Добавим для элемента "ластик" кнопку на левую панель инструментов и вызов соответствующей функции из класса *anEngine*.

Заодно продублируем элементы меню "Рисование", добавленные нами ранее.

Обработчик нажатия на новую кнопку выглядит следующим образом:

```
// обработка кнопки "ластик" на левой панели инструментов
private void toolStripButton6_Click( object sender, EventArgs e)
{
   // установка кисти-ластика
   ProgrammDrawingEngine.SetSpecialBrush(1);
}
```

Теперь перейдем к изменениям в исходном коде движка.

В первую очередь изменим систему создания (установки) кистей.

В классе *anBrush* появился закрытый флаг, сигнализирующий о том, что данная кисть является ластиком. Также появился метод для определения значения этого флага:

```
// флаг, сигнализирующий о том, что установленная кисть является ластиком
private bool IsErase = false ;
// функция, которая используется для получения информации
// о том, является ли данная кисть ластиком
public bool IsBrushErase()
{
   return IsErase;
}
```

При создании кисти теперь автоматически помечается, что она не является ластиком (кроме специальной кисти номер 1, которая как раз и есть ластик). Код конструктора с изменениями:

```
public anBrush( int Value, bool Special)
{
if (!Special)
{
myBrush = new Bitmap(Value, Value);
for ( int ax = 0; ax < Value; ax++)
for ( int bx = 0; bx < Value; bx++)
myBrush.SetPixel(0, 0, Color.Black);
// не является ластиком
IsErase = false ;
}
else
{
// здесь мы размещаем предустановленные кисти
// созданная нами ранее кисть в виде перекрестия двух линий будет кистью по
умолчанию
// на тот случай, если задан не описанный номер кисти
switch (Value)
{
// специальная кисть по умолчанию
default :
{
myBrush = new Bitmap(5, 5);
for ( int ax = 0; ax < 5; ax++)
for ( int bx = 0; bx < 5; bx++)
myBrush.SetPixel(ax, bx, Color.Red);
myBrush.SetPixel(0, 2, Color.Black);
myBrush.SetPixel(1, 2, Color.Black);
myBrush.SetPixel(2, 0, Color.Black);
myBrush.SetPixel(2, 1, Color.Black);
myBrush.SetPixel(2, 2, Color.Black);
myBrush.SetPixel(2, 3, Color.Black);
myBrush.SetPixel(2, 4, Color.Black);
myBrush.SetPixel(3, 2, Color.Black);
myBrush.SetPixel(4, 2, Color.Black);
// не является ластиком
IsErase = false ;
break ;
}
```

```
case 1: // стерка
{
// создается так же, как и обычная кисть,
// но имеет флаг IsErase равный true
myBrush = new Bitmap(5, 5);
for ( int ax = 0; ax < Value; ax++)
for ( int bx = 0; bx < Value; bx++)
myBrush.SetPixel(0, 0, Color.Black);
// является ластиком
IsErase = true ;
break ;
} } } }
```

Следующий шаг – рисование в слое.

Перейдите к функции *Draw* класса *anLayer*.

Здесь в цикле, где происходил перебор пикселей из маски кисти, произошли изменения.

Теперь перед тем как нарисовать конкретный пиксель, производится проверка – не является ли данная кисть ластиком. Если является и при этом пиксель помечен в маске непрозрачным (для этого в маске мы договорились использовать красный цвет), то на рисунке он будет помечен как отсутствующий (невизуализируемый):

```
// цикл по области с учетом смещения кисти и коррекции для невыхода за границы массива
for ( int ax = real_pos_draw_start_x; ax < boundary_x; ax++, count_x++)
{ count_y = 0;
   for ( int bx = real_pos_draw_start_y; bx < boundary_y; bx++, count_y++)
   { // проверяем, не является ли данная кисть ластиком
      if (BR.IsBrushErase())
      { // данная кисть - ластик
         // помечаем данный пиксель как незакрашенный
         // получаем текущий цвет пикселя маски
         Color ret = BR.myBrush.GetPixel(count_x, count_y);
         // цвет не красный
         if (!(ret.R == 255 && ret.G == 0 && ret.B == 0))
         { // заполняем данный пиксель активным цветом из маски
         DrawPlace[ax, bx, 3] = 1;
      } }
```

```
        else
        { // получаем текущий цвет пикселя маски
            Color ret = BR.myBrush.GetPixel(count_x, count_y);
            // цвет не красный
            if (!(ret.R == 255 && ret.G == 0 && ret.B == 0))
            { // заполняем данный пиксель активным цветом из маски
                DrawPlace[ax, bx, 0] = ActiveColor.R;
                DrawPlace[ax, bx, 1] = ActiveColor.G;
                DrawPlace[ax, bx, 2] = ActiveColor.B;
                DrawPlace[ax, bx, 3] = 0;
} } } }
```

Отметим, что происходит стирание только тех пикселей, которые были нарисованы в данном слое.

Элементы меню "Рисование"

В заключение создадим обработчики нажатия элементов меню "Рисование".

Добавьте для каждого пункта меню обработчик двойным щелчком мыши.

Затем добавьте вызовы уже существующих функций (обработчиков нажатия на кнопки панели инструментов):

```
// дублирование установки кисти "карандаш" из меню "рисование"
private void карандашToolStripMenuItem_Click( object sender, EventArgs e)
{
    // вызываем уже существующую функцию
    toolStripButton1_Click(sender, e);
}
// дублирование установки кисти "кисть" из меню "рисование"
private void кистьToolStripMenuItem_Click( object sender, EventArgs e)
{
    // вызываем уже существующую функцию
    toolStripButton3_Click(sender, e);
}
// дублирование установки кисти "ластик" из меню "рисование"
private void стеркаToolStripMenuItem_Click( object sender, EventArgs e)
{
    // вызываем уже существующую функцию
    toolStripButton6_Click(sender, e);
}
```

1.4. Растровый редактор: оболочка программы

Здесь мы завершим разработку оболочки программы, демонстрирующей принцип работы графического редактора. Работа сводится к развитию функциональности меню «Файл» – добавлению средств создания новых проектов, проектов на основе графического файла, а также сохранения текущего проекта в формате *jpg*.

Добавление возможности загрузки изображений в формате jpg

Загрузчик основан на существующих классах *.net*, что определяет их невысокое быстродействие. Подгружаемые изображения устанавливают на главный слой, после чего можно проводить редактирование и сохранение проекта в формате *jpg*.

Если размер загружаемого изображения окажется меньше области редактирования, оно установится в левом нижнем углу окна. Необходимо также добавить в исходный код движка функцию для создания результирующего слоя перед его сохранением. Первая функция, которую мы добавим в класс *anEngine*, – функция создания результирующего слоя. Неоптимальная по быстродействию, она имеет простую для понимания реализацию. Функция вызывается из оболочки при сохранении результатов работы в *jpg* файл.

```
// получение финального изображения
public Bitmap GetFinalImage()
{
// заготовка результирующего изображения
Bitmap resaultBitmap = new Bitmap(picture_size_x, picture_size_y);
// данное решение также не является оптимальным по быстродействию, но при
этом является самым простым способом решения задачи
for ( int ax = 0; ax < Layers.Count; ax++)
{
// получаем массив пикселей данного слоя
int [,,] tmp_layer_data = ((anLayer)Layers[ax]).GetDrawingPlace();
// пройдем двумя циклами по информации о пикселях данного слоя
for ( int a = 0; a < picture_size_x; a++)
{
for( int b = 0; b < picture_size_y; b++)
{
// если пиксель не помечен как "прозрачный"
if (tmp_layer_data[a, b, 3] != 1)
```

```
{
// устанавливаем данный пиксель на результирующее изображение
resaultBitmap.SetPixel(a,b, Color.FromArgb(tmp_layer_data[a, b, 0],
tmp_layer_data[a, b, 1], tmp_layer_data[a, b, 2]));
}
else
{
if (ax == 0) // нулевой слой - необходимо закрасить белым остутствующие пиксе-
ли
{
// закрашиваем белым цветом
resaultBitmap.SetPixel(a, b, Color.FromArgb(255, 255, 255));
} } } } }
// поворачиваем изображение для корректного отображения
resaultBitmap.RotateFlip(RotateFlipType.Rotate180FlipX);
// возвращаем результат
return resaultBitmap;
}
```

Добавим функцию *SetImageToMainLayer*, которая будет служить для отрисовки загружаемого изображения на рабочей плоскости редактора.

Принцип ее работы также прост. Скорость обработки невысока, но (поскольку мы не можем загружать изображения больше размера рабочей плоскости редактирования) достаточна.

```
// получение изображения для главного слоя
public void SetImageToMainLayer(Bitmap layer)
{
// поворачиваем изображение (чтобы оно корректно отображалось в области ре-
дактирования)
layer.RotateFlip(RotateFlipType.Rotate180FlipX);
// проходим двумя циклами по всем пикселям изображения, подгруженного в
класс Bitmap
// получая цвет пикселя, устанавливаем его в текущий слой с помощью функции
Drawing
// данный алгоритм является медленным, но простым оптимальным решением
здесь
// было бы написание собственного загрузчика файлов изображений, что дало бы
// возможность без "посредников" получать массив значений пикселей изобра-
жений,
// но данная задача является намного более сложной и выходит за рамки обуча-
ющей программы
```

```
for ( int ax = 0; ax < layer.Width; ax++)
{
for ( int bx = 0; bx < layer.Height; bx++)
{
// получения цвета пикселя изображения
SetColor(layer.GetPixel(ax,bx));
// отрисовка данного пикселя в слое
Drawing(ax, bx);
}
}
}
```

Развитие функциональности меню "Файл"

Теперь вернемся к оболочке нашей программы.

Сейчас мы должны реализовать функциональные возможности меню файл.

Сначала, воспользовавшись окном *ToolBox*, добавим к проекту объекты *openFileDialog* и *saveFileDialog*.

Визуально они разместятся рядом с объектами таймера и панелей инструментов.

Далее двойными щелчками мыши добавим обработчики события нажатия всех элементов меню «Файл»:

- Новый проект,
- Из файла,
- Сохранить,
- Выход.

Начнем с функции для нового проекта. Данная функция создает новый объект для *ProgrammDrawingEngine*, оставляя все ранее созданные подклассы и переменные в данном экземпляре этого класса на растерзание сборщика мусора, реализованного в среде *.NET*.

Функция довольно проста: пользователю предлагается сохранить результат его работы на компьютере и в случае положительного ответа мы сохраняем проект, используя рассмотренную ранее функцию *GetFinalImage* для получения результирующего изображения (суммированием всех слоев).

Код обработчика данного пункта меню выглядит следующим образом:

```
// функция создания нового проекта для рисования
private void чистыйПроектToolStripMenuItem_Click( object sender, EventArgs e)
```

```
{
// вызываем диалог подтверждения
DialogResult reslt = MessageBox.Show("В данный момент проект уже начат, со-
хранить изменения перед закрытием проекта?", "Внимание!",
MessageBoxButtons.YesNoCancel);
// в зависимости от результата нажатия кнопки пользователем в окне MessageBox
switch (reslt)
{
// если отказ пользователя
case DialogResult.No:
{
// просто создаем чистый проект
ProgrammDrawingEngine = new anEngine(AnT.Width, AnT.Height, AnT.Width,
AnT.Height);
// очищаем информацию о добавленных ранее слоях
LayersControl.Items.Clear();
// вновь инициализируем нулевой слой:
// текущий активный слой
ActiveLayer = 0;
// счетчик слоев
LayersCount = 1;
// счетчик всех создаваемых слоев для генерации имен
AllLayrsCount = 1;
// добавление элемента, отвечающего за управление главным слоем, в объект
LayersControl
LayersControl.Items.Add("Главный слой", true );
break ;
}
case DialogResult.Cancel:
{
// возвращаемся
return ;
}
case DialogResult.Yes:
{
// открываем окно сохранения файла и, если имя файла указано и DialogResult
вернуло сигнал об успешном нажатии кнопки ОК
if (saveFileDialog1.ShowDialog() == DialogResult.OK)
{
// получаем результирующее изображение слоя
Bitmap ToSave = ProgrammDrawingEngine.GetFinalImage();
// сохраняем, используя имя файла, указанное в диалоговом окне сохранения
файла
```

```
ToSave.Save(saveFileDialog1.FileName, System.Drawing.Imaging.ImageFormat.Jpeg);
// сохранили - начинаем новый проект:
// создаем новый объект "движка" программы
ProgrammDrawingEngine = new anEngine(AnT.Width, AnT.Height, AnT.Width,
AnT.Height);
// очищаем информацию о добавляемых ранее слоях
LayersControl.Items.Clear();
// вновь инициализируем нулевой слой:
// текущий активный слой
ActiveLayer = 0;
// счетчик слоев
LayersCount = 1;
// счетчик всех создаваемых слоев для генерации имен
AllLayrsCount = 1;
// добавление элемента, отвечающего за управление главным слоем, в объект
LayersControl
LayersControl.Items.Add("Главный слой", true );
}
else
{
// если сохранение не завершилось нормально (скорее всего пользователь закрыл
// окно сохранения файла), возвращаемся в проект
return ;
}
break ;
} }
```

Функция для элемента меню "Из файла" может показаться более сложной, но на самом деле все так же просто, только в этот раз дополнительно будет осуществляться взаимодействие с объектом *openFileDialog*, для того чтобы получить адрес открываемого файла, а также дополнительно проводится проверка его существования.

Комментарии пояснят этот объемный, но простой код:

```
// функция создания нового проекта для рисования
private void чистыйПроектToolStripMenuItem_Click( object sender, EventArgs e)
{
// вызываем диалог подтверждения
DialogResult reslt = MessageBox.Show("В данный момент проект уже начат, со-
хранить изменения перед закрытием проекта?", "Внимание!",
MessageBoxButtons.YesNoCancel);
// в зависимости от результата нажатия кнопки пользователем в окне MessageBox
switch (reslt)
```

```
{
// если отказ пользователя
case DialogResult.No:
{
// просто создаем чистый проект
ProgrammDrawingEngine = new anEngine(AnT.Width, AnT.Height, AnT.Width,
AnT.Height);
// очищаем информацию о добавляемых ранее слоях
LayersControl.Items.Clear();
// вновь инициализируем нулевой слой:
// текущий активный слой
ActiveLayer = 0;
// счетчик слоев
LayersCount = 1;
// счетчик всех создаваемых слоев для генерации имен
AllLayrsCount = 1;
// добавление элемента, отвечающего за управление главным слоем, в объект
LayersControl
LayersControl.Items.Add("Главный слой", true );
break ;
}
case DialogResult.Cancel:
{
// загрузка изображения в рабочую область программы
private void изФайлаToolStripMenuItem_Click( object sender, EventArgs e)
{
// вызываем диалог подтверждения
DialogResult reslt = MessageBox.Show("В данный момент проект уже начат, со-
хранить изменения перед закрытием проекта?", "Внимание!", MessageBoxBut-
tons.YesNoCancel); switch (reslt)
{
// если отказ пользователя
case DialogResult.No:
{
// просто создаем проект, подгружая изображение
if (openFileDialog1.ShowDialog() == DialogResult.OK)
{
// проверяем существование файла
if ( System.IO.File.Exists(openFileDialog1.FileName))
{
// загружаем изображение в экземпляр класса Bitmap
Bitmap ToLoad = new Bitmap(openFileDialog1.FileName);
// если размер изображения некорректен
if (ToLoad.Width > AnT.Width || ToLoad.Height > AnT.Height)
```

```
{
// сообщаем пользователю об ошибке
MessageBox.Show("Извините, но размер изображения превышает размеры обла-
сти рисования", "Внимание", MessageBoxButtons.OK, MessageBoxIcon.Stop);
// возвращаемся к функции
return ;
}
// если размер был меньше области редактирования программы
// создаем новый экземпляр класса anEngine
ProgrammDrawingEngine = new anEngine(AnT.Width, AnT.Height, AnT.Width,
AnT.Height);
// копируем изображение в нижний левый угол рабочей области
ProgrammDrawingEngine.SetImageToMainLayer(ToLoad);
// очищаем информацию о добавляемых ранее слоях
LayersControl.Items.Clear();
// вновь инициализируем нулевой слой:
// текущий активный слой
ActiveLayer = 0;
// счетчик слоев
LayersCount = 1;
// счетчик всех создаваемых слоев для генерации имен
AllLayrsCount = 1;
// добавление элемента, отвечающего за управление главным слоем, в объект
LayersControl
LayersControl.Items.Add("Главный слой", true );
} }
break ;
}
case DialogResult.Cancel:
{
// возвращаемся
return ;
}
case DialogResult.Yes:
{
// открываем окно сохранения файла, и если имя файла указано и DialogResult
вернуло сигнал об успешном нажатии кнопки OK
if (saveFileDialog1.ShowDialog() == DialogResult.OK)
{
// получаем результирующее изображение слоя
Bitmap ToSave = ProgrammDrawingEngine.GetFinalImage();
// сохраняем, используя имя файла, указанное в диалоговом окне сохранения
файла
ToSave.Save(saveFileDialog1.FileName, System.Drawing.Imaging.ImageFormat.Jpeg);
// сохранили – начинаем новый проект:
```

```
// просто создаем проект, подгружая изображения
if (openFileDialog1.ShowDialog() == DialogResult.OK)
{
// проверяем существование файла
if ( System.IO.File.Exists(openFileDialog1.FileName))
{
// загружаем изображение в экземпляр класса Bitmap
Bitmap ToLoad = new Bitmap(openFileDialog1.FileName);
// если размер изображения некорректен
if (ToLoad.Width > AnT.Width || ToLoad.Height > AnT.Height)
{
// сообщаем пользователю об ошибке
MessageBox.Show("Извините, но размер изображения превышает размеры обла-
сти рисования", "Внимание", MessageBoxButtons.OK, MessageBoxIcon.Stop);
// возвращаемся к функции
return ;
}
// если размер был меньше области редактирования программы
// создаем новый экземпляр класса anEngine
ProgrammDrawingEngine = new anEngine(AnT.Width, AnT.Height, AnT.Width,
AnT.Height);
// копируем изображение в нижний левый угол рабочей области
ProgrammDrawingEngine.SetImageToMainLayer(ToLoad);
// очищаем информацию о добавляемых ранее слоях
LayersControl.Items.Clear();
// вновь инициализируем нулевой слой:
// текущий активный слой
ActiveLayer = 0;
// счетчик слоев
LayersCount = 1;
// счетчик всех создаваемых слоев для генерации имен
AllLayrsCount = 1;
// добавление элемента, отвечающего за управление главным слоем, в объект
LayersControl
LayersControl.Items.Add("Главный слой", true );
}
}
break ;
}
else
{
return ;
}
break ;
} } }
```

Функция-обработчик сохранения в меню "Файл" (код довольно прост и снабжен комментариями):

```
// обработка нажатия кнопки "сохранить" в меню "файл"
private void сохранитьToolStripMenuItem_Click( object sender, EventArgs e)
{
   // открываем окно сохранения файла и, если имя файла указано и DialogResult
вернуло сигнал об успешном нажатии кнопки ОК
   if (saveFileDialog1.ShowDialog() == DialogResult.OK)
   {
      // получаем результирующее изображение слоя
      Bitmap ToSave = ProgrammDrawingEngine.GetFinalImage();
      // сохраняем, используя имя файла, указанное в диалоговом окне сохране-
ния файла
      ToSave.Save(saveFileDialog1.FileName, Sys-
tem.Drawing.Imaging.ImageFormat.Jpeg);
   }
}
```

Нам остается рассмотреть лишь код функции-обработчика события нажатия на элемент "Выход" в меню "Файл":

```
private void выходToolStripMenuItem_Click( object sender, EventArgs e)
{
Application.Exit();
}
```

1.5. Растровый редактор: оптимизация функций

Если вы попытались поработать с редактором, то уже заметили, что при большом количестве точек визуализация работает крайне медленно. Это и не удивительно: метод перебора массива вершин для каждого слоя и поочередного их рисования – это не лучшее решение данной задачи. Поэтому в данном разделе мы переработаем функциональную часть нашего графического *2D*-редактора, отвечающую за визуализацию.

Дисплейные списки

Самое главное – мы откажемся от непрерывной перерисовки всех слоев: это сильно замедляет работу приложения и может быть применено только на начальных этапах создания программы как основа визуализации.

Повысить быстродействие приложения нам помогут дисплейные списки *OpenGL* – пронумерованные области памяти, в которых хранится информация, используемая для визуализации.

Каждый слой (кроме текущего активного слоя) при вызове функции визуализации вызывает только заранее подготовленный дисплейный список, изменения в котором происходят лишь при внесении изменений в слой.

Тогда само применение слоев вместо понижения производительности, которое происходило при переборе и визуализации каждого слоя, даст ощутимый прирост быстродействия: неактивные слои вызовут только готовые дисплейные списки, откомпилированные в памяти графического адаптера, и их визуализация пройдет намного быстрее.

Так, например, поместив изображение в нижний слой и производя формирование новой графической сцены в другом слое, мы практически не тратим системные ресурсы.

Также оптимизируем сам алгоритм отрисовки при создании дисплейного списка – не отрисовываем каждую точку изображения поточечным перебором, а подготовим массив для быстрой отрисовки.

Простым примером иллюстрации работы дисплейных списков может служить *3D*-модель игрового персонажа – однократно построив и отрисовав ее в программе, нет необходимости затем каждый раз выполнять сложный алгоритм отрисовки, если можно сохранить модель в дисплейном списке и отрисовывать простым вызовом уже частично обработанных и кешированных команд, хранимых в специально отведенной памяти графического редактора.

Принцип построения дисплейного списка прост:

1. Генерируется номер дисплейного списка.

2. Этот номер сохраняется для дальнейшей работы с данным списком.

3. Единожды визуализируется графическая сцена в контейнере между вызовами функций *glNewList()* и *glEndList()*.

4. В функции, отвечающей за визуализацию кадра, вызывается необходимый дисплейный список с помощью команды *glCallList*.

Основные операции, дающие положительный эффект от использования списков отображения (помимо прироста производительности от кеширования в списке самой геометрии):

• Матричные операции (рассчитываемые матрицы могут быть сохранены в списке отображения *OpenGL*).

• Операции с текстурами (при использовании *OpenGL* версии 1.1; в остальных случаях лучше использовать текстурные объекты).

