

Лабораторная работа № 6

Тема: Моделирование 2D объектов с использованием OpenGL.

Цель: Научиться строить графические примитивы средствами OpenGL.

Краткая теория

Возможности OpenGL

OpenGL – Open Graphics Library, открытая графическая стандартная библиотека для программирования трехмерной графики для многих 32-разрядных операционных систем (Windows, Linux в том числе).

В отличие от Direct3D, которая характерна только для Windows, OpenGL содержит в себе более 250 процедур и функций для построения 3D графики и рендеринга. Они находятся в opengl32.dll (Windows\ system\) и в расширении glu32.dll.

К основным возможностям OpenGL можно отнести:

- *геометрические* (точки, линии, полигоны) и *растровые* (битовый массив (bitmap) и образ (image)) *примитивы*;
- *использование В-сплайнов* для рисования кривых по опорным точкам;
- *альфа-канал*. Позволяет делать предметы прозрачными, уровень прозрачности от 0 до 100 %;
- *антиалиасинг (сглаживание)* цветовых переходов для получения более реалистического изображения;
- *буфер аккумулятора*. Дополнительный буфер для спецэффектов и глобального сглаживания по всей сцене;
- *градиентная заливка* полигонов и отрезков;
- *двойная буферизация*. Для устранения мерцания при мультипликации. Изображение каждого кадра сначала рисуется во втором (невидимом) буфере, а потом, когда кадр полностью нарисован, весь буфер отображается на экране;
- *заливка и освещенность фактур*. К фактурам применяются эффекты освещенности и затенения в зависимости от характеристик «материала»;
- *пространственные преобразования*. Масштабирование, вращение и перемещение объектов в пространстве;
- *текстуры (меппинг)* Наложение двухмерных изображений на объемные поверхности для придания сцене реализма;
- *атмосферные эффекты*, такие как туман, дым, дымка делают изображения, созданные компьютером, более реалистичными.

OpenGL позволяет:

- 1 Создавать объекты из геометрических примитивов (точки, линии, грани и битовые изображения).
- 2 Располагать объекты в трёхмерном пространстве и выбирать способ и параметры проецирования.

3 Вычислять цвет всех объектов. Цвет может быть как явно задан, так и вычисляться с учётом источников света, параметров освещения, текстур.

4 Переводить математическое описание объектов и связанной с ними информации о цвете в изображение на экране.

При этом OpenGL может осуществлять дополнительные операции, такие, как удаление невидимых фрагментов изображения.

Основные типы данных OpenGL

Все команды (процедуры и функции) OpenGL начинаются с префикса **gl**, а все константы – с префикса **GL_**. Кроме того, в имена функций и процедур OpenGL входят суффиксы, несущие информацию о числе передаваемых параметров и об их типе. В таблице 1 приводятся вводимые OpenGL типы данных, стандартные типы языка C, которым они соответствуют, и суффиксы, которым они соответствуют.

Некоторые команды OpenGL оканчиваются на букву *v*. Это говорит о том, что команда получает указатель на массив значений, а не сами эти значения в виде отдельных параметров. Многие команды имеют как векторные, так и не векторные версии. Например, конструкции:

```
glColor3f(1.0, 1.0, 1.0);  
и  
GLfloat color[] = {1.0, 1.0, 1.0};  
glColor3fv(color);  
эквивалентны.
```

Таблица 1 – Типы данных OpenGL

Суффикс	Описание	Тип в C	Тип в OpenGL
b	8-битовое целое	char	GLbyte
s	16-битовое целое	short	GLshort
i	32-битовое целое	long	GLint GLsizei
f	32-битовое вещественное число	float	GLfloat, GLclampf
d	64-битовое вещественное число	double	GLdouble, GLclampd
ub	8-битовое беззнаковое целое	unsigned char	GLubyte, GLboolean
us	16-битовое беззнаковое целое	unsigned short	GLushort
ui	32-битовое беззнаковое целое	unsigned long	GLuint, GLenum, GLbitfield

OpenGL можно рассматривать как автомат, находящийся в одном из нескольких состояний. Внутри OpenGL содержится целый ряд переменных, например, текущий цвет или текущий режим закрашивания. Если установить

текущий цвет, то все последующие объекты будут этого цвета до тех пор, пока текущий цвет не будет изменён.

По умолчанию каждая системная переменная имеет своё значение, и в любой момент значение каждой из этих переменных можно узнать. Обычно для этого используется одна из следующих функций: `glGetBooleanv()`, `glGetDoublev()`, `glGetFloatv()` и `glGetIntegerv()`. Для определения значений некоторых переменных служат специальные функции.

Рисование геометрических объектов

Работа с буферами и задание цвета объектов

OpenGL содержит внутри себя несколько различных буферов. Среди них фрейм буфер (где строится изображение), z-буфер, служащий для удаления невидимых поверхностей, буфер трафарета и аккумулирующий буфер.

Для очистки внутренних буферов служит процедура `glClear(GLbitfield mask)`, очищающая буферы, заданные переменной `mask`. Параметр `mask` является комбинацией следующих констант:

`GL_COLOR_BUFFER_BIT` – очистить буфер изображения (фреймбуфер);

`GL_DEPTH_BUFFER_BIT` – очистить z-буфер;

`GL_ACCUM_BUFFER_BIT` – очистить аккумулирующий буфер;

`GL_STENCIL_BUFFER_BIT` – очистить буфер трафарета.

