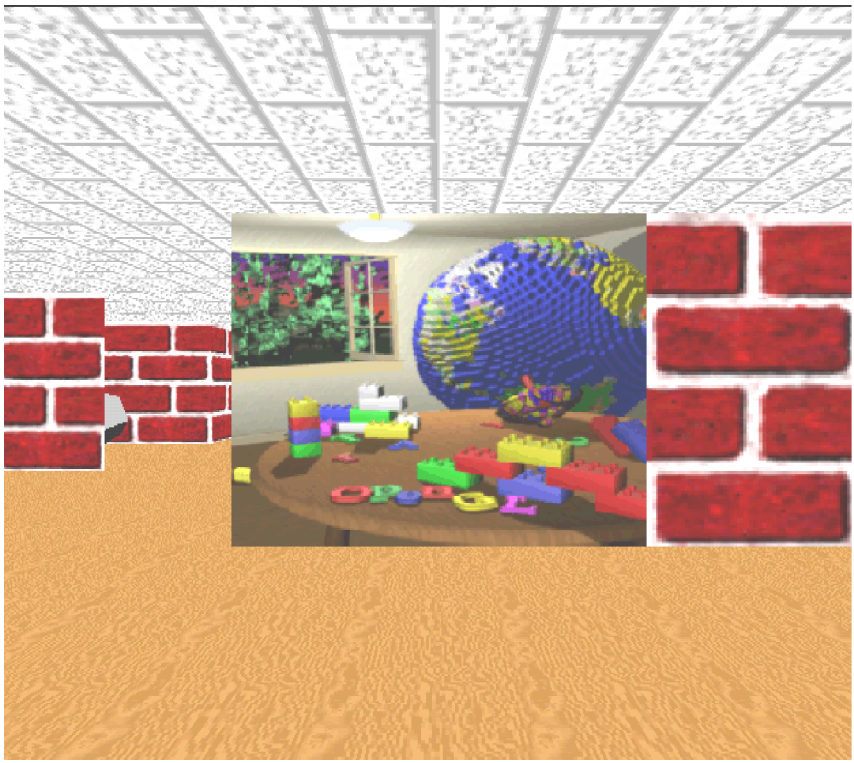
А.П. Полищук

Графические объекты в OpenGL



А.П.Полищук Графические объекты в OpenGL

А.П. Полищук

Графические объекты в OpenGL

Краткий практикум по программированию

Кривой Рог
Издательский отдел КГПУ
2000

Полищук А.П.

Графические объекты в OpenGL: Учебное пособие. – Кривой Рог: Издательский отдел КГПУ, 2000. – 105 с.

Учебное пособие, ориентированное на программирующего пользователя, посвящено методам визуализации графических объектов. Приведен лабораторный практикум по моделированию динамических трехмерных сцен средствами OpenGL с исходными текстами программ в Visual C++.

Для студентов высших учебных заведений, аспирантов, научных и инженерно-технических работников.

Рецензенты:

д-р техн. наук, проф. **А.Н. Марюта**

д-р физ.-мат. наук, проф. **В.Н. Соловьёв**

Оглавление

Введение

«Лучше один раз увидеть…» – эта непреложная истина обусловила привлекательность компьютерной графики во всех научных и инженерных приложениях компьютерных технологий. Но графические объекты на компьютерном экране, столь привлекательные для конечного пользователя, оказываются чрезвычайно трудоемкими для программиста (и ресурсоемкими для аппаратуры). Любая программа, использующая графику, помимо чисто прикладной части всегда содержит исполнительную часть, предназначенную для управления видеоаппаратурой компьютера, создания элементарных «строительных блоков» графических объектов и реализации различных манипуляций над ними – назовем ее условно администратором экрана. Созданию таких программных продуктов для повышения производительности прикладных программистов ведущие фирмы по производству инструментального ПО уделяют много внимания, современные инструменты графического программирования позволяют автоматизировать множество операций, но вряд ли в обозримом будущем отпадет необходимость в непосредственном программировании графических приложений. Поэтому для программирующего пользователя всегда актуально освоение удобных стандартных графических библиотек с достаточно широкими функциональными возможностями; к таким продуктам относится разработанный рядом фирм (включая Microsoft) в 1992 году графический стандарт OpenGL. Разработанная Microsoft графическая система, совместимая с Microsoft Windows, включает 3 объектных библиотеки – opengl32.lib, glu32.lib, glaux.lib, которые вам необходимо включить в состав C/C++ проекта при программировании в MS Visual C++. Основные возможности OpenGL:

- рисование графических примитивов – точек, отрезков, многоугольников – с настройкой их параметров (размеров, цветов и пр.);
- видовые и модельные преобразования с помощью стандартных матричных операций на плоскости и в пространстве для обычных и однородных координатных систем;
- удаление невидимых линий и поверхностей;

- использование сплайнов Безье и рациональных B–сплайнов для рисования кривых и поверхностей;
- наложение текстуры;
- возможность использования освещения;
- плавное сопряжение цветов, устранение ступенчатости изображений, возможность использования «тумана» и других эффектов повышения реалистичности и качества изображений;
- возможность работы с растровыми примитивами.

1. Инициализация связи с OpenGL в консольном приложении Windows

Для общения с любой библиотечной системой необходимо установить с ней связь, выполнив некоторый набор инициализирующих операций. Все возможности OpenGL доступны и в простейшем варианте программирования консольного приложения Windows и этот вариант мы рассмотрим в качестве базисного, чтобы сосредоточиться на приемах работы с библиотекой, не отвлекаясь на особенности программирования в графическом интерфейсе Windows.

Для созданиия консольного приложения сразу после выбора в меню File – New – Projects в среде разработки Developer Studio отметьте позицию Win32 Console Application и задайте имя проекта в соответствующей строке диалогового окна. Затем через меню Proect – Settings – Link в строковом окне Proect Options введите имена объектных библиотек opengl32.lib glu32.lib glaux.lib для включение их в проект при компоновке программы.

Первым очевидным действием при составлении текста программы является включение в C или C++ файл директивой препроцессора интерфейсных заголовочных файлов Windows API и связи с библиотекой OpenGl. Их всего 3:

```
#include <windows.h>
#include <gl\gl.h>
#include <gl\glu.h>
#include <gl\glaux.h>
```

С точки зрения программиста OpenGL – это множество команд для определения графических объектов и операций над ними. Консольное приложение Windows реализуется в окне и для работы с окном можно определить его размеры и положение на экране (если этого не сделать, OpenGL сама выберет параметры окна). Например, если мы создаем окно шириной w, высотой h, с верхним левым углом в точке (x,y), то можем записать определение в глобальной области

```
GLint w=400, h=300, x=200,y=100;
```

Любая C/C++ программа начинает выполняться (и составляться) с главной функции, в которой вы при необходимости обработаете командную строку (например если в каком-то пользовательском файле может храниться битовое изображение образа тексту-

ры или другие данные, которые на стадии составления программы предусмотреть невозможно), поработаете с кучей и выполните прочие необходимые для функционирования программы действия. Если опустить эти действия, то главная функция может выглядеть совсем просто – содержать вызов функции инициализации OpenGL, прототип которой желательно объявить либо непосредственно в рабочем файле, либо вынести в свой заголовочный файл. Для простоты объявим его сразу и напишем простейшую main():

```
//Прототипы функций
GLvoid Init(GLvoid);//Эту функцию мы определим ниже
//Остальные пока сделаем "пустышками"
static GLvoid CALLBACK Draw(GLvoid){}
static GLvoid CALLBACK Reshape(GLint width,GLint height)
{}
GLvoid CALLBACK funcKey(){}
GLvoid CALLBACK funcMouse(AUX_EVENTREC* a){}
void _CRTAPI1 main(void){ Init();}
```

Вас не должны смущать переименованные из int, void и т.д. типы GLint, GLvoid – это сделано для совместимости с версиями библиотеки для других аппаратных платформ.

А теперь обсудим содержание функции инициализации – она не может быть стандартизована, так как определяется конкретной прикладной задачей и мы рассмотрим только несколько наиболее распространенных вариантов. Разумеется, вам неизбежно надо подробно ознакомиться с содержанием включаемых заголовочных файлов и поработать с системой помощи по разделу OpenGL, чтобы лучше ориентироваться в функциональных возможностях библиотеки.

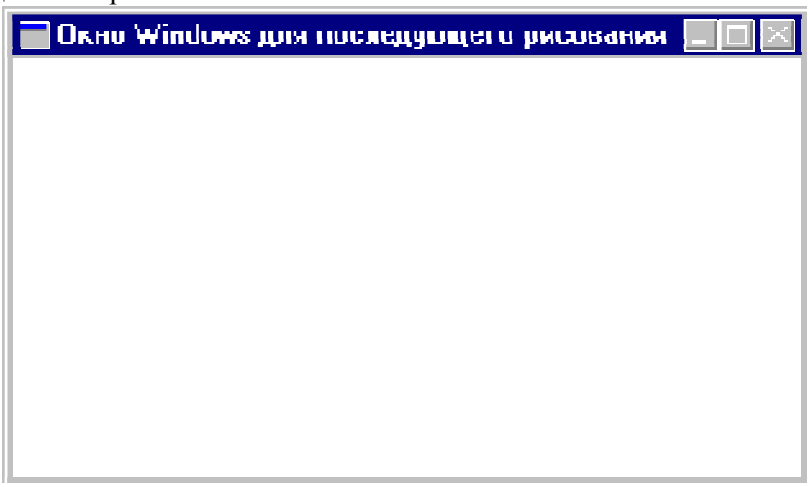
```
// GLOS.H
//
// This is an OS specific header file
#include <windows.h>
// disable data conversion warnings
#pragma warning(disable : 4305)
#pragma warning(disable : 4244) // MIPS
#pragma warning(disable : 4136) // X86
#pragma warning(disable : 4051) // ALPHA

GLvoid Init(GLvoid)
{
  /*Если хочется вывести окно заданных размеров в заданную
позицию*/
  auxInitPosition(x,y,w,h);
```

```
  /*Установка режима работы видеосистемы – задание цветов
в RGB – формате с фоновым буфером*/
  auxInitDisplayMode(AUX_RGB | AUX_DOUBLE);
  /*Зарегистрировать окно с заданным заголовком в операци-
онной системе и вывести его на экран – наличие этого вызо-
ва обязательно*/
  auxInitWindow("Окно Windows для последующего рисова-
ния");
  /*Определим фоновую функцию, которая будет выполняться,
когда очередь сообщений Windows пуста – это может быть и
наша функция рисования*/
  auxIdleFunc(Draw);
  /*Определим функцию, которая будет вызываться для пере-
рисовки окна при каждом изменении его размеров */
  auxReshapeFunc(Reshape);
  /*Укажем также функцию, выполняющую роль оконной проце-
дуры, то есть вызываемую каждый раз при получении окном
какого-либо сообщения и обрабатывающую все сообщения для
нашего окна – мы пока для простоты укажем ту же функцию
рисования изображения*/
  auxMainLoop(Draw);
  /*Если в программе будут обрабатываться нажатия клавиш
клавиатуры, для каждой из них надо указать функцию обра-
ботки отдельно, хотя сама функция обработки для всех кла-
виш может быть и одной и той же */
  auxKeyFunc(AUX_UP, funcKey);
  auxKeyFunc(AUX_DOWN, funcKey);
  auxKeyFunc(AUX_LEFT, funcKey);
  auxKeyFunc(AUX_RIGHT, funcKey);
  /*Если в программе будут обрабатываться нажатия клавиш
мыши, для каждой из них надо указать функцию обработки от-
дельно, хотя сама функция обработки для всех клавиш может
быть и одной и той же; первые 2 аргумента – клавиша и дей-
ствие над ней */
  auxMouseFunc(AUX_LEFTBUTTON, AUX_MOUSEDOWN, mouseFunc);
  /*При начальной инициализации можно, к примеру, устано-
вить цвет фона окна*/
  glClearColor( 0.0, 0.0, 0.0, 1.0 );//черный
}
```

Трудно перечислить все возможные виды инициализирующих действий – это может быть установка параметров осветителей и оптических свойств материала графического объекта, его допустимые размеры и начальные координаты, здесь можно заполнить массивы, описывающие функции одной или двух переменных, установить параметры проектирования изображения на плоскость отображения (картинную плоскость) и многое другое. В тех при-

мерах, которые будут приведены в настоящем пособии, мы будем варьировать содержание начальной инициализации и вы познакомитесь с этим разделом в контексте соответствующих прикладных задач.

Если вы откомпилируете и откомпонуете эту программу, то на экране будет создано стандартное окно Windows с заголовком и стандартными возможностями изменения его размеров, перемещения и пр.



2. Точка в окне Windows, созданная средствами OpenGL

Собственно функция рисования Draw не может рассматриваться в общем виде, ее мы отложили до конкретных примеров и первым таким примером будет создание базисного элементарного графического объекта, который мы назовем «точкой», понимая при этом, что графическая точка в отличие от математической – видимый объект с конечными размерами и цветом, над которым можно при необходимости выполнять различные операции (перемещения, вращения и пр.). Форма этого объекта в нашем сознании ассоциируется с «кружочком» малого размера на плоскости и , наверное, с «шариком» в пространстве. Но кружок, «сложенный» из прямоугольных экранных пикселов неизбежно будет иметь ступенчатую, «с зазубринами» форму и для придания ему естественного внешнего вида придется использовать поддерживаемые библиотекой приемы.

Для унификации методов перемещения точек в координатном пространстве в аналитической и проективной геометрии рассматривают ассоциированный с точкой вектор с началом в начале координат и концом в рассматриваемой точке – его составляющие по осям равны координатам точки. Так как операции над векторами сводятся к умножению его слева на матрицу преобразования, то их легко унифицировать – сделать однотипными и такой способ работы с точками используется в OpenGL. Чтобы операции преобразования включали в себя не только масштабирование, вращения и отражения, но и переносы, точку на плоскости формально представляют вместо двух координат – тремя (X, Y, V), а в пространстве вместо тройки – четверкой чисел (X, Y, Z, W). Это так называема **однороднаяя система координат**, причем декартовы координаты легко вычисляются по однородным:

- на плоскости: $x=X/V$, $y=Y/V$;
- в пространстве: $x=X/W$, $y=Y/W$, $z=Z/W$.

Преобразования однородных координат описываются соотношениями:

$$||X\ Y\ Z\ W||=\mathrm{M}\begin{Vmatrix} x \\ y \\ z \\ 1 \end{Vmatrix} \text{ и } ||x^1\ y^1\ z^1\ 1||=\left\|\frac{X}{W}\quad\frac{Y}{W}\quad\frac{Z}{W}\quad 1\right\|,$$

где M – матрица преобразования, общий вид которой:

$$\mathrm{M}=\begin{Vmatrix} a & d & h & l \\ b & e & i & m \\ c & f & j & n \\ p & q & r & s \end{Vmatrix}$$

При этом подматрица 3x3 $\begin{Vmatrix} a & d & h \\ b & e & i \\ c & f & j \end{Vmatrix}$ осуществляет масштабирование, вращения и отражения, вектор $\begin{Vmatrix} l \\ m \\ n \end{Vmatrix}$ производит перенос, а вектор $||p\ q\ r||$ – перспективное преобразование; последний скалярный элемент s – общее изменение масштаба. Обратите внимание на то, что матрица преобразования отличается от обычно используемой в литературе – она хранится не построчно, а «столбец за столбцом». Полное преобразование с помощью приведенной матрицы обеспечивает выполнение комплекса операций раздельного масштабирования по осям, вращения, отражения относительно осей, переноса и изменения масштаба в целом плюс преобразование, связанное с перспективными проекциями и называется **билинейным преобразованием**.

До попытки изобразить на экране наш графический объект вспомним о том, что реально он будет отображен на экранной плоскости или **картинной плоскости**, связанной с некоторым наблюдателем, который в свою очередь расположен в некоторой воображаемой **«мировой»** системе координат. Первоначально мы определяем положение графического объекта в мировой системе (она определена как правосторонняя – с осью *z*, направленной от экрана на вас), затем, зная позицию наблюдателя в мировой системе и направление его взгляда (по умолчанию – в начало мировой

системы), придется преобразовать координаты объекта в систему наблюдателя – ее называют **видовой системой координат** – она определена как левосторонняя, т.е. с осью *z*, направленной в глубину экрана. И, наконец, придется спроектировать 3-мерное изображение на картинную плоскость, т.е. пересчитать видовые координаты в экранные. Кроме того, перед выводом изображения необходимо задать **объем видимости** в мировом пространстве – графические объекты будут отсекаться по границам объема видимости и затем проецироваться на картинную плоскость.

Перейдем теперь к способам реализации приведенных методов работы в OpenGL. В ней предусмотрены 3 типа матриц 4x4 – видовая, проекций и текстуры (о ней попозже). Для работы с любой матрицей ее делают текущей с помощью команды **void glMatrixMode(Glenum mode)**, аргумент которой можно задать одной из символических констант, определяющих тип матрицы – GL_MODELVIEW, GL_PROECTION, GL_TEXTURE. После установки типа текущей матрицы задают ее элементы ссылкой на другую матрицу с их значениями командой **void glLoadMatrix[f d](Gltype* m)**. m – указатель на матрицу, хранящуюся в порядке расположения столбцов с элементами float или double. Заменить текущую матрицу единичной можно командой

glLoadIdentity().

Умножить заданную матрицу слева на текущую можно командой

glMultMatrix[f d](Gltype* m).

Для формирования матриц видового преобразования удобно использовать команды:

glRotate[f d] (Gltype angle, Gltype x, Gltype y, Gltype z) для вращения вектора против часовой стрелки на угол angle вокруг вектора с координатами x, y, z;

glTranslate[f d] (**Gltype x, Gltype y, Gltype z**) для переноса в заданную точку;

glScale[f d](Gltype x, Gltype y, Gltype z) для масштабирования по осям.

Определение области вывода осуществляется командой

GlViewPort(Glint x, Glint y, Glint w, Glint h), где x, y – координаты левого нижнего угла прямоугольника вывода в окне, w, h – ширина и высота области вывода.

Установить объем видимости и создать параллельную проекцию (она создает параллелепипед видимости) позволяет команда

GlOrtho(Gldouble left, Gldouble right, Gldouble bottom, Gldouble top, Gldouble near, Gldouble far) – 4 первых аргумента определяют координаты левой, правой, нижней и верхней плоскостей отсечения, 2 последних – расстояние до ближней и дальней плоскостей отсечения. Можно использовать также команду

GlOrtho2d(Gldouble left, Gldouble right, Gldouble bottom, Gldouble top), у которой 2 последних аргумента опущены и по умолчанию равны –1 и 1.

Установить объем видимости и создать перспективную проекцию (она создает усеченную пирамиду видимости) позволяет команда

GlFrustum(Gldouble left, Gldouble right, Gldouble bottom, Gldouble top, Gldouble near, Gldouble far) с теми же аргументами, что и Glortho.

Перспективная проекция с заданным конусом видимости реализуется командой

gluPerspective(Gldouble angley, Gldouble aspect, Gldouble znear, Gldouble zfar), первый аргумент которой задает угол видимости в градусах вдоль оси y, а угол видимости вдоль оси x задается отношением сторон aspect.

Чтобы однозначно определить положение и ориентацию трехмерного объекта в пространстве, необходимо задать 3 вектора: вектор, определяющий положение некоторой контрольной точки объекта, которую, например, хотелось бы разместить в центре экрана, вектор направления взгляда наблюдателя и «вектор вверх», указывающий направление, которое для объекта будет считаться верхним. Эти векторы можно задать в виде аргументов команды

gluLookAt(Gldouble eyex, Gldouble eyey, Gldouble eyez, Gldouble centerx, Gldouble centery, Gldouble centerz, Gldouble upx, Gldouble upy, Gldouble upz), первые 3 аргумента при этом – координаты точки наблюдения, вторая тройка – координаты центра сцены и третья – направление оси y сцены. Матрица отображает контрольную точку на отрицательную полуось z, а точку наблюдения – в начало системы видовых координат, чтобы центр сцены отображался в центре области вывода.

Осталось узнать:

Как в OpenGL задать координаты точки (вершины) – для этого существует команда

glVertex[2 3 4][s i f d](type coord) – здесь цифры в имени определяют количество аргументов (недостающие до 4-х – по умолчанию последний 1, третий – 0), а буквы суффикса – тип чисел.

Как задать размер точки – для этого есть команда установки текущего цвета:

GlPointSize(Glfloat size) с диаметром точки в пикселах в качестве аргумента; если включен режим устранения ступенчатости, то поддерживаются не все размеры и максимальный можно узнать с помощью команды **glGet(GL_POINT_SIZE)**. Включение и выключение режима устранения ступенчатости осуществляют универсальной командой **glEnable(GL_POINT_SMOOTH**), а выключение – командой **glDisable(GL_POINT_SMOOTH)**.

Как задать цвет точки – командой **glColor[3 4][b s i f d](Gltype components)** с заданием аргументов (компонентов цвета красного, зеленого, синего и прозрачности) в диапазоне 0…1.

Цвет оконного фона формируется командой

glClearColor(Glclampf red, Glclampf green, Glclampf blue, Glclampf alpha)

и устанавливается командой

GlClear(Glbitfield m) с аргументом GL_COLOR_BUFFER_BIT, который может комбинироваться операцией ИЛИ с аргументами GL_DEPTH_BUFFER_BIT (после команды glClearDepth), GL_ACCUM_BUFFER_BIT (после команды glClearAccum), GL_STENCIL_BUFFER_BIT (после команды glClearStencil).

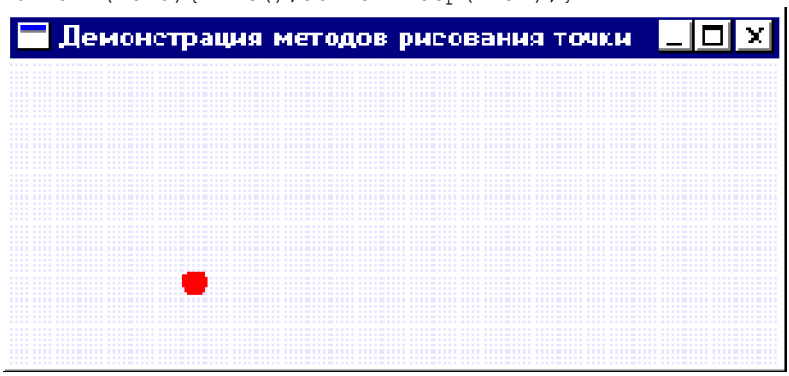
Вооружившись таким объемом знаний, можем попытаться нарисовать точку и что-нибудь с ней сделать – для этого придется наполнить фигурные скобки функции Draw реальным содержанием по созданию и отображению 3-мерной точки и составить некоторые вспомогательные функции.

```
/*Эта программа рисует «точку», которая перемещается в
зоне обзора по сложной траектории*/
#include<windows.h>
#include<gl\gl.h>
#include<gl\glu.h>
#include<gl\glaux.h>
```

```
GLint w=1024,h=768;//Размеры окна
//Полярные координаты зоны видимости
GLfloat alpha, //Азимут - широта
        betha, //Угол места - долгота
        radius; //Радиус
/*Прототипы функций инициализации, рисования и обновления
окна*/
GLvoid Init(GLvoid);
GLvoid CALLBACK Draw(GLvoid);
GLvoid CALLBACK Reshape(GLsizei, GLsizei);
//Повороты и перенос
void polarView( GLdouble, GLdouble, GLdouble, GLdouble);
//Отказываемся от раздражающих предупреждения компилятора
#pragma warning(disable : 4305)
#pragma warning(disable : 4244)

//Начальная инициализация
GLvoid Init(void)
{
  GLfloat maxObjectSize, aspect; /*Размер объекта и угол
обзора*/
  GLdouble near_plane, far_plane;//Отсечение по глубине
  auxInitPosition(0,0,w/4,h/4); //Позиция и размер окна
  auxInitDisplayMode( AUX_RGB | AUX_DEPTH | AUX_DOUBLE );
//Видеорежим
  auxInitWindow("Демонстрация методов рисования точки");
//Вывести окно
  auxIdleFunc(Draw);/*Функция вызывается при отсутствии
сообщений в очереди ОС*/
  auxReshapeFunc(Reshape);/*Функция вызывается при
изменениях размеров окна*/
  glClearDepth(1.0); //Очистка буфера глубины
  glEnable(GL_DEPTH_TEST);/*Разрешаем тестирование
глубины*/
  glMatrixMode(GL_PROJECTION);/*Текущей будет матрица
проектирования*/
  aspect=(GLfloat)w/h;
  gluPerspective(65.0, aspect, 3.0, 7.0);/*Формирование
матрицы проектирования*/
  glMatrixMode(GL_MODELVIEW); //Текущая – видовая матрица
  near_plane=3.0;
  far_plane=7.0;
  maxObjectSize=1.0;
  radius=near_plane+maxObjectSize/2.0;
  alpha=0.0;
  betha=0.0;
  //Серый цвет фона
  glClearColor(0.75f,0.75f,0.75f,1.0f);
```

```
}

/*При перерисовке повторяем фрагмент инициализации для
новых размеров окна*/
static void CALLBACK Reshape(int width, int height)
{
  GLfloat aspect;
  if (!height) return;
  glViewport(0, 0, width/2, height/2);
  aspect=(GLfloat)width/height;
  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();
  gluPerspective(65.0,aspect, 3.0, 7.0);
  glMatrixMode(GL_MODELVIEW);
}

//Повороты и перенос
void polarView(GLdouble radius, GLdouble twist, GLdouble
latitude, GLdouble longitude)
{
  glTranslated(0.0, 0.0,-radius);
  glRotated(-twist, 1.0, 0.0, 0.0);
  glRotated(-latitude, 0.0, 1.0, 0.0);
  glRotated(longitude, 0.0, 0.0, 1.0);
}

//Это сама функция рисования
static void CALLBACK Draw(void)
{
  static GLfloat angle;
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
  glPushMatrix();
  alpha+=2.0; betha+=2.0; angle+=1.0;
  polarView(radius,1,alpha,betha);
  glPushMatrix();
  //Матрицы поворота и переноса
  glRotatef(angle, 1.0, 0.0, 0.0);
  glTranslatef(0.3, 0.3, 0.1);
  glPointSize(10.0);//Размер точки
  glColor4f(1.0,0.0,0.0,1.0);//Цвет точки
  glEnable(GL_POINT_SMOOTH);/*Разрешим устранение
ступенчатости*/
  glBegin(GL_POINTS);
  glVertex4f(0.0,0.0,0.0,1.0);//Рисуем точку
  glEnd();
  glDisable(GL_POINT_SMOOTH);
  auxWireSphere(1.);//Сетчатая сфера просто для украшения
  glPopMatrix();
```

```
  glPopMatrix();
  glFinish();
  SwapBuffers(wglGetCurrentDC());
}

//Простейшая главная функция
void main(void){Init();auxMainLoop(Draw);}
```



3. Отрезки прямых в OpenGL

Рассмотренная в предыдущем примере точка является строительным элементом для рисования отрезков прямых. Для рисования отрезков прямых в OpenGL необходимо использовать скобочную команду **glBegin(GL_LINES)** и после нее до закрывающей команды **glEnd()** пары подлежащих соединению вершин с помощью команд glVertex; если число точек нечетное, то последняя точка будет потеряна. Если это нежелательно, используйте команду **glBegin(GL_LINE_STRIP)**, а если вы хотите соединить последнюю точку с первой в списке, то используйте **glBegin(GL_LINE_LOOP)**. Есть также возможность изменять параметры линий:

glLineWidth(Glfloat width) – для определения ширины линий;

glEnable(GL_LINE_SMOOTH), **glDisable(GL_LINE_SMOOTH)** – для режима устранения ступенчатости;

glLineStipple(Glint factor, Glushort pattern) – для определения шаблона штриховки линии; первый аргумент – счетчик повторов битов, второй – 16-разрядный шаблон, биты которого определяют рисуемые фрагменты, например glLineStipple(2,0x1C47).

