МИНИСТЕРСТВО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

Московский физико-технический институт
(государственный университет)

ИСПОЛЬЗОВАНИЕ БИБЛИОТЕКИ OPENGL В ЗАДАЧАХ ПЕРЕНОСА ЯДЕРНЫХ ИЗЛУЧЕНИЙ

Учебное пособие

МОСКВА 2005

УДК 517.8
Использование библиотеки OpenGL в задачах переноса ядерных излучений: Учебное пособие/ Дербакова Е.П., Клосс Ю.Ю., Колядко Г.С., Лихоманов А.А.–М.:МФТИ, 2005.–46с.
ISBN 5-7417-0189-2

Рецензент
Доктор технических наук Каландаришвили А.Г.

Печатается по решению Редакционно-издательского совета Московского физико-технического института

Использование библиотеки OpenGL в задачах переноса ядерных излучений

Учебное издание

Дербакова Елена Павловна, Клосс Юрий Юрьевич, Колядко Генрих Степанович, Лихоманов Андрей Андреевич

Оглавление

ВВЕДЕНИЕ

Одной из важных проблем моделирования физических процессов переноса ядерных и ионизирующих излучений является визуализация результатов экспериментальных и расчетно-теоретических исследований. Для физика-исследователя необходимо иметь удобную и гибкую среду интерактивной визуализации функционалов полей ядерных излучений внутри объектов различной геометрии, а также возможность манипуляции, как объектами, так и экспериментально или теоретически полученными полями. Существующие пакеты, такие как MathLab, Mathematica и др., обладая определенными преимуществами в использовании, тем не менее не имеют многих важных функций, необходимых для создания проблемно-ориентированной интерактивной оболочки визуализации полей ядерных излучений в трехмерных средах и объектах.

В настоящем учебном пособии дается последовательное введение в стандарт OpenGL, показаны главные приемы построения интерактивных оболочек на его основе для визуализации функционалов полей излучений в операционных средах Windows 2000 и Windows XP.

1. СТАНДАРТ OPENGL

За последние годы компьютерная графика получила очень широкое распространение. Сейчас трёхмерные изображения можно увидеть везде, начиная от простых

компьютерных игр и заканчивая системами моделирования в реальном времени. Раньше, когда трёхмерная графика существовала только на суперкомпьютерах, не было единого стандарта в области графики. Все программы писались с "нуля" или с использованием накопленного опыта, но в каждой программе реализовывались свои методы для отображения графической информации. С приходом мощных процессоров и графических ускорителей трёхмерная графика стала реальностью для персональных компьютеров. Но в то же время производители программного обеспечения столкнулись с серьёзной проблемой – это отсутствие каких-либо стандартов, которые позволяли писать программы, независимые от оборудования и операционной системы. Одним из первых таких стандартов, существующих и по сей день, является OpenGL.

OpenGL – это графический стандарт в области компьютерной графики. На данный момент он является одним из самых популярных графических стандартов во всём мире. Ещё в 1982 г. в Стенфордском университете была разработана концепция графической машины, на основе которой фирма Silicon Graphics в своей рабочей станции Silicon IRIS реализовала конвейер рендеринга. Таким образом, была разработана графическая библиотека IRIS GL, на основе которой в 1992 году разработан и утверждён графический стандарт OpenGL. Разработчики OpenGL – это крупнейшие фирмы- разработчики, как оборудования, так и программного обеспечения: Silicon Graphics Inc., Microsoft, IBM Corporation, Sun Microsystems, Inc., Digital Equipment Corporation (DEC), Evans & Sutherland, Hewlett-Packard Corporation, Intel Corporation и Intergraph Corporation.

OpenGL переводится как Открытая Графическая Библиотека (Open Graphics Library), это означает, что OpenGL – это открытый и мобильный стандарт. Программы, написанные с помощью OpenGL, можно переносить практически на любые платформы, получая при этом одинаковый результат, будь то графическая станция или суперкомпьютер. OpenGL

освобождает программиста от написания программ для конкретного оборудования. Если устройство поддерживает какую-то функцию, то эта функция выполняется аппаратно, если нет, то библиотека выполняет её программно.

Что же представляет OpenGL? С точки зрения программиста OpenGL – это программный интерфейс для графических устройств, таких, как графические ускорители. Он включает около 150 различных команд, с помощью которых программист может определять различные объекты и производить рендеринг. Говоря более простым языком, Вы определяете объекты, задаёте их местоположение в трёхмерном пространстве, определяете другие параметры (поворот, растяжение ...), задаёте свойства объектов (цвет, текстура, материал ...), положение наблюдателя, а библиотека OpenGL позаботится о том, чтобы отобразить всё это на экране. Поэтому можно сказать, что библиотека OpenGL является только воспроизводящей (Rendering) библиотекой, потому что она не поддерживает какие-либо периферийные устройства, такие, как клавиатура и мышь, она также не поддерживает никаких менеджеров окон. Программист должен сам позаботиться о том, как обеспечить взаимодействие периферийных устройств с библиотекой OpenGL.

OpenGL имеет хорошо продуманную внутреннюю структуру и довольно простой процедурный интерфейс. Несмотря на это, с помощью OpenGL можно создавать сложные и мощные программные комплексы, затрачивая при этом минимальное время по сравнению с другими графическими библиотеками.

В некоторых библиотеках OpenGL (например, под X Windows) имеется возможность изображать результат не только на локальной машине, но также и по сети. Приложение, которое вырабатывает команды OpenGL, называется клиентом, а приложение, которое получает эти команды и отображает результат, – сервером. Таким образом, можно строить очень мощные воспроизводящие комплексы

на основе нескольких рабочих станций или серверов, соединённых сетью.

1.1. Основные возможности OpenGL

Что предоставляет библиотека в распоряжение программиста? Основные возможности библиотеки следующие:

- **Геометрические и растровые примитивы.** На основе геометрических и растровых примитивов строятся все объекты. Из геометрических примитивов библиотека предоставляет точки, линии, полигоны. Из растровых – битовый массив (bitmap) и образ (image)
- **Использование B-сплайнов.** B-сплайны используются для рисования кривых по опорным точкам.
- **Видовые и модельные преобразования.** С помощью этих преобразований можно располагать объекты в пространстве, вращать их, изменять форму, а также изменять положение камеры, из которой ведётся наблюдение.
- **Работа с цветом.** OpenGL предоставляет программисту возможность работы с цветом в режиме RGBA (красный-зелёный-синий-альфа) или использовать индексный режим, когда цвет выбирается из палитры.
- **Удаление невидимых линий и поверхностей. Z-буферизация.**
- **Двойная буферизация.** OpenGL предоставляет как одинарную, так и двойную буферизацию. Двойная буферизация используется для того, чтобы устранить мерцание при мультипликации, т.е. изображение каждого кадра сначала рисуется во втором

(невидимом) буфере, а потом, когда кадр полностью нарисован, весь буфер отображается на экране.

- **Наложение текстуры.** Позволяет придавать объектам реалистичность. На объект, например, шар, накладывается текстура (просто какое-то изображение), в результате чего наш объект теперь выглядит не просто как шар, а как разноцветный мячик.
- **Сглаживание.** Сглаживание позволяет скрыть ступенчатость, свойственную растровым дисплеям. Сглаживание изменяет интенсивность и цвет пикселей около линии, при этом линия смотрится на экране без всяких зигзагов.
- **Освещение**. Позволяет задавать источники света, их расположение, интенсивность и т.д.
- **Атмосферные эффекты.** Например, туман, дым. Всё это позволяет придать объектам или сцене реалистичность, а также "почувствовать" трёхмерное изображение.
- **Прозрачность объектов.**
- **Использование списков изображений.**

1.2. Дополнительные библиотеки OpenGL

Несмотря на то, что библиотека OpenGL (сокращённо GL) предоставляет практически все возможности для моделирования и воспроизведения трёхмерных сцен, некоторые из функций, которые требуются при работе с графикой, напрямую отсутствуют в стандартной библиотеке OpenGL. Например, чтобы задать положение и направление камеры, с которой будет наблюдаться сцена, нужно самому рассчитывать модельную матрицу, а это далеко не так просто. Поэтому для OpenGL существуют так называемые вспомогательные библиотеки.