Цвет, которым очищается буфер изображения, задаётся процедурой `glClearColor(GLfloat red, GLfloat green, GLfloat blue, GLfloat alpha)`. Значение, записываемое в z-буфер, при очистке задаётся процедурой `glClearDepth(GLfloat depth)`. Значение, записываемое в буфер трафарета, при очистке задаётся процедурой `glClearStencil(GLint s)`. Цвет, записываемый в аккумулирующий буфер, при очистке задаётся процедурой `glClearAccum(GLfloat red, GLfloat green, GLfloat blue, GLfloat alpha)`.

Сама команда `glClear` очищает одновременно все заданные буферы, заполняя их соответствующими значениями. Для задания цвета объекта служит процедура: `glColor{3 4}{b s i f d u b u s u i}[v](TYPE red, ...)`.

Цифра 3 или 4 указывает на количество требуемых аргументов, а буква, следующая за цифрой, показывает тип аргументов. Например, в процедуру `glColor3i` будут переданы три параметра целого типа.

Если значение параметра не задано, то оно автоматически полагается равным единице. Версии процедуры `glColor`, где параметры являются переменными с плавающей точкой, автоматически обрезают переданные значения в отрезок `[0, 1]`.

Процедура `glFlush()` вызывает немедленное рисование ранее переданных команд. При этом ожидания завершения всех ранее переданных команд не происходит. С другой стороны, команда `glFinish()` ожидает, пока не будут завершены все ранее переданные команды.

Если нужно включить удаление невидимых поверхностей методом z-буфера, то z-буфер необходимо очистить и передать команду

`glEnable(GL_DEPTH_TEST)`. Команду `glEnable()` можно выполнить только один раз при инициализации системных переменных OpenGL. Очистку z-буфера необходимо производить перед началом построения очередного кадра изображения.

Задание графических примитивов

Все геометрические примитивы в OpenGL задаются вершинами (набором чисел, определяющих их координаты в пространстве).

OpenGL работает с однородными координатами (x, y, z, w). Если координата z не задана, то она считается равной нулю. Если координата w не задана, то она считается равной единице.

Под линией в OpenGL подразумевается отрезок, заданный своими начальной и конечной вершинами.

Под гранью (многоугольником) в OpenGL подразумевается замкнутый выпуклый многоугольник с несамопересекающейся границей.

Все геометрические объекты в OpenGL задаются посредством вершин, а сами вершины задаются процедурой:

```
glVertex{2 3 4}{s i f d}[v](TYPE x, ...),
```

где реальное количество параметров определяется первым суффиксом (2, 3 или 4), а суффикс **v** означает, что в качестве единственного аргумента выступает массив, содержащий необходимое количество координат. Например:

```
glVertex2s(1, 2);  
glVertex3f(2.3, 1.5, 0.2);  
GLdouble vect[] = { 1.0, 2.0, 3.0, 4.0};  
glVertex4dv(vect);
```

Для задания геометрических примитивов необходимо как-то выделить набор вершин, определяющих этот объект. Для этого служат процедуры `glBegin()` и `glEnd()`. Процедура `glBegin(GLenum mode)` обозначает начало списка вершин, описывающих геометрический примитив. Тип примитива задаётся параметром `mode`, который принимает одно из следующих значений:

- GL_POINTS – набор отдельных точек;
- GL_LINES – пары вершин, задающих отдельные точки;
- GL_LINE_STRIP – незамкнутая ломаная;
- GL_LINE_LOOP – замкнутая ломаная;
- GL_POLYGON – простой выпуклый многоугольник;
- GL_TRIANGLES – тройки вершин, интерпретируемые как вершины отдельных треугольников;
- GL_TRIANGLE_STRIP – связанная полоса треугольников;
- GL_TRIANGLE_FAN – веер треугольников;
- GL_QUADS – четвёрки вершин, задающие выпуклые четырёхугольники;
- GL_QUAD_STRIP – полоса четырёхугольников.

Процедура `glEnd()` отмечает конец списка вершин.

Между командами `glBegin()` и `glEnd()` могут находиться команды задания различных атрибутов вершин `glVertex()`, `glColor()`, `glNormal()`, `glCallList()`, `glCallLists()`, `glTexCoord()`, `glEdgeFlag()`, `glMaterial()`. Между командами `glBegin()` и `glEnd()` все остальные команды OpenGL недопустимы и приводят к возникновению ошибок. Рассмотрим в качестве примера задание окружности:

```
glBegin(GL_LINE_LOOP);
for (int i = 0; i < N; i++)
{
    float angle = 2 * M_PI * i / N;
    glVertex2f(cos(angle), sin(angle));
}
glEnd();
```

Хотя многие команды могут находиться между `glBegin()` и `glEnd()`, вершины генерируются при вызове `glVertex()`. В момент вызова `glVertex()` OpenGL присваивает создаваемой вершине текущий цвет, координаты текстуры, вектор нормали и т. д. Изначально вектор нормали полагается равным (0, 0, 1), цвет полагается равным (1, 1, 1, 1), координаты текстуры полагаются равными нулю.

Рисование точек, линий и многоугольников

Для задания размеров точки служит процедура `glPointSize(GLfloat size)`, которая устанавливает размер точки в пикселях, по умолчанию он равен единице.

Для задания ширины линии в пикселях служит процедура `glLineWidth(GLfloat width)`. Шаблон, которым будет рисоваться линия, можно задать при помощи процедуры `glLineStipple(GLint factor, GLushort pattern)`.