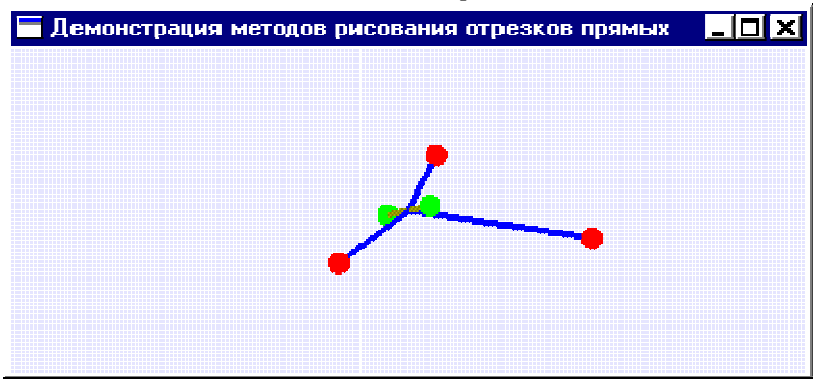
Вы можете поэкспериментировать с приведенной ниже программой, изменяя ширину и штриховку рисуемых в ней линий и различные способы соединения точек.

```
/*Эта программа рисует отрезок прямой, который
перемещается в зоне обзора по сложной траектории и
координатные полуоси – они тоже отрезки прямых линий*/
#include<windows.h>
#include<gl\gl.h>
#include<gl\glu.h>
#include<gl\glaux.h>

GLint w=1024,h=768;
GLfloat alpha, //Азимут - широта
        betha, //Угол места - долгота
        radius; //Радиус
/*Прототипы функций инициализации, рисования и обновления
окна*/
GLvoid Init(GLvoid);
GLvoid CALLBACK Draw(GLvoid);
GLvoid CALLBACK Reshape(GLsizei, GLsizei);
#pragma warning(disable : 4305)
```

```
#pragma warning(disable : 4244)

//Начальная инициализация
GLvoid Init(void)
{
  GLfloat maxObjectSize, aspect; /*Размер объекта и угол
обзора*/
  GLdouble near_plane, far_plane; //Отсечение по глубине
  auxInitPosition(0,0,w/2,h/2); //Позиция и размер окна
  auxInitDisplayMode(AUX_RGB | AUX_DEPTH | AUX_DOUBLE);
//Видеорежим
  //Вывести окно с заголовком
  auxInitWindow("Демонстрация методов отрезков прямых");
  /*Функция, которая будет вызываться при отсутствии
сообщений в очереди ОС*/
  auxIdleFunc(Draw);
  /*Функция, которая вызывается при изменениях размеров
окна*/
  auxReshapeFunc(Reshape);
  glClearDepth(1.0); //Очистка буфера глубины
  glEnable(GL_DEPTH_TEST);
  glMatrixMode(GL_PROJECTION);
  aspect=(GLfloat)w/h;
  //Формирование матрицы перспективного проектирования
  gluPerspective(65.0, aspect, 3.0, 7.0 );
  glMatrixMode(GL_MODELVIEW);
  near_plane=3.0;
  far_plane=7.0;
  maxObjectSize=1.0;
  radius=near_plane+maxObjectSize/2.0;
  alpha=0.0;
  betha=0.0;
  glClearColor(0.75f,0.75f,0.75f,1.0f); //Серый цвет фона
}

/*При перерисовке повторяем фрагмент инициализации для
новых размеров окна*/
static void CALLBACK Reshape(int width, int height){
  GLfloat aspect;
  glViewport(0, 0, width, height);
  aspect=(GLfloat)width/height;
  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();
  //gluPerspective(-65.0,aspect,60.0, 60.0 );
  glOrtho(-60.0,60.0,-60.0,60.0,-60.0,60.0);
  //glFrustum(-60.0,60.0,-60.0,60.0,-60.,60.);
  /*
    gluLookAt(0.0,0.0,0.0,//Позиция наблюдателя
```

```
              3.0,2.0,1.0, //Куда смотрит наблюдатель
              0.0,1.0,0.0); //Вектор вверх
  */
  glMatrixMode(GL_MODELVIEW);
}

//Подпрограмма рисования
static void CALLBACK Draw(void)
{
  static GLfloat angle;
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
  glPushMatrix();
  alpha+=1.0;
  betha+=1.0;
  //polarView(radius,1,alpha,betha);
  glPushMatrix();
  angle+=2.0;
  //Матрицы поворота и переноса
  //glTranslatef( 30.5, 30.5, 31.0);
  glRotatef(angle,1.0, 0.0, 0.0);
  glRotatef(angle,0.0,1.0, 0.0);
  glRotatef(angle,0.0,.0, 1.0);
  glPointSize(10.0);//Размер точки
  glLineWidth(3.0); //Ширина линий
  glEnable(GL_POINT_SMOOTH);/*Разрешим устранение
ступенчатости*/
  //Рисование точек
  glBegin(GL_POINTS);
  glColor4f(0.0,1.0,0.0,1.0); //Цвет точки зеленый
  glVertex4f(15.0,10.0,5.0,1.0);//Рисуем вторую точку
  glVertex4f(-15.0,-10.,-5.,1.0);
  //Красные точки на концах осей
  glColor4f(1.0,0.0,0.0,0.0);
  glVertex4f(30.0,0.0,.0,1.0);
  glColor4f(1.0,.0,0.0,1.0);
  glVertex4f(0.0,30.0,0.0,1.0);
  glColor4f(1.0,.0,0.0,.0);
  glVertex4f(.0,0.0,30.0,1.0);
  glEnd();
  //Рисование линий
  glBegin(GL_LINES);
  glColor4f(0.0,0.0,1.0,1.0);//Цвет линии синий
  //Нарисуем оси координат
  glVertex4f(0.0,0.0,0.0,1.0);
  glVertex4f(30.0,0.0,0.0,1.0);
  glVertex4f(0.0,0.0,0.0,1.0);
  glVertex4f(0.0,30.0,0.0,1.0);
  glVertex4f(0.0,0.0,0.0,1.0);
```

```
  glVertex4f(0.0,0.0,30.0,1.0);
  glColor4f(0.5,0.5,0.0,1.0);
  //Те же точки соединим линией
  glVertex4f(-15.0,-10.0,-5.0,1.0);
  glVertex4f(15.0,10.0,5.0,1.0);
  glEnd();
  glDisable(GL_POINT_SMOOTH );
  glColor4f(0.0,0.5,0.5,1.0);
  //auxWireSphere(20.0); /*Сетчатая сфера просто для
украшения*/
  glPopMatrix();
  glPopMatrix();
  glFinish();
  SwapBuffers(wglGetCurrentDC());
}

void main(void){Init();auxMainLoop(Draw);}
```



4. Многоугольники в OpenGL

В OpenGL предусмотрены раздельные возможности рисования треугольников, прямоугольников и многоугольников с призвольным числом вершин.

При рисовании треугольников можно задавать различные режимы:

- режим независимого рисования треугольников, при котором просто берутся тройки вершин, задается просто командой **glBegin(GL_TRIANGLES)** и тройками команд определения вершин glVertex*;
- режим рисования треугольников, связанных общими сторонами задается **glBegin(GL_TRIANGLE_STRIP)**;
- режим рисования треугольников, связанных общей вершиной, задается **glBegin(GL_TRIANGLE_FAN)**.

Режимы связывания треугольников используются при построении из них сложных фигур; простейший пример – построение конуса из связанных вершинами треугольников.

Для четырехугольников предусмотрено 2 режима – несвязанного рисования командой **glBegin(GL_QUADS)** и связанного сторонами рисования командой **glBegin(GL_QUAD_STRIP)**. При использовании связанных режимов обращайте внимание на порядок обхода вершин, иначе можно получить хаотичное нагромождение линий вместо осмысленного рисунка.

Любые многоугольники могут быть нарисованы либо точками – команда

glPolygonMode(GL_FRONT, GL_POINT), либо контурами командой

glPolygonMode(GL_FRONT, GL_LINE), либо с «заливкой» их лицевых и обратных сторон каким-либо трафаретом командой

glPolygonMode(GL_FRONT_AND_BACK, GL_FILL). Трафарет можно задать, например, так:

```
GLubyte pattern[] = {
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x03, 0x80, 0x01, 0xC0, 0x06, 0xC0, 0x03, 0x60,
0x04, 0x60, 0x06, 0x20, 0x04, 0x30, 0x0C, 0x20,
0x04, 0x18, 0x18, 0x20, 0x04, 0x0C, 0x30, 0x20,
0x04, 0x06, 0x60, 0x20, 0x44, 0x03, 0xC0, 0x22,
0x44, 0x01, 0x80, 0x22, 0x44, 0x01, 0x80, 0x22,
0x44, 0x01, 0x80, 0x22, 0x44, 0x01, 0x80, 0x22,
```

```
0x44, 0x01, 0x80, 0x22, 0x44, 0x01, 0x80, 0x22,
0x66, 0x01, 0x80, 0x66, 0x33, 0x01, 0x80, 0xCC,
0x19, 0x81, 0x81, 0x98, 0x0C, 0xC1, 0x83, 0x30,
0x07, 0xe1, 0x87, 0xe0, 0x03, 0x3f, 0xfc, 0xc0,
0x03, 0x31, 0x8c, 0xc0, 0x03, 0x33, 0xcc, 0xc0,
0x06, 0x64, 0x26, 0x60, 0x0c, 0xcc, 0x33, 0x30,
0x18, 0xcc, 0x33, 0x18, 0x10, 0xc4, 0x23, 0x08,
0x10, 0x63, 0xC6, 0x08, 0x10, 0x30, 0x0c, 0x08,
0x10, 0x18, 0x18, 0x08, 0x10, 0x00, 0x00, 0x08
};
```

и далее установить трафарет **glPolygonStipple(pattern)**, разрешив его наложение **glEnable(GL_POLYGON_STIPPLE)**.

Если используется команда рисования многоугольника с произвольным числом вершин **glBegin(GL_POLYGON)**, то можно осуществить только несвязанное рисование.

Для удобства программирования OpenGL предоставляет возможность расположить данные о вершинах (а также о цветах, координатах нормалей, и других ассоциированных с вершинами данных) многоугольника в массивах и затем использовать блоки из них для рисования одной единственной командой без командных скобок; для массивов вершин используют команду

glVertexPointer(Glint size, Glenum type, Glsizei stride, void* ptr), где первый аргумент задает количество координат у вершин (2, 3 или 4), второй определяет тип данных и может быть GL_SHORT, GL_INT, GL_FLOAT, GL_DOUBLE, третий задает смещение в байтах между данными соседних вершин и используется при хранении атрибутов вершин вместе с их координатами в одном и том же массиве, четвертый аргумент – адрес самого массива данных.

Как и для других режимов работы, OpenGL надо сообщить о намерении работать с массивом – для этого предусмотрена команда **glEnableClientState(Glenum array)**, аргумент которой может принимать значения GL_VERTEX_ARRAY, GL_INDEX_ARRAY, GL_NORMAL_ARRAY, GL_TEXTURE_COORD_ARRAY, GL_COLOR_ARRAY, GL_EDGE_FLAG_ARRAY. По окончании работы с массивом ее блокируют командой **glDisableClientState(Glenum array)**. После формирования массива и разрешения работы с ним воспроизведение хранимых в нем данных осуществляет команда **glDrawArrays(Glenum mode, Glint first, Glsizei count)**, которая воспроизводит примитив типа в пер-

вом аргументе, начиная с элемента, заданного вторым аргументом, отображая столько элементов, сколько задано в третьем аргументе. Например:

```
GlDrawArrays(GL_POLYGON, 0, 64);
```

Собственно описанная команда последовательно вызывает команду

glArrayElement(Glint index), аргумент которой задает номер отображаемого элемента.

Помимо рассмотренных, в OpenGL есть команды

glRect[](Gltype x1, Gltype y1, Gltype x2, Gltype y2) и

glRect[d f i s]v(Gltype* v1, Gltype* v2),

позволяющие задать прямоугольник в плоскости z=0 через координаты его двух противоположных углов.

Ниже приведен простейший пример рисования прямоугольных отображений координатных плоскостей – в нем не используются все перечисленные и многие не рассмотренные возможности; вам предлагается просто шаблон для экспериментирования, а для дальнейшего знакомства с возможностями библиотеки glaux и некоторого оживления рисунка в координатные отсеки помещены Платоновы тела.

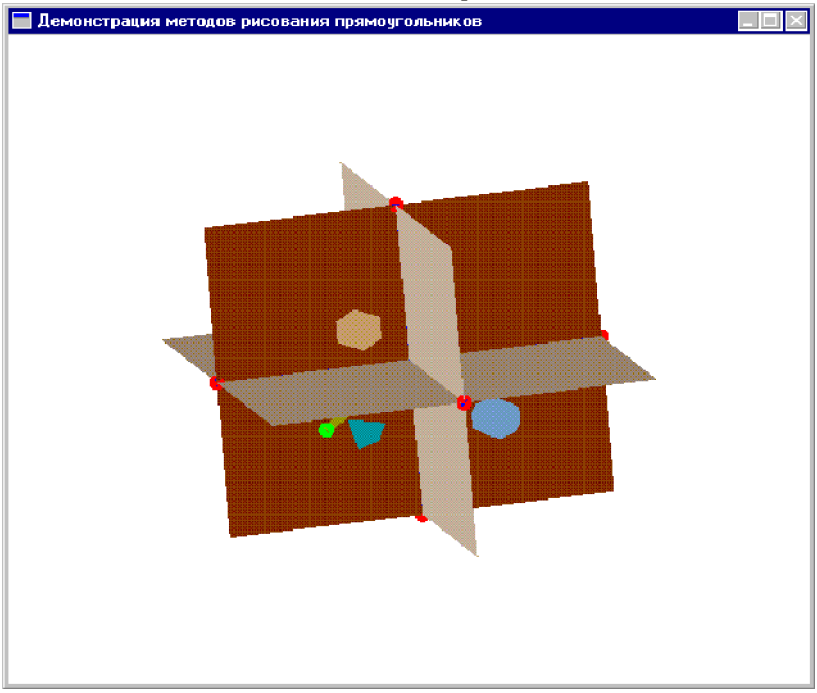
```
/* Эта программа добавляет координатные плоскости,
координатные оси и пару разного цвета точек в координатном
пространстве. После рисования координатных плоскостей
образуются 8 координатных подпространств - заполним их
правильными многогранниками - платоновыми телами (их всего
5) */
#include<windows.h>
#include<gl\gl.h>
#include<gl\glu.h>
#include<gl\glaux.h>

GLint w=800,h=600;
GLfloat alpha, //Азимут - широта
        betha, //Угол места - долгота
        radius; //Радиус
/*Прототипы функций инициализации, рисования и обновления
окна*/
GLvoid Init(GLvoid);
GLvoid CALLBACK Draw(GLvoid);
GLvoid CALLBACK Reshape(GLsizei, GLsizei);
#pragma warning(disable : 4305)
#pragma warning(disable : 4244)
#pragma warning(disable : 4101)
```

```
//Начальная инициализация
GLvoid Init(void)
{
  GLfloat maxObjectSize, aspect; /*Максимальный размер
объекта и угол обзора*/
  //GLdouble near_plane, far_plane;//Отсечение по глубине
  auxInitPosition(0,0,w,h); //Позиция и размер окна
  auxInitDisplayMode(AUX_RGB | AUX_DEPTH | AUX_DOUBLE);
//Видеорежим
  //Вывести окно с заголовком
  auxInitWindow("Демонстрация методов рисования
прямоугольников");
  auxIdleFunc(Draw); /*Функция, которая будет вызываться
при отсутствии сообщений в очереди ОС*/
  /*Функция, которая вызывается при изменениях размеров
окна*/
  auxReshapeFunc(Reshape);
  glEnable(GL_DEPTH_TEST);
  glMatrixMode(GL_PROJECTION);
  aspect=(GLfloat)w/h;
  //Формирование матрицы перспективного проектирования
  //gluPerspective(65.0, aspect, 3.0, 7.0);
  glMatrixMode(GL_MODELVIEW);
  //Черный цвет фона
  glClearColor(0.0f,0.0f,0.0f,1.0f);
}

/*При перерисовке повторяем фрагмент инициализации для
новых размеров окна*/
static void CALLBACK Reshape(int width, int height){
  GLfloat aspect;
  glViewport(0, 0, width, height);
  aspect=(GLfloat)width/height;
  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();
  //gluPerspective(65.0,aspect,60.0, 60.0 );
  glOrtho(-60.0,60.0,-60.0,60.0,-60.0,60.0);
  //glFrustum(-60.0,60.0,-60.0,60.0,-60.,60.);
  /*
    gluLookAt(0.0,0.0,0.0, //Позиция наблюдателя
              3.0,2.0,1.0, //Куда смотрит наблюдатель
              0.0,1.0,0.0); //Вектор вверх
  */
  glMatrixMode(GL_MODELVIEW);
}

//Подпрограмма рисования
static void CALLBACK Draw(void)
```

```
{
  static GLfloat angle;
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
  glPushMatrix();
  glPushMatrix();
  angle+=2.0;
  //Матрицы поворота и переноса
  //glTranslatef( 30.5, 30.5, 31.0);
  glRotatef(angle,1.0, 0.0, 0.0);
  glRotatef(angle,0.0,1.0, 0.0);
  glRotatef(angle/4,0.0,.0, 1.0);
  glPointSize(10.0); //Размер точки
  glLineWidth(5.0); //Ширина линий
  glEnable(GL_POINT_SMOOTH); /*Разрешим устранение
ступенчатости*/
  glEnable(GL_LINE_SMOOTH);
  //Рисование линий
  glBegin(GL_POINTS);
  glColor4f(0.0,1.0,0.0,1.0); //Цвет точки
  glVertex4f(15.0,10.0,5.0,1.0); //Рисуем точку
  glColor4f(0.0,0.0,1.0,1.0); //Цвет точки
  glVertex4f(-15.0,-10.,-5.,1.0);
  //Красные точки на концах осей
  glColor4f(1.0,0.0,0.0,0.0);
  glVertex4f(30.0,0.0,0.0,1.0);
  glVertex4f(-30.0,0.0,0.0,1.0);
  glVertex4f(0.0,30.0,0.0,1.0);
  glVertex4f(0.0,-30.0,0.0,1.0);
  glVertex4f(.0,0.0,30.0,1.0);
  glVertex4f(.0,0.0,-30.0,1.0);
  glEnd();
  //Рисуем линии
  glBegin(GL_LINES);
  glColor4f(0.0,0.0,1.0,1.0);//Цвет линии синий
  //Нарисуем оси координат
  glVertex4f(-30.0,0.0,0.0,1.0);
  glVertex4f(30.0,0.0,0.0,1.0);//Ось x
  glVertex4f(0.0,-30.0,0.0,1.0);
  glVertex4f(0.0,30.0,0.0,1.0);//Ось y
  glVertex4f(0.0,0.0,-30.0,1.0);// Ось z
  glVertex4f(0.0,0.0,30.0,1.0);
  glColor4f(0.5,0.5,0.0,1.0);
  //Точки отрезка соединим линией
  glVertex4f(-15.0,-10.0,-5.0,1.0);
  glVertex4f(15.0,10.0,5.0,1.0);
  glEnd();
  /*Нарисуем 3 координатные плоскости в виде
прямоугольников*/
```

```
//Зададим прямоугльники 4-мя вершинами каждый
glPolygonMode(GL_FRONT_AND_BACK,GL_FILL);
glBegin(GL_QUADS);
//Плоскость XY
glColor4f(0.4,0.1,0.0,1.0);
glVertex4f(30.0,30.0,0.0,1.0);
glVertex4f(-30.0,30.0,0.0,1.0);
glVertex4f(-30.0,-30.0,0.0,1.0);
glVertex4f(30.0,-30.0,0.0,1.0);
//Плоскость XZ
glColor4f(0.5,0.4,0.3,1.0);
glVertex4f(30.0,0.0,30.0,1.0);
glVertex4f(30.0,0.0,-30.0,1.0);
glVertex4f(-30.0,0.0,-30.0,1.0);
glVertex4f(-30.0,0.0,30.0,1.0);
//Плоскость YZ
glColor4f(0.7,0.6,0.5,1.0);
glVertex4f(0.0,30.0,30.0,1.0);
glVertex4f(0.0,-30.0,30.0,1.0);
glVertex4f(0.0,-30.0,-30.0,1.0);
glVertex4f(0.0,30.0,-30.0,1.0);
glEnd();
//Рисуем 3-мерные тела в координатных отсеках
glColor4f(0.0,0.5,0.5,1.0);
glTranslatef(10.0,10.0,10.0);
auxSolidTetrahedron(4.0); //Тетраэдр
glTranslatef(-10.0,-10.0,-10.0);
glColor4f(0.5,0.0,0.5,.0);
glTranslatef(10.0,10.0,-10.0);
auxSolidCube(4.0); //Гексаэдр - он же куб
glTranslatef(-10.0,-10.0,10.0);
glColor4f(0.5,0.5,0.0,.0);
glTranslatef(-10.0,10.0,-10.0);
auxSolidOctahedron(4.0); //Октаэдр
glTranslatef(10.0,-10.0,10.0);
glColor4f(0.3,0.5,0.7,.0);
glTranslatef(-10.0,10.0,10.0);
auxSolidDodecahedron(4.0); //Додекаэдр
glTranslatef(10.0,-10.0,-10.0);
glColor4f(0.7,0.5,0.3,.0);
glTranslatef(10.0,-10.0,10.0);
auxSolidIcosahedron(4.0); //Икосаэдр
glTranslatef(-10.0,10.0,-10.0);
/*
  glColor4f(.0,1.0,.0,.0);
  glTranslatef(10.0,-10.0,-10.0);
  auxSolidCube(4.0); //Цилиндр
  glTranslatef(-10.0,10.0,10.0);
```

```
    glColor4f(1.0,1.0,.0,.0);
    glTranslatef(-10.0,-10.0,-10.0);
    auxSolidCube(4.0); //Конус
    glTranslatef(10.0,10.0,10.0);
    glColor4f(1.0,.0,1.0,.0);
    glTranslatef(-10.0,-10.0,10.0);
    auxSolidCube(4.0); //Торус
    glTranslatef(10.0,10.0,-10.0);
  */
  glDisable(GL_LINE_SMOOTH);
  glDisable(GL_POINT_SMOOTH);
  glPopMatrix();
  glFinish();
  SwapBuffers(wglGetCurrentDC());
}

void main(void){Init();auxMainLoop(Draw);}
```



5. Раскрашивание видимых поверхностей с учетом освещения и оптических свойств материала рисуемого объекта

5.1. Краткие сведения из оптики

Нам хотелось бы, чтобы создаваемые программой на экране виртуальные объекты были похожи на объекты реального мира, зрительное восприятие которых зависит от характера освещения, оптических свойств поверхности объекта и взаимной ориентации источников света, отображаемого объекта и картинной плоскости, воспринимающей световые лучи.

Источники света могут генерировать рассеянный свет (освещенность поверхности не зависит от ее ориентации), свет от точечного источника (освещенность поверхности зависит от ее ориентации), направленный свет параллельных лучей типа прожектора.

Поверхность освещаемого объекта может рассеивать отражаемые лучи по разным направлениям или в одном определенном направлении, часть света пропускать через себя, а другие части отражать и поглощать. Количество поглощенной, отраженной и пропущенной световой энергии зависит от спектра источника излучения и свойств материала объекта освещения. Цвет объекта определяется спектром поглощения. Пропускание и отражение света может быть диффузным (рассеянным) или направленным (например, стекло или замерзшее стекло). При наличии нескольких источников света составляющие от каждого из них накладываются.

Для того чтобы понять, каким образом можно построить искусственное изображение сцены, рассмотрим, каким путем возникает изображение реальной сцены в глазе наблюдателя.

Пусть задана реальная сцена, состоящая из источника света и ряда объектов. Весь свет начинает свой путь из источника и распространяется от него по прямолинейным траекториям до попадания на объекты сцены. Попав на какой-либо объект сцены, луч света может преломиться и уйти внутрь объекта или отразиться (рассеяться). Отразившись от объекта, луч света опять распростра-

няется прямолинейно до попадания на следующий объект, и так далее. Часть лучей в конце концов попадает в глаз наблюдателя, формируя изображение сцены на его сетчатке. Поместим перед глазом воображаемую картинную плоскость (экран) и будем считать, что изображение формируется на этой плоскости. Каждый луч, попадающий в глаз, проходит через некоторую точку экрана, формируя там изображение. Тем самым для построения изображения достаточно проследить весь путь распространения света, начиная от его источника.

Выпустим из каждого источника света пучок лучей во все стороны и мысленно проследим (оттрассируем) дальнейшее распространение каждого из них до тех пор, пока либо он не попадет в глаз наблюдателя, либо не покинет сцену. При попадании луча на границу объекта выпускаем из точки попадания отраженный и преломленный лучи и отслеживаем их и все порожденные ими лучи. Описанный процесс называется *прямой трассировкой лучей*. В результате его выполнения можно получить изображение сцены, однако он требует огромных вычислительных затрат. Основным недостатком прямой трассировки лучей является то обстоятельство, что в получаемое изображение сколько-нибудь существенный вклад вносит лишь очень небольшая часть трассируемых лучей. Тем самым при реализации этого метода основная часть работы оказывается проделанной впустую.

