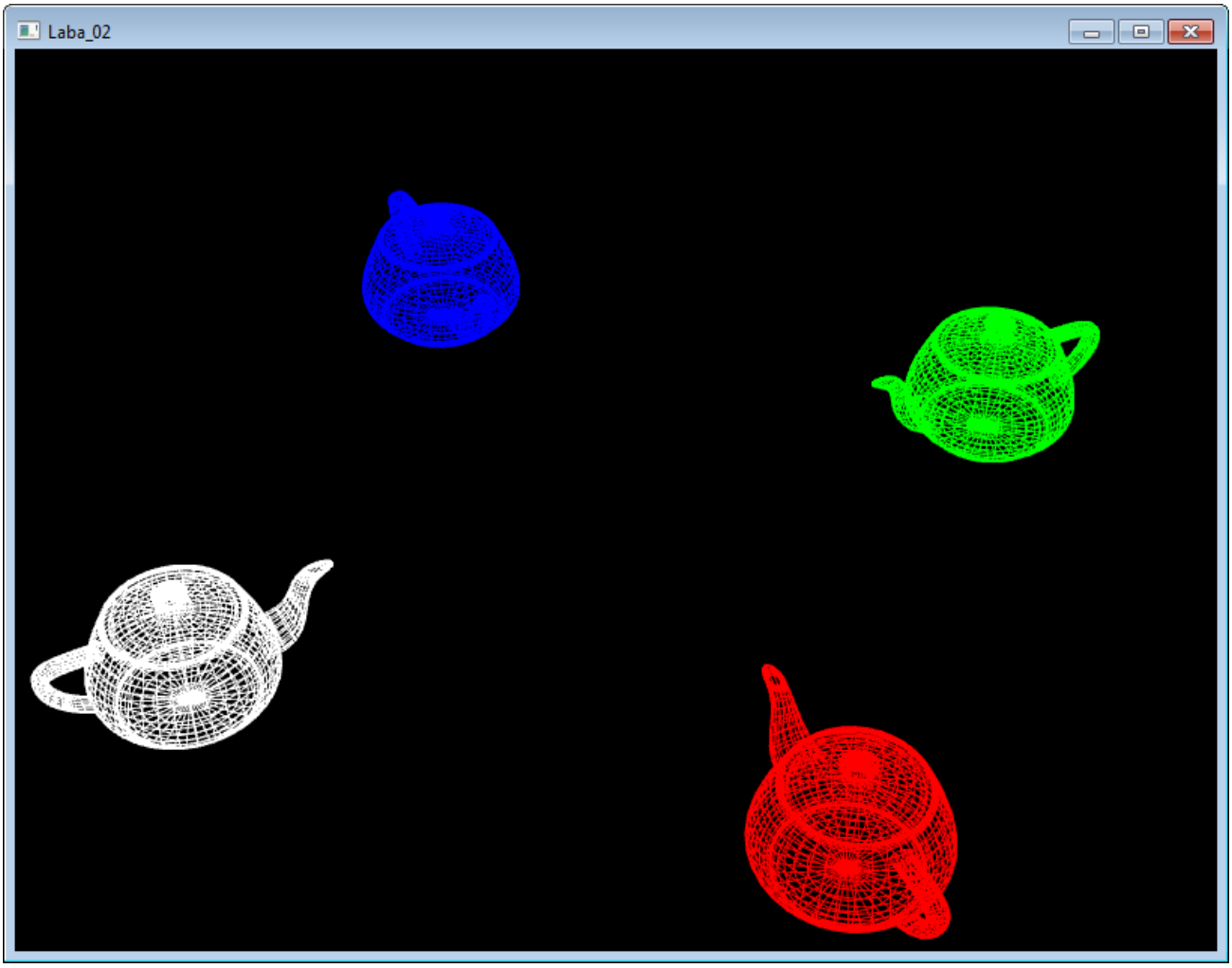
Лабораторная работа №2

Размещение объектов на сцене.

В данной лабораторной работе необходимо вывести на экран несколько трехмерных объектов в разных местах сцены. В качестве объектов по-прежнему выступают чайники, но в этот раз необходимо вывести четыре чайника разных цветов в разных точках сцены.



Целью лабораторной работы является изучение принципов задания позиций трехмерных объектов на сцене, их вывода с использованием матриц моделей, а так же изучение возможностей предоставляемых библиотекой OpenGL для работы с матрицами. Так же в рамках получения практических навыков программирования будет рассмотрен процесс создания класса для представления трехмерного объекта с реализацией необходимых методов и размещение его в отдельном модуле.

