Оглавление

[Лабораторная работа № 3. Основы программирования компьютерной графики при помощи OpenGL 1](#_Toc98286793)

[Знакомство с OpenGL 1](#_Toc98286794)

[Архитектура OpenGL 2](#_Toc98286795)

[Создание простейшего OpenGL приложения с использованием библиотеки GLUT. 3](#_Toc98286796)

[Создание простейшего OpenGL приложения с использованием подсистемы WGL. 11](#_Toc98286797)

[Используем ООП для облегчения работы с GLUT. 18](#_Toc98286798)

[Работа с OpenGL в WTL-проектах 25](#_Toc98286799)

[Практические задания 34](#_Toc98286800)

[Обязательные задания 35](#_Toc98286801)

[Дополнительные задания 40](#_Toc98286802)

# Лабораторная работа № 3. Основы программирования компьютерной графики при помощи OpenGL

## Знакомство с OpenGL

OpenGL (Open Graphics Library – открытая графическая библиотека) – спецификация, определяющая независимый от языка программирования кросс-платформенный программный интерфейс для написания приложений, использующих двухмерную и трехмерную графику.

Данная спецификация описывает набор функций и их точное поведение. На основе данной спецификации производители оборудования создают реализации – библиотеки функций, соответствующих набору функций спецификации. Реализация использует возможности оборудования там, где это возможно. Если аппаратура не позволяет реализовать какую-либо возможность, она должна быть эмулирована программно. Производители должны пройти специфические тесты (conformance tests — тесты на соответствие) прежде чем реализация будет классифицирована как OpenGL реализация. Таким образом, разработчикам программного обеспечения достаточно научиться использовать функции, описанные в спецификации, оставив эффективную реализацию последних разработчикам аппаратного обеспечения.

Существуют эффективные реализации OpenGL для Windows, Mac OS, Unix-платформ, Playstation 3. Данные реализации обычно предоставляются производителями видеоадаптеров и эффективно используют возможности последних.

Спецификация OpenGL пересматривается консорциумом ARB (Architecture Review Board), в состав которого входят ведущие производители программного и аппаратного обеспечения, заинтересованные в создании широкого и распространенного API.

### Архитектура OpenGL

Основным принципом работы OpenGL является получение набора векторных примитивов в виде точек, отрезков прямых, треугольников, многоугольников с последующей математической обработкой полученных данных и визуализацией в виде растрового изображения на экране или в памяти. Трансформации векторов и растеризация выполняются графическим конвейером (graphics pipeline), представляющий из себя дискретный автомат. Команды OpenGL либо добавляют графические примитивы на вход конвейера, либо конфигурируют конвейер для выполнения различных трансформаций.

OpenGL – низкоуровневой процедурный API, который требует от программиста диктовать точную последовательность шагов для построения результирующего растрового изображения. В отличие от дескрипторных подходов, когда вся трехмерная сцена передается в виде структуры данных (например, дерева), которое обрабатывается и строится на экране, императивный подход, используемый в OpenGL, требует от программиста глубокого знания трехмерной графики и математических моделей. С другой стороны, программисту предоставляется большая гибкость и свобода действий.

#### Расширения

Стандарт OpenGL, с появлением новых технологий , позволяет отдельным производителям добавлять в библиотеку функциональность с использованием механизма расширений. Расширения распространяются с помощью двух составляющих: заголовочный файл, в котором находятся прототипы новых функций и констант, а также драйвер устройства, поставляемого производителем. Прикладные приложения могут получить доступ к новой функции расширения, запросив у подсистемы OpenGL адрес функции по ее имени. Для системы Windows это функция [wglGetProcAddress](http://msdn.microsoft.com/en-us/library/dd374386%28VS.85%29.aspx).

#### Дополнительные библиотеки

Существует ряд библиотек, которые созданы поверх функциональности OpenGL или дополняющие его. Например, библиотека [GLU](http://ru.wikipedia.org/wiki/GLU), являющаяся практически стандартным дополнением OpenGL, построена поверх последней, используя функции OpenGL для реализации своих возможностей. Другие библиотеки (например, [GLFW](https://www.glfw.org/), [GLUT](http://www.opengl.org/resources/libraries/glut/)), созданы для реализации возможностей, недоступных OpenGL – построение пользовательского интерфейса, настройка контекста рисования, обработка сообщений от мыши и клавиатуры, работа с файлами.

#### Работа с OpenGL в различных оконных системах

Библиотека OpenGL изначально создавалась как кроссплатформенная библиотека, не зависящая от используемой оконной системы. Поэтому возможности OpenGL ограничиваются построением изображения в буфере кадра. Переносом содержимого буфера кадра на экран или другое графическое устройство занимается оконная система. По той же причине, в OpenGL отсутствуют функции по работе с файлами, а также устройствами ввода-вывода.

Для того, чтобы использовать возможности OpenGL на заданной программно-аппаратной платформе, приложение должно особым образом сконфигурировать оконную систему для работы с OpenGL. После того, как оконная система сконфигурирована должным образом, приложение может использовать команды OpenGL для построения изображения. В операционной системе Windows для связи OpenGL с оконной системой используется библиотека [WGL](http://msdn.microsoft.com/en-us/library/ee872084%28v=VS.85%29.aspx).

Существуют кроссплатформенные библиотеки, инкапсулирующие взаимодействие с оконной системой. Примеры таких библиотек – [GLFW](https://www.glfw.org/) и [GLUT](http://www.opengl.org/resources/libraries/glut/).

#### Работа с OpenGL в различных языках программирования

Для подтверждения независимости от языка программирования были [разработаны различные привязки](http://www.opengl.org/resources/bindings/) (binding) функций OpenGL для различных языков программирования.

Программы на языках C и С++ могут напрямую использовать заголовочный файл **GL/gl.h** для работы с функциями OpenGL, а также заголовочный файл **GL/glu.h** библиотеки GLU.

Все стандартные функции OpenGL (для языка Си) имеют префикс gl. Например, glVertex.

### Создание простейшего OpenGL приложения с использованием библиотеки GLUT.

Библиотека [GLUT](http://www.opengl.org/resources/libraries/glut/) (OpenGL Utility Toolkit) – библиотека утилит для приложений OpenGL, которая отвечает за системный уровень операций ввода-вывода при работе с операционной системой – создание и управление окном, создание простейших меню, опрос клавиатуры и мыши. Кроме того, в ней есть функции для рисования ряда геометрических примитивов: куб, сфера, чайник.

Целью создания библиотеки GLUT является создание кроссплатформенного кода, а также облегчение изучения OpenGL – для начала программирования OpenGL используя GLUT, требуется написать всего около полутора десятков строк кода.

Все функции библиотеки GLUT объявлены в заголовочном файле **GL/glut.h** и начинаются с префикса **glut**, например **glutCreateWindow**. Реализация библиотеки располагается для 32-разрядных систем Windows в динамической библиотеке glut32.dll. Ссылки на функции данной библиотеки находятся в библиотеке glut32.lib.

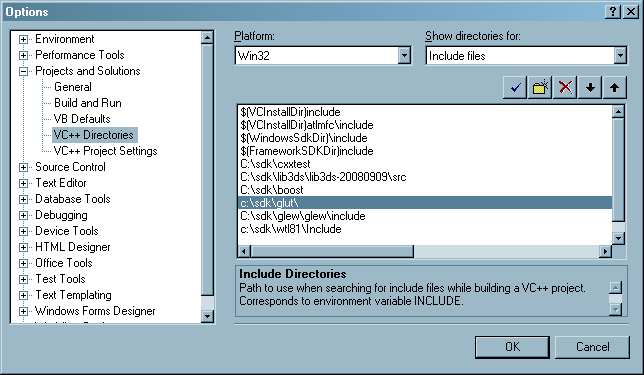
#### Установка GLUT в системе Windows и использованием в Visual C++ 2008

Библиотека GLUT по умолчанию не поставляется с компилятором Visual Studio и должна быть скачана со страницы проекта:

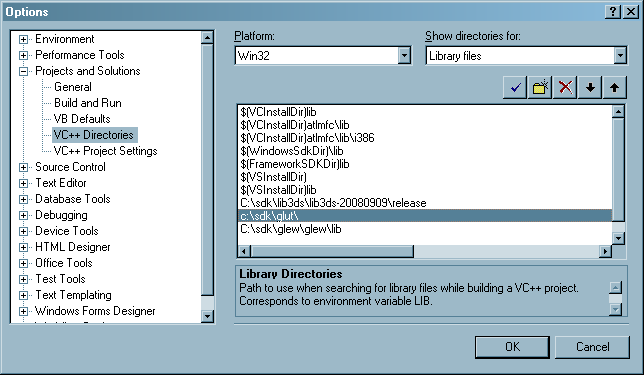
<http://www.opengl.org/resources/libraries/glut/>

В скачанном архиве нам понадобятся файлы glut.h, glut32.lib и glut32.dll.

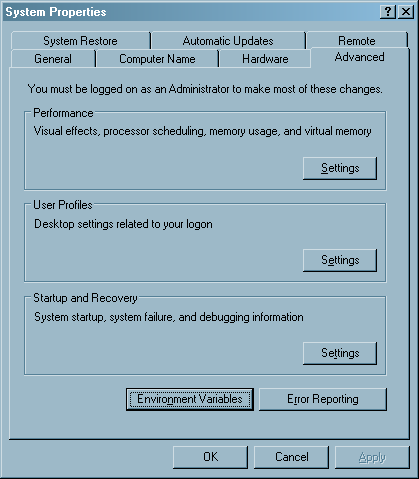
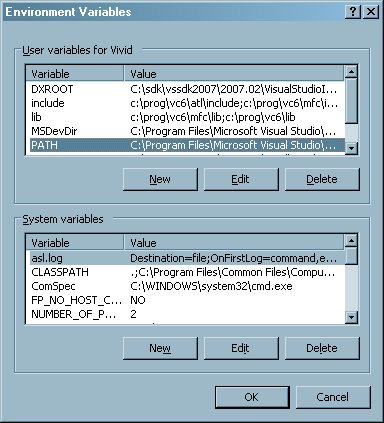
Дабы не распространять заголовочные и библиотечные файлы GLUT с исходным кодом каждой из использующей эту библиотеку программ, поместим файл **glut.h** в каталог **c:\sdk\glut\GL**[[1]](#footnote-1). Для того, чтобы компилятор Visual C++ мог найти данный заголовочный файл, укажем каталог **c:\sdk\glut** в разделе **Include files** диалога настройки среды Visual Studio (меню Tools->Options):

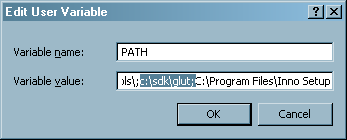


В каталог **c:\sdk\glut** скопируем файл библиотеки **glut32.lib**, содержащий ссылки на функции библиотеки внутри **glut32.dll**, и укажем данный каталог в разделе Library files диалога настройки среды Visual Studio:



Осталось разобраться с файлом **glut32.dll**. В данном файле находится реализация функций GLUT, необходимых для работы использующих библиотеку GLUT приложений. Не найдя данный файл операционная система не сможет загрузить Ваше приложение, поэтому поместим данный файл в каталог c:\sdk\glut[[2]](#footnote-2), и пропишем путь к данному каталогу в переменную окружения PATH текущего пользователя:



При распространении Вашего приложения (например, при сдаче ее преподавателю) может оказаться ситуация, когда на другом компьютере не окажется библиотеки GLUT, либо будет установлена похожим способом библиотека GLUT другой версии (не совместимой с GLUT, используемой Вашей программой). В результате, Ваше приложение не сможет запуститься.

Чтобы этого избежать – **всегда** **поставляйте glut32.dll в комплекте со своим приложением**, поместив его в каталог с исполняемым файлом своего приложения.