Чтобы избежать этого, попытаемся вместо трассирования всех лучей отслеживать лишь те лучи, которые вносят заметный вклад в строящееся изображение. Ясно, что это те лучи, которые попадают в глаз наблюдателя.

Для определения освещенности (цвета) точки экрана можно проследить путь, по которому мог пройти луч света, попавший в эту точку и сформировавший там изображение. Очевидно, что таким путем является путь луча, выходящего из глаза наблюдателя и проходящего через соответствующую точку экрана. Будем идти вдоль этого луча от глаза до точки ближайшего пересечения с каким-либо объектом сцены (при этом мы будем перемешаться в направлении, обратном направлению распространения света). Цвет соответствующей точки экрана будет определяться долей световой энергии, попадающей в эту точку и покидающей ее в направлении глаза. Для определения этой энергии необходимо

найти освещенность точки объекта, для чего из нее выпускаются лучи в тех направлениях, из которых может прийти энергия. Это, в свою очередь, может привести к определению точек пересечения соответствующих лучей с объектами сцены, выпускания новых лучей и так далее.

Описанный процесс называется ***обратной трассировкой лучей*** или просто трассировкой лучей. Именно этот метод и будет рассматриваться далее.

Ключевая задача метода трассировки лучей – определение освещенности произвольной точки объекта и той части световой энергии, которая уходит в заданном направлении. Эта энергия складывается из двух частей – непосредственной (первичной) освещенности, то есть энергии, непосредственно получаемой от источников света, и вторичной освещенности, то есть энергии, идущей от других объектов.

Конечно, такое деление носит условный характер. Ясно, что непосредственная освещенность вносит существенно больший вклад в изображение. Поэтому обычно первичная и вторичная освещенность рассматриваются по-разному. Отбрасываемая энергия состоит из той энергии, которая отражается и преломляется в заданном направлении. Для эффективного пользования методом трассировки лучей необходимо понимание физики процессов отражения и преломления.

Рассмотрим процесс распространения света. Известно, что свет можно рассматривать и как поток частиц, распространяющихся по прямолинейным траекториям, и как электромагнитную волну, распространяющуюся в пространстве. При этом интенсивность света определяется амплитудой волны, а его цвет – частотой или длиной волны. Сам процесс распространения света описывается уравнениями Максвелла.

Произвольный луч света можно рассматривать как сумму волн с различными длинами, распространяющихся в одном направлении. Вклад волны с дайной λ определяется функцией, называемой спектральной характеристикой данного луча света.

В действительности один и тот же воспринимаемый глазом цвет может вызываться бесконечным количеством различных источников света с различными спектральными кривыми характеристиками. Поэтому при исследовании обычно ограничиваются ко-

нечным набором значений длин волн, например для чистых красного, зеленого и синего цветов, и представляют все цвета в виде линейной комбинации этих базовых цветов.

Процесс распространения света распадается на две части – распространение света в однородной среде и взаимодействие света с границей раздела двух сред.

Распространение света в однородной среде происходит вдоль прямолинейной траектории с постоянной скоростью. Отношение скорости распространения света в вакууме к этой скорости называется коэффициентом преломления (индексом рефракции) среды. Обычно этот коэффициент зависит от длины волны.

При распространении света в среде может иметь место экспоненциальное затухание с некоторым коэффициентом затухания.

При взаимодействии с границей двух сред происходит отражение и преломление света. Рассмотрим несколько идеальных моделей, в каждой из которых границей раздела сред является плоскость.

1. Зеркальное отражение

Отраженный луч падает в точку P в направлении $\boldsymbol{i}$ и отражается в направлении, задаваемом вектором $\boldsymbol{r}$, определяемым следующим законом: вектор $\boldsymbol{r}$ лежит в той же плоскости, что и вектор $\boldsymbol{i}$ и единичный вектор внешней нормали к поверхности $\boldsymbol{n}$, а угол падения равен углу отражения.

2. Диффузное отражение

Идеальное диффузное отражение описывается законом Ламберта, согласно которому падающий свет рассеивается во все стороны с одинаковой интенсивностью. Таким образом, не существует однозначно определенного направления, в которым бы отражаются падающий луч, все направления равноправны и освещенность точки пропорциональна только доле площади, видимой от источника.

3. Идеальное преломление

Луч, падающий в точку P в направлении вектора $\mathbf{i}$, преломляется внутрь второй среды в направлении вектора $\boldsymbol{t}$. Преломление подчиняется закону Снеллиуса, согласно которому векторы $\boldsymbol{i}$, $\boldsymbol{n}$ и $\boldsymbol{t}$ лежат в одной плоскости.

4. Диффузное преломление

Диффузное преломление полностью аналогично диффузному отражению, при этом преломленный луч идет по всем направлениям с одинаковой интенсивностью.

Ясно, что все рассмотренные примеры являются идеализациями. На самом деле нет ни идеальных зеркал, ни идеально гладких поверхностей.

На практике обычно считают, что поверхность состоит из множества случайно ориентированных плоских идеальных микрозеркал (микрограней) с заданным законом распределения.

Теоретически возможно построить формулу, полностью описывающую энергию (и отраженную, и преломленную) в заданном направлении. Для этого необходимо выпустить лучи во все возможные стороны и вычислить приходящую оттуда энергию. Ясно, что на практике это невозможно. Поэтому желательно взять алгоритм, отслеживающий лишь конечное число направлений, вносящих в искомую величину наибольший вклад.

Введем некоторые ограничения на рассматриваемую сцену:

– будем рассматривать только точечные источники света;
– при трассировании преломленного луча будем игнорировать зависимость его направления от длины волны;
– будем считать освещенность объекта состоящей из диффузной и зеркальной частей (с заданными весами).

Для определения освещенности точки P определим сначала непосредственную освещенность этой точки от источников света (выпустив из нее лучи ко всем источникам).

Для определения вторичной освещенности выпустим из точки P один луч для отраженного направления и один луч для преломленного. Тем самым для определения освещенности точки необходимо будет отслеживать лишь небольшое количество лучей.

При этом неидеально зеркальное отражение лучей, идущих от других объектов, игнорируется.

Обычно для компенсации всех таких неучитываемых величин вводится так называемое фоновое освещение – равномерное освещение со всех сторон, которое ни от чего не зависит и не затеняется.

Несмотря на то, что коэффициенты Френеля заметно влияют на степень реалистичности изображения, на практике их применяют очень редко. Дело в том, что их использование наталкивается

на ряд серьезных препятствий, одним из которых является сложность вычисления, а другим – отсутствие точной информации о зависимости величин, входящих в состав соответствующей формулы, от длины волны.

Существуют определенные методы интерполяции коэффициентов Френеля (например, линейная), которые в сочетании с табличным способом задания способны заметно ускорить процесс их вычисления.

Часто используется модель Уиттеда трассировки лучей: через каждый пиксел экрана луч трассируется до ближайшего пересечения с объектами сцены. Из точки пересечения выпускаются лучи ко всем источникам света для проверки их видимости и определения непосредственной освещенности точки пересечения. Выпускаются также отраженный и преломленный лучи, которые, трассируются, в свою очередь, до ближайшего пересечения с объектами сцены, и так далее. Получается рекурсивный алгоритм трассировки.

В качестве критерия остановки обычно используется отсечение по глубине (не более заданного количества уровней рекурсии) и по весу (чем дальше, тем меньше вклад каждого луча в итоговой цвет пиксела, и, как только этот вклад опускается ниже некоторого порогового значения, дальнейшая трассировка этого луча прекращается).

5.2. Освещение и свойства отображаемых объектов в OpenGL

Нормаль – это вектор, определяющий направление зеркального отражения и определяющий ориентацию грани. Это одна из характеристик, ассоциированных с текущей точкой и для ее определения в OpenGL есть специальные команды

GlNormal3[b s i f d](type coords)

GlNormal3[b s i f d]v(type coords)

для присвоения значений координатам вектора нормали в диапазоне [–1.0, 1.0] – изменение значений этих координат позволяет получить интересные эффекты при использовании освещения.

Свойства материала. В OpenGL задаются командами

GlMaterial[i f](Glenum face, Glenum pname, Gltype param),

GlMaterial[i f](Glenum face, Glenum pname, Gltype* param),

в которых первый аргумент задает, к каким поверхностям применяются задаваемые параметры, и может быть GL_FRONT, GL_BACK, GL_FRONT_AND_BACK. Второй аргумент в первом варианте команды может быть только GL_SHININESS для задания степени зеркального отражения, а во втором варианте может принимать значения

GL_AMBIENT	param содержит 4 значения RGBA – рассеянного цвета материала
GL_DIFFUSE	то же для цвета диффузного отражения
GL_SPECULAR	то же для цвета зеркального отражения
GL_EMISSION	то же излучаемого материалом света
GL_SHININESS	param содержит одно значение степени зеркального отражения в диапазоне 0–128

Третий аргумент содержит сами значения, которые присваиваются соответствующему параметру.

Каждый параметр устанавливается отдельным вызовом команды.

Грани – лицевые или «с изнанки» – плоские поверхности многоугольников. Какую грань считать лицевой определяет команда **glFrontFace(Glenum mode)**. Если ее аргумент CL_CW – лицевая та, для которой направление обхода вершин по часовой стрелке, а если CL_CCW – лицевая с направлением обхода против часовой стрелки.

Для исключения из отображения граней одного из типов используют команду **glCullFace(Glenum mode)**, аргумент которой может быть GL_FRONT, GL_BACK. Эта команда используется после разрешающей **glEnable(GL_CULL_FACE)**.

Источник света. Его параметры задаются командами

GlLight[i f](Glenum light, Glenum pname, Glfloat param),

где первый аргумент есть номер источника света от 0 до GL_MAX_LIGHTS<=8; второй аргумент определяет, для какого параметра будет значение в третьем аргументе и может быть:

GL_SPOT_EXPONENT	для задания числового значения распределения интенсивности света от 0 до 128
GL_SPOT_CUTOFF	для максимального угла разброса от 0 до 90 или 180
GL_CONSTANT_ATTENUATION	постоянное ослабление

GL_LINEAR_ATTENUATION	линейное
GL_QUADRATIC_ATTENUATION	квадратичное

Векторная версия команды **GlLight[i f]v(Glenum light, Glenum pname, Glfloat* param)** дает дополнительно возможность задать вторым аргументом

GL_AMBIENT	в 3-м аргументе 4 значения интенсивности фонового освещения
GL_DIFFUSE	то же для интенсивности диффузного освещения
GL_SPECULAR	то же для интенсивности освещения зеркального отражения
GL_POSITION	в третьем аргументе 4 значения мировых однородных координат источника света; если четвертая координата 0, то источник считается направленным и ослабление заблокировано; по умолчанию источник в точке (0,0,1,0) и направленное излучение вдоль оси *z*
GL_SPOT_DIRECTION	в третьем аргументе три значения составляющих вектора направления света, по умолчанию (0,0,–1)

Модель освещения. Ее параметры задаются командами

GlLightModel[i f](Glenum pname, Glenum param),

GlLightModel[i f]v(Glenum pname, Glenum* param),

первый аргумент которых определяет, что будет означать значение второго и может быть:

GL_LIGHT_MODEL_LOCAL_VIEWER – третий аргумент содержит булево значение положения наблюдателя: если FALSE – то наблюдатель в начале видовой системы координат, иначе – направление обзора вдоль оси *z*.

GL_LIGHT_MODEL_TWO_SIDE – третий аргумент булевское значение одно- или двустороннего освещения.

В векторной версии команды можно еще использовать

GL_LIGHT_MODEL_AMBIENT – в 3-м аргументе 4 значения полной фоновой интенсивности света.

В приведенном ниже примере мы пытаемся осветить различные координатные отсеки с объектами различных оптических свойств. В одном небольшом примере невозможно отразить мно-

гочисленные возможности библиотеки по работе с источниками света и свойствами отображаемых объектов, но хочется надеяться, что приведенная программа станет основой для вашего экспериментирования; к сожалению, квалификация авторов не позволяет дать исчерпывающие рекомендации по выбору численных кодов оптических свойств освещаемых материалов, источников света и их взаимной ориентации.

```
/*В этой программе добавлен движущийся источник света и
оптические свойства графических объектов, размещенных в
координатных отсеках, */
#include<windows.h>
#include<gl\gl.h>
#include<gl\glu.h>
#include<gl\glaux.h>

#pragma warning(disable : 4305)

GLint w=800,h=600;
GLfloat alpha, //Азимут - широта
        betha, //Угол места - долгота
        radius; //Радиус

//Свойства источника света
GLfloat ambient[4] = { 0.7, 0.7, 0.7, 1.0 };
GLfloat diffuse[4] = { 0.8, 0.8, 0.8, 1.0 };
GLfloat specular[4]= { 1.0, 1.0, 1.0, 1.0 };

//Определим позиции источников света в отсеках
GLfloat LposF[] = { 20.0, 20.0, 20.0, 1.0 };
GLfloat LposB[] = { -1.0, -1.0, -1.0, 1.0 };

//Вектор направления света при направленном источнике
GLfloat Ldir0[] = {0.0,0.0,1.0,1.0};

//Модель освещения
GLfloat lmodel_ambient[] = { 0.2, 0.2, 0.2, 1.0 };
GLfloat local_view[] = { 0.0 };
static GLfloat whiteAmbient[] = {0.3, 0.3, 0.3, 1.0};
static GLfloat redAmbient[] = {0.3, 0.1, 0.1, 1.0};
static GLfloat greenAmbient[] = {0.1, 0.3, 0.1, 1.0};
static GLfloat blueAmbient[] = {0.1, 0.1, 0.3, 1.0};
static GLfloat whiteDiffuse[] = {1.0, 1.0, 1.0, 1.0};
static GLfloat redDiffuse[] = {1.0, 0.0, 0.0, 1.0};
static GLfloat greenDiffuse[] = {0.0, 1.0, 0.0, 1.0};
static GLfloat blueDiffuse[] = {0.0, 0.0, 1.0, 1.0};
static GLfloat whiteSpecular[] = {1.0, 1.0, 1.0, 1.0};
```

```
static GLfloat redSpecular[] = {1.0, 0.0, 0.0, 1.0};
static GLfloat greenSpecular[] = {0.0, 1.0, 0.0, 1.0};
static GLfloat blueSpecular[] = {0.0, 0.0, 1.0, 1.0};
static GLfloat angle = 0.0;

/*Прототипы функций инициализации, рисования и обновления
окна*/
GLvoid Init(GLvoid);
GLvoid CALLBACK Draw(GLvoid);
GLvoid CALLBACK Reshape(GLsizei, GLsizei);
#pragma warning(disable : 4101)

//Начальная инициализация
GLvoid Init(void)
{
  GLfloat maxObjectSize, aspect; /*Максимальный размер
объекта и угол обзора*/
  GLdouble near_plane, far_plane; //Отсечение по глубине
  auxInitPosition(0,0,w,h); //Позиция и размер окна
  auxInitDisplayMode(AUX_RGB | AUX_DEPTH | AUX_DOUBLE);
//Видеорежим
  auxInitWindow("Демонстрация методов рисования
многоугольников, света и свойств материала"); /*Вывести
окно с заголовком*/
  /*Функция, которая будет вызываться при отсутствии
сообщений в очереди ОС*/
  auxIdleFunc(Draw);
  /*Функция, которая вызывается при изменениях размеров
окна*/
  auxReshapeFunc(Reshape);
  glClearDepth(1.0); //Очистка буфера глубины
  glEnable(GL_DEPTH_TEST);
  glEnable(GL_LIGHTING); //Разрешаем работу с осветителем
  //glLightfv(GL_LIGHT0, GL_POSITION,LposB);
  //Установим начальную позицию источника света
  glLightfv(GL_LIGHT0, GL_POSITION, LposF);
  //Зададим свойства излучателя
  glLightfv(GL_LIGHT0, GL_AMBIENT, ambient);
  glLightfv(GL_LIGHT0, GL_DIFFUSE, diffuse);
  glLightfv(GL_LIGHT0, GL_SPECULAR, specular);
  glLightModelf(GL_LIGHT_MODEL_TWO_SIDE, 1.0);
  glEnable(GL_LIGHT0); //Разрешаем истчник номер 0
  glMatrixMode(GL_PROJECTION);
  //aspect=(GLfloat)w/h;
  //Формирование матрицы перспективного проектирования
  //gluPerspective(65.0, aspect, 3.0, 7.0 );
  //glMatrixMode( GL_MODELVIEW );
  alpha=0.0;
```

```
  betha=0.0;
  //цвет фона
  glClearColor(0.7f,0.7f,0.7f,1.0f);
}

/*При перерисовке повторяем фрагмент инициализации для
новых размеров окна*/
static void CALLBACK Reshape(int width, int height){
  GLfloat aspect;
  glViewport(0, 0, width, height);
  aspect=(GLfloat)width/height;
  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();
  //gluPerspective(-65.0,aspect,60.0, 60.0 );
  glOrtho(-60.0,60.0,-60.0,60.0,-60.0,60.0);
  //glFrustum(-60.0,60.0,-60.0,60.0,-60.,60.);
  /*
    gluLookAt(0.0,0.0,0.0, //Позиция наблюдателя
              3.0,2.0,1.0, //Куда смотрит наблюдатель
              0.0,1.0,0.0); //Вектор вверх
  */
  glMatrixMode(GL_MODELVIEW);
}

//Рисование источника света
GLvoid drawLight(GLvoid)
{
  glPushAttrib(GL_LIGHTING_BIT);
  glDisable(GL_LIGHTING);
  glColor3f(1.0, 1.0, 1.0); //Белый свет
  auxSolidDodecahedron(1.0); /*Форма источника – маленький
додекаэдр*/
  glPopAttrib();
}

//Рисование всего на сцене
static void CALLBACK Draw(void)
{
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
  alpha+=6.0;
  glPushMatrix();
  angle+=1.0;
  //Матрицы поворота для всей системы
  glRotatef(angle,1.0, 0.0, 0.0);
  glRotatef(angle,0.0,1.0, 0.0);
  glRotatef(angle/4,0.0,.0, 1.0);
  glEnable(GL_LINE_SMOOTH);
```

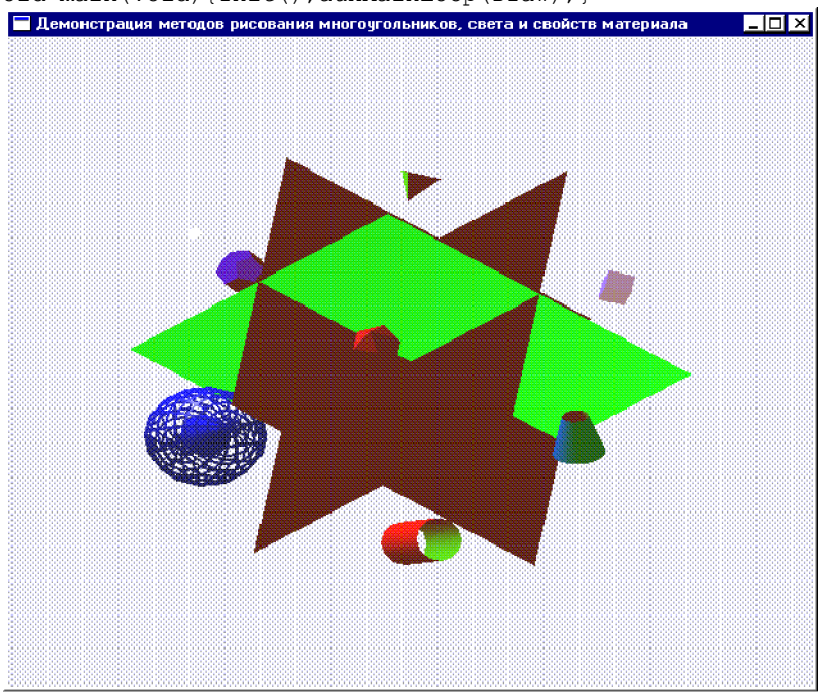
```
  /*Нарисуем 3 координатные плоскости в виде
прямоугольников*/
  //Зададим прямоугльники 4-мя вершинами каждый
  glPolygonMode(GL_FRONT_AND_BACK,GL_FILL);
  glBegin(GL_QUADS);
  //Плоскость XY
  glPushAttrib(GL_LIGHTING_BIT);
  //glEnable(GL_COLOR_MATERIAL);
  /*Оптические свойства лцевой и обратной сторон
поверхности*/
  glMaterialfv(GL_BACK, GL_AMBIENT, redAmbient);
  glMaterialfv(GL_BACK, GL_DIFFUSE, redDiffuse);
  glMaterialfv(GL_FRONT, GL_AMBIENT, redAmbient);
  glMaterialfv(GL_FRONT, GL_DIFFUSE, greenDiffuse);
  glMaterialfv(GL_FRONT, GL_SPECULAR,blueSpecular);
  glMaterialf(GL_FRONT, GL_SHININESS, 100.0);
  glMaterialfv(GL_BACK, GL_SPECULAR,redSpecular);
  glMaterialf(GL_BACK, GL_SHININESS, 100.0);
  /*Координаты вершин прямоугольника координатной
плоскости*/
  glVertex4f(30.0,30.0,0.0,1.0);
  glVertex4f(-30.0,30.0,0.0,1.0);
  glVertex4f(-30.0,-30.0,0.0,1.0);
  glVertex4f(30.0,-30.0,0.0,1.0);
  //glDisable(GL_COLOR_MATERIAL);
  glPopAttrib();
  //Плоскость XZ
  glPushAttrib(GL_LIGHTING_BIT);
  glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT,
greenAmbient);
  glMaterialfv(GL_FRONT_AND_BACK, GL_DIFFUSE,
greenDiffuse);
  glMaterialfv(GL_FRONT_AND_BACK, GL_SPECULAR,
greenSpecular);
  glMaterialf(GL_FRONT, GL_SHININESS, 100.0);
  glVertex4f(30.0,0.0,30.0,1.0);
  glVertex4f(30.0,0.0,-30.0,1.0);
  glVertex4f(-30.0,0.0,-30.0,1.0);
  glVertex4f(-30.0,0.0,30.0,1.0);
  glPopAttrib();
  //Плоскость YZ
  glPushAttrib(GL_LIGHTING_BIT);
  glEnable(GL_COLOR_MATERIAL);
  glMaterialfv(GL_FRONT_AND_BACK,GL_AMBIENT,redAmbient);
  glMaterialfv(GL_FRONT_AND_BACK, GL_DIFFUSE,
greenDiffuse);
  glMaterialfv(GL_FRONT_AND_BACK, GL_SPECULAR,
blueSpecular);
```

```
  glMaterialf(GL_FRONT, GL_SHININESS, 100.0);
  glVertex4f(0.0,30.0,30.0,1.0);
  glVertex4f(0.0,-30.0,30.0,1.0);
  glVertex4f(0.0,-30.0,-30.0,1.0);
  glVertex4f(0.0,30.0,-30.0,1.0);
  glPopAttrib();
  //glDisable(GL_COLOR_MATERIAL);
  glEnd();
  //Нарисуем источник света
  glPushMatrix();
  betha+=2.0;
  glRotatef(betha, 1.0, 0.0, 1.0);
  glRotatef(betha, 0.0, 0.0, 1.0);
  glTranslatef(30.0, 20.0, 20.0);
  glLightfv(GL_LIGHT0, GL_POSITION,LposF);
  drawLight();
  glPopMatrix();
  /*Рисуем 3-мерные тела в координатных отсеках со
свойствами материалов*/
  //1 - тетраэдр
  glPushAttrib(GL_LIGHTING_BIT);
  glMaterialfv(GL_FRONT, GL_AMBIENT, redAmbient);
  glMaterialfv(GL_FRONT, GL_DIFFUSE, greenDiffuse);
  glMaterialfv(GL_FRONT, GL_SPECULAR, whiteSpecular);
  glMaterialf(GL_FRONT, GL_SHININESS, 100.0);
  glPushMatrix();
  glTranslatef(20.0,20.0,20.0);
  glRotatef(alpha,1,0,0);
  auxSolidTetrahedron(4.0); //Тетраэдр
  glTranslatef(-20.0,-20.0,-20.0);
  glPopMatrix();
  glPopAttrib();
  glFinish();
  //2 - куб, излучающий свет
  glPushAttrib(GL_LIGHTING_BIT);
  glMaterialfv(GL_FRONT, GL_EMISSION, whiteAmbient);
  glMaterialfv(GL_FRONT, GL_DIFFUSE, blueDiffuse);
  glMaterialfv(GL_FRONT, GL_SPECULAR, whiteSpecular);
  glMaterialf(GL_FRONT, GL_SHININESS, 100.0);
//Можно задать свойства материала еще и так - попробуйте:
  //glEnable(GL_COLOR_MATERIAL);
  //glColorMaterial(GL_FRONT,GL_EMISSION);
  //glColor4f(0.8,0.0,0.6,1.0);
  glPushMatrix();
  glTranslatef(20.0,20.0,-20.0);
  glRotatef(alpha,0,1,0);
  auxSolidCube(4.0); //Гексаэдр - он же куб
  glTranslatef(-20.0,-20.0,20.0);
```

```
  glPopMatrix();
  glPopAttrib();
  //glDisable(GL_COLOR_MATERIAL);
  glFinish();
  //3 - додекаэдр
  glPushAttrib(GL_LIGHTING_BIT);
  glMaterialfv(GL_FRONT, GL_AMBIENT, redAmbient);
  glMaterialfv(GL_FRONT, GL_DIFFUSE, redDiffuse);
  glMaterialfv(GL_FRONT, GL_SPECULAR, whiteSpecular);
  glMaterialf(GL_FRONT, GL_SHININESS, 100.0);
  glPushMatrix();
  glTranslatef(-20.0,20.0,-20.0);
  glRotatef(alpha,0,0,1);
  auxSolidDodecahedron(4.0); //
  glTranslatef(20.0,-20.0,20.0);
  glPopMatrix();
  glPopAttrib();
  //4 - тоже додекаэдр другого цвета
  glPushAttrib(GL_LIGHTING_BIT);
  glMaterialfv(GL_FRONT, GL_AMBIENT, redAmbient);
  glMaterialfv(GL_FRONT, GL_DIFFUSE, blueDiffuse);
  glMaterialfv(GL_FRONT, GL_SPECULAR,whiteSpecular);
  glMaterialf(GL_FRONT, GL_SHININESS, 100.0);
  glPushMatrix();
  glTranslatef(-20.0,20.0,20.0);
  glRotatef(alpha,1,0,0);
  auxSolidDodecahedron(4.0); //Додекаэдр
  glTranslatef(20.0,-20.0,-20.0);
  glPopMatrix();
  glPopAttrib();
  //5 - икосаэдр
  glPushAttrib(GL_LIGHTING_BIT);
  glMaterialfv(GL_FRONT, GL_AMBIENT, redAmbient);
  glMaterialfv(GL_FRONT, GL_DIFFUSE, redDiffuse);
  glMaterialfv(GL_FRONT, GL_SPECULAR, whiteSpecular);
  glMaterialf(GL_FRONT, GL_SHININESS, 100.0);
  glPushMatrix();
  glTranslatef(20.0,-20.0,20.0);
  glRotatef(alpha,0,1,0);
  auxSolidIcosahedron(4.0); //Икосаэдр
  glTranslatef(-20.0,20.0,-20.0);
  glPopMatrix();
  glPopAttrib();
  glFinish();
  /*6 тор или усеченный конус – на ваш выбор манипуляциями
с комментариями*/
  glPushAttrib(GL_LIGHTING_BIT);
  glMaterialfv(GL_FRONT, GL_AMBIENT, greenAmbient);
```

```
 glMaterialfv(GL_FRONT, GL_DIFFUSE, blueDiffuse);
 glMaterialfv(GL_FRONT, GL_SPECULAR, whiteSpecular);
 glMaterialf(GL_FRONT, GL_SHININESS, 100.0);
 glPushMatrix();
 glTranslatef(20.0,-20.0,-20.0);
 glRotatef(alpha,0,1,1);
 //auxSolidTorus(2.0,4.0); //
//Создаем квадратичный объект и определяем его параметры
 GLint orientation = GLU_OUTSIDE,
 radius1 = 40,
 radius2 = 20,
 slices = 26,
 stacks = 20,
 height = 80,
 whichQuadric = 0;
 GLfloat angle1 = 90,
 angle2 = 180;
 GLUquadricObj *quadObj = gluNewQuadric();
 gluCylinder(quadObj,radius1/10.0, radius2/10.0,
             height/10.0, slices, stacks);
 glTranslatef(-20.0,12.0,20.0);
 glPopMatrix();
 glPopAttrib();
 //7 - цилиндр
 glPushAttrib(GL_LIGHTING_BIT);
 glMaterialfv(GL_FRONT, GL_AMBIENT, redAmbient);
 glMaterialfv(GL_FRONT, GL_DIFFUSE, redDiffuse);
 glMaterialfv(GL_FRONT, GL_SPECULAR, whiteSpecular);
 glMaterialf(GL_FRONT, GL_SHININESS, 100.0);
 glPushMatrix();
 glTranslatef(-20.0,-20.0,-20.0);
 glRotatef(alpha,0,0,1);
 auxSolidCylinder(4.0,10.0); //
 glTranslatef(20.0,20.0,20.0);
 glPopMatrix();
 //8 – проволочная модель сферы и конус
 glPushAttrib(GL_LIGHTING_BIT);
 glMaterialfv(GL_FRONT, GL_AMBIENT,blueAmbient);
 glMaterialfv(GL_FRONT, GL_DIFFUSE, blueDiffuse);
 glMaterialfv(GL_FRONT, GL_SPECULAR, whiteSpecular);
 glMaterialf(GL_FRONT, GL_SHININESS, 100.0);
 glTranslatef(-20.0,-20.0,20.0);
 glRotatef(alpha,1,0,0);
 glRotatef(alpha,0,1,0);
 auxSolidCone(4.0,8.0); //
 auxWireSphere(9);
 glTranslatef(20.0,20.0,-20.0);
 glPopMatrix();
```

```
  glPopAttrib();
  glDisable(GL_LINE_SMOOTH );
  glDisable(GL_POINT_SMOOTH );
  glPopMatrix();
  glFinish();
  SwapBuffers(wglGetCurrentDC());
}

void main(void){Init();auxMainLoop(Draw);}
```



6. Текстура поверхности материала и работа с ней в OpenGL

Под текстурой в компьютерной графике подразумевают одно- или двумерное изображение, накладываемое на отображаемую поверхность – это может быть или некоторый узор на гладкой поверхности, или преобразование поверхности путем создания на ней неровностей внесением возмущений в ее оптические параметры.