• Отрисовка битовых карт и изображений (формат хранения этих данных в приложении часто не идеален для аппаратуры видеоадаптера в отличие от формата дисплейного списка).

• Операции с источниками света, материалами и их свойствами (в дисплейных списках здесь также обеспечивается прирост производительности визуализации *OpenGL*, так как настройка материала сама по себе может стать довольно медленной операцией в связи с дополнительными вычислениями; в случае использования дисплейного списка хранятся кешированные значения этих вычислений).

Оптимизация приложения

Теперь перейдем к оптимизации функций визуализации нашего графического *2D*-редактора.

Сначала в класс *anLayer* добавим новые функции для управления созданием и удалением дисплейных списков.

```
// функции удаления слоя
public void ClearList()
{
// проверяем факт существования дисплейного списка с номером, хранимым
в ListNom
if ( Gl.glIsList(ListNom) == Gl.GL_TRUE)
{
// удаляем его в случае существования
Gl.glDeleteLists(ListNom,1);
}
}
public void CreateNewList()
{
// проверяем факт существования дисплейного списка с номером, хранимым
в ListNom
if ( Gl.glIsList(ListNom) == Gl.GL_TRUE)
{
// удаляем его в случае существования
```

```
Gl.glDeleteLists(ListNom,1);
// и генерируем новый номер
ListNom = Gl.glGenLists(1);
}
// создаем дисплейный список
Gl.glNewList(ListNom, Gl.GL_COMPILE);
// вызывая обычную визуализацию (не из списка)
RenderImage( false );
// завершаем создание дисплейного списка
Gl.glEndList();
}
```

Как видно из кода, при создании дисплейного списка для данного слоя вызывается отрисовка функции *RenderImage* с дополнительным параметром *false* – этот параметр указывает на то, что необходимо полностью провести визуализацию, а не использовать дисплейный список для данного слоя.

Теперь рассмотрим изменения функции *public void RenderImage()*.

Раньше мы отрисовывали геометрию, перебирая все точки в массиве и выводя каждую точку:

```
if (DrawPlace[ax, bx, 3] != 1)
{
   // устанавливаем заданный в ней цвет int col1 = DrawPlace[ax, bx, 0];
   int col2 = DrawPlace[ax, bx, 1];
   int col3 = DrawPlace[ax, bx, 2];
   Gl.glColor3f(( float )DrawPlace[ax, bx, 0] / 255.0f, ( float )DrawPlace[ax, bx, 1] /
   255.0f, ( float )DrawPlace[ax, bx, 2] / 255.0f);
   // и выводим ее на экран
   Gl.glVertex2i(ax, bx);
}
```

Теперь используем отрисовку сразу массивов вершин и цветов целиком. При этом выполняется отрисовка дисплейных списков (или отрисовка напрямую, но с использованием массива вершин).

Это намного экономичнее, чем сообщать при отрисовке о каждой вершине.

Нам нужно только правильно указать формат, в котором представлены данные, и создать сами массивы данных.

Обновленный код для визуализации слоя (с комментариями):

```
// функция визуализации слоя
public void RenderImage( bool FromList)
{
if (FromList)
// указана визуализация из дисплейного списка, следовательно данный слой не
активен
{
// вызываем дисплейный список
Gl.glCallList(ListNom);
}
else // данный слой активен и визуализацию необходимо делать на ходу
{
// счетчик номеров элементов, которые должны участвовать в визуализации
int count = 0;
// проходим по всем точкам рисунка
for ( int ax = 0; ax < Width; ax++)
{
for ( int bx = 0; bx < Heigth; bx++)
{
// если точка в координатах ax,bx не помечена флагом "прозрачная"
if (DrawPlace[ax, bx, 3] != 1)
{
// не лучший способ, но так мы подсчитаем количество действительно значимых
точек слоя,
// которые должны быть визуализированы
count++;
}
}
}
// данный массив будет заполнен, а затем передан для быстрой отрисовки гео-
метрии (в нашем случае - точек)
// колич. точек * 2 (для хранения координат x и y каждой точки, которая будет
отрисована)
int [] arr_date_vertex = new int [count * 2];
// данный массив будет содержать значения цветов для всех отрисовываемых точек
// колич. точек * 3 (для хранения координат R G B значений цветов каждой точ-
ки, которая будет отрисована)
float [] arr_date_colors = new float [count * 3];
// счетчик элементов для создания массивов, которые будут переданы
```

```
// в реализацию OpenGL с помощью функции glDrawArrays
int now_element = 0;
// теперь, когда мы выделили массив необходимого размера,
// заполним его необходимыми значениями
for ( int ax = 0; ax < Width; ax++)
{
for ( int bx = 0; bx < Heigth; bx++)
{
// если точка с координатами ax, bx не помечена флагом "прозрачная"
// если данная точка не помечена флагом, сигнализирующим о том, что она не
должна быть визуализирована
if (DrawPlace[ax, bx, 3] != 1)
{
// заносим координаты точки (ax , bx ) в массив, который передан для визуализа-
ции
arr_date_vertex[now_element * 2] = ax;
arr_date_vertex[now_element * 2 + 1] = bx;
// заносим значения составляющих цвета, сразу перенося их в формат float
arr_date_colors[now_element * 3] = ( float )DrawPlace[ax, bx, 0] / 255.0f;
arr_date_colors[now_element * 3 + 1] = ( float )DrawPlace[ax, bx, 1] / 255.0f;
arr_date_colors[now_element * 3 + 2] = ( float )DrawPlace[ax, bx, 2] / 255.0f;
// подсчет добавленных элементов в массивы
now_element++;
}
}
}
// теперь, когда массивы с геометрическими данными и данными о цветах подго-
товлены,
// включаем функцию использования массивов вершин и цветов
Gl.glEnableClientState( Gl.GL_VERTEX_ARRAY);
Gl.glEnableClientState( Gl.GL_COLOR_ARRAY);
// передаем массивы вершин и цветов, указывая количество элементов массива,
приходящихся
// на один визуализируемый элемент (в случае точек - 2 координаты: x и y, в слу-
чае цветов - 3 составляющие цвета)
Gl.glColorPointer(3, Gl.GL_FLOAT, 0, arr_date_colors);
Gl.glVertexPointer(2, Gl.GL_INT, 0, arr_date_vertex);
// вызываем функцию glDrawArrays, которая позволит нам визуализировать
наши массивы, передав их целиком,
// а не передавая в цикле каждую точку
Gl.glDrawArrays( Gl.GL_POINTS, 0, count);
```

```
// деактивируем режим использования массивов геометрии и цветов
Gl.glDisableClientState( Gl.GL_VERTEX_ARRAY);
Gl.glDisableClientState( Gl.GL_COLOR_ARRAY);
} }
```

Далее перейдем к классу *class anEngine*. Внесём изменения в функции установки активного слоя:

```
// функция для установки номера активного слоя
public void SetActiveLayerNom( int nom)
{
// текущий слой больше не будет активным, следовательно надо создать новый
дисплейный список для его быстрой визуализации
((anLayer)Layers[ActiveLayerNom]).CreateNewList(); // новый активный слой по-
лучает установленный активный цвет для предыдущего активного слоя
((anLayer)Layers[nom]).SetColor( ((anLayer)Layers[ActiveLayerNom]).GetColor() );
// установка номера активного слоя
ActiveLayerNom = nom;
}
```

Функция *SwapImage* теперь отрисовывает дисплейные списки для всех слоев (параметр *true* в функции *image*), которые не являются активными. Активный слой отрисовывается напрямую, так как он постоянно меняется.

```
// визуализация
public void SwapImage()
{
// вызываем функцию визуализации в нашем слое
for( int ax = 0; ax < Layers.Count; ax++)
{
// если этот слой является активным в данный момент
if( ax == ActiveLayerNom)
{
// вызываем визуализацию данного слоя напрямую
((anLayer)Layers[ax]).RenderImage( false );
}
else
{
// вызываем визуализацию слоя из дисплейного списка
((anLayer)Layers[ax]).RenderImage( true );
} } }
```

Созданный учебный редактор демонстрирует ряд важных методов работы с растровыми данными – развертывание классов для управления изображениями, отрисовкой, работа с дисплейными списками и наборами вершин и др.

Практическое задание

1. Ознакомиться с изложенным выше и представленным в литературе теоретическим материалом.

2. Выполнить действия, приведенные в пп. 1.1, 1.2, 1.3, 1.4, 1.5. При разработке программы имя проекта, создаваемого в *MS Visual Studio*, должно содержать фамилию студента и группу (например, *Ivanov_Ivan_ISG_105_lab_1*).

3. Добавить в программу кисть (возможно большого размера), отрисовывающую на плоскости ваши инициалы и номер группы. Добавить кнопку на панель программы для использования данной кисти. Добавить пункт в меню для активации данной кисти. Разработанная программа должна соответствовать п. 1.5.

Продемонстрировать работу программы на скриншотах (включая реализованные методы).

Скриншот должен включать заголовок окна консоли с полным путем к имени пользователя и папки с проектом.

Папка с программой должна содержать весь проект, выполненный в *MS Visual Studio 2008*, и исполняемые файлы.

Контрольные вопросы

1. Принципы организации программы – растрового редактора.
2. Инструменты и функции программы – растрового редактора.

Тема 2. АЛГОРИТМЫ ОБРАБОТКИ РАСТРОВЫХ ИЗОБРАЖЕНИЙ

Цель изучения темы. Освоение методов преобразования растровых изображений, приводящих к их визуальному изменению с целью улучшения и получения различных специальных эффектов.

2.1. Введение в алгоритмы обработки растровых изображений

Общие сведения

Ценность результатов решения задачи компьютерной обработки изображений зависит от того, что можно сделать с изображением, как только оно оказалось в компьютере. Существует много полезных манипуляций с фотографиями, введенными цифровым способом. Если снимок сделан с передержкой, выдержку можно сократить, уменьшив цветовые значения пикселей. При необходимости красную, зеленую и синюю компоненты можно изменять раздельно, чтобы получить наилучший цветовой баланс. В расплывчатых изображениях можно увеличить резкость, и наоборот, четкие, контрастные изображения можно размыть, имитируя эффект смягчающих фотофильтров.

Рассмотрим четыре широко используемых в компьютерной графике эффекта, которые дает обработка изображений: размывание, увеличение резкости, тиснение и акварельный эффект. При размывании перераспределяются цвета в изображении и смягчаются резкие границы, в то время как при увеличении резкости подчеркиваются различия между цветами смежных пикселей и выделяются незаметные детали. Тиснение преобразует изображение так, что объекты сцены выглядят выдавленными на металлической поверхности подобно чеканке на монетах. Акварельный эффект превращает фотографическое изображение в картинку, как будто бы написанную акварелью.

Матрица – ядро свертки

С алгоритмической точки зрения получение этих эффектов не представляет сложности. Каждый из них достигается применением матрицы чисел, которую называют ядром свертки. Матрица размера-

ми 3×3 содержит три строки по три числа в каждой. Чтобы преобразовать один пиксель в изображении, значение его цвета умножается на число в центре ядра. Затем восемь значений цветов пикселей, окружающих центральный пиксель, умножаются на соответствующие им коэффициенты ядра, все девять значений суммируются, и в результате получается новое значение цвета центрального пикселя. Этот процесс повторяется для каждого пикселя в изображении; тем самым изображение фильтруется. Коэффициенты ядра определяют результат процесса фильтрации. Ядро размывания, например, состоит из совокупности коэффициентов, каждый из которых меньше 1, а их сумма составляет 1. Это означает, что каждый пиксель поглотит что-то из цветов соседей, но полная яркость изображения останется неизменной (если сумма коэффициентов больше чем 1, яркость увеличится; если меньше чем 1, яркость уменьшится.) В ядре резкости центральный коэффициент больше 1, а окружен он отрицательными числами, сумма которых на единицу меньше центрального коэффициента. Таким образом, увеличивается любой существующий контраст между цветом пикселя и цветами его соседей. Изменяя определенным способом размер и состав ядра свертки, можно получить и другие полезные эффекты.

Алгоритм размывания

При подготовке к размыванию цифровое изображение считывается в память компьютера в виде красной, зеленой и синей компонент цвета каждого пикселя. Ядро размывания R размерами 3×3 применяется к красной, зеленой и синей компонентам цвета каждого пикселя в изображении:

$$R = \begin{vmatrix} 0,05 & 0,05 & 0,05 \\ 0,05 & 0,60 & 0,05 \\ 0,05 & 0,05 & 0,05 \end{vmatrix}.$$

Значение цвета пикселя, который находится под центром ядра, вычисляется умножением весовых коэффициентов ядра на соответствующие значения цвета в изображении и суммированием результатов.

Итоговое изображение получается размытым по сравнению с оригиналом потому, что цвет каждого пикселя распространился среди соседей. Степень размывания можно увеличить тремя способами: используя ядро большего размера, чтобы распределить цвета среди

большего числа соседей; подбирая коэффициенты ядра с целью уменьшения влияния центрального коэффициента; фильтруя изображение многократно с ядром размывания.

Алгоритм увеличения резкости

Увеличение резкости достигается точно так же, как и размывание, за исключением того, что используется другое ядро, так как цель преобразования – увеличить, а не уменьшить четкость изображения.

При обработке каждого пикселя в изображении используется ядро размывания G размерами 3×3

$$G = \begin{vmatrix} -0,1 & -0,1 & -0,1 \\ -0,1 & 1,8 & -0,1 \\ -0,1 & -0,1 & -0,1 \end{vmatrix}.$$

Как и прежде, красная, зеленая и синяя цветовые составляющие обрабатываются отдельно и позже объединяются, чтобы сформировать 24-битное значение цвета. Отрицательные веса вокруг центра ядра увеличивают контраст между центральным пикселем и соседями.

Конечное изображение получается более четким, чем оригинал. Дополнительные детали не возникли из ничего; процесс увеличения резкости просто повысил существующий контраст между пикселями. При повторной обработке изображения четкость может увеличиться еще больше.

Алгоритм тиснения

Тиснение выполняется почти так же, как размывание и увеличение резкости. Каждый пиксель обычного цветного изображения обрабатывается ядром тиснения T размерами 3 × 3:

$$T_1 = \begin{vmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{vmatrix}; \quad T_2 = \begin{vmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{vmatrix}; \quad T_3 = \begin{vmatrix} -0 & -1 & 0 \\ -1 & 4 & -1 \\ -0 & -1 & 0 \end{vmatrix}.$$

В отличие от ядер размывания и резкости, в которых сумма коэффициентов равна 1, сумма весов в ядре тиснения равна 0. Это означает, что фоновым пикселям (тем, которые не находятся на границах перехода от одного цвета к другому) присваиваются нулевые значения, а нефоновым – значения, отличные от нуля. То есть, если все девять пикселей, находящихся в области матрицы тиснения, имеют одинаковые визуальные свойства, то значение центрального пикселя после преобразования станет нулевым (черный цвет).

После того, как значение пикселя обработано ядром тиснения, к нему прибавляется 128. Таким образом, значением фоновых пикселей станет средний серый цвет (красный = 128, зеленый = 128, синий = 128). Суммы, превышающие 255, можно округлить до 255 или взять остаток по модулю 255, чтобы значение оказалось между 0 и 255.

В тисненном варианте изображения контуры кажутся выдавленными над поверхностью. Направление подсветки изображения можно изменять, меняя позиции 1 и –1 в ядре. Если, например, поменять местами значения 1 и –1, то реверсируется направление подсветки:

$$T_4 = \begin{vmatrix} 0 & 1 & 2 \\ 0 & 0 & 0 \\ 0 & -1 & -2 \end{vmatrix}; \quad T_5 = \begin{vmatrix} 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & -1 \end{vmatrix}; \quad T_6 = \begin{vmatrix} 2 & -1 & 0 \\ -1 & 0 & -1 \\ -0 & -1 & 2 \end{vmatrix}.$$

Алгоритм акварелизации

Акварельный фильтр преобразует исходное изображение так, что после обработки оно выглядит выполненным акварелью.

Первый шаг в применении акварельного фильтра – сглаживание цветов в изображении. Один из способов сглаживания – процесс медианного осреднения цвета в каждой точке. Значения цвета каждого пикселя и его 24 соседей помещаются в список и сортируются от меньшего к большему. Медианное (тринадцатое) значение цвета в списке присваивается центральному пикселю. Возможно также применение медианного фильтра M, выполняющего ту же операцию:

$$M = \frac{1}{16}\begin{vmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{vmatrix}.$$

Второй шаг после сглаживания цветов – обработка каждого пикселя в изображении ядром резкости R для выделения границы переходов цветов:

$$R = \begin{vmatrix} -0{,}5 & -0{,}5 & -0{,}5 \\ -0{,}5 & 5 & -0{,}5 \\ -0{,}5 & -0{,}5 & -0{,}5 \end{vmatrix}.$$

Результирующее изображение напоминает акварельную живопись. Этот пример показывает также, что можно объединять различные методы обработки изображений и добиваться новых визуальных эффектов.

2.2. Реализация фильтров

Реализуем фильтры в уже разработанном ранее приложении, демонстрирующем основы создания графического редактора.

Добавим новый пункт меню, в котором перечислены показанные на рис. 2.1 фильтры:

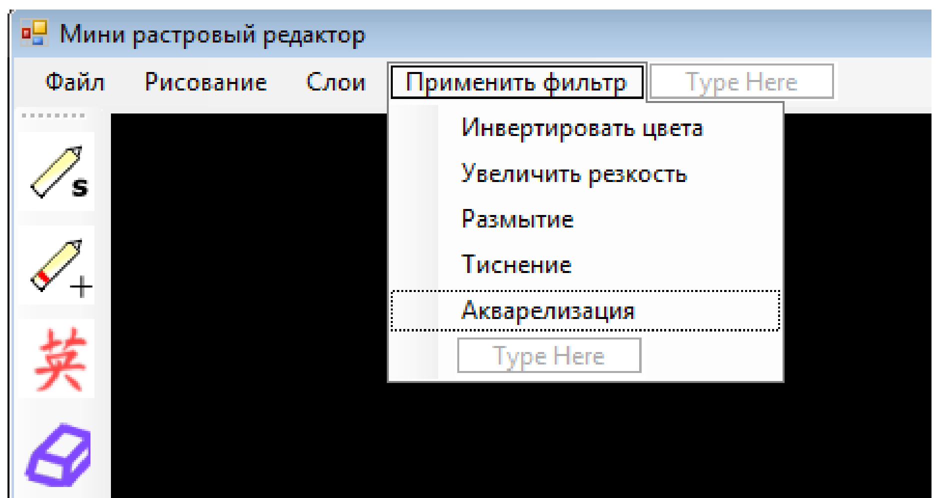


Рис. 2.1

Каждому пункту меню добавим свой обработчик. Коды этих обработчиков:

```
private void инвертироватьЦветаToolStripMenuItem_Click( object sender, EventArgs
e)
{
ProgrammDrawingEngine.Filter_0();
}
private void увеличитьРезкостьToolStripMenuItem_Click( object sender, EventArgs
e)
{
ProgrammDrawingEngine.Filter_1();
}
private void размытиеToolStripMenuItem_Click_1( object sender, EventArgs e)
{
ProgrammDrawingEngine.Filter_2();
}
private void тиснениеToolStripMenuItem_Click( object sender, EventArgs e)
{
ProgrammDrawingEngine.Filter_3();
}
```

```
private void акварелизацияToolStripMenuItem_Click( object sender, EventArgs e)
{
ProgrammDrawingEngine.Filter_4();
}
```

Как видно из кода, мы обращаемся к классу *ProgrammDrawing Engine*, а именно к функциям, реализованным в нем – *filter_0*, *filter_1* и т.д.

Инвертирование

Это самый простой фильтр. Код вызывает функцию *Inverse* из класса слоев для данного класса, в результате чего цвета в слое инвертированы.

```
// фильтр для инвертирования цветов
public void Filter_0()
{
// вызываем функцию инвертирования класса anLayer
((anLayer)Layers[ActiveLayerNom]).Invers();
}
```

Реализация функции *Inverse*:

```
// инвертирование цветов
public void Invers()
{
// циклами переберем все пиксели изображения
for ( int Y = 0; Y < Heigth; Y++)
{
for ( int X = 0; X < Width; X++)
{
// и инвертируем цвет, установленный в RGB составляющих, на обратный (255-
R) (255-G) (255-B)
DrawPlace[X, Y, 0] = 255-DrawPlace[X, Y, 0];
DrawPlace[X, Y, 1] = 255-DrawPlace[X, Y, 1];
DrawPlace[X, Y, 2] = 255-DrawPlace[X, Y, 2];
} } }
```

Следующие фильтры немного сложнее в реализации.

В классе движка указываем матрицу для обработки изображения, после чего вызываем функцию осуществления преобразования на основе матрицы и дополнительных параметров, которые мы рассмотрим позднее (при рассмотрении работы самой функции).

