Министерство образования и науки Российской Федерации

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

И. Е. ЖИГАЛОВ
И. А. НОВИКОВ

ПРОГРАММИРОВАНИЕ ТРЕХМЕРНОЙ КОМПЬЮТЕРНОЙ ГРАФИКИ

Учебное пособие

Владимир 2016

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

Жигалов, И. Е.

Ж68 Программирование трехмерной 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, 2016. – 92 с.

ISBN 978-5-9984-0685-0

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

Предназначено для студентов направлений 09.03.02, 09.04.02 – Информационные системы и технологии и 09.03.04, 09.04.04 – Программная инженерия по дисциплинам «Программирование компьютерной графики», «Программирование графических приложений».

Рекомендовано для формирования профессиональных компетенций в соответствии с ФГОС ВО.

Ил. 35. Библиогр.: 11 назв.

УДК 004.92
ББК 32.973.26-018

ISBN 978-5-9984-0685-0

© ВлГУ, 2016

ВВЕДЕНИЕ

В пособии рассматриваются важные для изучения компьютерной графики темы по программированию алгоритмов построения трехмерных графических объектов. Оно является логическим продолжением учебного пособия «Программирование двухмерной компьютерной графики», в котором были представлены вопросы изучения программирования алгоритмов в среде C# в MS VisualStudio с использованием библиотеки *OpenGL*.

В пособии 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 трехмерной графики и поясняют методику использования инструментальных средств для получения желаемого визуального эффекта.

Пособие включает такие темы трехмерной компьютерной графики, как «Тела вращения», «Преобразование объектов с использованием библиотеки GLUT», «Текстуры», «Описание *3D*-объектов», «Системы частиц» и помогает освоить средства разработки графического прикладного программного обеспечения.

Тема 1. **ТЕЛА ВРАЩЕНИЯ**

Цель изучения темы. Изучение методов формирования моделей объемных объектов в виде тел вращения, способов построения изображений на основе таких моделей, приобретение навыков использования тел вращения при составлении графических программ.

1.1. Тела вращения в компьютерной графике

Общие сведения

Один из способов формирования сложных объемных изображений на экране – применение скелетных тел вращения. При этом выбирается ось вращения и в одной плоскости с ней формируется кривая, аппроксимируемая обычно последовательностью отрезков прямых линий. В результате вращения кривой вокруг выбранной оси и фиксации положения этой кривой через определенные отсчеты угла вращения создается скелетное тело вращения.

Кодирования точек на поверхности тела вращения

Имея некоторую геометрию кривой, заданную рядом точек, мы можем построить объект, основываясь на повороте геометрии данной кривой. Разбив объект таким образом на N данных кривых и соединив их вершины, образовывая полигоны между кривыми, мы получаем оболочку тела вращения (рис. 1.1).

Количество отрезков исходной кривой и количество разбиений определяют гладкость объекта. Соответственно большое количество полигонов увеличивает нагрузку на графический адаптер.

Для построения трехмерного объекта с оболочкой также будет необходимо рассчитать нормали для определения полигонов.

1.2. Построение тел вращения

Разработка программы начинается с создания оболочки. Создайте окно программы и разместите на ней элемент *openglsimplecontrol*, как

показано на рис. 1.2, после чего установите его размеры 500 × 500. Переименуйте данный объект, назвав его *AnT*.

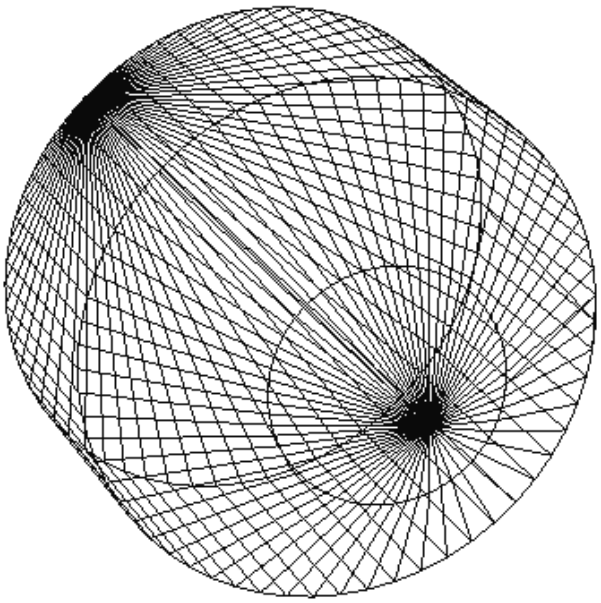

Рис. 1.1

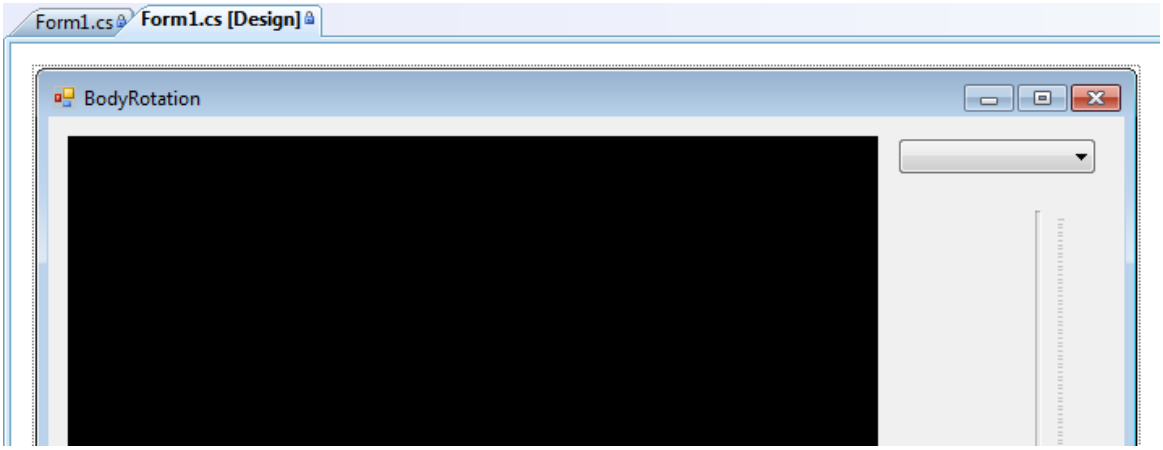

Рис. 1.2

Справа от данного элемента разместите элемент *comboBox*, после чего в его свойствах установите значение параметра *DropDownStyle* = *DropDownList*. После этого выпадающие элементы перестанут быть доступными для редактирования. Затем измените элементы *Items*, как показано на рис. 1.3.

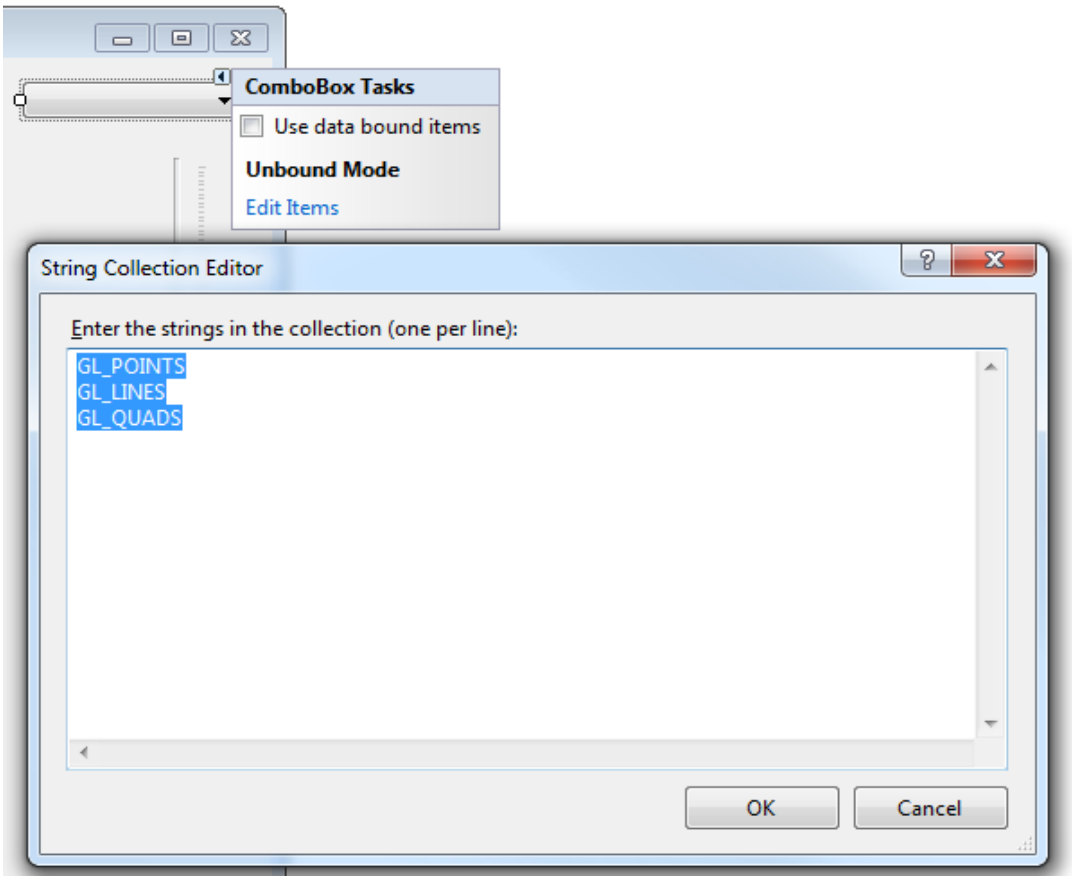

Рис. 1.3

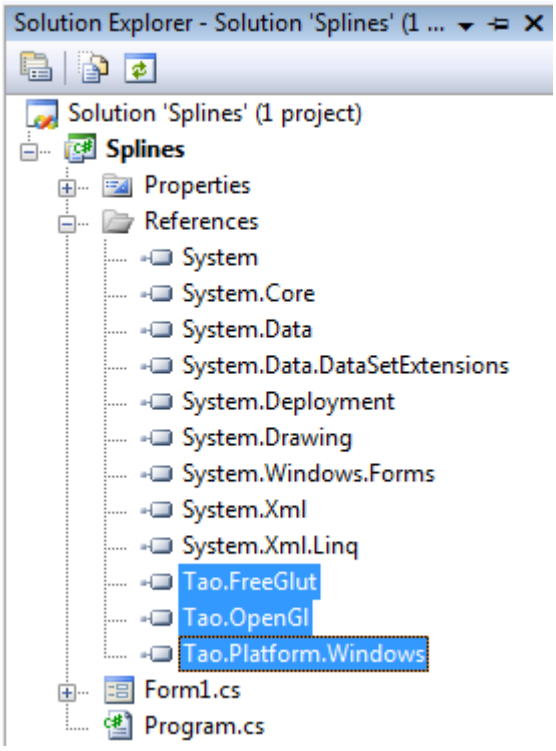

Рис. 1.4

Помимо этого установите элемент *trackBar* в окно формы. Перейдя к его свойствам, установите *orientation* равным *Vertical*. Максимальный диапазон установите равным ста.

Не забудьте установить ссылки на используемые библиотеки *Tao* (рис. 1.4).

Для реализации визуализации будет использоваться таймер, после инициализации окна он будет генерировать событие, называемое "тиком" таймера, раз в 30 мс добавьте элемент таймер, переименуйте экзем-

пляр в *RenderTimer* и установите время "тика" 30 мс (как показано на рис. 1.5), а также добавьте ему событие для обработки "тика".

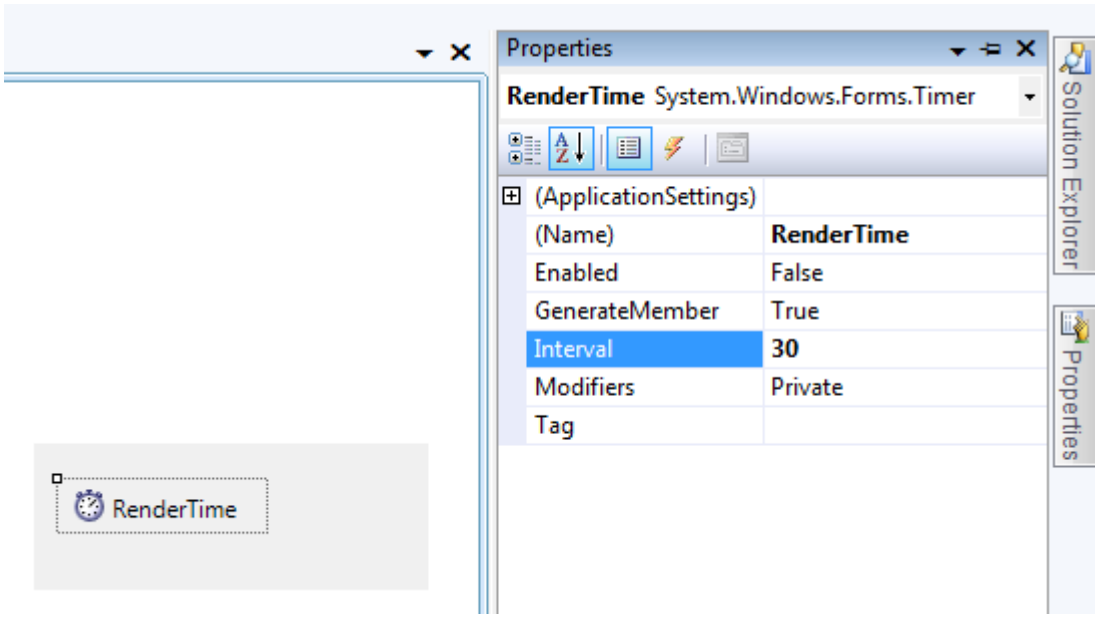

Рис. 1.5

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

Нам потребуется объявить ряд переменных для дальнейшей работы программы:

```
private int lastX, lastY; private float rot_1, rot_2;
private double[,] GeometricArray = new double[64,3];
private double[, ,] ResaultGeometric = new double[64, 64, 3];
private int init_mode = 0;
private int count_elements = 0;
private double Angle = 2*Math.PI / 64;
private int Iter = 64;
public Form1()
{
InitializeComponent();
AnT.InitializeContexts();
}
```

Как и раньше, функция *Form1_Load* отвечает за инициализацию *OpenGL*. Но теперь помимо этого здесь происходит построение массива геометрии тела, построенного вращением на основе заданного заранее массива *GeometricArray*.

Код этой функции подробно комментирован.

```
private void Form1_Load( object sender, EventArgs e)
{
// инициализация Glut
Glut.glutInit();
Glut.glutInitDisplayMode( Glut.GLUT_RGB | Glut.GLUT_DOUBLE |
Glut.GLUT_DEPTH);
// очистка окна
Gl.glClearColor(255, 255, 255, 1);
// установка порта вывода в соответствии с размерами элемента AnT
Gl.glViewport(0, 0, AnT.Width, AnT.Height);
// настройка проекции
Gl.glMatrixMode( Gl.GL_PROJECTION);
Gl.glLoadIdentity();
Glu.gluPerspective(45, (float)AnT.Width / (float)AnT.Height, 0.1, 200);
Gl.glMatrixMode( Gl.GL_MODELVIEW);
Gl.glLoadIdentity();
// настройка параметров OpenGL для визуализации
Gl.glEnable( Gl.GL_DEPTH_TEST);
Gl.glEnable( Gl.GL_LIGHTING);
Gl.glEnable( Gl.GL_LIGHT0);
// количество элементов последовательности геометрии, на основе которых бу-
дет строиться тело вращения
count_elements = 8;
// непосредственное заполнение точек
// после изменения данной геометрии мы сразу получим новое тело вращения
GeometricArray[0,0] = 0;
GeometricArray[0,1] = 0;
GeometricArray[0,2] = 0;
GeometricArray[1,0] = 0.7;
GeometricArray[1,1] = 0;
GeometricArray[1,2] = 1;
GeometricArray[2, 0] = 1.3;
GeometricArray[2, 1] = 0;
GeometricArray[2, 2] = 2;
GeometricArray[3,0] = 1.0;
GeometricArray[3,1] = 0;
GeometricArray[3,2] = 3;
GeometricArray[4, 0] = 0.5;
GeometricArray[4, 1] = 0;
GeometricArray[4, 2] = 4;
GeometricArray[5, 0] = 3;
GeometricArray[5, 1] = 0;
```

```
GeometricArray[5, 2] = 6;
GeometricArray[6, 0] = 1;
GeometricArray[6, 1] = 0;
GeometricArray[6, 2] = 7;
GeometricArray[7, 0] = 0;
GeometricArray[7, 1] = 0;
GeometricArray[7, 2] = 7.2f;
// по умолчанию мы будем отрисовывать фигуру в режиме GL_POINTS
comboBox1.SelectedIndex = 0;
// построение геометрии тела вращения
// принцип сводится к двум циклам, на основе первого перебираются
// вершины в геометрической последовательности
// второй использует параметр Iter – производит поворот последней линии геометрии вокруг центра тела вращения
// при этом используется заранее определенный угол angle, который определяется как 2*Pi / количество меридиан объекта
// за счет выполнения этого алгоритма получается набор вершин, описывающих оболочку тела вращения
// остается только соединить эти точки в режиме рисования примитивов для получения
// визуализированного объекта
// цикл по последовательности точек кривой, на основе которой будет построено тело вращения
for ( int ax = 0; ax < count_elements; ax++)
{
// цикл по меридианам объекта, заранее определенным в программе
for ( int bx = 0; bx < Iter; bx++)
{
// для всех (bx > 0) элементов алгоритма используется предыдущая построенная последовательность
// для ее поворота на установленный угол
if (bx > 0)
{
double new_x = ResaultGeometric[ax, bx - 1, 0] * Math.Cos(Angle) - ResaultGeometric[ax, bx - 1, 1] * Math.Sin(Angle);
double new_y = ResaultGeometric[ax, bx - 1, 0] * Math.Sin(Angle) + ResaultGeometric[ax, bx - 1, 1] * Math.Cos(Angle);
ResaultGeometric[ax, bx, 0] = new_x;
ResaultGeometric[ax, bx, 1] = new_y;
ResaultGeometric[ax, bx, 2] = GeometricArray[ax, 2];
}
else // для построения первого меридиана мы используем начальную кривую, описывая ее нулевым значением угла поворота
```

```
{
double new_x = GeometricArray[ax, 0] *Math.Cos(0) - GeometricArray[ax, 1] *
Math.Sin(0);
double new_y = GeometricArray[ax, 1] *Math.Sin(0) +
GeometricArray[ax, 1] * Math.Cos(0);
ResaultGeometric[ax, bx, 0] = new_x;
ResaultGeometric[ax, bx, 1] = new_y;
ResaultGeometric[ax, bx, 2] = GeometricArray[ax, 2];
}
}
}
// активация таймера
RenderTimer.Start();
}
```

Итак, геометрия объекта построена, остается обработать сообщение таймера для вызова функции отрисовки, а также реализовать непосредственно функцию *Draw*.

В функции *Draw* мы рассмотрим три вида визуализации, которые будут использованы в зависимости от установленного режима в элементе *comboBox*. Визуализация с помощью точек самая простая.