Для использования текстуры с целью придания реалистичности виртуальным графическим объектам необходимо выполнить следующие действия:

- получить из файла или создать программно рисунок текстуры;
- создать образ текстуры в памяти;
- задать параметры текстуры и способ ее взаимодействия с объектом, на который она будет наложена.

6.1. Получение текстурной заготовки в формате OpenGL

OpenGL имеет собственный внутренний формат графических образов, поэтому для воспроизведения растрового изображения необходимо преобразовать входной образ в этот внутренний формат, представляющий собой просто последовательность троек интенсивностей компонентов красного, зеленого и синего цветов. Самый простой способ преобразования формата Windows DIB в формат OpenGL – использовать специально предназначенную для этого команду

AUX_RGBImageRec *auxDIBImageLoad(strFile),

принимающую имя файла и возвращающую указатель на структуру с образом в формате OpenGL. Более сложный способ «ручного» преобразования вы найдете в [2].

6.2.Создание текстуры в памяти

Обязательное условие, которое необходимо выполнить – привести размеры изображения, используемого в качестве текстуры, к

кратным степеням двойки, ближайшим к исходным размерам. Алгоритм и реализующую его подпрограмму мы позаимствуем из [1]:

```
BOOL TransDibToOpenGL()
{
  GLint glMaxTexDim ;
  double xPow2, yPow2;
  int ixPow2, iyPow2;
  int xSize2, ySize2;
  /*Получим от OpenGL максимально допустимый размер
текстуры*/
  glGetIntegerv(GL_MAX_TEXTURE_SIZE, &glMaxTexDim);
  //Ограничимся размерами не более 256 пикселей
  glMaxTexDim = min(256, glMaxTexDim);
  //вычислим показатели степени двойки по каждому размеру
  if(m_iWidth<=glMaxTexDim)
    xPow2=log((double)m_iWidth)/log(2.0);
  else xPow2=log((double)glMaxTexDim)/log(2.0);
  if (m_iHeight <= glMaxTexDim)
    yPow2 = log((double)m_iHeight) / log(2.0);
  else yPow2 = log((double)glMaxTexDim) / log(2.0);
  ixPow2 = (int)xPow2;
  iyPow2 = (int)yPow2;
  //Если произошло усечение, добавим 1
  if (xPow2 != (double)ixPow2) ixPow2++;
  if (yPow2 != (double)iyPow2) iyPow2++;
  //Вычислим новые размеры
  xSize2 = 1 << ixPow2; ySize2 = 1 << iyPow2;
  //Выделим память под новый образ
 BYTE *pData=(BYTE*)malloc(xSize2*ySize2*3*sizeof(BYTE));
  if (!pData) return FALSE;
  //Преобразуем образ
  BOOL bRes = gluScaleImage(GL_RGB, m_iWidth, m_iHeight,
GL_UNSIGNED_BYTE, m_pBits, xSize2, ySize2,
GL_UNSIGNED_BYTE, pData);
  if(bRes){
    OutputGlError("Ошибка выполнения команды
gluScaleImage");
    return FALSE;
  }
  //Освобождаем память под старым образом
  free(m_pBits);
  //Переустановим размеры изображения текстуры
  m_pBits=pData; m_iWidth=xSize2; m_iHeight=ySize2;
  return TRUE;
}
```

Здесь используется команда OpenGL

```
gluScaleImage(
```

```
Glenum format, //(из набора форматов данных пикселей
//GL_COLOR_INDEX,GL_STENCIL_INDEX, GL_DEPTH_COMPONENT
//GL_RED, GL_GREEN, GL_BLUE, GL_ALPHA, GL_RGB, GL_RGBA,
//GL_LUMINANCE)
Glint win, Glint hin, /*ширина и высота источника
масштабируемого образа*/
Glenum typein, /*тип данных для datain - GL_BITMAP,
GL_INT, GL_FLOAT и пр.*/
const void* datain, //указатель на приемник образа
Glint wout, Glint hout, //размеры приемника образа
Glenum typeout, //тип данных для dataout
void * dataout) //адрес приемника образа
```

После подготовки образа можно создавать текстуру в памяти с помощью команд

void glTexImage1D(GLenum target, GLint level, GLint components, GLsizei width, GLint border, GLenum format, GLenum type, const GLvoid *pixels);

void glTexImage2D(GLenum target, GLint level, GLint components, GLsizei width, GLsizei height, GLint border, GLenum format, GLenum type, const GLvoid *pixels);

Аргументы этих команд:

target – тип создаваемой текстуры GL_TEXTURE_1D или GL_TEXTURE_2D;

level – число уровней детализации текстуры: 0 – базовый, k – уменьшенный в k раз

components – 1, 2, 3 или 4 цветовых компонента текстуры;

width, height – размеры образа = степень двойки + 2*border;

border = 0 или 1 – ширина границы;

format – формат данных пикселей:

GL_COLOR_INDEX	каждый элемент – индекс цвета
GL_RED	каждый элемент – красный компонент
GL_GREEN, GL_BLUE, GL_ALPHA	аналогично
GL_RGB	каждый элемент – трехкомпонентный
GL_RGBA	четырехкомпонентные элементы
GL_LUMINANCE	один компонент яркости
GL_LUMIMANCE_ALPHA	четыре компонента яркости

type – тип данных пикселя;

pixels – указатель на образ данных в памяти.

Для работы с текстурой необходимо разрешить соответствующий режим командой **glEnable(GL_TEXTURE_[1 2]D)**.

В OpenGL есть еще 2 команды, иногда позволяющие с меньшими затратами решить ту же задачу:

int gluBuild2DMipmaps(GLenum target, GLint components, GLint width, GLint height, GLenum format, GLenum type, const void *data);

int gluBuild1DMipmaps(GLenum target, GLint components, GLint width, GLenum format, GLenum type, const void *data);

Они получают входной образ data и формируют образы всех уровней детализации – параметры этих команд такие же, как у glTexImage[1 2]D.

6.3. Параметры текстуры

Несколько элементов текстурного образа могут отображаться в один элемент на экране – для полного соответствия образа и его отображения в массиве образа определяют 4 точки, которые будут отображены в 4 угла экранного элемента. Эти точки соединяются в четырехугольник, значения попадающих в него элементов взвешиваются с учетом доли каждого и суммируются. Настройка параметров текстуры для учета этих и других особенностей осуществляется вызовом команды

void glTexParameterf(GLenum target, GLenum pname, GLfloat param);

void glTexParameterfv(GLenum target, GLenum pname, GLfloat *params);

или их целочисленных модификаций.

Параметры команд:

target – GL_TEXTURE_1D, GL_TEXTURE_2D;

pname – имя параметра текстуры:

GL_TEXTURE_MIN_FILTER	определяет функцию уменьшения текстуры, если площадь пиксела больше элемента текстуры
GL_TEXTURE_MAX_FILTER	функция увеличения текстуры
GL_TEXTURE_WRAP_S	устанавливает параметр сворачивания координаты *s* тек-

	стуры
GL_TEXTURE_ WRAP_T	устанавливает параметр сворачивания координаты *s* текстуры
GL_TEXTURE_BORDER_COLOR	устанавливает цвет бордюра

param – значение параметра pname;

params – указатель на массив значений pname, задающий функцию подгонки текстуры к пикселю и может принимать значения:

GL_NEAREST	возвращает значение ближайшего к центру элемента текстуры
GL_LINEAR	среднее арифметическое значений четырех центральных элементов текстуры
GL_NEAREST_MIPMAP_NEAREST	выбирает уровень детализации
GL_LINEAR_MIPMAP_NEAREST	выбирает уровень детализации
GL_NEAREST_MIPMAP_LINEAR	выбирает 2 уровня детализации
GL_LINEAR_MIPMAP_LINEAR	выбирает 2 уровня детализации

6.4. Взаимодействие текстуры с объектом

Для указания способа взаимодействия текстуры и объекта используются команды:

void glTexEnvf(GLenum target, GLenum pname, GLfloat param);

void glTexEnvfv(GLenum target, GLenum pname, Glfloat* params);

и их целочисленные модификации.

Параметры команд:

target – GL_TEXTURE_ENV;

pname – GL_TEXTURE_ENV_MODE и для векторного варианта еще GL_TEXTURE+ENV_COLOR;

param = GL_MODULATE | GL_DECAL | GL_BLEND4

params – указатель на массив параметров или на константу или на цвет RGBA.

В режиме GL_DECAL текстура покрывает объект без учета его собственного цвета, в режиме GL_MODULATE цвета в текстуре зависят от цвета подложки.

6.5. Координаты текстуры

Преобразование координат при нанесении рисунка обязательно, но сначала надо каждой определяемой вершине приписать соответствующие точки в координатах текстуры; это можно сделать несколькими способами, один из которых состоит в использовании команд

void glTexCoord[1 2 3 4][s i f d](type coord);

void glTexCoord[1 2 3 4][s i f d](type* coord);

glTexCoord1* устанавливает координаты в значение (s,0,0,1), glTexCoord2* – в (s,t,0,1), glTexCoord3* в (s,t,r,1).

Второй путь – использовать функцию для расчета координат текстуры для каждой вершины:

void glTexGen[i f d](GLenum coord, GLenum pname, GLdouble param);

coord определяет координату текстуры, к которой будет применяться функция и может быть GL_S, GL_T,GL_R,GL_Q;

pname определяет саму функцию формирования координат и может быть GL_TEXTURE_GEN_MODE, GL_OBJECT_PLANE, GL_EYE_PLANE;

param определяет единственное значение текстуры и может быть: GL_OBJECT_LINEAR, GL_EYE_LINEAR, GL_SPHERE_MAP.

В приводимом ниже примере программы текстуры создаются из рисунков файлового хранения bmp–формата (мы использовали первые попавшиеся – бетон, металл и пейзаж) и накладываются на координатные плоскости.

В закомментированном виде приводится также способ программного создания образа текстуры – вы можете опробовать его, взяв в комментарий файловые фрагменты программы для создания текстуры.

```
/* Строится сцена в виде 2-х стен из бетона (более
подходящего файла не оказалось под рукой), пола, покрытого
чем-то похожим на цветной линолеум; на одной стене
```

```
установлена дверь из кованого металла, а на другой –
картина (лесной пейзаж). Все – наложением файловых
текстур. Вы можете модифицировать все по своему вкусу.*/

#include "glos.h"
#include<stdio.h>
#include<stdlib.h>
#include<math.h>
#include <GL/gl.h>
#include <GL/glu.h>
#include <GL/glaux.h>

void Init(void);
void CALLBACK Reshape(GLsizei w, GLsizei h);
void CALLBACK DrawScene(void);

//Параметры вращения системы координат
GLfloat latitude, longitude, radius;
//Прототипы функций
void polarView( GLdouble, GLdouble, GLdouble, GLdouble);

#define N 3 //Количество массивов контрольных точек
GLsizei w,h; //Для размеров окна
/*
GLfloat ctrlpoints[N][4][3]= //Массивы контрольных точек
{
{{1,1,0},{-1,1,0},{-1,-1,0},{1.-1,0}},
{{1,0,1},{1,0,-1},{-1,0,-1},{-1,0,1}},
{{0,1,1},{0,1,-1},{0,-1,-1},{0,-1,1}}
};
*/
GLfloat UMAX,UMIN,VMAX,VMIN,DU,DV;
GLfloat texpts[2][2][2] = {{{0.0, 0.0}, {0.0, 1.0}},{{1.0,
0.0}, {1.0, 1.0}}};

#define imageWidth 64 //Для размеров образа текстуры
#define imageHeight 64
#define NUM_TEXTURES 5

GLuint texobj[NUM_TEXTURES];
//Массивы текстур
GLubyte image[NUM_TEXTURES][3*imageWidth*imageHeight];
//Для указателей на текстуры, получаемые из файла
AUX_RGBImageRec* imF[NUM_TEXTURES];
GLint iW[NUM_TEXTURES],iH[NUM_TEXTURES];/*Для размеров
файловых текстур*/
//Для указателей битовых образов текстур
GLvoid * iBits[NUM_TEXTURES];
```

```
//Функция масштабирования файловых текстур
BOOL TexMapScalePow2(GLint i)
{
  GLint glMaxTexDim ;
  double xPow2, yPow2;
  int ixPow2, iyPow2, xSize2, ySize2;
  glGetIntegerv(GL_MAX_TEXTURE_SIZE, &glMaxTexDim);
  glMaxTexDim = min(256, glMaxTexDim);
  if (iW[i] <= glMaxTexDim)
    xPow2 = log((double)iW[i]) / log(2.0);
  else xPow2 = log((double)glMaxTexDim) / log(2.0);
  if (iH[i] <= glMaxTexDim)
    yPow2 = log((double)iH[i]) / log(2.0);
  else yPow2 = log((double)glMaxTexDim) / log(2.0);
  ixPow2 = (int)xPow2; iyPow2 = (int)yPow2;
  if (xPow2 != (double)ixPow2) ixPow2++;
  if (yPow2 != (double)iyPow2) iyPow2++;
  xSize2 = 1 << ixPow2;
  ySize2 = 1 << iyPow2;
 BYTE *pData=(BYTE*)malloc(xSize2*ySize2*3*sizeof(BYTE));
  if (!pData) return FALSE;
  BOOL bRes=gluScaleImage(GL_RGB,iW[i],iH[i],
GL_UNSIGNED_BYTE,iBits[i],xSize2,ySize2,GL_UNSIGNED_BYTE,
pData);
  if(bRes){
    printf("Ошибка выполнения команды gluScaleImage");
    return FALSE;
  }
  free(iBits[i]);Bits[i]=pData;iW[i]=xSize2;iH[i]=ySize2;
  return TRUE;
}

//Функция создания битовых образов текстур из файлов
void makeImageF(void)
{
  imF[0]=auxDIBImageLoad("backgrd1.bmp");//Пейзаж
  iW[0]=imF[0]->sizeX;
  iH[0]=imF[0]->sizeY;
  iBits[0]=imF[0]->data;
  TexMapScalePow2(0);
  gluBuild2DMipmaps(GL_TEXTURE_2D,3,iW[0],iH[0],GL_RGB,
GL_UNSIGNED_BYTE,iBits[0]);
  imF[1]=auxDIBImageLoad("Rock.bmp"); //Цемент
  iW[1]=imF[1]->sizeX;
  iH[1]=imF[1]->sizeY;
  iBits[1]=imF[1]->data;
  TexMapScalePow2(1);
```

```
  gluBuild2DMipmaps(GL_TEXTURE_2D,3,iW[1],iH[1],GL_RGB,
GL_UNSIGNED_BYTE,iBits[1]);
  imF[2]=auxDIBImageLoad("metalbal.bmp"); //Металл
  iW[2]=imF[2]->sizeX;
  iH[2]=imF[2]->sizeY;
  iBits[2]=imF[2]->data;
  TexMapScalePow2(2);
  gluBuild2DMipmaps(GL_TEXTURE_2D,3,iW[2],iH[2],GL_RGB,
GL_UNSIGNED_BYTE,iBits[2]);
  imF[3]=auxDIBImageLoad("brick.bmp"); /*Какой-то линолеум
на полу*/
  iW[3]=imF[3]->sizeX;
  iH[3]=imF[3]->sizeY;
  iBits[3]=imF[3]->data;
  TexMapScalePow2(3);
  gluBuild2DMipmaps(GL_TEXTURE_2D,3,iW[3],iH[3],GL_RGB,
GL_UNSIGNED_BYTE,iBits[3]);
}

/*
//Функция непосредственного создания битового образа
//текстуры
void makeImage(void)
{
  int i, j;
  float ti, tj;
  for (i = 0; i < imageWidth; i++) {
    ti = 2.0*3.14159265*i/imageWidth;
    for (j = 0; j < imageHeight; j++) {
      tj = 2.0*3.14159265*j/imageHeight;
      image[0][3*(imageHeight*i+j)] = (GLubyte)
20*(1.0+sin(ti));
      image[0][3*(imageHeight*i+j)+1] = (GLubyte)
40*(1.0+cos(2*tj));
      image[0][3*(imageHeight*i+j)+2] = (GLubyte)
60*(1.0+cos(ti+tj));
      image[1][3*(imageHeight*i+j)] = (GLubyte)
127*(1.0+sin(ti));
      image[1][3*(imageHeight*i+j)+1] = (GLubyte)
27*(1.0+cos(2*tj));
      image[1][3*(imageHeight*i+j)+2] = (GLubyte)
227*(1.0+cos(ti+tj));
      image[2][3*(imageHeight*i+j)] = (GLubyte)
227*(1.0+sin(ti));
      image[2][3*(imageHeight*i+j)+1] = (GLubyte)
127*(1.0+cos(2*tj));
      image[2][3*(imageHeight*i+j)+2] = (GLubyte)
27*(1.0+cos(ti+tj));
```

