ПРАКТИЧЕСКОЕ ЗАНЯТИЕ №2

ОСНОВНЫЕ ПРИМИТИВЫ OPENGL

Цели: формирование практических навыков по работе с графическими примитивами OpenGL.

Задачи: научиться устанавливать размеры наблюдаемого объема, изучить параметры функции glVertex, изучить основные параметры функции glBegin, сформировать понимание особенности использования функции glEnable с конкретными геометрическими примитивами, выяснить основы построения сплошных объектов.

Выполнение:

1. Ознакомиться с теоретическим материалом.
2. Выполнить основные задания.
3. Предоставить отчет, по каждому заданию содержащий: формулировку задания, исходный код программы, скриншоты работающей программы (один или несколько, если необходимо).
4. Ответить на вопросы преподавателя.

СИНТАКС КОМАНД OPENGL

Все команды OpenGL — это процедуры или функции. Для того, чтобы пространство имен OpenGL не пересекалось с пространствами имен других библиотек, для всех команд введен единый префикс «gl». Например: glCreateProgram или glGenFramebuffers.

Некоторые команды OpenGL могут выполнять одну и ту же функцию и различаться только тем, какие параметры они принимают. Так, к примеру, вершина в OpenGL может быть как двухмерной, так и трехмерной, как быть описана вещественными числами, так и целыми. Кроме того, в имена команд входят суффиксы, несущие информацию о числе и типе передаваемых параметров. В OpenGL полное имя команды имеет вид:

type **glCommand_name[1 2 3 4][b s i f d ub us ui][v]**

Имя состоит из нескольких частей:

gl - имя библиотеки, в которой описана эта функция: для базовых функций OpenGL, функций из библиотек GL, GLU, GLUT, GLAUX это gl, glu, glut, aux соответственно.

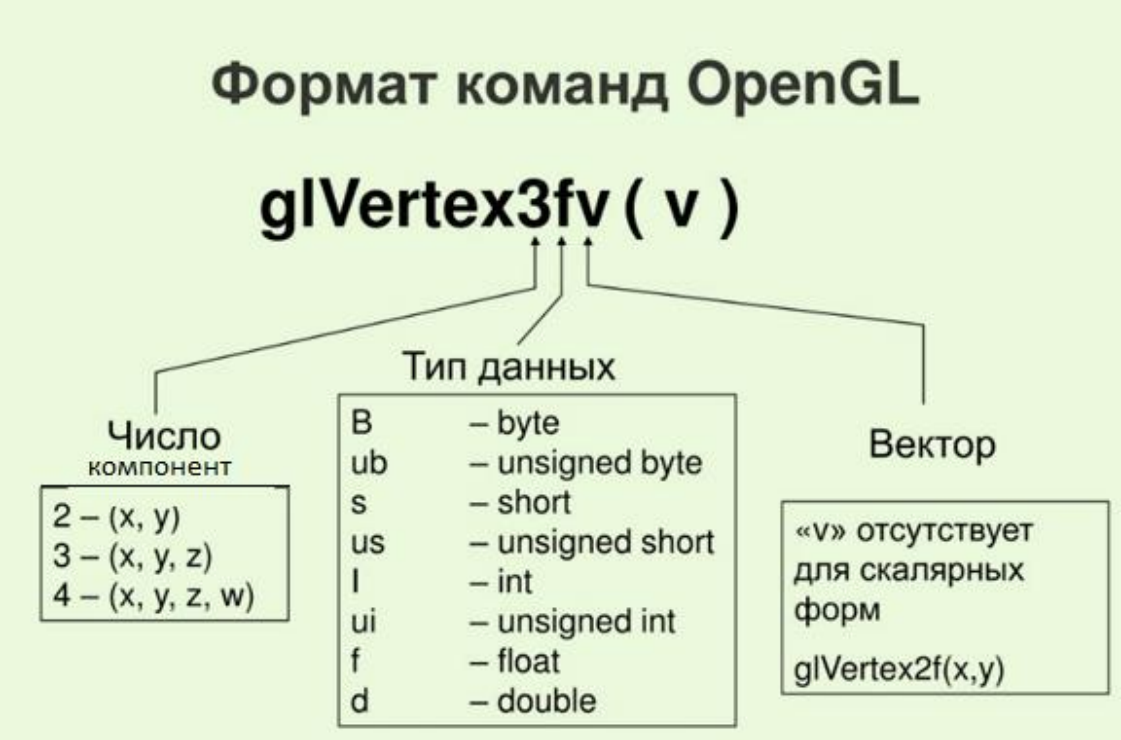
Command_name - имя команды (процедуры или функции).

[1 2 3 4] - число аргументов команды.

[b s i f d ub us ui] - тип аргумента: символ b – GLbyte (аналог char в C\C++), символ i – GLint (аналог int), символ f – GLfloat (аналог float) и так далее. Полный список типов и их описание можно посмотреть в файле gl.h

[v] - наличие этого символа показывает, что в качестве параметров функции используется указатель на массив значений.

Символы в квадратных скобках в некоторых названиях не используются. Например, команда glVertex2i() описана в библиотеке GL, и использует в качестве параметров два целых числа, а команда glColor3fv() использует в качестве параметра указатель на массив из трех вещественных чисел.



В своих функциях OpenGL использует следующие типы данных:

Имя типа	Размерность в байтах	Описание
GLboolean	8	Логический тип. GL_TRUE или GL_FALSE
GLbyte	8	Знаковый байт
GLubyte	8	Беззнаковый байт
GLshort	16	Знаковое слово
GLushort	16	Беззнаковое слово
GLint	32	Знаковое двойное слово
GLuint	32	Беззнаковое двойное слово
GLfloat	32	Вещественное число
GLdouble	64	Вещественное число двойной точности
GLclampf	32	Вещественное число [0;1]
GLclampd	64	Вещественное число двойной точности [0;1]
GLsizei	32	Размерность. Неотрицательное целое
GLenum	32	Перечисляемый тип. Целый
GLintptr	32	Указатель.
GLbitfield	32	Битовые поля. Используются как флаги. Целый

Что такое точки, линии и полигоны?

Вероятно, у вас имеется некоторое представление о математическом смысле терминов точка, линия и полигон. Их смысл в OpenGL почти тот же самый, но не идентичный.

Одно из различий проистекает из ограничений на компьютерные расчеты. В любой реализации OpenGL числа с плавающей точкой имеют конечную точность и, следовательно, могут возникать ошибки, связанные с округлением. Как следствие, координаты точек, линий и полигонов в OpenGL страдают этим же недостатком.

Более важное различие проистекает из ограничений растровых дисплеев. На таких дисплеях наименьшим отображаемым элементом является пиксель и, хотя пиксель по размеру может быть меньше 1/100 дюйма, он все равно значительно больше, чем математические понятия бесконечно малого элемента (для точек) и бесконечно короткого (для линий). Когда OpenGL производит вычисления, она предполагает, что точки представлены в виде векторов с координатами в формате с плавающей точкой.

Однако точка, как правило (но не всегда), отображается на дисплее в виде одного пикселя, и несколько точек, имеющих слегка различающиеся координаты, OpenGL может нарисовать на одном и том же пикселе.