Визуализация с помощью линий или полигонов уже сложнее. Постарайтесь максимально разобрать алгоритм, чтобы понять суть его работы.

```
// функция обработки сообщения таймера
private void RenderTimer_Tick( object sender, EventArgs e)
{
// вызываем функцию, отвечающую за отрисовку сцены
Draw();
}
// функция отрисовки сцены
private void Draw()
{
// два параметра, которые мы будем использовать для непрерывного вращения
сцены вокруг двух координатных осей
rot_1++;
rot_2++; // 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 положения камеры (наблюдателя). Как видно из кода,
// дополнительно на положение наблюдателя по оси Z влияет значение,
// установленное в ползунке, доступном для пользователя

// таким образом, при перемещении ползунка наблюдатель будет отдаляться или приближаться к объекту наблюдения
Gl.glTranslated(0, 0, -7 -trackBar1.Value);
// 2 поворота (углы rot_1 и rot_2)
Gl.glRotated(rot_1, 1, 0, 0);
Gl.glRotated(rot_2, 0, 1, 0);

// устанавливаем размер точек, равный 5
Gl.glPointSize(5.0f);

// условие switch определяет установленный режим отображения на основе выбранного пункта
// элемента comboBox, установленного в форме программы
switch (comboBox1.SelectedIndex)
{
case 0: // отображение в виде точек
{
// режим вывода геометрии – точки
Gl.glBegin( Gl.GL_POINTS);

// выводим всю ранее просчитанную геометрию объекта
for ( int ax = 0; ax < count_elements; ax++)
{
for ( int bx = 0; bx < Iter; bx++)
{
// отрисовка точки
Gl.glVertex3d(ResaultGeometric[ax, bx, 0], ResaultGeometric[ax, bx, 1], ResaultGeometric[ax, bx, 2]);
}
}
// завершаем режим рисования
Gl.glEnd();

break ;
}
case 1: // отображение объекта в сеточном режиме, используя режим GL_LINES_STRIP
{
// устанавливаем режим отрисовки линиями (последовательность линий)
```

```
Gl.glBegin( Gl.GL_LINE_STRIP);
for ( int ax = 0; ax < count_elements; ax++)
{
for ( int bx = 0; bx < Iter; bx++)
{

Gl.glVertex3d(ResaultGeometric[ax, bx, 0], ResaultGeometric[ax, bx, 1], ResaultGeometric[ax, bx, 2]);
Gl.glVertex3d(ResaultGeometric[ax + 1, bx, 0], ResaultGeometric[ax + 1, bx, 1], ResaultGeometric[ax + 1, bx, 2]);

if (bx + 1 < Iter - 1)
{
Gl.glVertex3d(ResaultGeometric[ax + 1, bx + 1, 0], ResaultGeometric[ax + 1, bx + 1, 1], ResaultGeometric[ax + 1, bx + 1, 2]);
}
else
{
Gl.glVertex3d(ResaultGeometric[ax + 1, 0, 0], ResaultGeometric[ax + 1, 0, 1], ResaultGeometric[ax + 1, 0, 2]);
}
}
}
Gl.glEnd();
break ;
}
case 2: // отрисовка оболочки с расчетом нормалей для корректного затенения граней объекта
{
Gl.glBegin( Gl.GL_QUADS); // режим отрисовки полигонов, состоящих из 4 вершин
for ( int ax = 0; ax < count_elements; ax++)
{
for ( int bx = 0; bx < Iter; bx++)
{
// вспомогательные переменные для более наглядного использования кода при расчете нормалей
double x1 = 0, x2 = 0, x3 = 0, x4 = 0, y1 = 0, y2 = 0, y3 = 0, y4 = 0, z1 = 0, z2 = 0, z3 = 0, z4 = 0;

// первая вершина
x1 = ResaultGeometric[ax, bx, 0];
y1 = ResaultGeometric[ax, bx, 1];
```

```
z1 = ResaultGeometric[ax, bx, 2];

if (ax + 1 < count_elements) // если текущий ax не последний
{
// берем следующую точку последовательности
x2 = ResaultGeometric[ax + 1, bx, 0];
y2 = ResaultGeometric[ax + 1, bx, 1];
z2 = ResaultGeometric[ax + 1, bx, 2];

if (bx + 1 < Iter - 1) // если текущий bx не последний
{
// берем следующую точку последовательности и следующий меридиан
x3 = ResaultGeometric[ax + 1, bx + 1, 0];
y3 = ResaultGeometric[ax + 1, bx + 1, 1];
z3 = ResaultGeometric[ax + 1, bx + 1, 2];

// точка, соответствующая по номеру только на соседнем меридиане
x4 = ResaultGeometric[ax, bx + 1, 0];
y4 = ResaultGeometric[ax, bx + 1, 1];
z4 = ResaultGeometric[ax, bx + 1, 2];
}
else
{
// если это последний меридиан, то в качестве следующего мы берем начальный
(замыкаем геометрию фигуры)
x3 = ResaultGeometric[ax + 1, 0, 0];
y3 = ResaultGeometric[ax + 1, 0, 1];
z3 = ResaultGeometric[ax + 1, 0, 2];

x4 = ResaultGeometric[ax, 0, 0];
y4 = ResaultGeometric[ax, 0, 1];
z4 = ResaultGeometric[ax, 0, 2];
}
}
else // данный элемент ax последний, следовательно мы будем использовать
начальный (нулевой) вместо данного ax
{
// следуещей точкой будет нулевая ax
x2 = ResaultGeometric[0, bx, 0];
y2 = ResaultGeometric[0, bx, 1];
z2 = ResaultGeometric[0, bx, 2];
```

```
if (bx + 1 < Iter - 1)
{
x3 = ResaultGeometric[0, bx + 1, 0];
y3 = ResaultGeometric[0, bx + 1, 1];
z3 = ResaultGeometric[0, bx + 1, 2];

x4 = ResaultGeometric[ax, bx + 1, 0];
y4 = ResaultGeometric[ax, bx + 1, 1];
z4 = ResaultGeometric[ax, bx + 1, 2];
}
else
{
x3 = ResaultGeometric[0, 0, 0];
y3 = ResaultGeometric[0, 0, 1];
z3 = ResaultGeometric[0, 0, 2];

x4 = ResaultGeometric[ax, 0, 0];
y4 = ResaultGeometric[ax, 0, 1];
z4 = ResaultGeometric[ax, 0, 2];
}
}


// переменные для расчета нормали
double n1 = 0, n2 = 0, n3 = 0;

// нормаль будем рассчитывать как векторное произведение граней полигона
// для нулевого элемента нормаль мы будем считать немного по-другому

// на самом деле разница в расчете нормали актуальна только для последнего и первого полигонов на меридиане

if (ax == 0) // при расчете нормали для ax мы будем использовать точки 1, 2, 3
{
n1 = (y2 - y1) * (z3 - z1) - (y3 - y1) * (z2 - z1);
n2 = (z2 - z1) * (x3 - x1) - (z3 - z1) * (x2 - x1);
n3 = (x2 - x1) * (y3 - y1) - (x3 - x1) * (y2 - y1);
}
else // для остальных – 1, 3, 4
{
n1 = (y4 - y3) * (z1 - z3) - (y1 - y3) * (z4 - z3);
n2 = (z4 - z3) * (x1 - x3) - (z1 - z3) * (x4 - x3);
n3 = (x4 - x3) * (y1 - y3) - (x1 - x3) * (y4 - y3);
```

```
}

// если не включен режим GL_NORMILIZE, то мы должны в обязательном по-
рядке
// произвести нормализацию вектора нормали, перед тем как передать инфор-
мацию о нормали
double n5 = (double)Math.Sqrt(n1 * n1 + n2 * n2 + n3 * n3);
n1 /= (n5 + 0.01);
n2 /= (n5 + 0.01);
n3 /= (n5 + 0.01);

// передаем информацию о нормали
Gl.glNormal3d(-n1, -n2, -n3);

// передаем 4 вершины для отрисовки полигона
Gl.glVertex3d(x1, y1, z1);
Gl.glVertex3d(x2, y2, z2);
Gl.glVertex3d(x3, y3, z3);
Gl.glVertex3d(x4, y4, z4);
}
}

// завершаем выбранный режим рисования полигонов
Gl.glEnd();
break ;
}
}

// возвращаем сохраненную матрицу
Gl.glPopMatrix();

// завершаем рисование
Gl.glFlush();

// обновляем элемент AnT
AnT.Invalidate();
}
```

На рис. 1.6, 1.7, 1.8 представлены результаты работы программы: вращающееся тело с различными режимами отрисовки геометрии.

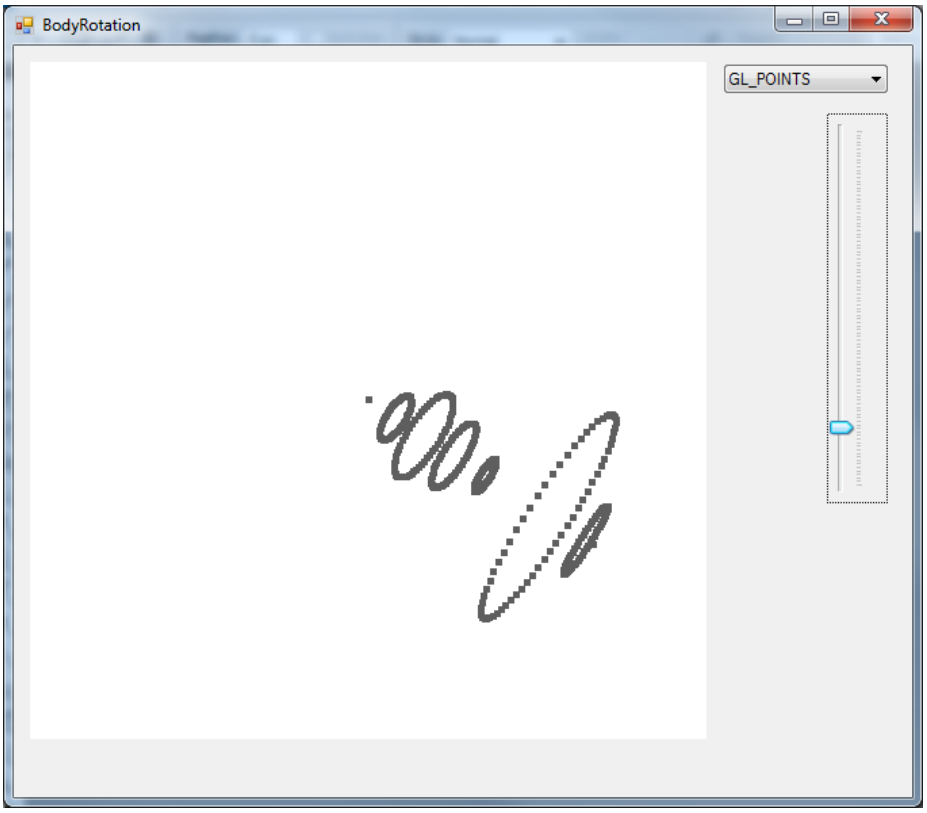

Рис. 1.6

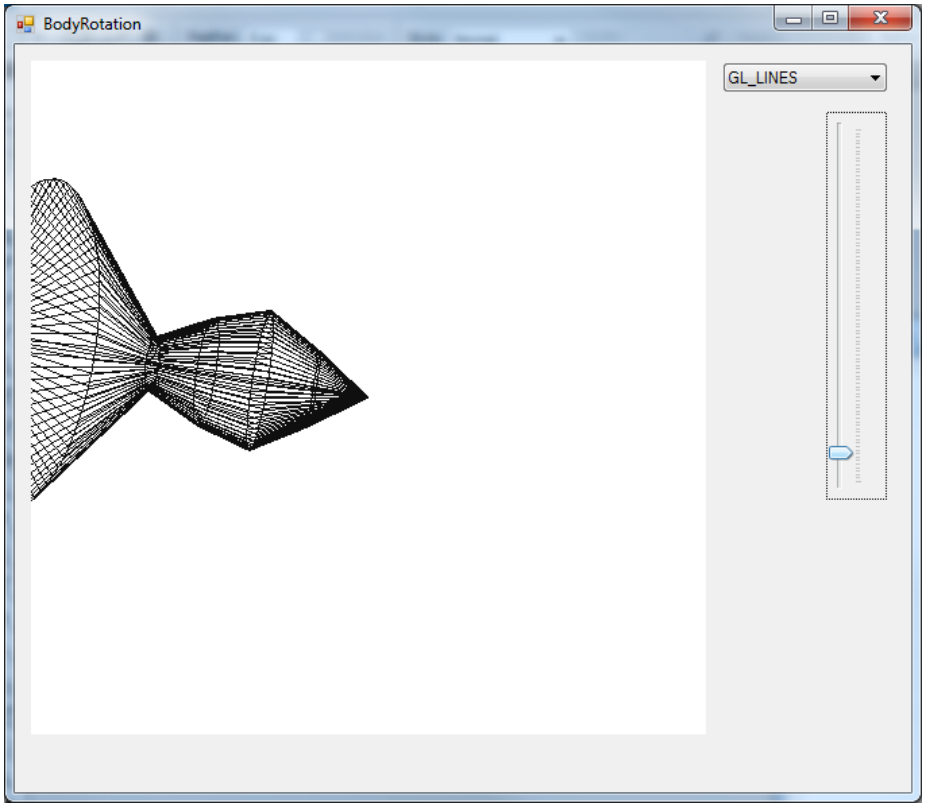

Рис. 1.7

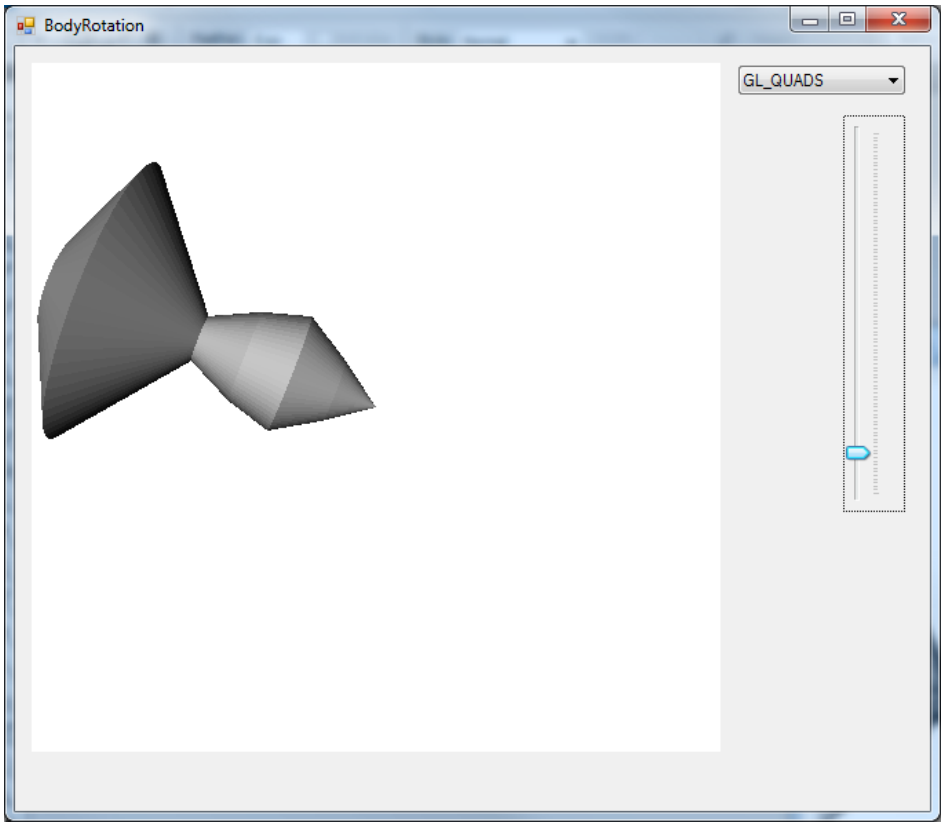

Рис. 1.8

Практическое задание

1. Ознакомиться по методическим указаниям и литературе с теоретическим материалом.

2. Выполнить действия, приведенные в п. 1.2. 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 программы тело вращения должно быть описано кривой, которая реализуется функцией, соответствующей вашему варианту.

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*, исполняемые файлы.

Контрольные вопросы

1. Опишите тела вращения в компьютерной графике.
2. Как построить тела вращения в OpenGL?

Тема 2. **ПРЕОБРАЗОВАНИЕ ОБЪЕКТОВ С ИСПОЛЬЗОВАНИЕМ *GLUT***

Цель изучения темы. Изучение способов реализации функций рисования и преобразования объектов с использованием библиотеки *GLUT*.

2.1. Использование библиотеки *GLUT*

OpenGL Utility Toolkit (GLUT) – библиотека утилит для приложений под *OpenGL*, которая в основном отвечает за системный уровень операций ввода-вывода при работе с операционной системой. Из основных функций можно перечислить следующие: создание окна, управление окном, мониторинг а ввод с клавиатуры и событий мыши. Она также включает функции для рисования ряда геометрических примитивов: куба, сферы, чайника. *GLUT* даже включает возможность создания несложных всплывающих меню.

GLUT была создана Марком Килгардом (Mark Kilgard) во время его работы в *Silicon Graphics Inc*.

Использование библиотеки *GLUT* преследует две цели. Во-первых, это создание кроссплатформенного кода. Во-вторых, *GLUT* позволяет облегчить изучение *OpenGL*. Чтобы начать программировать под *OpenGL*, используя *GLUT*, требуется всего страница кода. Разработка аналогичных вещей на *API* требует нескольких страниц, написанных со знанием *API* управления окнами операционной системы.

Здесь мы рассмотрим использование данной библиотеки для отрисовки *3D*-объектов, причем вопрос построения геометрии берет на себя библиотека *GLUT*. Нашей задачей будет лишь установка необходимых функций и параметров отрисовки.

Модельная трансформация

Ранее мы познакомились с геометрическими преобразованиями объектов. При этом мы производили операции переноса, вращения и

масштабирования, используя математический подход – умножая геометрические точки объекта на необходимую матрицу и меняя данные, на основе которых строилась геометрия объекта. *OpenGL* имеет ряд очень удобных функций для выполнения геометрических преобразований объектов.

Существуют три команды *OpenGL* для геометрических преобразований: *glTranslate*(), glRotate*()* и *glScale*()*. Эти команды трансформируют объект (или координатную систему), перенося, поворачивая, увеличивая, уменьшая или отражая его. Все эти команды эквивалентны созданию соответствующей матрицы переноса, поворота или масштабирования с последующим вызовом *glMultMatrix*()* с рассчитанной матрицей в качестве аргумента. Однако использование этих трех команд может быть быстрее, чем применение *glMultMatrix*() – OpenGL* автоматически вычисляет для них все необходимые матрицы.

В следующем описании команд каждое матричное преобразование рассматривается в терминах того, что оно делает с вершинами геометрического объекта (при использовании подхода с фиксированной системой координат), и в терминах того, что оно делает с локальной системой координат, привязанной к объекту (при использовании этого подхода).

Перенос

void glTranslate{fd} (TYPE x, TYPE y, TYPE z);

Умножает текущую матрицу преобразования на матрицу, переносящую объект на расстояния *x*, *y*, *z*, переданные в качестве аргументов команды, по соответствующим осям (или перемещает локальную координатную систему на те же расстояния).

На рис. 2.1 изображен эффект выполнения команды *glTranslate*()*.

Обратите внимание на то, что использование (0.0, 0.0, 0.0) в качестве аргумента *glTranslate*()* – это единичная операция, то есть она не влияет на объект или на его координатную систему.

Поворот

void glRotate{fd} (TYPE angle, TYPE x, TYPE y, TYPE z);

Умножает текущую матрицу преобразования на матрицу, которая поворачивает объект (или локальную координатную систему) в

направлении против часовой стрелки вокруг луча из начала координат, проходящего через точку (*x*, *y*, *z*). Параметр angle задает угол поворота в градусах.

Результат выполнения *glRotatef(45.0, 0.0,0.0,1.0)*, то есть поворот на 45 градусов вокруг оси *z*, показан на рис. 2.2.

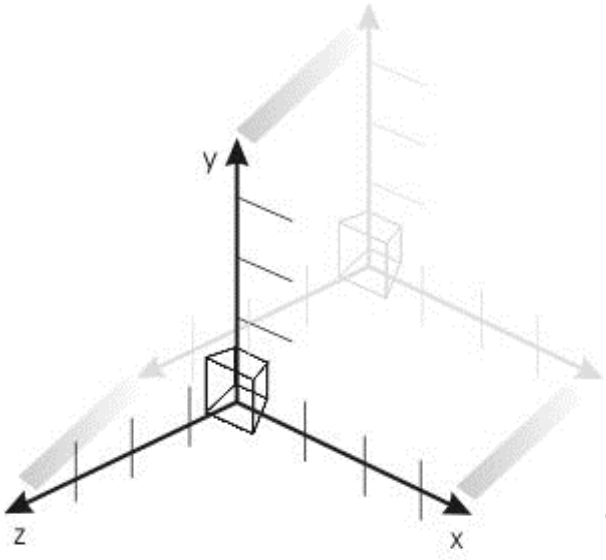

Рис. 2.1

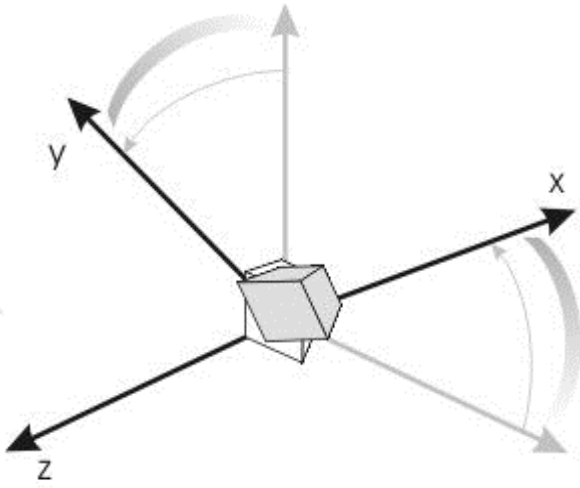

Рис. 2.2

Чем дальше объект от оси вращения, тем больше орбита его поворота и тем заметнее сам поворот. Вызов *glRotate*()* с параметром *angle,* равным нулю, не имеет никакого эффекта.

Масштабирование

void glScale{fd} (TYPE x, TYPE y, TYPE z);

Умножает текущую матрицу на матрицу, которая растягивает, сжимает или отражает объект вдоль координатных осей. Каждая *x-, y-* и *z*-координата каждой точки объекта будет умножена на соответствующий аргумент *x*, *y* или *z* команды *glScale*()*. При рассмотрении преобразования с точки зрения локальной координатной системы оси этой системы растягиваются, сжимаются или отражаются с учетом факторов *x*, *y* и *z*, и ассоциированный с этой системой объект меняется вместе с ней.

На рис. 2.3 показан эффект команды *glScalef(-2.0,0.5,1.0)*;

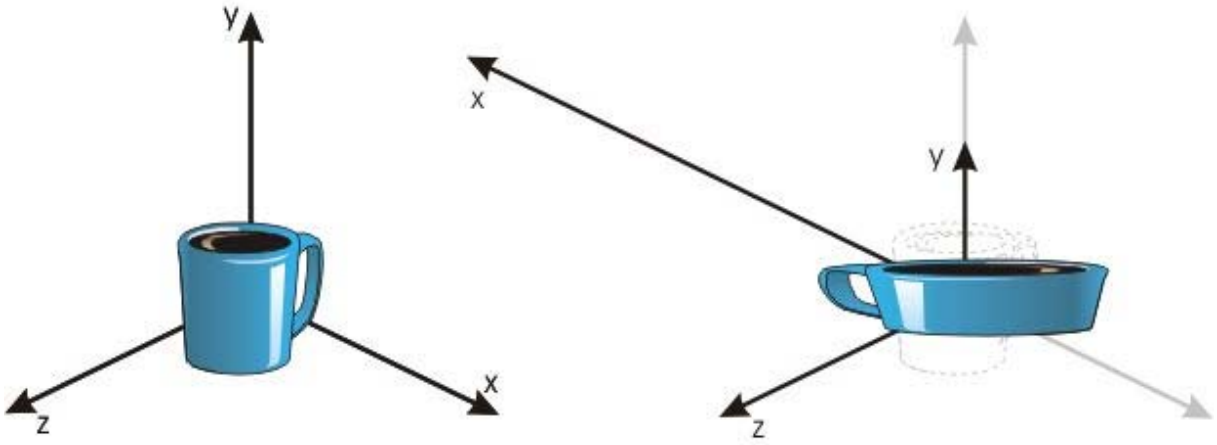

Рис. 2.3

glScale()* – это единственная из трех команд геометрических преобразований, изменяющая размер объекта: масштабирование с величинами более 1.0 растягивает объект, использование величин меньше 1.0 сжимает его. Масштабирование с величиной –1.0 отражает объект относительно оси или осей. Единичными аргументами (то есть аргументами, не имеющими эффекта) являются (1.0, 1.0, 1.0). Вообще следует ограничивать использование *glScale*()* в тех случаях, когда это действительно необходимо.