Реализация фильтров

```
public void Filter_1()
{
// собираем матрицу
float [] mat = new float [9]; mat[0] = -0.1f;
mat[1] = -0.1f;
mat[2] = -0.1f;
mat[3] = -0.1f;
mat[4] = 1.8f;
mat[5] = -0.1f;
mat[6] = -0.1f;
mat[7] = -0.1f;
mat[8] = -0.1f;
//вызываем функцию обработки, передавая туда матрицу и дополнительные па-
раметры
((anLayer)Layers[ActiveLayerNom]).PixelTransformation(mat, 0, 1, false );
}
public void Filter_2()
{
// собираем матрицу
float [] mat = new float [9];
mat[0] = 0.05f;
mat[1] = 0.05f;
mat[2] = 0.05f;
mat[3] = 0.05f;
mat[4] = 0.6f;
mat[5] = 0.05f;
mat[6] = 0.05f;
mat[7] = 0.05f;
mat[8] = 0.05f;
//вызываем функцию обработки, передавая туда матрицу и дополнительные па-
раметры
((anLayer)Layers[ActiveLayerNom]).PixelTransformation(mat, 0, 1, false );
}
public void Filter_3()
{
// собираем матрицу
float [] mat = new float [9];
mat[0] = -1.0f;
mat[1] = -1.0f;
mat[2] = -1.0f;
mat[3] = -1.0f;
```

```
mat[4] = 8.0f;
mat[5] = -1.0f;
mat[6] = -1.0f;
mat[7] = -1.0f;
mat[8] = -1.0f;
//вызываем функцию обработки, передавая туда матрицу и дополнительные параметры
((anLayer)Layers[ActiveLayerNom]).PixelTransformation(mat, 0, 2, true );
}
public void Filter_4()
{
// собираем матрицу
// для данного фильтра нам необходимо произвести два преобразования
float [] mat = new float [9];
mat[0] = 0.50f;
mat[1] = 1.0f;
mat[2] = 0.50f;
mat[3] = 1.0f;
mat[4] = 2.0f;
mat[5] = 1.0f;
mat[6] = 0.50f;
mat[7] = 1.0f;
mat[8] = 0.50f;
//вызываем функцию обработки, передавая туда матрицу и дополнительные параметры
((anLayer)Layers[ActiveLayerNom]).PixelTransformation(mat, 0, 2, true );
mat[0] = -0.5f;
mat[1] = -0.5f;
mat[2] = -0.5f;
mat[3] = -0.5f;
mat[4] = 6.0f;
mat[5] = -0.5f;
mat[6] = -0.5f;
mat[7] = -0.5f;
mat[8] = -0.5f;
//вызываем функцию обработки, передавая туда матрицу и дополнительные параметры
((anLayer)Layers[ActiveLayerNom]).PixelTransformation(mat, 0, 1, false );
}
```

Теперь нам осталось рассмотреть работу функции *PixelTransformation* и работа с фильтрами завершена.

Данная функция проводит все необходимые преобразования (см. комментарии):

```
// функция обработки слоя изображения на основе полученной матрицы и до-
полнительных параметров
// corr - коррекция составляющей цвета - после обработки каждого пикселя к
каждой его составляющей будет
// прибавлено данное значение
// COEFF - коэффициент, реализующий усиление работы фильтра
// need_count_correction - необходимость корректировки значения полученного
пикселя после прохода фильтра
// если данный параметр установлен, то каждая составляющая цвета, перед тем
как быть приведенной к виду 0-255
// будет разделена на количество произошедших с ней преобразований
// это необходимо для корректной работы некоторых фильтров
public void PixelTransformation( float [] mat, int corr, float COEFF, bool
need_count_correction)
{
// массив для получения результирующего пикселя
float [] resault_RGB = new float [3];
int count = 0;
// проходим циклом по всем пикселям слоя
for ( int Y = 0; Y < Heigth; Y++)
{
for ( int X = 0; X < Width; X++)
{
// цикл по всем составляющим (0-2, т.е. R G B)
for ( int c = 0, ax = 0, bx = 0; c < 3; c++)
{
// обнуление составляющей результата
resault_RGB[c] = 0;
// обнуление счетчика обработок
count = 0;
// два цикла для захвата области 3 × 3 вокруг обрабатываемого пикселя
for (bx = -1; bx < 2; bx++)
{
for (ax = -1; ax < 2; ax++)
{
// если мы не попали в рамки, просто используем центральный пиксель, и про-
должаем цикл
if (X + ax < 0 || X + ax > Width-1 || Y + bx < 0 || Y + bx > Heigth-1)
{
// считаем составляющую в одной из точек, используем коэффициент в матрице
(под номером текущей итерации), коэффициент усиления (COEFF) и прибавляем
коррекцию (corr)
```

```
resault_RGB[c] += ( float )(DrawPlace[X, Y, c]) * mat[count] * COEFF + corr;
// счетчик обработок = ячейке матрицы с необходимым коэффициентом
count++;
// продолжаем цикл
continue;
}
// иначе, если мы укладываемся в изображение (не пересекаем границы), используем соседние пиксели, корректируем ячейку массива параметрами ax, bx
resault_RGB[c] += ( float )(DrawPlace[X + ax, Y + bx, c]) * mat[count] * COEFF + corr;
// счетчик обработок = ячейке матрицы с необходимым коэффициентом
count++;
} } }
// теперь для всех составляющих корректируем цвет
for ( int c = 0; c < 3; c++)
{
// если требуется разделить результат до приведения к 0-255, разделив на количество проведенных операций
if( count != 0 && need_count_correction)
{
// выполняем данное деление
resault_RGB[c] /= count;
}
// если значение меньше нуля
if (resault_RGB[c] < 0)
{
// - приравниваем к нулю
resault_RGB[c] = 0;
}
// если больше 255
if (resault_RGB[c] > 255)
{
// приравниваем к 255
resault_RGB[c] = 255;
}
// записываем в массив цветов слоя новое значение
DrawPlace[X, Y, c] = ( int )resault_RGB[c];
} } } }
```

Далее следует изучить работу готовой программы, так как необходимо видеть, что было "до" и что стало "после". Размытие можно применить несколько раз, так как оно очень плавно меняет изображе-

ние. Потом можно несколько раз применить фильтр увеличения резкости и посмотреть, что получится (рис. 2.2).

Рис. 2.2.

Практическое задание

1. Ознакомиться с изложенным выше и представленным в литературе теоретическим материалом.

2. Выполнить действия, приведенные в п. 2.2, доработав программу из п. 1.5. При разработке программы имя проекта, создаваемого в *MS Visual Studio*, должно содержать фамилию студента и группу (например, *Ivanov_Ivan_ISG_105_lab_1*).

3. Продемонстрировать работу программы на скриншотах (включая реализованные методы).

Скриншот должен включать заголовок окна консоли с полным путем к имени пользователя и папки с проектом.

Папка с программой должна содержать весь проект, выполненный в *MS Visual Studio 2008*, исполняемые файлы.

Контрольные вопросы

1. Алгоритмы обработки растровых изображений.
2. Фильтры для обработки растровых изображений.

Тема 3. СПЛАЙНЫ

Цель изучения темы. Ознакомление с алгоритмическими основами применения сплайнов в компьютерной графике, освоение методов формирования изображений на основе сплайнов.

3.1. Сплайны в компьютерной графике

Общие сведения

Одна из типовых задач компьютерной графики – построение гладкой кривой или поверхности по набору заданных точек. Для кривой на плоскости при использовании кубических В-сплайнов из заданной последовательности точек выбираются две соседние точки, и между ними строится кривая кубического полинома на основе позиции четырех точек – двух выбранных и двух соседних с ними. В-сплайны обеспечивают получение более гладких кривых, чем другие способы сглаживания за счет того, что получаемые кривые не проходят точно через заданные точки.

Параметрическое представление сплайнов

При практическом использовании сложных кривых в компьютерной графике обычно используется их параметрическое представление. В этом случае любая точка на части кривой, лежащая между двумя заданными последовательными точками P_i и P_{i+1}, будет иметь координаты $(x(t), y(t))$, где значение t меняется от 0,0 до 1,0, если строится часть кривой от точки P_i до точки P_{i+1} (рис. 3.1.). Параметр t может интерпретироваться как нормированное время построения кривой между двумя соседними точками.

Если есть заданные точки $P_0(x_0, y_0)$, $P1(x_1, y_1)$, ... $P_n(x_n, y_n)$, то часть кривой В-сплайна между точками P_i и P_{i+1} определяется последовательностью точек с координатами $(x(t), y(t))$, при $0{,}0 \le t \le 1{,}0$:

$x(t) = a_0 + a_1 t + a_2 t^2 + a_3 t^3;$

$y(t) = b_0 + b_1 t + b_2 t^2 + b_3 t^3.$

Эти уравнения содержат коэффициенты:

$a_0 = (x_{i-1} + 4x_i + x_{i+1})/6;$

$a_1 = (-x_{i-1} + x_{i+1})/2;$

$a_2 = (x_{i-1} - 2x_i + x_{i+1})/2;$

$a_3 = (-x_{i-1} + 3x_i - 3x_{i+1} + x_{i+2})/6.$

Коэффициенты b_0, ... b_3 вычисляются аналогично по значениям y_{i-1}, ... y_{i+2}. Для производительности алгоритма важно, что a_i и b_i вычисляются только один раз для каждого сегмента кривой.

Свойства сплайнов

Математически гладкость кривых выражается в терминах непрерывности параметрических представлений $x(t)$ и $y(t)$ и их производных. Кривые типа B-сплайна обладают свойством непрерывности даже вторых производных $x''(t)$ и $y''(t)$ в точках стыковки двух соседних сегментов кривой.

Для определения свойств кривой в точках стыковки двух сегментов рассмотрим функцию $x(t)$ и ее первую и вторую производные для значений $t = 0$ и $t = 1$. Функция $y(t)$ будет обладать аналогичными свойствами.

$x(0) = a_0 = (x_{i-1} + 4x_i + x_{i+1})/6;$

$x(1) = a_1 + a_2 + a_3 + a_2 = (x_i + 4x_{i+1} + x_{i+2})/6.$

То есть значение $x(0)$ не равно в точности x-координате x_i точки P_i: оно зависит от позиций точек P_{i-1} и P_{i+1}.

Для трех последовательных точек на кривой A, B и C (рис. 3.2) точка B принадлежит сегменту AB и одновременно сегменту BC. Для первого сегмента $A=P_i$, $B=P_{i+1}$, $C=P_{i+2}$. Тогда $\tilde{x}_B = x(1) = (x_A + 4x_B + x_C)/6$, где $\tilde{x}_B$ – вычисленное значение координаты x для точки B.

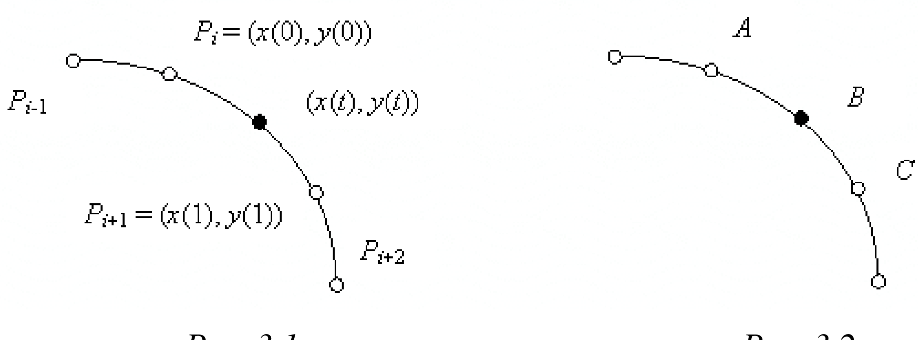


Рис. 3.1. *Рис. 3.2.*

Для второго сегмента $A = P_{i-1}$, $B = P_i$, $C = P_{i+1}$ и $\tilde{x}_B = x(0) = (x_A + 4x_B + x_C)/6$. Одинаковые результаты означают непрерывность функции $x(t)$ в точке B. Продифференцировав $x(t)$ дважды, найдем $x'(t)$ и $x''(t)$:

$x'(t) = a_1 + 2a_2t + 3a_3t^2;$ $x''(t) = 2a_2 + 6a_3t.$

Подставляя в них значения $t = 0$ и $t = 1$, находим

$x'(0) = a_1 = (-x_{i-1} + x_{i+1})/2;\quad x'(1) = a_1 + 2a_2 + 3a_3 = (-x_i + x_{i+2})/2;$

$x''(0) = 2a_2 = x_{i-1} - 2x_i + x_{i+1};\quad x''(1) = 2a_2 + 6a_3 = x_i - 2x_{i+1} + x_{i+2}.$

Тогда значение первой производной в точке B, вычисленное для первого сегмента кривой, $(\tilde{x}')_B = x'(1) = (-x_A + x_C)/\,2$ совпадает со значением первой производной в той же точке, вычисленное для следующего сегмента $(\tilde{x}')_B = x'(0)$.

Точно так же и значение второй производной в точке B, вычисленное для первого сегмента кривой, $(\tilde{x}'')_B = x''(1) = x_A - 2x_B + x_C$ совпадает со значением второй производной в той же точке, вычисленное для следующего сегмента $(\tilde{x}'')_B = x''(0)$.

Таким образом, не только сама кривая, но и ее первые две производные непрерывны в каждой точке кривой, то есть вся кривая является очень гладкой.

Для расчета любого сегмента кривой между точками P_i и P_{i+1} используются также точки P_{i-1} и P_{i+2}. Тогда, если есть точки P_0, P_1, … P_{n-1} и P_n, то первый сегмент кривой будет располагаться между точками P_1 и P_2, а последний – между точками P_{n-2} и P_{n-1}. То есть начальной и конечной точками кривой будут P_1 и P_{n-1}, а не P_0 и P_n. Поэтому для замкнутой кривой следует три точки в начале или в конце последовательности задавать дважды.

3.2. Построение В-сплайна

Разработка программы начинается с создания оболочки.

Создайте окно программы и разместите на нем элемент *openglsimplecontrol* как показано на рис. 3.3, после чего установите его размеры 500 × 500. Переименуйте данный объект, дав ему имя *AnT*.

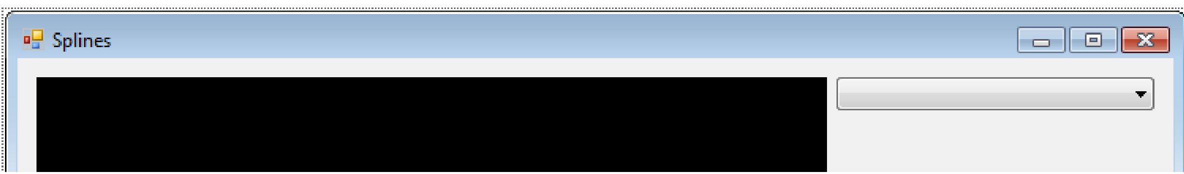


Рис. 3.3

Справа от данного элемента разместите элемент *comboBox*, после чего в его свойствах установите значение параметра *DropDownStyle* = = *DropDownList*. Тогда выпадающие элементы перестанут быть до-

ступными для редактирования. После этого измените элементы *Items* как показано на рис. 3.4.

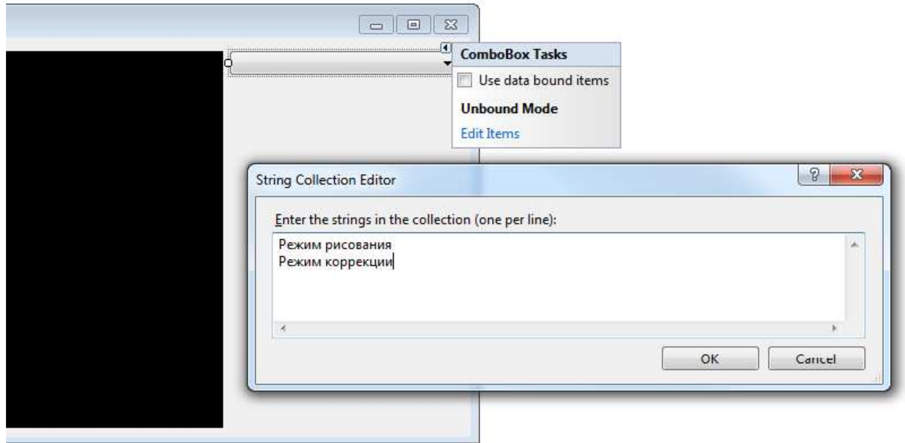
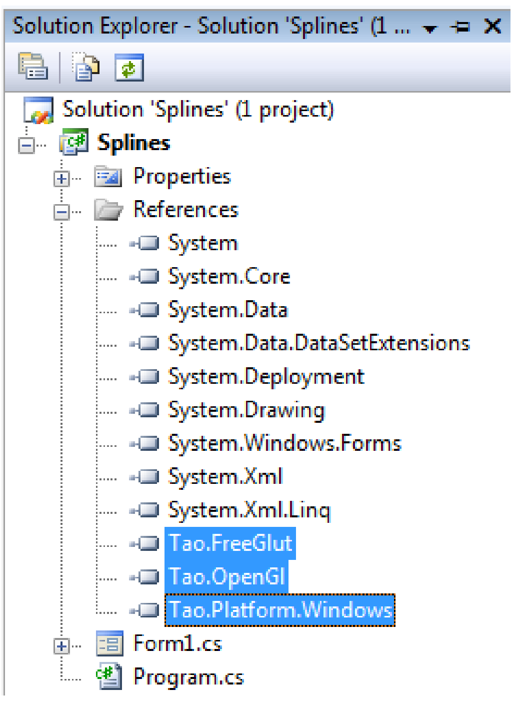


Рис. 3.4

Не забудьте установить ссылки на используемые библиотеки *Tao* (рис. 3.5).

Инициализация окна и *OpenGl* происходит так же, как и в предыдущих проектах, код приведен далее.

Для обработки событий мыши выделите объект *AnT*, перейдите к его свойствам, после чего перейдите к настройке событий и добавьте обработку событий мыши, как показано на рис. 3.6.



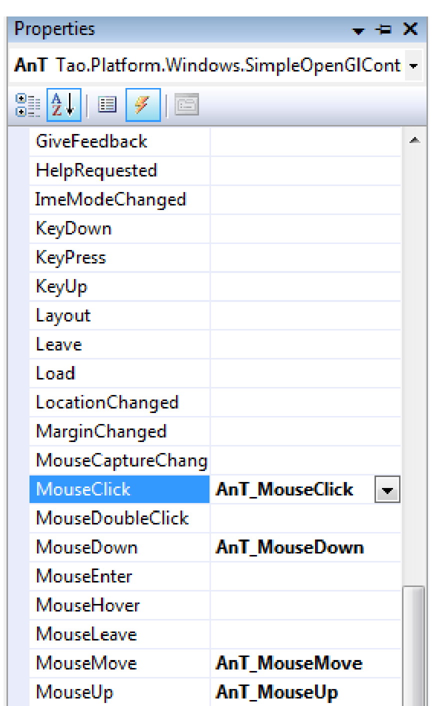


Рис. 3.5 *Рис. 3.6*

Для реализации визуализации используется таймер – после инициализации окна он генерирует событие, называемое "тиком" таймера. Раз в 30 миллисекунд добавьте элемент таймер, переименуйте экземпляр в *RenderTimer* и установите время "тика" 30 миллисекунд (как показано на рис. 3.7), а также добавьте ему событие для обработки "тика".

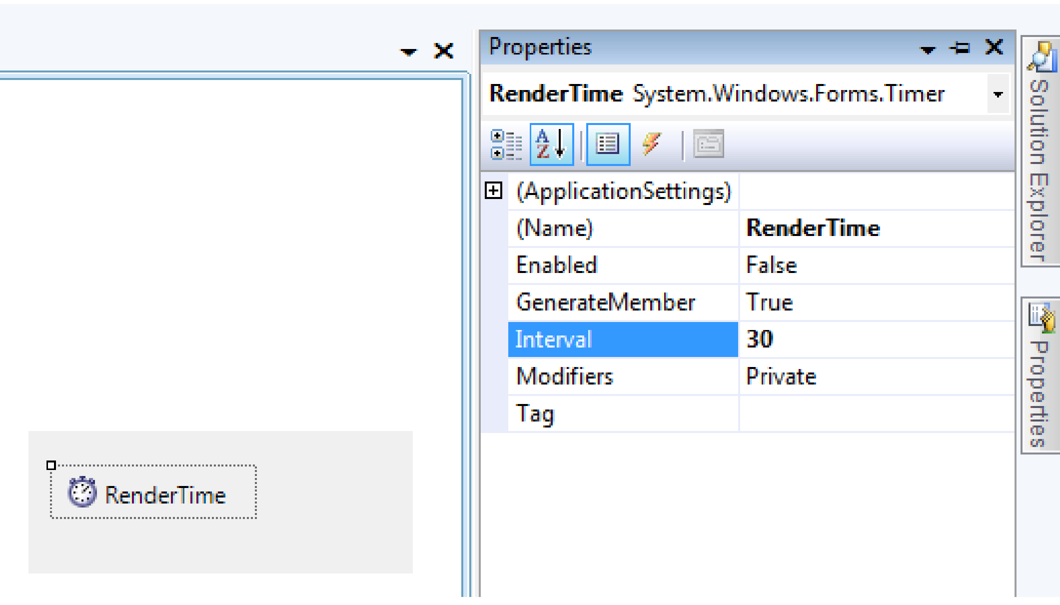


Рис. 3.7

Нам потребуется объявить ряд переменных для дальнейшей работы программы:

```
// массив, в который заносятся управляющие точки
private float [,] DrawingArray = new float [64, 2]; // количество точек
private int count_points = 0;
// 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;
// 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;
/*
```

```
* Состояние захвата вершины мышью (при редактировании)
*/
// -1 означает, что нет захваченной вершины,
// иначе номер указывает на элемент массива, хранящий захваченную вершину
int captured = -1;
```

Код инициализации окна, настройки визуализации в *OpenGL* и старт таймера после завершения начальной настройки:

```
public Form1()
{
InitializeComponent();
AnT.InitializeContexts();
}
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 сто-
рон элемента AnT.
if (( float )AnT.Width <= ( float )AnT.Height)
{
ScreenW = 500.0;
ScreenH = 500.0 * ( float )AnT.Height / ( float )AnT.Width;
Glu.gluOrtho2D(0.0, ScreenW, 0.0, ScreenH);
}
else
{
ScreenW = 500.0 * ( float )AnT.Width / ( float )AnT.Height;
ScreenH = 500.0;
Glu.gluOrtho2D(0.0, 500.0 * ( float )AnT.Width / ( float )AnT.Height, 0.0, 500.0);
}
// 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);
RenderTimer.Start();
comboBox1.SelectedIndex = 0;
}
```

Функция визуализации текста уже была нами рассмотрена:

```
// 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_8_BY_13, char_for_draw);
} }
```

Теперь переходим непосредственно к обработке нажатий мыши, таймера и построению сплайна.