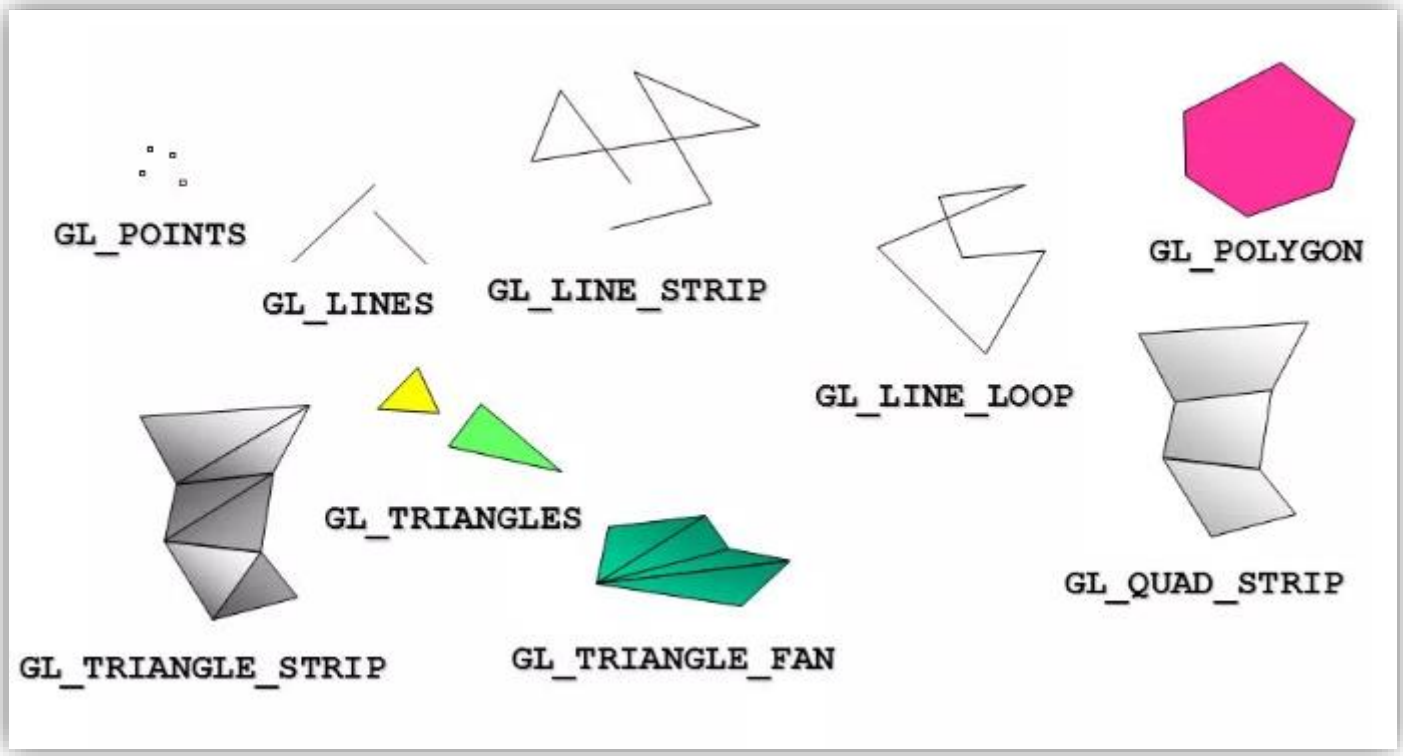
Определение примитива или последовательности примитивов происходит между вызовами команд

void **glBegin** (GLenum *mode*);

void **glEnd** (void);

Параметр *mode* определяет тип примитива, который задается внутри и может принимать следующие значения:

GL_POINTS	каждая вершина задает координаты некоторой точки.
GL_LINES	каждая отдельная пара вершин определяет отрезок; если задано нечетное число вершин, то последняя вершина игнорируется.
GL_LINE_STRIP	каждая следующая вершина задает отрезок вместе с предыдущей.
GL_LINE_LOOP	отличие от предыдущего примитива только в том, что последний отрезок определяется последней и первой вершиной, образуя замкнутую ломаную.
GL_TRIANGLES	каждые отдельные три вершины определяют треугольник; если задано не кратное трем число вершин, то последние вершины игнорируются.
GL_TRIANGLE_STRIP	каждая следующая вершина задает треугольник вместе с двумя предыдущими.
GL_TRIANGLE_FAN	треугольники задаются первой вершиной и каждой следующей парой вершин (пары не пересекаются).
GL_QUADS	каждая отдельная четверка вершин определяет четырехугольник; если задано не кратное четырем число вершин, то последние вершины игнорируются.
GL_QUAD_STRIP	четырехугольник с номером n определяется вершинами с номерами 2n-1, 2n, 2n+2, 2n+1.
GL_POLYGON	последовательно задаются вершины выпуклого многоугольника.



Точки

Точка определяется набором чисел с плавающей точкой, называемым вершиной. Все внутренние вычисления производятся в предположении, что координаты заданы в трехмерном пространстве. Для вершин, которые пользователь определил как двумерные (то есть задал только x- и y-координаты), OpenGL устанавливает z-координату равной 0.

Дополнительно: OpenGL работает в *однородных координатах* трехмерной проекционной геометрии, таким образом, во внутренних вычислениях все вершины представлены четырьмя числами с плавающей точкой (x,y,z,w). Если w отлично от 0, то эти координаты в Евклидовой геометрии соответствуют точке (x/w,y/w,z/w). Вы можете указывать w-координату в командах OpenGL, но это делается достаточно редко. Если w-координата не указана, она принимается равной 1.0.

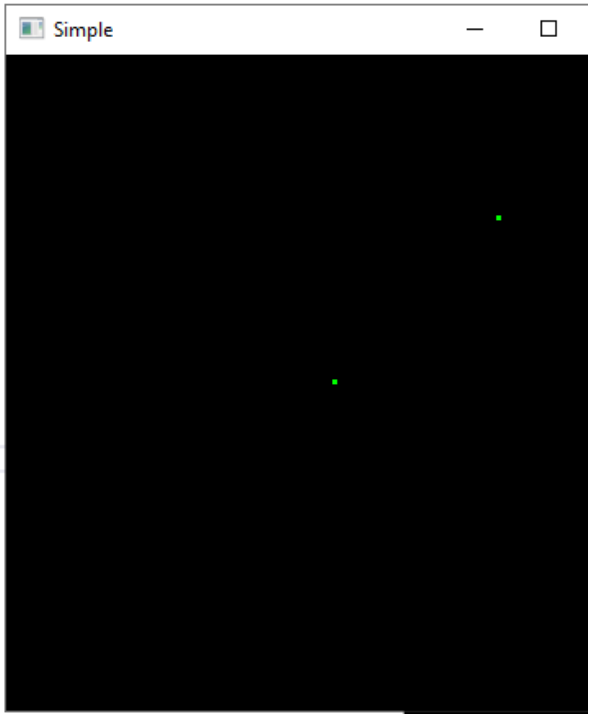
Для управления размером рисуемых точек используйте **glPointSize()** и укажите желаемый размер (в пикселях) как аргумент.

void glPointSize (GLfloat *size*);

Устанавливает длину и высоту (в пикселях) для визуализируемой точки, *size* должен быть больше 0.0 и, по умолчанию, равен 1.0. Дробные размеры точек округляются до ближайшего целого и для каждой точки на экране будет нарисован квадратный регион пикселей.