Шаблон задается переменной `pattern` и растягивается в `factor` раз. Использование шаблонов линии необходимо разрешить при помощи команды `glEnable(GL_LINE_STIPPLE)`. Запретить использование шаблонов линий можно командой `glDisable(GL_LINE_STIPPLE)`.

Многоугольники рисуются как заполненные области пикселей внутри границы, хотя их можно рисовать либо только как граничную линию, либо просто как набор граничных вершин.

Многоугольник имеет две стороны, лицевую и нелицевую, и может быть отрисован по-разному в зависимости от того, какая сторона обращена к наблюдателю. По умолчанию обе стороны рисуются одинаково. Для задания того, как именно следует рисовать переднюю и заднюю стороны многоугольника, служит процедура `glPolygonMode(GLenum face, GLenum mode)`. Параметр `face` может принимать значения `GL_FRONT_AND_BACK` (обе стороны), `GL_FRONT` (лицевая сторона) или `GL_BACK` (нелицевая сторона). Параметр `mode` может принимать значения `GL_POINT`, `GL_LINE` или `GL_FILL`, обозначая, что многоугольник должен рисоваться как набор граничных точек, граничная ломаная линия или заполненная область, например:

```
glPolygonMode(GL_FRONT, GL_FILL);  
glPolygonMode(GL_BACK, GL_LINE).
```

По умолчанию вершины многоугольника, которые появляются на экране в направлении против часовой стрелки, называются лицевыми. Это можно изменить при помощи процедуры `glFrontFace(GLenum mode)`. По умолчанию параметр `mode` равняется `GL_CCW`, что соответствует направлению обхода против часовой стрелки. Если задать этот параметр равным `GL_CW`, то лицевыми будут считаться многоугольники с направлением обхода вершин по часовой стрелке.

При помощи процедуры `glCullFace(GLenum mode)` вывод лицевых или нелицевых граней многоугольников можно запретить. Параметр `mode` принимает одно из значений `GL_FRONT` (оставить только лицевые грани), `GL_BACK` (оставить нелицевые) или `GL_FRONT_AND_BACK` (оставить все грани). Для отсечения граней необходимо разрешить отсечение при помощи команды `glEnable(GL_CULL_FACE)`.

Шаблон для заполнения грани можно задать при помощи процедуры `glPolygonStipple(const GLubyte * mask)`, где `mask` задает массив битов размером 32 на 32.

Для разрешения использования шаблонов при выводе многоугольников служит команда `glEnable(GL_POLYGON_STIPPLE)`.

Вектор нормали для каждой вершины можно задать при помощи одной из следующих процедур:

```
glNormal3{b s i d f}(TYPE nx, TYPE ny, TYPE nz);  
glNormal3{b s I d f}v(const TYPE * v).
```

В версиях с суффиксами `b`, `s` или `i` значения аргументов масштабируются в отрезок `[-1, 1]`.

В качестве примера приведем процедуру, строящую прямоугольный параллелепипед с ребрами, параллельными координатным осям, по диапазонам изменения `x`, `y` и `z`.

```
#include <windows.h>  
#include <gl\gl.h>  
drawBox(GLfloat x1, GLfloat x2, GLfloat y1, GLfloat y2, GLfloat z1,  
GLfloat z2)  
{  
    glBegin ( GL_POLYGON );  
        glNormal3f ( 0.0, 0.0, 1.0 );  
        glVertex3f ( x1, y1, z2 );  
        glVertex3f ( x2, y1, z2 );  
        glVertex3f ( x2, y2, z2 );  
        glVertex3f ( x1, y2, z2 );  
    glEnd ();  
    glBegin ( GL_POLYGON );  
        glNormal3f ( 0.0, 0.0, -1.0 );  
        glVertex3f ( x2, y1, z1 );  
        glVertex3f ( x1, y1, z1 );
```

```

        glVertex3f ( x1, y2, z1 );
        glVertex3f ( x2, y2, z1 );
    glEnd ();
    glBegin ( GL_POLYGON );
        glNormal3f ( -1.0, 0.0, 0.0 );
        glVertex3f ( x1, y1, z1 );
        glVertex3f ( x1, y1, z2 );
        glVertex3f ( x1, y2, z2 );
        glVertex3f ( x1, y2, z1 );
    glEnd ();
    glBegin ( GL_POLYGON );
        glNormal3f ( 1.0, 0.0, 0.0 );
        glVertex3f ( x2, y1, z2 );
        glVertex3f ( x2, y1, z1 );
        glVertex3f ( x2, y2, z1 );
        glVertex3f ( x2, y2, z2 );
    glEnd ();
    glBegin ( GL_POLYGON );
        glNormal3f ( 0.0, 1.0, 0.0 );
        glVertex3f ( x1, y2, z2 );
        glVertex3f ( x2, y2, z2 );
        glVertex3f ( x2, y2, z1 );
        glVertex3f ( x1, y2, z1 );
    glEnd ();
    glBegin ( GL_POLYGON );
        glNormal3f ( 0.0, -1.0, 0.0 );
        glVertex3f ( x2, y1, z2 );
        glVertex3f ( x1, y1, z2 );
        glVertex3f ( x1, y1, z1 );
        glVertex3f ( x2, y1, z1 );
    glEnd ();
}

```

Задание моделей закрашивания

Линия или заполненная грань могут быть нарисованы одним цветом (плоское закрашивание, GL_FLAT) или путём интерполяции цветов в вершинах (закрашивание Гуро, GL_SMOOTH).