Обработка события таймера:

```
private void RenderTime_Tick( object sender, EventArgs e)
{
// обработка "тика" таймера - вызов функции отрисовки
Draw();
}
```

Функция отрисовки:

```
// функция отрисовки, вызываемая событием таймера
private void Draw()
{
// количество сегментов при расчете сплайна
int N = 30; // вспомогательные переменные для расчета сплайна
double X, Y;
// n = count_points+1 означает, что мы берем все созданные контрольные
// точки + ту, которая следует за мышью, для создания интерактивности прило-
жения
```

```
int eps = 4, i, j, n = count_points+1, first;
double xA, xB, xC, xD, yA, yB, yC, yD, t;
double a0, a1, a2, a3, b0, b1, b2, b3;
// очистка буфера цвета и буфера глубины
Gl.glClear( Gl.GL_COLOR_BUFFER_BIT | Gl.GL_DEPTH_BUFFER_BIT);
Gl.glClearColor(255, 255, 255, 1);
// очищение текущей матрицы
Gl.glLoadIdentity();
// установка черного цвета
Gl.glColor3f(0, 0, 0);
// помещаем состояние матрицы в стек матриц
Gl.glPushMatrix();
Gl.glPointSize(5.0f);
Gl.glBegin( Gl.GL_POINTS);
Gl.glVertex2d(0, 0);
Gl.glEnd();
Gl.glPointSize(1.0f);
PrintText2D(devX * Mcoord_X + 0.2f, ( float )ScreenH - devY * Mcoord_Y + 0.4f, "[
x: " + (devX * Mcoord_X).ToString() + " ; y: " + (( float )ScreenH - devY *
Mcoord_Y).ToString() + "]");
// выполняем перемещение в пространстве по осям X и Y
// выполняем цикл по контрольным точкам
for (i = 0; i < n; i++)
{
// сохраняем координаты точки (более легкое представление кода)
X = DrawingArray[i, 0];
Y = DrawingArray[i, 1];
// если точка выделена (перетаскивается мышью)
if (i == captured)
{
// для ее отрисовки используются более толстые линии
Gl.glLineWidth(3.0f);
}
// начинаем отрисовку точки (квадрат)
Gl.glBegin( Gl.GL_LINE_LOOP);
Gl.glVertex2d(X - eps, Y - eps);
Gl.glVertex2d(X + eps, Y - eps);
Gl.glVertex2d(X + eps, Y + eps);
Gl.glVertex2d(X - eps, Y + eps);
Gl.glEnd();
// если была захваченная точка - необходимо вернуть толщину линий
if (i == captured)
```

```
{
// возвращаем прежнее значение
Gl.glLineWidth(1.0f);
}
}
// дополнительный цикл по всем контрольным точкам -
// подписываем их координаты и номер
for (i = 0; i < n; i++)
{
// координаты точки
X = DrawingArray[i, 0];
Y = DrawingArray[i, 1];
// выводим подпись рядом с точкой
PrintText2D(( float )(X - 20), ( float )(Y - 20), "P " + i.ToString() + ": " + X.ToString()
+ ", " + Y.ToString());
}
// начинает отрисовку кривой
Gl.glBegin( Gl.GL_LINE_STRIP);
// используем все точки -1 (т.к. алгоритм "зацепит" i+1 точку)
for (i = 1; i < n-1; i++)
{
// реализация представленного в теоретическом описании алгоритма для кальку-
ляции сплайна
first = 1;
xA = DrawingArray[i - 1, 0];
xB = DrawingArray[i, 0];
xC = DrawingArray[i + 1, 0];
xD = DrawingArray[i + 2, 0];
yA = DrawingArray[i - 1, 1];
yB = DrawingArray[i, 1];
yC = DrawingArray[i + 1, 1];
yD = DrawingArray[i + 2, 1];
a3 = (-xA + 3 * (xB - xC) + xD) / 6.0;
a2 = (xA - 2 * xB + xC) / 2.0;
a1 = (xC - xA) / 2.0;
a0 = (xA + 4 * xB + xC) / 6.0;
b3 = (-yA + 3 * (yB - yC) + yD) / 6.0;
b2 = (yA - 2 * yB + yC) / 2.0;
b1 = (yC - yA) / 2.0;
b0 = (yA + 4 * yB + yC) / 6.0;
// отрисовка сегментов
for (j = 0; j <= N; j++)
```

```
{
// параметр t на отрезке от 0 до 1
t = ( double )j / ( double )N;
// генерация координат
X = (((a3 * t + a2) * t + a1) * t + a0);
Y = (((b3 * t + b2) * t + b1) * t + b0);
// и установка вершин
if (first == 1)
{
first = 0;
Gl.glVertex2d(X, Y);
}
else
Gl.glVertex2d(X, Y);
}
}
Gl.glEnd();
// завершаем рисование
Gl.glFlush();
// сигнал для обновления элемента, реализующего визуализацию
AnT.Invalidate();
}
```

Обработка события мыши, установленная для объекта *AnT*, решает две задачи: при перемещении курсора интерактивная (т.е. еще не зафиксированная) точка создает эффект создания сплайна на ходу. Но если установлен режим редактирования сплайна, то в том случае, если какая-либо из точек захвачена, т.е. на нее наведен курсор и зажата левая клавиша мыши, то до тех пор, пока пользователь не отпустит клавишу мыши, мы вносим изменения в координаты данной (захваченной) вершины.

Эта разность определяется как разница между последними сохраненными координатами в данном режиме и текущими координатами указателя. Таким образом, начав на вершине, пользователь может передвинуть ее, при этом наблюдая в реальном времени, как изменяется геометрия данного сплайна.

Обработка событий мыши:

```
// обработка движения мыши
private void AnT_MouseMove( object sender, Mouse EventArgs e)
{
// если установлен режим создания сплайна
```

```
if (comboBox1.SelectedIndex == 0)
{
// сохраняем координаты мыши
Mcoord_X = e.X;
Mcoord_Y = e.Y; // вычисляем параметры для будущей дорисовки линий от ука-
зателя мыши к координатным осям.
lineX = devX * e.X;
lineY = ( float )(ScreenH - devY * e.Y);
// текущая (интерактивная) точка, добавляемая к уже установленным, непрерыв-
но изменяется от движения
// мыши и создает эффект интерактивности и наглядности приложения
DrawingArray[count_points, 0] = lineX;
DrawingArray[count_points, 1] = lineY;
}
else
{
// обычное протоколирование координат для подсвечивания вершины в случае
наведения
// сохраняем координаты мыши
Mcoord_X = e.X;
Mcoord_Y = e.Y;
// вычисляем параметры для будущей дорисовки линий от указателя мыши к ко-
ординатным осям
float _lastX = lineX;
float _lastY = lineY;
lineX = devX * e.X;
lineY = ( float )(ScreenH - devY * e.Y);
// если точка захвачена (т.е. пользователь удерживает кнопку мыши)
if (captured != -1)
{
// то мы вносим разницу с последними координатами курсора
// другими словами перемещаем захваченную точку
DrawingArray[captured, 0] -= _lastX-lineX;
DrawingArray[captured, 1] -= _lastY-lineY;
} } }
// щелчок мыши
private void AnT_MouseClick( object sender, Mouse EventArgs e)
{
// если мы находимся в режиме создания сплайна
if (comboBox1.SelectedIndex == 0)
{
// забираем координаты мыши
```

```
Mcoord_X = e.X;
Mcoord_Y = e.Y;
// приводим к нужному нам формату в соответствии с настройками проекции
lineX = devX * e.X;
lineY = ( float )(ScreenH - devY * e.Y);
// создаем новую контрольную точку
DrawingArray[count_points, 0] = lineX;
DrawingArray[count_points, 1] = lineY;
// и увеличиваем значение счетчика контрольных точек
count_points++;
} }
// если ОТПУЩЕНА клавиша мыши
private void AnT_MouseDown( object sender, Mouse EventArgs e)
{
// если режим редактирования сплайна
if (comboBox1.SelectedIndex == 1)
{
// получаем и преобразовываем координаты нажатия
Mcoord_X = e.X;
Mcoord_Y = e.Y;
lineX = devX * e.X;
lineY = ( float )(ScreenH - devY * e.Y);
// проходим циклом по всем установленным контрольным точкам
for ( int ax = 0; ax < count_points; ax++)
{
// если точка попадает под курсор
if (lineX < DrawingArray[ax, 0] + 5 && lineX > DrawingArray[ax, 0] - 5 && lineY <
DrawingArray[ax, 1] + 5 && lineY > DrawingArray[ax, 1] - 5)
{
// отмечаем ее как захваченную (записываем ее индекс в массив captured)
captured = ax;
// останавливаем цикл, мы нашли нужную точку
break ;
} } } }
// если клавиша мыши поднята - пользователь отпустил клавишу и перемещение
точки должно остановиться
private void AnT_MouseUp( object sender, Mouse EventArgs e)
{
// отмечаем, что нет захваченной точки
captured = -1;
}
```

Теперь можно протестировать работу программы (рис. 3.8).

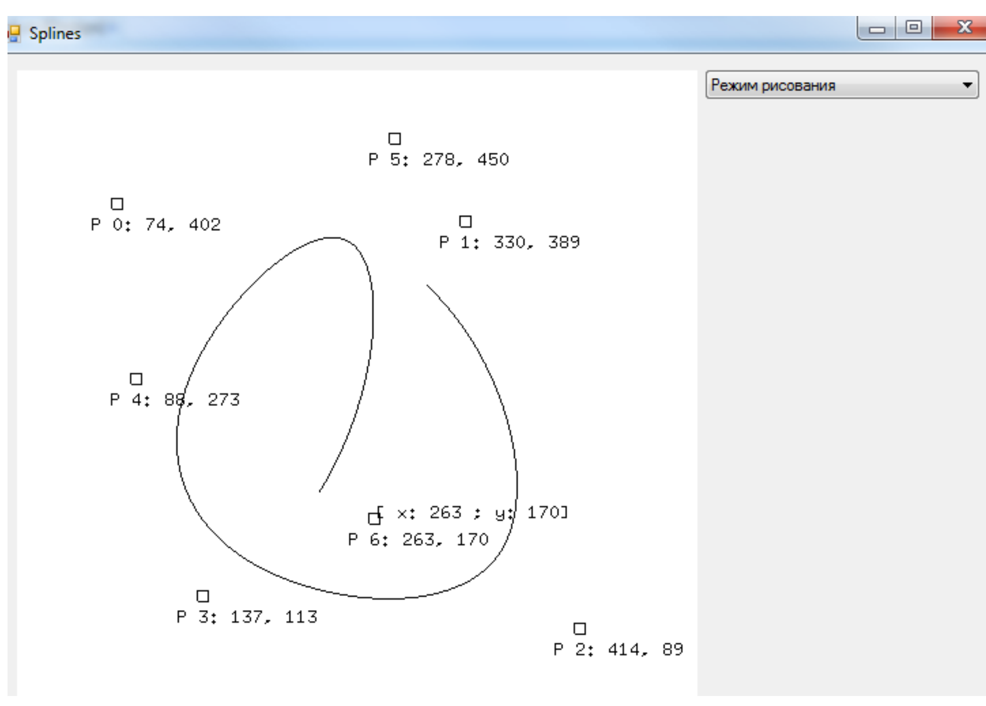


Рис. 3.8.

Практическое задание

1. Ознакомиться с изложенным выше и представленным в литературе теоретическим материалом.

2. Выполнить действия, приведенные в п. 3.2. При разработке программы имя проекта, создаваемого в *MS Visual Studio*, должно содержать фамилию студента и группу (например, *Ivanov_Ivan_ISG_105_lab_1*).

3. Выполнить реализацию сплайна в соответствии с вариантом задания.

4. Продемонстрировать работу программы на скриншотах (включая реализованные методы).

Скриншот должен включать заголовок окна консоли с полным путем к имени пользователя и папки с проектом.

Папка с программой должна содержать весь проект, выполненный в *MS Visual Studio 2008*, и исполняемые файлы.

Контрольные вопросы

1. Сплайны в компьютерной графике.
2. Доказательство гладкости кубических *B*-сплайнов.

Тема 4. ГЕОМЕТРИЧЕСКИЕ ПРЕОБРАЗОВАНИЯ В *2D*

Цель изучения темы. Освоение методов геометрических преобразований графических объектов, приобретение навыков использования средств геометрических преобразований при составлении графических программ.

4.1. Введение в геометрические преобразования

Общие сведения

При геометрических преобразованиях графических объектов не изменяется само пространство, меняется только положение, направление и масштаб координатной системы, в которой определяется положение точек в пространстве. То есть при этом не затрагивается прежняя структура изображения – сохраняются отношения типа принадлежности прямой, дуге, поверхности и т.д. Поэтому тип объекта не влияет на выполнение геометрического преобразования и, следовательно, преобразованиям подвергаются только данные (точки).

К операциям геометрического преобразования двухмерных (плоских) графических объектов относятся: поворот осей координат, перенос начала координат, изменение масштаба изображения.

Поворот

При повороте системы координат XY в декартовой плоскости относительно центра вращения, совпадающего с началом координат, на некоторый угол w координаты произвольной точки P в новой системе координат (x', y') могут быть выражены через координаты этой точки в прежней системе координат (x, y):

$$x' = x \cos(w) + y \sin(w),$$
$$y' = -x \sin(w) + y \cos(w).$$

Этот поворот можно интерпретировать как частный случай поворота в пространстве XYZ, а именно как плоский поворот относительно оси Z, проходящей через центр вращения перпендикулярно плоскости XY.

Когда преобразуется большое число точек, операция поворота выполняется с использованием матрицы поворота:

$$R = \begin{vmatrix} \cos w & \sin w & 0 \\ -\sin w & \cos w & 0 \\ 0 & 0 & 1 \end{vmatrix}.$$

Матрица R – квадратная, размерностью 3×3 для двухмерного пространства и размерностью 4×4 для трехмерного.

Перенос

При переносе начала координат направление осей новой системы координат остается прежним, сохраняется и масштаб. Однако новое начало координат соответствует точке (T_x, T_y) в прежней системе. Следовательно, в новой системе координат прежнее начало координат соответствует $(-T_x, -T_y)$, а точка (x, y) прежней системы координат становится точкой $(x - T_x, y - T_y)$ в новой системе координат.

Операция переноса выполняется с помощью матрицы переноса:

$$T = \begin{vmatrix} 1 & 0 & -T_x \\ 0 & 1 & -T_y \\ 0 & 0 & 1 \end{vmatrix}.$$

Масштабирование

При этом преобразовании начало координат и направление осей старой и новой систем координат одинаковы, но масштаб по осям различен. Пусть отрезок единичной длины на исходной оси X становится отрезком длины S_x на новой оси X', а единичный отрезок на прежней оси Y становится отрезком длиной S_y на новой. Тогда точка (x, y) в прежней системе имеет координаты (xS_x, yS_y) в новой системе; например точка (1, 1) становится точкой (S_x, S_y).

Преобразование масштабирования выполняется с помощью матрицы масштабирования:

$$S = \begin{vmatrix} S_x & 0 & 0 \\ 0 & S_y & 0 \\ 0 & 0 & 1 \end{vmatrix}.$$

Общий алгоритм двухмерных геометрических преобразований

Алгоритм выполнения двухмерных геометрических преобразований с произвольным графическим объектом может быть представлен в виде последовательности шагов:

1. Вычисление матрицы плоского (выполняемого на плоскости $X0Y$) поворота R на заданный угол α.

2. Вычисление матрицы переноса T на заданное расстояние, определяемое константами переноса вдоль каждой из двух осей на плоскости – T_x и T_y.

3. Вычисление матрицы масштабирования S на заданный коэффициент, определяемый масштабными коэффициентами по каждой из двух осей на плоскости – S_x и S_y.

4. Выполнение операций последовательного перемножения полученных матриц R, T, S в заданном порядке в зависимости от требуемого геометрического преобразования и получение результирующей матрицы преобразования Q.

5. Построение изображения в исходной системе координат с запоминанием вычисленных координат точек изображения в массивах, то есть формирование изображения без вывода его на экран.

6. Преобразование координат всех точек изображения по заданному правилу путем умножения результирующей матрицы геометрических преобразований Q на вектор координат каждой точки изображения $P=(x, y, 1)$.

7. Построение изображения на экране.

Шаги 5 – 7 алгоритма могут быть объединены. При этом координаты точек отдельного примитива (пикселя, отрезка прямой) вычисляются в исходной системе координат, затем преобразуются и сразу выводятся на экран. Тогда нет необходимости хранить в памяти компьютера структуру всего изображения целиком: точки объекта выводятся на экран по мере вычисления.

Любое изменение изображения при переходе от первоначально определенной системы координат к новой (экранной) будет состоять для плоских графических объектов из комбинации перечисленных трех типов преобразований. При этом важно соблюдать порядок их применения; изменение последовательности преобразований может привести к получению изображения, отличного от того, которое хотел получить программист.

Пример выполнения геометрических преобразований

Приведем пример аналитического выполнения двухмерного геометрического преобразования.

Пусть полное изменение системы координат достигается:

- переносом начала координат в точку (1, 0);
- поворотом осей координат на $\pi/4$ радиан;
- изменением масштаба по оси X в два раза (т.е. $S_x = 2$).

Комбинация этих трех изменений (в указанном порядке) формирует систему координат экрана. Что произойдет с отрезком прямой линии, соединяющей точки (3, 2) и (–1, –1) в прежней системе координат?

При всех геометрических преобразованиях прямая линия преобразуется также в прямую линию, поэтому необходимо рассмотреть, что произойдет с координатами начала и конца отрезка. Построим матрицы преобразований:

$$T = \begin{vmatrix} 1 & 0 & 1 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{vmatrix};\; R = \begin{vmatrix} 2^{(-1/2)} & 2^{(-1/2)} & 0 \\ -2^{(-1/2)} & -2^{(-1/2)} & 0 \\ 0 & 0 & 1 \end{vmatrix};\quad S = \begin{vmatrix} 2 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{vmatrix}.$$

Координаты исходных точек представим в виде векторов:

$$P_1 = \begin{vmatrix} 3 \\ 2 \\ 1 \end{vmatrix};\qquad P_2 = \begin{vmatrix} -1 \\ -1 \\ 1 \end{vmatrix}.$$

Тогда вектор координат новых точек получается (цифры обозначают последовательность выполнения матричных операций):

$$\begin{matrix} & (2) & (1) & (3) & \\ P_1' = S \times & R \times & T \times & P_1 & = \end{matrix} \begin{vmatrix} 2^{(1/2)} & 2^{(1/2)}) & -2^{(1/2)} \\ -2^{(-1/2)} & 2^{(-1/2)} & 2^{(-1/2)} \\ 0 & 0 & 1 \end{vmatrix} \times \begin{vmatrix} 3 \\ 2 \\ 1 \end{vmatrix}.$$

То есть новые координаты точки (3, 2) становятся $P_1' = (4 \cdot 2^{(1/2)}, 0)$. Точно так же:

$$\begin{matrix} & (2) & (1) & (3) & \\ P_2' = S \times & R \times & T \times & P_2 & = \end{matrix} \begin{vmatrix} 2^{(1/2)} & 2^{(1/2)}) & -2^{(1/2)} \\ -2^{(-1/2)} & 2^{(-1/2)} & 2^{(-1/2)} \\ 0 & 0 & 1 \end{vmatrix} \times \begin{vmatrix} -1 \\ -1 \\ 1 \end{vmatrix}.$$

То есть $P_2' = (-3 \cdot 2^{(1/2)}, 2^{(-1/2)})$.

Далее соединяем между собой отрезком прямой линии точки P_1' и P_2' – это и будет полученный отрезок прямой линии после выполнения заданных геометрических преобразований на плоскости над исходным отрезком (с начальной и конечной точками P_1 и P_2).

Во всех представленных выше выражениях знак "×" обозначает матричное умножение. Произведение двух матриц A и B:

$$\begin{vmatrix} a_{11}\ a_{12}\ a_{13} \\ a_{21}\ a_{22}\ a_{23} \\ a_{31}\ a_{32}\ a_{33} \end{vmatrix} \times \begin{vmatrix} b_{11}\ b_{12}\ b_{13} \\ b_{21}\ b_{22}\ b_{23} \\ b_{31}\ b_{32}\ b_{33} \end{vmatrix} = \begin{vmatrix} c_{11}\ c_{12}\ c_{13} \\ c_{21}\ c_{22}\ c_{23} \\ c_{31}\ c_{32}\ c_{33} \end{vmatrix},$$

где $c_{ij} = a_{i1}\, b_{1j} + a_{i2}\, b_{2j} + a_{i3}\, b_{3j}$.

OpenGL поддерживает встроенную систему матриц для выполнения геометрических преобразований всей сцены или отдельных объектов, но для понятия принципов работы матричных геометрических преобразований реализуем все матричные операции самостоятельно.

4.2. Программа геометрических преобразований в *2D*

Алгоритм выполнения двухмерных геометрических преобразований в нашей демонстрационной программе для наглядности выполнения отдельных операций отличается от классического общего алгоритма (п. 4.1) следующим:

- вычисление матрицы переноса на заданное расстояние выполняется отдельно по каждой оси (по оси *X* и по оси *Y*) с помощью вызова функции *CreateTranslate(T, Os)*, где *T* – константа переноса, *Os* – ось, по которой выполняется перенос;

- вычисление матрицы масштабирования на заданный коэффициент будет выполняться отдельно по каждой оси (по оси *X* и по оси *Y*) с помощью вызова функции *CreateZoom(S, Os)*, где *S* – масштабный коэффициент, *Os* – ось, по которой выполняется масштабирование;

- вместо предварительного выполнения операций перемножения всех вычисленных матриц отдельных геометрических преобразований выполняется умножение текущей матрицы (полученной по команде пользователя) каждого отдельного геометрического преобразования на векторы координат точек объекта;

- полученное на каждом шаге (по каждой отдельной команде пользователя) изображение сразу выводится на экран.

Вычисление матрицы поворота по оси *Z* на заданный угол выполняется с помощью вызова функции *CreateRotate(α)*, где α – угол поворота.

Константы переноса, поворота и масштабирования, применяемые при каждом нажатии пользователем соответствующей клавиши, установлены в коде функции обработки нажатия клавиш *AnT_KeyDown*.

Для выполнения всех операций вместо вектора однородных координат точки на плоскости $P = (x, y, 1)$ используем координаты точек $P = (x, y)$; третья координата в вычислениях не участвует – это позволит нам несколько упростить код. При построениях в пространстве везде будет приниматься координата $z = 0$.

Создайте окно программы и разместите на ней элемент *openglsimplecontrol*, как показано на рис. 4.1, после чего установите его размеры 500 × 500. Переименуйте данный объект, дав ему имя *AnT*.