Ну вот, с установкой GLUT разобрались. Пришло время написать наше первое приложение с ее использованием.

#### Наш первый рисунок с использованием OpenGL

Первое, что нам нужно сделать, это подключить заголовочный файл <GL/glut.h> для использования функций библиотеки GLUT. Данный заголовочный файл также содержит в себе включение файла GL/gl.h и GL/glu.h, так что подключать их явно не обязательно. Кроме того, также будут автоматически добавлено использование библиотек opengl32.lib, glu32.lib и glut32.lib, так что явно добавлять ссылки на них в настройках проекта также не нужно.

Важное замечание. В виду особенностей реализации библиотеки GLUT, при совместном использовании библиотеки GLUT с заголовочным файлом **stdlib.h** стандартной библиотеки языка C (и другими, его использующими косвенным образом), **файл glut.h следует подключать после включения других стандартных файлов**. В противном случае Вы можете столкнуться со следующим сообщением об ошибке:

g:\programs\vs9.0\vc\include\stdlib.h(371) : error C2381: 'exit' : redefinition; \_\_declspec(noreturn) differs

c:\sdk\glut\gl\glut.h(146) : see declaration of 'exit'

g:\programs\vs9.0\vc\include\stdlib.h(371) : warning C4985: 'exit': attributes not present on previous declaration.

c:\sdk\glut\gl\glut.h(146) : see declaration of 'exit'

Работа с библиотекой GLUT начинается с ее инициализации при помощи функции **glutInit**. В данную функцию передается адреса параметров argc и argv, передаваемые в функцию main консольного приложения. Библиотека glut может использовать некоторые из параметров командной строки для инициализации своего состояния.

static const int WINDOW\_WIDTH = 800;

static const int WINDOW\_HEIGTH = 600;

const char WINDOW\_TITLE[] = "My first OpenGL application";

const float M\_PI = 3.14159265358979323846f;

int main(int argc, char\* argv[])

{

// Инициализируем библиотеку GLUT

glutInit(&argc, argv);

Следующий шаг – инициализация графического режима OpenGL, используемого GLUT. Мы будем использовать RGBA-режим с двойной буферизацией и буфером глубины. Начальный графический режим будет задан при помощи функции **glutInitDisplayMode**.

/\*

Инициализируем формат буфера кадра, используемый OpenGL:

- 32-битный буфер цвета,

- наличие буфера глубины

- двойная буферизация

\*/

glutInitDisplayMode(GLUT\_RGBA | GLUT\_DOUBLE | GLUT\_DEPTH);

Конфигурируем и создаем окно:

// Задаем начальные размеры окна

glutInitWindowSize(WINDOW\_WIDTH, WINDOW\_HEIGTH);

glutCreateWindow(WINDOW\_TITLE);

После создания окна необходимо указать библиотеке GLUT адреса обработчиков событий. При помощи **glutDisplayFunc** и **glutReshapeFunc** задаются адреса функций нашего приложения, вызываемых при необходимости отрисовки изображения в окне, а также при изменении размеров окна пользователем:

// Задаем функцию перерисовки кадра

glutDisplayFunc(&OnDrawScene);

// Задаем обработчик события изменения размеров окна

glutReshapeFunc(&OnResizeWindow);

Последнее, что нам нужно сделать – это запустить главный цикл GLUT при помощи функции glutMainLoop. Данная функция не возвращает управления, а сразу вызывает функцию **exit** при закрытии окна OpenGL.

// Запускаем главный цикл обработки

glutMainLoop();

// Сюда мы, к сожалению, уже не вернемся,

// т.к. glutMainLoop() не возвращает управления и при закрытии окна

// сразу вызывает функцию exit(), которая завершает работу приложения

return 0;

}

Итак, всего за 7 вызовов функций библиотеки GLUT мы выполнили необходимые шаги по подготовке контекста рисования OpenGL, создали окно и запустили цикл выборки сообщений. Осталось задать необходимые обработчики событий по обновлению содержимого окна и обрабатывающие изменение его размеров.

Функция OnDrawScene будет вызываться библиотекой GLUT всякий раз, когда потребуется перерисовать содержимое окна. Задача данной функции – сформировать в буфере кадра необходимое изображение и показать его на экране. Первое, что мы сделаем – это очистим содержимое буфер кадра, заполнив его желтым цветом. Функция [glClearColor](http://msdn.microsoft.com/en-us/library/dd318377%28VS.85%29.aspx) задает цвет очистки буфера кадра, а функция [glClear](http://msdn.microsoft.com/en-us/library/dd318372%28v=VS.85%29.aspx) – выполняет очистку заданных буферов, привязанных к буферу кадра. В нашем случае мы очистим буфер цвета и буфер глубины, передав для этого необходимые флаги в функцию glClear:

void OnDrawScene()

{

/\*

Задаем цвет очистки буфера кадра в формате RGBA (желтый)

\*/

glClearColor(1, 1, 0, 0);

// Очищаем буфер цвета и буфер глубины

glClear(GL\_COLOR\_BUFFER\_BIT | GL\_DEPTH\_BUFFER\_BIT);

Следующим нашим шагом будет рисование синей пятиконечной звезды. Звезда будет задана в виде замкнутой ломаной линии из пяти вершин:

OpenGL позволяет рисовать группы различных примитивов путем перечисления вершин примитивов в нужном порядке между вызовами функций [glBegin](http://msdn.microsoft.com/en-us/library/dd318361%28VS.85%29.aspx) и [glEnd](http://msdn.microsoft.com/en-us/library/dd318849%28v=VS.85%29.aspx). Координаты вершин задаются при помощи функции [glVertex](http://msdn.microsoft.com/en-us/library/dd374200%28v=VS.85%29.aspx).

Вершины нашей пятиконечной звезды располагаются на окружности в точках, соответствующих углам -90, 54, 198, 342 и 486 (126) градусов – каждая последующая вершина сдвинута на 360 \* 2 / 5 = 144 градуса относительно предыдущей. Оформим рисование пятиконечной звезды в виде функции Draw5PointStar:

const float M\_PI = 3.14159265358979323846f;

// Рисуем пятиконечную звезду

void Draw5PointStar(float xCenter, float yCenter, float radius)

{

static const float STEP = M\_PI \* 4 / 5;

// Начинаем новую группу примитивов (замкнутая ломаная линия)

glBegin(GL\_LINE\_LOOP);

float angle = -M\_PI / 2;

// Соединяем отрезками прямой линии точки, расположенные на окружности

// в точках, с углами: -90, 54, 198, 342, 486 (126) градусов

for (int i = 0; i < 5; ++i, angle += STEP)

{

float x = xCenter + radius \* cosf(angle);

float y = yCenter + radius \* sinf(angle);

// функция glVertex2f добавляет в текущую группу примитивов

// точку, лежащую на плоскости z = 0

// суффикс 2f в названии функции обозначает, что задаются 2 координаты

// x и y типа GLfloat

glVertex2f(x, y);

}

// Заканчиваем группу примитивов

glEnd();

}

С каждой вершиной OpenGL связан определенный набор атрибутов (координаты, цвет, нормаль, текстурные координаты, и др.). При вызове одной из версий функции glVertex в группу примитивов добавляется новая вершина, координаты которой передаются в glVertex, а остальные атрибуты копируются из текущих атрибутов вершины, которые приложение может изменить до вызова glVertex. Один из таких атрибутов – цвет – мы заменим при помощи функции [glColor](http://msdn.microsoft.com/en-us/library/dd318816%28v=VS.85%29.aspx) перед вызовом функции Draw5PointStar.

// Делаем текущий цвет вершины синим

// суффикс 3f в названии функции обозначает, что будут указаны

// 3 составляющих цвета (r, g, b) в формате float (от 0 до 1).

glColor3f(0, 0, 1);

// Рисуем пятиконечную звезду

Draw5PointStar(300, 200, 120);

Функцию glColor, а также некоторые другие функции также разрешается вызывать[[3]](#footnote-3) между glBegin/glEnd, что позволяет вершинам группы примитивов задавать различные атрибуты. В процессе растеризации примитивов OpenGL будет выполнять интерполяцию данных атрибутов по поверхности примитива. Попробуйте, например, вставить в функцию Draw5PointStar перед вызовом glVertex следующую строчку:

glColor3f(i / 5.0f, 1.0f - i / 10.0f, (i + 2) / 7.0f);

Поскольку при инициализации библиотеки GLUT мы указали использование двойной буферизации, команды OpenGL будут работать с теневым буфером, не видимым в данный момент. При помощи команды glutSwapBuffers происходит обмен[[4]](#footnote-4) содержимого теневого буфера кадра с содержимым окна.

Функция **OnResizeWindow**, которую мы зарегистрировали в качестве обработчика события об изменении размеров окна, вызывается GLUT всякий раз, когда пользователь изменяет размеры окна. Новые размеры окна передаются в эту функцию в виде параметров width и height.

В своем обработчике данного события мы будет изменять устанавливать размеры **порта просмотра**[[5]](#footnote-5) при помощи функции [glViewport](http://msdn.microsoft.com/en-us/library/dd374202%28VS.85%29.aspx). Следующим нашим действием будет настройка матрицы проецирования, которая задает параметры трансформации из **видовой системы координат**[[6]](#footnote-6) в **нормализованную систему координат устройства**[[7]](#footnote-7).

Более подробно преобразования систем координат в OpenGL будут рассмотрены на следующих занятиях.

При помощи функции [glMatrixMode](http://msdn.microsoft.com/en-us/library/dd373949%28VS.85%29.aspx) с параметром GL\_PROJECTION мы переключаем OpenGL на использование матрицы проецирования[[8]](#footnote-8). Дальнейшие операции, такие как загрузка единичной матрицы с использованием функции [glLoadIdentity](http://msdn.microsoft.com/en-us/library/dd373933%28VS.85%29.aspx) и последующее умножение ее на матрицу **ортографического преобразования**[[9]](#footnote-9) с использованием функции [glOrtho](http://msdn.microsoft.com/en-us/library/dd373965%28VS.85%29.aspx) применяются к матрице проецирования. Фактически, загрузка единичной матрицы с последующим умножением ее на матрицу ортографического преобразования эквивалентна загрузке матрицы ортографического преобразования:

, где I – единичная матрица.

Матрица проецирования нами будет задана таким образом, чтобы левому верхнему углу просмотра соответствовала точка (0, 0), правому нижнему – (viewportWidth, viewportHeight), ближняя плоскость отсечения располагалась в точке z=-1, а дальняя – в точке z=+1. Перед выходом из функции вновь сделаем текущей матрицу моделирования-вида.

int g\_windowWidth = WINDOW\_WIDTH;

int g\_windowHeight = WINDOW\_HEIGTH;

// Данная функция вызывается при изменении пользователем размеров окна

// а также непосредственно после создания окна

void OnResizeWindow(int width, int height)

{

g\_windowWidth = width;

g\_windowHeight = height;

/\*

Задаем положение и размеры видового порта (порта просмотра) в буфере кадра

Вывод примитивов не будет осуществляться за его пределы

\*/

glViewport(0, 0, width, height);

// Делаем текущей матрицей OpenGL матрицу проецирования

glMatrixMode(GL\_PROJECTION);

// Загружаем в нее единичную матрицу

glLoadIdentity();

/\*

И умножаем ее на матрицу ортографического проецирования такую,

что левому верхнему углу просмотра будет соответствовать

точка (0, 0), правому нижнему - (width, height),

а сцена будет включать в себя объект, расположенные по глубине в диапазоне

от -1 (сзади) до +1 (спереди)

\*/

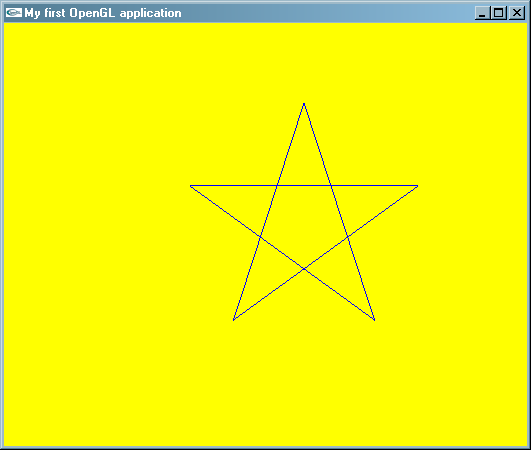
glOrtho(0, width, height, 0, -1, 1);

// Делаем текущей матрицей матрицу моделирования-вида

glMatrixMode(GL\_MODELVIEW);

}

В результате работы нашего приложения получится следующая картинка:



Разработанное нами приложение не использует явно функций, привязанных к конкретной оконной системе (все взаимодействие берет на себя библиотека GLUT), поэтому может быть без внесения каких-либо изменений исходного кода скомпилировано для работы на другой программно-аппаратной платформе, для которой есть реализация библиотеки GLUT.