Для задания режима закрашивания служит процедура `glShadeModel(GLenum mode)`, где параметр `mode` принимает значение GL_SMOOTH или GL_FLAT.

Освещение

OpenGL использует модель освещённости, в которой свет приходит из нескольких источников, каждый из которых может быть включён или выключен. Кроме того, существует еще общее фоновое (ambient) освещение.

Для правильного освещения объектов необходимо для каждой грани задать материал, обладающий определенными свойствами. Материал может испускать свой собственный свет, рассеивать падающий свет во всех направлениях (диффузное отражение) или подобно зеркалу отражать свет в определенных направлениях.

Пользователь может определить до восьми источников света и их свойства, такие, как цвет, положение и направление. Для задания этих свойств служит процедура:

`glLight{i f}[v](GLenum light, GLenum pname, TYPE param),`

которая задаёт параметры для источника света `light`, принимающего значения `GL_LIGHT0`, `GL_LIGHT1`, ..., `GL_LIGHT7`. Параметр `pname` определяет характеристику источника света, которая задается последним параметром.

Для использования источников света расчёт освещенности следует разрешить командой `glEnable(GL_LIGHTING)`, а применение соответствующего источника света разрешить (включить) командой `glEnable`, например: `glEnable(GL_LIGHT0)`.

Источник света можно рассматривать как имеющий вполне определенные координаты и светящий во всех направлениях или как направленный источник, находящийся в бесконечно удаленной точке и светящий в заданном направлении (x, y, z).

Если параметр w в команде `GL_POSITION` равен нулю, то соответствующий источник света – направленный и светит в направлении (x, y, z). Если же w отлично от нуля, то это позиционный источник света, находящийся в точке с координатами ($x/w, y/w, z/w$).

Заданием параметров `GL_SPOT_CUTOFF` и `GL_SPOT_DIRECTION` можно создавать источники света, которые будут иметь коническую направленность. По умолчанию значение параметра `GL_SPOT_CUTOFF` равно 180° , т. е. источник светит во всех направлениях с равной интенсивностью. Параметр `GL_SPOT_CUTOFF` определяет максимальный угол от направления источника, в котором распространяется свет от него. Он может принимать значение 180° (не конический источник) или от 0 до 90° .

Интенсивность источника с расстоянием – убывает (параметры этого убывания задаются при помощи параметров `GL_CONSTANT_ATTENUATION`, `GL_LINEAR_ATTENUATION` и `GL_QUADRATIC_ATTENUATION`). Только собственное свечение материала и глобальная фоновая освещенность с расстоянием не ослабевают.

Глобальное фоновое освещение можно задать при помощи команды `glLightModel{i f}[v](GL_LIGHT_MODEL_AMBIENT, ambientColor)`.

Местонахождение наблюдателя оказывает влияние на блики на объектах. По умолчанию при расчётах освещённости считается, что наблюдатель находится в бесконечно удалённой точке, т. е. направление источника света на наблюдателя постоянно для любой вершины. Можно включить более реалистичное освещение, когда направление источника

света на наблюдателя будет вычисляться для каждой вершины отдельно. Для этого служит команда:

`glLightModeli(GL_LIGHT_MODEL_LOCAL_VIEWER, GL_TRUE).`

Для задания освещения как лицевых, так и нелицевых граней (для нелицевых граней вектор нормали переворачивается) служит следующая команда:

`glLightModeli(GL_LIGHT_MODEL_TWO_SIDE, GL_TRUE).`

Причём существует возможность отдельного задания свойств материала для каждой из сторон.

Свойства материала, из которого сделан объект, задаются при помощи процедуры:

`glMaterial{ i f }[v](GLenum face, GLenum pname, TYPE param).`

Параметр `face` указывает, для какой из сторон грани задается свойство, и принимает одно из следующих значений: `GL_BACK`, `GL_FRONT_AND_BACK`, `GL_FRONT`.

Параметр `pname` указывает, какое именно свойство материала задается.

Расчёт освещённости в OpenGL не учитывает затенения одних объектов другими.

Полупрозрачность

Полупрозрачность объектов устанавливается с использованием α -канала (в RGBA-представлении цвета). Если его значение равно единице – объект непрозрачен. Задавая значения, отличные от единицы, можно смешивать цвет выводимого пикселя с цветом пикселя, уже находящегося в соответствующем месте на экране, создавая тем самым эффект прозрачности.

α -значение характеризует степень поглощения фрагментом проходящего через него света. Так, если у стекла установить значение, равное 0,2, то в результате вывода цвет получившегося фрагмента будет на 20 % состоять из собственного цвета стекла и на 80 % – из цвета фрагмента под ним.

Для использования α -канала необходимо сначала разрешить режим прозрачности и смешения цветов командой `glEnable(GL_BLEND).`

В процессе смешения цветов цветовые компоненты выводимого фрагмента $R_s G_s B_s A_s$ смешиваются с цветовыми компонентами уже выведенного фрагмента $R_d G_d B_d A_d$ по формуле

$$(R_s S_r + R_d D_r, G_s S_g + G_d D_g, B_s S_b + B_d D_b, A_s S_a + A_d D_a),$$

где (S_r, S_g, S_b, S_a) и (D_r, D_g, D_b, D_a) – коэффициенты смешения.