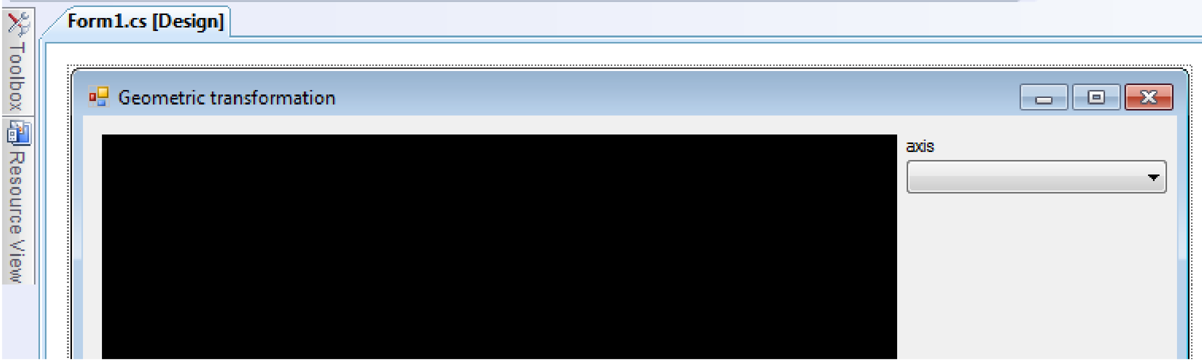


Рис. 4.1

Справа от данного элемента разместите элемент *comboBox*, после чего в его свойствах установите значение параметра *DropDownStyle* = *DropDownList*. Тогда выпадающие элементы перестанут быть доступными для редактирования. После этого измените элементы *Items* как показано на рис. 4.2.

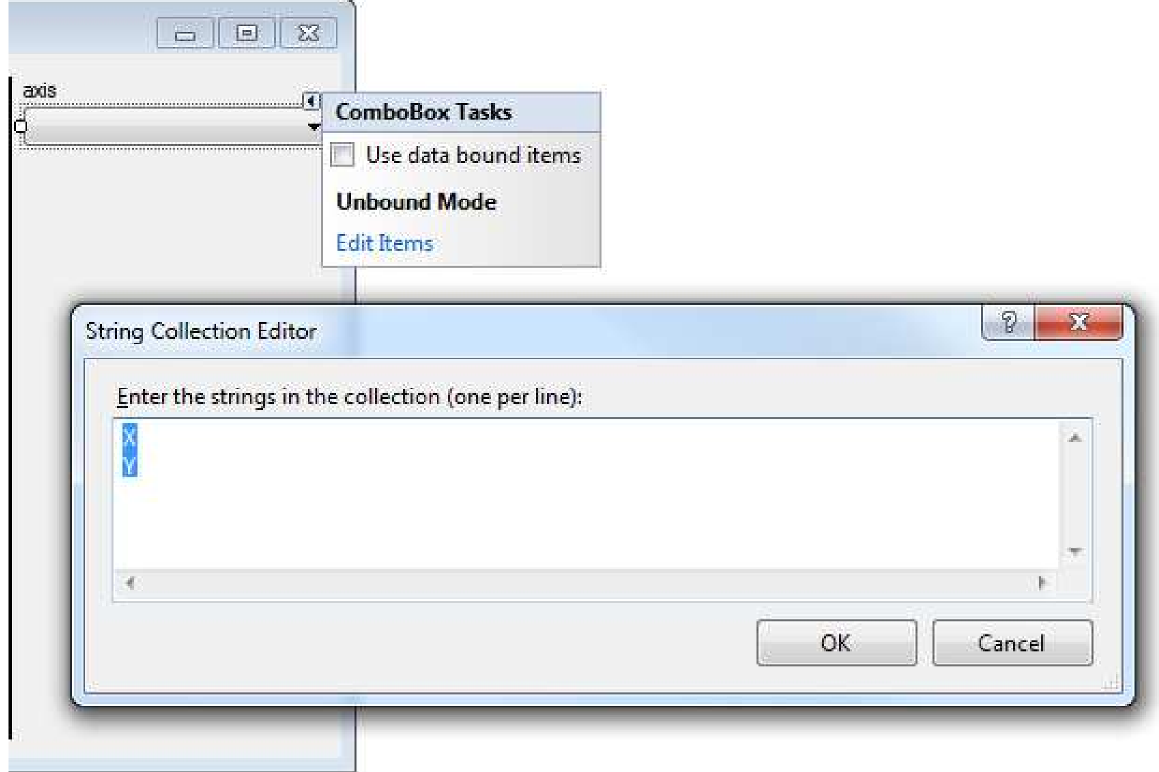


Рис. 4.2

Не забудьте установить ссылки на используемые библиотеки *Tao* (рис. 4.3).

Для обработки событий клавиатуры выделите объект *AnT*, перейдите к его свойствам, после чего перейдите к настройке событий и добавьте обработку событий мыши, как показано на рис. 4.4.

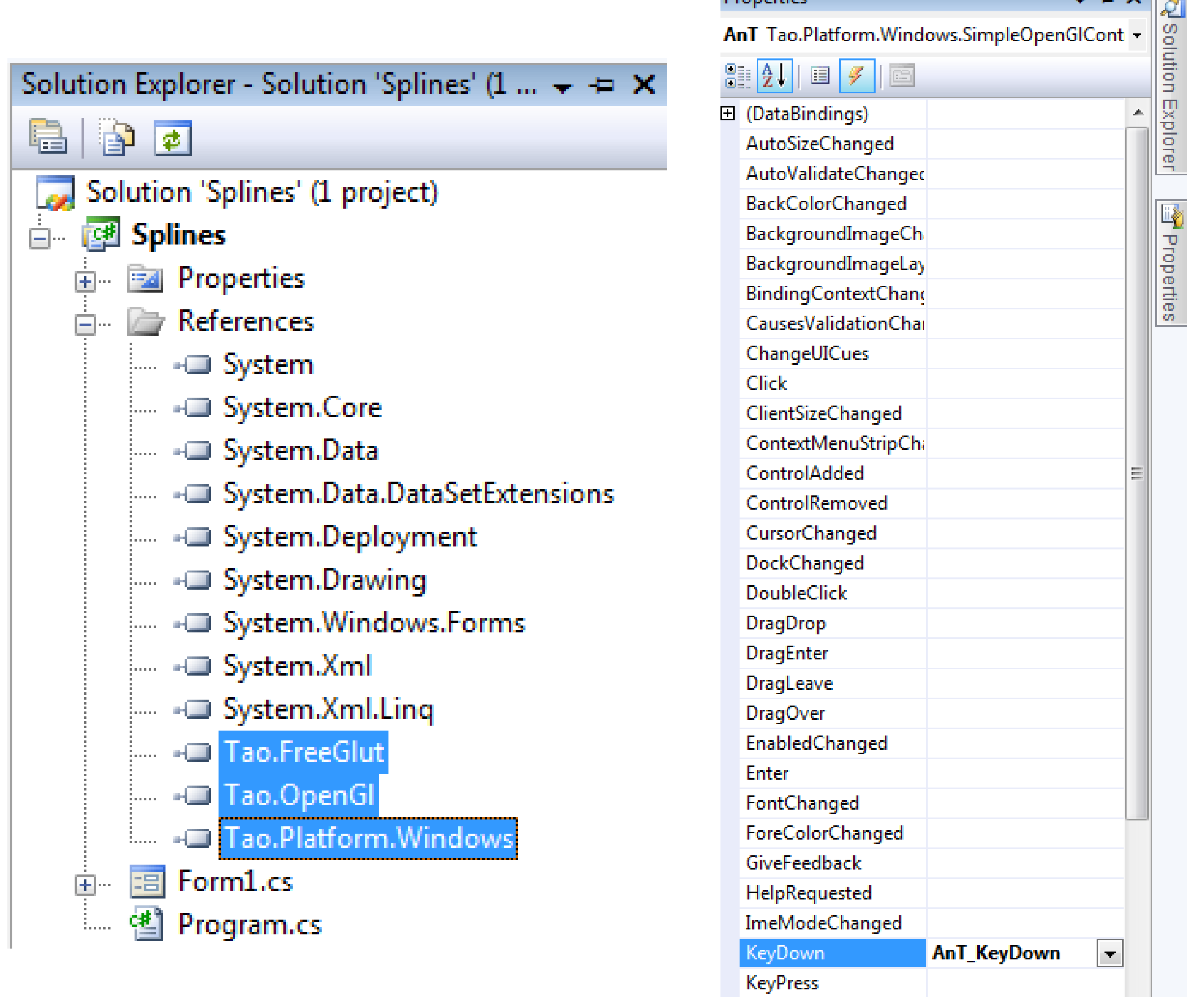


Рис. 4.3 *Рис. 4.4*

Для реализации визуализации используется таймер – после инициализации окна он генерирует событие, называемое "тиком" таймера, раз в 30 миллисекунд – добавьте элемент таймер, переименуйте экземпляр в *RenderTimer* и установите время "тика"

30 миллисекунд (как показано на рис. 4.5), а также добавьте ему функцию обработки события таймера.

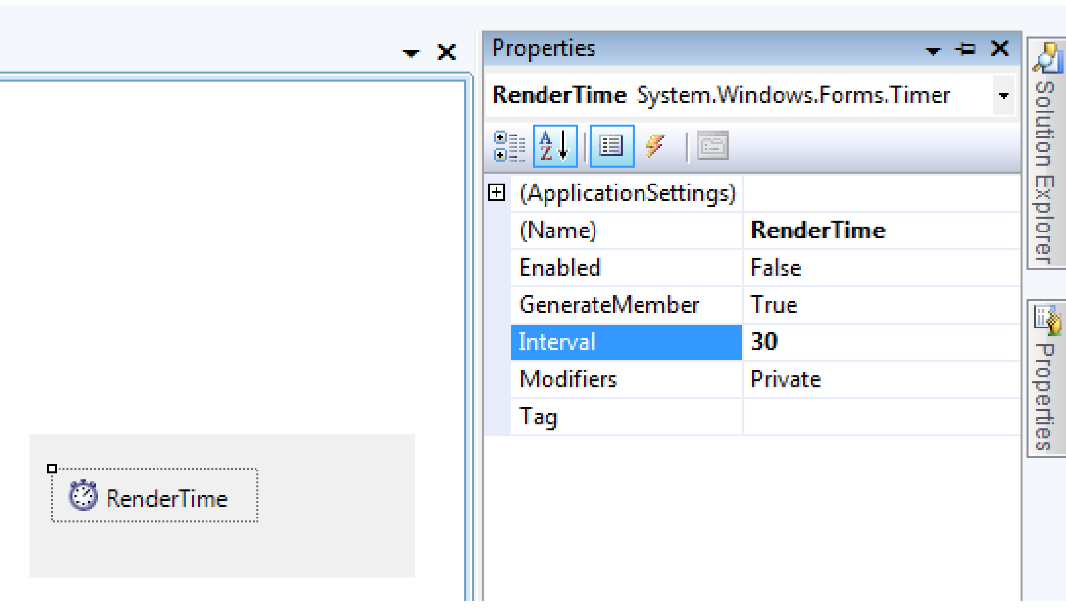


Рис. 4.5

Нам потребуется объявить ряд переменных для дальнейшей работы программы.

Инициализация окна и *OpenGl* происходит так же, как и в предыдущих проектах:

```
public Form1() {
InitializeComponent();
// инициализация для работы с openGL
AnT.InitializeContexts();
}
// массив вершин создаваемого геометрического объекта
private float[,] GeomObject = new float[32, 3];
// счетчик его вершин
private int count_elements = 0;
// событие загрузки формы окна
private void Form1_Load( object sender, EventArgs e)
{
// инициализация OpenGL, много раз комментированная ранее
// 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();
Glu.gluPerspective(45, (float)AnT.Width / (float)AnT.Height, 0.1, 200);
Gl.glMatrixMode( Gl.GL_MODELVIEW);
Gl.glLoadIdentity();
Gl.glEnable( Gl.GL_DEPTH_TEST);
// геометрический объект для визуализации
// (это четырехугольник; определим его 4 вершины)
GeomObject[0, 0] = -0.7f;
GeomObject[0, 1] = 0.0f;
GeomObject[1, 0] = 0.7f;
GeomObject[1, 1] = 0.0f;
GeomObject[2, 0] = 0.0f;
GeomObject[2, 1] = 1.0f;
GeomObject[3, 0] = 0.0f;
GeomObject[3, 1] = 0.7f;
// количество вершин рассматриваемого геометрического объекта
count_elements = 4;
// устанавливаем ось X по умолчанию
comboBox1.SelectedIndex = 0;
// начало визуализации (активируем таймер)
RenderTimer.Start();
}
```

Обработка события таймера:

```
private void RenderTime_Tick( object sender, EventArgs e)
{
// обработка "тика" таймера - вызов функции отрисовки
Draw();
}
```

Функция отрисовки:

```
// функция отрисовки
private void Draw()
{
// очистка буфера цвета и буфера глубины
Gl.glClear( Gl.GL_COLOR_BUFFER_BIT | Gl.GL_DEPTH_BUFFER_BIT);
```

```
Gl.glClearColor(255, 255, 255, 1);
// очищение текущей матрицы
Gl.glLoadIdentity();
// установка черного цвета
Gl.glColor3f(0, 0, 0);
// помещаем состояние матрицы в стек матриц
Gl.glPushMatrix();
// перемещаем камеру для лучшего обзора объекта
Gl.glTranslated(0, 0, -7);
// поворачиваем ее на 15 градусов
Gl.glRotated(15, 1, 1, 0);
// помещаем состояние матрицы в стек матриц
Gl.glPushMatrix();
// начинаем отрисовку объекта
Gl.glBegin( Gl.GL_LINE_LOOP);
// геометрические данные мы берем из массива GeomObject
// рисуем объект с помощью замкнутой линии (координата z=0)
Gl.glVertex3d(GeomObject[0, 0], GeomObject[0, 1], 0);
Gl.glVertex3d(GeomObject[1, 0], GeomObject[1, 1], 0);
Gl.glVertex3d(GeomObject[2, 0], GeomObject[2, 1], 0);
Gl.glVertex3d(GeomObject[3, 0], GeomObject[3, 1], 0);
// завершаем отрисовку примитивов
Gl.glEnd();
// возвращаем состояние матрицы
Gl.glPopMatrix();
// возвращаем состояние матрицы
Gl.glPopMatrix();
// отрисовываем геометрию
Gl.glFlush();
// обновляем состояние элемента
AnT.Invalidate();
}
```

Сами преобразования реализованы с помощью трёх функций, которые будут активированы при обработке соответствующих клавиш. Нажатие клавиш с символами *W / S* вызывает перемещение по указанной оси, клавиш *Z / X* – масштабирование по указанной оси, клавиш *A / D* – вращение вокруг оси *Z*.

Обработка данных клавиш клавиатуры и обработка события элемента *comboBox* при изменении его значения для исключения перехвата им нажатия клавиш:

```
// обработка событий от клавиатуры - нажатие (клавиша зажата!)
private void AnT_KeyDown( object sender, Key EventArgs e)
{
// Z и X отвечают за масштабирование
if (e.KeyCode == Keys.Z)
{
// вызов функции, в которой мы реализуем масштабирование -
// передаем коэффициент масштабирования и выбранную ось в окне программы
CreateZoom(1.05f, comboBox1.SelectedIndex);
}
if (e.KeyCode == Keys.X)
{
// вызов функции, в которой мы реализуем масштабирование -
// передаем коэффициент масштабирования и выбранную ось в окне программы
CreateZoom(0.95f, comboBox1.SelectedIndex);
}
// W и S отвечают за перенос
if (e.KeyCode == Keys.W)
{
// вызов функции, в которой мы реализуем перенос -
// передаем коэффициент переноса и выбранную ось в окне программы
CreateTranslate(0.05f, comboBox1.SelectedIndex);
}
if (e.KeyCode == Keys.S)
{
// вызов функции, в которой мы реализуем перенос -
// передаем коэффициент переноса и выбранную ось в окне программы
CreateTranslate(-0.05f, comboBox1.SelectedIndex);
} // A и D отвечают за поворот
if (e.KeyCode == Keys.A)
{
// вызов функции, в которой мы реализуем поворот -
// передаем угол поворота относительно оси Z
CreateRotate(0.05f);
}
if (e.KeyCode == Keys.D)
{
// вызов функции, в которой мы реализуем поворот -
// передаем угол поворота относительно оси Z
CreateRotate(-0.05f);
}
}
```

```
// дополнительное событие для элемента comboBox - если произошло
// изменение его значения - установить фокус в элемент AnT
// чтобы избежать перехват события нажатия клавиши данным элементом
private void comboBox1_SelectedIndexChanged( object sender, EventArgs e)
{
// устанавливаем фокус в AnT
AnT.Focus();
}
```

И наконец, собственная реализация матричных операций поворота, переноса и масштабирования:

```
// функция масштабирования
private void CreateZoom( float coef, int os)
{
// создаем матрицу
float[,] Zoom2D = new float[3, 3];
Zoom2D[0, 0] = 1;
Zoom2D[1, 0] = 0;
Zoom2D[2, 0] = 0;
Zoom2D[0, 1] = 0;
Zoom2D[1, 1] = 1;
Zoom2D[2, 1] = 0;
Zoom2D[0, 2] = 0;
Zoom2D[1, 2] = 0;
Zoom2D[2, 2] = 1;
// устанавливаем коэффициент масштабирования для необходимой (выбранной и
переданной в качестве параметра) оси
Zoom2D[os, os] = coef;
// вызываем функцию для выполнения умножения матрицы
// координат вершин геометрического объекта
// на созданную в данной функции матрицу
multiply(GeomObject, Zoom2D);
}
// функция переноса
private void CreateTranslate( float translate, int os)
{
// создаем матрицу
float[,] Tran2D = new float[3, 3];
Tran2D[0, 0] = 1;
Tran2D[1, 0] = 0;
Tran2D[2, 0] = 0;
```

```
Tran2D[0, 1] = 0;
Tran2D[1, 1] = 1;
Tran2D[2, 1] = 0;
Tran2D[0, 2] = 0;
Tran2D[1, 2] = 0;
Tran2D[2, 2] = 1;
// устанавливаем коэффициент переноса для необходимой (выбранной и пере-
данной в качестве параметра) оси
Tran2D[2, os] = translate;
// вызываем функцию для выполнения умножения матрицы
// координат вершин геометрического объекта
// на созданную в данной функции матрицу
multiply(GeomObject, Tran2D);
}
// реализация поворота
private void CreateRotate( float angle)
{
// массив, который будет содержать матрицу
float[,] Rotate2D = new float[3, 3];
Rotate2D[0, 0] = (float)Math.Cos(angle);
Rotate2D[1, 0] = (float)-Math.Sin(angle);
Rotate2D[2, 0] = 0;
Rotate2D[0, 1] = (float)Math.Sin(angle);
Rotate2D[1, 1] = (float)Math.Cos(angle);
Rotate2D[2, 1] = 0;
Rotate2D[0, 2] = 0;
Rotate2D[1, 2] = 0;
Rotate2D[2, 2] = 1;
// вызываем функцию для выполнения умножения матрицы
// координат вершин геометрического объекта
// на созданную в данной функции матрицу
multiply(GeomObject, Rotate2D);
}
// функция умножения вектора координат точек объекта на матрицу преобразо-
вания
private void multiply(float[,] obj, float[,] matrix)
{
// временные переменные
float res_1, res_2;
// проходим циклом по всем координатам (это вектор A [x,y,1])
// и умножаем каждый вектор на матрицу B (переданную)
```

```
// результат сразу заносим в массив геометрии
for ( int ax = 0; ax < count_elements; ax++)
{
res_1 = (obj[ax, 0] * matrix[0, 0] + obj[ax, 1] * matrix[0, 1] + matrix[0, 2]);
res_2 = (obj[ax, 0] * matrix[1, 0] + obj[ax, 1] * matrix[1, 1] + matrix[1, 2]);
obj[ax, 0] = res_1;
obj[ax, 1] = res_2;
} }
```

Результат работы программы можно увидеть на рис. 4.6. Там изображен двухмерный объект (в данном случае это четырехугольник), управляемый нажатиями кнопок *W, S, A, D, Z, X* и установкой оси в окне формы.

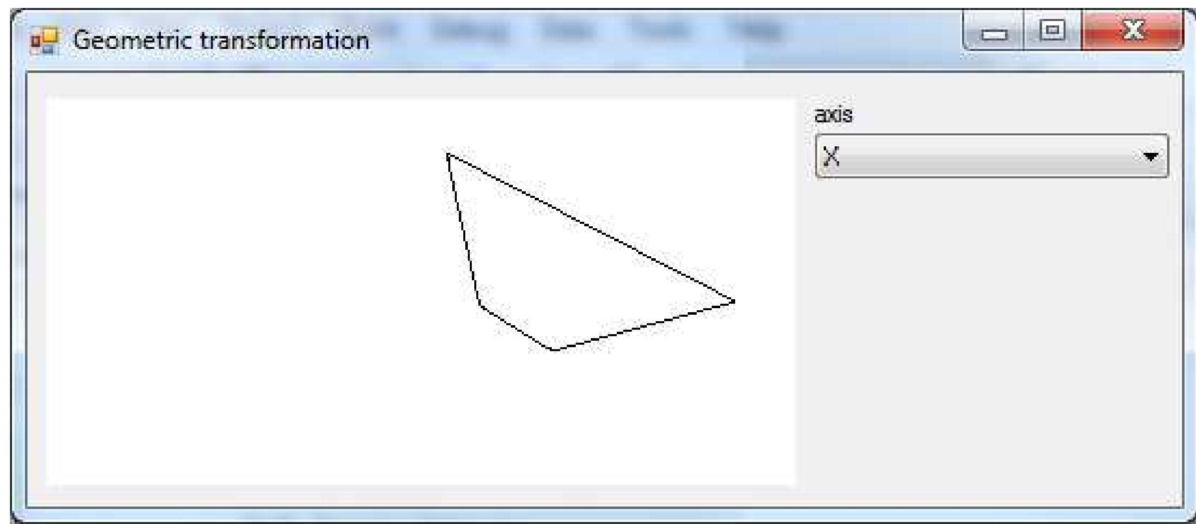


Рис. 4.6

Данное приложение демонстрирует математические основы геометрических преобразований.

При использовани функций *OpenGL* нет необходимости организовывать преобразования в коде приложения – достаточно сохранять в стек матриц текущее состояние объекта, после чего производить преобразования с видовой матрицей, отрисовывать объект и возвращать сохраненную матрицу.