### **Создание простейшего OpenGL приложения с использованием подсистемы WGL.**

Рассмотрим процесс создания сходного по функционалу приложения без использования библиотеки GLUT. Для этого нам понадобится проделать следующую последовательность действий:

* Создать главное окно приложения
* Выполнить инициализацию OpenGL
  + Получить дескриптор контекста устройства, связанного с клиентской областью окна
  + Выбрать желательный формат пикселей в контексте устройства, совместимый с OpenGL
  + Создать контекст рендеринга OpenGL для данного контекста устройства
  + Сделать данный контекст рендеринга активным
* Обновлять содержимое буфера кадра при получении события WM\_PAINT, а также обрабатывать другие необходимые сообщения (WM\_SIZE, WM\_ERASEBKGND)
* Перед разрушением окна освободить используемые системные ресурсы

#### Регистрация и создание главного окна приложения

Данный этап практически ничем не отличается от того, что мы делали на предыдущих занятиях. Единственное, чего требует Windows-реализация OpenGL, так это того, чтобы окно было создано с использованием стилей WS\_CLIPCHILDREN и WS\_CLIPSIBLINGS.

bool RegisterWndClass(HINSTANCE hInstance)

{

WNDCLASSEX wndClass =

{

sizeof(wndClass), //UINT cbSize;

CS\_HREDRAW | CS\_VREDRAW, //UINT style;

&WindowProc, //WNDPROC lpfnWndProc;

0, //int cbClsExtra;

0, //int cbWndExtra;

hInstance, //HINSTANCE hInstance;

NULL, //HICON hIcon;

LoadCursor(NULL, IDC\_ARROW), //HCURSOR hCursor;

(HBRUSH)(COLOR\_BTNFACE + 1), //HBRUSH hbrBackground;

NULL, //LPCTSTR lpszMenuName;

CLASS\_NAME, //LPCTSTR lpszClassName;

NULL, //HICON hIconSm;

};

return RegisterClassEx(&wndClass) != FALSE;

}

// Создаем главное окно приложения

HWND CreateMainWindow(HINSTANCE hInstance)

{

HWND hMainWindow = CreateWindowEx(

0,

CLASS\_NAME,

WINDOW\_TITLE,

WS\_OVERLAPPEDWINDOW |

WS\_CLIPCHILDREN | // Windows-реализация OpenGL требует,

WS\_CLIPSIBLINGS, // чтобы данные флаги окна были установлены

CW\_USEDEFAULT, CW\_USEDEFAULT,

CW\_USEDEFAULT, CW\_USEDEFAULT,

NULL,

NULL,

hInstance,

NULL);

return hMainWindow;

}

Функция WinMain и цикл выборки сообщений также не содержат ничего особенного:

int MainLoop()

{

MSG msg;

BOOL res;

while ((res = GetMessage(&msg, NULL, 0, 0)) != 0)

{

if (res == -1)

{

// произошла ошибка - нужно обработать ее и, вероятно,

// завершить работу приложения

}

else

{

// Если это сообщение о нажатии виртуальной клавиши,

// то добавляем в очередь сообщений сообщения, несущие информацию о

// коде вводимого пользователем символа

TranslateMessage(&msg);

// передаем сообщение в соответствующую оконную процедуру

DispatchMessage(&msg);

}

}

// сюда мы попадем только в том случае извлечения сообщения WM\_QUIT

// msg.wParam содержит код возврата,

// помещенный при помощи функции PostQuitMessage()

return msg.wParam;

}

int WINAPI WinMain(

HINSTANCE hInstance,

HINSTANCE /\*hPrevInstance\*/,

LPSTR /\*lpCmdLine\*/,

int nCmdShow)

{

// Регистрируем класс главного окна

if (!RegisterWndClass(hInstance))

{

return 1;

}

// Создаем главное окно приложения

HWND hMainWindow = CreateMainWindow(hInstance);

if (hMainWindow == NULL)

{

return 1;

}

// Показываем главное окно приложения

ShowWindow(hMainWindow, nCmdShow);

UpdateWindow(hMainWindow);

// Запускаем цикл выборки сообщений, пока не получим

// сигнал о завершении приложения

return MainLoop();

}

В оконной процедуре главного окна приложения нам нужно обработать сообщения WM\_CREATE, WM\_DESTROY, WM\_PAINT, WM\_ERASEBKGND и WM\_SIZE.

LRESULT CALLBACK WindowProc(

HWND hwnd,

UINT uMsg,

WPARAM wParam,

LPARAM lParam)

{

switch (uMsg)

{

HANDLE\_MSG(hwnd, WM\_DESTROY, OnDestroy);

HANDLE\_MSG(hwnd, WM\_PAINT, OnPaint);

HANDLE\_MSG(hwnd, WM\_CREATE, OnCreate);

HANDLE\_MSG(hwnd, WM\_ERASEBKGND, OnEraseBkgnd);

HANDLE\_MSG(hwnd, WM\_SIZE, OnSize);

}

return DefWindowProc(hwnd, uMsg, wParam, lParam);

}

#### Инициализация OpenGL

В обработчике сообщения WM\_CREATE вызовем функцию InitOpenGL, в которой сначала происходит подготовка контекста устройства для работы с OpenGL, а затем происходит создание контекста рендеринга[[10]](#footnote-10) OpenGL.

bool InitOpenGL(HWND hwnd, bool needDepth = true, bool needStencil = false);

// контекст устройства главного окна приложения для OpenGL

HDC g\_glDC = NULL;

// контекст рендеринга OpenGL

HGLRC g\_glRC = NULL;

BOOL OnCreate(HWND hwnd, LPCREATESTRUCT /\*lpCreateStruct\*/)

{

// Инициализируем OpenGL и возвращаем результат

return InitOpenGL(hwnd);

}

bool InitOpenGL(HWND hwnd, bool needDepth, bool needStencil)

{

// Настраиваем контекст устройства для рендеринга внутри окна

HDC hdc = CreateDeviceContext(hwnd, needDepth, needStencil);

if (!hdc)

{

return false;

}

HGLRC hglRC = CreateRenderContext(hdc);

if (!hglRC)

{

ReleaseDC(hwnd, hdc);

return false;

}

// Сохраняем дескриптоы в глобальных переменных

g\_glDC = hdc;

g\_glRC = hglRC;

return TRUE;

}

Процесс подготовки контекста устройства состоит из нескольких этапов.

Сначала при помощи функции **GetDC** мы получаем контекст устройства, связанный с клиентской областью окна.

Затем заполняем необходимые поля структуры [PIXELFORMATDESCRIPTOR](http://msdn.microsoft.com/en-us/library/dd368826%28VS.85%29.aspx), задающей наши предпочтения к формату пикселей буфера кадра OpenGL. Остальные поля данной структуры заполняем нулями. Следует отметить, что аппаратное ускорение визуализации OpenGL в ряде реализаций для системы Windows может поддерживаться только при включенной двойной буферизации.

Далее при помощи функции [ChoosePixelFormat](http://msdn.microsoft.com/en-us/library/dd318284%28v=VS.85%29.aspx) мы выбираем наиболее подходящий формат пикселей буфера кадра, поддерживаемый реализацией. Выбранный формат пикселей устанавливается в контексте устройства при помощи функции [SetPixelFormat](http://msdn.microsoft.com/en-us/library/dd369049(v=VS.85).aspx).

HDC CreateDeviceContext(HWND hwnd, bool depthBuffer, bool stencilBuffer)

{

HDC hdc = GetDC(hwnd);

if (hdc == NULL)

{

return NULL;

}

PIXELFORMATDESCRIPTOR pfd;

ZeroMemory(&pfd, sizeof(pfd));

// Заполняем поля структуры PIXELFORMATDESCRIPTOR,

// задающей желаемые параметры буфера кадра OpenGL

pfd.nSize = sizeof(pfd); // инициализируем размер структуры

pfd.nVersion = 1; // Windows поддерживает только версию 1

pfd.dwFlags =

PFD\_SUPPORT\_OPENGL | // буфер поддерживает рисование OpenGL

PFD\_DRAW\_TO\_WINDOW | // буфер поддерживает рисование в окне

PFD\_DOUBLEBUFFER | // используется двойная буферизация

PFD\_GENERIC\_ACCELERATED; // и аппаратное ускорение

pfd.iPixelType = PFD\_TYPE\_RGBA; // формат пикселей - RGBA

pfd.cColorBits = 32; // 32 бита на хранение цвета

pfd.cDepthBits =

depthBuffer ? 24 : 0; // если нужно, испольузем 24-битный буфер глубины

pfd.cStencilBits =

stencilBuffer ? 8 : 0;// если нужно, используем 8-битный буфер трафарета

// Выбираем наиболее подходящий формат пикселей,

// поддерживаемый контекстом устройства

int pixelFormat = ChoosePixelFormat(hdc, &pfd);

// Пытаемся установить данный формат пикселей в контексте устройства

if (

(pixelFormat == 0) ||

!SetPixelFormat(hdc, pixelFormat, &pfd)

)

{

// Если не получилось, то освобождаем контекст устройства и выходим

ReleaseDC(hwnd, hdc);

return NULL;

}

// Если все хорошо, то возвращаем настроенный контекст устройства

return hdc;

}

После того, как контекста устройства будет настроен, мы можем создать контекст визуализации для данного контекста устройства при помощи функции [wglCreateContext](http://msdn.microsoft.com/en-us/library/dd374379(v=VS.85).aspx) и сделать его текущим для данного потока с использованием функции [wglMakeCurrent](http://msdn.microsoft.com/en-us/library/dd374387(v=VS.85).aspx).

HGLRC CreateRenderContext(HDC hdc)

{

// Создаем контекст рендеринга OpenGL

HGLRC hrc = wglCreateContext(hdc);

if (!hrc)

{

return NULL;

}

// Пытаемся сделать данный контекст рендеринга текущим

// в контексте устройства

if (!wglMakeCurrent(hdc, hrc))

{

wglDeleteContext(hrc);

return NULL;

}

// Возвращаем контекст рендеринга в случае успеха

return hrc;

}

#### Деинициализация OpenGL

При разрушении окна следует освободить полученные нашим приложением ресурсы: удалить контекст рендеринга с использованием функции [wglDeleteContext](http://msdn.microsoft.com/en-us/library/dd374381(v=VS.85).aspx) и освободить контекст устройства с использованием функции **ReleaseDC**.