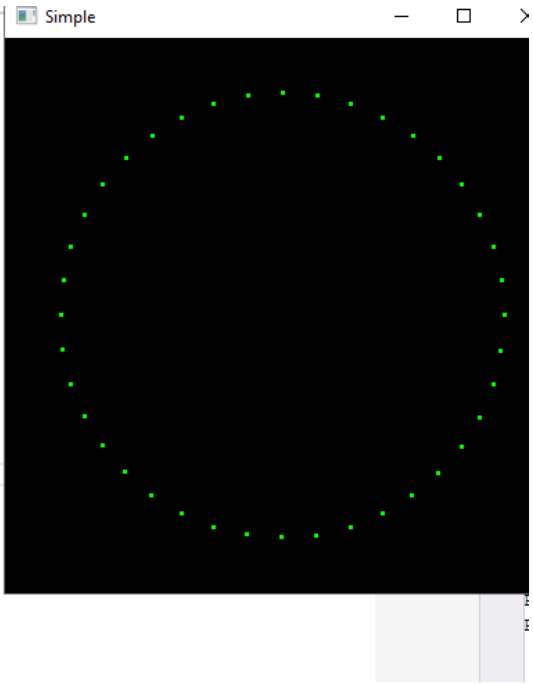
ЛИСТИНГ 1. Две точки.

```
#include "glew.h"
#include "glut.h"
void RenderScene(void)
{

    glClear(GL_COLOR_BUFFER_BIT);
    glColor3f(0.0f, 1.0f, 0.0f);
    glPointSize(3);
    glBegin(GL_POINTS);
        glVertex3f(0.0f, 0.0f, 0.0f); // VO
        glVertex3f(0.5f, 0.50f, 0.0f); // VI
    glEnd();
    // В буфер вводятся команды рисования
    glutSwapBuffers();
}
// Точка входа основной программы
void main(int argc, char** argv)
{
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_DOUBLE);
    glutInitWindowSize(400, 400);
    glutCreateWindow("Simple");
    glClearColor(0, 0, 0, 0);
    glutDisplayFunc(RenderScene);
    glutMainLoop();
}
```



ЛИСТИНГ 2. Окружность из точек.

```
#include "glew.h"
#include "glut.h"
#include "math.h"
float x2, y2, angle;
void RenderScene(void)
{
    glClear(GL_COLOR_BUFFER_BIT);
    glColor3f(0.0f, 1.0f, 0.0f);
    glPointSize(3);
    for (angle = 0.0f; angle <= 3.14*2; angle += (3.14 / 20.0f))
    {
        x2 = 0.8 * cos(angle);
        y2 = 0.8 * sin(angle);
        glBegin(GL_POINTS);
        glVertex3f(x2, y2, 0.0f);
        glEnd();
    }
        glFlush();}

void main(void)
{
    glutInitDisplayMode(GLUT_SINGLE);
    glutInitWindowSize(400, 400);
    glutCreateWindow("Simple");
    glClearColor(0, 0, 0, 0);
    glutDisplayFunc(RenderScene);
    glutMainLoop();
}
```



Линии

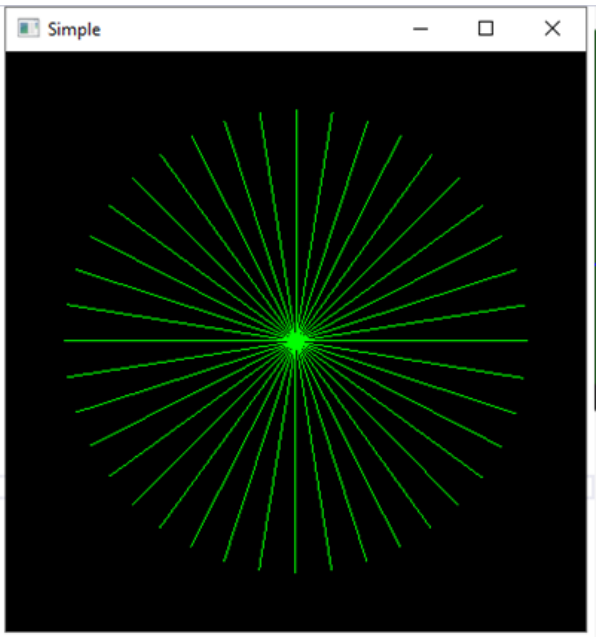
В OpenGL термин линия относится к сегменту прямой (отрезку), а не к математическому понятию прямой, которая предполагается бесконечной в обоих направлениях. Достаточно просто задавать серию соединенных отрезков или даже закрытую последовательность отрезков. В любом случае отрезки описываются в терминах вершин на их концах.

void **glLineWidth** (GLfloat *width*);

Устанавливает толщину линии в пикселях, *width* должно быть больше 0.0 и по умолчанию равно 1.0.

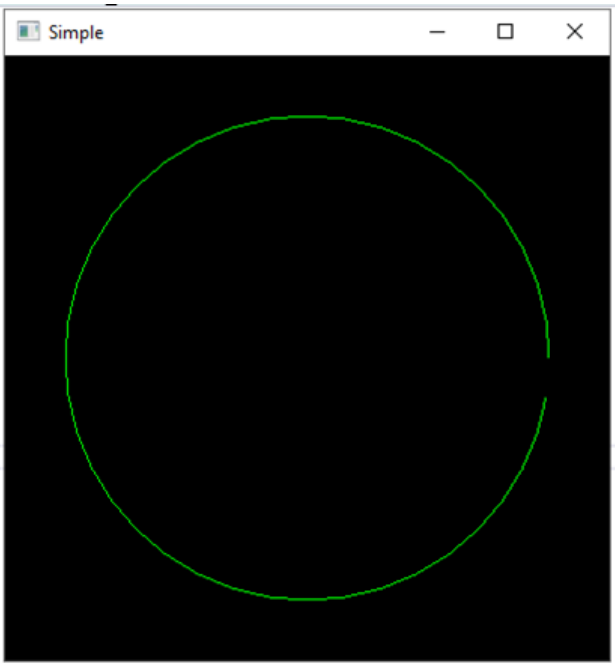
ЛИСТИНГ 3. Линии

```
#include "glut.h"
#include "math.h"
float x2, y2, angle;
void RenderScene(void)
{
    glClear(GL_COLOR_BUFFER_BIT);
    glColor3f(0.0f, 1.0f, 0.0f);
    glPointSize(3);
    for (angle = 0.0f; angle <= 3.14*2; angle += (3.14 / 20.0f))
    {
        x2 = 0.8 * cos(angle);
        y2 = 0.8 * sin(angle);
        glBegin(GL_LINES);
        glVertex3f(0.0f, 0.0f, 0.0f);
        glVertex3f(x2, y2, 0.0f);
        glEnd();
    }
        glFlush();}
void main(void)
{
    glutInitDisplayMode(GLUT_SINGLE);
    glutInitWindowSize(400, 400);
    glutCreateWindow("Simple");
    glClearColor(0, 0, 0, 0);
    glutDisplayFunc(RenderScene);
    glutMainLoop();
}
```



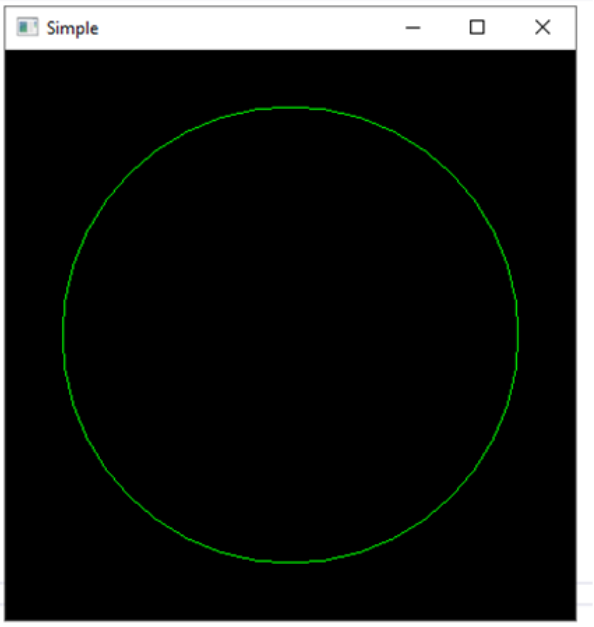
ЛИСТИНГ 4. Незамкнутая ломаная.

```
#include "glut.h"
#include "math.h"
float x2, y2, angle;
void RenderScene(void)
{
    glClear(GL_COLOR_BUFFER_BIT);
    glColor3f(0.0f, 1.0f, 0.0f);
    glBegin(GL_LINE_STRIP);
    for (angle = 0.0f; angle <= 3.14*2; angle += (3.14 / 20.0f))
    {
        x2 = 0.8 * cos(angle);
        y2 = 0.8 * sin(angle);

        glVertex3f(x2, y2, 0.0f);
    }
    glEnd();
    glFlush();
}

void main(void)
{
    glutInitDisplayMode(GLUT_SINGLE);
    glutInitWindowSize(400, 400);
    glutCreateWindow("Simple");
    glClearColor(0, 0, 0, 0);
    glutDisplayFunc(RenderScene);
    glutMainLoop();
}
```