Для задания связи этих коэффициентов с α -значениями используется следующая функция:

`glBlendFunc(GLenum sfactor, GLenum dfactor).`

Здесь параметр `sfactor` задаёт то, как нужно вычислять коэффициенты (S_r, S_g, S_b, S_a) , а параметр `dfactor` – коэффициенты (D_r, D_g, D_b, D_a) .

Наложение текстуры

Текстурирование позволяет наложить изображение на многоугольник и вывести этот многоугольник с наложенной на него текстурой, соответствующим образом преобразованной. OpenGL поддерживает одно- и двумерные текстуры, а также различные способы наложения текстуры.

Для использования текстуры надо сначала разрешить одно- или двумерное текстурирование при помощи команд `glEnable(GL_TEXTURE_1D)` или `glEnable(GL_TEXTURE_2D)`.

Для задания двумерной текстуры служит процедура: `glTexImage2D(GLenum target, GLint level, GLint component, GLsizei width, GLsizei height, GLint border, GLenum format, GLenum type, const GLvoid *pixels)`.

Параметр `target` зарезервирован для будущего использования и в текущей версии OpenGL должен быть равен `GL_TEXTURE_2D`. Параметр `level` используется в том случае, если задается несколько разрешений данной текстуры. При ровно одном разрешении он должен быть равным нулю.

Следующий параметр – `component` – целое число от 1 до 4, показывающее, какие из RGBA-компонентов выбраны для использования. Значение 1 выбирает компонент R, значение 2 выбирает R и A компоненты, 3 соответствует R, G и B, а 4 соответствует компонентам RGBA.

Параметры `width` и `height` задают размеры текстуры, `border` задает размер границы (бортика), обычно равный нулю. Как параметр `width`, так и параметр `height`, должны иметь вид $2^n + 2b$, где n – целое число, а b – значение параметра `border`. Максимальный размер текстуры зависит от реализации OpenGL, но он не менее 64 на 64.

При текстурировании OpenGL поддерживает использование пирамидального фильтрования (mip-mapping). Для этого необходимо иметь текстуры всех промежуточных размеров, являющихся степенями двух, вплоть до 1×1 , и для каждого такого разрешения вызвать `glTexImage2D` с соответствующими параметрами `level`, `width`, `height` и `image`. Кроме того, необходимо задать способ фильтрования, который будет применяться при выводе текстуры.

Под фильтрованием здесь подразумевается способ, которым для каждого пикселя будет выбираться подходящий элемент текстуры (тексель). При текстурировании возможна ситуация, когда 1 пикселю соответствует небольшой фрагмент текселя (увеличение) или же, наоборот, когда 1 пикселю соответствует целая группа текселей (уменьшение).

Способ выбора соответствующего текселя как для увеличения, так и для уменьшения (сжатия) текстуры необходимо задать отдельно. Для этого используется процедура:

`glTexParameterf(GL_TEXTURE_2D, GLenum p1, GLfloat p2),`

где параметр `p1` показывает, задается ли фильтр для сжатия или для растяжения текстуры, принимая значение `GL_TEXTURE_MIN_FILTER` или

GL_TEXTURE_MAG_FILTER. Параметр p2 задает способ фильтрации.

При использовании пирамидального фильтрации помимо выбора текстеля на одном слое текстуры появляется возможность либо выбрать один соответствующий слой, либо проинтерполировать результаты выбора между двумя соседними слоями. Для правильного применения текстуры каждой вершине следует задать соответствующие ей координаты текстуры при помощи процедуры:

```
glTexCoord{1 2 3 4}{s i f d}[v](TYPE coord, ...).
```

Этот вызов задаёт значения индексов текстуры для последующей команды glVertex.

Если размер грани больше, чем размер текстуры, то для циклического повторения текстуры служат команды:

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_S_WRAP,  
GL_REPEAT);
```

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_T_WRAP,  
GL_REPEAT).
```

Координаты текстуры обычно подвергаются преобразованию при помощи матрицы текстурирования. По умолчанию она совпадает с единичной матрицей, но пользователь сам имеет возможность задать преобразования текстуры, например, следующим образом:

```
glMatrixMode(GL_TEXTURE);  
glRotatef(...);  
glMatrixMode(GL_MODELVIEW).
```

При выводе текстуры OpenGL может использовать линейную интерполяцию или точно учитывать перспективное искажение. Для задания точного текстурирования служит команда:

```
glHint(GL_PERSPECTIVE_CORRECTION_HINT, GL_NICEST).
```

Если качество не играет большой роли, а нужна высокая скорость рендеринга, то в качестве последнего аргумента следует использовать константу GL_FASTEST.

Описанный выше способ работ с текстурами используется в OpenGL версии 1.0. В более новых версиях OpenGL, начиная с версии 1.1, введены дополнительные функции, повышающие удобство работы с текстурами. В OpenGL 1.0 процедуру glTexImage2D необходимо вызывать всякий раз, когда нужно сменить текущую текстуру. Это достаточно медленный способ. В OpenGL 1.1 имеется возможность присваивать имена текстурам и затем сменять текущую текстуру только указанием имени новой текстуры, без повторной её загрузки в память процедурой glTexImage2D.

Имя текстуры представляет собой уникальное значение типа GLuint. Перед использованием текстуры необходимо присвоить ей имя. Имена текстур можно сгенерировать при помощи процедуры glGenTextures(GLsizei n, GLuint *textures).