// Разрушаем контекст OpenGL и освобождаем контекст устройства

void DestroyOpenGL(HWND hwnd)

{

if (g\_glRC)

{

wglMakeCurrent(NULL, NULL);

wglDeleteContext(g\_glRC);

g\_glRC = NULL;

}

if (g\_glDC)

{

ReleaseDC(hwnd, g\_glDC);

g\_glDC = NULL;

}

}

void OnDestroy(HWND hWnd)

{

// Разрушаем OpenGL

DestroyOpenGL(hWnd);

PostQuitMessage(0);

}

#### Обновление содержимого буфера кадра

В обработчике WM\_PAINT мы вызываем BeginPaint и EndPaint, чтобы сообщить системе, что клиентская часть окна более не нуждается в перерисовке, и выполняем рисование изображения, используя команды, аналогичные предыдущей программы. Разница заключается лишь в том, что для обмена содержимого теневого и экранного буферов кадра используется функция [SwapBuffers](http://msdn.microsoft.com/en-us/library/dd369060(v=VS.85).aspx), специфичная для системы Windows.

void OnPaint(HWND hwnd)

{

PAINTSTRUCT ps;

HDC dc = BeginPaint(hwnd, &ps);

EndPaint(hwnd, &ps);

DrawScene();

}

void DrawScene()

{

/\*

Задаем цвет очистки буфера кадра (желтый)

\*/

glClearColor(1, 1, 0, 1);

// Очищаем буфер цвета и буфер глубины

glClear(GL\_COLOR\_BUFFER\_BIT | GL\_DEPTH\_BUFFER\_BIT);

// Делаем текущий цвет вершины синим

glColor3f(0, 0, 1);

// Рисуем эллипс (как ломаную линию)

DrawEllipse(300, 200, 150, 120);

// Обмениваем лицевой и нелицевой буферы кадра местами

SwapBuffers(g\_glDC);

}

В OpenGL отсутствуют функции для рисования криволинейных фигур (сделано это с целью повышения производительности), поэтому эллипс будет аппроксимирован в виде замкнутой ломаной линии, содержащей в нашем случае 360 вершин:

const float M\_PI = 3.14159265358979323846f;

void DrawEllipse(

float xCenter, float yCenter, float rx, float ry, int points = 360)

{

const float step = 2 \* M\_PI / points;

// Эллипс представлен в виде замкнутой ломаной линии, соединяющей

// points точек на его границе с шагом 2\*PI/points

glBegin(GL\_LINE\_LOOP);

for (float angle = 0; angle < 2 \* M\_PI; angle += step)

{

const float dx = rx \* cosf(angle);

const float dy = ry \* sinf(angle);

glVertex2f(dx + xCenter, dy + yCenter);

}

glEnd();

}

#### Обработка сообщения об изменении размеров окна

Тело обработчика сообщения WM\_SIZE практически ничем не отличается от функции OnResizeWindow предыдущй программы.

void OnSize(HWND /\*hwnd\*/, UINT /\*state\*/, int width, int height)

{

// Настраиваем порт просмотра

glViewport(0, 0, width, height);

// Делаем текущей матрицу проецирования и настраиваем ее параметры

glMatrixMode(GL\_PROJECTION);

glLoadIdentity();

glOrtho(0, width, height, 0, -1, 1);

// Делаем текущей матрицей матрицу моделирования-вида

glMatrixMode(GL\_MODELVIEW);

}

#### Устранение мерцания

Поскольку наше приложение обновляет клиентскую область окна целиком, нет необходимости в стирании окна перед перерисовкой. С этой целью в обработчике сообщения WM\_ERASEBKGND, дадим понять операционной системе, что очистку ей выполнять уже не нужно. Это также позволит устранить мерцание клиентской области при изменении размеров окна.

BOOL OnEraseBkgnd(HWND /\*hwnd\*/, HDC /\*hdc\*/)

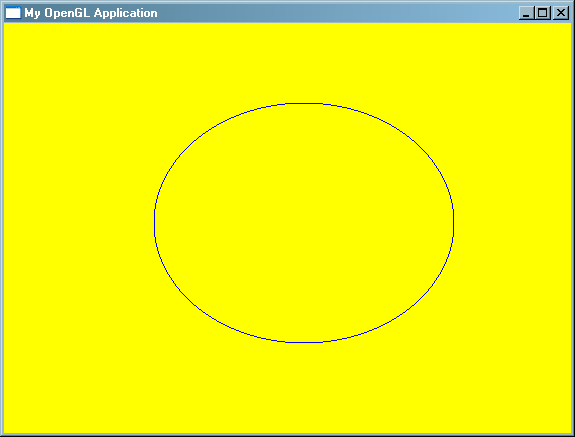
{

// Сообщаем Windows, что стирание заднего фона выполнено

return TRUE;

}

Результатом работы данной программы будет следующее изображение:



Как видно из данного примера, большую часть исходного кода приложения занимает создание окна и настройка контекста рендеринга.

### Используем ООП для облегчения работы с GLUT.

Код нашего GLUT-приложения можно разделить на две части: инициализация оконной подсистемы и, непосредственно, саму визуализацию. Код, выполняющий инициализацию, будет переходить из программы в программу практически без изменений, в то время как код, выполняющий визуализацию, будет в каждой программе изменяться.

Для улучшения повторного использования разработанного кода можно выделить создать класс CGLApplication, выполняющий инициализацию и настройку GLUT-приложения. Данный класс будет играть роль каркаса, от которого мы сможем породить класс конкретного приложения, перегрузив в нем необходимые виртуальные методы. Кроме того, это позволит свести к минимуму использование глобальных переменных, переместив их в класс конкретного приложения, а также решить проблему с освобождением ресурсов при выходе из приложения.

Класс CGLApplication:

class CGLApplication

{

public:

void MainLoop(void);

protected:

CGLApplication(

const char \* title,

int width = 0,

int height = 0,

bool needDepth = true,

bool needStencil = false

);

virtual ~CGLApplication(void);

virtual void OnInit();

// Методы-обработчики событий могут быть перегружены в классе-наследнике

virtual void OnDisplay() = 0;// данный метод всегда должен быть перегружен

virtual void OnReshape(int width, int height);

virtual void OnKeyboard(unsigned char key, int x, int y);

virtual void OnMouse(int button, int state, int x, int y);

virtual void OnMotion(int x, int y);

virtual void OnIdle();

// Инициирует перерисовку изображения в окне

void PostRedisplay();

// Установка обарботчика таймера

typedef void (GLUTCALLBACK \*TimerProc)(int value);

void SetTimer(int milliseconds, TimerProc proc, int value);

private:

void InitEventHandlers();

static void DisplayHandler();

static void ReshapeHandler(int width, int height);

static void KeyboardHandler(unsigned char key, int x, int y);

static void MouseHandler(int button, int state, int x, int y);

static void MotionHandler(int x, int y);

static void IdleHandler();

static CGLApplication \* m\_pApplication;

};

Конструктор класса CGLApplication выполняет инициализацию библиотеки GLUT, создание главного окна приложения и инициализацию внутренних обработчиков событий. Статическая переменная m\_pApplication класса CGLApplication используется для хранения адреса созданного экземпляра приложения, который понадобится обработчикам событий.

CGLApplication \* CGLApplication::m\_pApplication = NULL;

CGLApplication::CGLApplication(

const char \* title,

int width,

int height,

bool needDepth,

bool needStencil

)

{

// Только один экземпляр приложения разрешено создать

if (m\_pApplication)

{

assert(!"Only one instance of the application is permitted");

return;

}

m\_pApplication = this;

// Инициализируем библиотеку GLUT, подсовывая ей

// фиктивные параметры командой строки

int argc = 1;

char \*argv[] = {"", };

glutInit(&argc, argv);

// Задаем режим буфера кадра, запрошенный пользователем

glutInitDisplayMode(

GLUT\_RGBA |

GLUT\_DOUBLE |

(needDepth ? GLUT\_DEPTH : 0) |

(needStencil ? GLUT\_STENCIL : 0)

);

// Если пользователь указал размеры окна, сообщаем о них GLUT

if (width > 0 && height > 0)

{

glutInitWindowSize(width, height);

}

// Создаем окно приложения

glutCreateWindow(title);

// Задаем обработчики событий

InitEventHandlers();

}

void CGLApplication::InitEventHandlers()

{

// Инициализируем обработчики часто используемых событий

glutDisplayFunc(&DisplayHandler);

glutReshapeFunc(&ReshapeHandler);

glutKeyboardFunc(&KeyboardHandler);

glutMouseFunc(&MouseHandler);

glutMotionFunc(&MotionHandler);

glutIdleFunc(&IdleHandler);

// При желании можно добавить обработчики остальных

// событий, поддерживаемых glut

}

Сами внутренние обработчики представлены в виде приватных статических методов класса CGLApplication. Способ вызова статических методов класса ничем не отличается от вызова обычных функций. Кроме того, они имеют доступ к приватным и защищенным полям класса CGLApplication, что позволяет их использовать в качестве переходников между GLUT и конкретным экземпляром класса CGLApplication, в виртуальные защищенные методы которого будут перенаправлены вызовы из библиотеки GLUT. Вот здесь то нам и пригодится статическое поле m\_pApplication.

void CGLApplication::MotionHandler(int x, int y)

{

// Переадресовываем вызов обработчика события экземпляру класса приложения

m\_pApplication->OnMotion(x, y);

}

void CGLApplication::MouseHandler(int button, int state, int x, int y)

{

// Переадресовываем вызов обработчика события экземпляру класса приложения

m\_pApplication->OnMouse(button, state, x, y);

}

void CGLApplication::DisplayHandler()

{

// Переадресовываем вызов обработчика события экземпляру класса приложения

m\_pApplication->OnDisplay();

// Обмениваем местами теневой и лицевой буферы кадра

glutSwapBuffers();

}

void CGLApplication::IdleHandler()

{

// ничего не делаем

m\_pApplication->OnIdle();

}

void CGLApplication::ReshapeHandler(int width, int height)

{

// Переадресовываем вызов обработчика события экземпляру класса приложения

m\_pApplication->OnReshape(width, height);

}

void CGLApplication::KeyboardHandler(unsigned char key, int x, int y)

{

// Переадресовываем вызов обработчика события экземпляру класса приложения

m\_pApplication->OnKeyboard(key, x, y);

}

Виртуальные методы класса CGLApplication будучи перегруженными в классах наследниках, будут обрабатывать соответствующие события от библиотеки GLUT. В классе же CGLApplication для большинства из них будет пустая реализация

// Метод может быть перегружен в классе-наследнике

void CGLApplication::OnReshape(int width, int height)

{

// Обработчик по умолчанию всего лишь задает

// порт просмотра размером с клиентскую область окна

glViewport(0, 0, width, height);

}

// Метод может быть перегружен в классе-наследнике

void CGLApplication::OnKeyboard(unsigned char /\*key\*/, int /\*x\*/, int /\*y\*/)

{

}

// Метод может быть перегружен в классе-наследнике

void CGLApplication::OnMouse(

int /\*button\*/, int /\*state\*/, int /\*x\*/, int /\*y\*/)

{

}

// Метод может быть перегружен в классе-наследнике

void CGLApplication::OnMotion(int /\*x\*/, int /\*y\*/)

{

}

// Метод может быть перегружен в классе-наследнике

void CGLApplication::OnIdle()

{

}

// Метод может быть перегружен в классе-наследнике

void CGLApplication::OnInit()

{

}

Метод PostRedisplay используется для сообщения библиотеке GLUT о том, что следует обновить содержимое буфера кадра.

void CGLApplication::PostRedisplay()

{

// Инициируем перерисовку калпа

glutPostRedisplay();

}

Метод SetTimer позволяет классам-наследникам установить таймер на срабатывание через заданный временной интервал и вызвать некоторую функцию.

void CGLApplication::SetTimer(int milliseconds, TimerProc proc, int value)

{

// Устанавливаем таймер

glutTimerFunc(milliseconds, proc, value);

}

Метод MainLoop осуществлят вход в главный цикл обработки сообщений библиотеки glut, предварительно предоставляя классам-наследникам возможность выполнить инициализацию приложения.

void CGLApplication::MainLoop(void)

{

// Вызываем обработчик OnInit, который может быть

// перегружен в классе-наследнике

OnInit();