Для реализации операций перемещения, поворота и масштабирования нам достаточно использовать имеющиеся в *OpenGL* функции *glTranslate***, *glRotate**, glScale***.

4.3. Программа *2D* геометрических преобразований (добавление оси *Z*)

Начало разработки приложения аналогично п. 4.2.

Создайте окно программы и разместите на ней элемент *openglsimplecontrol*, как показано на рис. 4.7, после чего установите его размеры 500 × 500. Переименуйте данный объект, дав ему имя *AnT*.

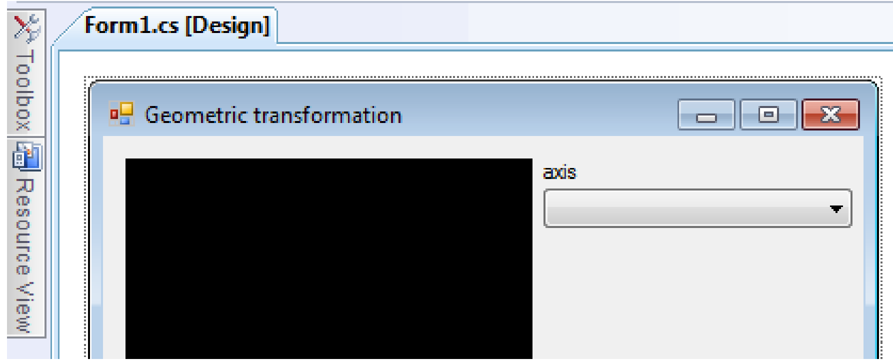


Рис. 4.7

Справа от данного элемента разместите элемент *comboBox*, после чего в его свойствах установите значение параметра *DropDownStyle* = *DropDownList*. Тогда выпадающие элементы перестанут быть доступными для редактирования. После этого измените элементы *Items* как показано на рис. 4.8 (обратите внимание на добавленную строку с символом *Z)*.

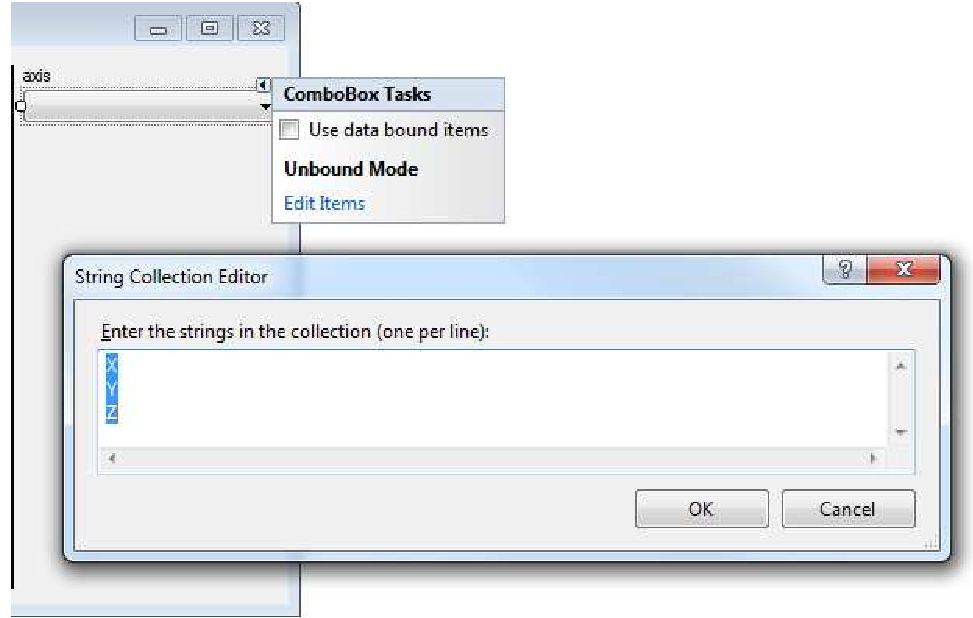


Рис. 4.8

Не забудьте установить ссылки на используемые библиотеки *Tao* (рис. 4.9).

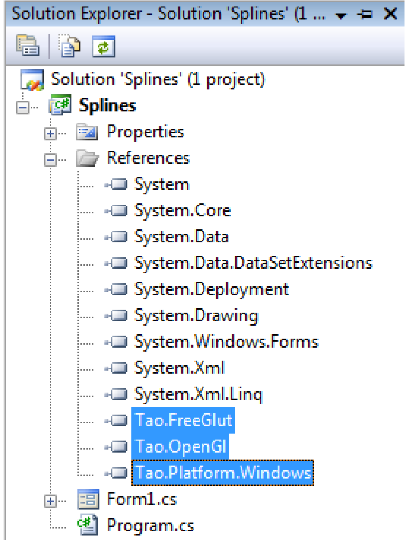


Рис. 4.9

Для обработки событий клавиатуры выделите объект *AnT*, перейдите к его свойствам, после чего перейдите к настройке событий и добавьте обработку событий мыши, как показано на рис. 4.10.

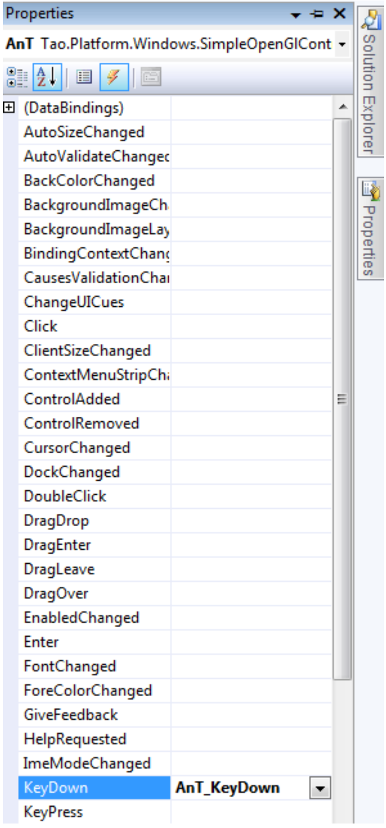


Рис. 4.10

Для реализации визуализации будет использоваться таймер – после инициализации окна он будет генерировать событие, называемое "тиком" таймера, раз в 30 миллисекунд – добавьте элемент таймер, переименуйте экземпляр в *RenderTimer* и установите время "тика" 30 миллисекунд (как показано на рис. 4.11), а также добавьте ему функцию обработки события таймера.

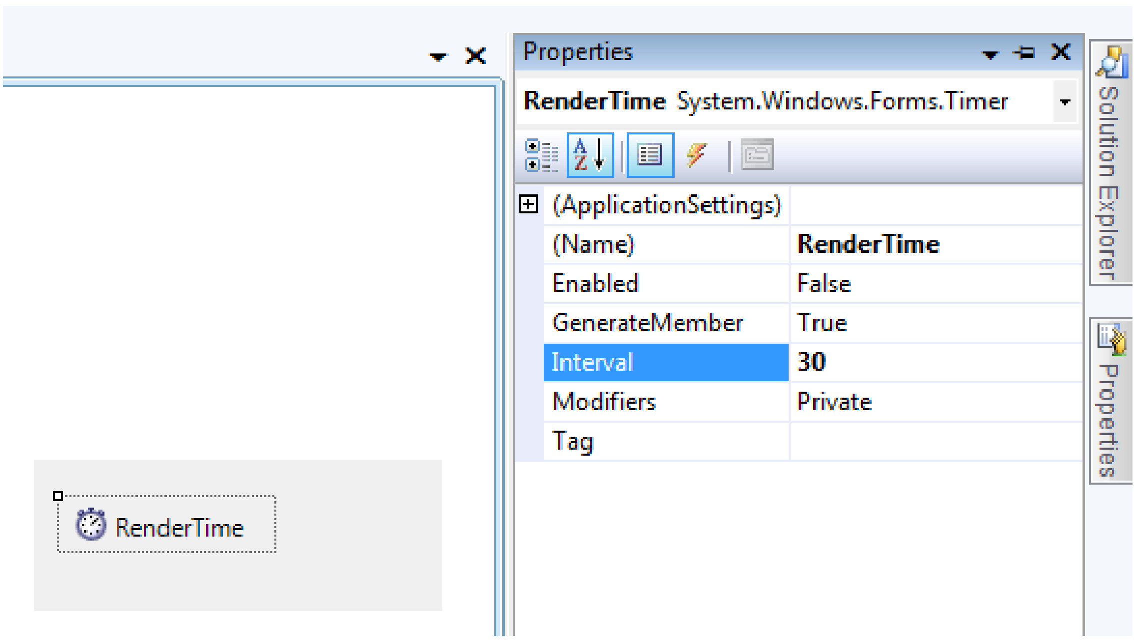


Рис. 4.11

Нам потребуется объявить ряд переменных для дальнейшей работы программы.

Инициализация окна и *OpenGl* происходит так же, как и в предыдущих проектах:

```
public Form1() {
InitializeComponent();
// инициализация для работы с openGL
AnT.InitializeContexts();
}
// массив вершин создаваемого геометрического объекта
private float[,] GeomObject = new float[32, 3];
// счетчик его вершин
private int count_elements = 0;
// событие загрузки формы окна
private void Form1_Load( object sender, EventArgs e)
```

```
{
// инициализация OpenGL, много раз комментированная ранее
// инициализация бибилиотеки 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();
Glu.gluPerspective(45, (float)AnT.Width / (float)AnT.Height, 0.1, 200);
Gl.glMatrixMode( Gl.GL_MODELVIEW);
Gl.glLoadIdentity();
Gl.glEnable( Gl.GL_DEPTH_TEST);
// геометрический объект для визуализации - пирамида (4 вершины)
GeomObject[0, 0] = -0.7f;
GeomObject[0, 1] = 0;
GeomObject[0, 2] = 0;
GeomObject[1, 0] = 0.7f;
GeomObject[1, 1] = 0;
GeomObject[1, 2] = 0;
GeomObject[2, 0] = 0.0f;
GeomObject[2, 1] = 0;
GeomObject[2, 2] = 1.0f;
GeomObject[3, 0] = 0;
GeomObject[3, 1] = 0.7f;
GeomObject[3, 2] = 0.3f;
// количество вершин рассматриваемого геометрического объекта
count_elements = 4;
// устанавливаем ось X по умолчанию
comboBox1.SelectedIndex = 0;
// начало визуализации (активируем таймер)
RenderTimer.Start();
}
```

Обработка события таймера:

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

```
{
// обработка "тика" таймера - вызов функции отрисовки
Draw();
}
```

Функция отрисовки:

```
// функция отрисовки
private void Draw()
{
// очистка буфера цвета и буфера глубины
Gl.glClear( Gl.GL_COLOR_BUFFER_BIT | Gl.GL_DEPTH_BUFFER_BIT);
Gl.glClearColor(255, 255, 255, 1);
// очищение текущей матрицы
Gl.glLoadIdentity();
// установка черного цвета
Gl.glColor3f(0, 0, 0);
// помещаем состояние матрицы в стек матриц
Gl.glPushMatrix();
// перемещаем камеру для лучшего обзора объекта
Gl.glTranslated(0, 0, -7);
// поворачиваем ее на 15 градусов
Gl.glRotated(15, 1, 1, 0);
// помещаем состояние матрицы в стек матриц
Gl.glPushMatrix();
// начинаем отрисовку объекта
Gl.glBegin( Gl.GL_LINE_LOOP);
// геометрические данные мы берем из массива GeomObject
// рисуем объект с помощью замкнутой линии
Gl.glVertex3d(GeomObject[0, 0], GeomObject[0, 1], GeomObject[0, 2]);
Gl.glVertex3d(GeomObject[1, 0], GeomObject[1, 1], GeomObject[1, 2]);
Gl.glVertex3d(GeomObject[2, 0], GeomObject[2, 1], GeomObject[2, 2]);
// завершаем отрисовку примитивов
Gl.glEnd();
// рисуем дополнительные линии у графического объекта
Gl.glBegin( Gl.GL_LINES);
Gl.glVertex3d(GeomObject[0, 0], GeomObject[0, 1], GeomObject[0, 2]);
Gl.glVertex3d(GeomObject[3, 0], GeomObject[3, 1], GeomObject[3, 2]);
Gl.glVertex3d(GeomObject[1, 0], GeomObject[1, 1], GeomObject[1, 2]);
Gl.glVertex3d(GeomObject[3, 0], GeomObject[3, 1], GeomObject[3, 2]);
Gl.glVertex3d(GeomObject[2, 0], GeomObject[2, 1], GeomObject[2, 2]);
Gl.glVertex3d(GeomObject[3, 0], GeomObject[3, 1], GeomObject[3, 2]);
// завершаем отрисовку примитивов
Gl.glEnd();
```

```
// возвращаем состояние матрицы
Gl.glPopMatrix();
// возвращаем состояние матрицы
Gl.glPopMatrix();
// отрисовываем геометрию
Gl.glFlush();
// обновляем состояние элемента
AnT.Invalidate(); }
```

Сами преобразования реализованы с помощью трёх функций, которые будут активированы при обработке соответствующих клавиш. Нажатие клавиш с символами *W / S* вызывает перемещение по указанной оси, клавиш *A / D* – вращение вокруг указанной оси, клавиш *Z / X* – масштабирование.

Обработка данных клавиш клавиатуры и обработка события элемента *comboBox* при изменении его значения для исключения перехвата им нажатия клавиш:

```
// обработка событий от клавиатуры - нажатие (клавиша зажата!)
private void AnT_KeyDown( object sender, Key EventArgs e)
{
// Z и X отвечают за масштабирование
if (e.KeyCode == Keys.Z)
{
// вызов функции, в которой мы реализуем масштабирование -
// передаем коэффициент масштабирования и выбранную ось в окне программы
CreateZoom(1.05f, comboBox1.SelectedIndex);
}
if (e.KeyCode == Keys.X)
{
// вызов функции, в которой мы реализуем масштабирование -
// передаем коэффициент масштабирования и выбранную ось в окне программы
CreateZoom(0.95f, comboBox1.SelectedIndex);
}
// W и S отвечают за перенос
if (e.KeyCode == Keys.W)
{
// вызов функции, в которой мы реализуем перенос -
// передаем коэффициент переноса и выбранную ось в окне программы
CreateTranslate(0.05f, comboBox1.SelectedIndex);
}
if (e.KeyCode == Keys.S)
```

```
{
// вызов функции, в которой мы реализуем перенос -
// передаем коэффициент переноса и выбранную ось в окне программы
CreateTranslate(-0.05f, comboBox1.SelectedIndex);
} // A и D отвечают за поворот
if (e.KeyCode == Keys.A)
{
// вызов функции, в которой мы реализуем поворот -
// передаем угол поворота и выбранную ось в окне программы
CreateRotate(0.05f, comboBox1.SelectedIndex);
}
if (e.KeyCode == Keys.D)
{
// вызов функции, в которой мы реализуем поворот -
// передаем угол поворота и выбранную ось в окне программы
CreateRotate(-0.05f, comboBox1.SelectedIndex);
}
}
// дополнительное событие для элемента comboBox - если произошло
// изменение его значения - установить фокус в элемент AnT
// чтобы избежать перехват события нажатия клавиши данным элементом
private void comboBox1_SelectedIndexChanged( object sender, EventArgs e)
{
// устанавливаем фокус в AnT
AnT.Focus();
}
```

И последнее – реализация поворота, переноса и масштабирования:

```
// функция масштабирования
private void CreateZoom( float coef, int os)
{
// создаем матрицу
float[,] Zoom3D = new float[3, 3]; Zoom3D[0, 0] = 1;
Zoom3D[1, 0] = 0;
Zoom3D[2, 0] = 0;
Zoom3D[0, 1] = 0;
Zoom3D[1, 1] = 1;
Zoom3D[2, 1] = 0;
Zoom3D[0, 2] = 0;
Zoom3D[1, 2] = 0;
Zoom3D[2, 2] = 1;
```

```
// устанавливаем коэффициент масштабирования для необходимой (выбранной и переданной в качестве параметра) оси
Zoom3D[os, os] = coef;
// вызываем функцию для выполнения умножения вектора
// координат вершин геометрического объекта
// на созданную в данной функции матрицу
multiply(GeomObject, Zoom3D);
}
// перенос
private void CreateTranslate( float translate, int os)
{
// для упрощения кода здесь изменена операция переноса -
// последовательное (а не одновременное за одну визуализацию)
// применение разных геометрических преобразований
// дает возможность использовать более простую операцию переноса:
// прибавление константы переноса к координатам объекта по выбранной и переданной оси
for ( int ax = 0; ax < count_elements; ax++)
{
// обновление координат (для выбранной оси)
GeomObject[ax, os] += translate;
} }
// реализация поворота
private void CreateRotate( float angle, int os)
{
// массив, который будет содержать матрицу
float[,] Rotate3D = new float[3, 3];
// в зависимости от оси матрицы будут различаться,
// поэтому создаем необходимую матрицу в зависимости от оси, используя switch
switch (os)
{
case 0: // вокруг оси X
{
Rotate3D[0, 0] = 1;
Rotate3D[1, 0] = 0;
Rotate3D[2, 0] = 0;
Rotate3D[0, 1] = 0;
Rotate3D[1, 1] = (float)Math.Cos(angle);
Rotate3D[2, 1] = (float)-Math.Sin(angle);
```

```
Rotate3D[0, 2] = 0;
Rotate3D[1, 2] = (float)Math.Sin(angle);
Rotate3D[2, 2] = (float)Math.Cos(angle);
break ;
}
case 1: // вокруг оси Y
{
Rotate3D[0, 0] = (float)Math.Cos(angle);
Rotate3D[1, 0] = 0;
Rotate3D[2, 0] = (float)Math.Sin(angle);
Rotate3D[0, 1] = 0;
Rotate3D[1, 1] = 1;
Rotate3D[2, 1] = 0;
Rotate3D[0, 2] = (float)-Math.Sin(angle);
Rotate3D[1, 2] = 0;
Rotate3D[2, 2] = (float)Math.Cos(angle);
break ; }
case 2: // вокруг оси Z
{
Rotate3D[0, 0] = (float)Math.Cos(angle);
Rotate3D[1, 0] = (float)-Math.Sin(angle);
Rotate3D[2, 0] = 0;
Rotate3D[0, 1] = (float)Math.Sin(angle);
Rotate3D[1, 1] = (float)Math.Cos(angle);
Rotate3D[2, 1] = 0;
Rotate3D[0, 2] = 0;
Rotate3D[1, 2] = 0;
Rotate3D[2, 2] = 1;
break ; } }
// вызываем функцию для выполнения умножения вектора координат вершин
// геометрического объекта на созданную в данной функции матрицу
multiply(GeomObject, Rotate3D);
}
// функция умножения матриц
private void multiply(float[,] obj, float[,] matrix)
{
// временные переменные
float res_1, res_2, res_3;
// проходим циклом по всем координатам (это вектор A [x,y,z])
// и умножаем вектор каждой точки на матрицу B (переданную)
// результат сразу заносим в массив геометрии
```

```
for ( int ax = 0; ax < count_elements; ax++)
{
res_1 = (obj[ax, 0] * matrix[0, 0] + obj[ax, 1] * matrix[0, 1] + obj[ax, 2] * matrix[0,
2]);
res_2 = (obj[ax, 0] * matrix[1, 0] + obj[ax, 1] * matrix[1, 1] + obj[ax, 2] * matrix[1,
2]);
res_3 = (obj[ax, 0] * matrix[2, 0] + obj[ax, 1] * matrix[2, 1] + obj[ax, 2] * matrix[2,
2]);
obj[ax, 0] = res_1;
obj[ax, 1] = res_2;
obj[ax, 2] = res_3;
} }
```

Результат работы программы можно увидеть на рис. 4.12. Там изображен двухмерный объект (в данном случае это проекция пирамиды на плоскость), управляемый нажатиями кнопок *W, S, A, D, Z, X* и установкой оси в окне формы.

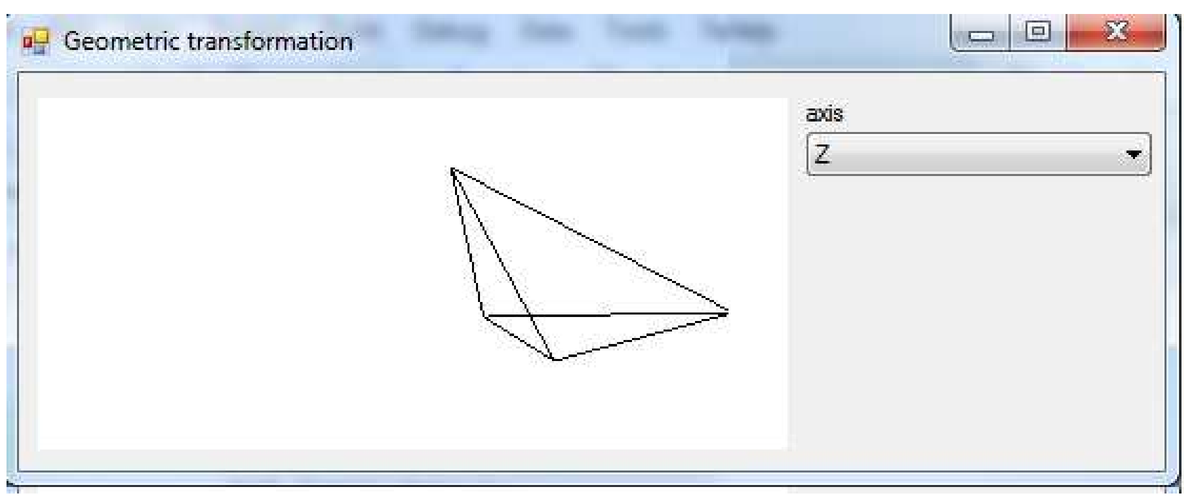


Рис. 4.12

Данное приложение демонстрирует математические основы геометрических преобразований.

При использовании функций *OpenGL* нет необходимости организовывать преобразования в коде приложения – достаточно сохранять в стек матриц текущее состояние объекта, после чего производить преобразования с видовой матрицей, отрисовывать объект и возвращать сохраненную матрицу.

Для реализации операций перемещения, поворота и масштабирования нам будет достаточно использовать имеющиеся в *OpenGL* функции *glTranslate**, glRotate**, glScale***.