Порядок выполнения лабораторной работы.

Лабораторная работа №2 строится на основе предыдущей работы с введением необходимых дополнений. Выполнять лабораторную работу рекомендуется в соответствии с приведенным порядком выполнения. В случае необходимости следует обратиться к дополнительной литературе для изучения принципов объектно-ориентрованного программирования и синтаксиса создания классов и объектов в C++.

В данной лабораторной работе необходимо решить две задачи:

1. Научиться задавать позицию и ориентацию трехмерного объекта с использованием матриц модели, а так же использовать данную матрицу в командах OpenGL для вывода трехмерного объекта на экран в заданном месте сцены.

2. Написать класс для представления одного трехмерного объекта, который включает в себя все необходимые свойства и методы для работы с ним.

При этом, если первая часть (задание позиции трехмерного объекта и его вывод на экран с использование функций OpenGL) была достаточно подробна рассмотрена на лекции, то вторая часть, а именно создание класса для представления трехмерного объекта, для студентов мало знакомых с ООП в C++, может представлять определенные трудности. Однако, тем большую пользу принесет выполнение данной работы.

Для того чтобы успешно выполнить задание и справиться с трудностями, которые могут возникнуть в процессе выполнения лабораторной работы, рекомендуется придерживаться следующего порядка выполнения лабораторной работы:

1. Создать класс GraphicObject для представления трехмерного объекта, а так же переменную этого типа. Необходимо определить тело класса, т.е. все необходимые свойства и методы класса, а так же реализацию этих методов. При написании реализации методов на данном этапе используются «заглушки» - то есть пустые методы, которые не выполняют никаких действий. Следует создать один объект данного типа и в нужных местах программы вызвать соответствующие методы, то есть в функции main вызываются методы для задания параметров трехмерного объекта, а в функции display метод для вывода данного трехмерного объекта на экран. Поскольку все методы пустые, на экран ничего выводиться не будет, но это позволит убедиться, что нет никаких синтаксических ошибок.

2. Оформить класс в виде отдельного модуля, создав заголовочный файл (*.h) и файл реализации (*.cpp). После чего следует подключить вновь созданный модуль к программе и убедиться, что программа по-прежнему компилируется. Следует так же проверить что работает защита от повторного включения модуля, специально явным образом подключив модуль два раза с помощью директивы #include.

3. Написать реализацию всех методов так, чтобы они выполняли все возложенные на них функции. Сложностью данного этапа является то, что для получения изображения на экране, корректно должны работать обе функции – и функция задания параметров трехмерного объекта, и функция вывода на экран. Иными словами, результат можно получить только после написания всех функций. На этом этапе необходимо вспомнить принципы создания матрицы модели, а так же функции OpenGL для работы с матрицами и вывода модели.

4. После того, как реализованный класс для представления трехмерных моделей был проверен, можно переходить к выполнению задания. В частности, необходимо вывести четыре чайника так, как показано на рисунке. Для этого удобнее создать массив объектов, проинициализировать каждый объект независимо друг от друга, а для вывода всех объектов использовать стандартный цикл for. Количество объектов в массиве желательно задавать с помощью константы. Следует так же иметь в виду, что в процессе защиты лабораторной работы количество чайников или их позиции могут быть изменены.

Создание класса для представления трехмерной модели.

В рамках данной лабораторной работы необходимо вывести на экран четыре трехмерных объекта, каждый из которых обладает как минимум двумя характеристиками - позиция объекта и цвет объекта.

Для хранения этих свойств можно было бы завести соответствующие переменные для каждого объекта, т.е. определить в программе массив позиций всех объектов и массив цветов всех объектов. При этом фактически данные переменные будут лежать в разных местах и никак не будут связанны друг с другом, т.е. для вывода определенного объекта необходимо будет получить его цвет из одного массива и его позицию из другого массива. Таким образом, задача согласованного использования переменных ложится на программиста и никак не контролируется компилятором. Легко представить ситуацию, в которой программист по ошибке использует не те данные или забывает проинициализировать какой-либо элемент массива.

Второй проблемой такого подхода является то, что потребуется создать несколько функций, одна из которых инициализирует эти переменные, а вторая использует их для вывода трехмерного объекта на экран. В этом случае необходимо будет передавать в функцию эти данные. Если функция использует сразу несколько переменных, то количество параметров функции может резко возрастать и использование таких функций будет неудобным. Рассмотрим, например, гипотетическую функцию вывода чайника. В этом случае необходимо будет передать в функцию, как позицию чайника, так и его цвет. С учетом того, что каждый из этих параметров является вектором из трех компонент, необходимо передавать в функцию целых шесть параметров.