Использование *glScale*()* снижает быстродействие расчетов освещенности, так как векторы нормалей должны быть нормализованы заново после преобразования.

Величина масштабирования, равная нулю, приводит к коллапсу всех координат объекта по оси или осям до нуля. Обычно это не является удачным приёмом, так как такая операция не может быть обращена. В данном случае матрица не может быть обращена, а обратные матрицы необходимы для многих расчетов, связанных с освещением. Иногда коллапс координат все же имеет смысл (например, для расчета теней на плоской поверхности), но лучше использовать для этого матрицу проецирования, а не масштабирования.

Оболочка программы

Создайте новый проект. После этого создайте окно приложения, разместив на нем элементы управления, как показано на рис. 2.4.

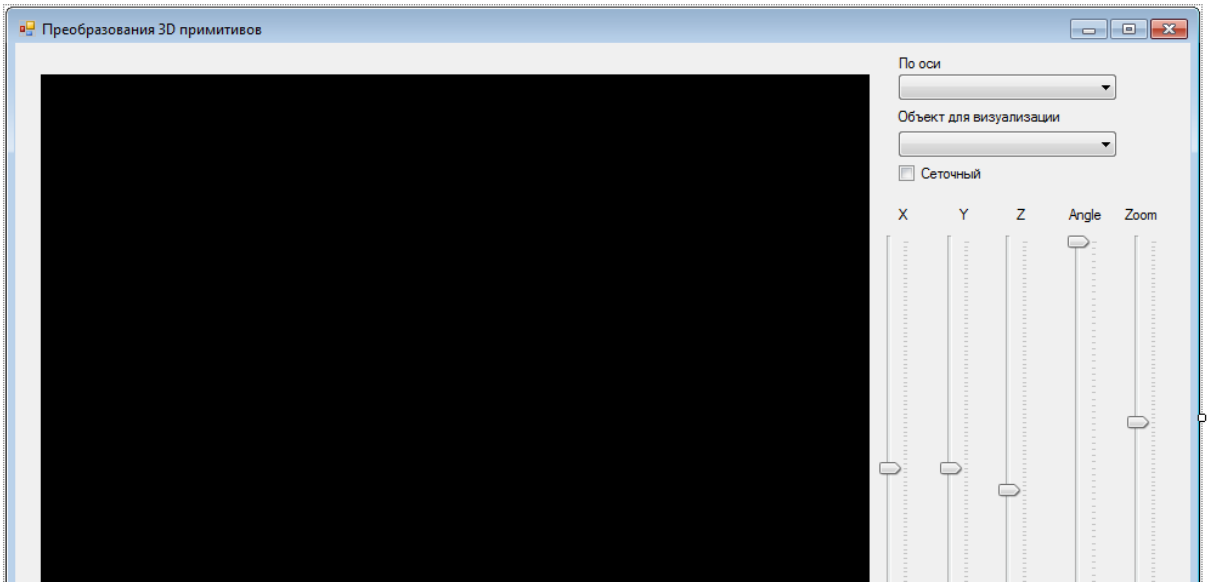

Рис. 2.4

Для элементов *combobox* сделайте установки в соответствии с рис. 2.5, 2.6.

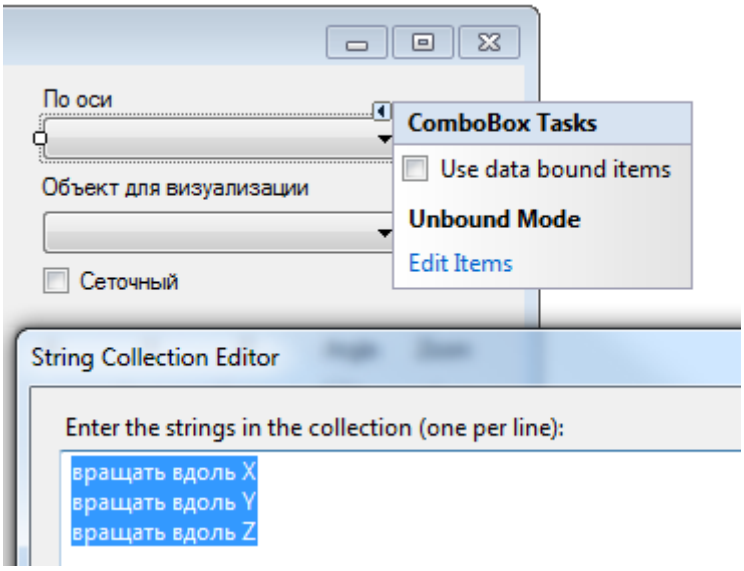

Рис. 2.5

Для элементов *trackBar* установите диапазон значений: от –50000 до 50000 для перемещений по осям; от –360 до 360 для угла поворота; от –5000 до 5000 для масштабирования.

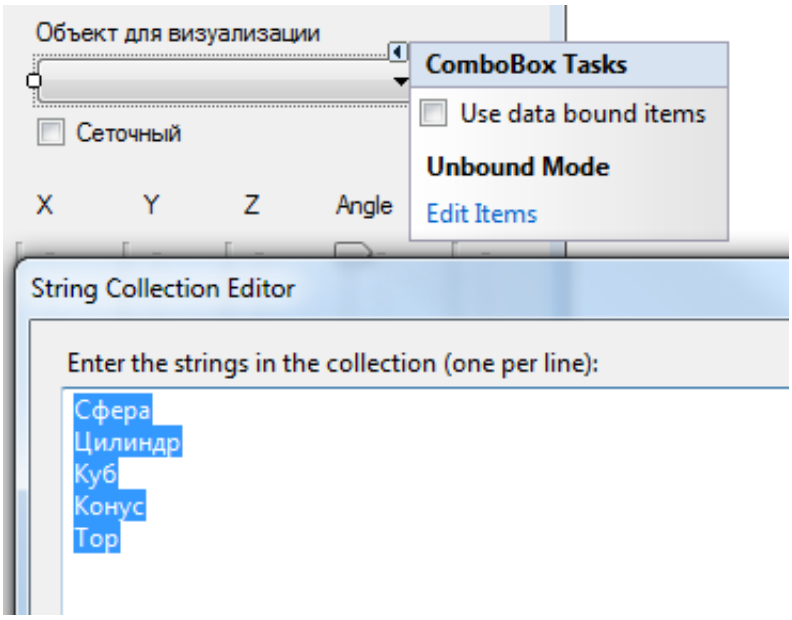

Рис. 2.6

Добавьте таймер, переименовав экземпляр в *RenderTimer*, и установите время отклика равное 30 мс.

2.2. Реализация функций рисования и преобразований объектов

Мы создали оболочку программы, разместив все необходимые элементы управления.

Принцип работы программы прост: в зависимости от значений элементов формы мы будем выбирать примитивы, параметры и режимы рисования.

Перед визуализацией примитива будут выполняться функции переноса, поворота и масштабирования. Они будут ориентироваться на состояние ранее объявленных переменных. Данные переменные, в свою очередь, обновляются при перетаскивании ползунков, нажатии на *checkBox* или изменении элементов *comboBox*.

Объявление начальных переменных и инициализация *OpenGL* выглядит следующим образом:

```
// вспомогательные переменные – в них будут храниться обработанные значения,
// полученные при перетаскивании ползунков пользователем
double a = 0, b = 0, c = -5, d = 0, zoom = 1; // выбранные оси
int os_x = 1, os_y = 0, os_z = 0;

// режим сеточной визуализации
```

```
bool Wire = false ;

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();

// установка перспективы
Glu.gluPerspective(45, (float)AnT.Width / (float)AnT.Height, 0.1, 200);
Gl.glMatrixMode( Gl.GL_MODELVIEW);
Gl.glLoadIdentity();

// начальная настройка параметров openGL (тест глубины, освещение и первый
источник света)
Gl.glEnable( Gl.GL_DEPTH_TEST);
Gl.glEnable( Gl.GL_LIGHTING);
Gl.glEnable( Gl.GL_LIGHT0);

// установка первых элементов в списках combobox
comboBox1.SelectedIndex = 0;
comboBox2.SelectedIndex = 0;

// активация таймера, вызывающего функцию для визуализации
RenderTimer.Start();
}
```

Мы должны добавить обработку событий изменения элементов *trackBar*, событие отклика таймера и обработку изменения значений элементов *checkbox* и *comboBox*.

```
// обрабатываем отклик таймера
private void RenderTimer_Tick( object sender, EventArgs e)
{
// вызываем функцию отрисовки сцены
Draw();
}

// событие изменения значения
private void trackBar1_Scroll( object sender, EventArgs e)
{
// переводим значение, установившееся в элементе trackBar, в необходимый нам
формат
a = (double)trackBar1.Value / 1000.0;
// подписываем это значение в элементе label под данным ползунком
label4.Text = a.ToString();
}

// событие изменения значения
private void trackBar2_Scroll( object sender, EventArgs e)
{
// переводим значение, установившееся в элементе trackBar, в необходимый нам
формат
b = (double)trackBar2.Value / 1000.0;
// подписываем это значение в элементе label под данным ползунком
label5.Text = b.ToString();
}

// событие изменения значения
private void trackBar3_Scroll( object sender, EventArgs e)
{
// переводим значение, установившееся в элементе trackBar, в необходимый нам
формат
c = (double)trackBar3.Value / 1000.0;
// подписываем это значение в элементе label под данным ползунком
label6.Text = c.ToString();
}

// событие изменения значения
private void trackBar4_Scroll( object sender, EventArgs e)
{
// переводим значение, установившееся в элементе trackBar, в необходимый нам
формат
d = (double)trackBar4.Value;
```

```
// подписываем это значение в элементе label под данным ползунком
label6.Text = d.ToString();
}

// событие изменения значения
private void trackBar5_Scroll( object sender, EventArgs e)
{
// переводим значение, установившееся в элементе trackBar, в необходимый нам
формат
zoom = (double)trackBar5.Value / 1000.0;
// подписываем это значение в элементе label под данным ползунком
label6.Text = zoom.ToString();
}

// изменения значения checkBox
private void checkBox1_CheckedChanged( object sender, EventArgs e)
{
// если отмечен
if (checkBox1.Checked)
{
// устанавливаем сеточный режим визуализации
Wire = true ;
}
else
{
// иначе полигональная визуализация
Wire = false ;
}
}

// изменение в элементах comboBox
private void comboBox1_SelectedIndexChanged( object sender, EventArgs e)
{
// в зависимости от выбранного режима
switch (comboBox1.SelectedIndex)
{
// устанавливаем необходимую ось (это действие будет использовано в функции
glRotate**)
case 0:
{
os_x = 1;
os_y = 0;
os_z = 0;
```

```
break ;
}
case 1:
{
os_x = 0;
os_y = 1;
os_z = 0;
break ;
}
case 2:
{
os_x = 0;
os_y = 0;
os_z = 1;
break ;
}
}
}
```

Осталась функция отрисовки сцены. Обратите внимание на то, что перед использованием трансформации обязательно следует загрузить состояние текущей матрицы в стек матриц, чтобы дальнейшие преобразования затронули только визуализируемые объекты.

```
// 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();

// помещаем состояние матрицы в стек матриц, дальнейшие трансформации затронут только визуализацию объекта
Gl.glPushMatrix();
// производим перемещение в зависимости от значений, полученных при перемещении ползунков
Gl.glTranslated(a, b, c);
// поворот по установленной оси
Gl.glRotated(d, os_x, os_y, os_z);
```

```
// и масштабирование объекта
Gl.glScaled(zoom, zoom, zoom);

// в зависимости от установленного типа объекта
switch (comboBox2.SelectedIndex)
{
// рисуем нужный объект, используя фунции библиотеки GLUT
case 0:
{
if (Wire) // если установлен сеточный режим визуализации
Glut.glutWireSphere(2, 16, 16); // сеточная сфера
else
Glut.glutSolidSphere(2, 16, 16); // полигональная сфера
break ;
}
case 1:
{
if (Wire) // если установлен сеточный режим визуализации
Glut.glutWireCylinder(1, 2, 32, 32); // цилиндр
else
Glut.glutSolidCylinder(1, 2, 32, 32);
break ;
}
case 2:
{
if (Wire) // если установлен сеточный режим визуализации
Glut.glutWireCube(2); // куб
else
Glut.glutSolidCube(2);
break ;
}
case 3:
{
if (Wire) // если установлен сеточный режим визуализации
Glut.glutWireCone(2, 3, 32, 32); // конус
else
Glut.glutSolidCone(2, 3, 32, 32);
break ;
}
case 4:
{
if (Wire) // если установлен сеточный режим визуализации
```

```
Glut.glutWireTorus(0.2, 2.2, 32, 32); // тор
else
Glut.glutSolidTorus(0.2, 2.2, 32, 32);
break ;
}
}

// возвращаем состояние матрицы
Gl.glPopMatrix();

// завершаем рисование
Gl.glFlush();

// обновляем элемент AnT
AnT.Invalidate();
}
```

Примеры работы программы представлены на рис. 2.7, 2.8.

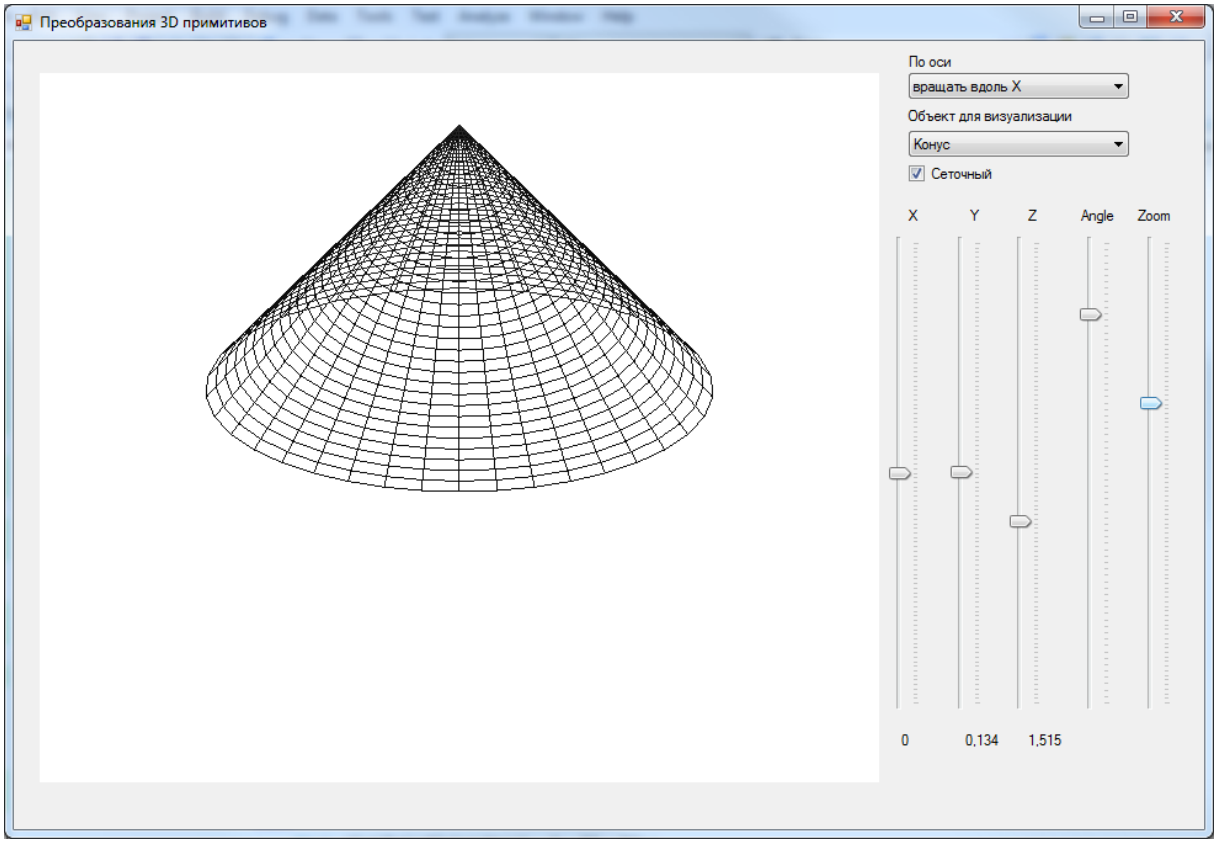

Рис. 2.7

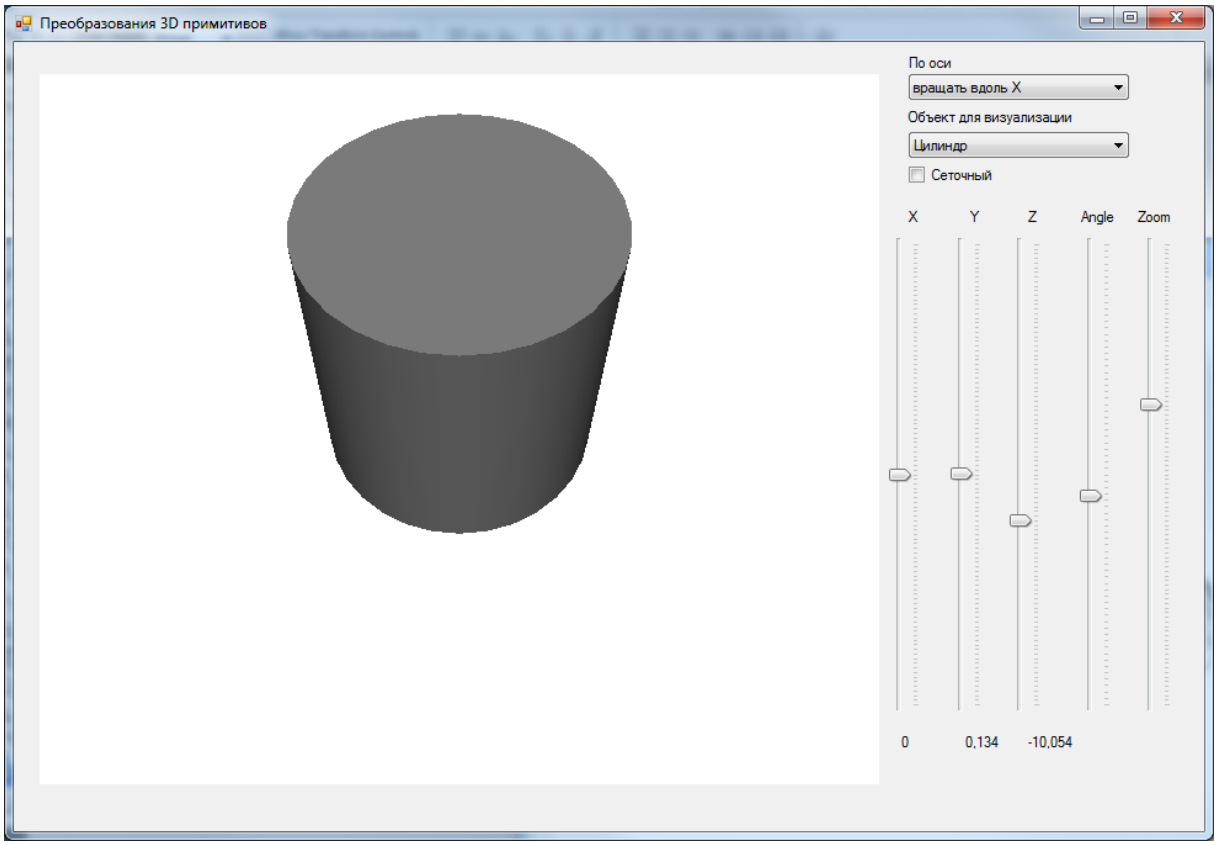

Рис. 2.8

Практическое задание

1. Ознакомиться по методическим указаниям и литературе с теоретическим материалом.

2. Выполнить действия, приведенные в п. 2.2. 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 п. 2.2 изменить программу таким образом, чтобы одновременно отрисовывались *N* выбранных объектов в режиме (по умолчанию), указанном для вашего варианта.

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*, исполняемые файлы.

Контрольные вопросы

1. Опишите библиотеку *GLUT*.
2. Каковы функции отрисовки и преобразования объектов в *OpenGL*?

Тема 3. **ТЕКСТУРЫ**

Цель изучения темы. Изучение способов реализации стилей заполнения поверхностей и закрашивания, которые имитируют сложную рельефную объемную поверхность, выполненную из определенного материала.

3.1. Текстурирование в компьютерной графике

Текстурирование

До сих пор мы рисовали все геометрические объекты либо одним цветом, либо с плавной закраской (когда внутренние цвета рассчитываются интерполяцией цветов в вершинах), то есть объекты рисовались без текстурирования. Если вы хотите нарисовать массивную кирпичную стену без текстурирования, каждый кирпичик должен быть нарисован в виде отдельного полигона. Без текстурирования такая стена может потребовать тысяч полигонов, хотя сама она является прямоугольной, и даже тогда кирпичи будут выглядеть слишком гладкими и одинаковыми, то есть недостаточно реалистичными.