Практическое задание

1. Ознакомиться с изложенным выше и представленным в литературе теоретическим материалом.

2. Выполнить действия, приведенные в п. 4.2. При разработке программы имя проекта, создаваемого в *MS Visual Studio*, должно содержать фамилию студента и группу (например, *Ivanov_Ivan_ISG_105_lab_1*).

3. При разработке приложения из п. 4.2 двухмерный объект, начальный угол поворота, константы переноса и масштабирования использовать в соответствии с вашим вариантом.

4. Изменить программу из п. 4.2, введя в нее возможность двухмерных геометрических преобразований не только в плоскости *X*0*Y*, но и в плоскостях *X*0*Z* и *Y*0*Z* для точек, описываемых пространственными координатами $P = (x, y, z)$. У пользователя должна быть возможность выбора третьей оси для переноса и масштабирования – *Z*. Плоский поворот точек объекта при этом должен выполняться вокруг каждой их трех осей по выбору – *X, Y, Z*.

5. Продемонстрировать работу программы на скриншотах (включая реализованные методы).

Скриншот должен включать заголовок окна консоли с полным путем к имени пользователя и папки с проектом.

Папка с программой должна содержать весь проект, выполненный в *MS Visual Studio 2008*, и исполняемые файлы.

Контрольные вопросы

1. Геометрические преобразования в *2D*.

2. Матричные уравнения геометрических преобразований в *2D*.

Тема 5. ФРАКТАЛЫ

Цель изучения темы. Освоение методов построения изображений с использованием фрактальной геометрии, приобретение навыков использования алгоритмического аппарата фракталов при составлении графических программ.

5.1. Фракталы в компьютерной графике

Общие сведения

Фракталы встречаются везде, где заканчиваются правильные формы евклидовой геометрии. Описать форму природных объектов со всеми шероховатостями можно только приблизительно. Здесь на помощь приходят фракталы, которые с большой точностью описывают многие физические явления и природные образования: горы, облака, турбулентные течения, корни, ветви и листья деревьев, кровеносные сосуды, что далеко не соответствует простым геометрическим фигурам.

Понятия фрактал и фрактальная геометрия (*fractus* – состоящий из фрагментов, лат.) были предложены математиком Б. Мандельбротом в 1975 г. для обозначения нерегулярных, но самоподобных структур. Рождение фрактальной геометрии связывают с выходом в 1977 г. книги Б. Мандельброта "Фрактальная геометрия природы", в которой объединены в единую систему научные результаты ученых, работавших в период 1875 – 1925 гг. в этой области (Пуанкаре, Жюлиа, Кантор и др.). Одним из основных свойств фракталов является самоподобие. В самом простом случае небольшая часть фрактала содержит информацию о всем фрактале.

С точки зрения машинной графики фрактальная геометрия незаменима при генерации сложных неевклидовых объектов, образы которых весьма похожи на природные, и когда требуется с помощью нескольких коэффициентов задать линии и поверхности очень сложной формы.

Классификация фракталов

Чтобы представить все многообразие фракталов, удобно прибегнуть к их общепринятой классификации.

Геометрические фракталы – самые наглядные. Их получают с помощью некоторой ломаной линии (в случае *3D*-поверхности), называемой генератором. За один шаг алгоритма каждый из отрезков,

составляющих ломаную, заменяется на ломаную-генератор в соответствующем масштабе. В результате бесконечного повторения этой процедуры получается геометрический фрактал.

Алгебраические фракталы – самая крупная группа фракталов. Получают их с помощью нелинейных процессов в n-мерных пространствах. Наиболее изучены двухмерные процессы.

Известно, что нелинейные динамические системы обладают несколькими устойчивыми состояниями. То состояние, в котором оказалась динамическая система после некоторого числа итераций, зависит от ее начального состояния. Поэтому каждое устойчивое состояние (аттрактор) обладает некоторой областью начальных состояний, из которых система обязательно попадет в рассматриваемые конечные состояния. Таким образом, фазовое пространство системы разбивается на области притяжения аттракторов. Если фазовым является двухмерное пространство, то окрашивая области притяжения различными цветами, можно получить цветовой фазовый портрет этой системы (итерационного процесса). Меняя алгоритм выбора цвета, можно получить сложные фрактальные картины с причудливыми многоцветными узорами. К этому классу фракталов принадлежат множества Мандельброта и Жюлиа.

Стохастические фракталы получаются в том случае, если в итерационном процессе случайным образом менять какие-либо его параметры. При этом получаются объекты очень похожие на природные – несимметричные деревья, изрезанные береговые линии и т.д. Двумерные стохастические фракталы используются при моделировании рельефа местности и поверхности моря.

Системы итерируемых функций обозначают метод получения фрактальных структур в виде системы функций из некоторого фиксированного класса функций, отображающих одно многомерное множество на другое. Наиболее простая система состоит из геометрических преобразований плоскости:

$X' = AX + BY + C$

$Y' = DX + EY + F.$

Существует метод фрактального сжатия и хранения графической информации, который позволяет сжимать информацию в 500 – 1000 раз. В нем изображение кодируется коэффициентами системы итерируемых функций (A, B, C, D, E, F). Например, закодировав какое-то изображение двумя геометрическими преобразованиями, мы однозначно определяем его с помощью двенадцати коэффициентов. Если теперь задаться какой-либо начальной точкой (например, $X = 0, Y = 0$)

и запустить итерационный процесс, то после первой итерации мы получим две точки, после второй – четыре, после третьей – восемь и т.д. Через несколько десятков итераций совокупность полученных точек будет описывать закодированное изображение. Проблема состоит в том, что очень трудно найти коэффициенты системы функций, кодирующей произвольное изображение.

Существуют и другие классификации фракталов, например деление фракталов на детерминированные (алгебраические и геометрические) и недетерминированные (стохастические).

Множества Мандельброта и Жюлиа

Алгоритм построения множества Мандельброта достаточно прост и основан на простом итеративном выражении:

$Z[i+1] = Z[i]\, Z[i] + C,$

где $Z[i]$ и C – комплексные переменные. Итерации выполняются для каждой стартовой точки C прямоугольной области – подмножестве комплексной плоскости. Итерационный процесс продолжается до тех пор, пока $Z[i]$ не выйдет за пределы окружности радиуса 2, центр которой лежит в точке (0, 0) (это означает, что аттрактор динамической системы находится в бесконечности), или после достаточно большого числа итераций (например, 200 – 500) $Z[i]$ сойдется к какой-нибудь точке окружности. В зависимости от количества итераций, в течение которых $Z[i]$ оставалась внутри окружности, можно установить цвет точки C (если $Z[i]$ остается внутри окружности в течение достаточно большого количества итераций, итерационный процесс прекращается, и эта точка растра окрашивается в черный цвет).

Вышеописанный алгоритм дает приближение к множеству Мандельброта (рис. 5.1).

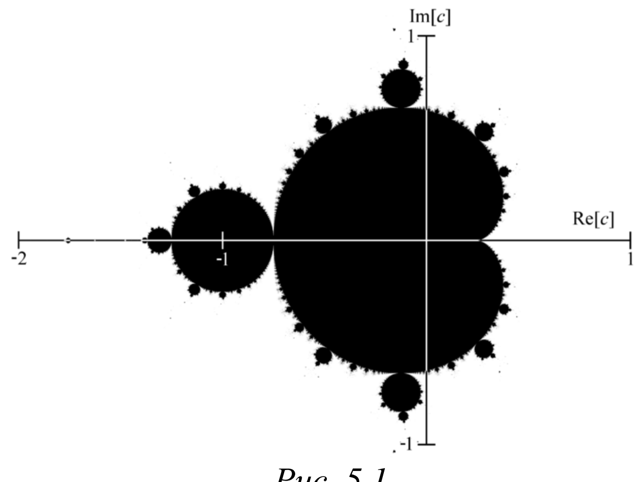


Рис. 5.1

Множеству Мандельброта принадлежат точки (им присваивается черный цвет), которые в течение бесконечного числа итераций не уходят в бесконечность. Точки, принадлежащие границе множества (именно там возникают сложные структуры), уходят в бесконечность за конечное число итераций, а точки, лежащие за пределами множества, уходят в бесконечность через несколько итераций (белый фон).

На рис. 5.2 приведен еще один вариант множества Мандельброта, на рис. 5.3 – увеличенный участок его границы.

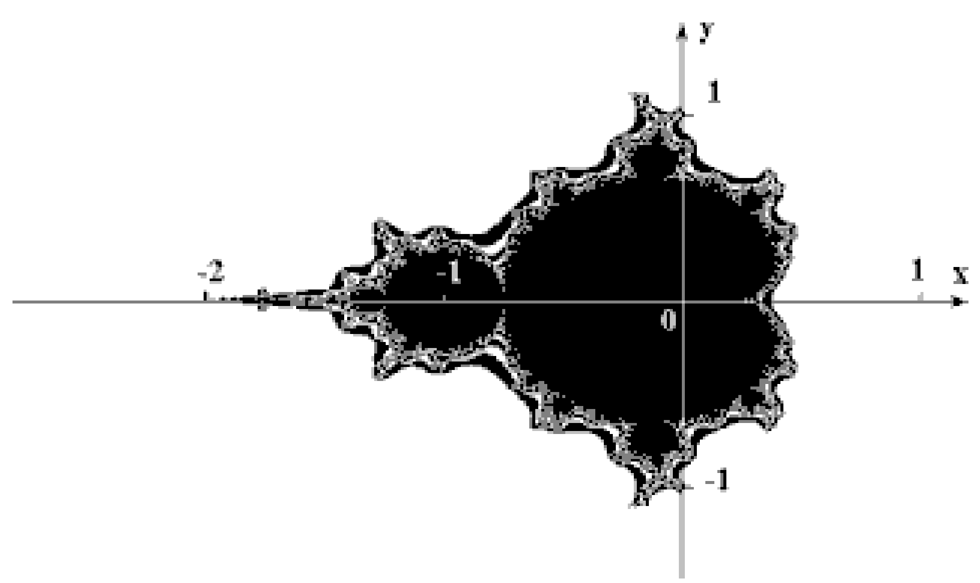


Рис. 5.2

Рис. 5.3

В общем случае рассматриваются процессы с обратной связью, в которых k-я точка (x_k, y_k) порождает $(k+1)$-ю точку (x_{k+1}, y_{k+1}) при помощи закона:

$$x_{k+1} = f(x_k, y_k, p),$$
$$y_{k+1} = g(x_k, y_k, q), \tag{5.1}$$

где p и q – параметры, постоянные в течение каждой итерации $(x_0, y_0) \to$ $\to (x_1, y_1) \to \ldots \to (x_k, y_k) \to \ldots$

Существует две возможности.

Первая состоит в том, что плоскость (x, y) рассматривается при фиксированных p и q. Анализируется динамическая связь точки (x, y) с соответствующим аттрактором. В этом случае мы исследуем структуру областей притяжения и их границ, т.е. множеств Жюлиа.

Вторая возможность состоит в том, что выбирается фиксированная точка (x, y) и прослеживается ее судьба при различных значениях параметров. Результаты наносятся точка за точкой на плоскость (p, q). Таким образом мы получаем изображения типа множества Мандельброта.

Алгоритм построения множества Жюлиа

Будем рассматривать последовательности комплексных чисел $\{Z_n\}$. Возьмем произвольное комплексное число c. Теперь для любого комплексного числа k рассмотрим последовательность $\{Z_n(k)\}$:

$Z_0 = k,$

$Z_{i+1} = Z_i^2 + c.$

Зададимся вопросом: сходится ли Z_n к нулю или стремится к бесконечности при n, стремящемся к бесконечности? Пусть J – множество всех комплексных чисел $\{k\}$, таких, что $\{Z_n(k)\}$ стремится к 0, при n, стремящемся к бесконечности. Если теперь мы возьмем все такие k и отобразим их на комплексной плоскости, то получим множество Жюлиа. Меняя c, мы получим бесконечный набор самоподобных образов – множеств Жюлиа.

Рассмотрим комплексный процесс с обратной связью $z \to z^2 + c$. Запишем комплексные числа z и c в виде $z = x + iy$, $c = p + iq$. Тогда закон, которым описывается процесс, примет вид (5.1):

$$x_{k+1} = x^2_k - y^2_k + p;$$
$$y_{k+1} = 2x_k y_k + q. \tag{5.2}$$

Поскольку бесконечность – всегда аттрактор для такого процесса, зададимся целью раскрасить области притяжения каждой точки. Установим соответствие между цветом и временем, за которое точка (x, y) уходит на бесконечность.

Предположим, что разрешающая способность экрана $a \times b$ точек. Предположим также, что он может быть одновременно раскрашен в $K + 1$ цветов, которые мы пронумеруем от 0 до K, причём числу 0 сопоставим черный цвет.

Шаг 1. Выберем параметр $c = x + iy$.
Выберем $x_{\min} = y_{\min} = -1{,}5$; $x_{\max} = y_{\max} = 1{,}5$; $M = 100$.
Положим $\Delta x = \dfrac{(x_{\max} - x_{\min})}{(a-1)}$, $\Delta y = \dfrac{(y_{\max} - y_{\min})}{(b-1)}$.
Для всех пар (n_x, n_y), где $n_x = (0, \ldots a - 1)$, и $n_y = (0, \ldots b - 1)$, выполним следующую процедуру:

Шаг 2. Положим
$x_0 = x_{\min} + n_x \Delta x$, $y_0 = y_{\min} + n_y \Delta y$, $k = 0$.

Шаг 3. Итерация.
Вычислим (x_{k+1}, y_{k+1}) по (x_k, y_k), используя закон (5.2). Увеличим счетчик k на 1 $(k := k + 1)$.
Шаг 4. Оценка.
Вычислим $r = x^2{}_k + y^2{}_k$:
1) если $r > M$, выберем цвет k; переход к *шагу 5*;
2) если $k = K$, выберем цвет 0 (черный); переход к *шагу 5*;
3) $r \le M$, $k < K$. Возврат к *шагу 3.*

Шаг 5. Припишем цвет k точке экрана (n_x, n_y). Переход к следующей точке, начиная с *шага 2.*

Обычно используют большое число цветов K (например, 200). Если же допустимое количество цветов невелико, удобно использовать их периодическим образом. И даже с помощью двух цветов (белого и черного) можно этим способом получить интересные картины. Варьируя исходные данные на *шаге 1*, можно получить сильно увеличенные изображения.

Алгоритм построения множества Мандельброта

Рассмотрим в общем случае набор множеств Жюлиа и зададимся вопросом: связно ли оно. Пусть M – множество всех множеств Жюлиа, которые связны. Это множество и называется множеством Мандельброта. Теперь возьмем любое множество Жюлиа J и комплексное число c, которое его породило. Если J содержится в M, то

изобразим точку черным на комплексной плоскости, в противном случае – белым. Это простейшее графическое отображение множества Мандельброта.

Есть более легкий путь изображения множества Мандельброта, чем рисование каждого множества Жюлиа и выяснения, связно ли оно. Опять рассмотрим итерационную последовательность для любого k и выясним, сходится ли она к нулю:

$$Z_{i+1} = Z_i^2 + c.$$

Заметим, что c здесь уже не константа. Для любой точки комплексной плоскости присваиваем значение k и выполняем итерации. Этот метод также дает изображение множества Мандельброта. Алгоритм описанного метода:

```
For each point k on the complex plane do:
   let x=0
   repeat infinite times:
      x = x^2 + k
   end repeat
   if x goes to infinity,
      k is not in the set. Color is white.
   else
      k is in the set. Color is black.
```

Чтобы избежать бесконечных циклов, выполним I итераций (I – некоторое большое число; чем больше I, тем точнее ответ). Из практики известно, что число 4000 дает довольно хороший результат. Для сокращения числа циклов воспользуемся эмпирическими данными. Оказывается, что если на любой итерации для некоторого k расстояние от $z_i(k)$ до начала координат окажется больше 2, то можно принять, что данная точка $\{Z_n(k)\}$ уйдет в бесконечность (в алгоритме, чтобы не извлекать квадратный корень, используется сравнение квадрата расстояния с числом 4). Итак, теперь алгоритм выглядит следующим образом:

```
For each point k in the complex plane do:
   let x = 0
   repeat 4000 times
      let x = x^2+k
      if x^2 > 4 then
         Color it white. Break.
   end repeat
   if we reached 4000 then
      Color it black.
```

Этот метод дает черно-белое изображение множества Мандельброта. Теперь надо решить, как сделать его разноцветным.

Если точка принадлежит множеству Мандельброта, то она рисуется черным. Общепринятый способ выбора цвета для других точек – в соответствии с тем, как быстро $\{Z_n(k)\}$ стремится к бесконечности (на какой итерации мы ее исключаем из рассмотрения). Например, точка, для которой расстояние до начала координат больше чем два, уже на третьей итерации должна быть почти белой, а та точка, которая “продержалась“ до 3995 итерации – почти черной. Перепишем алгоритм для изображения в градациях серого:

```
For each point k in the complex plane do:
   let x := 0
   for i := 0 to 4000
      let x = x^2 + k
      if (|x|^2 > 4) then
         Assign for point k color i. Break.
      end if
   end for
   if (i = 4000)
     Assign for point k color black.
   end if.
```

Конечно, просто рисовать точку цветом i нельзя, так как у нас есть только 256 градаций серого, а значение i меняется от 0 до 4000. Надо найти способ отображения i на доступный диапазон цветов.

После того, как получено изображение в градациях серого, легко изменить алгоритм для получения цветного изображения. Например, в изображении в градациях серого, если точка вышла из области на n-й итерации, можно рисовать ее цветом (n, n, n). Возможны и другие варианты, например $(n, 255 - n, 50 \bmod n * 3)$.

Обычно все множество Мандельброта расположено от –2,0 до 0,5 по действительной оси и от –1,25 до 1,25 по мнимой оси.

Доказано, что у процесса (5.2) нет второго (помимо бесконечности) аттрактора, если критическая точка $z_0 = 0$ стремится к бесконечности (z_0 является критической точкой процесса $z_0 \to f(z)$, если $f'(z_0) = 0$). На этом строится алгоритм, с помощью которого получается окрашенное в черный цвет множество Мандельброта с окружностями, раскрашенными в разные цвета.

Разрешающая способность экрана $a \times b$ точек. Он может быть одновременно раскрашен в $K + 1$ цветов, которые мы пронумеруем от 0 до K, причём числу 0 сопоставим черный цвет. Алгоритм:

Шаг 1.
Выберем $p_{\min} = -2{,}25$; $q_{\min} = -1{,}5$; $p_{\max} = 0{,}57$; $q_{\max} = 1{,}5$; $M = 100$.
Положим $\Delta p = \frac{(p_{\max} - p_{\min})}{(a-1)}$, $\Delta q = \frac{(q_{\max} - q_{\min})}{(b-1)}$.
Для всех пар (n_p, n_q), где $n_p = (0, \ldots a - 1)$, и $n_q = (0, \ldots b - 1)$, выполним следующие шаги.
Шаг 2. Положим
$p_0 = p_{\min} + n_p\,\Delta p$, $q_0 = q_{\min} + n_q\,\Delta q$, $k = 0$, $x_0 = y_0 = 0$.
Шаг 3. Итерация.
Вычислим (x_{k+1}, y_{k+1}) по (x_k, y_k), используя закон (5.2). Увеличим счетчик k на 1 $(k := k + 1)$.
Шаг 4. Оценка.
Вычислим $r = x^2_k + y^2_k$:
1) если $r > M$, выберем цвет k; переход к *шагу 5*;
2) если $k = K$, выберем цвет 0 (черный); переход к *шагу 5*;
3) $r \le M$, $k < K$. Возврат к *шагу 3*.
Шаг 5. Припишем цвет k точке экрана (n_p, n_q). Переход к следующей точке, начиная с *шага 2*.

Алгоритм даёт наилучшие результаты, если точка $c = p + iq$ выбрана вблизи границы множества Мандельброта. Следует добиться больших увеличений на этой границе в плоскости (p, q) с учетом того, что чем ближе к границе, тем больше нужно итераций, чтобы определить: уходит или нет критическая точка в бесконечность.

5.2. Применение фрактальной графики

Создадим оболочку программы. В ней будет расположен объект *AnT* для визуализации изображения, элемент *comboBox*1, в котором можно будет выбрать режим отрисовки фрактала, и кнопка для старта визуализации фрактала (рис. 5.4).

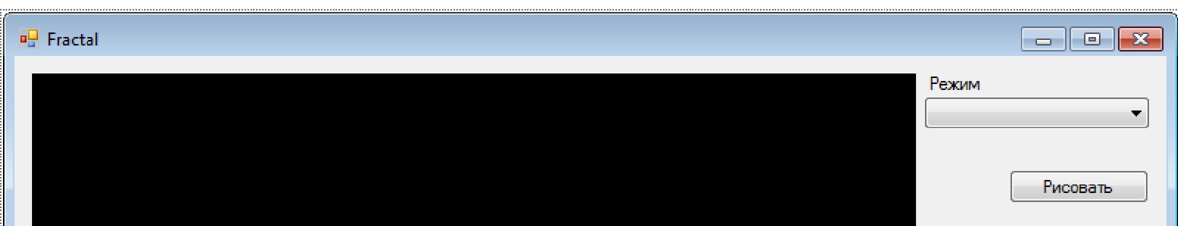


Рис. 5.4

Параметры элементов *comboBox*1 вы можете видеть на рис. 5.5.