Третьим недостатком является отсутствие механизмов защит данных, то есть, если используемые данные являются глобальными, то их может изменить любая функция непредсказуемым образом. В случае если разрабатываемая программа является достаточно большой или над ней работают несколько программистов, всегда существует вероятность того, что в какой-либо функции происходит нежелательное изменение глобальных переменных и определить где именно будет затруднительно.

И, наконец, самым существенным недостатком является сложность изменения используемых алгоритмов работы. Например, если требуется изменить способ вывода объектов на экран, добавив к позиции объекта еще и угол поворота объекта относительно оси OY, придётся менять не только структуру данных для хранения информации об объекте, но и все функции которые используют эти структуры данных.

Для устранения всех этих недостатков как раз и использует объектно-ориентированный подход. В этом случае для представления какой-либо сущности, например, рассматриваемого трехмерного объекта, создают класс, который содержит в себе все необходимые данные для представления этой сущности, а так же все методы для работы с ней.

Классы в ООП — это абстракция, описывающая свойства и методы некоторой абстрактной, ещё не существующей сущности. Данные свойства и методы присущи всем будущим экземпляром класса. Например, в рамках данной лабораторной работы можно считать, что все трехмерные объекты обладают такими свойствами как позиция и цвет, а так же такими методами как вывод объекта, задание его цвета и позиции.

Объекты — это конкретное представление абстракции, имеющее свои собственные значения указанных свойств и методы, которые работают именно со свойствами данного конкретного объекта. Объекты, созданные на основе какого-либо класса, называются экземплярами данного класса.

Инкапсуляция - один из трех базовых принципов ООП который заключается в объединение в классе данных и методов для работы с этими данными. Инкапсуляция подразумевает так же сокрытие деталей реализации этого класса, то есть часть свойств или методов может быть объявлена скрытой (private) и таким образом быть невидимой для кода, лежащего вне самого класса.

Одной из важнейших задач разработчика программного обеспечения является определение необходимого состава классов, описание интерфейса каждого класса (то есть методов, предоставляемых классом для работы с ним), а так же организация взаимодействия различных классов друг с другом!

Для того чтобы правильно определить все необходимые методы и свойства, необходим определенный опыт в разработке программ и знание предметной области. На данном этапе необходимо использовать класс со структурой приведенной ниже:

```
// КЛАСС ДЛЯ ПРЕДСТАВЛЕНИЯ ОДНОГО ГРАФИЧЕСКОГО ОБЪЕКТА
class GraphicObject
{
private:
	// Позиция и угол поворота для объекта
	Glfloat position[3];
	Glfloat angle;
	// Матрица модели (расположение объекта) - чтоб не вычислять каждый раз
	Glfloat modelMatrix[16];
	// Цвет модели
	Glfloat color[3];
public:
	// Конструктор
	GraphicObject (void);
	// Задать позицию объекта
	void   setPosition (float x, float y, float z);
	// Задать угол поворота в градусах относительно оси OY
	void   setAngle (float a);
	// Задать цвет модели
	void   setColor (float r, float g, float b);
	// Вывести объект
	void   draw (void);
};
```

Назначение всех полей и методов является очевидным по их названию и приведённым комментариям. Тем не менее, рассмотрим элементы этого класса чуть-чуть подробнее.

Прежде всего, следует обратить внимание на то, что все поля класса объявлены приватными (private). Это приводит к тому, что доступ к ним возможен только из самого класса. При этом программа, использующая данный класс, не может получить доступ к этим данным напрямую, но может использовать их через набор определенных интерфейсных методов (например, метод setColor, который используется для установки внутреннего поля, отвечающего за цвет объекта). Эта возможность является весьма ценной, поскольку позволяет с одной стороны защитить внутренние данные от несанкционированного доступа, а с другой стороны гарантирует, что другие участки кода не используют эти переменные, что, в свою очередь, позволит в дальнейшем изменять внутреннюю структуру класса, не внося изменения в другие участки кода.

Следом идут методы, объявленные публичными, поэтому данные методы могут использоваться из любой точки программы. Совокупность публичных методов составляет интерфейс класса. Интерфейс класса определяет все действия, которые можно производить над объектом. В общем случае единожды определенный интерфейс не должен изменяться, поскольку остальные участки кода могут рассчитывать на соответствующие методы. При изменении интерфейса класса существует вероятность того, что придется вносить изменения в другие модули.