Текстурирование (или наложение текстур) позволяет прикрепить изображение кирпичной стены (полученное, возможно, сканированием реальной фотографии) к полигону и нарисовать всю стену в виде одного полигона. При использовании текстурирования можно быть уверенным в том, что с полигоном в процессе всех преобразований будут происходить правильные изменения. Например, если вы наблюдаете стену в перспективе, кирпичи, находящиеся дальше от вас, будут выглядеть меньше тех, которые ближе. Другие примеры использования текстур: моделирование растительности на полигонах, представляющих землю; наложение обоев на стены; придание объектам вида физических материалов (например, мрамора, дерева или гранита). Хотя наиболее естественным является наложение текстур на полигоны, текстуру можно наложить на любой примитив – точку, линию, полигон, битовую карту или изображение.

Текстуры – это просто прямоугольный массив данных – цветовых, световых или цветовых и альфа. Индивидуальные элементы (значения) текстуры часто называются тэкселями (*texels*). Сложным текстурирование делает то, что прямоугольная текстура может быть наложена на непрямоугольный объект.

Библиотека DEVIL

Для работы с текстурами мы будем использовать библиотеку *DevIL*. При описании библиотеки *Tao* уже говорилось о том, что *DevIL 1.6.8.3* (она же *OpenIL*), включенная в *Tao 2.1.0*, – кроссплатформенная библиотека, реализующая программный интерфейс для работы с изображениями, которая поддерживает работу с изображениями 43 форматов для чтения и 17 – для записи.

С помощью *DevIL* мы сможем быстро без реализации собственного объемного кода загрузить данные графического файла.

Чтобы использовать наложение текстуры, необходимо выполнить следующие шаги:

1. Создать текстурный объект и задать текстуру для него.
2. Задать, как текстура должна воздействовать на каждый пиксель.
3. Активизировать механизм текстурирования.
4. Нарисовать сцену, передавая на конвейер визуализации геометрические координаты и координаты текстуры.

3.2. Программа с текстурированием объектов

Разработка программы начинается с создания оболочки.

Создайте окно программы и разместите в нем элемент *openglsimplecontrol*, как показано на рис. 3.1, после этого установите его размеры 500 × 500. Переименуйте данный объект, назвав его *AnT*.

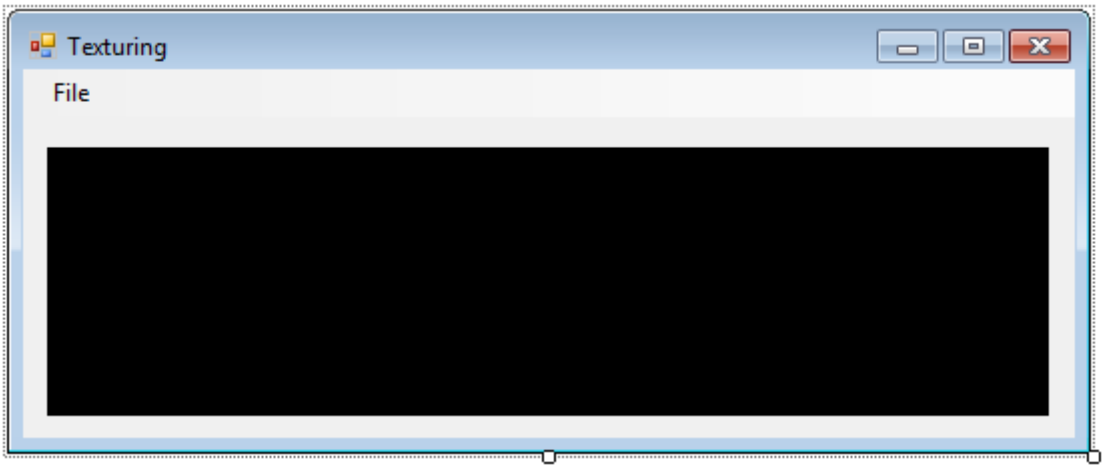

Рис. 3.1

Не забудьте установить ссылки на используемые библиотеки *Tao* (рис. 3.2). Обратите внимание на ссылку на *Tao.DevIL*, данная библиотека необходима нам для загрузки текстур (и не забудьте using *Tao.DevIl*, иначе вы не сможете работать с данной библиотекой).

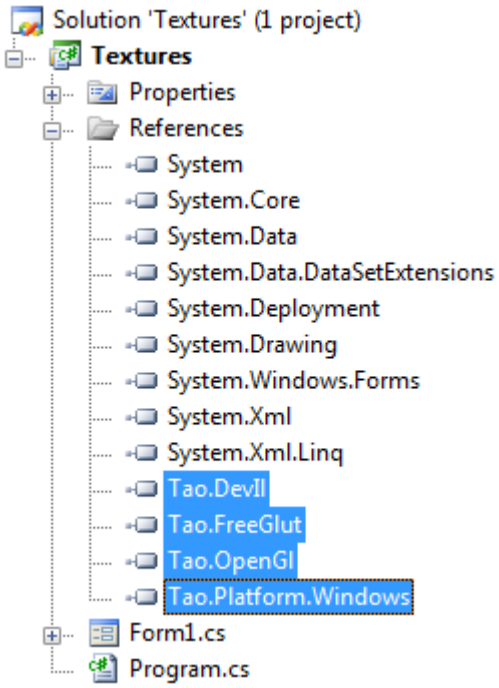

Рис. 3.2

Для реализации визуализации будет использоваться таймер, после инициализации окна он будет генерировать событие, называемое "тиком" таймера, раз в 30 мс. Добавьте элемент таймер, переименуйте экземпляр в *RenderTimer* и установите время "тика" 30 мс (как показано на рис. 3.3), а также добавьте ему событие для обработки "тика".

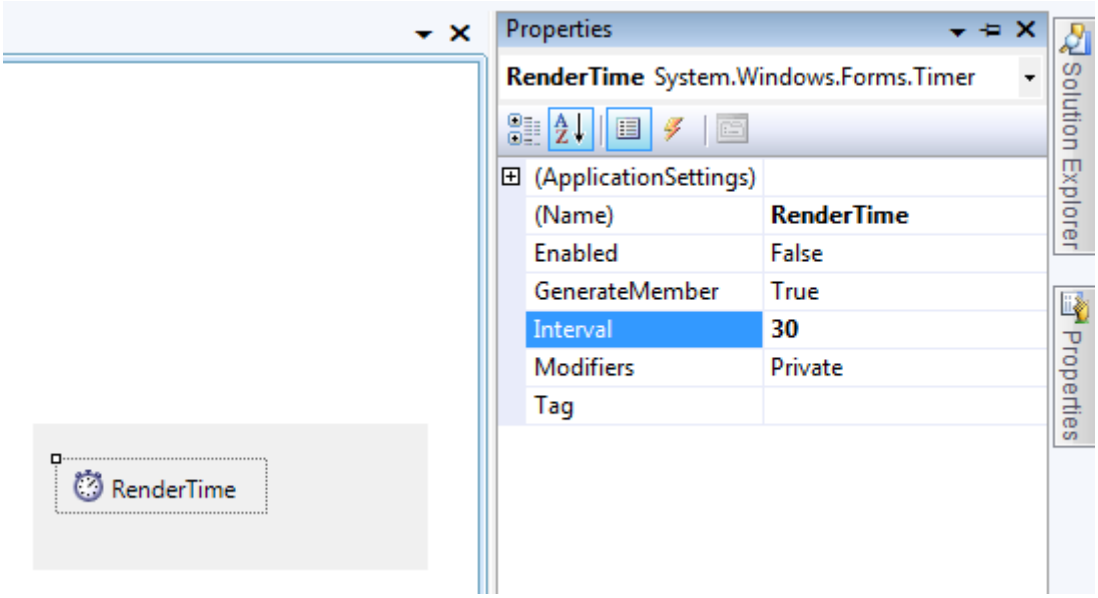

Рис. 3.3

Также необходимо добавить меню для выбора файлов. Для этого добавьте новое меню на форму, объект *openFileDialog*. В свойствах объекта *openFileDialog* установите значение параметра *Filter,* равное «*JPG files|*.jpg|All files|*.*»* (рис. 3.4).

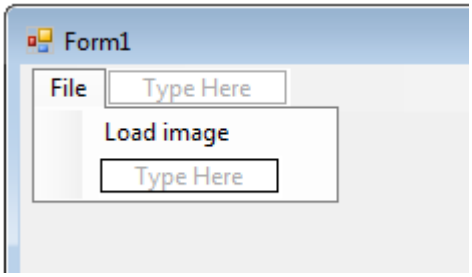

Рис. 3.4

Инициализация *OpenGl* происходит как обычно, следует отметить только дополнительную инициализацию библиотеки *OpenIL*. Нам потребуется объявить ряд переменных для дальнейшей работы программы:

```
// событие загрузки формы
private void Form1_Load( object sender, EventArgs e)
{
// инициализация библиотеки glut
Glut.glutInit();
// инициализация режима экрана
Glut.glutInitDisplayMode( Glut.GLUT_RGB | Glut.GLUT_DOUBLE);

// инициализация библиотеки OpenIL
Il.ilInit();
Il.ilEnable( Il.IL_ORIGIN_SET);

// 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();

// установка перспективы
Glu.gluPerspective(30, AnT.Width / AnT.Height, 1, 100);

// установка объектно-видовой матрицы
Gl.glMatrixMode( Gl.GL_MODELVIEW);
Gl.glLoadIdentity();

// начальные настройки OpenGL
Gl.glEnable( Gl.GL_DEPTH_TEST);
Gl.glEnable( Gl.GL_LIGHTING);
Gl.glEnable( Gl.GL_LIGHT0);
```

```
// активация таймера
RenderTimer.Start();
}
```

Теперь рассмотрим процесс загрузки текстуры и создания файла. Для этого мы реализуем две функции. Первая выполняется как обработка события активации меню, размещенного на нашей форме. Получив имя текстуры, мы проведем начальную подготовку к загрузке текстуры в память графического адаптера.

Далее выполним дополнительные настройки текстуры, после которых сразу же увидим результат.

```
// обработка пункта меню загрузки изображения
private void loadImageToolStripMenuItem_Click( object sender, EventArgs e)
{
// открываем окно выбора файла
DialogResult res = openFileDialog1.ShowDialog(); // если файл выбран и возвращен результат – OK
if (res == DialogResult.OK)
{
// создаем изображение с идентификатором imageId
Il.ilGenImages(1, out imageId);
// делаем изображение текущим
Il.ilBindImage(imageId);

// адрес изображения, полученный с помощью окна выбора файла
string url = openFileDialog1.FileName;

// пробуем загрузить изображение
if ( Il.ilLoadImage(url))
{
// если загрузка прошла успешно,
// сохраняем размеры изображения
int width = Il.ilGetInteger( Il.IL_IMAGE_WIDTH);
int height = Il.ilGetInteger( Il.IL_IMAGE_HEIGHT);

// определяем число бит на пиксель
int bitspp = Il.ilGetInteger( Il.IL_IMAGE_BITS_PER_PIXEL);

switch (bitspp) // в зависимости от полученного результата
{
// создаем текстуру, используя режим GL_RGB или GL_RGBA
```

```
case 24:
mGlTextureObject = MakeGlTexture( Gl.GL_RGB, Il.ilGetData(), width, height);
break ;
case 32:
mGlTextureObject = MakeGlTexture( Gl.GL_RGBA, Il.ilGetData(), width, height);
break ;
}

// активируем флаг, сигнализирующий загрузку текстуры
textureIsLoad = true ;
// очищаем память
Il.ilDeleteImages(1, ref imageId);
}
}
}

// создание текстуры в памяти openGL
private static u int MakeGlTexture( int Format, IntPtr pixels, int w, int h)
{
// идентификатор текстурного объекта
u int texObject;

// генерируем текстурный объект
Gl.glGenTextures(1, out texObject);

// устанавливаем режим упаковки пикселей
Gl.glPixelStorei( Gl.GL_UNPACK_ALIGNMENT, 1);

// создаем привязку к только что созданной текстуре
Gl.glBindTexture( Gl.GL_TEXTURE_2D, texObject);

// устанавливаем режим фильтрации и повторения текстуры
Gl.glTexParameteri( Gl.GL_TEXTURE_2D, Gl.GL_TEXTURE_WRAP_S,
Gl.GL_REPEAT);
Gl.glTexParameteri( Gl.GL_TEXTURE_2D, Gl.GL_TEXTURE_WRAP_T,
Gl.GL_REPEAT);
Gl.glTexParameteri( Gl.GL_TEXTURE_2D, Gl.GL_TEXTURE_MAG_FILTER,
Gl.GL_LINEAR);
Gl.glTexParameteri( Gl.GL_TEXTURE_2D, Gl.GL_TEXTURE_MIN_FILTER,
Gl.GL_LINEAR);
Gl.glTexEnvf( Gl.GL_TEXTURE_ENV, Gl.GL_TEXTURE_ENV_MODE,
Gl.GL_REPLACE);
```

```
// создаем RGB- или RGBA-текстуру
switch (Format)
{
case Gl.GL_RGB:
Gl.glTexImage2D( Gl.GL_TEXTURE_2D, 0, Gl.GL_RGB, w, h, 0, Gl.GL_RGB,
Gl.GL_UNSIGNED_BYTE, pixels);
break ;

case Gl.GL_RGBA:
Gl.glTexImage2D( Gl.GL_TEXTURE_2D, 0, Gl.GL_RGBA, w, h, 0, Gl.GL_RGBA,
Gl.GL_UNSIGNED_BYTE, pixels);
break ;
}

// возвращаем идентификатор текстурного объекта

return texObject;
}
```

Остается обработать событие отклика таймера и реализовать функцию визуализации сцены. При визуализации текстуры нам необходимо включить режим текстурирования, а при выводе вершин объектов привязывать к ним текстурные координаты, в соответствии с которыми произойдет наложение текстуры.

```
// отклик таймера
private void RenderTimer_Tick( object sender, EventArgs e)
{
// вызов функции отрисовки сцены
Draw();
} // функция отрисовки
private void Draw()
{
// если текстура загружена
if (textureIsLoad)
{
// увеличиваем угол поворота
rot++;
// корректируем угол
if (rot > 360)
rot = 0;

// 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();

// включаем режим текстурирования
Gl.glEnable( Gl.GL_TEXTURE_2D);
// включаем режим текстурирования, указывая идентификатор mGlTextureObject
Gl.glBindTexture( Gl.GL_TEXTURE_2D, mGlTextureObject);

// сохраняем состояние матрицы
Gl.glPushMatrix();

// выполняем перемещение для более наглядного представления сцены
Gl.glTranslated(0, -1, -5);
// реализуем поворот объекта
Gl.glRotated(rot, 0, 1, 0);

// отрисовываем полигон
Gl.glBegin( Gl.GL_QUADS);

// указываем поочередно вершины и текстурные координаты
Gl.glVertex3d(1, 1, 0);
Gl.glTexCoord2f(0, 0);
Gl.glVertex3d(1, 0, 0);
Gl.glTexCoord2f(1, 0);
Gl.glVertex3d(0, 0, 0);
Gl.glTexCoord2f(1, 1);
Gl.glVertex3d(0, 1, 0);
Gl.glTexCoord2f(0, 1);

// завершаем отрисовку
Gl.glEnd();

// возвращаем матрицу
Gl.glPopMatrix();
// отключаем режим текстурирования
Gl.glDisable( Gl.GL_TEXTURE_2D);

// обновляем элемент со сценой
AnT.Invalidate();
}
}
```

Результат работы программы – вращающаяся плоскость с изображением текстуры (рис. 3.5).

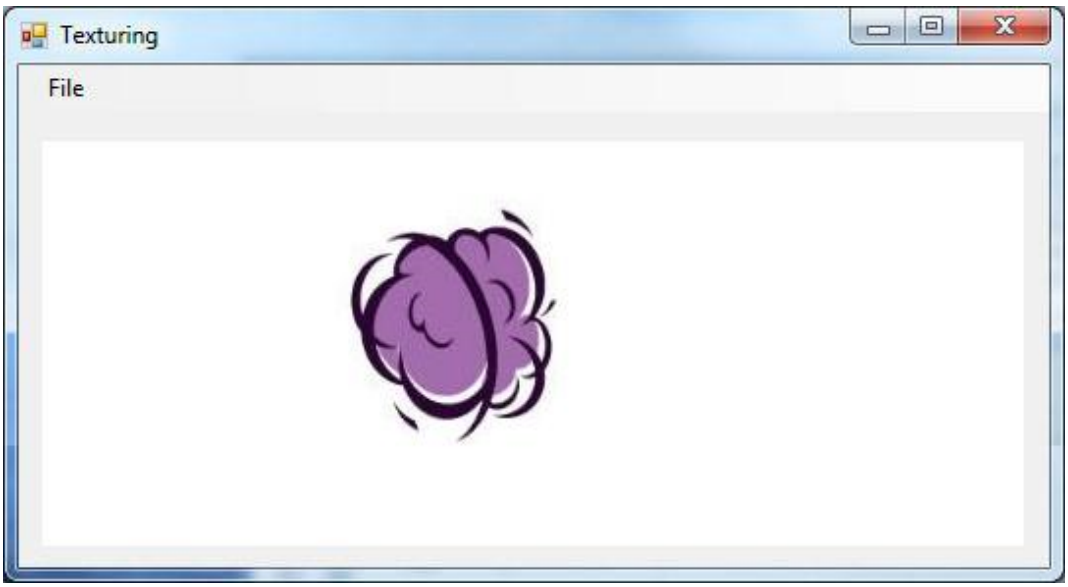

Рис. 3.5

Практическое задание

1. Ознакомиться по методическим указаниям и литературе с 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. При выполнении п. 3.2 использовать загрузку текстуры с именем, фамилией и группой студента, выполнявшего работу.

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*, исполняемые файлы.

Контрольные вопросы

1. Каково применение текстур в компьютерной графике?
2. Опишите библиотеку DEVIL.
3. Каким образом используют текстуры в OpenGL?

Тема 4. **ОПИСАНИЕ *3D*-ОБЪЕКТОВ**

Цель изучения темы. Изучение методов описания графических изображений, состоящих из трехмерных объектов, приобретение навыков использования соответствующих алгоритмов при составлении графических программ.

4.1. Представление *3D*-объектов в компьютерной графике

Представление 3D-объектов и формат ASE

Библиотека *GLUT* позволяет создавать большое количество *3D*-объектов. Мы можем строить объекты и "вручную".

Тем не менее для разработки сложных графических сцен используются трехмерные объекты, созданные в крупных пакетах трехмерного моделирования, например *3D Studio Max*. Экспортируя трехмерный объект из пакета трехмерного моделирования, мы получаем файл, который хранит геометрические данные, текстурные координаты, источники света и так далее в зависимости от настроек экспорта и самого формата файла.

Способ хранения данных описывается спецификацией для данного формата.

Метод хранения геометрии в сторонних файлах удобен. Имея класс загрузки трехмерных объектов, мы можем на ходу преобразовывать геометрию сцены, меняя файл с описанием трехмерного объекта.

Хранение геометрии *3D*-объекта сводится к хранению *N* узлов, определяющих объект.

Узел (или подобъект) представляет собой набор координат в трехмерном пространстве. Представление такого объекта можно разделить на два больших массива. Так как соседние полигоны, как правило, используют общие вершины, хранить координаты узлов отдельно для каждого полигона невыгодно, поэтому и используются два геометрических массива. Первый массив содержит описание всех вершин, второй – полигон. Каждый полигон описывается тремя индексами координат, которые мы должны взять из первого массива (вершин).

Загрузку трехмерной модели рассмотрим на примере формата *ASE*, в котором можно легко сохранить трехмерную модель, используя пакет трехмерного моделирования *3D Studio Max*.

Формат является текстовым, и вы можете самостоятельно изучить, как описывается экспортируемая геометрия.

Так как мы реализуем загрузку трехмерной модели для обучения, не будем использовать многие параметры, хранящиеся в данном файле. Нормали мы посчитаем самостоятельно. Также загрузим текстурные координаты для загрузки текстуры, наложенной на объект.

Экспорт данных из 3D Studio Max

В качестве примера создадим в *3D Studio Max* модель чайника (ее тоже можно отрисовать, используя библиотеки *openGL*, но в данном случае мы выбираем простейшую *3D*-модель для демонстрации алгоритма загрузки).

Создайте новые модели чайника и куба. С помощью редактора материалов назначьте данным объектам текстуры (рис. 4.1, 4.2).