ЛИСТИНГ 5. Замкнутая ломаная

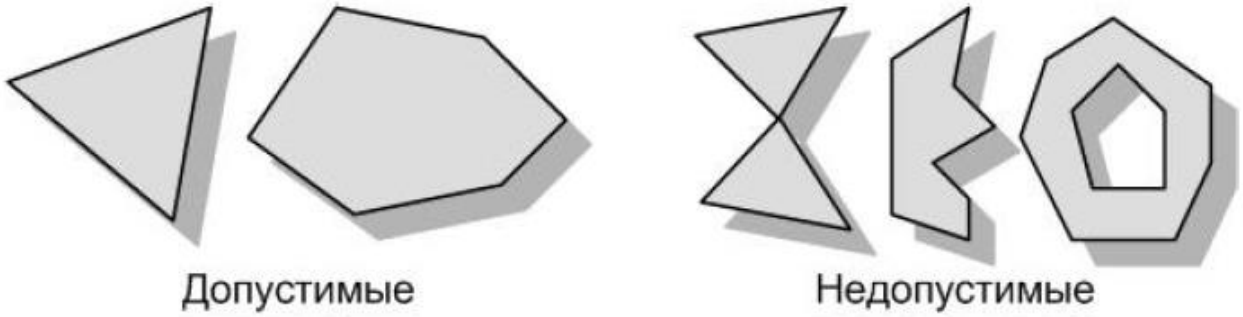
```
#include "glut.h"
#include "math.h"
float x2, y2, angle;
void RenderScene(void)
{
    glClear(GL_COLOR_BUFFER_BIT);
    glColor3f(0.0f, 1.0f, 0.0f);
    glBegin(GL_LINE_LOOP);
    for (angle = 0.0f; angle <= 3.14*2; angle += (3.14 / 20.0f))
    {
        x2 = 0.8 * cos(angle);
        y2 = 0.8 * sin(angle);

        glVertex3f(x2, y2, 0.0f);
    }
    glEnd();
    glFlush();
}

void main(void)
{
    glutInitDisplayMode(GLUT_SINGLE);
    glutInitWindowSize(400, 400);
    glutCreateWindow("Simple");
    glClearColor(0, 0, 0, 0);
    glutDisplayFunc(RenderScene);
    glutMainLoop();
}
```



Полигоны

Полигон (или многоугольник) – это область, ограниченная одной замкнутой ломаной, при этом каждый отрезок ломаной описывается вершинами на его концах (вершинами полигона). Обычно полигоны рисуются заполненными, но вы можете также рисовать полигоны в виде их границ или в виде точек на концах отрезков, образующих полигон. В общем случае полигоны могут быть достаточно сложны, поэтому OpenGL накладывает очень серьезные ограничения в отношении того, что считается полигональным примитивом. Во-первых, ребра полигонов в OpenGL не могут пересекаться (в математике полигон, удовлетворяющий этому условию, называется *простым*). Во-вторых, полигоны OpenGL должны быть выпуклыми. Полигон является выпуклым, если отрезок, соединяющий две точки полигона (точки могут быть и внутренними, и граничными) целиком лежит внутри полигона (то есть все его точки также принадлежат полигону). На рисунке приведены примеры нескольких допустимых и недопустимых полигонов. OpenGL не накладывает ограничений на количество вершин полигона (и, как следствие, на количество отрезков, определяющих его границу). Заметьте, что полигоны с дырами не могут быть описаны, так как они не являются выпуклыми, и могут быть заданы при помощи одной замкнутой ломаной. Если вы поставите OpenGL перед вопросом о рисовании *невыпуклого* полигона, результат может быть не таким, какой вы ожидаете. Например, на

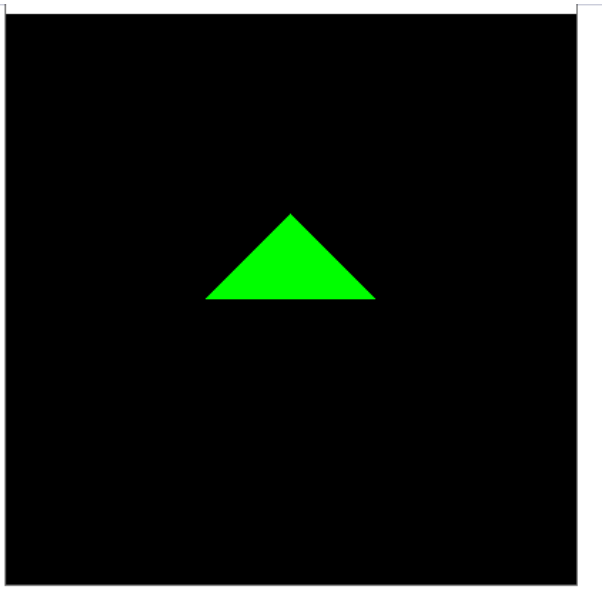
большинстве систем в ответ будет заполнена только выпуклая оболочка полигона (на многих системах не произойдет и этого).



Представим себе четырехугольник, в котором точки немного отклоняются от единой плоскости, и посмотрим на него со стороны ребра. В таких условиях может оказаться, что полигон не является простым и, как следствие, может быть не визуализирован верно. Ситуация не является такой уж необычной в случае, если вы аппроксимируете изогнутые поверхности с помощью четырехугольников. Однако вы всегда можете избежать подобных проблем, применяя треугольники, так как три отдельно взятые точки всегда лежат в одной плоскости.

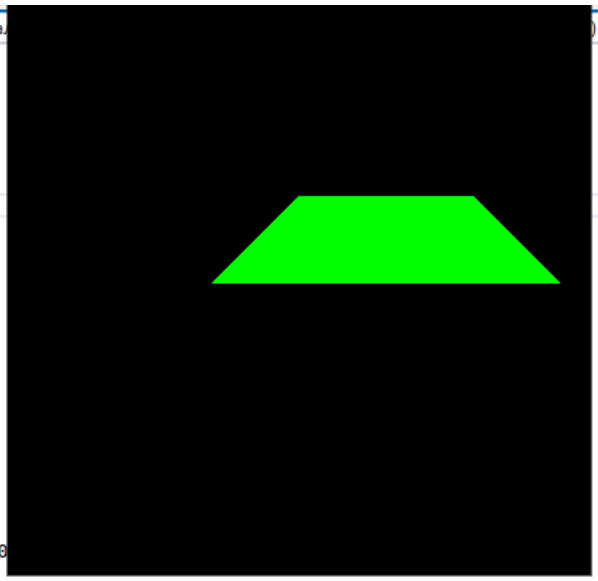
ЛИСТИНГ 6. Треугольник

```
void RenderScene(void)
{
    glClear(GL_COLOR_BUFFER_BIT);
    glColor3f(0.0f, 1.0f, 0.0f);
    glBegin(GL_TRIANGLES);

    glVertex3f(-30.f, 0.f, 0.f);
    glVertex3f(0.f,30.0f, 0.f);
    glVertex3f(30.f, 0.f, 0.f);
    glEnd();
    glFlush();
}

void main(void)
{
    glutInitDisplayMode(GLUT_SINGLE);
    glutInitWindowSize(400, 400);
    glutCreateWindow("Simple");
    glClearColor(0, 0, 0, 0);
    glOrtho(-100.0, 100.0, -100.0, 100.0, -1.0, 1.0);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    // Рисует прямоугольник,
    glutDisplayFunc(RenderScene);
    glutMainLoop();
}
```