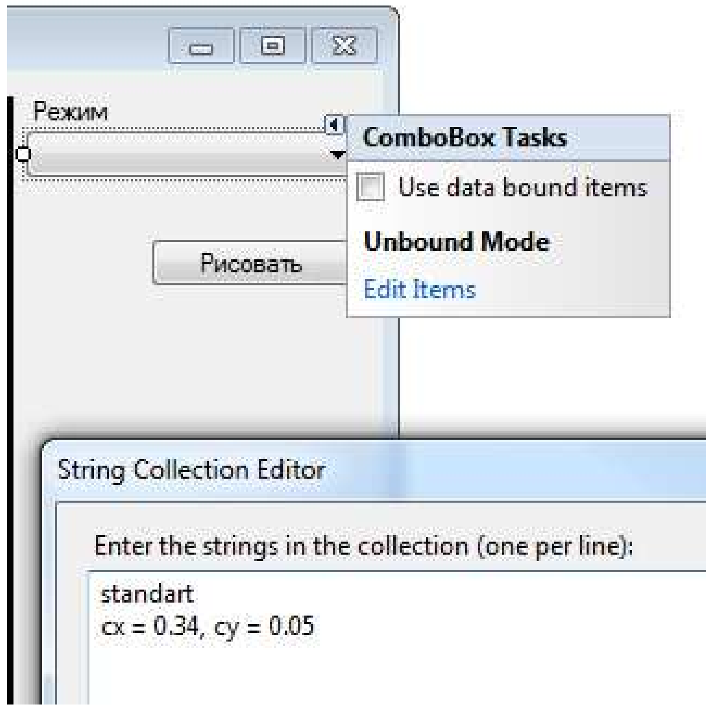


Рис. 5.5

Обратите внимание на то, что элемент *comboBox*1 имеет тип *dropDownList*.

Первоначальное объявление переменных в теле программы выглядит следующим образом:

```
static private double[,] mode_ = new double[5, 2];

// объекты, содержащие настройки для потоков
ParamsForThread[] threadInputParams = new ParamsForThread[8];

// Слово делегат (delegate) используется в C# для обозначения хорошо
// известного понятия. Делегат задает определение функционального типа
// (класса) данных. Экземплярами класса являются функции. Описание делегата
// в языке C# представляет собой описание еще одного частного случая класса.
// Каждый делегат описывает множество функций с заданной сигнатурой.
// Каждая функция (метод), сигнатура которого совпадает с сигнатурой делегата,
// может рассматриваться как экземпляр класса, заданного делегатом.
delegate void RenderDLG();

// массив пикселей
static private byte[,,] PixelsArray = new byte[600, 600, 3];
```

```
// объявляем объекты для управления потоками
Thread th_1 = null;
Thread th_2 = null;
Thread th_3 = null;
Thread th_4 = null;
Thread th_5 = null;
Thread th_6 = null;
Thread th_7 = null;
Thread th_8 = null;
```

Функция инициализации *openGL* реализована даже проще, чем обычно (так как вся визуализация сводится к вызову функции *glDrawPixels*, которая будет визуализировать весь массив точек, представляющих собой результирующее изображение).

```
// инициализация при загрузке формы
private void Form1_Load( object sender, EventArgs e)
{
// инициализация openGL (glut)
Glut.glutInit();
Glut.glutInitDisplayMode( Glut.GLUT_RGB | Glut.GLUT_DOUBLE |
Glut.GLUT_DEPTH);

// цвет очистки окна
Gl.glClearColor(255, 255, 255, 1);

// настройка порта просмотра
Gl.glViewport(0, 0, AnT.Width, AnT.Height);

// очищаем массив пикселей
for ( int ax = 0; ax < 600; ax++)
{
for ( int bx = 0; bx < 600; bx++)
{
PixelsArray[ax, bx, 0] = 0;
PixelsArray[ax, bx, 1] = 0;
PixelsArray[ax, bx, 2] = 0;
} }
// устанавливаем первый элемент в comboBox
comboBox1.SelectedIndex = 0;
// и добавляем 1 режим в массив _mode
mode_[1, 0] = 0.34;
mode_[1, 1] = 0.05;
}
```

Функция *Draw*, а также отклик таймера реализуют вызов функции *glDrawPixels*:

```
// функция отрисовки
private void Draw()
{
// устанавливаем позицию вывода
Gl.glRasterPos2i(-1, -1);
// визуализируем массив
Gl.glDrawPixels(600, 600, Gl.GL_RGB, Gl.GL_UNSIGNED_BYTE, PixelsArray);
Gl.glFlush();
AnT.Invalidate();
}
// отклик таймера
private void RenderTimer_Tick( object sender, EventArgs e)
{
Gl.glRasterPos2i(-1, -1);
Gl.glDrawPixels(600, 600, Gl.GL_RGB, Gl.GL_UNSIGNED_BYTE, PixelsArray);
Gl.glFlush();

AnT.Invalidate();
}
```

Класс *ParamsForThread* будет иметь ряд полей, которые будут хранить в себе необходимые данные для работы потока – каждый поток вычисляет небольшую часть изображения.

```
// данный класс - это набор параметров, которые мы будем передавать в поток
public class ParamsForThread
{
public ParamsForThread( int startH, int endH, int Width)
{
_FromImageH = startH;
_ToImageH = endH;
_ImageW = Width;
}

// элемент, установленный в comboBox1
public int code_mode;

public delegate void _RenderDLG();
// указатель на функцию отрисовки
public _RenderDLG _pointerToDraw = null;
```

```
// параметры части изображения, которое будет рассчитываться в данном потоке
public int _FromImageH;
public int _ToImageH;
public int _ImageW;
}
```

Теперь рассмотрим функцию, реализующую старт отрисовки, – здесь все основано на уже изученных нами методах многопоточного программирования: мы создаем потоки, передаем в них информацию (объекты *ParamsForThread*), настраиваем приоритет и запускаем на выполнение.

```
private void button1_Click( object sender, EventArgs e)
{
// делим строки обрабатываемого изображения между потоками
threadInputParams[0] = new ParamsForThread(0, 75, 600);
threadInputParams[1] = new ParamsForThread(75, 150, 600);
threadInputParams[2] = new ParamsForThread(150, 225, 600);
threadInputParams[3] = new ParamsForThread(225, 300, 600);
threadInputParams[4] = new ParamsForThread(300, 375, 600);
threadInputParams[5] = new ParamsForThread(375, 450, 600);
threadInputParams[6] = new ParamsForThread(450, 525, 600);
threadInputParams[7] = new ParamsForThread(525, 600, 600);

threadInputParams[0]._pointerToDraw = new ParamsForThread._RenderDLG(Draw);
threadInputParams[1]._pointerToDraw = new ParamsForThread._RenderDLG(Draw);
threadInputParams[2]._pointerToDraw = new ParamsForThread._RenderDLG(Draw);
threadInputParams[3]._pointerToDraw = new ParamsForThread._RenderDLG(Draw);
threadInputParams[4]._pointerToDraw = new ParamsForThread._RenderDLG(Draw);
threadInputParams[5]._pointerToDraw = new ParamsForThread._RenderDLG(Draw);
threadInputParams[6]._pointerToDraw = new ParamsForThread._RenderDLG(Draw);
threadInputParams[7]._pointerToDraw = new ParamsForThread._RenderDLG(Draw);

threadInputParams[0].code_mode = comboBox1.SelectedIndex;
threadInputParams[1].code_mode = comboBox1.SelectedIndex;
threadInputParams[2].code_mode = comboBox1.SelectedIndex;
threadInputParams[3].code_mode = comboBox1.SelectedIndex;
threadInputParams[4].code_mode = comboBox1.SelectedIndex;
threadInputParams[5].code_mode = comboBox1.SelectedIndex;
threadInputParams[6].code_mode = comboBox1.SelectedIndex;
threadInputParams[7].code_mode = comboBox1.SelectedIndex;

/*создаем 8 потоков, в качестве параметров передаем имя выполняемой функции*/
th_1 = new Thread(CalculateImage);
th_2 = new Thread(CalculateImage);
```

```
th_3 = new Thread(CalculateImage);
th_4 = new Thread(CalculateImage);
th_5 = new Thread(CalculateImage);
th_6 = new Thread(CalculateImage);
th_7 = new Thread(CalculateImage);
th_8 = new Thread(CalculateImage);

//расставляем приоритеты для потоков ниже среднего
th_1.Priority = ThreadPriority.Lowest;
th_2.Priority = ThreadPriority.Lowest;
th_3.Priority = ThreadPriority.Lowest;
th_4.Priority = ThreadPriority.Lowest;
th_5.Priority = ThreadPriority.Lowest;
th_6.Priority = ThreadPriority.Lowest;
th_7.Priority = ThreadPriority.Lowest;
th_8.Priority = ThreadPriority.Lowest;

/*запускаем каждый поток, в качестве параметра передаем выводимый символ,
этот символ передастся в качестве параметра в функцию WriteString*/

RenderTimer.Start();

th_1.Start(threadInputParams[0]);
th_2.Start(threadInputParams[1]);
th_3.Start(threadInputParams[2]);
th_4.Start(threadInputParams[3]);
th_5.Start(threadInputParams[4]);
th_6.Start(threadInputParams[5]);
th_7.Start(threadInputParams[6]);
th_8.Start(threadInputParams[7]);
}
```

И остается рассмотреть только функцию вычисления изображения фрактала.

```
// функция вычисления изображения
static void CalculateImage( object Settings)
{
// получаем параметры через объект Settings, приводя его к типу
ParamsForThread
ParamsForThread thisThreadSettings = (ParamsForThread)Settings;

// инициализация начальных значений переменных
double xmin = -3;
double ymin = -2;
double xmax = 1;
```

```
double ymax = 2;
int W = 600;
int H = 600;

double dx = (xmax - xmin) / (double)(W - 1);
double dy = (ymax - ymin) / (double)(H - 1);

double x, y, X, Y, Cx, Cy;

// циклы по всем пикселям результирующего изображения
for ( int ax = thisThreadSettings._FromImageH; ax < thisThreadSettings._ToImageH;
ax++)
{
for ( int bx = 0; bx < thisThreadSettings._ImageW; bx++)
{
// подготовка к выполнению итерации

x = xmin + ax * dx;
y = ymin + bx * dy;

Cx = x;
Cy = y;
X = x;
Y = y;

double ix = 0, iy = 0, n = 0;

// выполнение итерации
while ((ix * ix + iy * iy < 4) && (n < 64))
{
// если режим установлен - standart
if( thisThreadSettings.code_mode == 0)
{
ix = X * X - Y * Y + Cx;
iy = 2 * X * Y + Cy;
}
else
{
// иначе используем значения из массива _mode, используя индекс comboBox1,
// переданный в поток через thisThreadSettings
ix = X * X - Y * Y + mode_[thisThreadSettings.code_mode,0];
iy = 2 * X * Y + mode_[thisThreadSettings.code_mode,1];
}

n++;
X = ix;
Y = iy;
}
```

```
// заносим значение цвета в массив, описывающий результирующее изображение
PixelsArray[bx, ax, 0] = (byte)(255 - n * 4);
PixelsArray[bx, ax, 1] = (byte)(255 - n * 4);
PixelsArray[bx, ax, 2] = (byte)(255 - n * 4);

// вызываем функцию отрисовки
thisThreadSettings._pointerToDraw();
} } }
```

Результаты работы программы при

```
double xmin = -2.0;
double ymin = -1.0;
double xmax = 0.0;
double ymax = 1.0
```

приведены на рис. 5.6.

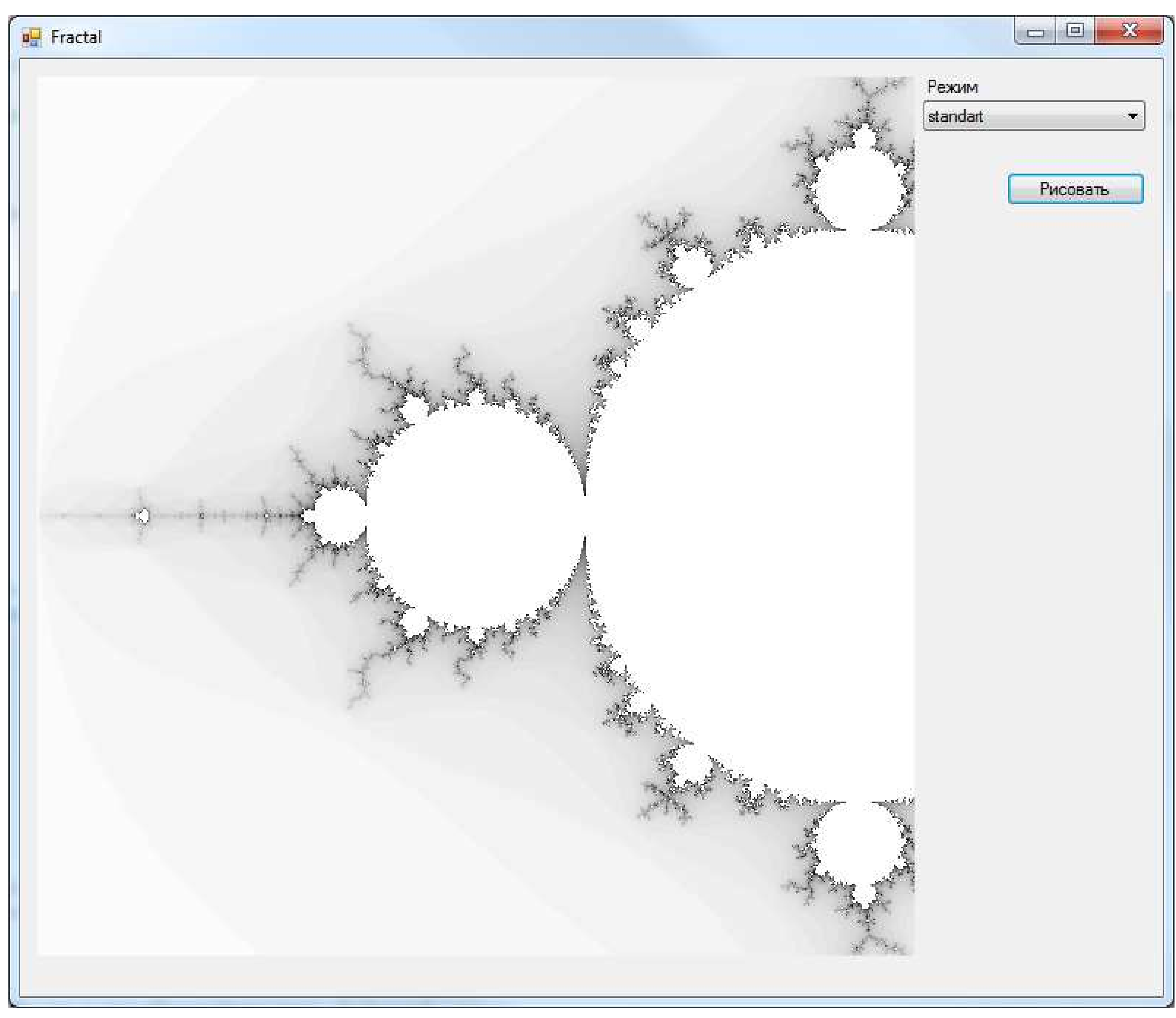


Рис. 5.6

Результаты работы программы при
double xmin = -0.5;
double ymin = -0.5;
double xmax = 0.5;
double ymax = 0.5
приведены на рис. 5.7.

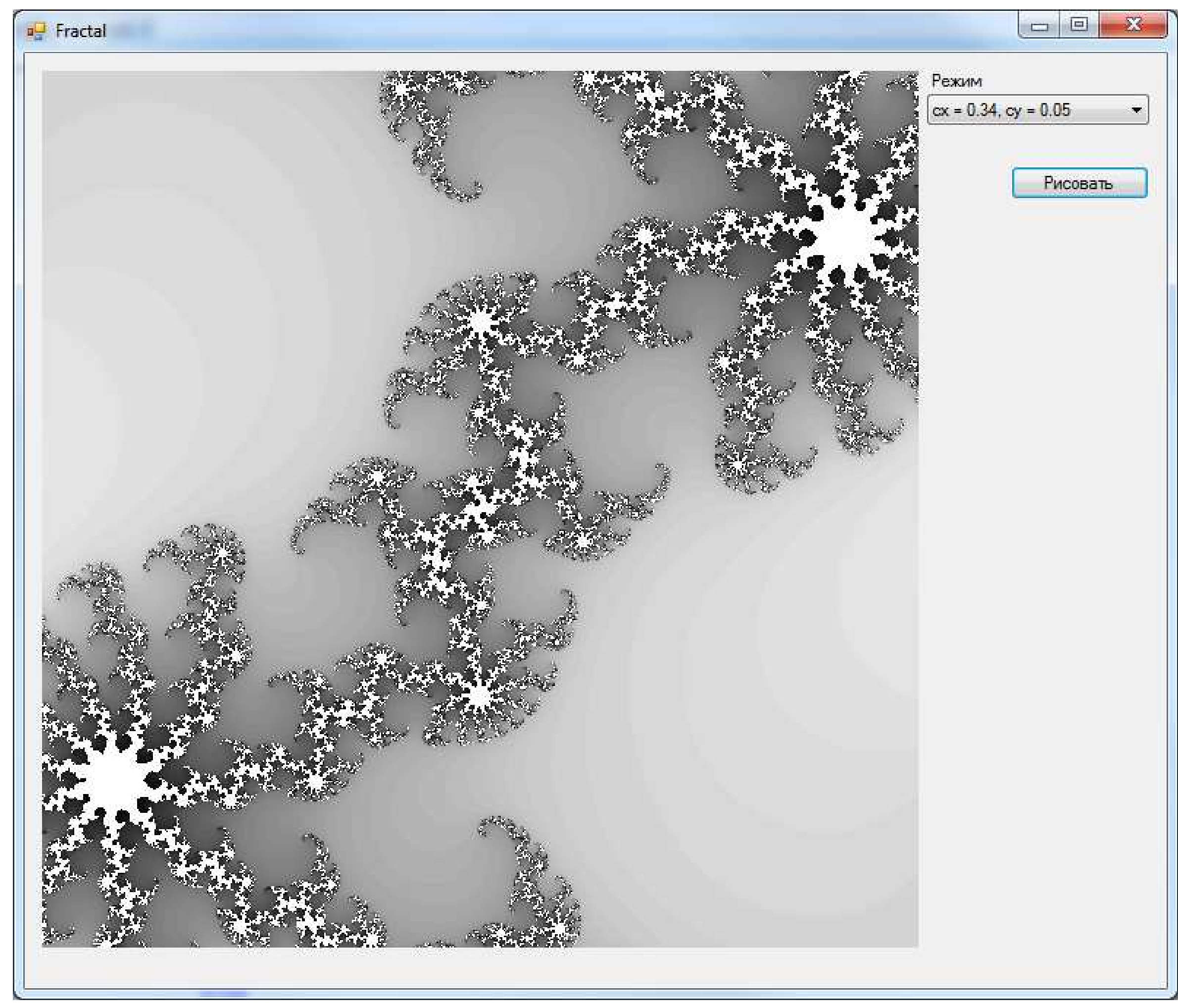


Рис. 5.7

Практическое задание

1. Ознакомиться с изложенным выше и представленным в литературе теоретическим материалом.

2. Выполнить действия, приведенные в п. 5.2. При разработке программы имя проекта, создаваемого в *MS Visual Studio*, должно содержать фамилию студента и группу (например, *Ivanov_Ivan_ISG_105_lab_1*).

3. При выполнении работы изменить программу, применив к фракталу параметры в соответствии с вашим вариантом.

4. Дополнить программу возможностью отрисовки множества Жулиа. Параметры множества использовать в соответствии с вашим вариантом.

Продемонстрировать работу программы на скриншотах (включая реализованные методы).

Скриншот должен включать заголовок окна консоли с полным путем к имени пользователя и папки с проектом.

Папка с программой должна содержать весь проект, выполненный в *MS Visual Studio 2008* и исполняемые файлы.

Контрольные вопросы

1. Понятие и виды фракталов.
2. Применение фракталов в компьютерной графике.

ЗАКЛЮЧЕНИЕ

В учебном пособии рассматривались базовые алгоритмы и инструментальные средства программирования двухмерной компьютерной графики, применяемые при решении реальных практических задач.

Алгоритмам и методам реализации прикладного программного обеспечения, формирующего объемные изображения, будет посвящено следующее пособие «Программирование трехмерной компьютерной графики», планируемое к изданию авторами.

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. *Жигалов, И. Е.* 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.

2. *Жигалов, И. Е.* Компьютерная графика : курс лекций / И. Е. Жигалов ; Владим. гос. ун-т. – Владимир, 2004. – 124 с.

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

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

5. *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.

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

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

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

9. *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.

10. *Перемитина, Т. О.* Компьютерная графика : учеб. пособие / Т. О. Перемитина. – Томск : Эль Контент, 2012. – 144 с. – ISBN 978-5-4332-0077-7.

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ 3

Тема 1. ВВЕДЕНИЕ В ПРОГРАММИРОВАНИЕ *2D* ГРАФИКИ 4
1.1. Разработка растрового редактора 4
1.2. Растровый редактор: инструменты 19
1.3. Растровый редактор: система слоев 25
1.4. Растровый редактор: оболочка программы 35
1.5. Растровый редактор: оптимизация функций 43

Тема 2. АЛГОРИТМЫ ОБРАБОТКИ РАСТРОВЫХ ИЗОБРАЖЕНИЙ 51
2.1. Введение в алгоритмы обработки растровых изображений 51
2.2. Реализация фильтров 55

Тема 3. СПЛАЙНЫ 62
3.1. Сплайны в компьютерной графике 62
3.2. Построение В-сплайна 64

Тема 4. ГЕОМЕТРИЧЕСКИЕ ПРЕОБРАЗОВАНИЯ В *2D* 75
4.1. Введение в геометрические преобразования 75
4.2. Программа геометрических преобразований в *2D* 79
4.3. Программа *2D* геометрических преобразований (добавление оси *Z*) 89

Тема 5. ФРАКТАЛЫ 100
5.1. Фракталы в компьютерной графике 100
5.2. Применение фрактальной графики 108

ЗАКЛЮЧЕНИЕ 117

БИБЛИОГРАФИЧЕСКИЙ СПИСОК 118

Учебное издание

ЖИГАЛОВ Илья Евгеньевич
НОВИКОВ Иван Андреевич

ПРОГРАММИРОВАНИЕ ДВУХМЕРНОЙ
КОМПЬЮТЕРНОЙ ГРАФИКИ

Учебное пособие

Редакторы А. А. Амирсейидова, Е. В. Невская
Технический редактор С. Ш. Абдуллаева
Корректор В. С. Теверовский
Компьютерная верстка Е. А. Балясовой

Подписано в печать 29.10.15.
Формат 60×84/16. Усл. печ. л. 6,98. Тираж 80 экз.
Заказ
Издательство
Владимирского государственного университета
имени Александра Григорьевича и Николая Григорьевича Столетовых.
600000, Владимир, ул. Горького, 87.