Первая из этих библиотек называется GLU. Она уже стала стандартом и поставляется вместе с главной библиотекой OpenGL. В состав этой библиотеки вошли более сложные функции. Также в библиотеку вошли функции для работы со сплайнами, реализованы дополнительные операции над матрицами и дополнительные виды проекций.

Следующая библиотека, широко используемая, – это GLUT. Это независимая от платформы библиотека. Она реализует не только дополнительные функции OpenGL, но и предоставляет функции для работы с окнами, клавиатурой и мышкой. Для того чтобы работать с OpenGL в конкретной операционной системе (например, Windows или X Windows), надо провести некоторую предварительную настройку, которая зависит от конкретной операционной системы. С библиотекой GLUT всё намного упрощается, буквально несколькими командами можно определить окно, в котором будет работать OpenGL, определить прерывание от клавиатуры или мышки, и всё это не будет зависеть от операционной системы. Библиотека предоставляет также некоторые функции, с помощью которых можно определять сложные фигуры, такие, как конусы, тетраэдры, и даже можно с помощью одной команды определить чайник!

Есть ещё одна библиотека, похожая на GLUT, которая называется GLAUX. Это библиотека разработана фирмой Microsoft для операционной системы Windows. Она похожа на библиотеку GLUT, но немного отстаёт от неё по своим возможностям. И ещё один недостаток заключается в том, что библиотека GLAUX предназначена только для Windows, в то время как GLUT поддерживает разные операционные системы.

Существуют и другие дополнительные библиотеки для OpenGL. Все они добавляют что-то своё или ориентированы на какую-то платформу. Например, существует такая библиотека, как GLX, – это расширение для X Windows, позволяющее использовать в X Windows

OpenGL. GLX предоставляет не только локальный рендеринг, но и рендеринг по сети.

1.3. Альтернативы OpenGL

Хотя библиотека OpenGL и считается одной из лучших библиотек, как для профессионального применения, так и для игр, у неё существуют и конкуренты.

Одним из главных конкурентов считается Direct3D из пакета DirectX, разработанный фирмой Microsoft. Direct3D создавался исключительно для игровых приложений. Если сравнивать эти две библиотеки, то нельзя сказать, что одна из них лучше, а другая хуже, у каждой библиотеки имеются свои особенности. Например, если сравнивать их в плане переносимости программ с одной платформы на другую, то Direct3D будет работать только на Intel платформах под управлением операционной системы Windows, а программы, написанные с помощью OpenGL, можно успешно перенести на такие платформы как Unix, Linux, SunOS, IRIX, Windows, MacOS и многие другие. В плане объектно-ориентированного подхода OpenGL уступает Direct3D. OpenGL работает по принципу конечного автомата, переходя из одного состояния в другое, совершая при этом какие-то преобразования. Ещё одним преимуществом Direct3D является поддержка дешёвого оборудования. OpenGL поддерживается не на всех графических картах, но для профессиональных ускорителей OpenGL является стандартом де-факто. И ещё, OpenGL легче, чем Direct3D, для изучения основ графики. OpenGL можно применять, например, для начального изучения трёхмерной графики.

GLide до недавнего времени тоже являлся довольно широко используемым стандартом для игровых приложений. Этот стандарт ввела фирма 3Dfx и библиотека GLide создавалась исключительно для видеоускорителей фирмы 3Dfx Voodoo и оптимизирована исключительно под них. Стандарт GLide более низкого уровня, чем OpenGL, и по

своим командам похож на него. GLide мало, чем отличается от OpenGL по своим возможностям, за исключением некоторых функций, которые специально предназначались для Voodoo ускорителей. Но, к сожалению, 3Dfx отказалась от этого стандарта, передав его разработчикам открытого программного обеспечения.

Итак, OpenGL представляет единый стандарт для разработки трёхмерных приложений, сочетает такие качества, как мощь и в то же время простоту. Мультиплатформенность позволяет без труда переносить программное обеспечение с одной операционной системы в другую. OpenGL предоставляет в распоряжение всю мощь аппаратных возможностей, которые имеются на данном компьютере, и при написании программ не нужно будет беспокоиться о конкретных деталях используемого оборудования, за вас побеспокоится драйвер OpenGL. OpenGL прекрасно подходит как для профессионалов, так и для новичков в области компьютерной графики.

2. Возможности OpenGL

2.1. Геометрические примитивы

2.1.1. Точки

Знакомство с работой графики на любой компьютерной системе, как правило, начинается с точек. Точка является мельчайшим элементом на мониторе и может иметь любой доступный цвет. Такова простейшая компьютерная графика: нарисовать точку где-нибудь на экране и задать ей определенный цвет. Затем с помощью языков программирования можно нарисовать линии, полигоны, окружности и другие различные фигуры.

Вывод графики в OpenGL имеет существенное отличие. Больше нет связи с физическими координатами

экрана, теперь все действия производятся в системе координат воображаемого трехмерного пространства. OpenGL сам решает, как спроецировать созданное трехмерное пространство на двухмерную картинку монитора.

Для изображения точки в трехмерном пространстве используется функция *glVertex,* без сомнения, самая используемая функция в OpenGL, которая является «наименьшим общим знаменателем» для всех примитивов OpenGL. Функция *glVertex* может принимать от двух до четырех параметров любого численного типа, от *byte* до *double.* На рис.1 представлен пример прорисовки точки с координатами (50, 50, 0):

glVertex (50.0f, 50.0f, 0.0f);

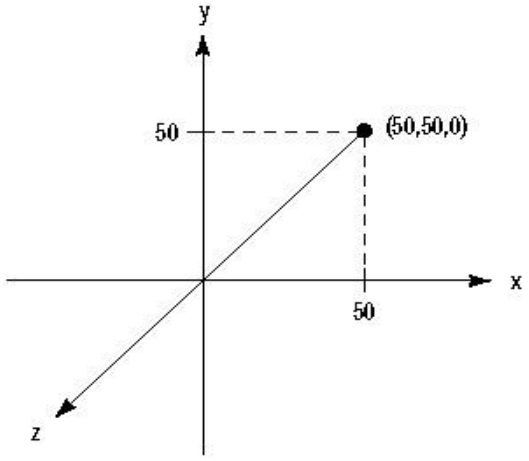


Рис. 1

Для прорисовки нескольких точек используется комбинация функций *glBegin (GLenum mode*) и *glEnd* (). Параметр *mode*

может принимать 10 значений: *GL_POINTS, GL_LINES, GL_LINE_STRIP, GL_LINE_LOOP, GL_TRIANGLES, GL_ TRIANGLE_STRIP, GL_TRIANGLE_FAN, GL_QUADS, GL_QUAD_STRIP* и *GL_POLIGON*. Часть из них будет рассмотрена ниже.

2.1.2. Линии

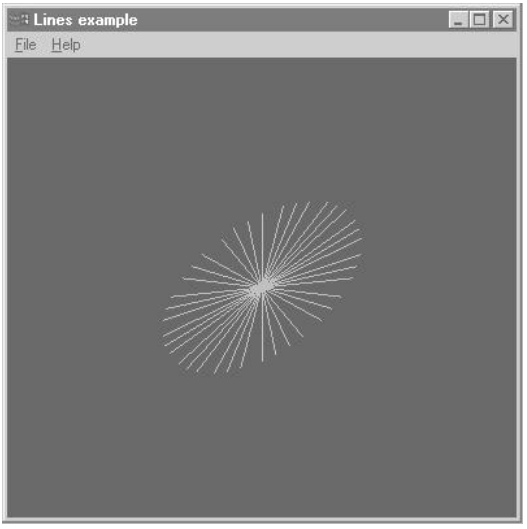
Для представления линии воспользуемся примитивом GL_LINES. Следующий кусок кода рисует линию между двумя точками (0, 0, 0) и (50, 50, 50):

```
glBegin (GL_LINES);

    glVertex3f (0.0f, 0.0f, 0.0f);

    glVertex3f (50.0f, 50.0f, 50.0f);

glEnd ();
```