ЛИСТИНГ 7. Треугольники.

```
prob2
    glClear(GL_COLOR_BUFFER_BIT);
    glColor3f(0.0f, 1.0f, 0.0f);
    glBegin(GL_TRIANGLE_STRIP);

    glVertex3f(-30.f, 0.f, 0.f);
    glVertex3f(0.f,30.0f, 0.f);
    glVertex3f(30.f, 0.f, 0.f);
    glVertex3f(60, 30, 0);
    glVertex3f(90, 0, 0);

    glEnd();
    glFlush();
}

void main(void)
{
    glutInitDisplayMode(GLUT_SINGLE);
    glutInitWindowSize(400, 400);
    glutCreateWindow("Simple");
    glClearColor(0, 0, 0, 0);
    glOrtho(-100.0, 100.0, -100.0, 100.0, -1.0
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    // Рисует прямоугольник,
    glutDisplayFunc(RenderScene);
    glutMainLoop();
```



ШАБЛОНЫ ОТРЕЗКОВ

Чтобы задать шаблон отрезка (например, для получения пунктирных или штриховых отрезков) следует использовать команду **glLineStipple()** и затем включить шаблонирование командой **glEnable()**.

glLineStipple(1,0x3F07);

glEnable(GL_LINE_STIPPLE);

Устанавливает текущий шаблон для отрезка
void **glLineStipple** (Glint *factor*, GLushort *pattern*);

Аргумент *pattern* – это 16-битная серия из нулей и единиц, определяющая, как будет рисоваться отрезок. Она повторяется по необходимости для шаблонирования всего отрезка. Единица означает, что соответствующая точка отрезка будет нарисована на экране, ноль означает, что точка нарисована не будет (на попиксельной основе). Шаблон применяется, начиная с младшего бита аргумента pattern. Шаблон может быть растянут с учетом значения фактора повторения *factor*. Каждый бит шаблона при наложении на отрезок расценивается как *factor* битов того же значения, идущих друг за другом. Например, если в шаблоне встречаются подряд три единицы, а затем два нуля и *factor* равен 3, то шаблон будет трактоваться как содержащий 9 единиц и 6 нулей. Допустимые значения аргумента *factor* ограничены диапазоном от 1 до 256. Шаблонирование должно быть включено передачей аргумента GL_LINE_STIPPLE в функцию **glEnable()**. Оно блокируется передачей того же аргумента в **glDisable()**.

Итак, в предыдущем примере с шаблоном равным 0x3F07 (что в двоичной системе счисления соответствует записи 0011111100000111) отрезок будет выведен на экран, начинаясь (по порядку) с 3 нарисованных пикселей, 5 отсутствующих, 6 нарисованных и 2 отсутствующих (если вам кажется, что мы применили шаблон задом – наперед вспомните, что он применяется, начиная с младшего бита). Если длина нашего отрезка на экране больше 16 пикселей, начиная с 17-го, шаблон будет применен заново и так далее до конца отрезка. Если бы *factor* был равен 2, шаблон был бы растянут, и отрезок выглядел бы следующим образом: вначале 6 нарисованных пикселей, затем 10 отсутствующих, 12 нарисованных и 4 отсутствующих. На рисунке показаны отрезки, нарисованные с применением различных шаблонов и факторов повторения шаблона. Если шаблонирование заблокировано, все отрезки рисуются таким же образом, как если бы шаблон был установлен в 0xFFFF, а фактор повторения в 1.

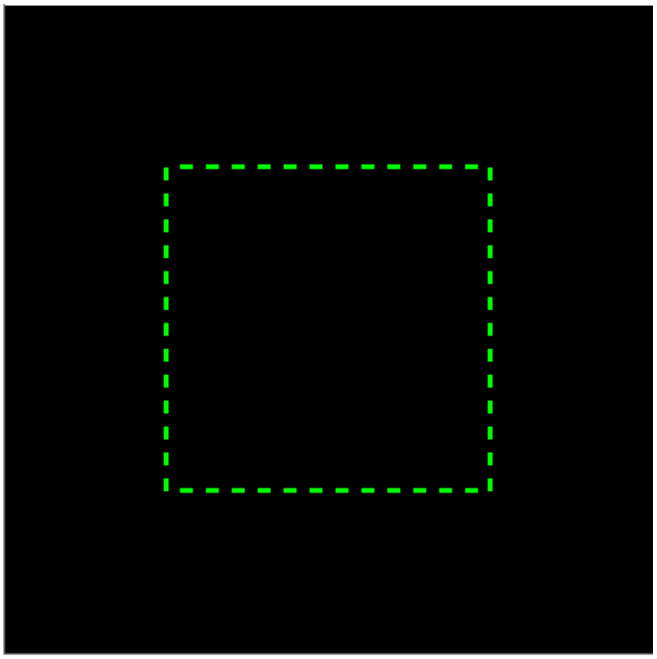
Обратите внимание, что шаблонирование может применяться в комбинации с линиями различной толщины.

SABLON	FACTOR	
0x00FF	1	———— ———— ————
0x00FF	2	———————— ————————
0x0C0F	1	—— — —— — —— —
0x0C0F	3	———————— ————
0xAAAA	1	- -
0xAAAA	2	— — — — — — — — — — — —
0xAAAA	3	— — — — — — — — — — — —
0xAAAA	4	— — — — — — — — — — — —

Как правило, разные типы примитивов имеют различную скорость визуализации на разных платформах. Для увеличения производительности предпочтительнее использовать примитивы, требующие меньшее количество информации для передачи на сервер, такие как **GL_TRIANGLE_STRIP**, **GL_QUAD_STRIP**, **GL_TRIAGLE_FAN.**

ЛИСТИНГ 8. Шаблон- пунктирная линия.

```
#include "glut.h"


void RenderScene(void)
{
    glClear(GL_COLOR_BUFFER_BIT);
    glColor3f(0.0f, 1.0f, 0.0f);
    glLineWidth(3);
    glEnable(GL_LINE_STIPPLE);
    glLineStipple(1, 0x00ff);
    glBegin(GL_LINE_LOOP);
    glLineWidth(10);
    glVertex3f(-50, -50, 0);
    glVertex3f(-50, 50, 0);
    glVertex3f(50, 50, 0);
    glVertex3f(50, -50, 0);
    glEnd();
    glFlush();
}
void main(void)
{
    glutInitDisplayMode(GLUT_SINGLE);
    glutInitWindowSize(400, 400);
    glutCreateWindow("Simple");
```



Если рисуется ломаная (с помощью GL_LINE_STRIP или GL_LINE_LOOP) то шаблон накладывается на нее непрерывно, независимо от того, где кончается один сегмент и начинается другой. В противовес этому для каждой индивидуальной линии (рисуемой с помощью GL_LINES) шаблон начинается заново, даже если все команды указания вершин вызываются внутри одного блока **glBegin() – glEnd()**.