Еще одним важным преимуществом использования классов является наличие конструктора. Конструктор – это специальный метод, чье имя совпадает с именем класса. Конструктор вызывается автоматически самим компилятором в момент создания объекта и может использоваться для инициализации внутренних переменных объекта. Таким образом, мы можем гарантировать, что перед первым использованием данного объекта все его внутренние переменные будут проинициализированы должным образом. Следует учесть, что реализацию конструктора программист должен писать самостоятельно. Конструкторы могут иметь параметры и, соответственно, может быть несколько различных конструкторов. В данном случае используется простейший конструктор без параметров.

Далее идут обычные методы, в частности:

1. Метод позволяющий задать позицию объекта. Метод просто копирует свои параметры в соответствующие внутренние переменные и вычисляет матрицу модели:

```
// Задать позицию объекта
void setPosition (float x, float y, float z);
```

2. Метод позволяющий задать угол поворота в градусах относительно оси OY. Метод так же копирует параметр во внутреннее поле и вычисляет матрицу модели. Обратите внимание, что для вычисления матрицы модели используются как текущая позиция, так и угол поворота:

```
// Задать угол поворота в градусах относительно оси OY
void setAngle (float a);
```

3. Метод позволяющий задать цвет объекта. Здесь так же следует запомнить передаваемый цвет и использовать его при выводе объекта на экран:

```
// Задать цвет модели
void setColor (float r, float g, float b);
```

4. И, наконец, метод выводящий трехмерный объект на экран с учетом ранее установленных позиций, цвета и угла поворота. Следует обратить внимание, что сам метод не содержит никаких параметров, а осуществляет необходимую функциональность, используя внутренние поля самого объекта, к которому применяется данный метод.

```
// Вывести объект
void draw (void);
```

В теле класса указывается только заголовок методов, сами методы определяются вне класса, как правило в отдельном файле, о чем будет сказанно ниже. После того, как было написанно тело класса, необходимо написать реализацию каждого метода, чтобы класс можно было использовать, а программу скомпилировать. На первом этапе для каждого метода можно использовать «заглушку» – то есть тело метода в котором не выполняются никакие инструкции.

После того, как класс определен (даже если в качестве методов пока используются заглушки) можно создать объекты данного класса и даже вызывать для них все методы. Уже на данном этапе работу с объектом можно построить так, будто бы класс целиком написан, то есть в самом начале программы можно вызвать методы для установки позиции объекта и его цвета, а в функции Display вызвать метод для вывода объекта на экран. Приведем пример объявления глобальных переменных для представления всех трехмерных объектов программы:

```
// МАССИВ ОБЪЕКТОВ ДЛЯ ВЫВОДА
const int           graphicObjectCount = 4;
CGraphicObject      graphicObjects[graphicObjectCount];
```