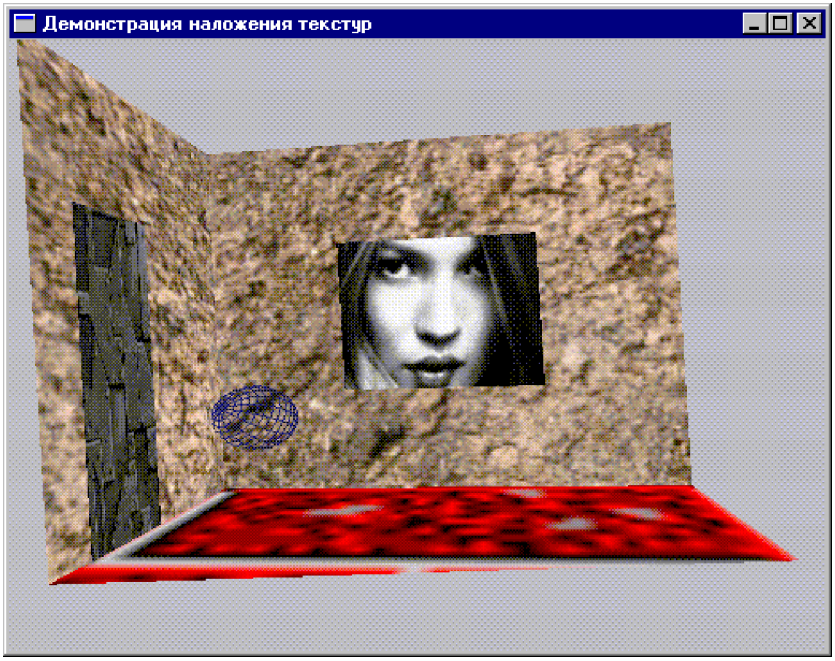
```
    }
  }
}
*/

//Переносы и вращения
void polarView(GLdouble radius, GLdouble twist, GLdouble
latitude,GLdouble longitude)
{
  glTranslated(-1.5, -1.5,-radius);
  glRotated( -twist, 1.0, 0.0, 0.0 );
  glRotated( -latitude, 0.0, 1.0, 0.0);
  glRotated( longitude, 0.0, 0.0, 1.0);
}

//Функция рисования
void CALLBACK DrawScene(void)
{ //Оптические свойства материалов будут из этого набора
  static GLfloat whiteAmbient[] = {0.3, 0.3, 0.3, 1.0};
  static GLfloat redAmbient[] = {0.3, 0.1, 0.1, 1.0};
  static GLfloat greenAmbient[] = {0.1, 0.3, 0.1, 1.0};
  static GLfloat blueAmbient[] = {0.1, 0.1, 0.3, 1.0};
  static GLfloat whiteDiffuse[] = {1.0, 1.0, 1.0, 1.0};
  static GLfloat redDiffuse[] = {1.0, 0.0, 0.0, 1.0};
  static GLfloat greenDiffuse[] = {0.0, 1.0, 0.0, 1.0};
  static GLfloat blueDiffuse[] = {0.0, 0.0, 1.0, 1.0};
  static GLfloat whiteSpecular[] = {1.0, 1.0, 1.0, 1.0};
  static GLfloat redSpecular[] = {1.0, 0.0, 0.0, 1.0};
  static GLfloat greenSpecular[] = {0.0, 1.0, 0.0, 1.0};
  static GLfloat blueSpecular[] = {0.0, 0.0, 1.0, 1.0};
  //Позиция источника света
  static GLfloat lightPosition0[] = {1.0, 1.0, 1.0, 1.0};
  static double al;/*Угол вращения тел в координатных
отсеках*/
  al+=2.;
  glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );
  //Вращаем всю систему
  glPushMatrix();
  latitude = 6.5;
  longitude = 2.5;
  polarView( radius, 0, latitude, longitude );
  //Рисуем и вращаем шар
  glPushMatrix();
  glTranslatef(1.0,1.0,1.9);
  glRotatef(al,1.0,0.5,0.2);
  glRotatef(al,.0,1,0);
  glPushAttrib(GL_LIGHTING_BIT);
  glMaterialfv(GL_BACK, GL_AMBIENT, blueAmbient);
```

```
  glMaterialfv(GL_BACK, GL_DIFFUSE, blueDiffuse);
  glMaterialfv(GL_FRONT, GL_AMBIENT, blueAmbient);
  glMaterialfv(GL_FRONT, GL_DIFFUSE, blueDiffuse);
  glMaterialfv(GL_FRONT, GL_SPECULAR, redSpecular);
  glMaterialf(GL_FRONT, GL_SHININESS, 100.0);
  auxWireSphere(0.2);
  glPopAttrib(); glPopMatrix();
  //glEvalMesh2(GL_FILL, 0, 20, 0, 20);
  /*Рисуем координатные плоскости, привязывая к каждой
свою текстуру*/
  glEnable(GL_TEXTURE_2D);
  for (int i = 0; i < NUM_TEXTURES; i++) texobj[i] = i+1;
  glBindTexture(GL_TEXTURE_2D,texobj[1]);
  glBegin(GL_QUADS);
  glTexCoord2f(1.0f, 1.0f); glVertex3f(3.5f, 3.5f, 0.0f);
  glTexCoord2f(0.0f, 1.0f); glVertex3f(0.0f, 3.5f, 0.0f);
  glTexCoord2f(0.0f, 0.0f); glVertex3f(0.0f, 0.0f, 0.0f);
  glTexCoord2f(1.0f, .0f); glVertex3f( 3.5f, 0.0f, 0.0f);
  glEnd();
  glBindTexture(GL_TEXTURE_2D,texobj[0]);
  glBegin(GL_QUADS);
  glTexCoord2f(1.0f, 1.0f); glVertex3f(2.5f,2.5f,0.01f);
  glTexCoord2f(.0f, 1.0f); glVertex3f(1.f, 2.5f, 0.01f);
  glTexCoord2f(.0f, .0f); glVertex3f(1.f, 1.f, 0.01f);
  glTexCoord2f(1.0f, 0.0f); glVertex3f(2.5f, 1.f, 0.01f);
  glEnd();
  glBindTexture(GL_TEXTURE_2D,texobj[3]);
  glBegin(GL_QUADS);
  glTexCoord2f(0.0f, 0.0f); glVertex3f( 3.5f, .0f, 3.5f);
  glTexCoord2f(0.0f, 1.0f); glVertex3f( 3.5f, .0f, 0.0f);
  glTexCoord2f(1.0f, 1.0f); glVertex3f(0.f, .0f, 0.0f);
  glTexCoord2f(1.0f, 0.0f); glVertex3f(0.0f, .0f, 3.5f);
  glEnd();
  glBindTexture(GL_TEXTURE_2D,texobj[1]);
  glBegin(GL_QUADS);
  glTexCoord2f(0.0f, 0.0f); glVertex3f(.0f, 3.5f, 3.5f);
  glTexCoord2f(0.0f, 1.0f); glVertex3f(.0f, 3.5f, 0.0f);
  glTexCoord2f(1.0f, 1.0f); glVertex3f(.0f, 0.0f, 0.0f);
  glTexCoord2f(1.0f, 0.0f); glVertex3f(.0f, 0.0f, 3.5f);
  glEnd();
  glBindTexture(GL_TEXTURE_2D,texobj[2]);
  glBegin(GL_QUADS);
  glTexCoord2f(0.0f, 0.0f); glVertex3f(.01f, 2.5f, 1.7f);
  glTexCoord2f(0.0f, 1.0f); glVertex3f(.01f, 2.5f, 1.0f);
  glTexCoord2f(1.0f, 1.0f); glVertex3f(.01f, 0.0f, 1.0f);
  glTexCoord2f(1.0f, 0.0f); glVertex3f(.01f, 0.0f, 1.7f);
  glEnd();
  glDisable(GL_TEXTURE_2D);
```

```
  /* glBegin(GL_QUADS);
  glColor3f(1.0f, 0.0f, 0.0f);
  glVertex3f(-1.0f, 1.0f, -1.0f);
  glVertex3f(-1.0f, 1.0f, 1.0f);
  glVertex3f( 1.0f, 1.0f, 1.0f);
  glVertex3f( 1.0f, 1.0f, -1.0f);
  glColor3f(0.0f, 1.0f, 0.0f);
  glVertex3f( 1.0f, -1.0f, -1.0f);
  glVertex3f( 1.0f, -1.0f, 1.0f);
  glVertex3f(-1.0f, -1.0f, 1.0f);
  glVertex3f(-1.0f, -1.0f, -1.0f);
  glColor3f(0.0f, 0.0f, 1.0f);
  glVertex3f(-1.0f, -1.0f, -1.0f);
  glVertex3f(-1.0f, 1.0f, -1.0f);
  glVertex3f(-1.0f, 1.0f, 1.0f);
  glVertex3f(-1.0f, -1.0f, 1.0f);
  glColor3f(1.0f,0.0f,0.0f); glVertex3f(1.0f,-1.0f,1.0f);
  glColor3f(0.0f,1.0f,0.0f); glVertex3f(1.0f,1.0f,1.0f);
  glColor3f(0.0f,0.0f,1.0f); glVertex3f(1.0f,1.0f,-1.0f);
  glColor3f(1.0f,0.0f,1.0f);glVertex3f(1.0f,-1.0f,-1.0f);
  glEnd();
  */
  glPopMatrix();
  glFinish();
  SwapBuffers(wglGetCurrentDC());
}

void Init(void)
{
  GLint i,j;
  GLfloat u,v,r;
  GLfloat maxObjectSize, aspect;
  GLdouble near_plane, far_plane;
  GLsizei w,h;
  GLfloat ambientProperties[] = {0.7, 0.7, 0.7, 1.0};
  GLfloat diffuseProperties[] = {0.8, 0.8, 0.8, 1.0};
  GLfloat specularProperties[] = {1.0, 1.0, 1.0, 1.0};

/*
  UMAX=VMAX=1.2;
  UMIN=VMIN=-1.2;
  DU=(UMAX-UMIN)/(N-1);
  DV=(VMAX-VMIN)/(N-1);
  for(i=0,u=UMIN;i<N;i++,u+=DU)
    for(j=0,v=VMIN;j<N;j++,v+=DV)
    {
      ctrlpoints[j][i][0]=u;
      ctrlpoints[j][i][1]=v;
```

```
      r=u*u+v*v;
      ctrlpoints[j][i][2]=0.0;//*cos(r)/(r+1);
    }
*/
  w = 1024.0;
  h = 768.0;
  auxInitPosition( w/4, h/4, w/2, h/2);
  auxInitDisplayMode( AUX_RGB | AUX_DEPTH | AUX_DOUBLE );
  auxInitWindow( "Демонстрация наложения текстур" );
  auxIdleFunc( DrawScene );
  auxReshapeFunc(Reshape);
  glClearColor( 0.0, 0.0, 0.0, 1.0 );
  glClearDepth( 1.0 );
  glEnable(GL_DEPTH_TEST);
  glEnable(GL_LIGHTING);
  glLightfv( GL_LIGHT0, GL_AMBIENT, ambientProperties);
  glLightfv( GL_LIGHT0, GL_DIFFUSE, diffuseProperties);
  glLightfv( GL_LIGHT0, GL_SPECULAR, specularProperties);
  glLightModelf(GL_LIGHT_MODEL_TWO_SIDE, 1.0);
  glEnable( GL_LIGHT0 );
  glMatrixMode( GL_PROJECTION );
  aspect = (GLfloat) w/h;
  gluPerspective(65.0, aspect, 3.0, 7.0 );
  glMatrixMode( GL_MODELVIEW );
  near_plane = 2.0;
  far_plane = 7.0;
  maxObjectSize = 6.0;
  radius = near_plane + maxObjectSize/2.0;
  latitude = 0.0;
  longitude = 0.0;
  //Цвет фона
  glClearColor (0.7, 0.7, 0.7, 1.0);
  makeImageF();//Создаем файловые текстуры
  for (i = 0; i < NUM_TEXTURES; i++) texobj[i] = i+1;
  for (i = 0; i < NUM_TEXTURES; i++)
  {
    glBindTexture(GL_TEXTURE_2D, texobj[i]);
    glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE,
GL_DECAL);
    glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S,
GL_REPEAT);
    glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T,
GL_REPEAT);
    glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER,
GL_LINEAR);
    glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
GL_LINEAR);
```

```
    glTexImage2D(GL_TEXTURE_2D, 0, 3,iW[i],iH[i]/*
imageWidth, imageHeight*/,0,GL_RGB, GL_UNSIGNED_BYTE,
iBits[i]/*&image[i][0]*/);
  }
  glHint(GL_PERSPECTIVE_CORRECTION_HINT, GL_NICEST);
  /*glTexImage2D(GL_TEXTURE_2D, 0, 3, imageWidth,
imageHeight, 0,GL_RGB, GL_UNSIGNED_BYTE, image[1]);*/
  /*glTexImage2D(GL_TEXTURE_2D, 0, 3, imageWidth,
imageHeight, 0,GL_RGB, GL_UNSIGNED_BYTE, image[2]);*/
  glEnable(GL_TEXTURE_2D);
  glEnable (GL_DEPTH_TEST);
  /*glMap2f(GL_MAP2_TEXTURE_COORD_2, 0, 1, 2, 2,0, 1, 4,
2, &texpts[0][0][0]);*/
  glEnable(GL_MAP2_TEXTURE_COORD_2);
  /*
  glMap2f(GL_MAP2_VERTEX_3, 0, 1, 3, 4,0, 1, 3*N, 4,
&ctrlpoints[0][0][0]);
  glEnable(GL_MAP2_VERTEX_3);
  glEnable(GL_AUTO_NORMAL);
  glEnable(GL_NORMALIZE);
  glMapGrid2f(20, 0.0, 1.0, 20, 0.0, 1.0);
*/
  auxMainLoop(DrawScene);
}

void CALLBACK Reshape(GLsizei w, GLsizei h)
{
  GLfloat aspect=w/h;
  glViewport( 0, 0, w,h);
  glMatrixMode( GL_PROJECTION );
  glLoadIdentity();
  gluPerspective(65.0, aspect, 3.0, 7.0 );
  glMatrixMode( GL_MODELVIEW );
  glLoadIdentity();
}
int main(int argc, char** argv){ Init(); return(0);}
```


Демонстрация наложения текстур

7. Кривые и поверхности

7.1. Методы интерполяции и сглаживания дискретных зависимостей

Достаточно типичной является следующая задача: по заданному массиву точек на плоскости (2D) или в пространстве (3D) построить кривую либо проходящую через все эти точки (задача интерполяции), либо проходящую вблизи от этих точек (задача сглаживания). Совершенно естественно возникают вопросы: 1) в каком классе кривых искать решение поставленной задачи? и 2) как искать?

Обратимся для определенности к задаче интерполяции и начнем рассмотрение с обсуждения правил выбора класса кривых. Ясно, что допустимый класс кривых должен быть таким, чтобы решение задачи было единственным (это обстоятельство сильно помогает в преодолении многих трудностей поиска). Кроме того, желательно, чтобы построенная кривая изменялась плавно.

Пусть на плоскости задан набор точек (X_i, Y_i), i=0, 1, …, m, таких, что $X_0<X_i<...<X_{m-1}<X_m$. То обстоятельство, что точки заданного набора занумерованы в порядке возрастания их абсцисс, позволяет искать кривую в классе графиков функций. Мы сможем описать основные проблемы сглаживания этого дискретного набора, ограничившись случаем многочленов.

Как известно из курса математического анализа, существует интерполяционный многочлен Лагранжа, график которого проходит через все заданные точки.

Это обстоятельство и простота описания (заметим, что многочлен однозначно определяется набором своих коэффициентов; в данном случае их число совпадает с количеством точек в заданном наборе) являются несомненными достоинствами построенного интерполяционного многочлена (разумеется, есть и другие).

Однако нам полезно остановиться и на некоторых недостатках предложенного подхода.

1. Степень многочлена Лагранжа на единицу меньше числа заданных точек. Поэтому, чем больше точек задано, тем выше степень такого многочлена. И хотя график интерполяционного многочлена Лагранжа всегда будет проходить через все точки массива,

его уклонение от ожидаемого может оказаться довольно значительным.

2. Изменение одной точки (ситуация, довольно часто встречаемая на практике) требует полного пересчета коэффициентов интерполяционного многочлена и к тому же может существенно повлиять на вид задаваемой им кривой.

Приближающую кривую можно построить и совсем просто: если последовательно соединить точки заданного набора прямолинейными отрезками, то в результате получится ломаная. При такой, кусочно-линейной, интерполяции требуется найти всего $2m$ чисел (каждый прямолинейный отрезок определяется ровно двумя коэффициентами), но, к сожалению, построенная таким образом аппроксимирующая кусочно-линейная функция не обладает нужной гладкостью: уже первая производная этой функции терпит разрывы в узлах интерполяции.

Рассмотрев эти две крайние ситуации, попробуем найти класс функций, которые в основном сохранили бы перечисленные выше достоинства обоих подходов и одновременно были бы в известной степени свободны от их недостатков.

Для этого поступим так: будем использовать многочлены (как и в первом случае) и строить их последовательно, звено за звеном (как во втором случае). В результате получится так называемый полиномиальный многозвенник. При подобном подходе важно правильно выбрать степени привлекаемых многочленов, а для плавного изменения результирующей кривой необходимо еще тщательно подобрать коэффициенты многочленов (из условий гладкого сопряжения соседних звеньев).

То, что получится в результате описанных усилий, называют сплайн-функциями или просто сплайнами.

Историю сплайнов принято отсчитывать от момента появления первой работы Шенберга в 1946 году. Сначала сплайны рассматривались как удобный инструмент в теории и практике приближения функций. Однако довольно скоро область их применения начала быстро расширяться и обнаружилось, что существует очень много сплайнов самых разных типов. Сплайны стали активно использоваться в численных методах, в системах автоматического проектирования и автоматизации научных исследований, во

многих других областях человеческой деятельности и, конечно, в компьютерной графике.

Сам термин «сплайн» происходит от английского spline. Именно так называется гибкая полоска стали, при помощи которой чертежники проводили через заданные точки плавные кривые. В былые времена подобный способ построения плавных обводов различных тел, таких, как, например, корпус корабля, кузов автомобиля, а потом фюзеляж или крыло самолета, был довольно широко распространен в практике машиностроения. В результате форма тела задавалась при помощи набора очень точно изготовленных сечений – плазов. Появление компьютеров позволило перейти от этого, плазово-шаблонного, метода к более эффективному способу задания поверхности обтекаемого тела. В основе этого подхода к описанию поверхностей лежит использование сравнительно несложных формул, позволяющих восстанавливать облик изделия с необходимой точностью.

Ясно, что для большинства тел, встречающихся на практике, вряд ли возможно отыскание простых универсальных формул, которые описывали бы соответствующего поверхность глобально, то есть, как принято говорить, в целом. Это означает, что при решении задачи построения достаточно произвольной поверхности обойтись небольшим количеством формул, как правило, не удается. Вместе с тем аналитическое описание (описание посредством формул) внешних обводов изделия, то есть задание в трехмерном пространстве двумерной поверхности, должно быть достаточно экономным. Это особенно важно, когда речь идет об обработке изделий на станках с числовым программным управлением. Обычно поступают следующим образом: задают координаты сравнительно небольшого числа опорных точек, лежащих на искомой поверхности, и через эти точки проводят плавные поверхности. Именно так поступает конструктор при проектировании кузова автомобиля (ясно, что на этой стадии процесс проектирования сложного объекта содержит явную неформальную составляющую). На следующем шаге конструктор должен получить аналитическое представление для придуманных кривых или поверхностей. Вот для таких задач и используются сплайны.

Средства компьютерной графики, особенно визуализация, существенно помогают при проектировании, показывая конструк-

тору, что может получиться в результате, давая ему многовариантную возможность сравнить это с тем, что сложилось у него в голове. Мы не ставим перед собой и читателем задачи рассказать обо всех сплайнах, в частности потому, что это отдельная большая тема, требующая и большего внимания, и большего объема. Во вводном курсе нам кажется более уместным показать в сравнении некоторые из преимуществ использования сплайнов в задачах геометрического моделирования при проектировании кривых и поверхностей. Такое представление полезно начинающему пользователю для его ориентации в стремительно расширяющемся мире сплайнов. При этом мы ограничимся лишь сплайнами, в построении которых используются кубические (в случае одномерных сплайнов – сплайновых кривых) и бикубические (в случае двумерных сплайнов – сплайновых поверхностей) многочлены. В компьютерной графике подобные сплайны применяются наиболее часто.

Для того, чтобы понять, какое отношение имеют сплайн-функции к чертежным сплайнам, возьмем гибкую стальную линейку, поставим ее на ребро и, закрепив один из концов в заданной точке, поместим ее между опорами, которые располагаются в плоскости Oxy в точках (X_i, Y_i), $i=0, 1, \ldots, m$, где $X_0<X_1<\ldots<X_m$.

Интересно отметить, что функция $y=S(x)$, описывающая профиль линейки, обладает следующими интересными свойствами:

- с довольно большой точностью часть графика этой функции, заключенную между любыми двумя соседними опорами, можно считать многочленом третьей степени;
- на всем промежутке $[X_0, X_m]$ функция $y=S(x)$ дважды непрерывно дифференцируема.

Построенная функция $S(x)$ относится к так называемым интерполяционным кубическим сплайнам. Этот класс в полной мере удовлетворяет высказанным выше требованиям и обладает еще целым рядом замечательных свойств.

Перейдем, однако, к точным формулировкам. **Интерполяционным кубическим сплайном** называется функция $S(x)$, обладающая следующими свойствами:

1) график этой функции проходит через каждую точку заданного массива: $S(x_i)=y_i$, $i=0, 1, \ldots, m$;

2) на каждом из отрезков $[X_i, X_{i-1}]$, $i=0, 1, \ldots, m-1$, функция является многочленом третьей степени,

3) на всем отрезке задания $[X_0, X_m]$ функция $S(x)$ имеет непрерывную вторую производную.

Так как на каждом из отрезков $[X_i, X_{i+1}]$ сплайн $S(x)$ определяется четырьмя коэффициентами, то для его полного построения на всем отрезке задания необходимо найти $4m$ чисел.

Третье условие будет выполнено, если потребовать непрерывности сплайна во всех внутренних узлах X_i, i=1, …, m–1 (это дает m–1 условий на коэффициенты), а также его первой (m–1 условий) и второй (еще m–1 условий) производных в этих узлах. Вместе с первым условием получаем m–1+m–1+m–1+m+1=4m–2 равенства.

Недостающие два условия для полного определения коэффициентов можно получить, задав, например, значения первых производных на концах отрезка $[X_0, X_m]$ (граничные условия):

$$S'(X_0)=l_0,\ S'(X_m)=l_m.$$

Существуют граничные условия и других типов.

Случай двух переменных.

Более сложная задача построения по заданному набору точек в трехмерном пространстве интерполяционной функции двух переменных решается похожим образом.

Расскажем прежде всего, что такое интерполяционный бикубический сплайн.

Пусть на плоскости задан набор из $(m+1)(n+1)$ точек (X_i, Y_j), i=0, 1, …, m; j=0, 1, …, n, где $X_0<X_1<\ldots<X_m$, $Y_0<Y_1<\ldots<Y_n$. Добавим к каждой паре (X_i, Y_j) третью координату $Z_{ij}(X_i, Y_j, Z_{ij})$.

Тем самым мы получаем массив (X_i, Y_j, Z_{ij}), i=0, 1, …, m; j=0, 1, …, n. Прежде чем строить поверхность, проходящую через все точки заданного массива, определим функцию, графиком которой будет эта поверхность.

Интерполяционным бикубическим сплайном называется функция двух переменных $S(x, y)$, обладающая следующими свойствами:

1) график этой функции проходит через каждую точку заданного массива:

$$S(X_i, Y_j)=Z_{ij},\ i=0, 1, \ldots, m;\ j=0, 1, \ldots, n;$$

2) на каждом частичном прямоугольнике

$$[X_i, X_{i+1}] \times [Y_j, Y_{j+1}],\ i=0, 1, \ldots, m-1;\ j=0, 1, \ldots, n-1,$$

функция представляет собой многочлен третьей степени по каждой из переменных:

$$S(x, y)= \sum_{l,k=0}^{3} a_{lk}^{ij} (x - x_i)^l (y - y_j)^k ;$$

3) на всем прямоугольнике задания $[X_0, X_m]$ x $[Y_0, Y_n]$ функция $S(x, y)$ имеет по каждой переменной непрерывную вторую производную. Для того, чтобы построить по заданному массиву $\{(X_i, Y_j, Z_{ij})\}$ интерполяционный бикубический сплайн, достаточно определить все $16mn$ коэффициентов. Как и в одномерном случае, отыскание коэффициентов сплайн-функции сводится к построению решения системы линейных уравнений, связывающих искомые коэффициенты a_{lk}^{ij}.

Последняя возникает из первого и третьего условий после добавления к ним недостающих соотношений путем задания значений производной искомой функции в граничных узлах прямоугольника $[X_0, X_m]$ x $[Y_0, Y_n]$ (или иных соображений).

Подведем некоторые итоги.

Достоинства предложенного способа несомненны: для решения линейных систем, возникающих в ходе построения сплайн-функций, существует много эффективных методов, к тому же эти системы достаточно просты; графики построенных сплайн-функций проходят через все заданные точки, полностью сохраняя первоначально заданную информацию.

Вместе с тем изменение лишь одной точки (случай на практике довольно типичный) при описанном подходе заставляет пересчитывать заново, как правило, все коэффициенты.

К тому же во многих задачах исходный набор точек задается приближенно и, значит, требование неукоснительного прохождения графика искомой функции через каждую точку этого набора оказывается излишним.

От этих недостатков свободны некоторые из методов сглаживания, к описанию которых мы и переходим. Но прежде всего мы значительно расширим классы, в которых будет вестись поиск соответствующих кривых и поверхностей. Более точно, мы откажемся от требования однозначного проектирования искомой кривой на координатную ось, а поверхности – на координатную плоскость. Такой подход позволяет ослабить и требования к задаваемому массиву. Сказанное требует небольшого геометрического введения. Начнем, как и прежде, с кривых.

Сплайновые кривые.

Нам будет удобно пользоваться параметрическими уравнениями кривой. Напомним необходимые понятия.

Параметрически заданной кривой называется множество у точек $M(x, y, z)$, координаты x, y, z которых определяются соотношениями

$$x=x(t),\ y=y(t),\ z=z(t),$$

$a \le t \le b$, где $x(t), y(t), z(t)$ – функции, непрерывные на отрезке $[a, b]$.

Без ограничения общности можно считать, что $a=0$ и $b=1$; этого всегда можно добиться при помощи замены вида

$$U=\frac{t-a}{b-a}.$$

Полезна векторная форма записи параметрических уравнений $\boldsymbol{r}=\boldsymbol{r}(t)$, $0 \le t \le 1$, где $\boldsymbol{r}(t)=(x(t), y(t), z(t))$.

Параметр t задает ориентацию параметризованной кривой γ (порядок прохождения точек при монотонном возрастании параметра).

Кривая γ называется регулярной кривой, если $\boldsymbol{r}'(t) \ne 0$ в каждой ее точке. Это означает, что в каждой точке кривой существует касательная к ней, и эта касательная меняется непрерывно вслед за перемещающейся вдоль кривой ее текущей точки. Единичный вектор касательной к кривой γ равен

$$\boldsymbol{T}(t)=\boldsymbol{r}'(t)/|\boldsymbol{r}'(t)|.$$

Если дополнительно потребовать, чтобы задающая кривую векторная функция имела вторую производную, то будет определен вектор кривизны кривой, модуль которого характеризует степень ее отклонения от прямой. В частности, если γ – отрезок прямой, то $r=0$.

Замечание.

При дальнейшем изложении мы имеем в виду расположение рассматриваемых объектов в трехмерном пространстве. Практически все сказанное будет верно и для плоского случая (более общего, чем рассмотренный выше). Дело в том, что параметрическое описание плоской кривой не накладывает никаких ограничений на ее расположение относительно координатных осей: кривая не обязана однозначно проектироваться на координатную ось, как это имеет место в случае ее явного задания $y=y(x)$.

В частности, кривая может быть замкнутой, самопересекающейся и так далее. Все последующие построения законны и в этих сложных случаях.

Рассмотрим некоторые подходы к построению сглаживающей кривой. Пусть на плоскости или в пространстве задан упорядоченный набор точек, определяемых векторами $\boldsymbol{V}_0$, $\boldsymbol{V}_1$, ..., $\boldsymbol{V}_m$. Ломаная $V_0V_1\ldots V_m$ называется *контрольной ломаной*, порожденной массивом $V=\{V_0, V_1, \ldots, V_m\}$.

Кривой Безье, определяемой массивом V, называется кривая, определяемая векторным уравнением

$$r(t)=\sum_{i=0}^{m} C_m^i t^i (1-t)^{m-i} V_i \ , \ 0\le t\le 1$$

где $C_m^i = \dfrac{m!}{i!(m-i)!}$ – коэффициенты в разложении бинома Ньютона (число сочетаний из m элементов по i). Кривая Безье обладает замечательными свойствами:

- она является гладкой;
- начинается в точке V_0 и заканчивается в точке V_m , касаясь при этом отрезков V_0V_1 и $V_{m-1}V_m$ контрольной ломаной;
- функциональные коэффициенты $C_m^i t^i (1-t)^{m-1}$ при вершинах V_i, $i=0, 1, \ldots, m$, суть универсальные многочлены (многочлены Бернцггейна); они неотрицательны, и их сумма равна единице:

$$\sum_{i=0}^{m} C_m^i t^i (1-t)^{m-i} = (t+(1-t))^m = 1 .$$

Поэтому кривая Безье целиком лежит в выпуклой оболочке, порождаемой массивом.

При $m=3$ получаем (элементарную) кубическую кривую Безье, определяемую четверкой точек V_0, V_1, V_2, V_3 и описываемую векторным параметрическим уравнением

$$\boldsymbol{r}(t)=(((1-t)V_0+3tV_1)(1-t)+3t^2V_2)(1-t)+t^3V_3,\ 0\le t\le 1,$$

или, в матричной записи,

$$\boldsymbol{r}(t)=\mathrm{VMT},\ 0\le t\le 1,$$

где

$$\boldsymbol{r}(t)=\begin{pmatrix} x(t) \\ y(t) \\ z(t) \end{pmatrix},\ \mathrm{V}=\begin{pmatrix} V_0 & V_1 & V_2 & V_3 \end{pmatrix}=\begin{pmatrix} x_0 & x_1 & x_2 & x_3 \\ y_0 & y_1 & y_2 & y_3 \\ z_0 & z_1 & z_2 & z_3 \end{pmatrix},$$

$$\mathrm{M}=\begin{pmatrix} 1 & -3 & 3 & 1 \\ 0 & 3 & -6 & 3 \\ 0 & 0 & 3 & -3 \\ 0 & 0 & 0 & 1 \end{pmatrix},\ \mathrm{T}=\begin{pmatrix} 1 \\ t \\ t^2 \\ t^3 \end{pmatrix}$$

Матрица M называется базисной матрицей кубической кривой Безье.

Порядок точек в заданном наборе существенно влияет на вид элементарной кривой Безье.

Наряду с отмеченными достоинствами, кривые Безье обладают и определенными недостатками. Основных недостатков у элементарных кривых Безье три:

1) степень функциональных коэффициентов напрямую связана с количеством точек в заданном наборе (на единицу меньше);

2) при добавлении хотя бы одной точки в заданный набор необходимо провести полный пересчет функциональных коэффициентов в параметрическом уравнении кривой;

3) изменение хотя бы одной точки приводит к заметному изменению вида всей кривой.

В практических вычислениях часто оказывается удобным пользоваться кривыми, составленными из элементарных кривых Безье, как правило, кубических.

Важное замечание.

При построении кривой из определенным образом подобранных фрагментов важна не только регулярность самих этих фрагментов, но и выполнение некоторых условий гладкости в точках их состыковки. Только в этом случае составная кривая, получающаяся в результате проведенных построений, будет обладать достаточно хорошими геометрическими характеристиками. Однако при построении составных кривых часто приходится сталкиваться с ситуацией, когда каждый из регулярных фрагментов, участвующих в создании новой кривой, имеет свою собственную параметризацию. Чтобы учесть это обстоятельство, удобно использовать класс так называемых *геометрически непрерывных кривых*.