Приведем фрагмент кода, который демонстрирует многократную прорисовку линий, в данном случае линии являются диаметрами окружности:

```
// Call only once for all remaining points
   glBegin (GL_LINES);
   // All lines lie in the xy plane.
   z = 0.0f;
   for (angle = 0.0f; angle <= GL_PI*3.0f; angle + = 0.5f)
         {
         // Top half of the circle
         x = 50.0f*sin (angle);
         y = 50.0f*cos (angle);
         glVertex3f(x, y, z);      // First end point of line
```

```
        // Bottom half of the circle
        x = 50.0f*sin (angle+3.1415f);
        y = 50.0f*cos (angle+3.1415f);
        glVertex3f(x, y, z);       // Second end point of line
        }

 // Done drawing points
glEnd ();
```



Если мы воспользуемся примитивом GL_LINE_STRIP, то линии будут прорисовываться от точки к точке, образуя непрерывный сегмент. Этот код рисует две линии в плоскости *XY* по трем точкам.

```
glBegin (GL_LINE_STRIP);
    glVertex3f (0.0f, 0.0f, 0.0f);     // V0
    glVertex3f (50.0f, 50.0f, 0.0f);   // V1
    glVertex3f(50.0f, 100.0f, 0.0f);  // V2
glEnd();
```

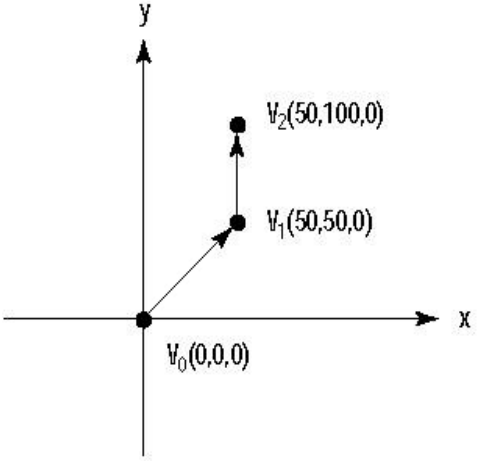


Рис. 2

Последний примитив, основанный на линиях, – GL_LINE_LOOP. Он практически идентичен предыдущему примитиву, за исключением того, что начальная и конечная точки также соединяются линией.

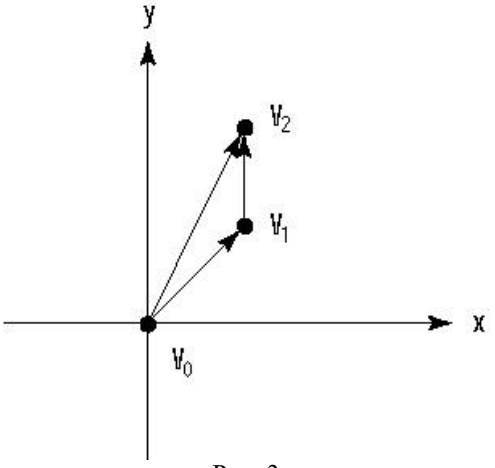


Рис. 3

Также можно задавать различную ширину линий. Это делается с помощью функции:

void glLineWidth (GLfloat width);

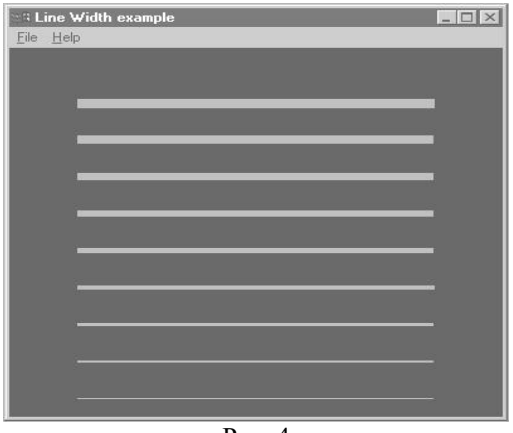


Рис. 4

Параметр *width* задает ширину линии. По умолчанию этот параметр равен 1.0. В дополнение к ширине линий можно создавать пунктирные линии с помощью функций

Void glEnable (Glenum cap); Void glDisable (Glenum cap);

Эти функции разрешают или отменяют определенные возможности *OpenGL*. Возможности задаются строковой константой *cap*, которая может принимать множество значений. В частности, для создания пунктирных линий используется следующий параметр:

glEnable (GL_LINE_STIPPLE);

После этого вызывается функция *glLineStipple*, которая задает стиль прорисовки линий:

void glLineStipple(GLint factor, GLushort pattern);

Рис. 5

Параметр *pattern* задается 16-битным значением и определяет шаблон прорисовки. Каждый бит представляет часть сегмента линии, который имеет значения 1 или 0 (то есть прорисовывается или нет). По умолчанию, каждый бит отвечает за одну точку, но параметр *factor* служит как коэффициент для увеличения ширины шаблона. Например, параметр *factor*, равный 5, означает, что на каждый бит шаблона приходится 5 точек.

Pattern = 0X00FF=255

0 0 F F

Binary= 0000 0000 1111 1111

Line pattern=■■■■■■■■□□□□□□□□

Line= _____ .

2.1.3. Треугольники

Теперь понятно, как строятся точки и линии и даже как рисуются замкнутые полигоны с помощью примитива GL_LINE_LOOP. С помощью только этих примитивов можно создавать любой объект в трехмерном пространстве. Например, можно создать 6 квадратов и расположить их так, чтобы они образовали куб.

Однако можно заметить, что любой предмет, созданный этими примитивами, не может быть закрашен,

ведь мы рисуем только линии. Фактически в предыдущем примере рисуется каркасный куб. Для того чтобы нарисовать полный куб, надо иметь нечто большее, нежели только точки и линии, нам необходимы полигоны. Полигон – это замкнутая форма, которая имеет возможность закраски выбранным цветом, и он является основным элементом построения объектов в OpenGL.

Простейшим полигоном является треугольник. Для построения треугольников используется примитив GL_TRIANGLES, который попросту соединяет три точки между собой.

```
glBegin (GL_TRIANGLES);
      glVertex2f(0.0f, 0.0f);       // V0
      glVertex2f(25.0f, 25.0f);     // V1
      glVertex2f(50.0f, 0.0f);      // V2

      glVertex2f (-50.0f, 0.0f);    // V3
      glVertex2f (-75.0f, 50.0f);   // V4
      glVertex2f (-25.0f, 0.0f);    // V5
glEnd ();
```

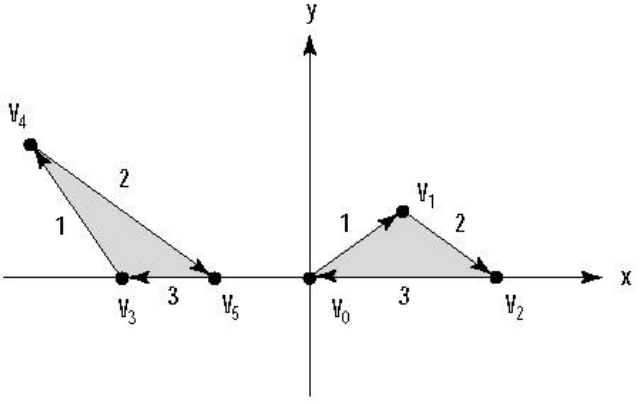


Рис. 6

Для многих предметов и поверхностей нам может потребоваться несколько соединенных треугольников. Для этого используется примитив GL_TRIANGLE_STRIP.

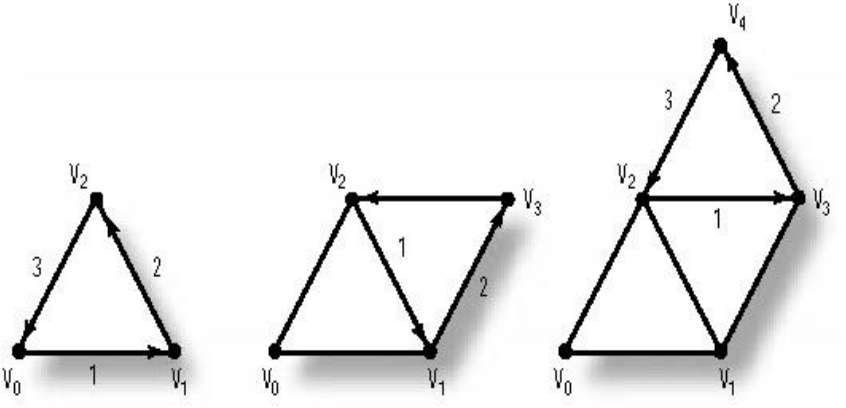


Рис. 7

2.1.4. Четырехугольники

Следующая более сложная форма примитивов – четырехугольники. Для их отображения используется GL_QUADS примитив.