Параметр n определяет количество текстур, для которых необходимо сгенерировать имена. Параметр textures является указателем на массив

переменных типа GLuint, состоящим из n элементов. После вызова процедуры каждый элемент массива будет содержать уникальное имя текстуры, которое затем может быть использовано при работе.

Для выбора текущей (активной) текстуры используется функция: `glBindTexture(GLenum target, GLuint texture)`.

Параметр `target` определяет тип текстуры (одномерная – `GL_TEXTURE_1D` или двумерная – `GL_TEXTURE_2D`). На практике более часто используются двумерные текстуры, которые представляют собой обычные двумерные изображения. Параметр `texture` определяет имя текстуры, которую необходимо сделать активной.

После того, как установлена активная текстура, можно вызвать процедуру `glTexImage2D` и задать параметры текстуры, а также сами её тексели. После вызова процедуры `glTexImage2D` текстура готова к применению.

Для того чтобы наложить текстуру на объект или многоугольник достаточно установить активную текстуру (процедура `glBindTexture`) и определить текстурные координаты при помощи процедуры `glTexCoord`.

Достоинство использования функций OpenGL 1.1 для работы с текстурами заключается не только в более высоком быстродействии по сравнению с использованием процедуры `glTexImage2D`, но и в повышенном удобстве работы с текстурами. Создав массив текстурных имён можно работать одновременно с несколькими текстурами, вызывая лишь функцию `glBindTexture` по мере необходимости.

При написании программы, которая покрывает куб текстурой, необходимо учитывать то, что саму текстуру требуется загружать из отдельного файла.

Для загрузки текстуры, а также задания ее параметров рекомендуется написать отдельную функцию. Например, если текстура размером 64x64, то считывание ее в массив выглядит так:

```
Bits: Array [0..63, 0..63, 0..2] of GLubyte;  
For i := 0 to 63 do  
  For j := 0 to 63 do begin  
    bits[i, j, 0] := GetRValue(bitmap.Canvas.Pixels[i, j]);  
    bits[i, j, 1] := GetGValue(bitmap.Canvas.Pixels[i, j]);  
    bits[i, j, 2] := GetBValue(bitmap.Canvas.Pixels[i, j]);  
  end;
```

Последний компонент трехмерного массива – это цветовые составляющие пиксела RGB. Загрузить текстуру в память можно следующей командой:

```
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, 64, 64, 0, GL_RGB,  
GL_UNSIGNED_BYTE, @Bits);
```

После этого необходимо разрешить работать с текстурой:

```
glEnable(GL_TEXTURE_2D);
```

Следующий этап – это привязка текстурных координат к координатам объектов. В обработчике создания окна пишем (для одной грани куба):

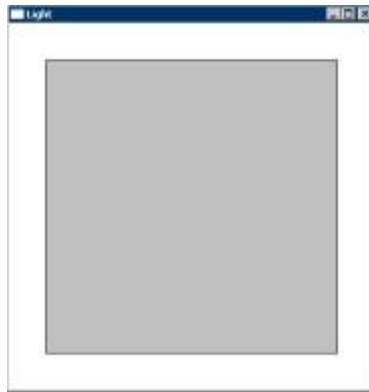
```

glBegin (GL_QUADS);
    glTexCoord2d (1.0, 0.0);
    glVertex3f (-1.0, -1.0, 1.0);
    glTexCoord2d (1.0, 1.0);
    glVertex3f (1.0, -1.0, 1.0);
    glTexCoord2d (0.0, 1.0);
    glVertex3f (1.0, -1.0, -1.0);
    glTexCoord2d (0.0, 0.0);
    glVertex3f (-1.0, -1.0, -1.0);
glEnd.

```

Пример построения квадрата.

Результат выполнения программы



Текст программы

```

#include <windows.h>
#include <GL/gl.h>
#include <GL/glu.h>
#include <GL/glaux.h>
void CALLBACK resize(int width,int height)
{
    glViewport(0,0,width,height);
    glMatrixMode( GL_PROJECTION );
    glLoadIdentity();
    glOrtho(-5,5, -5,5, 2,12);
    gluLookAt( 0,0,5, 0,0,0, 0,1,0 );
    glMatrixMode( GL_MODELVIEW );
}
void CALLBACK display(void)
{
    glClear( GL_COLOR_BUFFER_BIT |GL_DEPTH_BUFFER_BIT);

    glColor3d(1,1,1);

```

```

glBegin(GL_QUADS);
glVertex3d(-4,-4,0);
glVertex3d(-4, 4,0);
glVertex3d( 4, 4,0);
glVertex3d( 4,-4,0);
glEnd();
/*
glColor3d(1,0,0);
auxSolidSphere(0.1);
*/
    auxSwapBuffers();
}
void main()
{
float pos[4] = { 3,3,3,0.5 };
float dir[3] = { -1,-1,-1 };
    GLfloat mat_specular[] = { 1,1,1,1 };
    auxInitPosition( 50, 10, 400, 400);
    auxInitDisplayMode( AUX_RGB | AUX_DEPTH | AUX_DOUBLE );
    auxInitWindow( "Light" );
auxIdleFunc(display);
auxReshapeFunc(resize);
glEnable(GL_DEPTH_TEST);
glEnable(GL_COLOR_MATERIAL);
glEnable(GL_LIGHTING);
glEnable(GL_LIGHT0);
glLightfv(GL_LIGHT0, GL_POSITION, pos);
glLightfv(GL_LIGHT0, GL_SPOT_DIRECTION, dir);
glMaterialfv(GL_FRONT, GL_SPECULAR, mat_specular);
glMaterialf(GL_FRONT, GL_SHININESS, 128.0);
glLighti(GL_LIGHT0, GL_SPOT_EXPONENT, 0);
glLighti(GL_LIGHT0, GL_SPOT_CUTOFF, 90);
auxMainLoop(display);
}

```

Пример построения 2D объектов.