Цвет вершины

Для задания текущего цвета вершины используются команды :

void **glColor[3 4][b s i f]** (GLtype *components*)
void **glColor[3 4][b s i f]v** (GLtype *components*)

Первые три параметра задают R, G, B компоненты цвета, а последний параметр определяет коэффициент непрозрачности (так называемая альфа-компонента). Если в названии команды указан тип ‘f’ (float), то значения всех параметров должны принадлежать отрезку [0,1], при этом по умолчанию значение альфа-компоненты устанавливается равным 1.0, что соответствует полной непрозрачности. Тип ‘ub’ (unsigned byte) подразумевает, что значения должны лежать в отрезке [0,255].

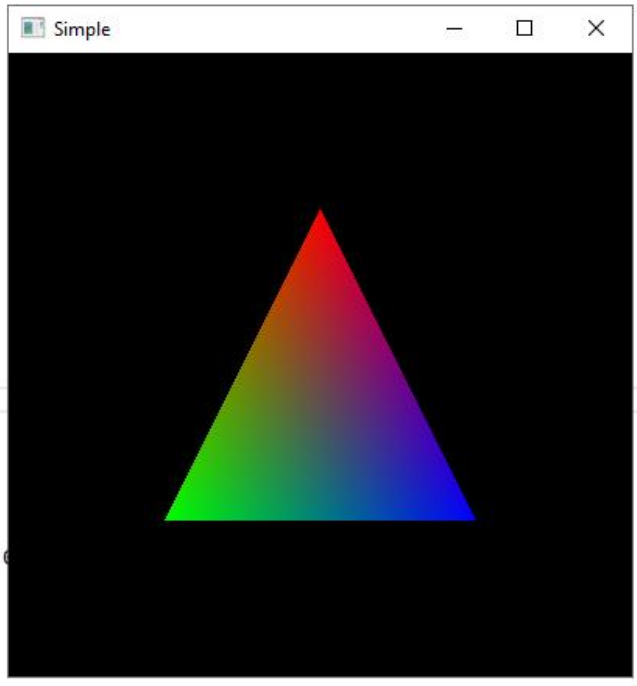
Вершинам можно назначать различные цвета, по умолчанию будет проводиться линейная интерполяция цветов по поверхности примитива.

Для управления режимом интерполяции используется команда

void **glShadeModel** (GLenum *mode*)
вызов которой с параметром **GL_SMOOTH** включает интерполяцию (установка по умолчанию), а с **GL_FLAT** – отключает.

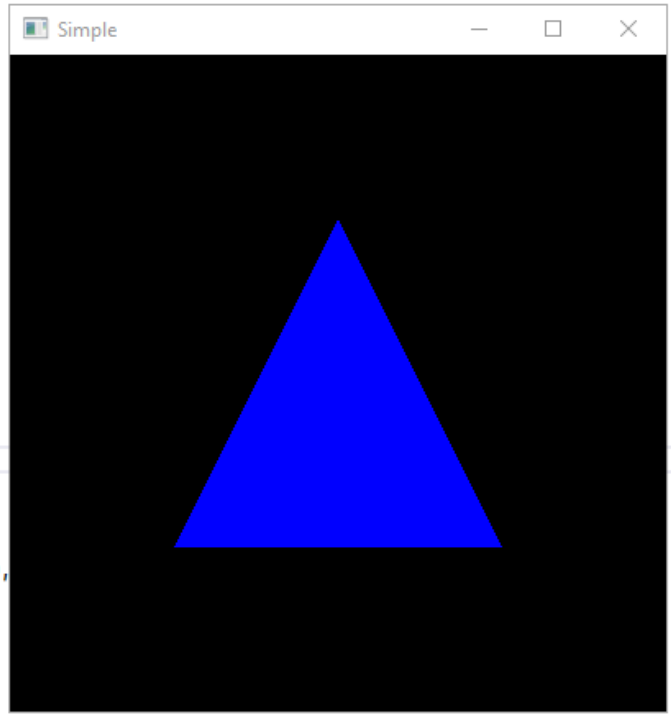
ЛИСТИНГ 9. Интерполяция цветов

```
void RenderScene(void)
{
    glClear(GL_COLOR_BUFFER_BIT);
    glShadeModel(GL_SMOOTH);
    glBegin(GL_TRIANGLES);
    glColor3f(0.0f, 1.0f, 0.0f);
    glVertex3f(-50, -50, 0);
    glColor3f(1.0f, 0.0f, 0.0f);
    glVertex3f(0, 50, 0);
    glColor3f(0.0f, 0.0f, 1.0f);
    glVertex3f(50, -50, 0);
    glEnd();
    glFlush();
}
void main(void)
{
    glutInitDisplayMode(GLUT_SINGLE);
    glutInitWindowSize(400, 400);
    glutCreateWindow("Simple");
    glClearColor(0, 0, 0, 0);
    glOrtho(-100.0, 100.0, -100.0, 100.
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    glutDisplayFunc(RenderScene);
    glutMainLoop();
}
```



ЛИСТИНГ 10. Треугольник без интерполяции цветов.

```
void RenderScene(void)
{
    glClear(GL_COLOR_BUFFER_BIT);
    glShadeModel(GL_FLAT);
    glBegin(GL_TRIANGLES);
    glColor3f(0.0f, 1.0f, 0.0f);
    glVertex3f(-50, -50, 0);
    glColor3f(1.0f, 0.0f, 0.0f);
    glVertex3f(0, 50, 0);
    glColor3f(0.0f, 0.0f, 1.0f);
    glVertex3f(50, -50, 0);
    glEnd();
    glFlush();
}
void main(void)
{
    glutInitDisplayMode(GLUT_SINGLE);
    glutInitWindowSize(400, 400);
    glutCreateWindow("Simple");
    glClearColor(0, 0, 0, 0);
    glOrtho(-100.0, 100.0, -100.0, 100.0,
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    glutDisplayFunc(RenderScene);
    glutMainLoop();
}
```



ЗАПОЛНЕНИЕ МНОГОУГОЛЬНИКОВ.

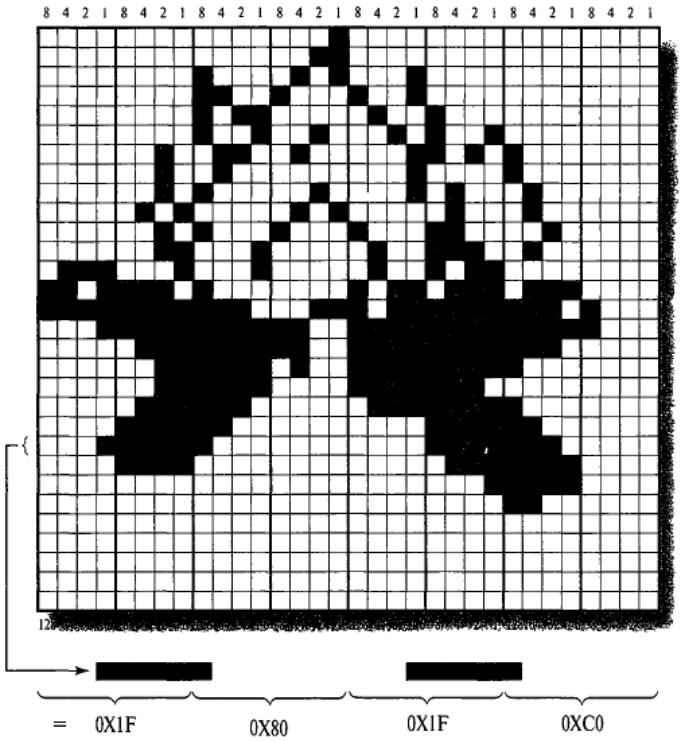
Существует два метода наложения узора на сплошные многоугольники. Обычно применяется наложение текстуры, когда изображение отображается на поверхность многоугольника. Другой способ заключается в задании фактурного узора, как это было для линий. Фактурный узор многоугольника — это не более, чем монохромное растровое изображение 32 x 32, используемое как узор-заполнитель. Чтобы активизировать заполнение многоугольника фактурой, нужно вызывать следующую функцию: glEnable(GL_POLYGON_STIPPLE);

После этого вызывается такая функция**:**

glPolygonStipple(pBitmap);

pBitmap — это указатель на область данных, содержащую узор-заполнитель. Далее все многоугольники заполняются с помощью узора, заданного функцией pBitmap (GLubyte *). Этот узор подобен используемому в наложении фактуры на линию, только в этот раз бу-фер должен быть достаточно большим, чтобы вместить узор 32 x 32. Кроме того, биты считываются начиная со старшего разряда, т.е. в обратном, по сравнению с линиями, порядке. На рис. 38 показан растровый образ костра, который используется как узор-заполнитель.