Так же, как и для треугольников, мы можем задать комбинацию соединенных четырехугольников с помощью GL_QUAD_STRIP. На рисунке показан пример построения двух соединенных квадратов, для которых необходимо задать 6 точек.

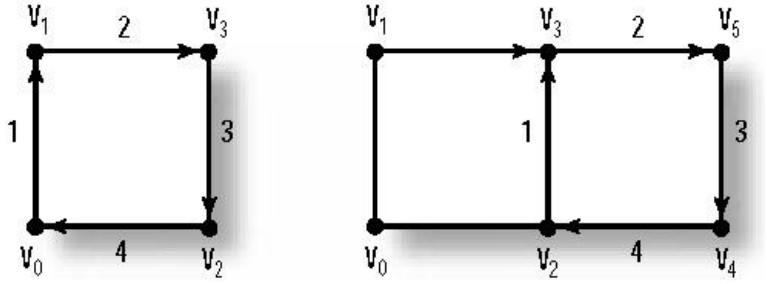


Рис. 8

2.1.5. Полигоны

Последним примитивом OpenGL является GL_POLYGON, который может быть использован для создания полигонов, имеющих произвольное количество сторон.

При использовании полигонов для построения сложных поверхностей нужно помнить два правила: полигон должен быть плоским и выпуклым.

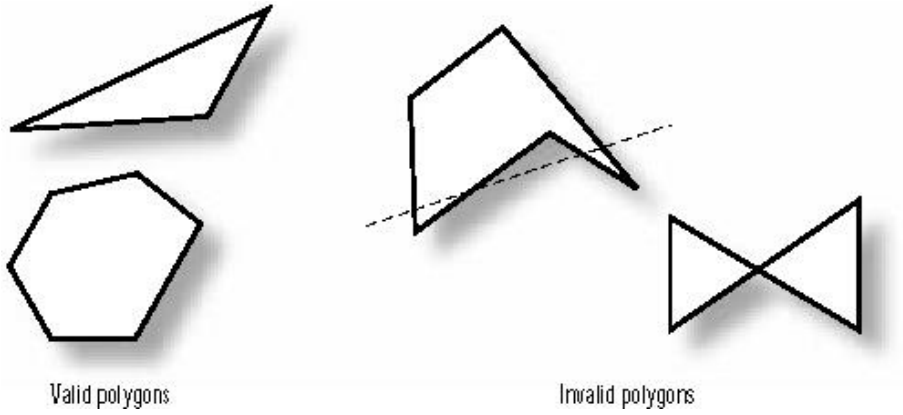


Рис. 9

2.2. ПРЕОБРАЗОВАНИЕ КООРДИНАТ

Рассмотрим три основных функции преобразования систем координат.

Void glTranslatef (GLfloat x, GLfloat y, GLfloat z);

Эта функция задает перемещение объекта в пространстве на вектор (*x, y, z*).

glRotatef (GLfloat x, GLfloat y, GLfloatf z);

Функция *glRotatef* задает поворот вокруг оси вектора (*x, y, z*) на угол *angle.* Вращение производится против часовой стрелки, а угол задается в градусах.

glScalef (GLfloat x, GLfloat y, GLfloat z);

Функция *glScalef* задает коэффициенты изменения координат вдоль осей *x, y, z.* При $x = y = z$ это фактически изменение масштаба.

2.3. РАБОТА С ЦВЕТОМ

Рассмотрим одну из функций glColor:

Void glColor4f (GLfloat red, GLfloat green, GLfloat blue, GLfloat alpha);

Переменные *red, green* и *blue* отвечают за степень градации красного, зеленого и синего цвета соответственно. Параметр *alpha* отвечает за степень прозрачности и принимает значения от 0 до 1.

2.4. СПЛАЙНЫ и NURBS

2.4.1. Параметрическое представление

В *OpenGL* кривая всегда задается параметрически:

$$x = f(u);$$
$$y = g(u);$$
$$z = h(u);$$

Параметр кривой, который мы обозначили u и его диапазон значений называются областью определения кривой. Поверхности будут описываться двумя параметрами u и v. Рисунок показывает кривую и поверхность, которые заданы параметрически.

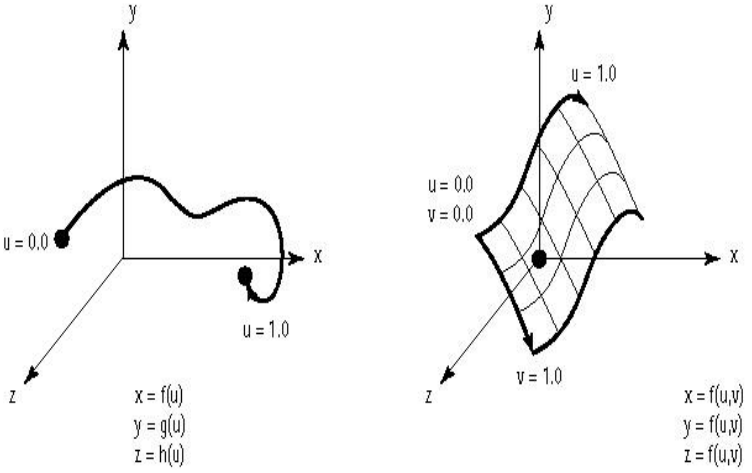


Рис. 10

2.4.2. Контрольные точки

Кривые задаются набором контрольных точек, которые задают форму кривой. Для кривых Безье начальная и

конечная контрольные точки принадлежат кривой. Остальные контрольные точки действуют как магниты, «притягивая» кривую к себе. На рисунке показаны примеры с различным количеством контрольных точек.

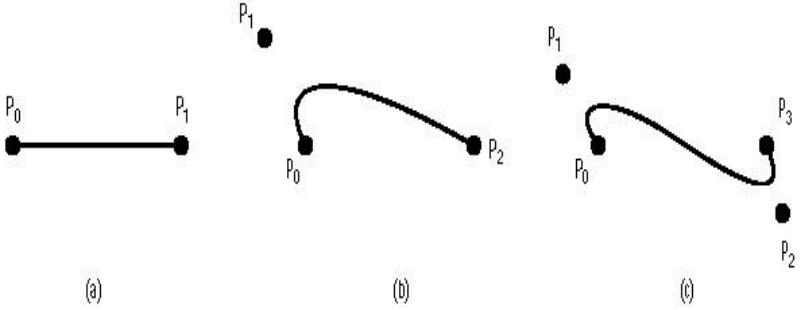


Рис. 11

Порядок кривой представлен числом контрольных точек. Степень кривой на единицу меньше порядка. Математически эти термины имеют отношение к уравнениям, которые точно описывают кривую, для которых порядок – число коэффициентов, а степень – высшая степень многочлена. Напомним, что данная теория строится на интерполяции кривых многочленами. Чем больше задано точек, через которые проходит кривая, тем многочленом более высокого порядка она может быть описана. Так, для трех точек (все нижесказанное верно для любого конечного числа точек) в случае их точного совпадения с кривой мы получим систему:

$$\begin{cases} y_1 = a_1 x_1^2 + a_2 x_1 + a_3 \\ y_2 = a_1 x_2^2 + a_2 x_2 + a_3 \\ y_3 = a_1 x_3^2 + a_2 x_3 + a_3 \end{cases}$$

Так как точки (x_1, y_1), (x_2, y_2), (x_3, y_3) нам известны, то получаем три уравнения с тремя неизвестными. Такие системы легко решаются методами аналитической алгебры. Итак, кривая на рисунке 11b имеет степень 2, а кривая 11c – степень 3. Рекомендуется интерполяция кривыми третьей степени. Теоретически мы можем определить кривую любого порядка, но кривые высших порядков начинают «осциллировать непредсказуемо» и сильно изменяются при мельчайшем сдвиге контрольной точки.

2.4.3. Непрерывность

Рассмотрим соединение двух соседних кривых. Возможны четыре случая.