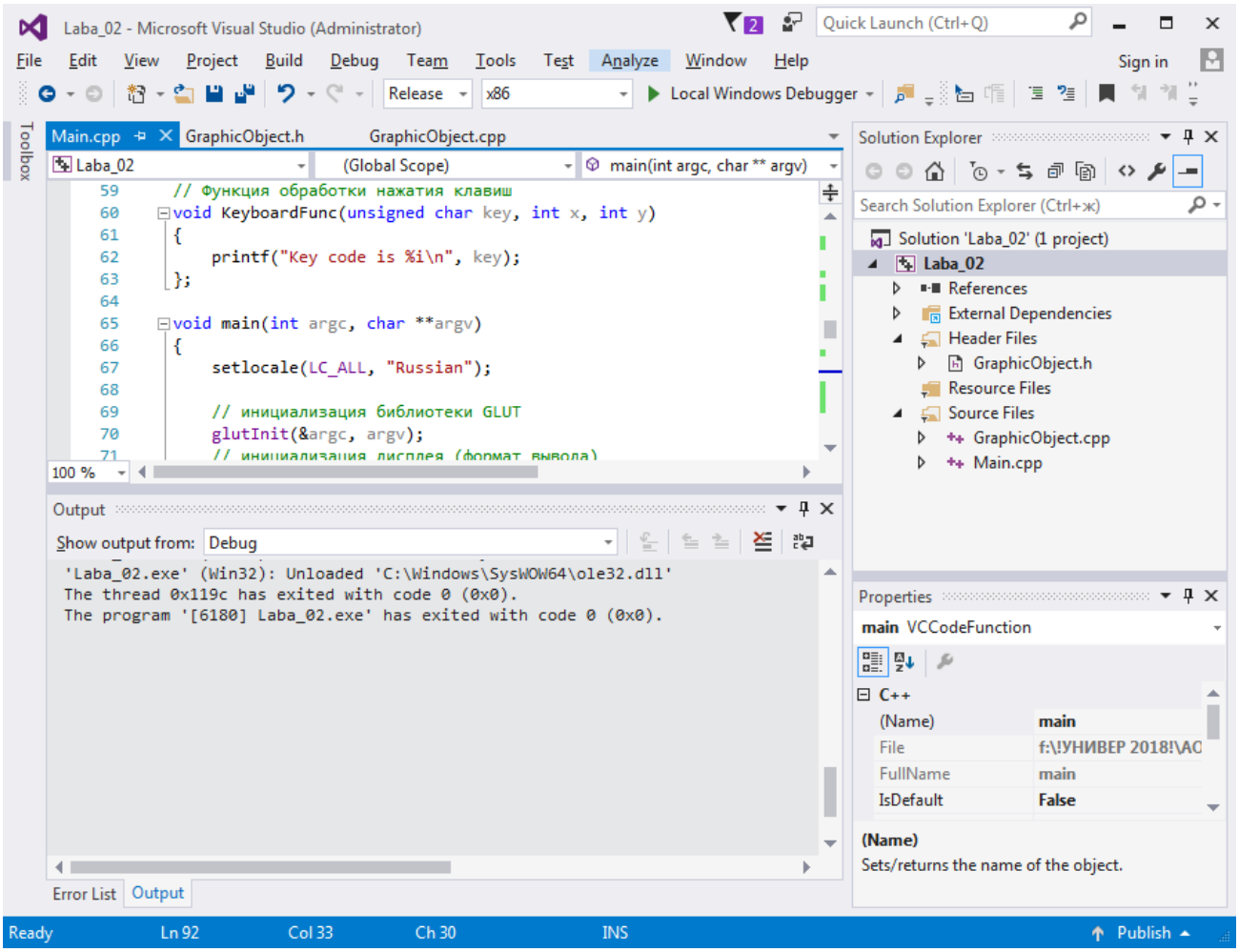
При этом функция Display для вывода объектов на экран будет выглядеть так, как показано ниже. Обратите внимание, что в этом случае, функция выглядит весьма лаконично, скрывая сложности вывода объекта на экран за одним методом draw:

```
 // функция вызывается при перерисовке окна
// в том числе и принудительно, по командам glutPostRedisplay
void Display (void)
{
	// отчищаем буфер цвета
	glClearColor(0.0, 0.0, 0.0, 1.0);
	glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

	// включаем тест глубины
	glEnable(GL_DEPTH_TEST);

	// устанавливаем камеру
	glMatrixMode(GL_MODELVIEW);
	glLoadIdentity();
	gluLookAt(15, 10, 20, 0, 0, 0, 0, 1, 0);

	// выводим объекты
	for (int i = 0; i < graphicObjectCount; i++) {
		graphicObjects[i].Draw();
	};

	// смена переднего и заднего буферов
	glutSwapBuffers();
};
```

Создание модуля для представления класса.

После того, как класс для работы с объектом написан, можно заметить, что он стал занимать достаточно много места в главном файле (в файле в котором определена функция main и другие глобальные функции). Навигация по такому файлу становится затруднительной, поэтому рекомендуется вынести определения класса и реализацию всех его методов в отдельный модуль. Еще одним стимулом выделения класса в отдельный модуль является то, что, возможно, в будущем потребуется использовать этот класс в других программах или других модулях.

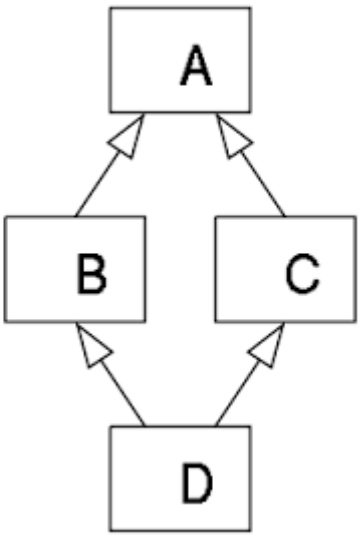
Обычно, каждый модуль состоит из двух файлов - заголовочного файла (*.h) и файла реализации (*.cpp). В первом содержится описание всех переменных, констант и типов данных (включая прототип классы), а во втором непосредственно реализация описанных функций или методов класса. Поэтому первым делом необходимо создать два файла (GraphicObject.h и GraphicObject.cpp) и подключить их к проекту:



После создания заголовочного файла (*.h) необходимо перенести в него определение класса, то есть тело класса без реализации методов, и подключить соответствующий заголовочный файл с помощью директивы #include к тому файлу, в котором его планируется использовать. В данном случае в файле main.cpp необходимо указать, что будет использовать модуль для работы с трехмерными объектами:

```
#include "GraphicObject.h"
```

Таким образом, если к какому либо файлу с помощью директивы #include подключен заголовочный файл этого модуля, то в этом файле видны все переменные, константы, типы данных и классы, определенные в подключаемом заголовочном файле. При этом возможна ситуация, когда модуль подключается к файлу несколько раз, например, в результате ромбовидного подключения модулей.