// И переходим в цикл обработки сообщений GLUT

glutMainLoop();

}

В классе CMyApplication останется лишь код, специфичный для нашего приложения:

#include "glapplication.h"

class CMyApplication :

public CGLApplication

{

public:

CMyApplication(const char \*title);

~CMyApplication(void);

protected:

// Перегружаем необходимые виртуальные методы родительского класса

virtual void OnInit();

virtual void OnDisplay(void);

virtual void OnReshape(int width, int height);

public:

static void FillEllipse(

float xCenter, float yCenter, float rx, float ry, int points = 360);

};

CMyApplication::CMyApplication(const char \*title)

:CGLApplication(title)

{

}

CMyApplication::~CMyApplication(void)

{

}

void CMyApplication::OnInit()

{

// Задаем цвет очистки буфера кадра

glClearColor(1, 1, 0, 1);

}

void CMyApplication::OnDisplay(void)

{

// Очищаем буфер цвета и буфер глубины

glClear(GL\_COLOR\_BUFFER\_BIT | GL\_DEPTH\_BUFFER\_BIT);

// Настраиваем цвет вершин

glColor3f(0, 0, 1);

// Рисуем закрашенный эллипс

FillEllipse(150, 120, 100, 90);

}

void CMyApplication::OnReshape(int width, int height)

{

// Настраиваем порт просмотра

glViewport(0, 0, width, height);

// Делаем текущей матрицу проецирования и настраиваем ее параметры

glMatrixMode(GL\_PROJECTION);

glLoadIdentity();

glOrtho(0, width, height, 0, -1, 1);

// Делаем текущей матрицей матрицу моделирования-вида

glMatrixMode(GL\_MODELVIEW);

}

Метод CMyApplication::FillEllipse рисования эллипса использует группу примитивов GL\_TRIANGLE\_FAN (веер из треугольников) для рисования закрашенного эллипса. Способ задания вершин группы примитивов Triangle Fan показан на следующем рисунке:

// Рисуем закрашенный эллипс

void CMyApplication::FillEllipse(

float xCenter, float yCenter, float rx, float ry, int points)

{

const float step = 2 \* M\_PI / points;

// Эллипс представлет в виде "веера" из треугольников

glBegin(GL\_TRIANGLE\_FAN);

// Начальная точка веера располагается в центре эллипса

glVertex2d(xCenter, yCenter);

// Остальные точки - равномерно по его границе

for (float angle = 0; angle <= 2 \* M\_PI; angle += step)

{

float a = (fabsf(angle - 2 \* M\_PI) < 1e-5) ? 0 : angle;

const float dx = rx \* cosf(a);

const float dy = ry \* sinf(a);

glVertex2f(dx + xCenter, dy + yCenter);

}

glEnd();

}

Для использования класса CMyApplication достаточно просто создать его экземпляр и вызвать его метод MainLoop.

#include "MyApplication.h"

// Создаем экземпляр нашего приложения

CMyApplication app("test");

int \_tmain(int argc, \_TCHAR\* argv[])

{

// и запускаем его

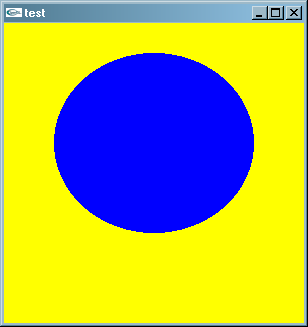
app.MainLoop();

return 0;

}

Объявление класса CMyApplication в области глобальных переменных обуславливается тем, что в таком случае при выходе из функции glutMainLoop при закрытии окна по exit будет вызван деструктор класса CMyApplication. В противном случае (переменная app объявлена внутри функции \_tmain) деструктор при выходе из приложения по exit() вызван не будет.

Результат работы программы представлен на следующем рисунке:



Наше текущее достижение - мы вынести код, ответственный за инициализацию OpenGL в базовый класс CGLApplication, оставив в классе, производном от него, специфичный для конкретного приложения код. Вместо глобальной области, данные приложения могут быть размещены внутри класса CMyApplication. Кроме того, деструктор производного класса может выполнить освобождение ресурсов при выходе из приложения, чего мы не могли[[11]](#footnote-11) сделать в первом приложении в предыдущем GLUT-приложении.

### Работа с OpenGL в WTL-проектах[[12]](#footnote-12)

Библиотека WTL предоставляет больше возможностей по созданию пользовательского интерфейса графических приложений, нежели GLUT. Рассмотрим пригодный для повторного использования способ работы с OpenGL из WTL-приложений.

#### Выносим код инициализации и деинициализации OpenGL в отдельный класс

Создадим класс COpenGL, в котором разместим код, инициализирующий и деинициализирующий OpenGL. Благодаря использованию классов WTL, работа с контекстом устройства осуществляется проще, нежели с использованием «голого» WinAPI.

class COpenGL

{

public:

COpenGL(HWND hwnd, bool needDepth = true, bool needStencil = false);

~COpenGL(void);

void Destroy(void);

void SwapBuffers();

private:

HWND m\_hwnd;

HGLRC m\_rc;

CClientDC m\_dc;

};

Конструктор классы выполняет настройку контекста устройства и создание контекста рендеринга.

COpenGL::COpenGL(HWND hwnd, bool needDepth, bool needStencil)

:m\_dc(hwnd)

,m\_rc(NULL)

,m\_hwnd(hwnd)

{

PIXELFORMATDESCRIPTOR pfd;

ZeroMemory(&pfd, sizeof(pfd));

// Заполняем поля структуры PIXELFORMATDESCRIPTOR,

// задающей желаемые параметры буфера кадра OpenGL

pfd.nSize = sizeof(pfd); // инициализируем размер структуры

pfd.nVersion = 1; // Windows поддерживает только версию 1

pfd.dwFlags =

PFD\_SUPPORT\_OPENGL | // буфер поддерживает рисование OpenGL

PFD\_DRAW\_TO\_WINDOW | // буфер поддерживает рисование в окне

PFD\_DOUBLEBUFFER | // используется двойная буферизация

PFD\_GENERIC\_ACCELERATED; // и аппаратное ускорение

pfd.iPixelType = PFD\_TYPE\_RGBA; // формат пикселей - RGBA

pfd.cColorBits = 32; // 32 бита на хранение цвета

pfd.cDepthBits =

needDepth ? 24 : 0;// если нужно, испольузем 24-битный буфер глубины

pfd.cStencilBits =

needStencil ? 8 : 0;// если нужно, используем 8-битный буфер трафарета

int pixelFormat = m\_dc.ChoosePixelFormat(&pfd);

if (!pixelFormat)

{

Destroy();

return;

}

if (!m\_dc.SetPixelFormat(pixelFormat, &pfd))

{

Destroy();

return;

}

m\_rc = wglCreateContext(m\_dc);

if (m\_rc)

{

if (!wglMakeCurrent(m\_dc, m\_rc))

{

Destroy();

}

}

}

Деструктор и метод Destroy() освобождают системные ресурсы, связанные с OpenGL.

COpenGL::~COpenGL(void)

{

Destroy();

}

// Освобождаем контекст устройства и ресурсы, связанные с OpenGL

void COpenGL::Destroy(void)

{

if (m\_rc)

{

wglMakeCurrent(NULL, NULL);

wglDeleteContext(m\_rc);

m\_rc = NULL;

}

if (m\_dc)

{

::ReleaseDC(m\_hwnd, m\_dc.Detach());

}

}

Метод SwapBuffers обменивает теневой и лицевой буферы кадра местами.

// Выполняем обмен лицевого и теневого буферов кадра

void COpenGL::SwapBuffers()

{

if (m\_dc && m\_rc)

{

::SwapBuffers(m\_dc);

}

}

#### Создаем реализацию видового окна, поддерживающего OpenGL-рендеринг

На этом можно было бы остановиться и напрямую использовать класс COpenGL в нашем приложении. Однако мы пойдем дальше. Создадим шаблонный класс **CGLViewImpl**, на базе класса **CWindowImpl**, предоставляющий реализацию видового окна с поддержкой OpenGL. На базе данного класса приложение может создавать конкретные видовые окна.

#include "OpenGL.h"

template <

class T, class TBase = CWindow, class TWinTraits = CControlWinTraits>

class ATL\_NO\_VTABLE CGLViewImpl

: public ATL::CWindowImpl<T, TBase, TWinTraits>

{

public:

CGLViewImpl(bool needDepth = true, bool needStencil = false)

:m\_needDepth(needDepth) // Запоминаем параметры буфера кадра

,m\_needStencil(needStencil)

{

}

DECLARE\_WND\_CLASS(NULL);

// Обрабатываем минимум необходимых сообщений

BEGIN\_MSG\_MAP(CGLViewImpl)

MESSAGE\_HANDLER(WM\_DESTROY, OnDestroy)

MESSAGE\_HANDLER(WM\_CREATE, OnCreate)

MESSAGE\_HANDLER(WM\_PAINT, OnPaint)

MESSAGE\_HANDLER(WM\_ERASEBKGND, OnEraseBkgnd)

END\_MSG\_MAP()

protected:

// Данный метод должен быть перегружен в классах-наследниках

virtual void DrawScene() = 0;

virtual void AfterCreate()

{

// Ничего не делаем - можно перегрузить в классе наследнике

}

virtual void BeforeDestroy()

{

// Ничего не делаем - можно перегрузить в классе наследнике

}

private:

LRESULT OnPaint

(UINT /\*uMsg\*/, WPARAM /\*wParam\*/, LPARAM /\*lParam\*/, BOOL& /\*bHandled\*/)

{

CPaintDC dc(\*this);

// Если OpenGL был проинициализирован, то вызываем метод рисования сцены

if (m\_pGL.get())

{

DrawScene();

m\_pGL->SwapBuffers();

}

return 0;

}

LRESULT OnCreate

(UINT /\*uMsg\*/, WPARAM /\*wParam\*/, LPARAM /\*lParam\*/, BOOL& /\*bHandled\*/)

{

// Инициализируем OpenGL

m\_pGL.reset(new COpenGL(\*this, m\_needDepth, m\_needStencil));

AfterCreate();

return 0;

}

LRESULT OnDestroy

(UINT /\*uMsg\*/, WPARAM /\*wParam\*/, LPARAM /\*lParam\*/, BOOL& /\*bHandled\*/)

{

BeforeDestroy();

// Деинициализируем OpenGL

if (m\_pGL.get())

{

m\_pGL.release();

}

return 0;

}

LRESULT OnEraseBkgnd

(UINT /\*uMsg\*/, WPARAM /\*wParam\*/, LPARAM /\*lParam\*/, BOOL& /\*bHandled\*/)

{

// ничего не делаем, сообщая системе о том, что очистка

// клиентской области окна проведена

return 0;

}

private:

std::auto\_ptr<COpenGL> m\_pGL;

bool m\_needDepth;

bool m\_needStencil;

};

#### Создаем видовое окно нашего приложения

На базе класса CGLViewImpl в нашем WTL-приложении будет создан класс CMainView, представляющий собой класс видового окна приложения. Обратите внимание на карту сообщений данного класса – при помощи макроса **CHAIN\_MSG\_MAP** мы передаем не обрабатываемые классом CMainView сообщения карте сообщений родительского класса.

include "glviewimpl.h"

class CMainView : public CGLViewImpl<CMainView>

{

enum

{

ANIMATION\_TIMER = 1

};

public:

DECLARE\_WND\_CLASS(NULL)

virtual BOOL PreTranslateMessage(MSG\* pMsg);

BEGIN\_MSG\_MAP(CMainView)

MSG\_WM\_SIZE(OnSize)

MSG\_WM\_TIMER(OnTimer);

// вызываем карту сообщений родительского окна

CHAIN\_MSG\_MAP(CGLViewImpl<CMainView>)

END\_MSG\_MAP()

protected:

// Обновляем содержимое буфера кадра

virtual void DrawScene();

virtual void AfterCreate();

virtual void BeforeDestroy();

private:

void OnSize(UINT uMsg, CSize size);

void OnTimer(UINT\_PTR nIDEvent);

DWORD m\_lastTick;

};

На этот раз мы нарисуем что-нибудь более интересное, а именно вращающиеся «цветочки». Каждый цветочек будет представлен в виде искаженного круга, радиус которого – функция, зависящая от угла. Поскольку рисование круга вовлекает достаточно большое количество вычислений, воспользуемся [**дисплейными списками**](http://msdn.microsoft.com/en-us/library/dd374300%28v=VS.85%29.aspx) (Display Lists) - возможностью OpenGL, позволяющей сохранить последовательность вызовов функций OpenGL, а затем воспроизвести их произвольное количество раз. Такой подход может ускорить работу нашего приложения в некоторых реализациях OpenGL.