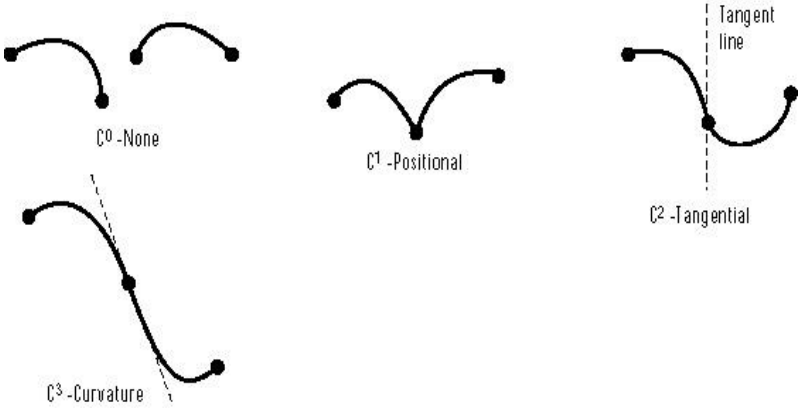


Рис. 12

Как видно из рисунка 12, непрерывности нет (C^0), если концы кривых не совпадают. Позиционная непрерывность достигается, когда концы совпадают (C^1). Тангенциальная непрерывность (C^2) наблюдается, когда непрерывна производная. И, наконец, гладкая непрерывность (C^3)

означает, что вторая производная кривой также непрерывна. Когда рассчитываются составные поверхности или кривые, мы должны стараться использовать C^2 и C^3 непрерывности. Чтобы получить желаемую непрерывность кривых и поверхностей, некоторые параметры интерполяционных функций должны быть выбраны должным образом.

2.4.4. Вычислители

В OpenGL содержится несколько функций (evaluators), которые рисуют Безье кривые и поверхности, заданные своими контрольными точками и соответствующими диапазонами *u* и *v* параметров. Мы не будем останавливаться на этих функциях, поскольку они хорошо описывают только кривые и поверхности, заданные небольшим количеством контрольных точек.

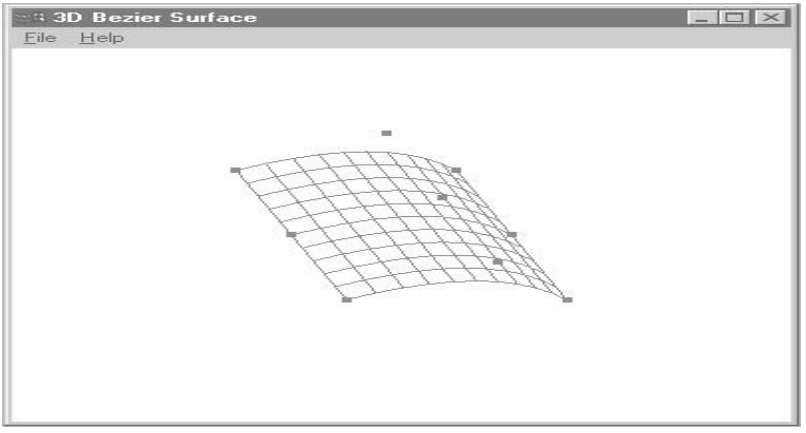


Рис. 12

2.4.5. Nurbs

Можно использовать функции (evaluators) для оценки Безье поверхностей любой степени, но при большом количестве контрольных точек становиться сложно создавать кривые «хорошей» непрерывности. В этом случае лучше воспользоваться функциями более высокого уровня NURBS из библиотеки glu. NURBS (Non Uniform Rational B-Spline) – это рациональный би-сплайн (B-spline), задаваемый на равномерной сетке и являющийся стандартным для компьютерной графики способом определения параметрических кривых и поверхностей. Фактически би-сплайны – это общая форма кривых и поверхностей, которые могут порождать кривые и поверхности Безье и других видов. Они позволяют менять влияние контрольных точек, для того чтобы создавать более гладкие кривые и поверхности с большими наборами контрольных точек.

2.4.6. От кривых Безье к би-сплайнам

Кривые Безье задаются двумя закрепленными крайними точками и любым конечным количеством других контрольных точек, которые влияют на форму кривой. Три кривых Безье, изображенные на рис. 13 имеют 3, 4 и 5 контрольных точек соответственно. Кривая касается прямой, которая соединяет крайние точки с соседними. Если для первой и второй кривых наблюдается гладкая непрерывность (вида C^3), то для большего количества контрольных точек непрерывность начинает «ухудшаться», так как дополнительные контрольные точки «оттягивают» на себя кривую.

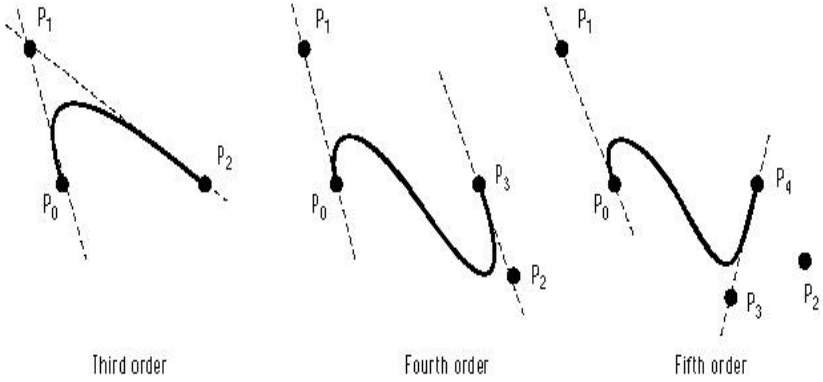


Рис. 13

Би-сплайны, или кубические би-сплайны, работают подобно кривым Безье, но кривая разбивается на сегменты. На форму каждого сегмента влияют только ближайшие четыре контрольные точки. Сегменты склеиваются, и в итоге получается кривая, сходная по характеристикам с кривой Безье четвертого порядка. Таким образом, длинную кривую с большим набором контрольных точек можно сделать гладкой (C^3), если мы будем склеивать «гладко» соседние сегменты. Это также означает, что кривая теперь не обязана проходить через контрольные точки.

2.4.7. Узлы

Реальная же сила NURBS состоит в том, что мы можем изменять влияние четырех контрольных точек для любого заданного сегмента кривой, чтобы достигать желаемую гладкость. Этот контроль обеспечивает последовательность величин называемых knots. Для каждой точки определены две knot величины. Диапазоны knots

совпадают с диапазонами параметров u и v и должны быть неубывающими. Это происходит потому, что величины knots определяют влияние контрольных точек на параметрических диапазонах *u*/*v*. На рисунке показана кривая, демонстрирующая влияние контрольных точек на суммарную кривую. Видно, что точки (в данном случае точки из области параметра *u*) в середине имеют больший вес, и вообще только точки на интервале [0, 3] имеют ненулевой вес, и, следовательно, только они оказывают влияние на результирующую кривую.

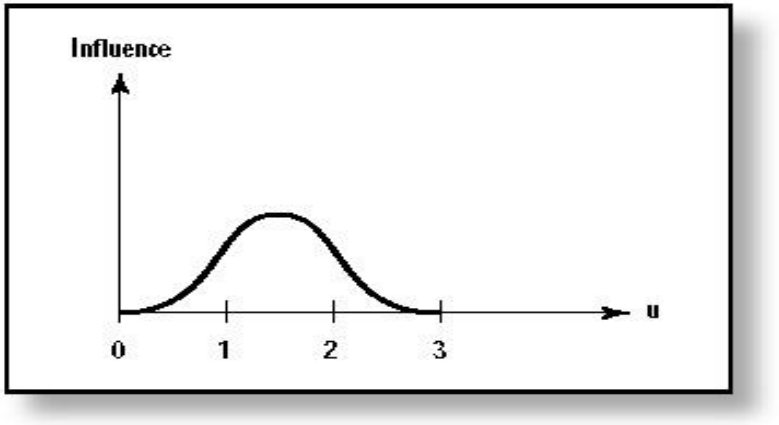


Рис. 14

Главный смысл заключается в том, что такие кривые существуют для каждой контрольной точки в каждом из параметрических направлении. Knots же в данном случае, фактически задают диапазон влияния конкретной контрольной точки.