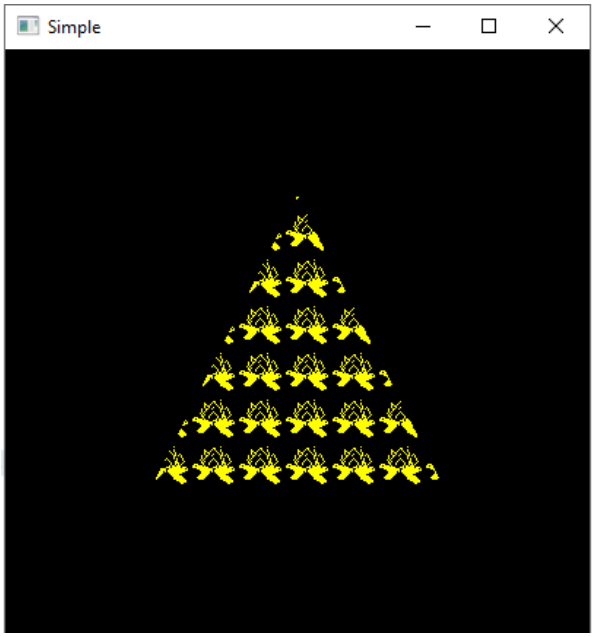
Чтобы построить маску, представляющую данный узор, записывают его снизу вверх по одной строке. К счастью, в отличие от узоров-заполнителей линий данные по умолчанию интерпретируются так, как записаны: первым читается самый старший бит. Таким образом, слева направо можно прочитать все байты и сохранить их в массиве величин типа GLubyte**,** достаточно большом, чтобы вместить 32 строки по 4 байт в каждой.



В простых случаях можно битовую маску вообще задать в цикле.

ЛИСТИНГ 11. Определение маски костра

```
#include "glut.h"
GLubyte fire[128] = { 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0xc0,
0x00, 0x00, 0x01, 0xf0,
0x00, 0x00, 0x07, 0xf0,
0x0f, 0x00, 0x1f, 0xe0,
0x1f, 0x80, 0x1f, 0xc0,
0x0f, 0xc0, 0x3f, 0x80,
0x07, 0xe0, 0x7e, 0x00,
0x03, 0xf0, 0xff, 0x80,
0x03, 0xf5, 0xff, 0xe0,
0x07, 0xfd, 0xff, 0xf8,
0x1f, 0xfc, 0xff, 0xe8,
0xff, 0xe3, 0xbf, 0x70,
0xde, 0x80, 0xb7, 0x00,
0x71, 0x10, 0x4a, 0x80,
0x03, 0x10, 0x4e, 0x40,
0x02, 0x88, 0x8c, 0x20,
0x05, 0x05, 0x04, 0x40,
0x02, 0x82, 0x14, 0x40,
0x02, 0x40, 0x10, 0x80,
0x02, 0x64, 0x1a, 0x80,
0x00, 0x92, 0x29, 0x00,
0x00, 0xb0, 0x48, 0x00,
0x00, 0xc8, 0x90, 0x00,
0x00, 0x85, 0x10, 0x00,
0x00, 0x03, 0x00, 0x00,
0x00, 0x00, 0x10, 0x00 };
```



```
void RenderScene(void)
{
	glClear(GL_COLOR_BUFFER_BIT);
glColor3f(1.0f, 1.0f, 0.0f);
// Активизируется заполнение многоугольника
glEnable(GL_POLYGON_STIPPLE);
// Задается узор-заполнитель
glPolygonStipple(fire);
	glBegin(GL_TRIANGLES);

	glVertex3f(-50, -50, 0);

	glVertex3f(0, 50, 0);

	glVertex3f(50, -50, 0);
	glEnd();
	glFlush();
}
void main(void)
{
	glutInitDisplayMode(GLUT_SINGLE);
	glutInitWindowSize(400, 400);
	glutCreateWindow("Simple");
	glClearColor(0, 0, 0, 0);
	glOrtho(-100.0, 100.0, -100.0, 100.0, -1.0, 1.0);
	glMatrixMode(GL_PROJECTION);
	glLoadIdentity();
	glutDisplayFunc(RenderScene);
	glutMainLoop();
}
```

АНИМАЦИЯ

Часто нужно перемещать или поворачивать сцены, создавать анимационные эффекты. Возвращаясь к рассмотренному выше примеру с нарисованным квадратом, можно изменить его так, чтобы он рикошетом отскакивал от сторон окна. Можно создать цикл, который непрерывно меняет координаты объекта, перед вызовом функции RenderScene. В результате квадрат будет перемещаться в пределах окна.

Библиотека GLUT позволяет регистрировать функцию обратного вызова, которая облегчает установку простых анимированных последовательностей: glutTimerFunc принимает имя функции, которую нужно вызывать, и время ожидания до вызова функции.

void glutTimerFunc(unsigned int msecs, void (*func)(int value), int value);

В данном коде указывается, что GLUT должна ожидать msecs миллисекунд перед вызовом функции func. Параметру value можно передать определенное пользователем значение. Функция, вызываемая с помощью этого таймера, имеет следующий прототип:

void TimerFunction (int value);

В отличие от таймера Windows, эта функция срабатывает только один раз. Чтобы создать непрерывную анимацию, следует обновить таймер в соответствующей функции.

В программе GLRect (из практической работы №1) можно заменить жестко запрограммированное положение прямоугольника переменными, а затем постоянно

модифицировать эти переменные в функции-таймере. В результате будет казаться, что прямоугольник движется по окну. Рассмотрим пример анимации такого типа.

ЛИСТИНГ 12. Анимированный квадрат

```
// подключаем заголовочные файлы библиотек
#include "glut.h"
// Исходное положение и размер прямоугольника
GLfloat x1 = 0.0f; GLfloat y1 = 0.0f;
GLfloat xstep = 1.0f; GLfloat ystep = 1.0f;

void RenderScene(void)
{
	// Очищаем окно, используя текущий цвет очистки
	glClear(GL_COLOR_BUFFER_BIT);
	// В качестве текущего цвета рисования задает красный //RGB
	glColor3f(1.0f, 0.0f, 0.0f);
	// Рисует прямоугольник, закрашенный текущим цветом
	glRectf(x1, y1, x1 + 25, y1 - 25);
	// Устанавливает поле просмотра с размерами окна
	glViewport(0, 0, 500, 500);
	// Обновляет систему координат
	glOrtho(-100.0, 100.0, -100, 100, 1.0, -1.0);
	glMatrixMode(GL_PROJECTION);
	glLoadIdentity();
	// Очищает очередь текущих команд и переключает буферы
	glutSwapBuffers();
}

void TimerFunction(int value)
{
	// Меняет направление на противоположное при подходе
	// к левому или правому краю
	if (x1 > 100 - 25 || x1 < -100)
		xstep = -xstep;
	// Меняет направление на противоположное при подходе
	// к верхнему или нижнему краю
	if (y1 > 100 || y1 < -100 + 25)
		ystep = -ystep;
	// Перемещает квадрат
	x1 += xstep; y1 += ystep;
	// Перерисовывает сцену с новыми координатами
	glutPostRedisplay();
	glutTimerFunc(33, TimerFunction, 1);
}

//Точка входа основной программы
void main(void)
{
	glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB);
	glutInitWindowSize(500, 500);
	glutCreateWindow("Bounce");
	glClearColor(0.0f, 0.0f, 1.0f, 1.0f);
	glutDisplayFunc(RenderScene);
	glutTimerFunc(33, TimerFunction, 1);
	glutMainLoop();
}
```

В листинге 13 модифицирован листинг 12, чтобы квадрат отскакивал от внутренних границ окна. Теперь отслеживается положение и размер прямоугольника, а также учитываются любые изменения размера окна.