Результат выполнения программы



Текст программы

```
#include <windows.h>
#include <GL/gl.h>
#include <GL/glu.h>
#include <GL/glaux.h>
void CALLBACK resize(int width,int height)
{
    glViewport(0,0,width,height);
    glMatrixMode( GL_PROJECTION );
    glLoadIdentity();
    glOrtho(-5,5, -5,5, 2,12);
    gluLookAt( 0,0,5, 0,0,0, 0,1,0 );
    glMatrixMode( GL_MODELVIEW );
}
void CALLBACK display(void)
{
    glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glColor3d(1,1,1);
    glBegin(GL_QUADS);
        glVertex3d(-4,-4,0);
        glVertex3d(-4, 4,0);
        glVertex3d( 4, 4,0);
        glVertex3d( 4,-4,0);
    glEnd();
    glColor3d(0,1,1);
    glBegin(GL_QUADS);
        glVertex3d(-3,-3,1);
        glVertex3d(-3, 3,1);
        glVertex3d( 3, 3,1);
        glVertex3d( 3,-3,1);
    glEnd();
    glColor3d(0,0,1);
```

```

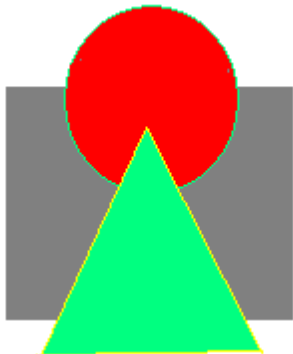

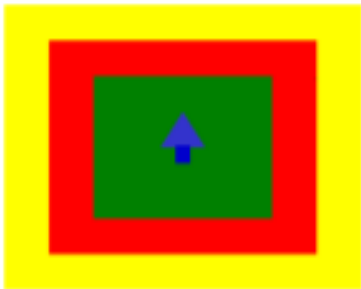
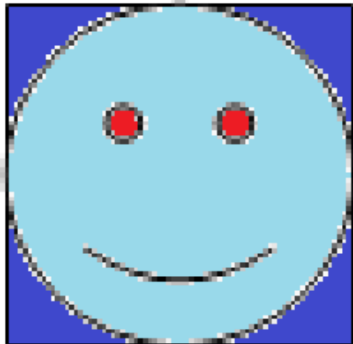
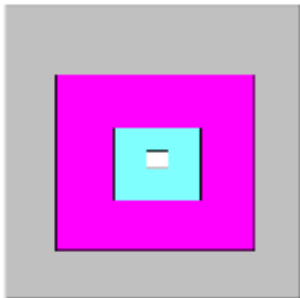


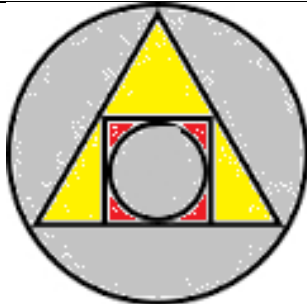
glBegin(GL_TRIANGLES);
    glVertex3d(-3,-3,2);
    glVertex3d(-3, 3,2);
    glVertex3d( 3, 3,2);
    //glVertex3d( 3,-3,1);
glEnd();
glColor3d(1,0,1);
glBegin(GL_QUADS);
    glVertex3d(-1,-1,3);
    glVertex3d(-1, 1,3);
    glVertex3d( 1, 1,3);
    glVertex3d( 1,-1,3);
glEnd();
/*
glColor3d(1,0,0);
    auxSolidSphere(0.1);
*/
    auxSwapBuffers();
}
void main()
{
float pos[4] = {3,3,3,0.5};
float dir[3] = {-1,-1,-1};
    GLfloat mat_specular[] = {1,1,1,1};
    auxInitPosition( 50, 10, 400, 400);
    auxInitDisplayMode( AUX_RGB | AUX_DEPTH | AUX_DOUBLE );
auxInitWindow( "Light" );
auxIdleFunc(display);
    auxReshapeFunc(resize);
glEnable(GL_DEPTH_TEST);
glEnable(GL_COLOR_MATERIAL);
glEnable(GL_LIGHTING);
glEnable(GL_LIGHT0);
glLightfv(GL_LIGHT0, GL_POSITION, pos);
glLightfv(GL_LIGHT0, GL_SPOT_DIRECTION, dir);
    glMaterialfv(GL_FRONT, GL_SPECULAR, mat_specular);
    glMaterialf(GL_FRONT, GL_SHININESS, 128.0);
    glLighti(GL_LIGHT0, GL_SPOT_EXPONENT, 0);
    glLighti(GL_LIGHT0, GL_SPOT_CUTOFF, 90);
    auxMainLoop(display);
}


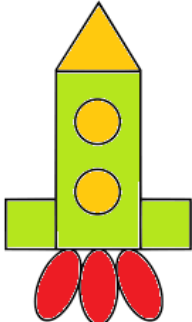

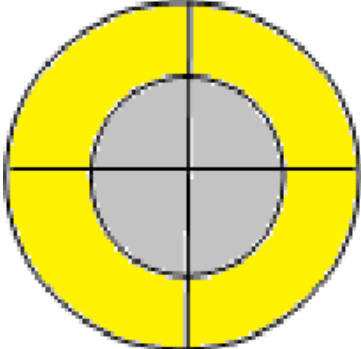
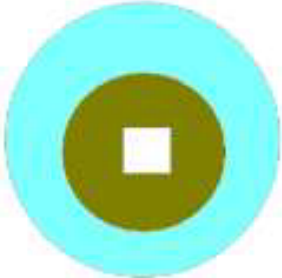
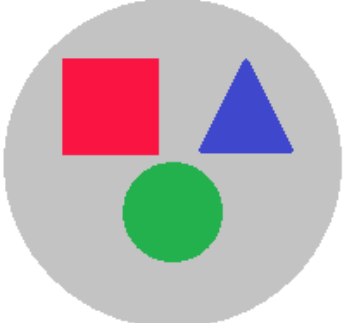
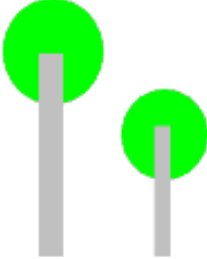
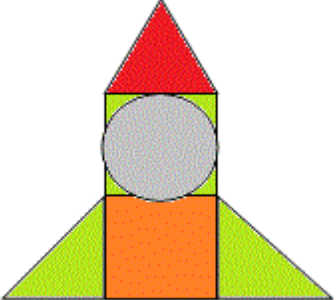
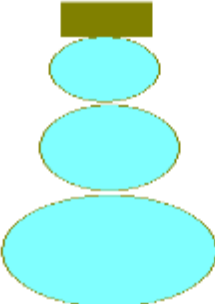
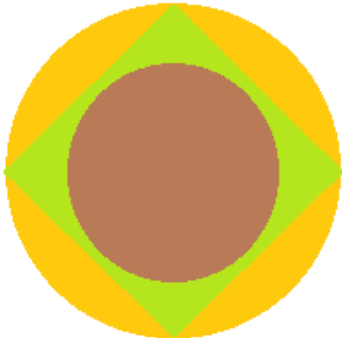
```