Использовать дисплейные списки очень просто. Сначала нужно сгенерировать имя для дисплейного списка (или нескольких дисплейных списков) при помощи функции [glGenLists](http://msdn.microsoft.com/en-us/library/dd373538(v=VS.85).aspx). Затем вызвать команды OpenGL между вызовами функций [glNewList](http://msdn.microsoft.com/en-us/library/dd373951(v=VS.85).aspx) и [glEndList](http://msdn.microsoft.com/en-us/library/dd318850(v=VS.85).aspx). Воспроизвести записанную последовательность команд OpenGL можно при помощи функции [glCallList](http://msdn.microsoft.com/en-us/library/dd318369(v=VS.85).aspx) или [glCallLists](http://msdn.microsoft.com/en-us/library/dd318371(v=VS.85).aspx).

Реализуем данный функционал внутри класса CFlower:

class CFlower

{

public:

CFlower(

float minRadius, float maxRadius,

int petals,

COLORREF centerColor, COLORREF petalColor);

void Draw()const;

~CFlower(void);

CFlower(CFlower const& f);

private:

// запрещаем присваивание объектов CFlower

CFlower& operator=(CFlower const&);

mutable GLuint m\_displayList;

float m\_minRadius;

float m\_maxRadius;

int m\_petals;

COLORREF m\_centerColor;

COLORREF m\_petalColor;

};

Конструктор класса сохраняет параметры цветка, а копирующий конструктор копирует все поля, за исключением идентификатора дисплейного списка (копия будет использовать собственный дисплейный список в случае необходимости).

CFlower::CFlower(

float minRadius, float maxRadius,

int petals,

COLORREF centerColor, COLORREF petalColor)

:m\_minRadius(minRadius)

,m\_maxRadius(maxRadius)

,m\_petals(petals)

,m\_centerColor(centerColor)

,m\_petalColor(petalColor)

,m\_displayList(0)

{

}

CFlower::CFlower(CFlower const& f)

:m\_minRadius(f.m\_minRadius)

,m\_maxRadius(f.m\_maxRadius)

,m\_petals(f.m\_petals)

,m\_centerColor(f.m\_centerColor)

,m\_petalColor(f.m\_petalColor)

,m\_displayList(0) // Дисплейный список у копии будет создан свой

{

}

CFlower::~CFlower(void)

{

if (m\_displayList)

{

glDeleteLists(m\_displayList, 1);

m\_displayList = 0;

}

}

Метод Draw выполняет рисование цветка. Отметим, что центр цветка всегда к точке (0, 0, 0). Для помещения цветка в нужное место мы в дальнейшем воспользуемся матрицей моделирования вида (но об этом чуть позже).

void CFlower::Draw()const

{

// если дисплейный список еще не был создан, то для начала создаем его

if (!m\_displayList)

{

// Генерируем новый дисплейный список для цветка

m\_displayList = glGenLists(1);

{

glNewList(m\_displayList, GL\_COMPILE);

static const float M\_PI = 3.14159265358979323846f;

// угловой шаг (для рисования каждого лепестка

// нам понадобится как минимум 20 вершин)

const float step = 2 \* M\_PI / (m\_petals \* 20);

glBegin(GL\_TRIANGLE\_FAN);

// Центральная вершина будет иметь цвет centerColor

// суффикс 3ub означает, что

// функция принимает 3 параметра типа unsigned byte

glColor3ub(

GetRValue(m\_centerColor),

GetGValue(m\_centerColor),

GetBValue(m\_centerColor));

glVertex2f(0, 0);

// вершины лепестка будут иметь цвет petalColor

glColor3ub(

GetRValue(m\_petalColor),

GetGValue(m\_petalColor),

GetBValue(m\_petalColor));

for (float angle = 0; angle <= 2 \* M\_PI + step / 2; angle += step)

{

// в дискретном мире компьютеров sin(2 \* PI) может отличаться

// от sin(0). Поэтому если angle подошел близко к 2\*PI,

// считаем его равным 0

// это нужно для того, чтобы начало и конец веера

// сошлись в одной точке

float a = (fabsf(angle - 2 \* M\_PI) < 1e-4) ? 0 : angle;

// вычисляем радиус искривленной окружности для данного угла

float radius =

0.5f \* (m\_maxRadius - m\_minRadius) \* cosf(a \* m\_petals) +

m\_minRadius;

float x = radius \* cosf(a);

float y = radius \* sinf(a);

// создаем новую вершину

glVertex2f(x, y);

}

glEnd();

}

// завершаем дисплейный список

glEndList();

}

// вызываем ранее созданный дисплейный список

glCallList(m\_displayList);

}

Здесь нам пригодилось поле **m\_displayList**, хранящее идентификатор дисплейного списка данного цветка. Метод Draw устроен так, что построение дисплейного списка выполняется только при первом вызове метода Draw. При всех остальных вызовах используется ранее созданный дисплейный список. Т.к. метод Draw объявлен константным, поле **m\_displayList** объявлено с квалификатором **mutable**.

Для создания анимации вращения цветков таймера нам понадобится таймер, который мы проинициализируем в виртуальном методе AfterCreate, вызываемом обработчиком сообщения WM\_CREATE класса CGLViewImpl, а уничтожим в методе BeforeDestroy(), вызываемом обработчиком сообщения WM\_DESTROY класса CGLViewImpl.

void CMainView::AfterCreate()

{

SetTimer(ANIMATION\_TIMER, 10);

m\_lastTick = GetTickCount();

}

void CMainView::BeforeDestroy()

{

KillTimer(ANIMATION\_TIMER);

}

void CMainView::OnTimer(UINT\_PTR nIDEvent)

{

RedrawWindow();

}

Метод OnSize также претерпел ряд изменений. Матрица проецирования настраивается таким образом, чтобы по координате Y видимая через порт просмотра область всегда располагалась в диапазоне от -1 до +1, а по координате X – в соответствии с соотношением сторон клиентской области. В результате содержимое окна будет масштабироваться при изменении его размеров.

void CMainView::OnSize(UINT uMsg, CSize /\*size\*/)

{

&uMsg;

CRect rc;

GetClientRect(rc);

int width = rc.Width();

int height = rc.Height();

// Настраиваем порт просмотра

glViewport(0, 0, width, height);

// Делаем текущей матрицу проецирования и настраиваем ее параметры

glMatrixMode(GL\_PROJECTION);

glLoadIdentity();

// область проецирования на порт просмотра по вертикали

// располагается от -1 до +1, а по горизонтали - пропорционально

// соотношению сторон клиентской области окна

double aspect = (double)width / height;

double viewHeight = 2;

double viewWidth = aspect \* viewHeight;

glOrtho(

-viewWidth / 2, +viewWidth / 2,

-viewHeight / 2, +viewHeight / 2,

-1, 1);

// Делаем текущей матрицей матрицу моделирования-вида

glMatrixMode(GL\_MODELVIEW);

}

Для обработки цветков в виртуальном методе DrawScene, вызываемом из обработчика WM\_PAINT класса CGLViewImpl, предатавив информацию о них в виде массива структур FlowerInfo:

void CMainView::DrawScene()

{

// стурктура хранит информацию о цветке, его положении,

// угле и скорости вращения

struct FlowerInfo

{

float x;

float y;

float angle;

float speed;

CFlower flower;

};

// статическая переменная функции будет инициализирована только раз

static FlowerInfo flowers[] =

{

{0, 0, 0, 34, CFlower(0.3f, 0.2f, 9, RGB(255, 0, 0), RGB(255, 255, 0))},

{0.5f, 0.8f, 0, -57,

CFlower(0.1f, 0.08f, 7, RGB(0, 0, 255), RGB(255, 0, 255))},

{-0.7f, 0.3f, 0, 49,

CFlower(0.2f, 0.11f, 13, RGB(125, 22, 255), RGB(255, 200, 133))},

{-0.4f, -0.5f, 0, 68,

CFlower(0.25f, 0.2f, 11, RGB(13, 239, 0), RGB(255, 200, 133))},

{0.7f, -0.4f, 0, -42,

CFlower(0.3f, 0.11f, 9, RGB(223, 0, 64), RGB(112, 220, 173))},

};

static const size\_t NUM\_FLOWERS = sizeof(flowers) / sizeof(\*flowers);

Перед рисованием проведем расчет времени, прошедшего с предыдущего кадра, а также очистку буфера кадра и загрузку единичной матрицы в матрицу моделирования вида.

DWORD currentTick = GetTickCount();

float delta = (currentTick - m\_lastTick) \* 0.001f;

m\_lastTick = currentTick;

// Очищаем буфер кадра

glClear(GL\_COLOR\_BUFFER\_BIT);

glLoadIdentity();

Для рисования цветов в цикле произведем приращение угла поворота цветка на величину, равную произведению угловой скорости вращения на интервал времени, прошедший с предыдущего када, а затем при помощи функций [glTranslate](http://msdn.microsoft.com/en-us/library/dd368646%28VS.85%29.aspx) и [glRotate](http://msdn.microsoft.com/en-us/library/dd374027%28v=VS.85%29.aspx) осуществим умножение матрицы моделирования-вида на матрицы переноса и поворота соответственно. Порядок применения трансформаций к вершинам объекта является обратным порядку умножения матриц. Следовательно, сначала к вершинам объекта будут применена трансформация поворота вокруг центра цветка на заданный угол, а затем операция переноса центра цветка в нужное место на сцене.

Данные трансформации будут применены к каждой вершине группы примитивов, составляющих цветок. Для того, чтобы трансформации, примененные к одному цветку, не оказывали влияния на последующие, текущая матрица до применения данных преобразований сохраняется в стеке матриц при помощи функции [glPushMatrix](http://msdn.microsoft.com/en-us/library/dd373984(v=VS.85).aspx), а после рисования цветка восстанавливается при помощи функции [glPopMatrix](http://msdn.microsoft.com/en-us/library/dd373977(v=VS.85).aspx).

// Рисуем и вращаем все цветы в массиве

for (size\_t i = 0; i < NUM\_FLOWERS; ++i)

{

FlowerInfo & flowerInfo = flowers[i];

// выполняем изменение угла вращения

flowerInfo.angle =

fmodf(flowerInfo.angle + flowerInfo.speed \* delta, 360);

// сохраняем текущую матрицу трансформации

glPushMatrix();

// вращаем (вокруг оси Z) и переносим цветок

glTranslatef(flowerInfo.x, flowerInfo.y, 0);

glRotatef(flowerInfo.angle, 0, 0, 1);

// рисуем цветок

flowerInfo.flower.Draw();

// восстанавливаем текущую матрицу

glPopMatrix();

}

}

Подведем итоги.