ЛИСТИНГ 13. Анимированный квадрат с учетом меняющихся границ

```
// подключаем заголовочные файлы библиотек
#include "glut.h"
// Исходное положение и размер прямоугольника
GLfloat x1 = 0.0f; GLfloat y1 = 0.0f; GLfloat rsize = 25;
// Величина шага в направлениях x и y (число пиксе-лей,
// на которые на каждом шаге перемещается прямоуголь-ник)
GLfloat xstep = 1.0f; GLfloat ystep = 1.0f;
// Отслеживание изменений ширины и высоты окна
GLfloat windowWidth; GLfloat windowHeight;
//Вызывается для рисования сцены
void RenderScene(void)
{
	// Очищаем окно, используя текущий цвет очист-ки
	glClear(GL_COLOR_BUFFER_BIT);
	// В качестве текущего цвета рисования задает красный //RGB
	glColor3f(1.0f, 0.0f, 0.0f);
	// Рисует прямоугольник, закрашенный текущим цветом
	glRectf(x1, y1, x1 + rsize, y1 - rsize);
	// Очищает очередь текущих команд и переключает буферы
	glutSwapBuffers();
}
//Вызывается библиотекой GLUT в холостом состоянии (окно не меняет
//размера и не перемещается)
void TimerFunction(int value)
{
	// Меняет направление на противоположное при подходе
	// к левому или правому краю
	if (x1 > windowWidth - rsize || x1 < -windowWidth)
		xstep = -xstep;
	// Меняет направление на противоположное при подходе
	// к верхнему или нижнему краю
	if (y1 > windowHeight || y1 < -windowHeight + rsize)
		ystep = -ystep;
	// Перемещает квадрат
	x1 += xstep; y1 += ystep;
	// Проверка границ. Если окно меньше прямо-угольника,
	// который прыгает внутри, и прямоугольник об-наруживает
	// себя вне нового объема отсечения
	if (x1 > (windowWidth - rsize + xstep))
		x1 = windowWidth - rsize - 1; else if (x1 < -(windowWidth + xstep)) x1 = -window-
Width - 1; if (y1 > (windowHeight + ystep))

		y1 = windowHeight - 1; else if (y1 < -(windowHeight - rsize + ystep)) y1 = -win-
dowHeight + rsize - 1;
	// Перерисовывает сцену с новыми координатами
	glutPostRedisplay();
	glutTimerFunc(33, TimerFunction, 1);
}
//Задает состояние визуализации
void SetupRC(void)
{
	// Устанавливает в качестве цвета очистки синий
	glClearColor(0.0f, 0.0f, 1.0f, 1.0f);
}
//Вызывается библиотекой GLUT при изменении размеров окна
void ChangeSize(GLsizei w, GLsizei h)
{
	GLfloat aspectRatio;
	// Предотвращает деление на нуль
	if (h == 0) h = 1;
	// Устанавливает поле просмотра с размерами ок-на
	glViewport(0, 0, w, h);
	// Обновляет систему координат
```

```
	glMatrixMode(GL_PROJECTION);
	glLoadIdentity();
	//С помощью плоскостей отсечения (левая, пра-вая, нижняя,
	// верхняя, ближняя, дальняя) устанавливает объем отсечения
	aspectRatio = (GLfloat)w / (GLfloat)h;
	if (w <= h)
	{
		windowWidth = 100;
		windowHeight = 100 / aspectRatio;
		glOrtho(-100.0, 100.0, -windowHeight, windowHeight, 1.0, -1.0);
	}
	else
	{
		windowWidth = 100 * aspectRatio; windowHeight = 100;
		glOrtho(-windowWidth, windowWidth, -100.0, 100.0, 1.0, -1.0);
	}
	glMatrixMode(GL_MODELVIEW);
	glLoadIdentity();
}
//Точка входа основной программы
void main(void)
{
	glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB); glutCreateWindow("Bounce");
	glutDisplayFunc(RenderScene);
	glutReshapeFunc(ChangeSize);
	glutTimerFunc(33, TimerFunction, 1);
	SetupRC();
	glutMainLoop();
}
```

Задание 1. Используя все графические примитивы (перечисленные на странице 4), а также маску, создать изображение.

Пример для оценки сложности изображения.

Задание 2. Создать анимацию с помощью изменения **координат объекта**.

1) выполнить вращение квадрата относительно центра,

2) выполнить пульсирующее масштабирование квадрата относительно одной из его вершин,

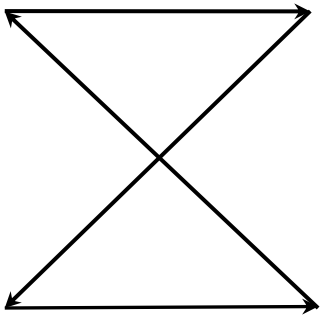
3) выполнить перемещение квадрата по окружности (по часовой стрелке),

4) выполнить перемещение квадрата по окружности (против часовой стрелки),

5) выполнить пульсирующее масштабирование квадрата с центром в произвольной координате,

6) выполнить пульсирующее масштабирование прямоугольника с центром в середине окна,

7)выполнить передвижение квадрата по траектории:



,

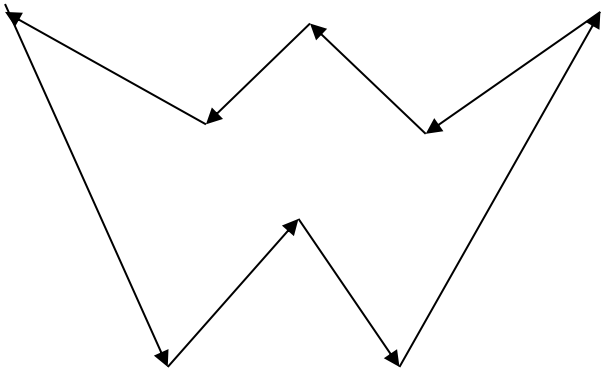
8) выполнить движение треугольника по горизонтальной восьмерке,

9) выполнить движение треугольника по горизонтальному эллипсу,

10) выполнить пульсирующее масштабирование движущегося прямоугольника,

11) выполнить движение треугольника по вертикальному эллипсу,

12) выполнить передвижение квадрата по траектории



13) выполнить передвижение квадрата по спирали,

14) выполнить вращение треугольника относительно центра,

15) выполнить вращение шестиугольника относительно центра,

16) выполнить пульсирующее масштабирование пятиугольника относительно левой верхней вершины,

17) выполнить пульсирующее масштабирование пятиугольника относительно правой нижней вершины,

18) выполнить пульсирующее масштабирование пятиугольника относительно его центра,

19) выполнить пульсирующее масштабирование треугольника относительно верхней вершины,

20) выполнить движение пятиугольника по траектории