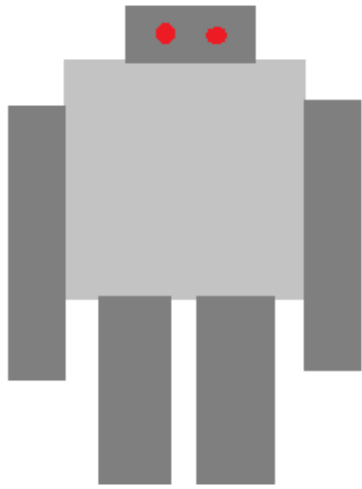

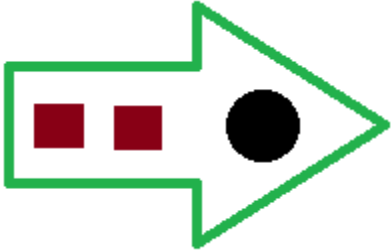




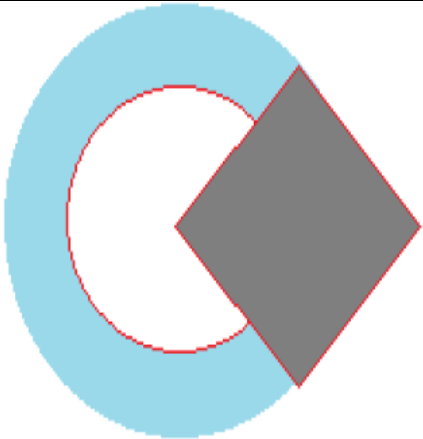

Ход работы




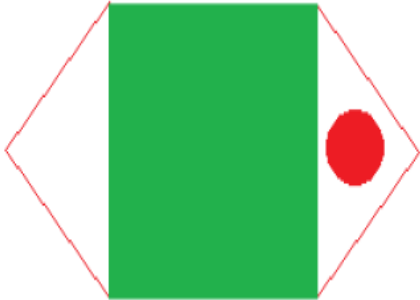
В рамках данной лабораторной работы необходимо, используя C++ и OpenGL, разработать программу построения 2D объектов в соответствии со своим вариантом задания.

Варианты заданий:

Номер вариан нта	Задание	Номер вариан нта	Задание
1		16	
2		17	
3		18	
4		19	

Номер вариан- та	Задание	Номер вариан- та	Задание
5		20	
6		21	
7		22	
8		23	
9		24	

Номер вариан- та	Задание	Номер вариан- та	Задание
10		25	
11		26	
12		27	
13		28	

Номер варианта	Задание	Номер варианта	Задание
14		29	
15		30	

Контрольные вопросы

1. Для чего предназначена библиотека OpenGL?
2. Перечислите основные возможности OpenGL.
3. Какие буферы присутствуют в библиотеке OpenGL?
4. Каким образом в OpenGL задаются графические примитивы?
5. Какие виды закрашивания объектов существуют в OpenGL?
6. Какое максимальное количество источников света можно задать для объекта?
7. Перечислите параметры процедуры `glTexImage2D`.
8. Для чего служат процедуры `glBegin()` и `glEnd()`?