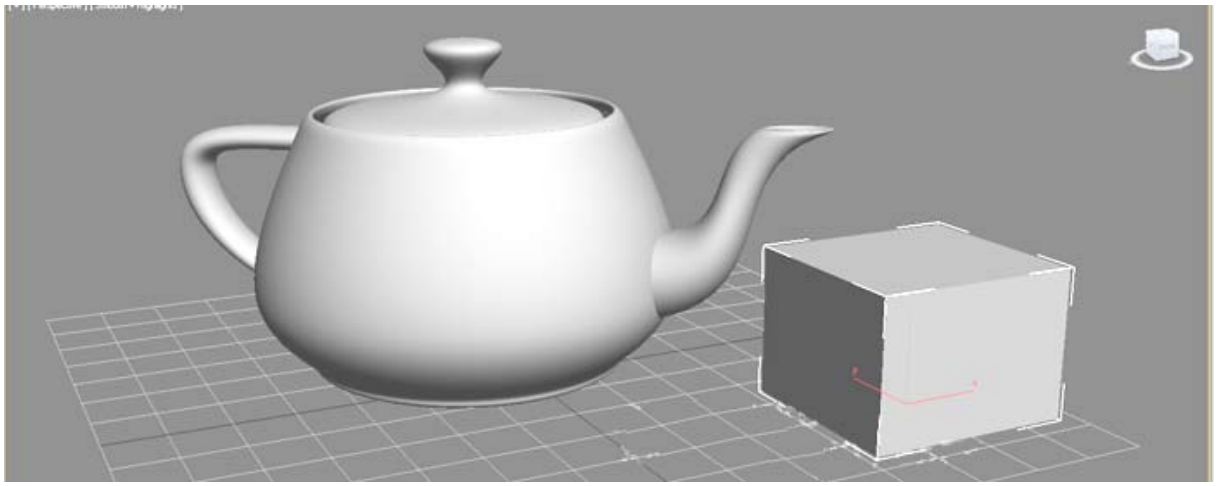

Рис. 4.1

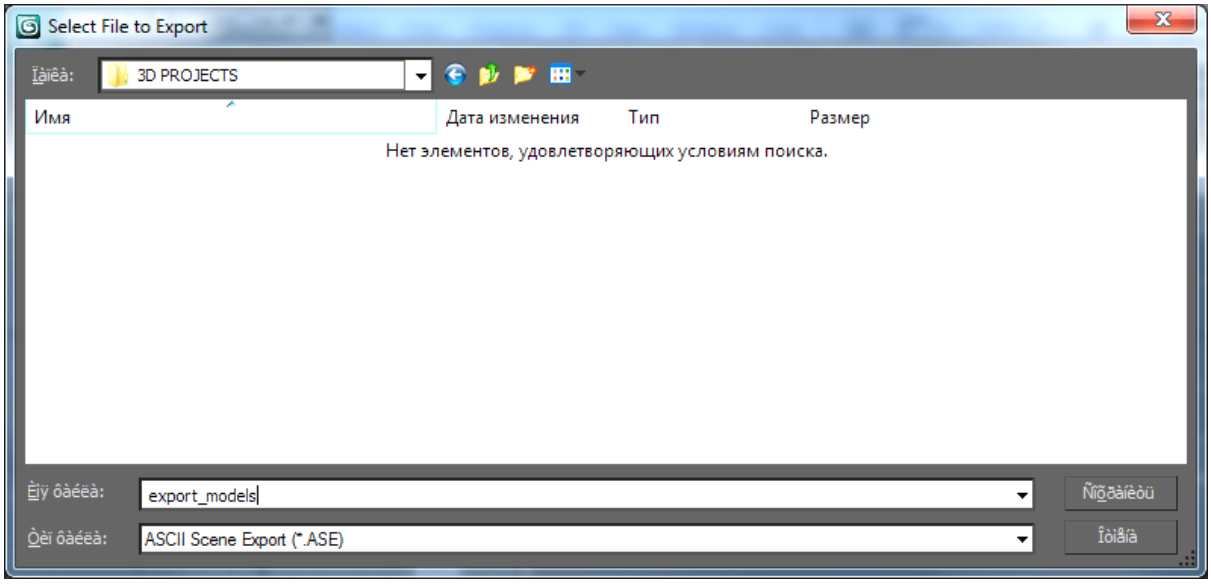

Рис. 4.2

Теперь воспользуйтесь меню *File -> Export*. В качестве формата установите формат *ASE* (рис. 4.3)

При настройках экспорта необходимо указать, что мы экспортируем исключительно геометрию.

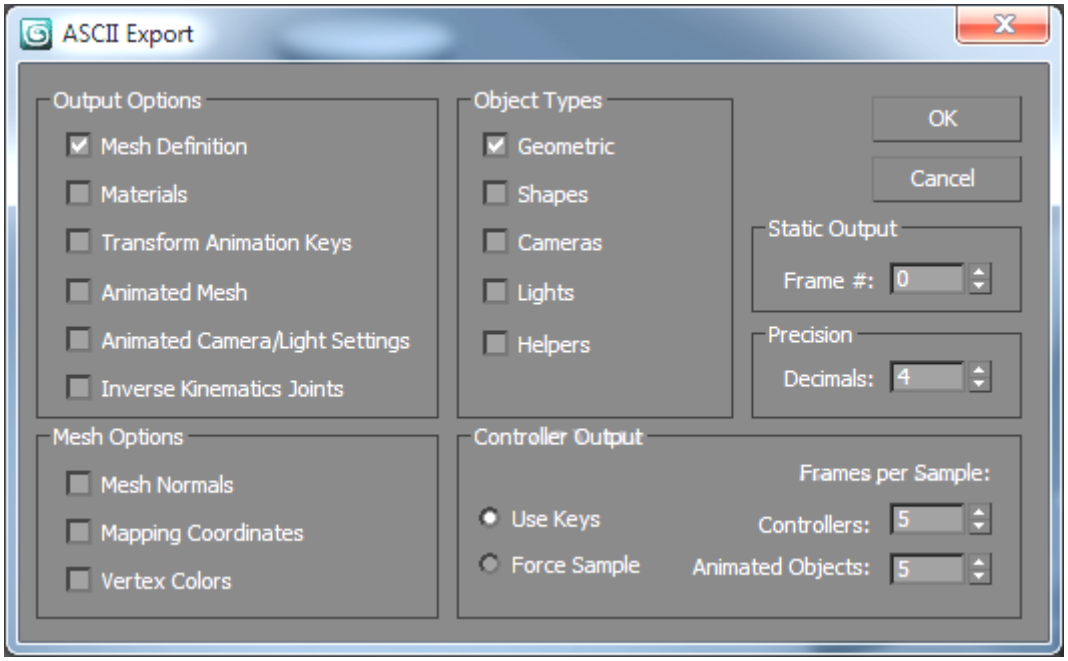

Рис. 4.3

Для использования сохраненной в файле модели необходимо создать класс для её загрузки и подключить его к оболочке приложения.

4.2. Загрузка и визуализация *3D*-модели

Для создания программы загрузки *3D*-моделей из формата *ASE* воспользуемся оболочкой приложения для отрисовки моделей с помощью стандартных функций библиотеки *GLUT* (рис. 4.4).

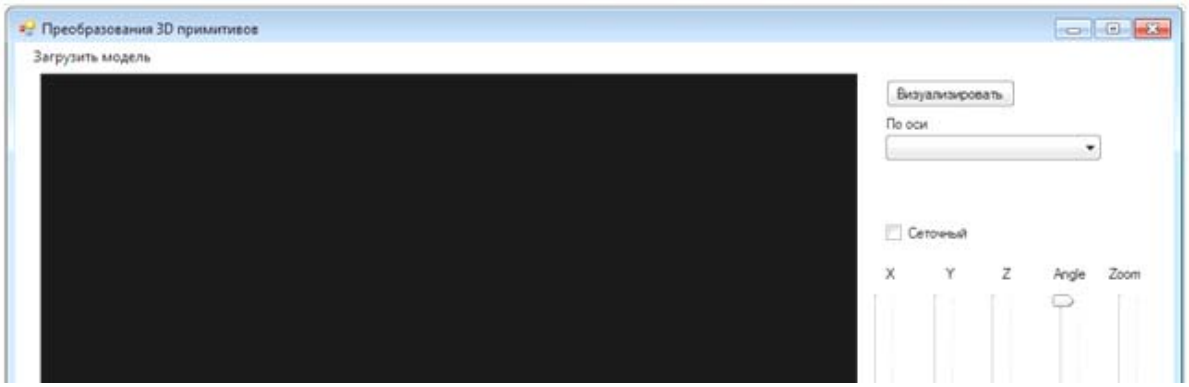

Рис. 4.4

Мы должны создать меню для отображения окна выбора файла, который в дальнейшем будет загружен (рис. 4.5).

В остальном код оболочки программы остался неизменным – за некоторыми исключениями, которые мы рассмотрим после того, как осуществим реализацию класса для загрузки *3D*-модели.

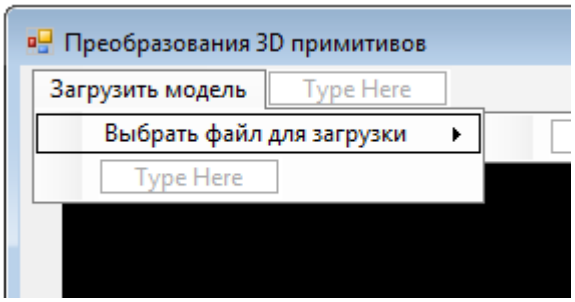

Рис. 4.5

Добавьте к проекту файл *anModelLoader.cs*. В нем реализованы классы, которые будут отвечать за загрузку, хранение и отрисовку *3D*-модели.

На рис. 4.6 представлено изображение двух полигонов, имеющих одно общее ребро.

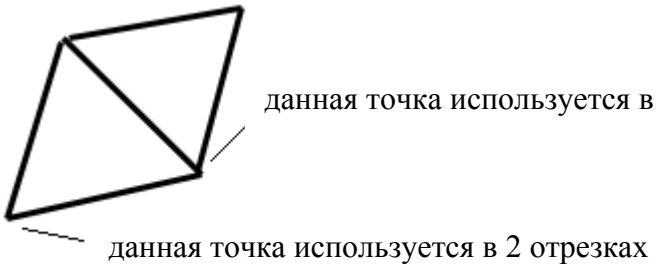

Рис. 4.6

На рис. 4.6 выделены две вершины, которые неоднократно используются для описания (построения) отрезков, образующих полигоны.

Если описывать по отдельности каждый полигон, из которых состоит трехмерная модель, мы получим избыточность данных. Поэтому для хранения информации о геометрии трехмерного массива используют два массива:

- Первый массив содержит координаты всех вершин (каждая вершина описывается 3 координатами).
- Второй массив содержит последовательности номеров вершин, которые мы должны соединить, чтобы восстановить геометрию загружаемой *3D*-модели.

Если мы дополнительно вычислим (или загрузим из файла) координаты векторов нормалей для каждого отрисовываемого полигона, то сможем отобразить трехмерную модель.

Также мы можем загрузить из файла, который должен быть загружен в качестве текстуры, информацию о текстуре, после чего можем визуализировать трехмерную модель с текстурой. Таким образом любая сцена, созданная в *3D Studio Max* или другом пакете трехмерного моделирования, может быть загружена в программу.

Реализовав камеру для перемещения по трехмерной сцене или какие-либо другие опции просмотра данной модели, мы можем добиться высокого уровня интерактивности создаваемого приложения.

Стоит также отметить вопрос визуализации данной *3D*-модели.

После загрузки модели геометрия будет записана в массив объектов *limb*, каждый элемент которого будет отвечать за один из логических элементов сцены. То есть если в трехмерную сцену было экспортировано два объекта – модель куба и *3D*-модель чайника, то мы получим массив из двух экземпляров класса *limb*, один из которых будет описывать геометрию чайника, второй – геометрию куба.

После загрузки производится визуализация данной модели в дисплейный список. Потом нам достаточно будет вызывать данный дисплейный список для отрисовки геометрии.

Объектно-ориентированное программирование позволит нам удобно загружать любое количество *3D*-моделей, а также легко управлять перемещением моделей в сцене и их отрисовкой.

Теперь вернемся к рассмотрению кода, реализующего функциональность нашей программы. Класс *LIMB*:

```
// класс LIMB отвечает за логические единицы 3D-объектов в загружаемой сцене
class LIMB
{
// при инициализации мы должны указать количество вершин (vertex) и полигонов (face), которые описывают геометрию подобъекта
public LIMB( int a, int b)
{
if (temp[0] == 0)
temp[0] = 1;

// записываем количество вершин и полигонов
VandF[0] = a;
VandF[1] = b;

// выделяем память
```

```
memcompl();
}

public int Itog; // флаг успешности

// массивы для хранения данных (геометрии и текстурных координат)
public float[,] vert;
public int[,] face;
public float[,] t_vert;
public int[,] t_face;

// номер материала (текстуры) данного подобъекта
public int MaterialNom = -1;

// временное хранение информации
public int[] VandF = new int[4];
private int[] temp = new int[2];

// флаг, говорящий о том, что модель использует текстуру
private bool ModelHasTexture = false ;

// функция для определения значения флага (наличие текстуры)
public bool NeedTexture()
{
// возвращаем значение флага
return ModelHasTexture;
}

// массивы для текстурных координат
public void createTextureVertexMem( int a)
{
VandF[2] = a;
t_vert = new float[3, VandF[2]];
}

// привязка значений текстурных координат к полигонам
public void createTextureFaceMem( int b)
{
VandF[3] = b;
t_face = new int[3, VandF[3]];

// отмечаем флаг наличия текстуры
ModelHasTexture = true ;
```

```
}

// память для геометрии
private void memcompl()
{
vert = new float[3, VandF[0]];
face = new int[3, VandF[1]];
}

// номер текстуры
public int GetTextureNom()
{
return MaterialNom;
}
};
```

Использование библиотеки *DevIL* мы уже рассмотрели. Теперь должны немного изменить реализацию таким образом, чтобы включить в нее класс, который позволит быстро загружать и использовать текстуру прямо при чтении файла с геометрией сцены.

```
// класс для работы с текстурами
public class TexturesForObjects
{

public TexturesForObjects()
{
}

// имя текстуры
private string texture_name = "";
// ee ID
private int imageId = 0;

// идентификатор текстуры в памяти openGL
private u int mGlTextureObject = 0;

// получение этого идентификатора
public u int GetTextureObj()
{
return mGlTextureObject;
}
```

```
// загрузка текстуры
public void LoadTextureForModel( string FileName)
{
// запоминаем имя файла
texture_name = FileName;
// создаем изображение с идентификатором imageId
Il.ilGenImages(1, out imageId);
// делаем изображение текущим
Il.ilBindImage(imageId);

string url = "";
// получаем адрес текущей директории
url = Directory.GetCurrentDirectory();
url += "\\";
// добавляем имя текстуры
url += texture_name;

// если загрузка удалась
if ( Il.ilLoadImage(url))
{
// если загрузка прошла успешно
// сохраняем размеры изображения
int width = Il.ilGetInteger( Il.IL_IMAGE_WIDTH);
int height = Il.ilGetInteger( Il.IL_IMAGE_HEIGHT);

// определяем число бит на пиксель
int bitspp = Il.ilGetInteger( Il.IL_IMAGE_BITS_PER_PIXEL);

switch (bitspp) // в зависимости от полученного результата
{
// создаем текстуру, используя режиы GL_RGB или GL_RGBA
case 24:
mGlTextureObject = MakeGlTexture( Gl.GL_RGB, Il.ilGetData(), width, height);
break ;
case 32:
mGlTextureObject = MakeGlTexture( Gl.GL_RGBA, Il.ilGetData(), width, height);
break ;
}

// очищаем память
Il.ilDeleteImages(1, ref imageId);
}
}
```

```
// создание текстуры в памяти OpenGL
private static u int MakeGlTexture( int Format, IntPtr pixels, int w, int h)
{
// идентификатор текстурного объекта
u int texObject;

// генерируем текстурный объект
Gl.glGenTextures(1, out texObject);

// устанавливаем режим упаковки пикселей
Gl.glPixelStorei( Gl.GL_UNPACK_ALIGNMENT, 1);

// создаем привязку к только что созданной текстуре
Gl.glBindTexture( Gl.GL_TEXTURE_2D, texObject);

// устанавливаем режим фильтрации и повторения текстуры
Gl.glTexParameteri( Gl.GL_TEXTURE_2D, Gl.GL_TEXTURE_WRAP_S,
Gl.GL_REPEAT);
Gl.glTexParameteri( Gl.GL_TEXTURE_2D, Gl.GL_TEXTURE_WRAP_T,
Gl.GL_REPEAT);
Gl.glTexParameteri( Gl.GL_TEXTURE_2D, Gl.GL_TEXTURE_MAG_FILTER,
Gl.GL_LINEAR);
Gl.glTexParameteri( Gl.GL_TEXTURE_2D, Gl.GL_TEXTURE_MIN_FILTER,
Gl.GL_LINEAR);
Gl.glTexEnvf( Gl.GL_TEXTURE_ENV, Gl.GL_TEXTURE_ENV_MODE,
Gl.GL_REPLACE);

// создаем RGB- или RGBA-текстуру
switch (Format)
{
case Gl.GL_RGB:
Gl.glTexImage2D( Gl.GL_TEXTURE_2D, 0, Gl.GL_RGB, w, h, 0, Gl.GL_RGB,
Gl.GL_UNSIGNED_BYTE, pixels);
break ;

case Gl.GL_RGBA:
Gl.glTexImage2D( Gl.GL_TEXTURE_2D, 0, Gl.GL_RGBA, w, h, 0, Gl.GL_RGBA,
Gl.GL_UNSIGNED_BYTE, pixels);
break ;
}
// возвращаем идентификатор текстурного объекта
return texObject;
}
}
```

Рассмотрим формат *ASE*, в который мы экспортировали геометрию и который будем использовать для хранения и загрузки трехмерных сцен.

Небольшой отрывок файла в формате *ASE* (формат является текстовым, благодаря чему можно изучить его внутреннее устройство):

```
*3DSMAX_ASCIIEXPORT 200
*COMMENT "AsciiExport Version 2,00 - Mon Jan 18 01:42:41 2010"
*SCENE {

*SCENE_FILENAME ""
*SCENE_FIRSTFRAME 0
*SCENE_LASTFRAME 100
*SCENE_FRAMESPEED 30
*SCENE_TICKSPERFRAME 160
*SCENE_BACKGROUND_STATIC 0.0000 0.0000 0.0000
*SCENE_AMBIENT_STATIC 0.0000 0.0000 0.0000
}
*MATERIAL_LIST {

*MATERIAL_COUNT 0
}
*GEOMOBJECT {

*NODE_NAME "Box01"
*NODE_TM {

*NODE_NAME "Box01"
*INHERIT_POS 0 0 0
*INHERIT_ROT 0 0 0
*INHERIT_SCL 0 0 0
*TM_ROW0 0.3309 0.0000 0.0000
*TM_ROW1 0.0000 0.3309 0.0000
*TM_ROW2 0.0000 0.0000 0.3309
*TM_ROW3 -21.0978 -6.6046 7.9985
*TM_POS -21.0978 -6.6046 7.9985
*TM_ROTAXIS 0.0000 0.0000 0.0000
*TM_ROTANGLE 0.0000
*TM_SCALE 0.3309 0.3309 0.3309
*TM_SCALEAXIS 0.0000 0.0000 0.0000
*TM_SCALEAXISANG 0.0000
}
*MESH {
```

```
*TIMEVALUE 0
*MESH_NUMVERTEX 8
*MESH_NUMFACES 12
*MESH_VERTEX_LIST {

*MESH_VERTEX 0 -25.6486 -11.3866 7.9985
*MESH_VERTEX 1 -16.5471 -11.3866 7.9985
*MESH_VERTEX 2 -25.6486 -1.8227 7.9985
*MESH_VERTEX 3 -16.5471 -1.8227 7.9985
*MESH_VERTEX 4 -25.6486 -11.3866 15.3495
*MESH_VERTEX 5 -16.5471 -11.3866 15.3495
*MESH_VERTEX 6 -25.6486 -1.8227 15.3495
*MESH_VERTEX 7 -16.5471 -1.8227 15.3495
}
*MESH_FACE_LIST {

*MESH_FACE 0: A: 0 B: 2 C: 3 AB: 1 BC: 1 CA: 0 *MESH_SMOOTHING 2
*MESH_MTLID 1
*MESH_FACE 1: A: 3 B: 1 C: 0 AB: 1 BC: 1 CA: 0 *MESH_SMOOTHING 2
*MESH_MTLID 1
*MESH_FACE 2: A: 4 B: 5 C: 7 AB: 1 BC: 1 CA: 0 *MESH_SMOOTHING 3
*MESH_MTLID 0
*MESH_FACE 3: A: 7 B: 6 C: 4 AB: 1 BC: 1 CA: 0 *MESH_SMOOTHING 3
*MESH_MTLID 0
*MESH_FACE 4: A: 0 B: 1 C: 5 AB: 1 BC: 1 CA: 0 *MESH_SMOOTHING 4
*MESH_MTLID 4
*MESH_FACE 5: A: 5 B: 4 C: 0 AB: 1 BC: 1 CA: 0 *MESH_SMOOTHING 4
*MESH_MTLID 4
*MESH_FACE 6: A: 1 B: 3 C: 7 AB: 1 BC: 1 CA: 0 *MESH_SMOOTHING 5
*MESH_MTLID 3
*MESH_FACE 7: A: 7 B: 5 C: 1 AB: 1 BC: 1 CA: 0 *MESH_SMOOTHING 5
*MESH_MTLID 3
*MESH_FACE 8: A: 3 B: 2 C: 6 AB: 1 BC: 1 CA: 0 *MESH_SMOOTHING 6
*MESH_MTLID 5
*MESH_FACE 9: A: 6 B: 7 C: 3 AB: 1 BC: 1 CA: 0 *MESH_SMOOTHING 6
*MESH_MTLID 5
*MESH_FACE 10: A: 2 B: 0 C: 4 AB: 1 BC: 1 CA: 0 *MESH_SMOOTHING 7
*MESH_MTLID 2
*MESH_FACE 11: A: 4 B: 6 C: 2 AB: 1 BC: 1 CA: 0 *MESH_SMOOTHING 7
*MESH_MTLID 2
}
*MESH_NUMTVERTEX 12
```

```
*MESH_TVERTLIST {

*MESH_TVERT 0 0.0000 0.0000 0.0000
*MESH_TVERT 1 1.0000 0.0000 0.0000
*MESH_TVERT 2 0.0000 1.0000 0.0000
*MESH_TVERT 3 1.0000 1.0000 0.0000
*MESH_TVERT 4 0.0000 0.0000 0.0000
*MESH_TVERT 5 1.0000 0.0000 0.0000
*MESH_TVERT 6 0.0000 1.0000 0.0000
*MESH_TVERT 7 1.0000 1.0000 0.0000
*MESH_TVERT 8 0.0000 0.0000 0.0000
*MESH_TVERT 9 1.0000 0.0000 0.0000
*MESH_TVERT 10 0.0000 1.0000 0.0000
*MESH_TVERT 11 1.0000 1.0000 0.0000
}
*MESH_NUMTVFACES 12
*MESH_TFACELIST {

*MESH_TFACE 0 9 11 10
*MESH_TFACE 1 10 8 9
*MESH_TFACE 2 8 9 11
*MESH_TFACE 3 11 10 8
*MESH_TFACE 4 4 5 7
*MESH_TFACE 5 7 6 4
*MESH_TFACE 6 0 1 3
*MESH_TFACE 7 3 2 0
*MESH_TFACE 8 4 5 7
*MESH_TFACE 9 7 6 4
*MESH_TFACE 10 0 1 3
*MESH_TFACE 11 3 2 0
}
}
*PROP_MOTIONBLUR 0
*PROP_CASTSHADOW 1
*PROP_RECVSHADOW 1
```

Как видно из кода, файл представляет собой управляющие конструкции. Ключевые слова, указывающие на новую управляющую конструкцию, отмечены знаком «*» в начале слова.