3. Оболочка визуализации полей трехмерных излучений

Разработана пилотная версия программной оболочки для визуализации нейтронных и фотонных полей, полученных с помощью программы MCNP4B, для решения широкого круга задач переноса ядерных излучений. Программный пакет позволяет визуализировать различные трехмерные функционалы полей излучений, отображать интерактивно различные физические характеристики этих полей (поток, доза, энергия и т.д.), имеется возможность визуализации полей в различных трехмерных объектах с помощью создания эффекта полупрозрачности. Также есть возможность интерактивно манипулировать объектом (изменение размеров, перемещение, вращение и т.д.). Для написании программы использовалась среда MS Visual C++, а также библиотеки OpenGL, MFC (Microsoft Foundation Classes). Был выбран проект mfc appwizard на основе спецификации SDI (Single-Document Interface – интерфейс программ, обеспечивающих единовременную обработку только одного документа). Основные объекты и классы являются открытыми для возможности дальнейшего усовершенствования.

3.1. Составные части

Как уже сказано, для написания программы визуализации трехмерных полей ядерных излучений было создано MFC приложение. На скриншоте мы видим стандартный интерфейс MFC приложения. Для создания дополнительных разделов меню использовались стандартные средства MS Visual Studio. Из основных возможностей меню стоит отметить перемещение в пространстве и изменение масштаба всей сцены.

Для построения детали, в которой распространяется излучение, использовалось свойство прозрачности. Это позволяет одновременно представить общую картину эксперимента и видеть результирующее распределение

внутри детали. Для построения распределения (то есть поверхности) применялся NURBs (см. п. 2. 4. 5. и Приложение п. 4.). Была выбрана каркасная модель прорисовки, поскольку она показалась более наглядной. Есть возможность менять уровень разбиения и цвет закраски.

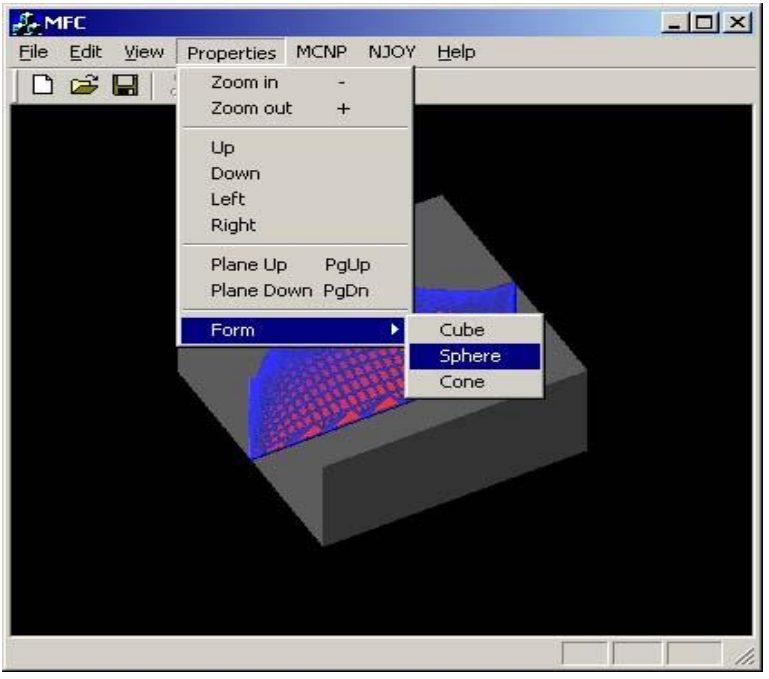


Рис. 15

Также имеется возможность определения величины функционала по выбранной точке на плоскости среза. Для создания радиационной карты использовались примитивы GL_QUAD_STRIP (см. п. 2.1.4), вершинам которых присваивались значения

между красным и синим цветом пропорционально значениям распределения в этих вершинах. Точка изображается с помощью примитива glVertex (см. п. 2.1.1).

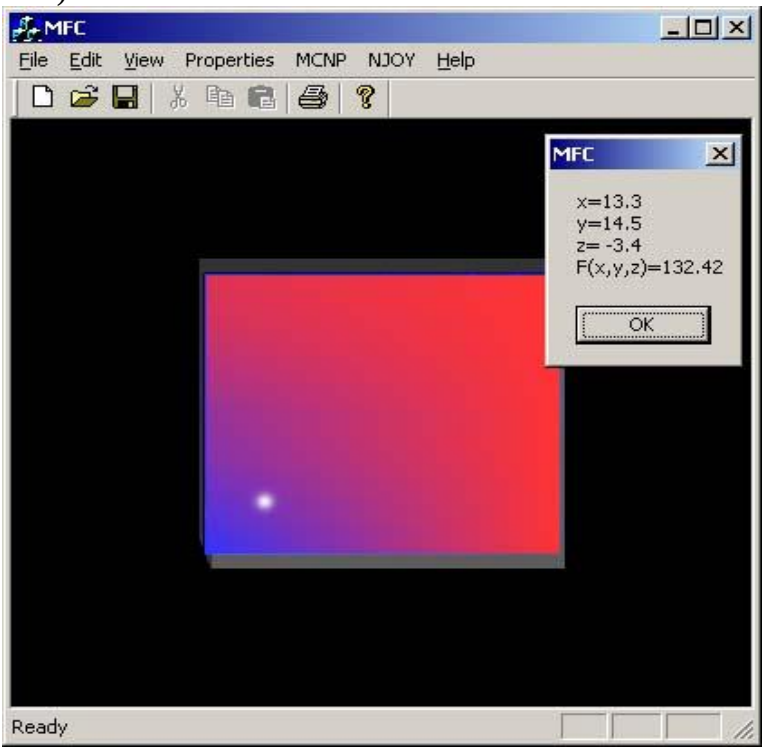


Рис. 16

Ниже представлена реализация функции изменения масштаба. Для этого использовалась функция *glScalef(mm, mm, mm*) (см. гл. 7 [5]) с пропорциональным изменением масштаба по осям.

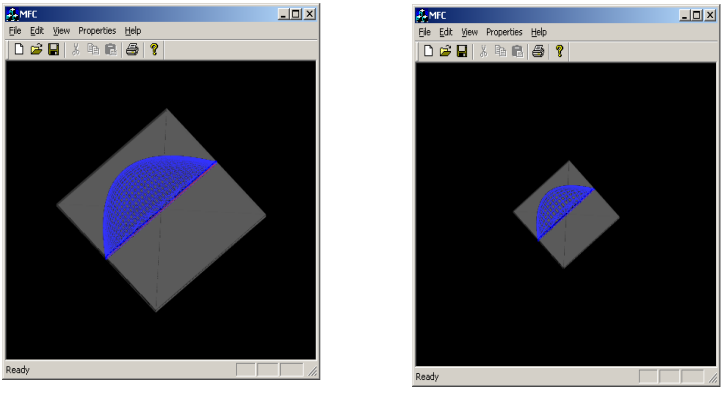


Рис. 17

Теперь рассмотрим эффект перемещения. Для этого использовалась функция glTranslatef(x,y,z) (см. гл. 7 [5]).

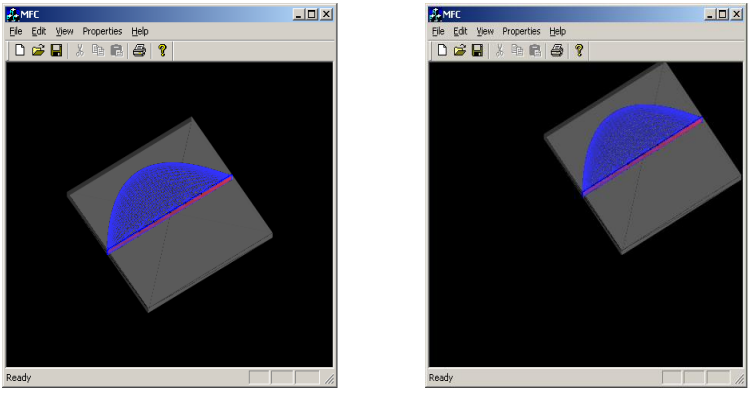
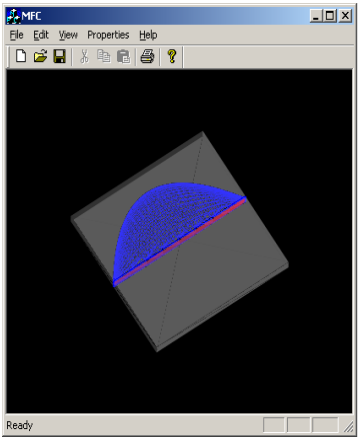


Рис. 18

Эффект перемещения плоскости среза вдоль своей нормали.