К примеру, если к файлу D, подключены два модуля – B и C, каждый из которых использует модуль A, то получается, что модуль A оказался косвенно подключенным к файлу D два раза, а это значит, что все переменные, типы данных и функции, объявленные в модуле A, оказались объявлены по два раза, что является ошибкой.

В частности, если к нашему проекту подключить заголовочный файл GraphicObject.h два раза, компилятор выдаст следующее сообщение об ошибке, которое как раз и говорит о том, что класс GraphicObject был переопределен (определен повторно):

Severity	Code	Description	Project	File	Line	Suppression State
Error	C2011	'GraphicObject': 'class' type redefinition	Laba_02	f:\!универ 2018!\аокг\labs\laba2\GraphicObject.h	11	

Чтобы избежать этой проблемы используют так называемую защиту от повторного включения заголовка, которая может быть реализована двумя способами. Первый способ - это использование директив условной компиляции:

```
#ifndef GRAPHICOBJECT_H
#define GRAPHICOBJECT_H

class GraphicObject
{
};

#endif
```

Второй способ - использование директивы компилятора #pragma once. Данный способ несколько более лаконичен, но может не работать с компиляторами, отличными от Visual Studio:

```
#pragma once
```

Таким образом, заголовочный файл модуля для работы с трехмерными объектами (GraphicObject.h) принимает следующий вид:

```
#pragma once

#include <windows.h>
#include "GL/freeglut.h"
#include "math.h"

//     КЛАСС ДЛЯ ПРЕДСТАВЛЕНИЯ ОДНОГО ГРАФИЧЕСКОГО ОБЪЕКТА
class GraphicObject
{
private:
	// Позиция и угол поворота для объекта
	Glfloat position[3];
	Glfloat angle;
	// Матрица модели (расположение объекта) - чтоб не вычислять каждый раз
	Glfloat modelMatrix[16];
	// Цвет модели
	Glfloat color[3];
public:
	// Конструктор
	GraphicObject (void);
	// Задать позицию объекта
	void   setPosition (float x, float y, float z);
	// Задать угол поворота в градусах относительно оси OY
	void   setAngle (float a);
	// Задать цвет модели
	void   setColor (float r, float g, float b);
	// Вывести объект
	void   draw (void);
};
```

Далее необходимо перенести реализацию всех методов в файл реализации (GraphicObject.cpp). Следует обратить внимание, что поскольку описание самого класса располагается в другом файле, то к файлу реализации надо подключить заголовочный файл этого же модуля. Следовательно, файл реализации выглядит следующим образом:

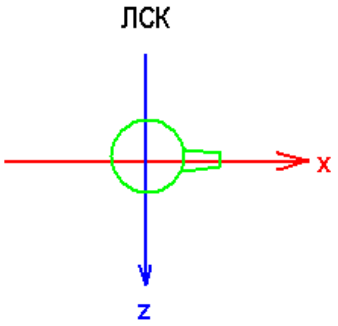
```
#include "GraphicObject.h"

// Конструктор
GraphicObject::GraphicObject(void)
{
}

// задать позицию объекта
void   GraphicObject::setPosition (float x,float y,float z)
{
}

// задать угол поворота в градусах относительно оси OY
void   GraphicObject::setAngle (float a)
{
}

// Задать цвет модели
void   GraphicObject::setColor (float r, float g, float b)
{
}

// вывести объект
void   GraphicObject::draw (void)
{
}
```

Задание позиции объекта.