Составная кривая называется G^1-(геометрически) непрерывной, если вдоль этой кривой единичный вектор ее касательной изменяется непрерывно, и G^2-(геометрически) непрерывной, если вдоль этой кривой изменяется непрерывно, кроме того, и вектор кривизны. Обратимся к рассмотрению составных кривых Безье. Составная кубическая кривая Безье представляет собой объединение элементарных кубических кривых Безье $\gamma_1, \ldots, \gamma_m$ таких, что

$$\boldsymbol{r}_i(1)=\boldsymbol{r}_{i-m}(0),\ i=0, \ldots, m-1,$$

где $\boldsymbol{r}=\boldsymbol{r}_i(t)$, $0\le t\le 1$ – параметрическое уравнение кривой γ_i. Чтобы составная кривая Безье, определяемая набором вершин V_0, V_1, …, V_{m-1}, V_m, была

1) G^1-непрерывной кривой, необходимо, чтобы каждые три точки V_{3i-1}, V_{3i}, V_{3i+1} этого набора лежали на одной прямой;

2) замкнутой G^1-непрерывной кривой, необходимо, кроме того, чтобы совпадали первая и последняя точки, $V_0=V_m$, и три точки V_{m-1} ,$V_m=V_0$,V_1 лежали на одной прямой;

3) G^2-непрерывной кривой, необходимо, чтобы каждые пять точек V_{3i-2}, V_{3i-1}, V_{3i}, V_{3i+1}, V_{3i+2} ($i\ge 1$) заданного набора лежали в одной плоскости.

```
// File Bezier.cpp
#include <math.h>
double Bezier ( double p [], int i, double t )
{
  double s = 1- t;
  double t2 = t * t;
  double t3 = t2 * t;
  return ((p[3*i]*s+3*t*p[3*i+1])*s+
          3*t2*p[3*i+2])*s+t3*p[3*i+3];
}
```

Попытаемся найти другой класс кривых, сохраняющих перечисленные достоинства кривых Безье и лишенных их недостатков.

Так как в векторном уравнении, задающем кривую Безье, векторные составляющие постоянны (это просто вершины массива), то мы уделим основное внимание выбору новых функциональных коэффициентов, стараясь (разумеется, по возможности) сохранить при этом замечательные свойства многочленов Бернштейна, ограничив наши рассмотрения кубическими многочленами. По заданному набору точек V_0, V_1, V_2, V_3 элементарная кубическая B-сплайновая кривая определяется при помощи векторного параметрического уравнения следующего вида:

$$\boldsymbol{r}(t)=\frac{(1-t)^3}{6}V_0+\frac{3t^3-6t^2+4}{6}V_1+\frac{-3t^3+3t^2+3t+1}{6}V_2+\frac{t^3}{6}V_3,$$

или, в матричной форме,

$$\boldsymbol{r}(t)=\mathrm{VMT},\ 0\le t\le 1,$$

где

$$\boldsymbol{r}(t)=\begin{pmatrix}x(t)\\ y(t)\\ z(t)\end{pmatrix},\ \mathrm{V}=\begin{pmatrix}V_0 & V_1 & V_2 & V_3\end{pmatrix}=\begin{pmatrix}x_0 & x_1 & x_2 & x_3\\ y_0 & y_1 & y_2 & y_3\\ z_0 & z_1 & z_2 & z_3\end{pmatrix},$$

$$\mathrm{M}=\begin{pmatrix}1 & -3 & 3 & 1\\ 4 & 0 & -6 & 3\\ 1 & 3 & 3 & -3\\ 0 & 0 & 0 & 1\end{pmatrix},\ \mathrm{T}=\begin{pmatrix}1\\ 1\\ t^2\\ t^3\end{pmatrix}$$

Матрица M называется *базисной матрицей* B-сплайновой кривой. Функциональные коэффициенты в уравнении, определяющем элементарную B-сплайновую кубическую кривую, неотрицательны, в сумме составляют единицу, универсальны (не зависят от конкретного вида точек в заданной четверке).

Это означает, что рассматриваемый элементарный фрагмент лежит внутри выпуклой оболочки заданных вершин – четырехугольника (в плоском случае) или тетраэдра (в пространственном случае).

Составная кубическая B-сплайновая кривая, задаваемая параметрическим уравнением

$$\boldsymbol{r}=\boldsymbol{r}(t),\ 0\le t\le m-2,$$

и определяемая набором точек V_0, V_1, …, V_m ($m\ge 3$), представляет собой объединение $m-2$ элементарных кубических B-сплайновых кривых, γ_1, …, γ_m, описываемых уравнениями вида

$$\boldsymbol{r}=\boldsymbol{r}_i(t)=(V_{i-1}\ V_i\ V_{i+1}\ V_{i+2})\mathrm{M}\begin{pmatrix}1\\ t-i+1\\ (t-i+1)^2\\ (t-i+1)^3\end{pmatrix}$$

$$i-1\le t\le i,\ i=1,\ \ldots,\ m-2.$$

Замечание

Область изменения параметра t и расположение на ней точек, соответствующих стыковочным узлам, могут быть совершенно произвольными. Наиболее простой является равномерная параметризация с равноотстоящими целочисленными узлами.

Указанная выше составная B-сплайновая кубическая кривая является C^2-гладкой кривой и лежит в объединении m–2 выпуклых оболочек, порожденных последовательными четверками точек заданного набора. Добавляя в исходный набор дополнительные точки, можно получить составную B-сплайновую кубическую кривую с разными свойствами. Например, при добавлении к заданному набору двух точек

$$V_{-1}=(V_0-V_1)+V_0,\ V_{m+1}=(V_m-V_{m-1})+V_m$$

и соответствующем расширении отрезка изменения параметра до $[0, m]$ получим составную B-сплайновую кубическую кривую, которая будет начинаться в точке V_0, касаясь отрезка V_0V_1, и заканчиваться в точке V_m, касаясь отрезка $V_{m-1}V_m$.

А для того, чтобы по заданному массиву построить C^2-гладкую замкнутую B-сплайновую кривую, достаточно выбрать три дополнительные точки

$$V_{m+1}=V_0,\ V_{m+2}=V_1,\ V_{m+3}=V_2$$

и рассмотреть набор

$$V_0, V_1, V_2, \ldots, V_m, V_{m+1}, V_{m+2}, V_{m+3}.$$

```
// File Bspline.cpp
#include <math.h>
double BSpline ( double p [], int i, double t )
{
  double s = 1.0- t; double t2 = t-t; double t3 = t2 * t;
  return (s*s*s*p[i] + (3*t3 - 6*t2 + 4) * p[i+1] +
         (-3*t3+3*t2+3*t+1)*p[i+2]+t3*p[i+3])/6.0;
}
```

Перейдем к случаю, когда узлы расположены на отрезке изменения параметра t неравномерно. Заменим в векторном уравнении (2) многочлены Бернштейна на B-сплайны (базовые (base) сплайны), введя новые функциональные коэффициенты при помощи рекуррентных формул.

Пусть $0=t_0<t_1<\ldots<t_m=1$ – разбиение отрезка [0, 1].

Положим $N_{i,1}(t)=1$, $t\in[t_i, t_{i+1}]$, $N_{i,1}(t)=0$, $t\in[t_i, t_{i+1}]$ и далее

$$N_{i,q}(t)=\frac{t-t_i}{t_{i+q-1}}N_{i,q-1}(t)+\frac{t_{i+q}-t}{t_{i+q}-t_{i+1}}N_{i+1,q-1}(t).$$

Заметим, что с увеличением индекса q степень многочленов, определяющих вводимые функции $N_{iq}(t)$, растет: для функций на отрезке $[t_i, t_{i+q}]$ она равна $q-1$.

Отметим еще некоторые очевидные свойства этих функций:

- $Ni,q(t)>0$ на интервале (t_i, t_{i+q});
- $Ni,q(t)=0$ вне интервала (t_i, t_{i+q});
- на всей области задания функция $Ni,q(t)$, $q\geq 3$, имеет непрерывные производные до порядка $q-2$ включительно.

Кроме того для введенных функций сохраняется равенство

$$\sum_{i=0}^{m} N_{i,q}(t) = 1 .$$

Это означает, что кривая, заданная векторным уравнением

$$\boldsymbol{r}(t)=\sum_{i=0}^{m} N_{i,q}(t)V_i$$

всегда принадлежит выпуклой оболочке вершин заданного массива.

Замечание

На самом деле кривая лежит в объединении выпуклых оболочек, порожденных последовательными наборами из $q+1$ точек заданного массива. Построенная кривая обладает важным локальным свойством: изменение одной вершины в массиве (или добавление новой вершины к имеющимся) уже не ведет, как прежде, к полному изменению всей кривой.

В силу третьего свойства сохраняется достаточная гладкость кривой: если взять $q\geq 4$, то все функциональные коэффициенты будут иметь непрерывные вторые производные. Для практических задач большей гладкости, как правило, не требуется. Поэтому обычно ограничиваются рассмотрением случая, когда $q=4$.

Замечание

Для построения кубического B-сплайна $N_{i,4}(t)$ требуется 5 узлов разбиения отрезка [0, 1]. Поэтому если узлов не хватает, то их набор определенным образом расширяют. Дополнительно введенные отрезки имеют нулевую длину, и первоначальные первый $t_0=0$ и последний $t_m=1$ узлы становятся кратными.

Замечание

Выбор узлов параметризации может быть совершенно произвольным. Однако часто удобной оказывается параметризация, в

которой промежуток изменения параметра t и узлы t_i определяются длинами соответствующих хорд:

$t_0=0$;

$t_1=|V_2V_0|$,

$t_i=t_{i-1}+|V_{i+1}V_{i-2}|$, $i=2, \ldots, m-3$,

$t_{m-2}=t_{m-3}+|V_mV_{m-2}|$.

Обратимся к следующей достаточно типичной ситуации: по заданному набору точек мы построили B-сплайновую кривую, вывели полученный результат на экран и, внимательно изучив то, что предстало перед глазами, ощутили необходимость подправить кривую в одном или нескольких местах, не изменяя исходного набора точек.

Наиболее подходящим инструментом для подобной процедуры являются числовые или функциональные параметры, заранее введенные в уравнения кривых.

Такую возможность предоставляют некоторые обобщения кубических B-сплайнов, а именно рациональные кубические B-сплайны и бета-сплайны, к описанию которых мы и обратимся.

Рациональные кубические B-сплайны

По заданному набору V_0, V_1, V_2, V_3 рациональная кубическая B-сплайновая кривая определяется уравнением следующего вида:

$$\boldsymbol{r}(t)=\frac{\sum_{i=0}^{3} w_i n_i(t) V_i}{\sum_{i=9}^{3} w_i n_i(t)},\ 0\le t\le 1,$$

где

$$n_0(t)=\frac{(1-t)^3}{6},\ n_1(t)=\frac{3t^3-6t^2+4}{6},$$

$$n_2(t)=\frac{-3t^3+3t^2+3t+1}{6},\ n_3(t)=\frac{t^3}{6}$$

а величины w_i, называемые весами (или параметрами формы), – неотрицательные числа, сумма которых положительна.

Замечания:

1. В случае, если все веса равны между собой, приведенное уравнение описывает элементарную кубическую B-сплайновую кривую.

2. Построение составной рациональной B-сплайновой кубической кривой проводится по той же схеме, что и в полиномиальном случае.

3. В последнее время значительный интерес пользователей вызывает класс сплайнов, известный под названием NURBS – nonuniform rational B-splines – рациональных B-сплайнов, задаваемых на неравномерной сетке.

Бета-сплайны

Применение составных бета-сплайновых кривых основывается на важном свойстве геометрической непрерывности.

Как отмечалось выше, в построении составной регулярной кривой важную роль играют условия сопряжения в точках контакта слагающих ее отрезков регулярных кривых.

Пусть γ_1 и γ_2 – регулярные кривые, заданные параметрическими уравнениями

$$\boldsymbol{r}=\boldsymbol{r}_1(t),\ 0\le t\le 1;\ \boldsymbol{r}=\boldsymbol{r}_2(t),\ 0\le t\le 1,$$

соответственно и имеющие общую точку $\boldsymbol{r}_1(1)=r_2(0)$.

Для того, чтобы кривая γ, составленная из кривых γ_1 и γ_2, была регулярной, потребуем совпадения в общей точке единичных касательных векторов

$$\frac{r'(1)}{|r'(1)|}=\frac{r'(0)}{|r'(0)|}$$

и векторов кривизны

$$\frac{\left[r_1'(1)\times r_1''(1)\times r_1'(1)\right]}{\left|r_1'(1)^4\right|}=\frac{\left[r_2'(1)\times r_2''(1)\times r_2'(1)\right]}{\left|r_2'(1)^4\right|}$$

сопрягаемых кривых γ_1 и γ_2.

Нетрудно проверить, что если радиусы-векторы кривых γ_1 и γ_2 связаны условиями геометрической непрерывности

$$r_2(0)=r_1(1),$$
$$r_2'(0)=\beta_1 r_1'(1),$$
$$r_2''(0)=\beta_1^2 r_1''(1)+\beta_2 r_1'(1),$$

где $\beta_1>0$, $\beta_2\ge 0$ – числовые параметры, то каждое из условий будет выполнено.

Рассмотрим набор из $m+1$ точек V_0, V_1, ..., V_m, заданных своими радиусами-векторами. Будем искать сглаживающую составную регулярную кривую γ при помощи частичных кривых γ_i, описываемых уравнениями вида

$$\boldsymbol{r}_i(t)=\sum_{j=-2}^{1} b_j(t)V_{ij}\ ,\ 0\le t\le 1,$$

где

$$b_j(t)=\sum_{k=0}^{3} c_{kj}(\beta_1,\beta_{12})t^k\ ,\ j=-2,\ -1,\ 0,\ 1\ -$$

не зависящие от i весовые функциональные коэффициенты.

Для того, чтобы найти эти весовые коэффициенты, потребуем, чтобы векторы $\boldsymbol{r}_i(1)$ и $\boldsymbol{r}_{i+1}(t)$ в точке сопряжения удовлетворяли условиям геометрической непрерывности. Эти условия можно записать так:

$$\sum_{j=-2}^{1} b_j(0)V_{i+1+j}=\sum_{j=-2}^{1} b_j(1)V_{1+j},$$

$$\sum_{j=-2}^{1} b'_j(0)V_{i+1+j}=\beta_1\sum_{j=-2}^{1} b'_j(1)V_{1+j}$$

$$\sum_{j=-2}^{1} b''_j(0)V_{i+1+j}=\beta_1^2\sum_{j=-2}^{1} b''_j(1)V_{1+j}+\beta_2\sum_{j=-2}^{1} b'_j(1)V_{1+j}$$

Полученные соотношения позволяют найти все функциональные коэффициенты

$$b_j(t),\ j=-2,\ -1,\ 0,\ 1.$$

Расписав, например, первое из равенств подробнее:

$$b_{-2}(0)V_{i-1}+b_{-1}(0)V_i+b_0(0)V_{i+1}+b_1(0)V_{i+2}=$$
$$=b_{-2}(1)V_{i-2}+b_{-1}(1)V_{i-1}+b_0(1)V_i+b_1(1)V_{i+1}$$

и приравняв коэффициенты при одинаковых векторах, получим:

$$0=b_{-2}(1),\ b_{-2}(0)=b_{-1}(1),\ b_{-1}(0)=b_0(1),\ b_0(0)=b_1(1),\ b_1(0)=0.$$

Подобным же образом из двух других векторных равенств получаются соотношения, связывающие значения в точках 0 и 1 первых и вторых производных функциональных коэффициентов.

В итоге получаем линейную систему для искомых чисел c_{kj}, определитель которой

$$\delta=2\beta_1^3+4\beta_1^2+4\beta_1+\beta_2+2>0.$$

Разрешая линейную алгебраическую систему, найдем величины c_{kj} и затем подставим полученные выражения. Выражения для функциональных коэффициентов

$$b_{-2}(t)=\frac{2\beta_1^3}{\delta}(1-t)^3,$$

$$b_{-1}(t)=(1/\delta)[2\beta_1^3 t(t^2-3t+3)+2\beta_1^2(t^3-3t^2+2)]+[2\beta_1(t^3-3t^2+2)+$$

$$\beta_2(2t^3-3t^2+1)],$$ г

$$b_0(t)=(1/\delta)[2\beta_1^2 t^2(-t+3)+2\beta_1 t(-t^2+3)+\beta_2 t^2(-2t+3)+2(-t^3+1)],$$

$$b_1(t)=\frac{2t^3}{\delta}$$

одятся для всей конструкции. Подставляя, получаем значения векторных функций

$$\boldsymbol{r}_2(t),\ \ldots,\ \boldsymbol{r}_{m-1}(t).$$

Заметим, что кривая, определяемая векторной функцией $\boldsymbol{r}_1(t)$ и, значит, вершинами V_{i-1}, V_i, V_{i+1}, V_{i+2}, лежит в их выпуклой оболочке. Запишем уравнение элементарной бета-сплайновой кривой, порожденной набором точек V_{i-1}, V_i, V_{i+1}, V_{i+2} в матричном виде. Имеем:

$$\boldsymbol{r}(t)=\mathrm{VMT},\ 0\le t\le 1,$$

где

$$\boldsymbol{r}(t)=\{x(t)\ y(t)\ z(t)\},\ \mathrm{T}=\{1\ 1\ t^2\ t^3\},$$

$$\mathrm{V}=\{V_{i-1}\ V_i\ V_{i+1}\ V_{i+2}\}=\begin{pmatrix} x_0 & x_1 & x_2 & x_3 \\ y_0 & y_1 & y_2 & y_3 \\ z_0 & z_1 & z_2 & z_3 \end{pmatrix},$$

$$\mathrm{M}=\frac{1}{\delta}\begin{pmatrix} 2\alpha & -6\alpha & 6\alpha & -2\alpha \\ 4(\beta_1^2+\beta_1)+\beta_2 & 6(\alpha-\beta_1) & -3(2\alpha+\mu) & 2(\alpha+\upsilon) \\ 2 & 6\beta_1 & 3\mu & -2(\upsilon+1) \\ 0 & 0 & 0 & 2 \end{pmatrix}.$$

Здесь $\alpha=\beta_1{}^3$, $\mu=2\beta_1{}^2+\beta_2$, $\upsilon=\beta_1{}^2+\beta_1+\beta_2$.

Матрица M называется *базисной матрицей бета-сплайновой кривой*.

Замечания:

Числовые параметры β_1 и β_2 называются *параметрами формы* бета-сплайновой кривой, причем первый из них называют *параметром скоса*, а второй – *параметром напряжения*.

При $\beta_1=1$, $\beta_2=0$ получается кубическая B-сплайновая кривая.

Подбором дополнительных вершин можно влиять на поведение составной бета-сплайновой кривой вблизи ее концов. Например, для того, чтобы составная кривая γ проходила через вершины V_0 и V_m, касаясь отрезков V_0V_1 и $V_{m-1}V_m$ контрольной ломаной, следует добавить к полученному набору векторных функций еще 4:

$$\boldsymbol{r}_0(t)=(1-\frac{2t^3}{\delta})V_0+\frac{2t^3}{\delta}V_1,$$

$$\boldsymbol{r}_1(t)=[b_{-2}(t)+b_{-1}(t)]V_0+b_0(t)V_i+b_1(t)V_2,$$

$$\boldsymbol{r}_m(t)=b_{-2}(t)V_{m-2}+b_{-1}(t)V_{m-1}+[b_0(t)+b_1(t)]V_{\mathrm{m}},$$

$$\boldsymbol{r}_{m+1}(t)=\frac{2\beta_1^3}{\delta}(1-t)^3V_{m-1}+[1-\frac{2\beta_1^3}{\delta}(1-t)^3].$$

Итак, искомая составная кривая γ построена. Вот ее уравнения:

$$\boldsymbol{r}_0(t),\ \boldsymbol{r}_1(t),\ \boldsymbol{r}_2(t),\ \ldots,\ \boldsymbol{r}_{m-1}(t),\ \boldsymbol{r}_m(t),\ \boldsymbol{r}_{m+1}(t),\ 0\le t\le 1.$$

```
// Beta.cpp
#include <math.h>
double BetaSpline (double betal, double beta2, double p
[], int i, double t)
{
  double s = 1.0-t; double t2 = t*t; double t3 = t2*t;
  double bl2 = beta1*beta1; double b13 = b12*beta1;
  double delta = 2.0*b13+4.0*b12+4.0*beta1+beta2+2.0;
  double d = 1.0 / delta; double b0 = 2*b13*d*s*s*s;
  double b3 = 2*t3*d;double b1 = d*(2*b13*t(t2-3*t+3)+
              2*b12*(t3-3*t2+2)+2*beta1*(t3-3*t+2)+beta2*
              ( 2* t3 -3*t2+1));
  double b2 = d*(2*b12*t2*(-t+3)+2*beta1*t*(-t2+3)+
              beta2*t2*(-2*t+3)+2*(-t3+1));
  return b0*p [i] + b1*p [i+1] + b2*p [i+2] + b3*p [1+3];
}
```

Перейдем теперь к двумерному случаю – сплайновым поверхностям.

Сплайновые поверхности

Напомним некоторые понятия.

Регулярной поверхностью называется множество точек $M(x, y, z)$ пространства, координаты x, y, z которых определяются из соотношений

$$x=x(u, v),\ y=y(u, v),\ z=z(u, v),\ (u, v)\in D,$$

где $x(u, v)$, $y(u, v)$, $z(u, v)$ – гладкие функции своих аргументов, причем выполнено соотношение

$$\text{rang}\begin{pmatrix} x_u(u,v) & y_u(u,v) & z_u(u,v) \\ x_v(u,v) & y_v(u,v) & z_v(u,v) \end{pmatrix}=2.$$

D – некоторая область на плоскости параметров u и v.

Последнее равенство означает, что в каждой точке регулярной поверхности существует касательная плоскость и эта плоскость при непрерывном перемещении по поверхности текущей точки изменяется непрерывно. Эти уравнения называются параметрическими уравнениями поверхности. Их часто записывают также в векторной форме:

$$\boldsymbol{r}=\boldsymbol{r}(u, v),\ (u, v)\in D, \text{ где } \boldsymbol{r}(u, v)=(x(u, v), y(u, v), z(u, v)).$$

Будем считать для простоты, что область на плоскости параметров представляет собой стандартный единичный квадрат. Ограничим наши рассмотрения наборами точек вида

$$V_{ij},\ i=0, 1, \ldots, m;\ j=0, 1, \ldots, n.$$

Соединяя соответствующие вершины прямолинейными отрезками, получаем контрольный многогранник (точнее, контрольный, или опорный, граф) заданного массива V.

Сглаживающая поверхность строится относительно просто, в виде так называемого тензорного произведения. Так принято называть поверхности, описываемые параметрическими уравнениями вида

$$\boldsymbol{r}(u, v)=\sum_{i=0}^{m}\sum_{j=0}^{n} a_i(u)b_j(v)V_{ij},$$

где $\alpha\le u\le\beta,\ \gamma\le v\le\delta$.

То обстоятельство, что приведенное выше уравнение можно записать в следующей форме:

$$\boldsymbol{r}(u, v)=\sum_{i=0}^{m} a_i(u)r_i(v), \text{ где } r_i=\sum_{j=0}^{n} b_j(v)V_{ij},\ i=0, \ldots, m$$

позволяет переносить на двумерный случай многие свойства, результаты и наблюдения, полученные при исследовании кривых. Если при проводимом обобщении не сильно отклоняться от рассмотренных выше классов кривых, то так построенные поверхности будут «наследовать» многие свойства одноименных кривых. В этом бесспорное преимущество задания поверхности в виде тензорного произведения.

Замечание

При повышении размерности задачи неизбежно возникает значительное число новых проблем. Предложенные ограничения на структуру заданного набора точек (естественно обобщающую структуру плоского сеточного прямоугольника) и выбор в качестве рабочих наиболее простых классов поверхностей дают определенную возможность удержать это число в рамках, разумных для первого знакомства.

Построение сглаживающих поверхностей, как и в рассмотренном выше случае кривых, удобно начать с описания уравнений элементарных фрагментов.

Ограничившись бикубическим случаем (именно такие сплайновые поверхности наиболее часто используются в задачах компьютерной графики), когда функциональные коэффициенты $a_i(u)$ и $b_j(v)$ представляют собой многочлены третьей степени относительно соответствующих переменных (кубические многочлены), запишем для заданного набора из 16 точек

$$V_{ij},\ i=0,\ 1,\ 2,\ 3,\ j=0,\ 1,\ 2,\ 3,$$

параметрические уравнения элементарных фрагментов некоторых поверхностей, считая для простоты, что область изменения параметров u и v представляет собой единичный квадрат.

Начнем с элементарной бикубической поверхности Безье. Параметрические уравнения фрагмента этой поверхности имеют следующий вид:

$$\boldsymbol{r}(u,\ v)=\sum_{i=0}^{3}\sum_{j=0}^{3}C_3^iC_3^ju^j(1-u)^{3-i}v^j(1-v)^{3-j}V_{ij},\ 0\le u\le 1,\ 0\le v\le 1.$$

или, в матричной форме:

$$\begin{pmatrix}x(u,v)\\ y(u,v)\\ z(u,v)\end{pmatrix}=(1\quad u\quad u^2\quad u^3)M^T\begin{pmatrix}V_{00}&V_{01}&V_{02}&V_{03}\\ V_{10}&V_{11}&V_{12}&V_{13}\\ V_{20}&V_{21}&V_{22}&V_{23}\\ V_{30}&V_{31}&V_{32}&V_{33}\end{pmatrix}M\begin{pmatrix}1\\ v\\ v^2\\ v^3\end{pmatrix}.$$

Здесь

$$\mathrm{M}=\begin{pmatrix}1&-3&3&1\\ 0&3&-6&3\\ 0&0&3&-3\\ 0&0&0&1\end{pmatrix}-$$

базисная матрица Безье; знаком T обозначена операция транспонирования.

Элементарная бикубическая поверхность Безье наследует многие свойства элементарной кубической кривой Безье:

- лежит в выпуклой оболочке порождающих ее точек;
- является гладкой поверхностью;
- упираясь в точки V_{00}, V_{01}, V_{10}, V_{11}, касается исходящих из них отрезков контрольного графа заданного набора.

Из элементарных вырезков поверхностей Безье подобно тому, как это делалось в одномерном случае, можно строить составные поверхности. Поговорим немного об условиях гладкости таких составных бикубических поверхностей Безье.

Пусть

$$\boldsymbol{r}=\boldsymbol{r}^{(1)}(u, v),\ 0\le u\le 1,\ 0\le v\le 1,$$

и

$$\boldsymbol{r}=\boldsymbol{r}^{(2)}(u, v),\ 0\le u\le 1,\ 0\le v\le 1 -$$

параметрические уравнения двух элементарных бикубических поверхностей Безье, порожденных наборами $V_{ij}^{(1)}$, i=0, 1, 2, 3, j=0, 1, 2, 3, и $V_{ij}^{(2)}$, i=0, 1, 2, 3, j=0, 1, 2, 3, соответственно и такими, что $V_3^{(1)} = V_0^{(2)}$.