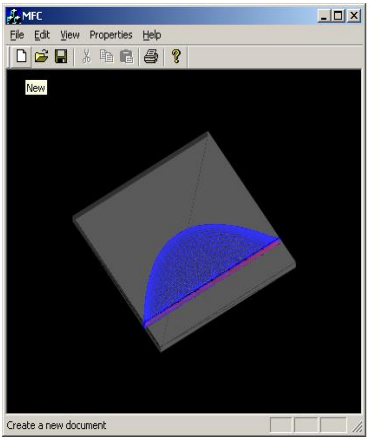
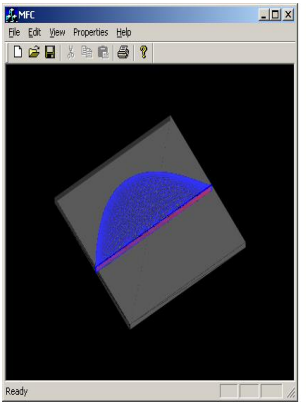


Рис.19

Опция вращения сцены. Если для трех предыдущих опций использовался обработчик нажатий клавиатуры, то для этой использовался обработчик мыши, плюс комбинация функций *glRotatef* (см. гл. 7 [5]).



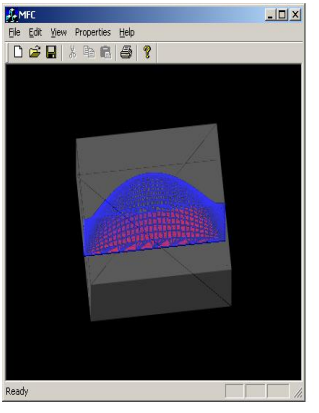


Рис. 20

Для включения эффекта полупрозрачности использовалась функция (см. [6]):

glBlendFunc (GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);

При прорисовке полупрозрачного объекта в функции, задающей цвет объекта, наряду с тремя градациями цвета задается четвертый параметр, отвечающий за «степень полупрозрачности» объекта:

glColor4d(1, 1, 1, 0.20).

3.2. Перспективы

Сейчас для создания геометрии эксперимента используются только простейшие детали из стандартного набора библиотеки *glut.lib*. Понятно, что для создания более сложной геометрии потребуется использование какого-нибудь CAD пакета. Это позволит сравнительно легко задать геометрию для эксперимента любой сложности.

Также в планах ввести сравнительную характеристику для двух различных распределений.

4. Приложение

4.1. Пример создания приложения под MFC

Создаем проект:

- Запустите *MS Visual C++* 6.0.
- Щелкните меню *File->New->MFC AppWizard*(exe).
- Выберите каталог, и имя проекта задайте *mfc*, щелкните OK.

- Step1: Поставьте переключатель на *Single document*, далее OK.
- Step3: Уберите флажок *ActiveX Controls*, далее OK.
- Щелкните *Finish.*
- Щелкните *Build->Set Active Configuration* и установите тип проекта *MFC - Win32 Release.*
- Далее щелкаете *Project->Settings->Link->Object/library modules*: и добавьте *opengl32.lib*, *glu32.lib* и *glut.lib*

В *CMFCView* объявите закрытую (*private*) переменную *hGLRC* типа *HGLRC*. Там же объявите функцию *int SetWindowPixelFormat(HDC*) и открытую (*public*) функцию *display*. Вот что должно получиться:

```
 class CMFCView: public CView
{
private:
CClientDC *pdc;
HGLRC hGLRC;
int SetWindowPixelFormat(HDC);

public:
      void display();
...
```

Вставьте код этой функции в файл *MFCView.cpp*. Код возьмите из предыдущего раздела. Отредактируйте функцию *CMFCView::PreCretaeWindow* следующим образом:

```
BOOL CMFCView::PreCreateWindow(CREATESTRUCT&
cs)
{
```

```
        // TODO: Modify the Window class or styles here by
modifying
        //  the CREATESTRUCT cs
        cs.style |= (WS_CLIPCHILDREN |
WS_CLIPSIBLINGS);
        return CView::PreCreateWindow(cs);
}
```

Добавьте также функцию *display* сюда:

```
void CMFCView::display()
{
  glClear ( GL_COLOR_BUFFER_BIT |
GL_DEPTH_BUFFER_BIT );

        glTranslated (0.01,0,0);
        glColor3d (1,0,0);
        auxSolidSphere ( 1 );


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

Теперь запустите *View->Class Wizard* и добавьте обработчик *WM_CREATE,WM_DESTROY* и *WM_SIZE* в класс *CMFCView*. Отредактируйте их следующим образом:

```
int CMFCView::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
    if (CView::OnCreate(lpCreateStruct) ==  – 1)
        return – 1;

    pdc = new CClientDC(this);
```

```
        if(SetWindowPixelFormat(pdc->m_hDC)==FALSE)
        return - 1;

 hGLRC = wglCreateContext(pdc->m_hDC);
    if(hGLRC == NULL)
        return  - 1;

    if(wglMakeCurrent(pdc->m_hDC, hGLRC)==FALSE)
        return  - 1;

    glEnable(GL_ALPHA_TEST);
    glEnable(GL_DEPTH_TEST);
    glEnable(GL_COLOR_MATERIAL);
    glEnable(GL_LIGHTING);
    glEnable(GL_LIGHT0);
    glEnable(GL_BLEND);
    glBlendFunc(GL_SRC_ALPHA,
GL_ONE_MINUS_SRC_ALPHA);

 float pos[4] = {3, 3, 3, 1};
    float dir[3] = {- 1, - 1, - 1};
    glLightfv(GL_LIGHT0, GL_POSITION, pos);
    glLightfv(GL_LIGHT0, GL_SPOT_DIRECTION, dir);
   return 0;
}

void CMFCView::OnDestroy()
{
 if(wglGetCurrentContext()!=NULL)
   wglMakeCurrent(NULL, NULL) ;

 if(hGLRC!=NULL)
        {
```

```
    wglDeleteContext(hGLRC);
    hGLRC = NULL;
   }
 delete pdc;
 CView::OnDestroy ();
}

void CMFCView::OnSize (UINT nType, int cx, int cy)
{
      CView::OnSize (nType, cx, cy);
   glViewport (0, 0, cx, cy);
  glMatrixMode( GL_PROJECTION );
  glLoadIdentity ();
  glOrtho ( - 5, 5, - 5, 5, 2,12);
  gluLookAt ( 0, 0, 5, 0, 0,  0, 0, 1,0 );
  glMatrixMode ( GL_MODELVIEW );

  }
```

В *StdAfx.h* включите заголовочные файлы после строчки
#include <afxext.h>

```
#include <gl/gl.h>
#include <gl/glu.h>
#include <gl/glut.h>
```

Запустите еще раз *Class Wizard* и добавьте функцию *OnIdle* в класс *CMFCApp*. Эта функция вызывается всякий раз, когда очередь сообщений окна пуста. Отредактируйте ее:

```
BOOL CMFCApp::OnIdle (LONG lCount)
{
 ( (CMFCView*) ((CMainFrame*) m_pMainWnd)-
>GetActiveView () )->display ();
```

```
    return 1;//CWinApp::OnIdle(lCount);
}
```

После компиляции и запуска программы получаем красную сферу, которая медленно движется слева направо.

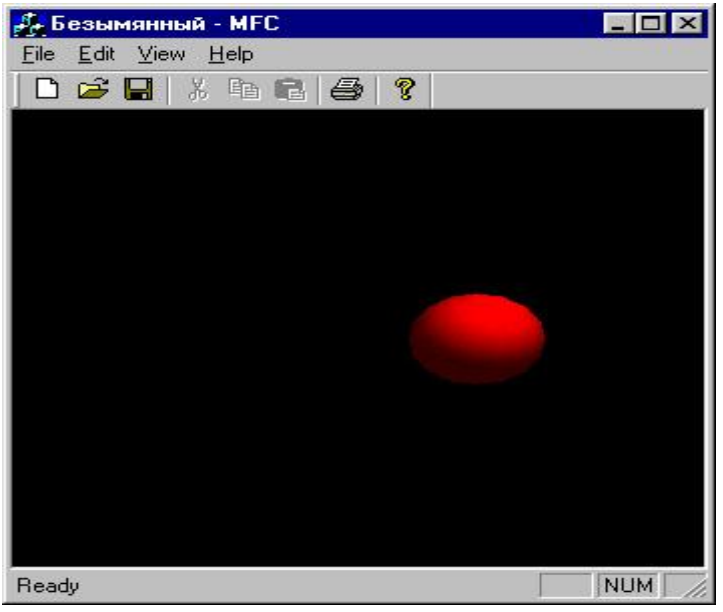


Рис. 21

4.2. Построение *NURBS* поверхности

Приведем фрагмент кода на языке C++ для построения *NURBS* поверхности.