Тогда алгоритм чтения файла будет выглядеть следующим образом:

1. Начинается построковое чтение файла (вся необходимая информация для любой управляющей конструкции будет находиться в рамках одной строки, поэтому построковое чтение нам подходит).

2. Отсечение первого слова в строке.

3. Проверка первого символа в полученном первом слове: если он равен «*», то это управляющее слово.

4. Если это управляющее слово, то проводится его идентификация. Если оно относится к тем, которые нам необходимы (описывают геометрию, текстурные координаты), то обрабатываем эту строку, отрезая первые слова и переводя данные из строкового представления к числовому.

Опишем те управляющие слова, которые нам необходимы:

*MATERIAL_COUNT – количество текстур, используемых в данной *3D*-сцене.

Если значение этого параметра не равно нулю, то дальше будет идти описание каждого материала. Среди параметров *ambient*, *diffuse*, *specular*, которые отражают свойства материала (и обработку которых мы не рассматривали), в блоке описания материала будет также указана информация о текстуре.

*MATERIAL – начало блока описания материала. Далее следует указание номера материала.

Пример блока описания материала:

```
*MATERIAL_LIST {
*MATERIAL_COUNT 1
*MATERIAL 0 {
*MATERIAL_NAME "01 - Default"
*MATERIAL_CLASS "Standard"
*MATERIAL_AMBIENT 0.5882 0.5882 0.5882
*MATERIAL_DIFFUSE 0.5882 0.5882 0.5882
*MATERIAL_SPECULAR 0.9000 0.9000 0.9000
*MATERIAL_SHINE 0.1000
*MATERIAL_SHINESTRENGTH 0.0000
*MATERIAL_TRANSPARENCY 0.0000
*MATERIAL_WIRESIZE 1.0000
*MATERIAL_SHADING Blinn
*MATERIAL_XP_FALLOFF 0.0000
*MATERIAL_SELFILLUM 0.0000
*MATERIAL_FALLOFF In
```

```
*MATERIAL_XP_TYPE Filter
*MAP_DIFFUSE {
*MAP_NAME "Map #1"
*MAP_CLASS "Bitmap"
*MAP_SUBNO 1
*MAP_AMOUNT 1.0000
*BITMAP "C:\Users\anvi\Desktop\13-3.png"
*MAP_TYPE Screen
*UVW_U_OFFSET 0.0000
*UVW_V_OFFSET 0.0000
*UVW_U_TILING 1.0000
*UVW_V_TILING 1.0000
*UVW_ANGLE 0.0000
*UVW_BLUR 1.0000
*UVW_BLUR_OFFSET 0.0000
*UVW_NOUSE_AMT 1.0000
*UVW_NOISE_SIZE 1.0000
*UVW_NOISE_LEVEL 1
*UVW_NOISE_PHASE 0.0000
*BITMAP_FILTER Pyramidal
}
}
}
```

*BITMAP – адрес текстуры для данного материала.

*MATERIAL_REF – указание на то, какой номер материала назначен данному объекту.

Теперь рассмотрим описание геометрии и текстурных координат.

*GEOMOBJECT указывает на новый геометрический подобъект в загружаемой из файла трехмерной сцене.

Этот блок также включает довольно большое количество информации: – имя объекта, его начальные координаты, ориентацию в пространстве и т. д.

*MESH_NUMVERTEX указывает на количество вершин, которые будут описывать геометрию данного *3D*-объекта (подобъекта). Мы будем использовать это значение (а также значение, определяющее количество полигонов) для выделения памяти при создании экземпляра, описывающего геометрию подобъекта.

*MESH_NUMFACES указывает количество полигонов в данном геометрическом подобъекте.

*MESH_NUMTVERTEX, *MESH_NUMTVFACES – описание текстурных координат и их привязка к полигонам.

*MESH_VERTEX, *MESH_FACE, *MESH_TVERT, *MESH_TFACE – непосредственное описание вершин и полигонов (геометрии и текстурных координат). Самые часто встречаемые ключевые слова.

За данными ключевыми словами следует ряд параметров. В случае вершин это номер вершины и три ее координаты. В случае описания полигонов это номер полигона (после него стоит знак «:», который необходимо отрезать) и далее номера вершин, координаты которых должны быть использованы для построения полигона. Часть информации из данной строки мы не используем.

Разобравшись с тем, как определены данные в описании *3D*-модели, мы переходим к алгоритму загрузки *3D*-модели и ее визуализации.

Сначала объявляются необходимые для дальнейшей работы переменные, массивы для объектов и т. д.

```
// класс, выполняющий загрузку 3D-модели
public class anModelLoader
{
public anModelLoader()
{
}

// имя файла
public string FName = "";

// загружен ли (флаг)
private bool isLoad = false ;
// счетчик подобъектов
private int count_limbs;
// переменная для хранения номера текстуры
private int mat_nom = 0;

// номер дисплейного списка с данной моделью
private int thisList = 0;

// данная переменная будет указывать на количество прочитанных символов в
строке при чтении информации из файла
private int GlobalStringFrom = 0;
```

```
// массив подобъектов
LIMB[] limbs = null;

// массив для хранения текстур
TexturesForObjects[] text_objects = null;

// описание ориентации модели
Model_Prop coord = new Model_Prop();
. . .
```

Функция загрузки *3D*-модели – это построчное чтение файла и поиск управляющих слов; внесение описываемых ими данных в объекты описания геометрии; создание и загрузка текстурных объектов в память OpenGL:

```
// загрузка модели
public int LoadModel( string FileName)
{
// модель может содержать до 256 подобъектов
limbs = new LIMB[256];
// счетчик скинут
int limb_ = -1;

// имя файла
FName = FileName;

// начинаем чтение файла
StreamReader sw = File.OpenText(FileName);

// временные буферы
string a_buff = "";
string b_buff = "";
string c_buff = "";

// счетчики вершин и полигонов
int ver = 0, fac = 0;

// если строка успешно прочитана
while ((a_buff = sw.ReadLine()) != null)
{
// получаем первое слово
b_buff = GetFirstWord(a_buff, 0);
if (b_buff[0] == '*') // определяем, является ли первый символ звездочкой
```

```
{
switch (b_buff) // если да, то проверяем, какое управляющее слово содержится в
первом прочитанном слове
{
case "*MATERIAL_COUNT": // счетчик материалов
{
// получаем первое слово от символа, указанного в GlobalStringFrom
c_buff = GetFirstWord(a_buff, GlobalStringFrom);
int mat = System.Convert.ToInt32(c_buff);

// создаем объект для текстуры в памяти
text_objects = new TexturesForObjects[mat];
continue;
}

case "*MATERIAL_REF": // номер текстуры
{
// записываем для текущего подобъекта номер текстуры
c_buff = GetFirstWord(a_buff, GlobalStringFrom);
int mat_ref = System.Convert.ToInt32(c_buff);

// устанавливаем номер материала, соответствующий данной модели
limbs[limb_].SetMaterialNom(mat_ref);
continue;
}

case "*MATERIAL": // указание на материал
{
c_buff = GetFirstWord(a_buff, GlobalStringFrom);
mat_nom = System.Convert.ToInt32(c_buff);
continue;
}

case "*GEOMOBJECT": // начинается описание геометрии подобъекта
{
limb_++; // записываем в счетчик подобъектов
continue;
}

case "*MESH_NUMVERTEX": // количество вершин в подобъекте
{
c_buff = GetFirstWord(a_buff, GlobalStringFrom);
ver = System.Convert.ToInt32(c_buff);
```

```
continue;
}

case "*BITMAP": // имя текстуры
{
c_buff = ""; // обнуляем временный буфер

for ( int ax = GlobalStringFrom + 2; ax < a_buff.Length - 1; ax++)
c_buff += a_buff[ax]; // считываем имя текстуры

text_objects[mat_nom] = new TexturesForObjects(); // новый объект для текстуры

text_objects[mat_nom].LoadTextureForModel(c_buff); // загружаем текстуру

continue;
}

case "*MESH_NUMTVERTEX": // количество текстурных координат; данное
слово говорит о наличии текстурных координат, следовательно, мы должны вы-
делить память для них
{

c_buff = GetFirstWord(a_buff, GlobalStringFrom);
if (limbs[limb_] != null)
{
limbs[limb_].createTextureVertexMem( System.Convert.ToInt32(c_buff));
}
continue;
}

case "*MESH_NUMTVFACES": // память для текстурных координат (faces)
{
c_buff = GetFirstWord(a_buff, GlobalStringFrom);

if (limbs[limb_] != null)
{
// выделяем память для текстурных координат
limbs[limb_].createTextureFaceMem( System.Convert.ToInt32(c_buff));
}
continue;
}

case "*MESH_NUMFACES": // количество полигонов в подобъекте
```

```
{
c_buff = GetFirstWord(a_buff, GlobalStringFrom);
fac = System.Convert.ToInt32(c_buff);

// если было объявляющее слово *GEOMOBJECT (гарантия выполнения условия
limb_ > -1) и было указано количество вершин
if (limb_ > -1 && ver > -1 && fac > -1)
{
// создаем новый подобъект в памяти
limbs[limb_] = new LIMB(ver, fac);
}
else
{
// иначе завершаем неудачей
return -1;
}
continue;
}

case "*MESH_VERTEX": // информация о вершине
{
// подобъект создан в памяти
if (limb_ == -1)
return -2;
if (limbs[limb_] == null)
return -3;

string a1 = "", a2 = "", a3 = "", a4 = "";
// получаем информацию о координатах и номере вершины
// (получаем все слова в строке)
a1 = GetFirstWord(a_buff, GlobalStringFrom);
a2 = GetFirstWord(a_buff, GlobalStringFrom);
a3 = GetFirstWord(a_buff, GlobalStringFrom);
a4 = GetFirstWord(a_buff, GlobalStringFrom);

// преобразовываем в целое число
int NomVertex = System.Convert.ToInt32(a1);

// заменяем точки в представлении числа с плавающей точкой на запятые, чтобы
правильно выполнилась функция
// преобразование строки в дробное число
a2 = a2.Replace('.', ',');
a3 = a3.Replace('.', ',');
```

```
a4 = a4.Replace('.', ',');

// записываем информацию о вершине
limbs[limb_].vert[0, NomVertex] = (float) System.Convert.ToDouble(a2); // x
limbs[limb_].vert[1, NomVertex] = (float) System.Convert.ToDouble(a3); // y
limbs[limb_].vert[2, NomVertex] = (float) System.Convert.ToDouble(a4); // z

continue;
}

case "*MESH_FACE": // информация о полигоне
{
// подобъект создан в памяти
if (limb_ == -1)
return -2;
if (limbs[limb_] == null)
return -3;

// временные переменные
string a1 = "", a2 = "", a3 = "", a4 = "", a5 = "", a6 = "", a7 = "";

// получаем все слова в строке
a1 = GetFirstWord(a_buff, GlobalStringFrom);
a2 = GetFirstWord(a_buff, GlobalStringFrom);
a3 = GetFirstWord(a_buff, GlobalStringFrom);
a4 = GetFirstWord(a_buff, GlobalStringFrom);
a5 = GetFirstWord(a_buff, GlobalStringFrom);
a6 = GetFirstWord(a_buff, GlobalStringFrom);
a7 = GetFirstWord(a_buff, GlobalStringFrom);

// получаем номер полигона из первого слова в строке, заменив последний сим-
вол ":" после номера на флаг окончания строки
int NomFace = System.Convert.ToInt32(a1.Replace(':', '\0'));

// записываем номера вершин, которые нас интересуют
limbs[limb_].face[0, NomFace] = System.Convert.ToInt32(a3);
limbs[limb_].face[1, NomFace] = System.Convert.ToInt32(a5);
limbs[limb_].face[2, NomFace] = System.Convert.ToInt32(a7);

continue;
}

// текстурные координаты
```

```
case "*MESH_TVERT":
{
// подобъект создан в памяти
if (limb_ == -1)
return -2;
if (limbs[limb_] == null)
return -3;

// временные переменные
string a1 = "", a2 = "", a3 = "", a4 = "";

// получаем все слова в строке
a1 = GetFirstWord(a_buff, GlobalStringFrom);
a2 = GetFirstWord(a_buff, GlobalStringFrom);
a3 = GetFirstWord(a_buff, GlobalStringFrom);
a4 = GetFirstWord(a_buff, GlobalStringFrom);

// преобразуем первое слово в номер вершины
int NomVertex = System.Convert.ToInt32(a1);

// заменяем точки в представлении числа с плавающей точкой на запятые, чтобы
правильно выполнилась функция
// преобразование строки в дробное число
a2 = a2.Replace('.', ',');
a3 = a3.Replace('.', ',');
a4 = a4.Replace('.', ',');

// записываем значение вершины
limbs[limb_].t_vert[0, NomVertex] = (float) System.Convert.ToDouble(a2); // x
limbs[limb_].t_vert[1, NomVertex] = (float) System.Convert.ToDouble(a3); // y
limbs[limb_].t_vert[2, NomVertex] = (float) System.Convert.ToDouble(a4); // z

continue;
}

// привязка текстурных координат к полигонам
case "*MESH_TFACE":
{
// подобъект создан в памяти
if (limb_ == -1)
return -2;
if (limbs[limb_] == null)
return -3;
```

```
// временные переменные
string a1 = "", a2 = "", a3 = "", a4 = "";

// получаем все слова в строке
a1 = GetFirstWord(a_buff, GlobalStringFrom);
a2 = GetFirstWord(a_buff, GlobalStringFrom);
a3 = GetFirstWord(a_buff, GlobalStringFrom);
a4 = GetFirstWord(a_buff, GlobalStringFrom);

// преобразуем первое слово в номер полигона
int NomFace = System.Convert.ToInt32(a1);

// записываем номера вершин, которые описывают полигон
limbs[limb_].t_face[0, NomFace] = System.Convert.ToInt32(a2);
limbs[limb_].t_face[1, NomFace] = System.Convert.ToInt32(a3);
limbs[limb_].t_face[2, NomFace] = System.Convert.ToInt32(a4);

continue;
}
}
}
}
// пересохраняем количество полигонов
count_limbs = limb_;


// получаем ID для создаваемого дисплейного списка
int nom_l = Gl.glGenLists(1);
thisList = nom_l;
// генерируем новый дисплейный список
Gl.glNewList(nom_l, Gl.GL_COMPILE);
// отрисовываем геометрию
CreateList();
// завершаем дисплейный список
Gl.glEndList();

// загрузка завершена
isLoad = true ;

return 0;
}
```

Как видно из кода, последним этапом была генерация дисплейного списка – мы сгенерировали его номер, затем начали создание дисплейного списка и вызвали функцию *CreateList()*;

В данной функции производилась визуализация полученной геометрии *3D*-модели с учетом установленных текстур и текстурных координат.

Все довольно просто: мы в цикле отрисовываем массивы вершин, перебирая массив подобъектов, затем массив полигонов. В случае необходимости включаем режим текстурирования и указываем текстурные координаты.

Код данной функции:

```
// функция отрисовки
private void CreateList()
{
// сохраняем текущую матрицу

Gl.glPushMatrix();

// проходим циклом по всем подобъектам
for ( int l = 0; l <= count_limbs; l++)
{
// если текстура необходима
if( limbs[l].NeedTexture() )
if( text_objects[limbs[l].GetTextureNom()] != null ) // текстурный объект существует
{
Gl.glEnable( Gl.GL_TEXTURE_2D); // включаем режим текстурирования

// ID текстуры в памяти
u int nn = text_objects[limbs[l].GetTextureNom()].GetTextureObj();
// активируем (привязываем) эту текстуру
Gl.glBindTexture( Gl.GL_TEXTURE_2D, nn);
}

Gl.glEnable( Gl.GL_NORMALIZE);

// начинаем отрисовку полигонов
Gl.glBegin( Gl.GL_TRIANGLES);

// по всем полигонам
for ( int i = 0; i < limbs[l].VandF[1]; i++)
```

```
{
// временные переменные, чтобы код был более понятен
float x1, x2, x3, y1, y2, y3, z1, z2, z3 = 0;

// вытаскиваем координаты треугольника (полигона)
x1 = limbs[l].vert[0, limbs[l].face[0, i]];
x2 = limbs[l].vert[0, limbs[l].face[1, i]];
x3 = limbs[l].vert[0, limbs[l].face[2, i]];
y1 = limbs[l].vert[1, limbs[l].face[0, i]];
y2 = limbs[l].vert[1, limbs[l].face[1, i]];
y3 = limbs[l].vert[1, limbs[l].face[2, i]];
z1 = limbs[l].vert[2, limbs[l].face[0, i]];
z2 = limbs[l].vert[2, limbs[l].face[1, i]];
z3 = limbs[l].vert[2, limbs[l].face[2, i]];

// рассчитываем нормаль
float n1 = (y2 - y1) * (z3 - z1) - (y3 - y1) * (z2 - z1);
float n2 = (z2 - z1) * (x3 - x1) - (z3 - z1) * (x2 - x1);
float n3 = (x2 - x1) * (y3 - y1) - (x3 - x1) * (y2 - y1);

// устанавливаем нормаль
Gl.glNormal3f(n1, n2, n3);

// если установлена текстура
if (limbs[l].NeedTexture() && (limbs[l].t_vert != null) && (limbs[l].t_face != null))
{
// устанавливаем текстурные координаты для каждой вершины и сами вершины
Gl.glTexCoord2f(limbs[l].t_vert[0, limbs[l].t_face[0, i]], limbs[l].t_vert[1,
limbs[l].t_face[0, i]]);
Gl.glVertex3f(x1, y1, z1);

Gl.glTexCoord2f(limbs[l].t_vert[0, limbs[l].t_face[1, i]], limbs[l].t_vert[1,
limbs[l].t_face[1, i]]);
Gl.glVertex3f(x2, y2, z2);

Gl.glTexCoord2f(limbs[l].t_vert[0, limbs[l].t_face[2, i]], limbs[l].t_vert[1,
limbs[l].t_face[2, i]]);
Gl.glVertex3f(x3, y3, z3);
}
else // иначе – отрисовка только вершин
{
Gl.glVertex3f(x1, y1, z1);
Gl.glVertex3f(x2, y2, z2);
```

```
Gl.glVertex3f(x3, y3, z3);
}
}

// завершаем отрисовку
Gl.glEnd();
Gl.glDisable( Gl.GL_NORMALIZE);

// отключаем текстурирование
Gl.glDisable( Gl.GL_TEXTURE_2D);
}

// возвращаем сохраненную ранее матрицу
Gl.glPopMatrix();
}
```

Также рассмотрим код функции *GetFirstWord*, которая отвечает за получение первого слова в строке.

```
// функция получения первого слова строки
private string GetFirstWord( string word, int from)
{
// from указывает на позицию, начиная с которой будет выполняться чтение файла
char a = word[from]; // первый символ
string res_buff = ""; // временный буфер
int L = word.Length; // длина слова

if (word[from] == ' ' || word[from] == '\t') // если первый символ, с которого предстоит искать слово, является пробелом или знаком табуляции.
{
// необходимо вычислить наличие секции пробелов или знаков табуляции и исключить их
int ax = 0;
// проходим до конца слова
for( ax = from; ax < L; ax++)
{
a = word[ax];
if( a != ' ' && a != '\t') // если встречаем символ пробела или табуляции
break ; // выходим из цикла
// таким образом, мы исключаем все последовательности пробелов или знаков табуляции, с которых могла начинаться переданная строка
```

```
}

if( ax == L) // если вся представленная строка является набором пробелов или знаков табуляции – возвращаем res_buff
return res_buff;
else
from = ax; // иначе сохраняем значение ax
}
int bx = 0;

// теперь, когда пробелы и табуляция исключены, мы непосредственно вычисляем слово
for (bx = from; bx < L; bx++)
{
// если встретили знак пробела или табуляции, завершаем чтение слова
if (word[bx] == ' ' || word[bx] == '\t')
break ;
// записываем символ во временный буфер, постепенно получая таким образом слово
res_buff += word[bx];
}

// если дошли до конца строки
if (bx == L)
bx--; // убираем последнее значение

GlobalStringFrom = bx; // позиция в данной строке для чтения следующего слова в данной строке

return res_buff; // возвращаем слово
}
```

Функция, которая будет использоваться для вызова отрисовки модели из оболочки программы:

```
// функция отрисовки 3D-модели
public void DrawModel()
{
// если модель не загружена, возврат из функции
if (!isLoad)
return ;

// сохраняем матрицу
```

```
Gl.glPushMatrix();

// масштабирование по умолчанию
Gl.glScalef(0.05f, 0.05f, 0.05f);

// вызов дисплейного списка

Gl.glCallList(thisList);

// возврат матрицы
Gl.glPopMatrix();
}
```

В коде оболочки произошли небольшие изменения, которые сводятся к нескольким строкам для описания переменных класса, загрузки *3D*-модели, выбора файла и отрисовки модели.