Последнее означает, что эти элементарные фрагменты имеют общую граничную кривую. Поверхность, составленная из этих двух фрагментов, будет иметь непрерывную касательную плоскость, если каждая тройка точек вида

$$V_{2j}^{(1)}, V_{3j}^{(1)}, V_{0j}^{(2)}, V_{1j}^{(2)}$$

лежит на одной прямой и, кроме того, отношения $\dfrac{|V_{2j}^{(1)}V_{3j}^{(1)}|}{|V_{0j}^{(2)}V_{1j}^{(2)}|}$ не зависят от номера j.

Векторное параметрическое уравнение элементарного фрагмента бикубической B-сплайновой поверхности, порожденной набором 16 точек

$$V_{ij},\ i=0, 1, 2, 3, j=0, 1, 2, 3,$$

имеет следующий вид:

$$\boldsymbol{r}(u, v)=\sum_{i=0}^{3}\sum_{j=0}^{3} n_i(u)n_j(v)V_{ij},\ 0\le u\le 1,\ 0\le v\le 1$$

(функциональные коэффициенты n_0, n_1, n_2, n_3 те же, что и выше) или, в матричной форме,

$$\boldsymbol{r}(u, v)=\mathrm{U}^{\mathrm{T}}\mathrm{M}^{\mathrm{T}}\mathrm{WMV},\ 0\le u\le 1,\ 0\le v\le 1$$

где $\boldsymbol{r}(u, v)=\begin{pmatrix} x(u,v) \\ y(u,v) \\ z(u,v) \end{pmatrix}$, $\mathrm{U}=\begin{pmatrix} 1 \\ u \\ u^2 \\ u^3 \end{pmatrix}$, $\mathrm{V}=\begin{pmatrix} 1 \\ v \\ v^2 \\ v^3 \end{pmatrix}$, $\mathrm{W}=\begin{pmatrix} V_{00} & V_{01} & V_{02} & V_{03} \\ V_{10} & V_{11} & V_{12} & V_{13} \\ V_{20} & V_{21} & V_{22} & V_{23} \\ V_{30} & V_{31} & V_{32} & V_{33} \end{pmatrix}$.

Здесь M – базисная матрица кубического B-сплайна.

Как и бикубическая поверхность Безье, элементарная бикубическая B-сплайновая поверхность наследует многие свойства элементарной кубической B-сплайновой кривой:

- является гладкой;
- лежит в выпуклой оболочке порождающих ее 16 вершин;
- «повторяет» контрольную многогранную поверхность.

Построение составной бикубической B-сплайновой поверхности (обладающей весьма привлекательными геометрическими свойствами) на прямоугольнике

$$[0, m] \times [0, n]$$

с равномерными узлами (i, j), $i=0, 1, ..., m-1, m$; $j=0, 1, ..., n-1, n$ проводится во многом подобно тому, как это делается в одномерном случае.

Разумеется, существуют и весьма эффективно используются двумерные аналоги и рациональных B-сплайновых (как на равномерной сетке, так и на неравномерной (NURBS)) и бета-сплайновых кривых.

Выпишем, например, векторное уравнение элементарной бета-сплайновой поверхности – $(k, 1)$-вырезка для заданного набора $(m+1)(n+1)$ вершин. Имеем:

$$r_{k1} = r_{k1}(u,v) = \sum_{i=-2}^{1}\sum_{j=-2}^{1} b_i(u)b_j(v)V_{i+k,j+l},\ 0\le u\le 1,\ 0\le v\le 1.$$

Мы остановились лишь на некоторых простых способах построения плавно изменяющихся кривых и поверхностей.

К сказанному следует также добавить, что построение искривленных пространственных объектов является действительно непростой задачей. Она требует достаточно развитого пространственного воображения и почти постоянной готовности к встрече с

вещами неожиданными. Хотя и объяснимыми, но не сразу, а после заметных усилий. Тем не менее мы стремились к тому, чтобы по отобранному материалу и программным реализациям представленных алгоритмов у читателя сложилось в целом правильное начальное представление о геометрических сплайнах и том месте, которое они занимают в компьютерной графике.

Даже небольшая самостоятельная попытка компьютерной реализации высказанных здесь сравнительно несложных геометрических соображений будет, несомненно, полезна в освоении практически неисчерпаемых возможностей компьютерной графики.

7.2. Кривые, поверхности и сплайны в OpenGL

OpenGL предоставляет много возможностей для рисования плоских и пространственных кривых; простейший способ рисования кривой без интерполяции и сглаживания – соединить отрезками прямых все точки, определяющие кривую. Если диапазон изменения аргументов разбит на достаточно большое количество интервадов (кривая представлена большим количеством вершин), то визуально она будет выглядеть гладкой. Фрагмент функции рисования кривой набором прямолинейных отрезков выглядит просто:

```
glBegin(GL_POINTS);
for (i = 0; i < nuknot; i++)
  for (j = 0; j < nvknot; j++)
    glVertex3fv(ctlarray[i][j]);
glEnd();
```

Если необходимо сглаживание с помощью Безье–сплайнов, придется вначале в инициализирующей части программы создать таблицу вершин, сославшись на содержащий их координаты массив и указав ряд параметров функции

```
glMap1f(
GL_MAP1_VERTEX_3, //Тип вычисляемых значений
0.0, 1.0, //Пределы линейной табуляции между узлами
3, //Количество чисел, описывающих точку кривой
30, //Количество точек кривой
ctlarray[0][0]); //Массив координат точек кривой
```

Затем надо разрешить использование таблицы для построения кривой:

```
glEnable(GL_MAP1_VERTEX_3);
```

В функции рисования рисующий фрагмент выглядит так:

```
glBegin(GL_LINE_STRIP);
```

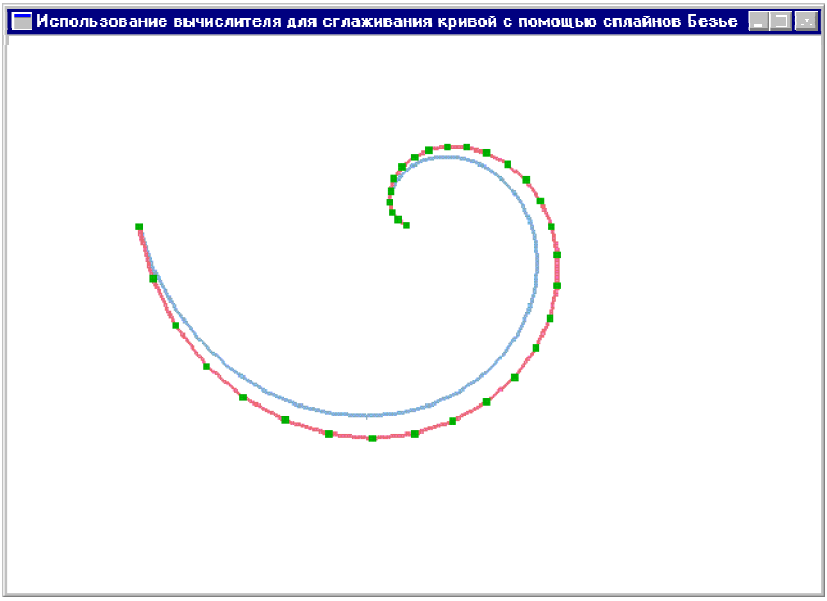
```
for (i = 0; i<(nuknot); i++)
for (j = i; j<20; j++) glEvalCoord1f((GLfloat)j/20);
glEnd();
```

Теперь мы приведем программу, рисующую сами точки кривой, кривую, построенную из отрезков прямых без сглаживания и Безье–кривую. Поэкспериментируйте с количеством точек и оцените эффективность метода.

```
#include "glos.h"
#include<math.h>
#include <GL/gl.h>
#include <GL/glu.h>
#include <GL/glaux.h>

void myinit(void);
void CALLBACK myReshape(GLsizei w, GLsizei h);
void CALLBACK display(void);

const GLdouble PI=3.14159265358979323846;
const nuknot=1, //Одна кривая
nvknot=32; //из 32 точек
GLfloat knot[nuknot][nvknot];//Массив значений параметра
/*Массив значений координат точек кривой – по 3 координаты
для каждой точки*/
GLfloat ctlarray[nuknot][nvknot][3];

//Начальная инициализация
void myinit(void)
{//Заполняем массив узлов
  int i,j,k;
  double pi,pj,r;
  for(i=0,pi=-PI;i<nuknot;i++,pi+=2*PI/nuknot)
    for(j=0,pj=-PI;j<nvknot;j++,pj+=2*PI/nvknot)
    {knot[i][j]=pi+pj;}
  //Заполняем массив значений координат
  for(k=0;k<nuknot;k++)
    for(i=0;i<nvknot;i++)
    {
      r=0.6*(knot[k][i]);//(knot[i][j]);
      ctlarray[k][i][0]= r*cos(knot[k][i]);
      ctlarray[k][i][1]=r*sin(knot[k][i]);
      //ctlarray[k][i][2]=knot[k][i]*knot[k][i];
      ctlarray[k][i][2]=cos(knot[k][i]);
    }
  glClearColor(0.0, 0.0, 0.0, 1.0);
  //Создаем таблицу координат вершин
  glMap1f(GL_MAP1_VERTEX_3, 0.0, 1.0,3,nvknot,
ctlarray[0][0]);
```

```
  glEnable(GL_MAP1_VERTEX_3); /*Разрешаем ее использование
в вычислениях*/
}

//Функция рисования
void CALLBACK display(void)
{
  int i,j;
  glPointSize(5.0);
  glLineWidth(3.0);
  glColor3f(0.4, 0.6, 0.8);
  glClear(GL_COLOR_BUFFER_BIT);
  //Рисуем кривую по результатам вычислений
  glBegin(GL_LINE_STRIP);
  for (i = 0; i<(nuknot); i++)
    for (j = i; j<nvknot; j++)
      glEvalCoord1f((GLfloat)j/nvknot);
  glEnd();
  //Рисуем кривую прямыми отрезками
  glColor3f(0.9, 0.3, 0.4);
  glBegin(GL_LINE_STRIP);
  for (i = 0; i<(nuknot); i++)
    for (j = i; j<nvknot; j++)
      glVertex3fv(ctlarray[i][j]);
  glEnd();
  //Рисуем точки кривой
  glColor3f(0.0, 0.6, 0.0);
  glBegin(GL_POINTS);
  for (i = 0; i < nuknot; i++)
    for (j = 0; j < nvknot; j++)
      glVertex3fv(ctlarray[i][j]);
  glEnd();
  glFlush();
}

void CALLBACK myReshape(GLsizei w, GLsizei h)
{
  if (!h) return;
  glViewport(0, 0, w, h);
  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();
  if (w <= h)
    glOrtho(-5.0, 5.0, -5.0*(GLfloat)h/(GLfloat)w,
5.0*(GLfloat)h/(GLfloat)w, -5.0, 5.0);
  else
    glOrtho(-5.0*(GLfloat)w/(GLfloat)h,
5.0*(GLfloat)w/(GLfloat)h, -5.0, 5.0, -5.0, 5.0);
  glMatrixMode(GL_MODELVIEW);
```

```
  glLoadIdentity();
}

int main(int argc, char** argv)
{
  auxInitDisplayMode (AUX_SINGLE | AUX_RGB);
  auxInitPosition (0, 0, 500, 500);
  auxInitWindow ("Использование вычислителя для
сглаживания кривой с помощью сплайнов Безье");
  myinit();
  auxReshapeFunc (myReshape);
  auxMainLoop(display);
  return(0);
}
```



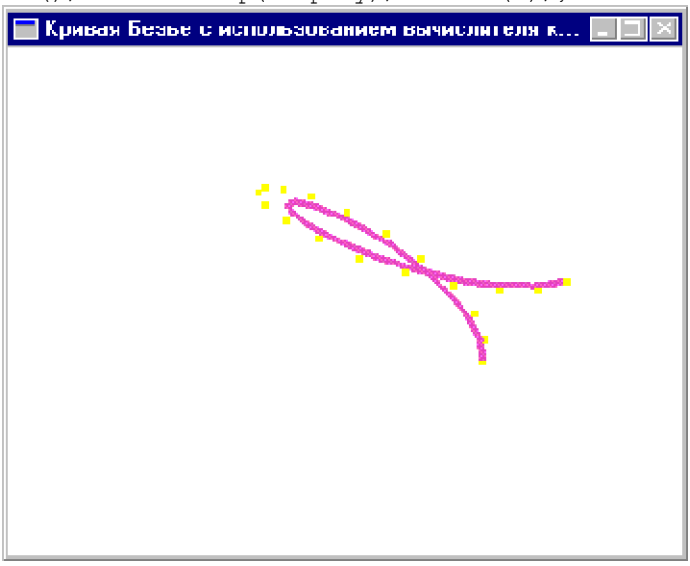
Для разнообразия приведем аналогичный пример с вращающейся кривой:

```
/*
  Эта программа рисует график функции с использованием
сплайнов Безье. Назначение и аргументы используемых
функций получите по клавише F1, установив курсор на имени
функции
*/

#include <windows.h>
// Запрещаем предупреждения о преобразовании данных
#pragma warning(disable : 4244) // MIPS
```

```
#pragma warning(disable : 4136) // X86
#pragma warning(disable : 4051) // ALPHA
#include <GL/gl.h>
#include <GL/glu.h>
#include <GL/glaux.h>
#include <math.h>

//Прототипы наших функций
void myinit(void);
static void CALLBACK myReshape(GLsizei w, GLsizei h);
static void CALLBACK display(void);

#define N 20 //Количество точек нашей функции
#define PI 3.14159265358979323846
GLint w=400,h=300;
GLfloat R1=1.0,R2=3.0f; /*Радиусы эллипса, который мы
будем рисовать*/
GLdouble al;
GLfloat ctrlpoints[N][3];/*Массив точек функции - у каждой
по 3 координаты */

void myinit(void)
{
  GLint i;
  GLfloat alpha;
  if (!h) return;
  auxInitPosition(0,0,w,h);
  auxInitDisplayMode( AUX_RGB | AUX_DEPTH | AUX_DOUBLE );
  auxInitWindow( "Кривая Безье с использованием
вычислителя координат" );
  auxReshapeFunc(myReshape);
  auxIdleFunc(display );
  glClearColor( 0.0, 0.0, 0.0, 1.0 );
  glClearDepth( 1.0 );
  glEnable(GL_DEPTH_TEST);

  glMatrixMode( GL_PROJECTION );
  if (w <= h)
  //glOrtho умножает текущую матрицу на ортографическую
    glOrtho(-5.0, 5.0, -5.0*(GLfloat)h/(GLfloat)w,
5.0*(GLfloat)h/(GLfloat)w, -5.0, 5.0);
  else
    glOrtho(-5.0*(GLfloat)w/(GLfloat)h,
5.0*(GLfloat)w/(GLfloat)h, -5.0, 5.0, -5.0, 5.0);
  glMatrixMode( GL_MODELVIEW );

  for(alpha=0.0f,i=0;i<N;i++,alpha+=2.0f*(GLfloat)PI/N)
  {
```

```
    ctrlpoints[i][0]=R2*cos(alpha); //Координаты x
    ctrlpoints[i][1]=R1*sin(alpha); //Координаты y
    ctrlpoints[i][2]=alpha;
    //5.0*sin(alpha)*cos(alpha); //Координаты z
  }
  glClearColor(0.0, 0.0, 0.0, 1.0); //Черный фон
  glEnable(GL_MAP1_VERTEX_3);
  glMap1f(GL_MAP1_VERTEX_3, 0.0f, 1.0f, 3, N,
&ctrlpoints[0][0]);//1-мерный вычислитель
  //glShadeModel(GL_FLAT);
}

static void CALLBACK display(void)
{
  int i;
  al+=0.2;
  glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );
  glPushMatrix();
  //glTranslatef(1,1,1);
  glRotatef(al,1.0,0.0,0.0);
  glRotatef(al,.0,1.0,0.0);
  glRotatef(al,.0,0.0,1.0);
  glPushAttrib(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);
  glColor3f(0.9f, 0.2f, 0.7f);//Цвет линии
  glLineWidth(4.0f); //Ширина линии

  glBegin(GL_LINE_STRIP); //Рисуем линию
  for (i = 0; i <=N ; i++) glEvalCoord1f((GLfloat)i/N);
  glEnd();

  glPointSize(4.0); //Размер точек
  glColor3f(1.5, 1.0, 0.0); //Цвет точек

  glBegin(GL_POINTS); // Вывод точек
  for (i = 0; i < N; i++) glVertex3fv(&ctrlpoints[i][0]);
  glEnd();

  glPopAttrib();
  glPopMatrix();
  glFinish();
  SwapBuffers(wglGetCurrentDC());
}

static void CALLBACK myReshape(GLsizei w, GLsizei h)
{
  if (!h) return;
  glViewport(0, 0, w, h);
  glMatrixMode(GL_PROJECTION);
```

```
  glLoadIdentity();
  if (w <= h)
  //glOrtho умножает текущую матрицу на ортографическую
    glOrtho(-5.0, 5.0, -5.0*(GLfloat)h/(GLfloat)w,
5.0*(GLfloat)h/(GLfloat)w, -5.0, 5.0);
  else
    glOrtho(-5.0*(GLfloat)w/(GLfloat)h,
5.0*(GLfloat)w/(GLfloat)h, -5.0, 5.0, -5.0, 5.0);
  glMatrixMode(GL_MODELVIEW);
  glLoadIdentity();
}

int _CRTAPI1 main()
{ myinit(); auxMainLoop(display); return(0);}
```



Помимо рассмотренных можно использовать также рациональные B–сплайны (NURBS–кривые). Оставляем эту возможность на самостоятельное освоение, а мы перейдем к краткому рассмотрению методов построения поверхностей.

Построение поверхностей с помощью Безье–сплайнов отличается от построения кривых необходимостью строить двумерные таблицы, сетки и вызывать на выполнение двумерный вычислитель координат.

Фрагмент инициализации при этом имеет вид:

```
glMap2f(GL_MAP2_VERTEX_3, 0, 1, 3, 4,0, 1, 3*N, 4,
&ctrlpoints[0][0][0]);
```

```
glEnable(GL_MAP2_VERTEX_3);
glMapGrid2f(40, -1.2, 1.2, 40, -1.2, 1.2);
```
а фрагмент рисования:
```
glEvalMesh2(GL_FILL, 0, 40, 0, 40);
```

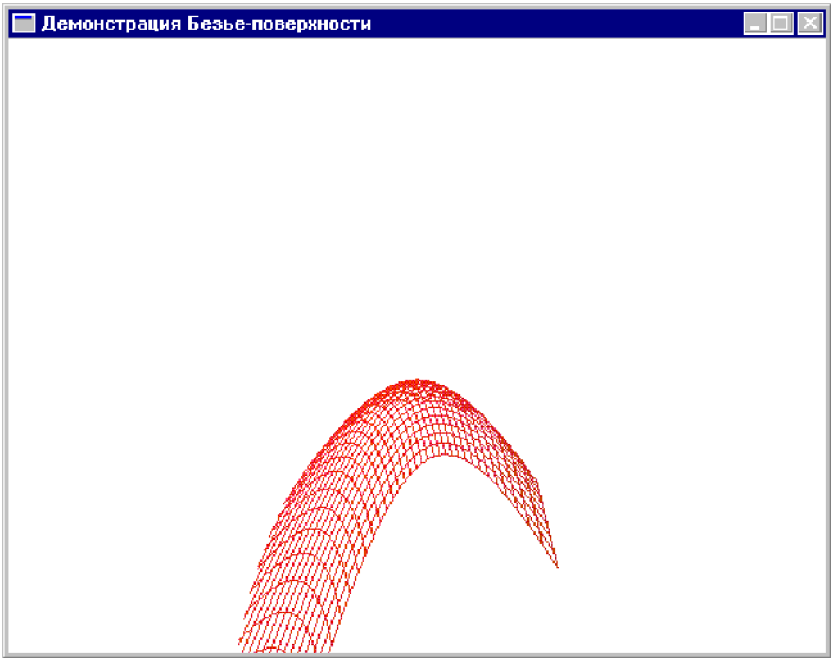
В приводимом ниже примере лицевой и тыльной сторонам поверхности присвоены различные оптические свойства, установлен движущийся источник света и вращается система координат с перспективной матрицей проектирования.

```
#include "glos.h"
#include<math.h>
#include <GL/gl.h>
#include <GL/glu.h>
#include <GL/glaux.h>

void Init(void);
void CALLBACK Reshape(GLsizei w, GLsizei h);
void CALLBACK DrawScene(void);

GLfloat latitude, longitude, radius;
GLvoid drawLight(GLvoid);
void polarView( GLdouble, GLdouble, GLdouble, GLdouble);

#define N 4
GLsizei w,h;
GLfloat ctrlpoints[N][N][3], */Массив координат и пределы
изменения параметров*/
        UMAX,UMIN,VMAX,VMIN,DU,DV;

//Подпрограмма вращения координатной системы
void polarView(GLdouble radius, GLdouble twist, GLdouble
latitude, GLdouble longitude)
{
  glTranslated(-0.0, 0.0,-radius);
  glRotated( -twist, 0.0, 0.0, 1.0 );
  glRotated( -latitude, 1.0, 0.0, 0.0);
  glRotated( longitude, 0.0, 0.0, 1.0);
}

//Рисование
void CALLBACK DrawScene(void)
{
  static GLfloat whiteAmbient[] = {0.3, 0.3, 0.3, 1.0};
  static GLfloat redAmbient[] = {0.3, 0.1, 0.1, 1.0};
  static GLfloat greenAmbient[] = {0.1, 0.3, 0.1, 1.0};
  static GLfloat blueAmbient[] = {0.1, 0.1, 0.3, 1.0};
  static GLfloat whiteDiffuse[] = {1.0, 1.0, 1.0, 1.0};
  static GLfloat redDiffuse[] = {1.0, 0.0, 0.0, 1.0};
```

```
  static GLfloat greenDiffuse[] = {0.0, 1.0, 0.0, 1.0};
  static GLfloat blueDiffuse[] = {0.0, 0.0, 1.0, 1.0};
  static GLfloat whiteSpecular[] = {1.0, 1.0, 1.0, 1.0};
  static GLfloat redSpecular[] = {1.0, 0.0, 0.0, 1.0};
  static GLfloat greenSpecular[] = {0.0, 1.0, 0.0, 1.0};
  static GLfloat blueSpecular[] = {0.0, 0.0, 1.0, 1.0};
  static GLfloat lightPosition0[] = {1.0, 1.0, 1.0, 1.0};
  static GLfloat param = 0.0;
  glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );
  glPushMatrix();
  latitude += 0.2;
  longitude += 0.2;
  polarView( radius, 0, latitude, longitude );
  glPushMatrix();
  //param += 2.0;
  glRotatef(param, 1.0, 0.0, 1.0);
  glTranslatef( 0.0, 1.5, 0.0);
  glLightfv(GL_LIGHT0, GL_POSITION, lightPosition0);
  //drawLight();
  glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT, redAmbient);
  glMaterialfv(GL_FRONT_AND_BACK, GL_DIFFUSE, redDiffuse);
  glMaterialfv(GL_FRONT_AND_BACK, GL_SPECULAR,
whiteSpecular);
  glMaterialf(GL_FRONT, GL_SHININESS, 100.0);
  //auxSolidCylinder( 0.3, 0.6 );
  glPopMatrix();
  glPushAttrib(GL_LIGHTING_BIT);
  //Здесь рисуем
  glMaterialfv(GL_BACK, GL_AMBIENT, blueAmbient);
  glMaterialfv(GL_BACK, GL_EMISSION/*GL_DIFFUSE*/,
blueDiffuse);
  glMaterialfv(GL_FRONT, GL_AMBIENT, redAmbient);
  glMaterialfv(GL_FRONT, GL_DIFFUSE, redDiffuse);
  glMaterialfv(GL_FRONT, GL_SPECULAR, redSpecular);
  glMaterialf(GL_FRONT, GL_SHININESS, 100.0);
  //auxSolidCone( 0.3, 0.6 );
  glEvalMesh2(GL_FILL, 0, 40, 0, 40);
  glPopMatrix();
  glFinish();
  SwapBuffers(wglGetCurrentDC());
}

void Init(void)
{
  GLint i,j;
  GLfloat u,v,r;
  GLfloat maxObjectSize, aspect;
  GLdouble near_plane, far_plane;
```

```
  GLsizei w,h;
  GLfloat ambientProperties[] = {0.7, 0.7, 0.7, 1.0};
  GLfloat diffuseProperties[] = {0.8, 0.8, 0.8, 1.0};
  GLfloat specularProperties[] = {1.0, 1.0, 1.0, 1.0};
  UMAX=VMAX=1.2;
  UMIN=VMIN=-1.2;
  DU=(UMAX-UMIN)/(N-1);
  DV=(VMAX-VMIN)/(N-1);
  for(i=0,u=UMIN;i<N;i++,u+=DU)
    for(j=0,v=VMIN;j<N;j++,v+=DV)
    {
      ctrlpoints[j][i][0]=u; ctrlpoints[j][i][1]=v;
      r=u*u+v*v; ctrlpoints[j][i][2]=r;//cos(r)/(r+1);
    }
  w = 1024.0;
  h = 768.0;
  auxInitPosition( w/4, h/4, w/2, h/2);
  auxInitDisplayMode( AUX_RGB | AUX_DEPTH | AUX_DOUBLE );
  auxInitWindow( "AUX Library Demo" );
  auxIdleFunc( DrawScene );
  auxReshapeFunc(Reshape);
  glClearColor( 0.0, 0.0, 0.0, 1.0 );
  glClearDepth( 1.0 );
  glEnable(GL_DEPTH_TEST);
  glEnable(GL_LIGHTING);
  glLightfv( GL_LIGHT0, GL_AMBIENT, ambientProperties);
  glLightfv( GL_LIGHT0, GL_DIFFUSE, diffuseProperties);
  glLightfv( GL_LIGHT0, GL_SPECULAR, specularProperties);
  glLightModelf(GL_LIGHT_MODEL_TWO_SIDE, 1.0);
  glEnable( GL_LIGHT0 );
  glMatrixMode( GL_PROJECTION );
  aspect = (GLfloat) w/h;
  gluPerspective(25.0, aspect, 3.0, 5.0 );
  glMatrixMode( GL_MODELVIEW );
  near_plane = 1.0;
  far_plane = 5.0;
  maxObjectSize = 7.0;
  radius = near_plane + maxObjectSize/2.0;
  latitude = 0.0;
  longitude = 0.0;
  //Цвет фона
  glClearColor (0.0, 0.0, 0.0, 1.0);
  glEnable (GL_DEPTH_TEST);
  glMap2f(GL_MAP2_VERTEX_3, 0, 1, 3, 4,0, 1, 3*N, 4,
&ctrlpoints[0][0][0]);
  glEnable(GL_MAP2_VERTEX_3);
  glEnable(GL_AUTO_NORMAL);
  glEnable(GL_NORMALIZE);
```

```
  glMapGrid2f(40, -1.2, 1.2, 40, -1.2, 1.2);
}

void CALLBACK Reshape(GLsizei w, GLsizei h)
{
  GLfloat aspect=w/h;
  glViewport( 0, 0, w,h);
  glMatrixMode( GL_PROJECTION );
  glLoadIdentity();
  gluPerspective(125.0, aspect, 3.0, 7.0 );
  glMatrixMode( GL_MODELVIEW );
  glLoadIdentity();
}

GLvoid drawLight(GLvoid)
{
  glPushAttrib(GL_LIGHTING_BIT);
  glDisable(GL_LIGHTING);
  glColor3f(1.0, 1.0, 1.0);
  auxSolidDodecahedron(0.1);
  glPopAttrib();
}

int main(int argc, char** argv)
{
  Init();
  auxMainLoop(DrawScene);
  return(0);
}
```


Демонстрация Безье-поверхности

8. Обработка сигналов (событий) от мыши и клавиатуры

Мы уже рассматривали в самой первой программе объявления функций, вызываемых при работе с клавиатурой и мышью. Размещаются они в инициализирующей части программы и выглядят, например, так:

auxMouseFunc(AUX_LEFTBUTTON, AUX_MOUSEDOWN, funcLBMouse);
auxMouseFunc(AUX_RIGHTBUTTON, AUX_MOUSEDOWN, funcRBMouse);
auxKeyFunc(AUX_LEFT, funcLKey);
auxKeyFunc(AUX_RIGHT, funcRKey);

Сами функции мы сделаем чрезвычайно простыми – они будут присваивать значения целочисленному флагу переключения; по этому флагу мы будем менять картину на стене нашей сцены, запрограммированной в работе, посвященной текстурам.