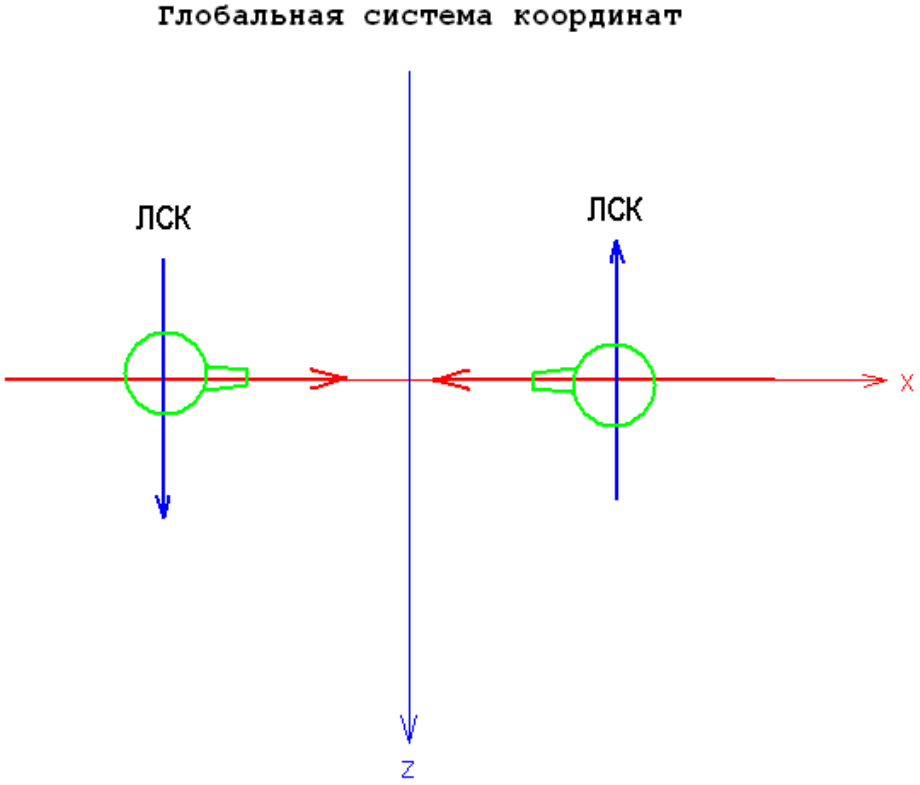
В данной лабораторной работе необходимо вывести несколько объектов в разных местах сцены. В качестве моделей для объектов выступает один и тот же чайник, который описывается в своей собственной локальной системе координат. Для того чтобы вывести чайник в разных местах, необходимо задать матрицу модели, которая переводит локальную систему координат в глобальную систему координат. Данный раздел посвящен процессу построения такой матрицы модели и передачи её в OpenGL.

Рассмотрим следующий пример. Необходимо разместить на сцене два объекта (чайника) в точках (4,0,0) и (-4,0,0) так, чтобы их носики были направлены в центр сцены.

Для начала разберемся, как же именно строятся чайники в локальной системе координат. Freeglut создает чайники так, что они находятся в центре локальной системы координат, а их носики располагаются вдоль вектора Ox локальной системы координат. Пример такого чайника в локальной системе координат (вид сверху) приведен правее. Вектор OY направлен вверх.



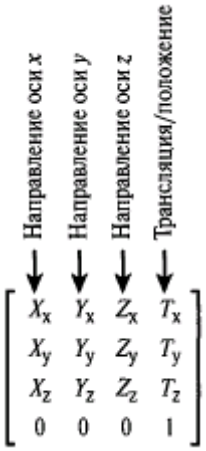
Итак, требуется создать вышеописанную сцену с двумя чайниками. Для этого необходимо разместить в глобальной системе координат две локальные системы координат и выразить их оси и центр, через координаты глобальной системы координат.



Отдельно следует отметить, что при размещении объектов на сцене, то есть при указании локальной системы координат в глобальной, следует придерживаться трех основных правил:

1. Оси локальной системы координат должны иметь единичную длину. Если они имеют не единичную длину, то это приводит к растяжению или сжатию по соответствующей оси.
2. Оси локальной системы координат должны быть взаимно перпендикулярны, иначе будут перекосы объекта.
3. Оси по-прежнему должны составлять правую тройку векторов. Например, для правого чайника (чтоб его носик смотрел влево) мы не можем просто изменить знак оси Ox, поскольку это приведет к тому, что мы получим левую тройку векторов. Поэтому мы так же должны изменить знак оси Oz.