Должна быть объявлена переменная:

```
anModelLoader Model = null;
```

Функция *Form1_Load*:

```
. . .
// опции для загрузки файла
openFileDialog1.Filter = "ase files (*.ase)|*.ase|All files (*.*)|*.*";
. . .
```

Отрисовка модели в функции *Draw*:

```
. . .

if( Model != null)
Model.DrawModel();
. . .
```

Загрузка модели:

```
// загрузка модели
private void ToolStripMenuItem_Click( object sender, EventArgs e)
{
if (openFileDialog1.ShowDialog() == DialogResult.OK)
{
Model = new anModelLoader();
Model.LoadModel(openFileDialog1.FileName);
RenderTimer.Start();
}
}
```

На приведенных ниже скриншотах можно увидеть сцены в окне *3D-Studio Max* (рис. 4.7, 4.9, 4.10) и сцены, визуализированные после загрузки *3D*-модели из формата *ASE* (рис. 4.8, 4.11).

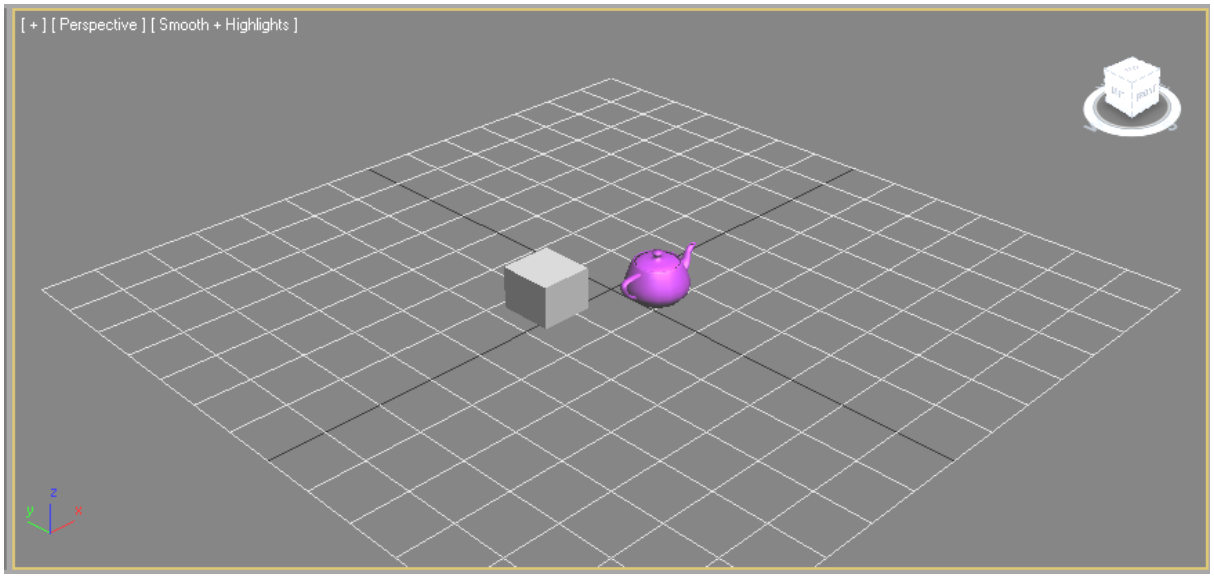

Рис. 4.7

Во втором примере модель (рис. 4.9) текстурирована (см. рис. 4.11). По качеству она уступает визуализированной модели в *3D Studio Max*, но это уже задача создания качественного рендера для вашего приложения (свойств материалов, освещения, сглаживания, различных шейдеров, что поднимет уровень визуализации сцены, но является уже совсем другой задачей).

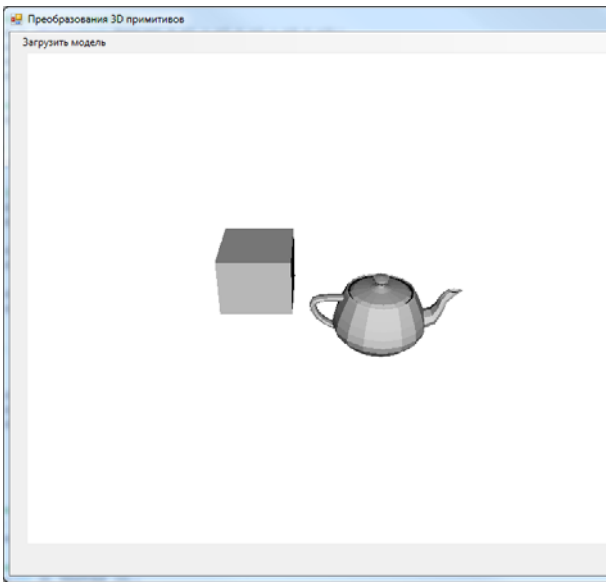

Рис. 4.8

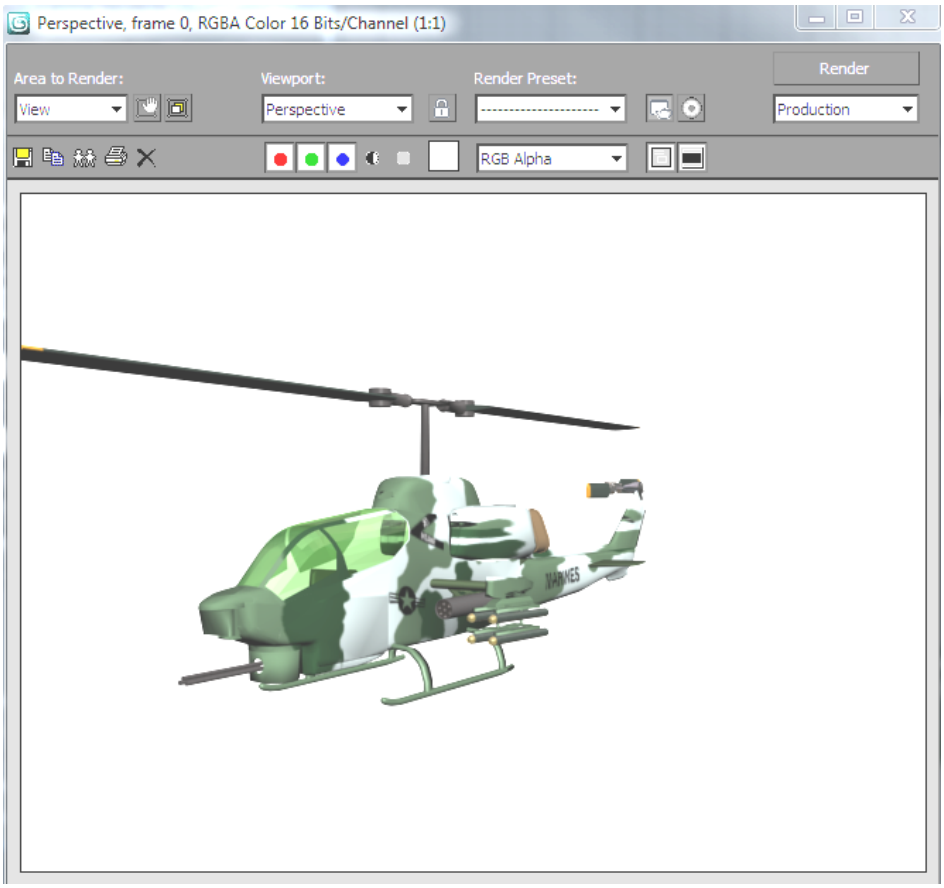

Рис. 4.9

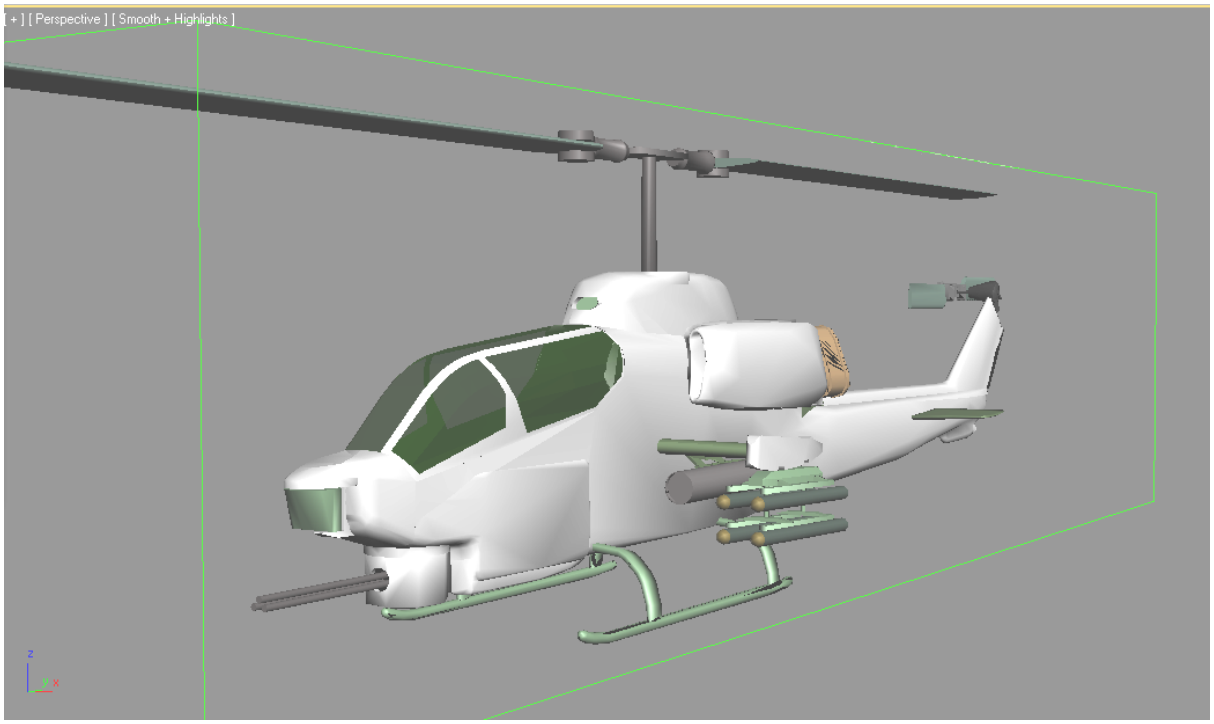

Рис.4.10

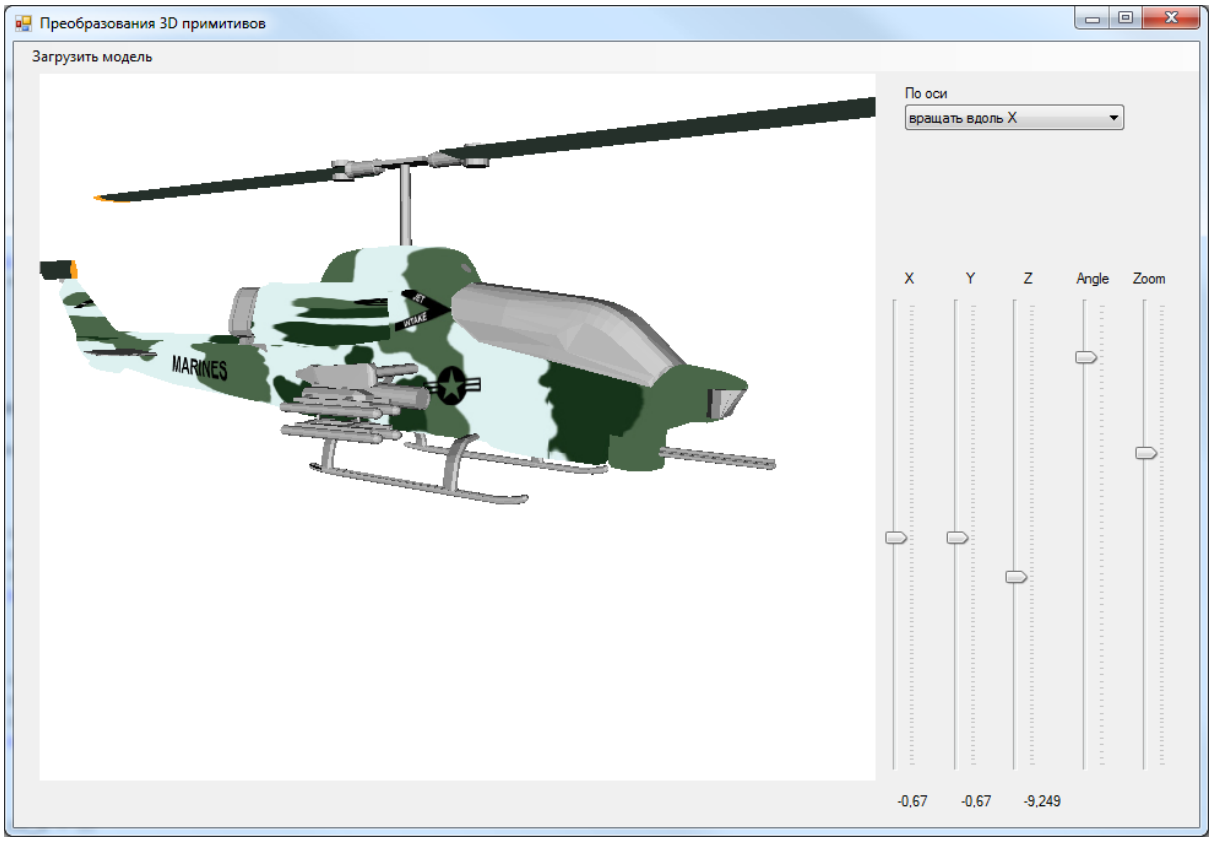

Рис. 4.11

Практическое задание

1. Ознакомиться по методическим указаниям и литературе с 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 выполнении п. 4.2 реализовать программу создания сцены, содержащей ландшафт минимум из 5 подобъектов (выполнены и экспортированы из *3D Studio Max*). Все объекты должны быть протекстурированы методом наложения текстурной развертки. Помимо ландшафта загрузить и разместить в сцене еще 3 протекстурированные модели. Тематика сцены определяется вариантом задания.

4. Представить визуализацию (скриншоты) моделей в *3D Studio Max*, а также полученную визуализацию в реализованной программе.

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*, исполняемые файлы.

Контрольные вопросы

1. Как осуществляется представление *3D*-объектов в компьютерной графике?
2. Каковы способы экспорта данных из *3D Studio Max* в *OpenGL*?
3. Как производится визуализация *3D*-модели в *OpenGL*?

Тема 5. **СИСТЕМЫ ЧАСТИЦ**

Цель изучения темы. Изучение методов использования в компьютерной графике *3D*-объектов, не имеющих чётких геометрических границ и описываемых стохастическими алгоритмами.

5.1. Системы частиц в компьютерной графике

Система частиц – используемый в компьютерной графике способ представления *3D*-объектов, не имеющих чётких геометрических границ (облака, туманности, взрывы, струи пара, шлейфы от ракет, дым, снег, дождь и т. п.). Системы частиц могут быть реализованы как в двумерной графике, так и в трёхмерной.

Система частиц состоит из определенного (фиксированного или произвольного) количества частиц. Математически каждая частица представляется как материальная точка с дополнительными атрибутами, такими как скорость, цвет, ориентация в пространстве, угловая скорость и т. п. В ходе работы программы, моделирующей частицы, каждая частица изменяет своё состояние по определенному общему для всех частиц системы закону. Например, частица может подвергаться воздействию гравитации, менять размер, цвет, скорость и так далее, после проведения всех расчётов она визуализируется. Частица может быть визуализирована точкой, треугольником, спрайтом или даже полноценной трехмерной моделью.

В большинстве реализаций новые частицы испускаются так называемым эмиттером. Эмиттером может быть точка, тогда новые частицы будут возникать в одном месте пространства. Так можно

смоделировать, например, взрыв: эмиттером будет его эпицентр. Эмиттером может быть отрезок прямой или плоскость: например, частицы дождя или снега должны возникать на высоко расположенной горизонтальной плоскости. Эмиттером может быть и произвольный геометрический объект, в этом случае новые частицы будут возникать на всей его поверхности.

На протяжении жизни частица редко остаётся в покое. Частицы могут двигаться, вращаться, менять свой цвет и/или прозрачность, сталкиваться с трёхмерными объектами. Часто у частиц задана максимальная продолжительность жизни, по истечении которой частица исчезает.

В трёхмерных приложениях реального времени (например, в компьютерных играх) обычно считается, что частицы не отбрасывают тени друг на друга, а также на окружающую геометрию и что они не поглощают, а излучают свет. Без этих упрощений обсчёт системы частиц будет требовать больше ресурсов: в случае с поглощением света потребуется сортировать частицы по удалённости от камеры, а в случае с тенями каждую частицу придётся рисовать несколько раз.

5.2. Классы для описания системы частиц

Класс для описания частицы

Создание приложения начнем с разработки класса, описывающего частицу. Создайте новый класс *Partlice*.

Перейдите к редактированию файла *Partlice.cs*.

Наш класс будет иметь ряд параметров, влияющих на расчет позиции частицы через определенный промежуток времени. Эти параметры могут быть установлены при инициализации частицы (передаются в конструктор класса), а также установлены напрямую.

Класс имеет ряд функций для изменения каких-либо параметров, например для изменения вектора скорости по одной из осей (необходимо для простого отражения частицы при столкновении).

Функция *UpdatePosition* принимает в качестве параметра текущий временной штамп и, используя разницу во времени после последнего обновления, вычисляет новую позицию частицы, а также отвечает за затухание ее ускорения.

Текущие координаты частицы могут быть в любой момент времени получены с помощью функций *GetPosition*()*.

Код класса прост и полностью комментирован.

Класс будет выглядеть следующим образом:

```
namespace ParticlesSystems
{
class Partlice
{
// позиция частицы
private float[] position = new float[3];
// размер
private float _size;
// время жизни
private float _lifeTime;

// вектор гравитации
private float[] Grav = new float[3];
// ускорение частицы
private float[] power = new float[3];
// коэффицент затухания силы
private float attenuation;

// набранная скорость
private float[] speed = new float[3];

// временной интервал активации частицы
private float LastTime = 0;

// конструктор класса
public Partlice( float x, float y, float z, float size, float lifeTime, float start_time)
{
// записываем все начальные настройки частицы, устанавливаем начальный ко-
эффицент затухания
// и обнуляем скорость и силу, приложенную к частице
_size = size;
_lifeTime = lifeTime;

position[0] = x;
position[1] = y;
position[1] = z;

speed[0] = 0;
speed[1] = 0;
speed[2] = 0;
```

```
Grav[0] = 0;
Grav[1] = -9.8f;
Grav[2] = 0;

attenuation = 3.33f;

power[0] = 0;
power[0] = 0;
power[0] = 0;

LastTime = start_time;
}

// функция установки ускорения, действующего на частицу
public void SetPower( float x, float y, float z)
{
power[0] = x;
power[1] = y;
power[2] = z;
}

// инвертирование скорости частицы по заданной оси с указанным затуханием
// удобно использовать для простой демонстрации столкновений, например с
землей
public void InvertSpeed( int os, float attenuation)
{
speed[os] *= -1 * attenuation;
}

// получение размера частицы
public float GetSize()
{
return _size;
}

// установка нового значения затухания
public void setAttenuation (float new_value)
{
attenuation = new_value;
}

// обновление позиции частицы
public void UpdatePosition (float timeNow)
```

```
{
// определяем разницу во времени, прошедшую с последнего обновления
// позиции частицы (ведь таймер может быть нефиксированный)
float dTime = timeNow - LastTime;
_lifeTime -= dTime;

// обновляем последнюю отметку временного интервала
LastTime = timeNow;

// перерассчитываем ускорение, движущее частицу, с учетом затухания
for( int a = 0; a < 3; a++)
{
if (power[a] > 0)
{
power[a] -= attenuation * dTime;

if (power[a] <= 0)
power[a] = 0;
}

// перерассчитываем позицию частицы с учетом гравитации, вектора ускорения и
прошедшего промежутка времени
position[a] += (speed[a] * dTime + (Grav[a] + power[a]) * dTime * dTime);

// обновляем скорость частицы
speed[a] += (Grav[a] + power[a]) * dTime;
}
}

// проверка, не закончилось ли время жизни частицы
public bool isLife()
{
if (_lifeTime > 0)
{
return true;
}
else
{
return false;
}
}

// получение координат частицы
```

```
public float GetPositionX()
{
return position[0];
}
public float GetPositionY()
{
return position[1];
}
public float GetPositionZ()
{
return position[2];
}
}
}
```

Класс, реализующий взрыв

Теперь мы можем реализовывать различные поведения частиц. Мы опишем класс, демонтирующий «взрыв» на основе большого количества частиц.

Класс, реализующий взрыв, будет параметризирован, чтобы в данном приложении можно было демонстрировать поведение системы частиц при различных настройках.

Класс будет отвечать за то, чтобы инициализировать массив частиц и в момент взрыва, поместив их в эпицентр взрыва, сгенерировать всем частицам случайное направление и случайное ускорение в пределах, заданных мощностью взрыва.