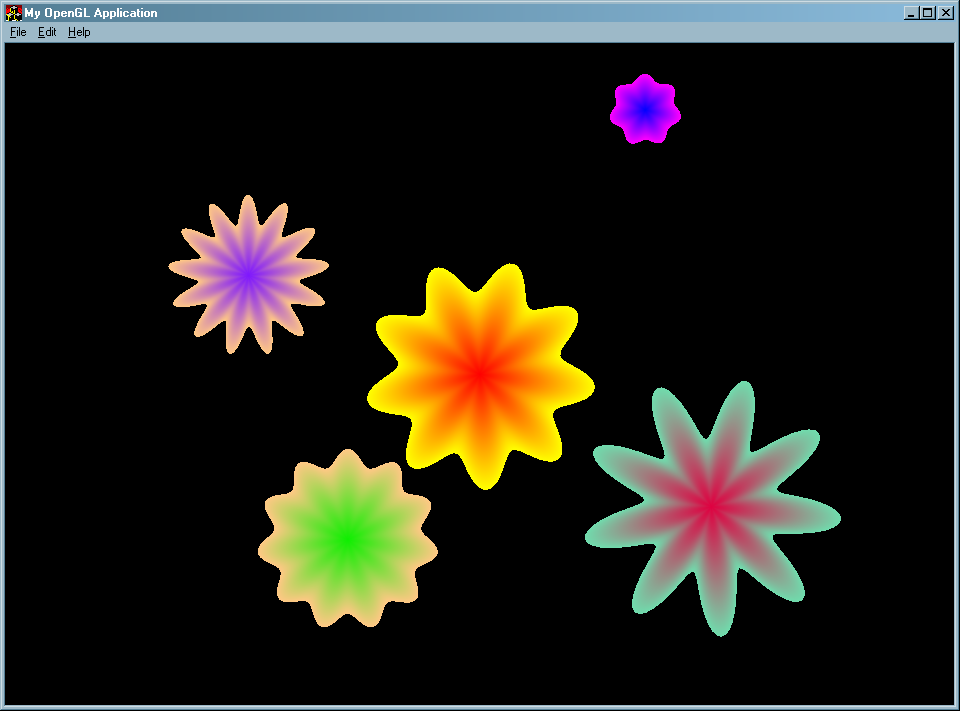
Итак, использование матрицы моделирования-вида позволяет применить различные преобразования к одному и тому же объекту без изменения перечня его вершин (напомню, что класс CFlower задает вершины лишь однажды – в момент инициализации дисплейного списка).

С помощью функций **glTranslate** и **glRotate** происходит умножение текущей матрицы на матрицы переноса и поворота соответственно. В OpenGL есть также функция [glScale](http://msdn.microsoft.com/en-us/library/dd368578(v=VS.85).aspx), умножающая текущую матрицу на матрицу масштабирования вдоль координатных осей x, y, z.

Порядок умножения матриц является обратным к порядку применения соответствующих преобразований вершин объекта. При преобразовании системы координат (например, при перемещении виртуальной камеры), напротив, порядок умножения матриц совпадает с порядком применения преобразований к системе координат.

Функции glPushMatrix и glPopMatrix позволяют временно сохранить текущую матрицу в стеке матриц и восстановить ее впоследствии. Для каждого типа матриц (моделирования-вида, проецирования, текстурных координат) в OpenGL имеется собственный стек матриц.

Результат работы программы представлен на следующем рисунке:



## Практические задания

Для получения оценки «**удовлетворительно**» необходимо выполнить часть обязательных заданий, набрав при этом **не менее 50 баллов**.

Для получения оценки «**хорошо**» необходимо выполнить часть обязательных заданий и, возможно, часть дополнительных, набрав при этом **не менее 100 баллов**.

Для получения оценки «**отлично**» необходимо выполнить часть обязательных и дополнительных заданий, набрав при этом **не менее 350 баллов**.

Дополнительные задания принимаются только после успешной защиты обязательных заданий.

### Обязательные задания

#### Задание 1

Разработайте приложение, инициализирующее OpenGL любым доступным вам способом. Приложение должно визуализировать график одной из следующих функций на заданном интервале.

График должен включать в себя координатные оси X и Y, оформленные в виде стрелочек с нанесенными на них делениями (цену делений подберите такой, чтобы на экране помещалось не менее 10 делений по каждой из координатных осей), а также сам график, точки которого соединены ломаной линией с шагом, обеспечивающим достаточно гладкий вид линии графика.

Цвет графика должен отличаться от цвета координатных осей.

При изменении размеров окна график должен помещаться внутри окна, сохраняя свои оригинальные пропорции.

##### Вариант 1 – Сумма гармонических колебаний – 10 баллов

на интервале

##### Вариант 2 – Парабола – 10 баллов

на интервале

##### Вариант 3 – Гипербола – 10 баллов

на интервале

Предусмотреть корректную обработку точки x=0

##### Вариант 4 – Первый замечательный предел – 10 баллов

на интервале

##### Вариант 5 – [Архимедова спираль](http://ru.wikipedia.org/wiki/%D0%90%D1%80%D1%85%D0%B8%D0%BC%D0%B5%D0%B4%D0%BE%D0%B2%D0%B0_%D1%81%D0%BF%D0%B8%D1%80%D0%B0%D0%BB%D1%8C) – 15 баллов

на интервале (полярная система координат)

##### Вариант 6 – [Логарифмическая спираль](http://ru.wikipedia.org/wiki/%D0%9B%D0%BE%D0%B3%D0%B0%D1%80%D0%B8%D1%84%D0%BC%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%B0%D1%8F_%D1%81%D0%BF%D0%B8%D1%80%D0%B0%D0%BB%D1%8C) – 15 баллов

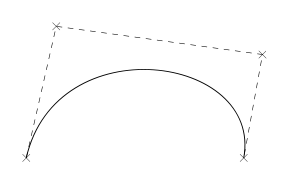
на интервале (полярная система координат)

##### Вариант 7 – [Кардиоида](http://ru.wikipedia.org/wiki/%D0%9A%D0%B0%D1%80%D0%B4%D0%B8%D0%BE%D0%B8%D0%B4%D0%B0) – 15 баллов

на интервале (полярная система координат)

##### Вариант 8 – [Кривая Безье](http://ru.wikipedia.org/wiki/%D0%9A%D1%80%D0%B8%D0%B2%D0%B0%D1%8F_%D0%91%D0%B5%D0%B7%D1%8C%D0%B5) 3 порядка (кубическая) – 25 баллов

Поскольку в OpenGL отсутствуют встроенные средства рисования криволинейных примитивов, потребуется аппроксимировать кубическую кривую Безье при помощи ломаной линии. На рисунке также потребуется изобразить местоположение контрольных точек кривой, соединенных пунктирными линиями



###### Бонус в 15 баллов за возможность изменения формы кривой

Бонус начисляется за возможность изменения формы кривой Безье путем изменения положения ее контрольных точек при помощи мыши.

#### Задание 2

Разработайте OpenGL-приложение, визуализирующее с использованием **векторных примитивов** OpenGL двухмерную картинку, соответствующую одному из перечисленных ниже вариантов.

* Приложение должно использовать режим двойной буферизации и не должно мерцать при изменении размеров окна, а также в случае анимации
* При изменении размеров окна должно сохраняться соотношение сторон объектов. Например, круги не должны превращаться в эллипсы. Достичь этого можно, настроив матрицу ортографического проецирования соответствующим образом.

**Фотографического сходства достигать не требуется: все-таки дисциплина связана с программированием, а не с рисованием**. Для решения поставленной задачи можно найти подходящее растровое изображение в Интернет и выполнить его аппроксимацию с помощью векторных примитивов OpenGL. Тем не менее, будет оцениваться качество исполнения результирующей картинки.

Для сокращения объема программного кода спроектируйте необходимые классы для представления, параметризации и визуализации графических примитивов более высокого уровня из примитивов Open GL и используйте их для построения изображения. Весьма полезной может оказаться возможность загрузки информации об изображении из внешнего файла, в котором были бы записаны команды рисования высокоуровневых примитивов. Это избавит от необходимости перекомпиляции приложения после каждой правки.

Обращайте внимание на расширяемость программы и лёгкость её модификации. Например, структура программы должна вам легко позволять нарисовать не один, а несколько экземпляров изображения в разных масштабах и разном месте окна. Разные экземпляры картинки могут отличаться не только размером, но и цветом отдельных частей.

##### Вариант 1 – Персонаж мультсериала «Смешарики» - 70 баллов



###### Бонус в 20 баллов за возможность перетаскивания картинки с помощью мыши

Бонус начисляется, если программа предоставляет возможность изменения положения картинки при помощи метода Drag and Drop.

##### Вариант 2 – Луг – 70 баллов

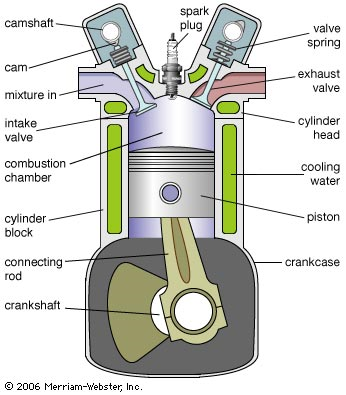
Луг, состоящий из различных цветов (не менее 3-4 типов) и травы (не менее 3-4 типов), над которыми порхают разноцветные бабочки (не менее 3-4 типов). Над лугом - небо и Солнце, а также несколько белых облаков.

###### Бонус в 50 баллов за анимацию

Бабочки перелетают с цветка на цветок, облака двигается под действием ветра, Солнце медленно садится за горизонт, в результате чего небо темнеет, и на нем появляются звезды и встает месяц.

##### Вариант 3 – Двигатель внутреннего сгорания в разрезе – 80 баллов

Программа визуализирует внутреннее устройство двигателя внутреннего сгорания, что-то типа такого (без надписей):



###### Бонус в 40 баллов за анимацию

Бонус начисляется за анимацию изображения, при котором видно открытие клапанов, вспышки топлива, вращение маховика и движение цилиндров, выпуск отработавших газов.

##### Вариант 4 - Аквариум с рыбками – 70 баллов

Программа визуализирует внутреннее содержимое аквариума, с находящимися в нем рыбками (3-4 вида) и водорослями. Дно аквариума выстлано камушками (не менее 5-6 видов).

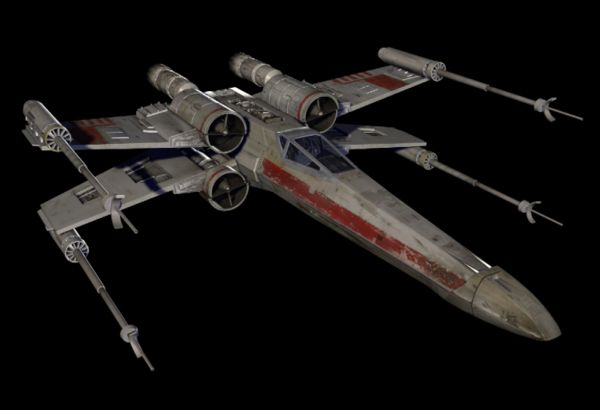
###### Бонус за анимацию – 30 баллов

Бонус начисляется за движение рыб влево-вправо, выпускание ими время от времени пузырьков воздуха, поднимающихся вверх.

##### Вариант 5 - Космический корабль из «Звездных Войн» - 70 баллов

Программа визуализирует один из космических кораблей фильма «Звездные Войны». Например:

X-Wing



Или тайский истребитель:

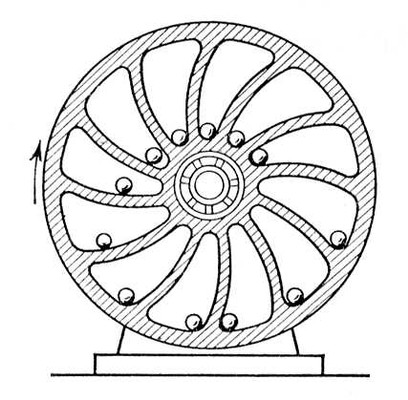


###### Бонус в 20 баллов за возможность перетаскивания изображения

Бонус начисляется за возможность перетаскивания изображения при помощи Drag and Drop

##### Вариант 6 – Схема работы одного из типичных вечных двигателей – 80 баллов

Например, такого (можно предложить и другую, достаточно известную модель):



Задание предполагает демонстрацию анимации работы «вечного двигателя», как он должен был бы работать по замыслу автора.