Соответствующая программа повторена ниже с небольшими вставками, обрабатывающими события от мыши и от клавиш – стрелок влево–вправо для смены привязки текстуры к прямоугольнику «картины на стене».

```
#include "glos.h"
#include<stdio.h>
#include<stdlib.h>
#include<math.h>
#include <GL/gl.h>
#include <GL/glu.h>
#include <GL/glaux.h>

void Init(void);
void CALLBACK Reshape(GLsizei w, GLsizei h);
void CALLBACK DrawScene(void);

//Параметры вращения системы координат
GLfloat latitude, longitude, radius;

//Прототипы функций
void polarView( GLdouble, GLdouble, GLdouble, GLdouble);

#define N 3 //Количество массивов контрольных точек
GLsizei w,h; //Для размеров окна
/*
```

```
GLfloat ctrlpoints[N][4][3]= //Массивы контрольных точек
{
{{1,1,0},{-1,1,0},{-1,-1,0},{1.-1,0}},
{{1,0,1},{1,0,-1},{-1,0,-1},{-1,0,1}},
{{0,1,1},{0,1,-1},{0,-1,-1},{0,-1,1}}
};
*/
GLfloat UMAX,UMIN,VMAX,VMIN,DU,DV;
GLfloat texpts[2][2][2] = {{{0.0, 0.0}, {0.0, 1.0}},{{1.0,
0.0}, {1.0, 1.0}}};
GLint texFlag;

#define imageWidth 64 //Для размеров образа текстуры
#define imageHeight 64
#define NUM_TEXTURES 5

GLuint texobj[NUM_TEXTURES];
//Массивы текстур
GLubyte image[NUM_TEXTURES][3*imageWidth*imageHeight];
//Для указателей на текстуры, получаемые из файла
AUX_RGBImageRec* imF[NUM_TEXTURES];
GLint iW[NUM_TEXTURES],iH[NUM_TEXTURES];/*Для размеров
файловых текстур*/
//Для указателей битовых образов текстур
GLvoid *iBits[NUM_TEXTURES];
//Функция реакции на мышиные кнопки
GLvoid CALLBACK funcLBMouse(AUX_EVENTREC * a){texFlag=1;}
GLvoid CALLBACK funcRBMouse(AUX_EVENTREC *a){texFlag=0;}
void APIENTRY funcLKey(){texFlag=1;}
void APIENTRY funcRKey(){texFlag=0;}

//Функция масштабирования файловых текстур
BOOL TexMapScalePow2(GLint i)
{
  GLint glMaxTexDim ;
  double xPow2, yPow2;
  int ixPow2, iyPow2;
  int xSize2, ySize2;
  glGetIntegerv(GL_MAX_TEXTURE_SIZE, &glMaxTexDim);
  glMaxTexDim = min(256, glMaxTexDim);
  if (iW[i] <= glMaxTexDim)
    xPow2 = log((double)iW[i]) / log(2.0);
  else xPow2 = log((double)glMaxTexDim) / log(2.0);
  if (iH[i] <= glMaxTexDim)
    yPow2 = log((double)iH[i]) / log(2.0);
  else yPow2 = log((double)glMaxTexDim) / log(2.0);
  ixPow2 = (int)xPow2;
  iyPow2 = (int)yPow2;
```

```
  if (xPow2 != (double)ixPow2) ixPow2++;
  if (yPow2 != (double)iyPow2) iyPow2++;
  xSize2 = 1 << ixPow2;
  ySize2 = 1 << iyPow2;
 BYTE *pData=(BYTE*)malloc(xSize2*ySize2*3*sizeof(BYTE));
  if (!pData) return FALSE;
  BOOL bRes = gluScaleImage(GL_RGB, iW[i], iH[i],
  GL_UNSIGNED_BYTE, iBits[i], xSize2, ySize2,
GL_UNSIGNED_BYTE, pData);
  if(bRes){
    printf("Ошибка выполнения команды gluScaleImage");
    return FALSE;
  }
  free(iBits[i]);
  iBits[i] = pData;
  iW[i] = xSize2;
  iH[i] = ySize2;
  return TRUE;
}

//Функция создания битовых образов текстур из файлов
void makeImageF(void)
{
  imF[0]=auxDIBImageLoad("backgrd1.bmp");//Пейзаж
  iW[0]= imF[0]->sizeX;
  iH[0]= imF[0]->sizeY;
  iBits[0] = imF[0]->data;
  TexMapScalePow2(0);
  gluBuild2DMipmaps(GL_TEXTURE_2D,3,iW[0],iH[0],GL_RGB,
GL_UNSIGNED_BYTE, iBits[0]);
  imF[1]=auxDIBImageLoad("Rock.bmp"); //Цемент
  iW[1]= imF[1]->sizeX;
  iH[1]= imF[1]->sizeY;
  iBits[1] = imF[1]->data;
  TexMapScalePow2(1);
  gluBuild2DMipmaps(GL_TEXTURE_2D,3,iW[1],iH[1],GL_RGB,
GL_UNSIGNED_BYTE,iBits[1]);
  imF[2]=auxDIBImageLoad("metalbal.bmp"); //Металл
  iW[2]= imF[2]->sizeX;
  iH[2]= imF[2]->sizeY;
  iBits[2] = imF[2]->data;
  TexMapScalePow2(2);
  gluBuild2DMipmaps(GL_TEXTURE_2D,3,iW[2],iH[2],GL_RGB,
GL_UNSIGNED_BYTE,iBits[2]);
  imF[3]=auxDIBImageLoad("brick.bmp"); /*Какой-то линолеум
на полу*/
  iW[3]= imF[3]->sizeX;
  iH[3]= imF[3]->sizeY;
```

```
  iBits[3] = imF[3]->data;
  TexMapScalePow2(3);
  gluBuild2DMipmaps(GL_TEXTURE_2D,3,iW[3],iH[3],GL_RGB,
GL_UNSIGNED_BYTE,iBits[3]);
  imF[4]=auxDIBImageLoad("phong7-1.bmp"); /*Какой-то
линолеум на полу*/
  iW[4]= imF[4]->sizeX;
  iH[4]= imF[4]->sizeY;
  iBits[4] = imF[4]->data;
  TexMapScalePow2(4);
  gluBuild2DMipmaps(GL_TEXTURE_2D,3,iW[4],iH[4],GL_RGB,
GL_UNSIGNED_BYTE,iBits[4]);
}

/*
//Функция непосредственного создания битового образа
//текстуры
void makeImage(void)
{
  int i, j;
  float ti, tj;
  for (i = 0; i < imageWidth; i++) {
    ti = 2.0*3.14159265*i/imageWidth;
    for (j = 0; j < imageHeight; j++) {
      tj = 2.0*3.14159265*j/imageHeight;
      image[0][3*(imageHeight*i+j)] = (GLubyte)
20*(1.0+sin(ti));
      image[0][3*(imageHeight*i+j)+1] = (GLubyte)
40*(1.0+cos(2*tj));
      image[0][3*(imageHeight*i+j)+2] = (GLubyte)
60*(1.0+cos(ti+tj));
      image[1][3*(imageHeight*i+j)] = (GLubyte)
127*(1.0+sin(ti));
      image[1][3*(imageHeight*i+j)+1] = (GLubyte)
27*(1.0+cos(2*tj));
      image[1][3*(imageHeight*i+j)+2] = (GLubyte)
227*(1.0+cos(ti+tj));
      image[2][3*(imageHeight*i+j)] = (GLubyte)
227*(1.0+sin(ti));
      image[2][3*(imageHeight*i+j)+1] = (GLubyte)
127*(1.0+cos(2*tj));
      image[2][3*(imageHeight*i+j)+2] = (GLubyte)
27*(1.0+cos(ti+tj));
    }
  }
}
*/
```

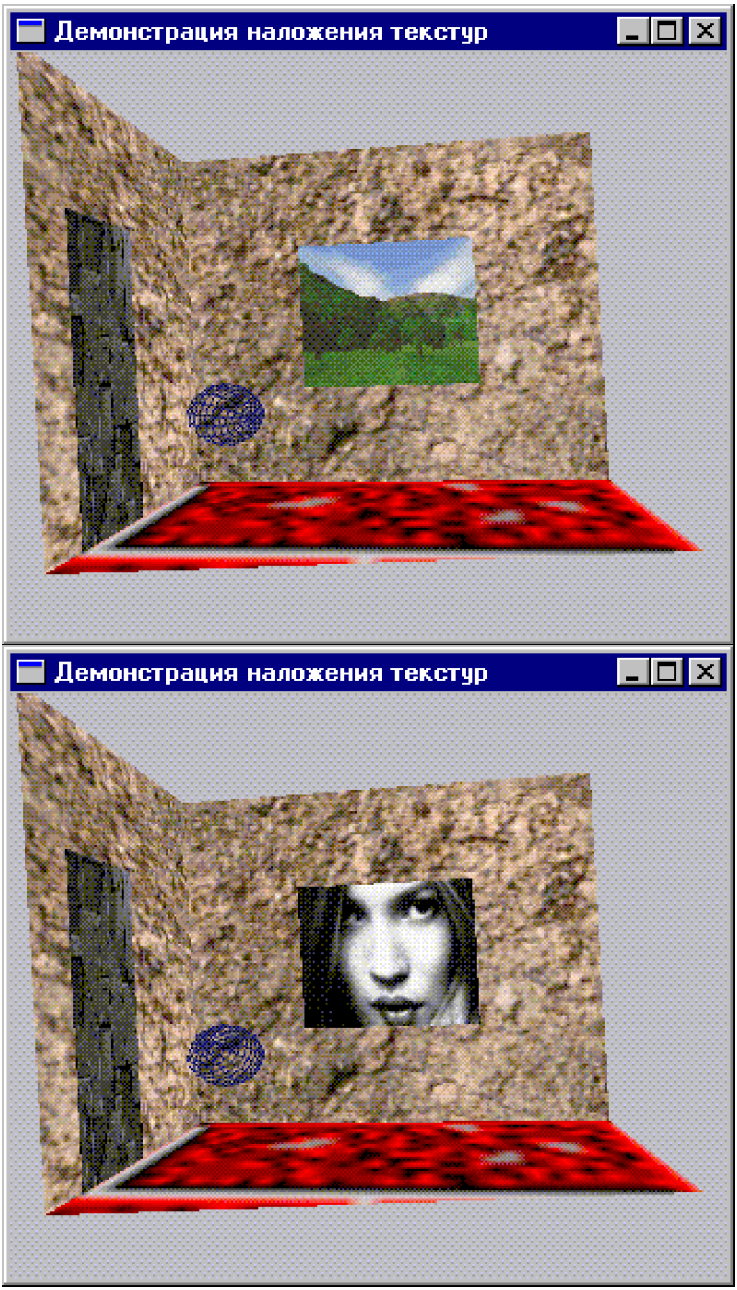
```
//Переносы и вращения
void polarView(GLdouble radius, GLdouble twist, GLdouble
latitude,GLdouble longitude)
{
  glTranslated(-1.5, -1.5,-radius);
  glRotated( -twist, 1.0, 0.0, 0.0 );
  glRotated( -latitude, 0.0, 1.0, 0.0);
  glRotated( longitude, 0.0, 0.0, 1.0);
}

//Функция рисования
void CALLBACK DrawScene(void)
{
  //Оптические свойства материалов будут из этого набора
  static GLfloat whiteAmbient[] = {0.3, 0.3, 0.3, 1.0};
  static GLfloat redAmbient[] = {0.3, 0.1, 0.1, 1.0};
  static GLfloat greenAmbient[] = {0.1, 0.3, 0.1, 1.0};
  static GLfloat blueAmbient[] = {0.1, 0.1, 0.3, 1.0};
  static GLfloat whiteDiffuse[] = {1.0, 1.0, 1.0, 1.0};
  static GLfloat redDiffuse[] = {1.0, 0.0, 0.0, 1.0};
  static GLfloat greenDiffuse[] = {0.0, 1.0, 0.0, 1.0};
  static GLfloat blueDiffuse[] = {0.0, 0.0, 1.0, 1.0};
  static GLfloat whiteSpecular[] = {1.0, 1.0, 1.0, 1.0};
  static GLfloat redSpecular[] = {1.0, 0.0, 0.0, 1.0};
  static GLfloat greenSpecular[] = {0.0, 1.0, 0.0, 1.0};
  static GLfloat blueSpecular[] = {0.0, 0.0, 1.0, 1.0};
  //Позиция источника света
  static GLfloat lightPosition0[] = {1.0, 1.0, 1.0, 1.0};
  static double al;/*Угол вращения тел в координатных
отсеках*/
  al+=2.;
  glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );
  //Вращаем всю систему
  glPushMatrix();
  latitude = 6.5;
  longitude = 2.5;
  polarView( radius, 0, latitude, longitude );
  //Рисуем и вращаем шар
  glPushMatrix();
  glTranslatef(1.0,1.0,1.9);
  glRotatef(al,1.0,0.5,0.2);
  glRotatef(al,.0,1,0);
  glPushAttrib(GL_LIGHTING_BIT);
  glMaterialfv(GL_BACK, GL_AMBIENT, blueAmbient);
  glMaterialfv(GL_BACK, GL_DIFFUSE, blueDiffuse);
  glMaterialfv(GL_FRONT, GL_AMBIENT, blueAmbient);
  glMaterialfv(GL_FRONT, GL_DIFFUSE, blueDiffuse);
  glMaterialfv(GL_FRONT, GL_SPECULAR, redSpecular);
```

```
  glMaterialf(GL_FRONT, GL_SHININESS, 100.0);
  auxWireSphere(0.2);
  glPopAttrib();
  glPopMatrix();
  //glEvalMesh2(GL_FILL, 0, 20, 0, 20);
  /*Рисуем координатные плоскости, привязывая к каждой
свою текстуру*/
  glEnable(GL_TEXTURE_2D);
  for (int i = 0; i < NUM_TEXTURES; i++) texobj[i] = i+1;
  glBindTexture(GL_TEXTURE_2D,texobj[1]);
  glBegin(GL_QUADS);
  glTexCoord2f(1.0f, 1.0f); glVertex3f(3.5f, 3.5f, 0.0f);
  glTexCoord2f(0.0f, 1.0f); glVertex3f(0.0f, 3.5f, 0.0f);
  glTexCoord2f(0.0f, 0.0f); glVertex3f(0.0f, 0.0f, 0.0f);
  glTexCoord2f(1.0f, .0f); glVertex3f( 3.5f, 0.0f, 0.0f);
  glEnd();

  //Картина на стене в зависимости от нажатой кнопки мыши
  //или клавиш - стрелок влево - вправо
  if(texFlag&1)
    glBindTexture(GL_TEXTURE_2D,texobj[0]);
  else
    glBindTexture(GL_TEXTURE_2D,texobj[4]);
  glBegin(GL_QUADS);
  glTexCoord2f(1.0f,1.0f); glVertex3f(2.5f, 2.5f, 0.01f);
  glTexCoord2f(0.0f, 1.0f); glVertex3f(1.f, 2.5f, 0.01f);
  glTexCoord2f(.0f, .0f); glVertex3f(1.f, 1.f, 0.01f);
  glTexCoord2f(1.0f,0.0f); glVertex3f( 2.5f, 1.f, 0.01f);
  glEnd();
  glBindTexture(GL_TEXTURE_2D,texobj[3]);
  glBegin(GL_QUADS);
  glTexCoord2f(0.0f, 0.0f); glVertex3f( 3.5f, .0f, 3.5f);
  glTexCoord2f(0.0f, 1.0f); glVertex3f( 3.5f, .0f, 0.0f);
  glTexCoord2f(1.0f, 1.0f); glVertex3f(0.f, .0f, 0.0f);
  glTexCoord2f(1.0f, 0.0f); glVertex3f(0.0f, .0f, 3.5f);
  glEnd();

  glBindTexture(GL_TEXTURE_2D,texobj[1]);
  glBegin(GL_QUADS);
  glTexCoord2f(0.0f, 0.0f); glVertex3f(.0f, 3.5f, 3.5f);
  glTexCoord2f(0.0f, 1.0f); glVertex3f(.0f, 3.5f, 0.0f);
  glTexCoord2f(1.0f, 1.0f); glVertex3f(.0f, 0.0f, 0.0f);
  glTexCoord2f(1.0f, 0.0f); glVertex3f(.0f, 0.0f, 3.5f);
  glEnd();
  glBindTexture(GL_TEXTURE_2D,texobj[2]);
  glBegin(GL_QUADS);
  glTexCoord2f(0.0f, 0.0f); glVertex3f(.01f, 2.5f, 1.7f);
  glTexCoord2f(0.0f, 1.0f); glVertex3f(.01f, 2.5f, 1.0f);
```

```
  glTexCoord2f(1.0f, 1.0f); glVertex3f(.01f, 0.0f, 1.0f);
  glTexCoord2f(1.0f, 0.0f); glVertex3f(.01f, 0.0f, 1.7f);
  glEnd();
  glDisable(GL_TEXTURE_2D);
  /*
  glBegin(GL_QUADS);
  glColor3f(1.0f, 0.0f, 0.0f);
  glVertex3f(-1.0f, 1.0f, -1.0f);
  glVertex3f(-1.0f, 1.0f, 1.0f);
  glVertex3f( 1.0f, 1.0f, 1.0f);
  glVertex3f( 1.0f, 1.0f, -1.0f);
  glColor3f(0.0f, 1.0f, 0.0f);
  glVertex3f( 1.0f, -1.0f, -1.0f);
  glVertex3f( 1.0f, -1.0f, 1.0f);
  glVertex3f(-1.0f, -1.0f, 1.0f);
  glVertex3f(-1.0f, -1.0f, -1.0f);
  glColor3f(0.0f, 0.0f, 1.0f);
  glVertex3f(-1.0f, -1.0f, -1.0f);
  glVertex3f(-1.0f, 1.0f, -1.0f);
  glVertex3f(-1.0f, 1.0f, 1.0f);
  glVertex3f(-1.0f, -1.0f, 1.0f);
  glColor3f(1.0f,0.0f,0.0f); glVertex3f(1.0f,-1.0f,1.0f);
  glColor3f(0.0f,1.0f,0.0f); glVertex3f(1.0f,1.0f,1.0f);
  glColor3f(0.0f,0.0f,1.0f); glVertex3f(1.0f,1.0f,-1.0f);
  glColor3f(1.0f,0.0f,1.0f);glVertex3f(1.0f,-1.0f,-1.0f);
  glEnd();
  */
  glPopMatrix();
  glFinish();
  SwapBuffers(wglGetCurrentDC());
}

void Init(void)
{
  GLint i,j;
  GLfloat u,v,r;
  GLfloat maxObjectSize, aspect;
  GLdouble near_plane, far_plane;
  GLsizei w,h;
  GLfloat ambientProperties[] = {0.7, 0.7, 0.7, 1.0};
  GLfloat diffuseProperties[] = {0.8, 0.8, 0.8, 1.0};
  GLfloat specularProperties[] = {1.0, 1.0, 1.0, 1.0};
  /*
  UMAX=VMAX=1.2;
  UMIN=VMIN=-1.2;
  DU=(UMAX-UMIN)/(N-1);
  DV=(VMAX-VMIN)/(N-1);
  for(i=0,u=UMIN;i<N;i++,u+=DU)
```

```
    for(j=0,v=VMIN;j<N;j++,v+=DV)
    {
      ctrlpoints[j][i][0]=u;
      ctrlpoints[j][i][1]=v;
      r=u*u+v*v;
      ctrlpoints[j][i][2]=0.0;//*cos(r)/(r+1);
    }
  */
  w = 1024.0;
  h = 768.0;
  auxInitPosition( w/4, h/4, w/2, h/2);
  auxInitDisplayMode( AUX_RGB | AUX_DEPTH | AUX_DOUBLE );
  auxInitWindow( "Демонстрация наложения текстур" );
  auxIdleFunc( DrawScene );
  auxReshapeFunc(Reshape);
  auxMouseFunc(AUX_LEFTBUTTON,AUX_MOUSEDOWN,funcLBMouse);
 auxMouseFunc(AUX_RIGHTBUTTON,AUX_MOUSEDOWN,funcRBMouse);
  auxKeyFunc(AUX_LEFT,funcLKey);
  auxKeyFunc(AUX_RIGHT,funcRKey);
  glClearColor( 0.0, 0.0, 0.0, 1.0 );
  glClearDepth( 1.0 );
  glEnable(GL_DEPTH_TEST);
  glEnable(GL_LIGHTING);
  glLightfv( GL_LIGHT0, GL_AMBIENT, ambientProperties);
  glLightfv( GL_LIGHT0, GL_DIFFUSE, diffuseProperties);
  glLightfv( GL_LIGHT0, GL_SPECULAR, specularProperties);
  glLightModelf(GL_LIGHT_MODEL_TWO_SIDE, 1.0);
  glEnable( GL_LIGHT0 );
  glMatrixMode( GL_PROJECTION );
  aspect = (GLfloat) w/h;
  gluPerspective(65.0, aspect, 3.0, 7.0 );
  glMatrixMode( GL_MODELVIEW );
  near_plane = 2.0;
  far_plane = 7.0;
  maxObjectSize = 6.0;
  radius = near_plane + maxObjectSize/2.0;
  latitude = 0.0;
  longitude = 0.0;
  //Цвет фона
  glClearColor (0.7, 0.7, 0.7, 1.0);
  makeImageF();//Создаем файловые текстуры
  for (i = 0; i < NUM_TEXTURES; i++) texobj[i] = i+1;
  for (i = 0; i < NUM_TEXTURES; i++)
  {
    glBindTexture(GL_TEXTURE_2D, texobj[i]);
    glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE,
GL_DECAL);
```

```
    glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S,
GL_REPEAT);
    glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T,
GL_REPEAT);
    glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER,
GL_LINEAR);
    glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
GL_LINEAR);
    glTexImage2D(GL_TEXTURE_2D, 0, 3,iW[i],iH[i]/*
imageWidth, imageHeight*/,0,GL_RGB, GL_UNSIGNED_BYTE,
iBits[i]/*&image[i][0]*/);
  }
  glHint(GL_PERSPECTIVE_CORRECTION_HINT, GL_NICEST);
  /*glTexImage2D(GL_TEXTURE_2D, 0, 3, imageWidth,
imageHeight, 0,GL_RGB, GL_UNSIGNED_BYTE, image[1]);*/
  /*glTexImage2D(GL_TEXTURE_2D, 0, 3, imageWidth,
imageHeight, 0,GL_RGB, GL_UNSIGNED_BYTE, image[2]);*/
  glEnable(GL_TEXTURE_2D);
  glEnable(GL_DEPTH_TEST);
  /*glMap2f(GL_MAP2_TEXTURE_COORD_2,0,1,2,2,0,1,4,2,
&texpts[0][0][0]);*/
  glEnable(GL_MAP2_TEXTURE_COORD_2);
  /*
  glMap2f(GL_MAP2_VERTEX_3, 0, 1, 3, 4,0, 1, 3*N, 4,
&ctrlpoints[0][0][0]);
  glEnable(GL_MAP2_VERTEX_3);
  glEnable(GL_AUTO_NORMAL);
  glEnable(GL_NORMALIZE);
  glMapGrid2f(20, 0.0, 1.0, 20, 0.0, 1.0);
  */
  auxMainLoop(DrawScene);
}

void CALLBACK Reshape(GLsizei w, GLsizei h)
{
  GLfloat aspect=w/h;
  glViewport( 0, 0, w,h);
  glMatrixMode( GL_PROJECTION );
  glLoadIdentity();
  gluPerspective(65.0, aspect, 3.0, 7.0 );
  glMatrixMode( GL_MODELVIEW );
  glLoadIdentity();
}

int main(int argc, char** argv) { Init(); return(0); }
```


Демонстрация наложения текстур
Демонстрация наложения текстур

Литература

1. Тихомиров Ю. Программирование трехмерной графики. – СПб.: БХВ – Санкт-Петербург, 1999. – 256 с.

2. Шикин Е.В., Боресков А.В. Компьютерная графика. – М.: Диалог-МИФИ, 1998. – 288 с.

Учебное пособие

Александр Павлович Полищук

Графические объекты в OpenGL

Подп. к печати 21.08.2000 Формат 80x84 1/16.
Бумага офсетная №1 Усл. печ. л. 6,1
Усл. кр.-от. 6,27 Уч.-изд. л. 7,39
Тираж 300 Зак. №08-3806

КГПУ, 50086, Кривой Рог-86, пр. Гагарина, 54

Криворожская городская типография
50050, Кривой Рог-50, пр. Металлургов, 28.