После того, оси локальной системы координат и её центр были выражены через координаты глобальной системы координат, можно задать матрицу модели, используя следующий шаблон:



Следует помнить, что OpenGL использует матрицы развернутые по столбцам, поэтому полученные матрицы модели кодируются следующим образом:

```
//    Матрицам модели правого чайника (в точке - (4,0,0))
GLfloat M0[16] = {-1,0,0,0,        0,1,0,0,         0,0,-1,0,         4,0,0,1};
//    Матрицам модели левого чайника (в точке - (-4,0,0))
GLfloat M1[16] = { 1,0,0,0,        0,1,0,0,         0,0, 1,0,        -4,0,0,1};
```

После того, как были сформированы матрицы модели, их можно использовать для вывода трехмерных объектов на экран. Для этого необходимо использовать стандартный алгоритм вывода:

1. Перед началом вывода модели необходимо установи матрицу наблюдения, то есть задать матрицу, которая описывает, в какой точке находится наблюдатель. Матрица наблюдения устанавливается один раз для всех моделей и не принадлежит классу GraphicObject, поэтому данная операция выполняется вне метода draw, применяемого к конкретной модели. Матрица модели устанавливается с помощью следующей последовательности кода в функции Display:

```
// устанавливаем камеру
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
gluLookAt(15, 15, 7.5, 0, 0, 0, 0, 1, 0);
```

   Процесс установки матрицы наблюдения будет рассмотрен в следующей лабораторной работе. В данном случае наблюдатель располагается в точке (15, 15, 7.5) и смотрит в центр сцены –в точку (0, 0, 0).

2. Далее для каждого объекта вызывается метод draw, который занимается непосредственно выводом данного объекта на экран. В частности, для вывода объекта в заданном месте необходимо выполнить следующую последовательность действий:

   2.1 Сохранить текущую матрицу наблюдения модели (GL_MODELVIEW) в стеке. В этот момент в вершине стека находится ранее установленная матрица наблюдения, которая понадобится для вывода последующих объектов, поэтому, чтобы её не портить, она сохраняется в стеке;

   2.2 Домножить матрицу модели, на текущую матрицу наблюдения модели (GL_MODELVIEW). В данный момент в вершине стека находилась ранее загруженная матрица наблюдения, поэтому в результате умножения мы получим матрицу, которая как раз переведет координаты модели из локальной системы координат сразу в систему координат наблюдателя;

   2.3 Вывести модель на экран, использую всю ту же функцию glutWireTeapot из первой лабораторной работы. Перед выводом модели необходимо также установить прочие параметры, такие, как, например, цвет объекта;

   2.4 Восстановить ранее сохраненную в стеке матрицу наблюдения модели (GL_MODELVIEW). В результате текущей матрицей наблюдения модели снова становится ранее выбранная матрица наблюдения.

Следует отметить, что все прочие шаги, такие как очистка буфера кадра и смена переднего и заднего буфера цвета остаются неизменными.

Задание к лабораторной работе.

Лабораторная работа №2 строится на основе предыдущей работы с внесением следующих изменений:

1. В отдельном модуле создать класс GraphicObject вышеописанной структуры для работы с трехмерным объектом.

2. Реализовать вывод ***четырех*** чайников разных цветов расположенных на осях OX и OZ с носиками, повернутыми в центр сцены, как изображено выше. Для хранения и вывода трехмерных объектов использовать ранее созданный класс GraphicObject.

3. Вывести все объекты, реализуя классический цикл программы с анимацией:

   1. Очистить буфер экрана (буферы цвета, глубины и т.д.)
   2. Установить матрицу камеры
   3. Для каждого объекта (в методе draw):
      a. сохранит матрицу наблюдения в стеке
      b. умножить матрицу наблюдения на матрицу модели
      c. Вывести модель (установив так же её цвет)
      d. Восстановить матрицу наблюдения из стека
   4. Поменять местами передний и задний буферы
   5. Перейти к шагу 1

Содержание отчета.

1. Титульный лист.
2. Задание к лабораторной работе.
3. Текст заголовочного файла модуля с классом GraphicObject.
4. Текст файла реализации модуля с классом GraphicObject.
5. Текст основной программы с комментариями.
6. Скриншот работы программы.

Критерии оценки и вопросы к защите.

Вопросы по теоретической части:

1.1 Что такое модель, объект, сцена?
1.2 Что такое локальная система координат (система координат модели)?
1.3 Что такое глобальная система координат (мировая система координат)?
1.4 Что такое система координат наблюдателя (видовая система координат)?
1.5 Для чего применяется матрица модели, матрица наблюдения и матрица проекции?
1.6 Напишите последовательность матричных операций по преобразованию вершины модели для вывода её на экран?

Вопросы по практической части:

2.1 Какие две матрицы применяются в OpenGL для преобразования вершин модели?
2.2 Для чего нужна команда glMatrixMode?
2.3 Как работает матричный стек в OpenGL?
2.4 Приведите алгоритм вывода одного объекта в заданном месте?

Требования к программе:

3.1 Реализация класса GraphicObject в отдельном модуле;
3.2 К каждому методу класса должен быть комментарий;
3.3 Глобальные переменные должны носить осмысленные имена;