Далее выполняются просчет текущего состояния взрыва и отрисовка частиц с использованием дисплейного списка.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

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

namespace ParticlesSystems
{
class Explosion
{
```

```
// позиция взрыва
private float[] position = new float[3];
// мощность
private float _power;
// максимальное количество частиц
private int MAX_PARTICLES = 1000;
// текущее установленное количество частиц
private int _particles_now;

// активирован
private bool isStart = false ;

// массив частиц на основе созданного ранее класса
private Partlice[] PartliceArray;

// дисплейный список для рисования частицы создан
private bool isDisplayList = false ;
// номер дисплейного списка для отрисовки
private int DisplayListNom = 0;

// конструктор класса; в него передаются координаты взрыва, мощность и коли-
чество частиц
public Explosion( float x, float y, float z, float power, int particle_count)
{
position[0] = x;
position[1] = y;
position[2] = z;

_particles_now = particle_count;
_power = power;

// если число частиц превышает максимально разрешенное
if( particle_count > MAX_PARTICLES)
{
particle_count = MAX_PARTICLES;
}

// создаем массив частиц необходимого размера
PartliceArray = new Partlice[particle_count];
}

// функция обновления позиции взрыва
public void SetNewPosition( float x, float y, float z)
```

```
{
position[0] = x;
position[1] = y;
position[2] = z;
}

// установка нового значения мощности взрыва
public void SetNewPower( float new_power)
{
_power = new_power;
}

// создание дисплейного списка для отрисовки частицы (так как отрисовывать
даже небольшой полигон такое количество раз очень накладно)
private void CreateDisplayList()
{
// генерация дисплейного списка
DisplayListNom = Gl.glGenLists(1);

// начало создания списка
Gl.glNewList(DisplayListNom, Gl.GL_COMPILE);

// режим отрисовки треугольника
Gl.glBegin( Gl.GL_TRIANGLES);

// задаем форму частицы
Gl.glVertex3d(0, 0, 0);
Gl.glVertex3d(0.02f, 0.02f, 0);
Gl.glVertex3d(0.02f, 0, -0.02f);

Gl.glEnd();

// завершаем отрисовку частицы
Gl.glEndList();

// флаг – дисплейный список создан
isDisplayList = true;
}

// функция, реализующая взрыв
public void Boooom( float time_start)
{
// инициализируем экземпляр класса Random
Random rnd = new Random();
```

```
// если дисплейный список не создан, надо его создать
if (!isDisplayList)
{
CreateDisplayList();
}

// по всем частицам
for ( int ax = 0; ax < _particles_now; ax++)
{
// создаем частицу
PartliceArray[ax] = new Partlice(position[0], position[1], position[2], 5.0f, 10,
time_start);

// случайным образом генерируем ориентацию вектора ускорения для данной
частицы
int direction_x = rnd.Next(1, 3);
int direction_y = rnd.Next(1, 3);
int direction_z = rnd.Next(1, 3);

// если сгенерировано число 2, то мы заменяем его на –1.
if (direction_x == 2)
direction_x = -1;


if (direction_y == 2)
direction_y = -1;

if (direction_z == 2)
direction_z = -1;

// задаем мощность в промежутке от 5 до 100 % от указанной (чтобы частицы
имели разное ускорение)
float _power_rnd = rnd.Next((int)_power/20, (int)_power);
// устанавливаем затухание, равное 50 % мощности
PartliceArray[ax].setAttenuation(_power / 2.0f);
// устанавливаем ускорение частицы, еще раз генерируя случайное число
// таким образом мощность определится от 10 до 100 % полученной
// Здесь же применяем ориентацию для векторов ускорения
PartliceArray[ax].SetPower(_power_rnd * ((float)rnd.Next(100, 1000) / 1000.0f) *
direction_x, _power_rnd * ((float)rnd.Next(100, 1000) / 1000.0f) * direction_y,
_power_rnd * ((float)rnd.Next(100, 1000) / 1000.0f) * direction_z);
}
```

```
// взрыв активирован
isStart = true ;
}

// калькуляция текущего взрыва
public void Calculate( float time)
{
// только в том случае, если взрыв уже активирован,
if (isStart)
{
// проходим циклом по всем частицам
for ( int ax = 0; ax < _particles_now; ax++)
{
// если время жизни частицы еще не вышло
if (PartliceArray[ax].isLife())
{
// обновляем позицию частицы
PartliceArray[ax].UpdatePosition(time);

// сохраняем текущую матрицу
Gl.glPushMatrix();
// получаем размер частицы
float size = PartliceArray[ax].GetSize();

// выполняем перемещение частицы в необходимую позицию
Gl.glTranslated(PartliceArray[ax].GetPositionX(), PartliceArray[ax].GetPositionY(),
PartliceArray[ax].GetPositionZ());
// масштабируем ее в соответствии с ее размером
Gl.glScalef(size, size, size);
// вызываем дисплейный список для отрисовки частицы из кеша видеоадаптера
Gl.glCallList(DisplayListNom);
// возвращаем матрицу
Gl.glPopMatrix();

// отражение от "земли"
// если координата Y стала меньше нуля (удар о землю)
if (PartliceArray[ax].GetPositionY() < 0)
{
// инвертируем проекцию скорости на ось Y, как будто частица ударилась и от-
скочила от земли
// причем скорость затухает на 40 %
PartliceArray[ax].InvertSpeed(1,0.6f);
} } } } } } }
```

5.3. Программа с использованием системы частиц

Разработка программы начинается с создания оболочки**.**

Создайте окно программы и разместите на нем элемент *openglsimplecontrol*, как показано на рис. 5.1, после этого установите его размеры 500 × 500. Переименуйте данный объект, назвав его *AnT*.

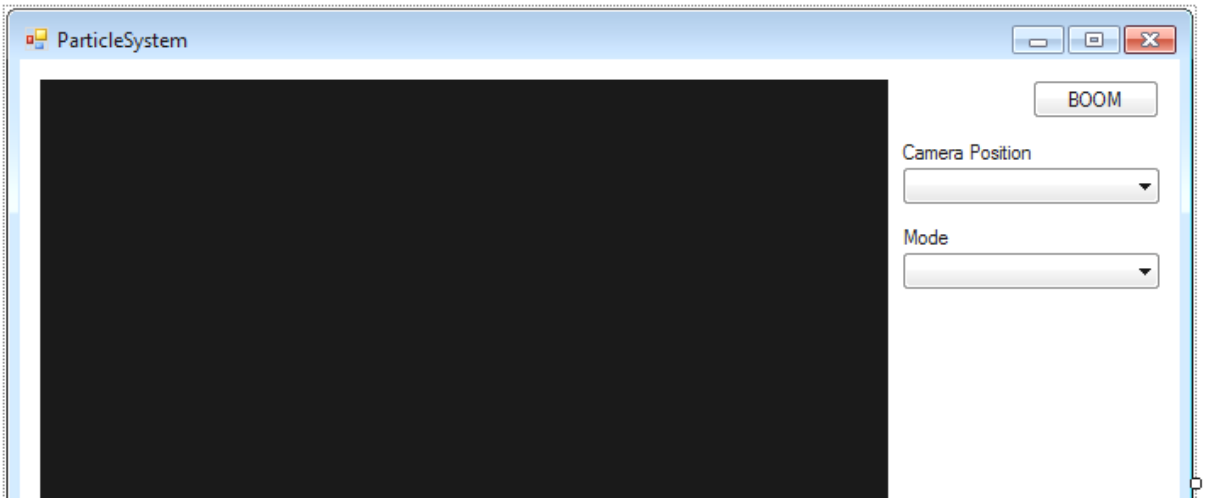

Рис. 5.1

Для визуализации будет использоваться таймер: после инициализации окна он будет генерировать событие, называемое "тиком" таймера, раз в 30 мс, добавьте элемент таймер, переименуйте экземпляр в *RenderTimer* и установите время "тика" 30 мс (как показано на рис. 5.2), а также добавьте ему событие для обработки "тика".

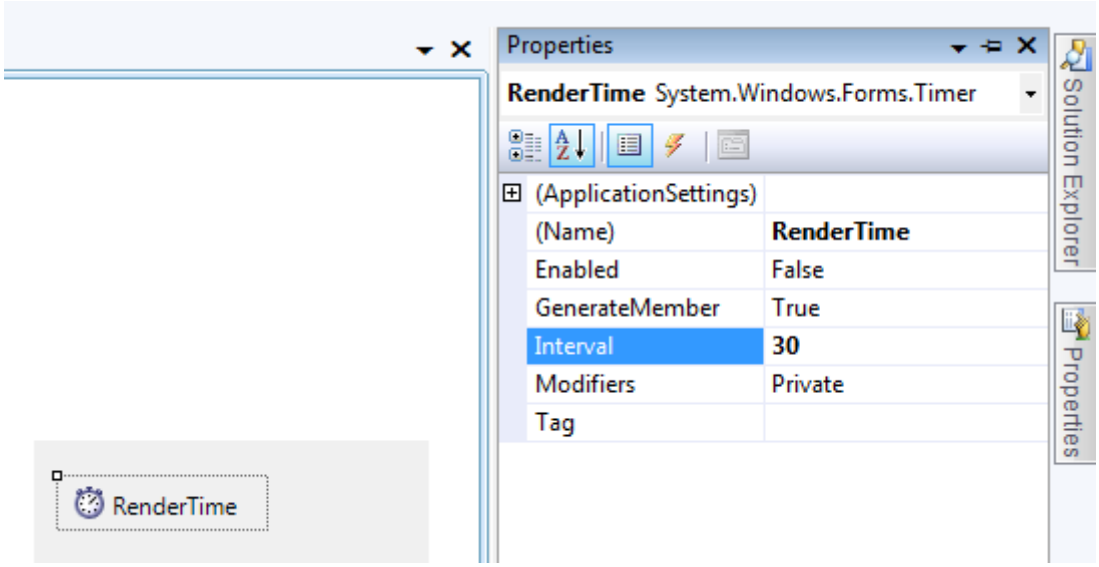

Рис. 5.2

Обратите внимание на значение элементов *comboBox*, показанных на рис. 5.3.

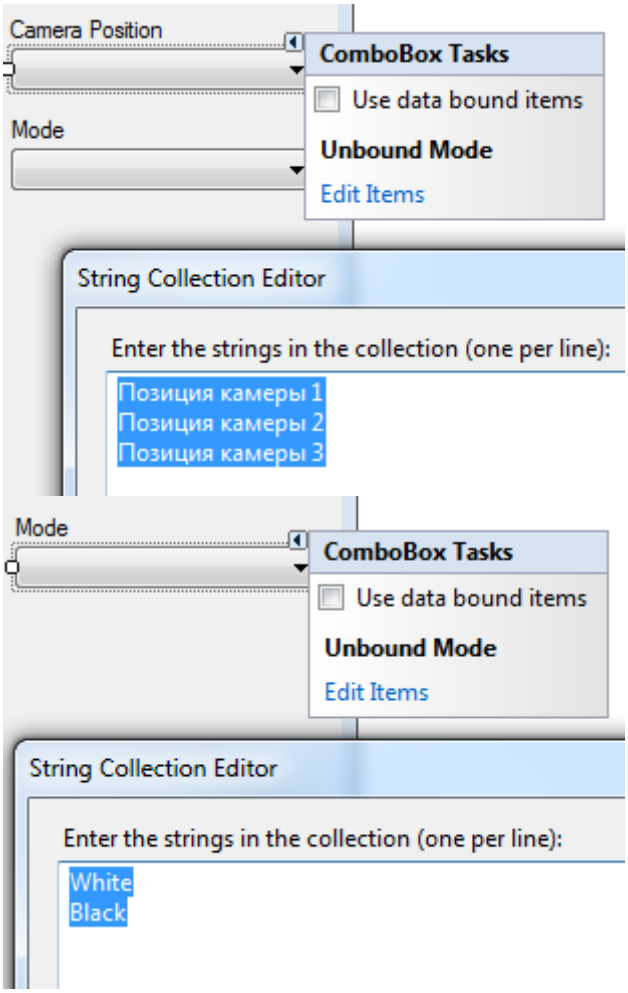

Рис. 5.3

Как и всегда, объявим ряд переменных, а также произведем инициализацию *OpenGL*. Помимо этого установим настройки для определения различных камер.

```
// отсчет времени
float global_time = 0; // массив с параметрами установки камеры
private float[,] camera_date = new float[3, 7];

// экземпляра класса Explosion
private Explosion BOOOOM_1 = new Explosion(1, 10, 1, 300, 500);

public Form1()
{

InitializeComponent();
```

```
AnT.InitializeContexts();
}

// генератор случайных чисел
Random rnd = new Random();

// загрузка формы
private void Form1_Load( object sender, EventArgs e)
{
// инициализация OpenGL
Glut.glutInit();
Glut.glutInitDisplayMode( Glut.GLUT_RGB | Glut.GLUT_DOUBLE |
Glut.GLUT_DEPTH);

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

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

Gl.glMatrixMode( Gl.GL_PROJECTION);
Gl.glLoadIdentity();

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

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

Gl.glEnable( Gl.GL_DEPTH_TEST);
Gl.glEnable( Gl.GL_LIGHTING);
Gl.glEnable( Gl.GL_LIGHT0);
Gl.glEnable( Gl.GL_COLOR_MATERIAL);

// установка начальных значений элементов comboBox
comboBox1.SelectedIndex = 1;
comboBox2.SelectedIndex = 0;

// позиция камеры 1:

camera_date[0,0] = -3;
camera_date[0,1] = 0;
camera_date[0,2] = -20;
camera_date[0,3] = 0;
camera_date[0,4] = 1;
camera_date[0,5] = 0;
```

```
camera_date[0,6] = 0;

// позиция камеры 2:

camera_date[1, 0] = -3;
camera_date[2, 1] = 2;
camera_date[1, 2] = -20;
camera_date[1, 3] = 30;
camera_date[1, 4] = 1;
camera_date[1, 5] = 0;
camera_date[1, 6] = 0;

// позиция камеры 3:

camera_date[2, 0] = -3;
camera_date[2, 1] = 2;
camera_date[2, 2] = -20;
camera_date[2, 3] = 30;
camera_date[2, 4] = 1;
camera_date[2, 5] = 1;
camera_date[2, 6] = 0;

// активация таймера
RenderTimer.Start();
}
```

Код программы снабжен комментариями:

```
public partial class Form1 : Form
{
// отсчет времени
float global_time = 0; // массив с параметрами установки камеры
private float[,] camera_date = new float[3, 7];

// экземпляр класса Explosion
private Explosion BOOOOM_1 = new Explosion(1, 10, 1, 300, 500);

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

// генератор случайных чисел
```

```
Random rnd = new Random();

// загрузка формы
private void Form1_Load( object sender, EventArgs e)
{
// инициализация OpenGL
Glut.glutInit();
Glut.glutInitDisplayMode( Glut.GLUT_RGB | Glut.GLUT_DOUBLE |
Glut.GLUT_DEPTH);

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

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

Gl.glMatrixMode( Gl.GL_PROJECTION);
Gl.glLoadIdentity();

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

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

Gl.glEnable( Gl.GL_DEPTH_TEST);
Gl.glEnable( Gl.GL_LIGHTING);
Gl.glEnable( Gl.GL_LIGHT0);
Gl.glEnable( Gl.GL_COLOR_MATERIAL);

// установка начальных значений элементов comboBox
comboBox1.SelectedIndex = 1;
comboBox2.SelectedIndex = 0;

// позиция камеры 1:
camera_date[0,0] = -3;
camera_date[0,1] = 0;
camera_date[0,2] = -20;
camera_date[0,3] = 0;
camera_date[0,4] = 1;
camera_date[0,5] = 0;
camera_date[0,6] = 0;

// позиция камеры 2:
camera_date[1, 0] = -3;
camera_date[2, 1] = 2;
```

```
camera_date[1, 2] = -20;
camera_date[1, 3] = 30;
camera_date[1, 4] = 1;
camera_date[1, 5] = 0;
camera_date[1, 6] = 0;

// позиция камеры 3:
camera_date[2, 0] = -3;
camera_date[2, 1] = 2;
camera_date[2, 2] = -20;
camera_date[2, 3] = 30;
camera_date[2, 4] = 1;
camera_date[2, 5] = 1;
camera_date[2, 6] = 0;

// активация таймера
RenderTimer.Start();
}

// отклик таймера
private void RenderTimer_Tick( object sender, EventArgs e)
{
// отсчитываем время
global_time += (float)RenderTimer.Interval / 1000;
// вызываем функцию отрисовки
Draw();
}

// функция отрисовки сцены
private void Draw()
{
// в зависимости от установленного режима отрисовываем сцену в черном или
белом цвете
if (comboBox2.SelectedIndex == 0)
{
// цвет очистки окна
Gl.glClearColor(255, 255, 255, 1);
}
else
{
Gl.glClearColor(0, 0, 0, 1);
}
```

```
Gl.glClear( Gl.GL_COLOR_BUFFER_BIT | Gl.GL_DEPTH_BUFFER_BIT);
Gl.glLoadIdentity();
// в зависимости от установленного режима отрисовываем сцену в черном или белом цвете
if (comboBox2.SelectedIndex == 0)
{
// цвет рисования
Gl.glColor3d(0, 0, 0);
}
else
{
Gl.glColor3d(255,255,255);
}

Gl.glPushMatrix();
// определяем установленную камеру
int camera = comboBox1.SelectedIndex;

// используем параметры для установленной камеры
Gl.glTranslated(camera_date[camera, 0], camera_date[camera, 1], cam-
era_date[camera, 2]);
Gl.glRotated(camera_date[camera, 3], camera_date[camera, 4], camera_date[camera, 5], camera_date[camera, 6]);

Gl.glPushMatrix();
// отрисовываем сеточную плоскость, которая нам будет напоминать, где находится земля
DrawMatrix(5);

// выполняем просчет взрыва
BOOOOM_1.Calculate(global_time);

Gl.glPopMatrix();

Gl.glPopMatrix();
Gl.glFlush();

// обновляем окно
AnT.Invalidate();
}

// функция для отрисовки матрицы
private void DrawMatrix( int x)
```

```
{
float quad_size = 1;

// две последовательные линии после пересечения создадут "матрицу", чтобы мы могли понимать, где находится земля
Gl.glBegin( Gl.GL_LINES);

for ( int ax = 0; ax < x+1; ax++)
{
Gl.glVertex3d(quad_size * ax, 0, 0);
Gl.glVertex3d(quad_size * ax, 0, quad_size * x);
}

for ( int bx = 0; bx < x+1; bx++)
{
Gl.glVertex3d(0, 0, quad_size * bx);
Gl.glVertex3d(quad_size * x, 0, quad_size * bx);
}

Gl.glEnd();
}

// обработка кнопки, реализующей взрыв
private void button1_Click( object sender, EventArgs e)
{
Random rnd = new Random();
// устанавливаем новые координаты взрыва
BOOOOM_1.SetNewPosition(rnd.Next(1, 5), rnd.Next(1, 5), rnd.Next(1, 5));
// случайную силу
BOOOOM_1.SetNewPower(rnd.Next(20,80));
// и активируем сам взрыв
BOOOOM_1.Boooom(global_time);
}
```

Практическое задание

1. Ознакомиться по методическим указаниям и литературе с теоретическим материалом.

2. Выполнить действия, приведенные в пп. 5.2, 5.3. 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 пп. 5.2, 5.3 изменить программу таким образом, чтобы частицы вращались в процессе полета. Взрыв должен быть применен в сцене, разработанной при изучении темы 4 согласно варианту задания.

4. Создать направленный взрыв. Направление взрыва выбрать произвольным. Увеличить сетку поверхности.

5. Добавить два новых положения камеры, которые максимально удобно и зрелищно демонстрировали бы направленность взрыва.

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*, исполняемые файлы.

Контрольные вопросы

1. Охарактеризуйте системы частиц в компьютерной графике.
2. Каковы средства описания частиц в *OpenGL*?
3. Как работает программа визуализация систем частиц в *OpenGL*?

ЗАКЛЮЧЕНИЕ

В учебном пособии рассмотрен ряд алгоритмов и инструментов программирования трехмерной компьютерной графики, применяемых при решении практических задач.

Предлагаемое издание направлено на изучение методов построения изображений на базе графических средств языка программирования C# и содержит материал по моделированию тел вращения, выполнению геометрических преобразований объектов в пространстве, текстурированию поверхностей, представлению и визуализации трехмерных моделей, использованию составных объемных объектов, описываемых стохастическими алгоритмами. Каждая тема содержит самостоятельный материал, который иллюстрируется текстами программ, формирующих различные изображения графических объектов с реализацией всех рассматриваемых подходов.

Пособие охватывает далеко не все аспекты программирования трехмерной компьютерной графики на языке C# с использованием графической библиотеки OpenGL и не все методы и алгоритмы построения трехмерных графических моделей, однако оно дает основу для дальнейшего углубленного изучения и практического использования этих средств при разработке прикладного программного обеспечения.

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. *Жигалов, И. Е.* Программирование компьютерной графики : учеб. пособие / И. Е. Жигалов, И. А. Новиков ; Владим. гос. ун-т. – 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. – ISBN 5-89368-459-1.

3. *Он же.* 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, 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.

11. *Жигалов, И. Е.* Программирование двухмерной компьютерной графики : учеб. пособие / Е. И. Жигалов, И. А. Новиков ; Владим. гос. ун-т. – Владимир : Изд-во ВлГУ, 2015. – 120 с. – ISBN 978-5-9984-0610-2.

ОГЛАВЛЕНИЕ

Учебное издание

ЖИГАЛОВ Илья Евгеньевич
НОВИКОВ Иван Андреевич

ПРОГРАММИРОВАНИЕ ТРЕХМЕРНОЙ
КОМПЬЮТЕРНОЙ ГРАФИКИ

Учебное пособие

Редактор Р. С. Кузина
Технический редактор С. Ш. Абдуллаева
Корректор Е.С. Глазкова
Компьютерная верстка Е. А. Кузьминой

Подписано в печать 19.10.16.
Формат 60x84/16. Усл. печ. л. 5,35. Тираж 80 экз.
Заказ
Издательство
Владимирского государственного университета
имени Александра Григорьевича и Николая Григорьевича Столетовых.
600000, Владимир, ул. Горького, 87.