```
// создаем указатель на NURBS объект
GLUnurbsObj *pNurb = NULL;
...
```

```
...

// создаем NURBS объект
   pNurb = gluNewNurbsRenderer();

...
// здесь идет основной кусок кода
...
...

// Удаление NURBS объекта, если он был создан
If (pNurb)
      gluDeleteNurbsRenderer (pNurb);
```

Свойства *NURBS* поверхности.
Раз мы создали *Nurbs* объект, нам необходимо задать различные свойства, такие, как:

```
// толерантность или частота разбиения
gluNurbsProperty (pNurb, GLU_SAMPLING_TOLERANCE,
25.0f);

// Рисует гладкую поверхность (использование
GLU_OUTLINE_POLYGON дает //решетчатую/поверхность
gluNurbsProperty (pNurb, GLU_DISPLAY_MODE,
(GLfloat)GLU_FILL);
```

Задание поверхности
Поверхность задается массивом контрольных точек и *knot* последовательностями. Все эти величины передаются функции *gluNurbsSurface*. Как видно на примере, она должна находиться между функцией *gluBeginSurface* и *gluEndSurface*.

```
// Создание NURB
// Начало определения NURB
gluBeginSurface(pNurb);
```

```
// Прорисовка поверхности

// указатель на  NURBS объект
gluNurbsSurface(pNurb,
// количество knots и массив knots в направлении u
  8, Knots,

// количество knots и массив knots в направлении
  8, Knots,

// Расстояние между элементами массива для контрольных
точек в ‘ u’ направлении
  4 * 3,

// то же самое в ’ v’ направлении
  3,

// Массив контрольных точек
  &ctrlPoints[0][0][0],

// u и v порядки поверхности
  4, 4,

// Вид поверхности
  GL_MAP2_VERTEX_3);

// Закрытие прорисовки
gluEndSurface(pNurb);
```

Мы можем делать несколько обращений к функции gluNurbsSurface и создать любое количество *NURBS* поверхностей, но свойства, которые мы задали в определении объекта, будут иметь тот же эффект. Как правило, так и требуется, в любом случае редко требуются две поверхности

(например, соединенные) разных стилей: одна гладкая, а другая решетчатая.

Используя массив контрольных точек и *knots*, показанных ниже, получим поверхность:

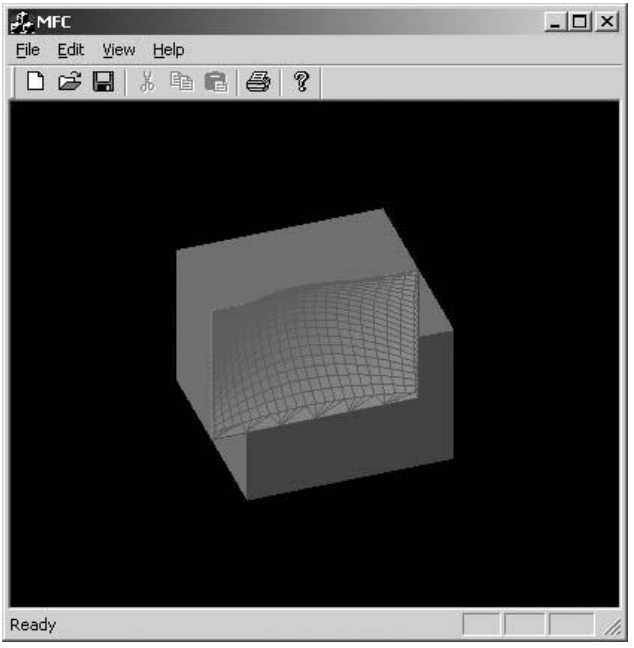


Рис. 22

```
//              u  v (x,y,z)
GLfloat ctrlPoints[4][4][3]={{{-6.0f,-6.0f,0.0f},   // u = 0,   v = 0
                              {-6.0f,-2.0f,0.0f}, //              v = 1
                              {-6.0f, 2.0f,0.0f},  //              v = 2
                              {-6.0f, 6.0f,0.0f}}, //              v = 3

                             {{-2.0f,-6.0f,0.0f},    // u = 1    v = 0
                              {-2.0f,-2.0f,8.0f},  //              v = 1
```

```
 {-2.0f, 2.0f,8.0f},  //            v = 2
{-2.0f, 6.0f,0.0f}}, //            v = 3

 {{ 2.0f,-6.0f,0.0f},  // u =2     v = 0
 { 2.0f,-2.0f,8.0f}, //            v = 1
 { 2.0f, 2.0f,8.0f}, //            v = 2
 { 2.0f, 6.0f,0.0f}},//            v = 3

 {{ 6.0f,-6.0f,0.0f},   // u = 3   v = 0
 { 6.0f,-2.0f,0.0f},  //            v = 1
 { 6.0f, 2.0f,0.0f},  //            v = 2
 { 6.0f, 6.0f,0.0f}}}; //           v = 3

// Последовательность Knots для NURB
GLfloat Knots[8]={0.0f,0.0f,0.0f,0.0f,1.0f,1.0f,1.0f,1.0f};
```

4.3. Задачи для самоподготовки

4.3.1. С помощью примитива *GL_QUAD_STRIP* нарисуйте квадрат, вершины которого имеют красный, зеленый, синий и белый цвет. Должен получиться эффект закраски с плавным перетеканием цветов друг в друга.

4.3.2. Нарисуйте крутящееся колесо. Для этого воспользуйтесь прозрачным цилиндром, с помощью линий создайте спицы. Для вращения используйте функцию *glRotatef*.

4.3.3. Изобразите пароболоид с помощью *Nurbs*. Достаточно сгенерировать массив опорных точек с размерностью [4][4][3]. Поварьируйте толерантность и посмотрите, как это действует на производительность программы.

4.3.4. Нарисуйте красный квадрат. При нажатии мышки на него цвет должен меняться на красный.

Воспользуйтесь примитивом *GL_QUAD_STRIP* и обработчиком мыши.

4.3.5. С помощью *GL_QUAD_STRIP* нарисуйте куб с разноцветными гранями. Примените к нему последовательность горизонтальных и вертикальных поворотов, так чтобы последовательно можно было увидеть все грани и только по одному разу. Затем зациклите этот процесс.

СПИСОК ЛИТЕРАТУРЫ

1. *Тарасов И.А.* Основы программирования OpenGL. – М.: Радио и Связь, 1999.
2. *Wright S.R.., Sweet M.* OpenGL Superbible: The Complete Guide to OpenGL Programming for Windows NT and Windows 95.– Waite Group Press, 1996. –750 с.
3. Microsoft MSDN Library.
4. http://romka.demonews.com/opengl/doc/
5. http://www.opengl.org
6. http://www.opengl.org.ru
7. *Тихомиров Ю.* Программирование трехмерной графики. – СПб.: БХВ – Петербург, 2001.
8. *Хилл Ф.* OpenGL. Программирование компьютерной графики для профессионалов.– СПб.: Питер, 2002.
9. *Краснов М.В.* OpenGL. Графика в проектах DELPHI. – СПб.: БХВ – Петербург, 2001. – 416 с.
10. MCNP – A General Monte Carlo N – particle Transport Mode, LA – 12625 – M Version 4B, Manual, Issued: March 1997.