##### Вариант 7 – Аналоговые часы – 50 баллов

Цифры на циферблате (при их наличии) должны быть нарисованы при помощи примитивов OpenGL. Часы должна показывать текущее время (часы, минуты, секунды), совпадающее со временем на компьютере пользователя.



Внимание, программа должна корректно отображать положение часовой стрелки в зависимости от количества минут, прошедших с начала текущего часа. Если в 9:05 часовая стрелка ближе к 9, то в 9:50 она ближе к 10.

##### Вариант 8 – симуляция движения заряженных частиц – 80 баллов

В замкнутом пространстве находятся положительно (красные) и отрицательно (синие) заряженные частицы (не менее 20-25 штук) в виде цветных кружков со знаками + и -. Между частицами действуют силы Кулона, заставляющие противоположно заряженные частицы притягиваться, а одноименно заряженные – отталкиваться. При сильном приближении друг к другу на частицы также должны действовать силы отталкивания, не позволяя им сблизиться неограниченно близко.

Параметры системы (массы частиц, коэффициенты силы Кулона и т.п.) должны быть подобраны таким образом, чтобы движение частиц при их адекватном количестве имело упорядоченный характер.

###### Бонус в 10 баллов за возможность интерактивного добавления и удаления частиц.

Щелчок левой кнопкой мыши должен добавлять частицу с положительным зарядом, щелчок правой – частицу с отрицательным зарядом в точке щелчка. Если щелчок пришелся по существующей частице, частица удаляется. Клавиша Del – удаляет все частицы из системы.

##### Вариант 9 – Домик в деревне – 40 баллов

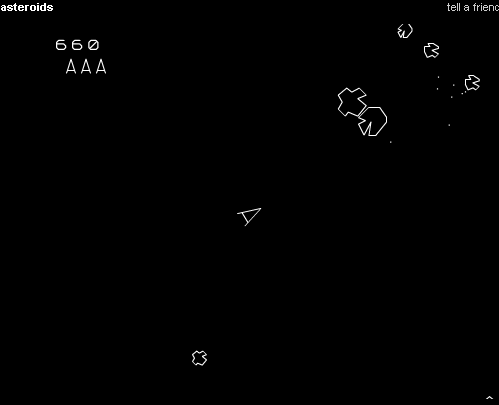
Программа должна визуализировать изображение деревянного дома с окном, забором и трубой, а также небом и солнцем

### Дополнительные задания

#### Задание 3

##### Вариант 1 - Asteroids – 250 баллов

Разработайте клон игры [**Asteroids**](http://en.wikipedia.org/wiki/Asteroids_%28video_game%29), в которой игроку предстоит провести небольшой космический корабль сквозь поле астероидов. Астероиды можно уничтожать из пушки, установленной на корабле, при этом большие астероиды при попадании раскалываются на средние, а те, в свою очередь, на мелкие. Мелкие астероиды после попадания в них из пушки уничтожаются бесследно. На прохождение игры игроку дается 3 попытки.



Кораблем можно управлять при помощи клавиш управления курсором. Клавиши «влево» и «вправо» вращают корабль против часовой стрелки, а клавиша «вверх» включает ускорение в направлении носа корабля (при этом видно огонь от двигателя). После отпускания клавиш корабль плавно замедляет скорость движения в заданном направлении и своё вращение. Как если бы в космосе на него действовала сила трения. Клавиша «Пробел» выпускает ракету по направлению движения космического корабля.

За уничтожение астероидов игроку начисляются призовые очки. Очки можно отображать при помощи примитивов OpenGL либо отображать в области заголовка окна. Столкновение корабля с астероидом приводит к потере одной из жизней. При уничтожении корабля он «разваливается» на части и вновь появляется в центре игрового поля (при наличии оставшихся жизней). Астероид, попавший в игрока, при столкновении также распадается на более мелкие части (либо уничтожается совсем, если распадаться уже некуда).

Как только жизней у игрока не останется, программа должна предложить начать игру заново, либо выйти из программы. Например, при помощи стандартного окна сообщения.

При определении коллизий патронов с астероидами понадобится определить, находится ли заданная точка внутри многоугольника (астероид, корабль и т.п.). Точка находится внутри многоугольника, если она находится внутри любого из треугольников, из которых можно построить заданный многоугольник.

Видео для иллюстрации геймплея игры: <https://www.youtube.com/watch?v=okRZNrqPjUk>

###### Бонус до 30 баллов за усовершенствование внешнего вида игры по сравнению с оригиналом.

При реализации игры приветствуется улучшение внешнего вида астероидов и космического корабля. Например, вместо «проволочной» графики можно визуализировать объекты путем закрашивания их внутренностей, добавить космическому кораблю и астероидам больше деталей, придать вращение астероидам вокруг собственной оси.

###### Бонус до 80 баллов за применение звуковых эффектов в игре

Бонус начисляется за проигрывание программой звуковых эффектов, соответствующих следующим событиям игры:

* Выстрел из пушки
* Попадание снаряда в астероид и его раскалывание
* Полное уничтожение астероида
* Взрыв корабля

Для проигрывания звуков можно воспользоваться, например, библиотекой [OpenAL](https://www.openal.org/), [Bass audio library](https://www.un4seen.com/) либо иными библиотеками.

##### Вариант 2 – Tetris – 250 баллов

Разработайте клон игры [Тетрис](http://ru.wikipedia.org/wiki/%D0%A2%D0%B5%D1%82%D1%80%D0%B8%D1%81), в котором в стакан шириной 10 и высотой 20 клеток падают случайные **разноцветные** фигурки, состоящие из четырех квадратов ([тетрамино](http://ru.wikipedia.org/wiki/%D0%A2%D0%B5%D1%82%D1%80%D0%B0%D0%BC%D0%B8%D0%BD%D0%BE)). При заполнении одной или нескольких линий стакана они удаляются, что приводит к падению вышестоящих линий, а игроку начисляются очки:

* За одну заполненную линию – 10 очков
* За две заполненные линии – 30 очков
* За три заполненные линии – 70 очков
* За четыре заполненные линии – 150 очков

Если падение очередной фигуры приведет к заполнению стакана, игра заканчивается и игроку предлагается начать игру заново, либо выйти из игры.

Пользователь может поворачивать падающие фигурки вокруг их центра на 90 градусов по часовой стрелке при помощи клавиши «курсор вверх», а также ускорять их падение при помощи клавиши «курсор вниз». Клавиши «курсор влево» и «курсор вправо» позволяют перемещать фигуру влево или вправо.

При помощи клавиши “P” игрок может поставить игру на паузу.

Информационная панель в игре отображает номер текущего уровня, количество линий, которое нужно заполнить для перехода на следующий уровень, набранное количество баллов и тип следующей фигуры.

Целью игры является заполнение заданного количества линий для прохождения на следующий уровень. При этом начисляется количество очков, равное количеству линий оставшихся полностью свободными от блоков, умноженное на 10. При переходе на новый уровень стакан очищается и увеличивается как скорость падения фигур, так и количество линий, которое нужно заполнить для его прохождения.



Видео, позволяющее оценить геймплей игр: <https://www.youtube.com/watch?v=CvUK-YWYcaE>

###### Бонус до 70 баллов за проигрывание фоновой мелодии и звуковых эффектов

Бонус начисляется за проигрывание фоновой мелодии, а также озвучивание следующих событий игры:

* Падение блока
* Уничтожение строки при ее заполнении
* Переход на новый уровень
* Конец игры

1. Разумеется, вы можете использовать любой другой каталог. Тем не менее, важно, чтобы glut.h находился в подкаталоге GL, т.к. общепринятой практикой использования данного заголовочного файла (рекомендуемой в официальной документации библиотеки) является включение файла **<GL/glut.h>**. Следование данному простому правилу облегчит сборку всевозможных проектов, использующих GLUT, которые Вы можете найти в сети Интернет. [↑](#footnote-ref-1)
2. Автор библиотеки GLUT [советует](http://www.cs.csustan.edu/~rsc/SDSU/GLUTinstall.html) поместить glut32.dll в каталог установки Windows, однако запись в данный каталог может быть запрещена для непривилегированного пользователя. Кроме того, Microsoft не рекомендует размещать файлы, не имеющие отношения к операционной системе и ее компонентам, в каталоге Windows. [↑](#footnote-ref-2)
3. Некоторые функции, например, glVertex, напротив, запрещается вызывать вне glBegin/glEnd. Более подробная информация доступна в спецификации OpenGL, а также в MSDN. [↑](#footnote-ref-3)
4. Строго говоря, это зависит от реализации OpenGL для данной оконной системы. Конкретные реализации в целях оптимизации могут, например, просто копировать содержимое теневого буфера на экран (в окно), не изменяя само содержимое теневого буфера. Поэтому никаких предположений насчет содержимого буфера кадра после выполнения данной операции приложение делать не должно и при следующей перерисовке изображения должно выполнить его очистку при помощи glClear. [↑](#footnote-ref-4)
5. **Порт просмотра** (видовой порт, Viewport) – прямоугольная область буфера кадра, ограничивающая область вывода примитивов OpenGL. [↑](#footnote-ref-5)
6. **Видовая система координат** (система координат наблюдателя) – система координат с началом в точке, совпадающей с глазом виртуального наблюдателя. В OpenGL координатная ось X в данной системе координат (ее еще называют u), направлена вправо, ось Y (ее еще называют v) – вверх, а ось Z (другое имя данной оси – n) – в направлении, противоположном направлению взгляда [↑](#footnote-ref-6)
7. **Нормализованная система координат устройства** – система координат, получаемая после выполнения операции переспективного или ортографического преобразования, в которой левому краю видового порта соответствует координатная плоскость X=-1, правому краю – X=1, нижнему краю – Y=-1, верхнему – Y=1, ближней плоскости отсечения – Z=-1, а дальней – Z=+1. При помощи преобразования в порт просмотра координаты вершин примитивов из нормализованной системы координат устройства преобразовываются в оконные координаты. [↑](#footnote-ref-7)
8. В OpenGL есть еще матрица моделирования-вида (GL\_MODELVIEW) и матрица преобразования текстурных координат (GL\_TEXTURE). [↑](#footnote-ref-8)
9. Ортографическое преобразование выполняет преобразование вершин, заданных в **системе координат наблюдателя** (видовой системе координат) в нормализованную систему координат. [↑](#footnote-ref-9)
10. [Контекст рендеринга](http://msdn.microsoft.com/en-us/library/dd369038%28v=VS.85%29.aspx) (контекст визуализации, rendering context) устанавливает связь между оконной системой и OpenGL. Каждый поток, использующий OpenGL должен иметь текущий контекст визуализации. Перед созданием контекста визуализации приложение должно установить формат пикселей в контексте устройства, на котором должна происходить визуализация посредством OpenGL. [↑](#footnote-ref-10)
11. Строго говоря, могли, задав при помощи функции [atexit](http://msdn.microsoft.com/en-us/library/tze57ck3%28v=VS.90%29.aspx) адрес процедуры, которая будет вызвана в процессе вызова exit, и разместив в этой процедуре код освобождения ресурсов. Но данный способ воспринимается автором не более чем как грязный «hack», который не подобает использовать в приложениях на C++. [↑](#footnote-ref-11)
12. Пытливый читатель может по аналогии интегрировать OpenGL в свой любимый оконный framework, либо воспользоваться готовыми решениями [↑](#footnote-ref-12)