МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ
ВОЛЖСКИЙ ПОЛИТЕХНИЧЕСКИЙ ИНСТИТУТ (ФИЛИАЛ)
ФЕДЕРАЛЬНОГО ГОСУДАРСТВЕННОГО БЮДЖЕТНОГО ОБРАЗОВАТЕЛЬНОГО
УЧРЕЖДЕНИЯ ВЫСШЕГО ОБРАЗОВАНИЯ
«ВОЛГОГРАДСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ»

Абрамова О.Ф.

КОМПЬЮТЕРНАЯ ГРАФИКА:
лабораторный практикум
Часть 2

Электронное учебно-методическое пособие

Волжский
2023

УДК 004.92(07)
ББК 32я73
А 161

Рецензенты:
зав. кафедрой методики преподавания математики и физики, ИКТ
ФГБОУ ВО «Волгоградский государственный социально-педагогический университет», д. п. н.
Смыковская Т.К.
доцент кафедры физики, математики и информатики
ФГБОУ ВО «Волгоградский государственный медицинский университет Минздрава РФ», к.п.н.
Филиппова Е.М.

Издается по решению редакционно-издательского совета
Волгоградского государственного технического университета

Абрамова, О. Ф.
Компьютерная графика: лабораторный практикум. Часть 2 [Электронный ресурс] : учебно-методическое пособие / О.Ф.Абрамова ; Министерство науки и высшего образования Российской Федерации, ВПИ (филиал) ФГБОУ ВО ВолгГТУ. – Электрон. текстовые дан. (1 файл: 776 КБ). – Волжский, 2023. – Режим доступа: http://lib.volpi.ru. – Загл. с титул. экрана.
ISBN 978-5-9948-4548-6

В учебном пособии собран теоретический материал о графической библиотеке Opengl и практические задания для лабораторных занятий по дисциплине «Компьютерная графика». Задача учебного пособия – помочь в практическом изучении основных тем дисциплины и получении навыков программирования с помощью графической библиотеки OpenGL студентам направлений 09.03.01 «Информатика и вычислительная техника» и 09.03.04 «Программная инженерия».

Илл. 6, табл. 1, библиограф.: 10 назв.

ISBN 978-5-9948-4548-6

ВВЕДЕНИЕ

Дисциплина «Компьютерная графика» читается студентам третьего курса, обучающимся по направлениям 09.03.01 «Информатика и вычислительная техника» и 09.03.04 «Программная инженерия». Целью изучения этой дисциплины является формирование у студентов, обучающихся по направлению 09.03.01 «Информатика и вычислительная техника», следующих компетенций:

1. ОПК-2 – способность использовать современные информационные технологии и программные средства, в том числе отечественного производства, при решении задач профессиональной деятельности.
2. ОПК-8 – способность разрабатывать алгоритмы и программы, пригодные для практического применения.
3. ОПК-9 – способность осваивать методики использования программных средств для решения практических задач.

Целью изучения дисциплины «Компьютерная графика» для студентов, обучающихся по направлению 09.03.04 «Программная инженерия», является формирование следующих компетенций:

1. ОПК-2 – способность понимать принципы работы современных информационных технологий и программных средств, в том числе отечественного производства, и использовать их при решении задач профессиональной деятельности.
2. ОПК-6 – способность разрабатывать алгоритмы и программы, пригодные для практического использования, применять основы информатики и программирования к проектированию, конструированию и тестированию программных продуктов.

Практикум содержит исчерпывающую теоретическую и практическую информацию к двум очередным лабораторным работам (начало в учебно-методическом пособии «Компьютерная графика: лабораторный практикум.

Часть 1»), раскрывающим основные возможности программирования компьютерной графики с использованием кроссплатформенной библиотеки OpenGL.

Данное пособие предназначено для студентов, изучающих программирование, для которых знание возможностей современных графических библиотек является необходимым компонентом профессиональных знаний.

В настоящее время наиболее известными и широко распространенными являются графические библиотеки DirectX и OpenGL. Первая библиотека является внутренним стандартом фирмы Microsoft и сильно привязана к операционной системе Windows. Библиотека OpenGL является открытым международным стандартом и не зависит от программного обеспечения, установленного на ПК.

Для выполнения лабораторных работ практикума студенты должны иметь навыки программирования на языке C++ (как минимум, хотя особых требований к языку программирования не предъявляется), а также первоначальные знания об операционной системе Windows и особенностях программирования под Windows. Хотя библиотека OpenGL является кроссплатформенной, т.е. не зависит от операционной системы, в работах предусматривается использование возможности именно операционной системы Windows, поскольку в данный момент она является наиболее распространенной операционной системой.

Каждая лабораторная работа снабжена дополнительным материалом по рассматриваемой теме, набором практических примеров реализации сопутствующих теме лабораторной работы задач.

Каждая лабораторная работа содержит набор заданий, которые выполняются в соответствии с вариантом, требования к оформлению отчета и работоспособный пример программной реализации. При защите лабораторной работы студенты должны предоставить работоспособные програм-

мы по каждому заданию данной лабораторной работы, отчет, оформленный в соответствии с требованиями, и ответы на контрольные вопросы.

ЛАБОРАТОРНАЯ РАБОТА №3

OpenGL: Разработка приложения для визуализации связного набора двумерных примитивов с использованием массива вершин

Цель работы

Изучение команд OpenGL для рисования графических примитивов, изучение функций OpenGL для работы с массивами вершин, цветов и нормалей.

Теоретическая информация

Примитивы OpenGL

Установка способа отрисовки

Точки, линии, треугольники, четырехугольники, многоугольники – простые объекты, из которых состоят любые сложные фигуры.

Для отрисовки любого примитива используется базовая конструкция, состоящая из командных скобок glBegin – glEnd с указанием способа соединения вершин, которые перечисляются внутри этих скобок. Т.е. примитивы создаются следующим образом:

```
glBegin (GLenum mode);      // указываем, что будем рисовать
 glVertex[2 3 4][s i f d](...); // первая вершина
                                // тут остальные вершины
 glVertex[2 3 4][s i f d](...); // последняя вершина
glEnd();                       // закончили рисовать примитив
```

Возможные значения *mode* перечислены ниже в таблице 1.1.

Задать вершину можно одним из четырех способов (рассматриваем двумерную сцену):

glVertex2d(x,y); // две переменных типа double

glVertex3d(x,y,z); // три переменных типа double

glVertex2dv(array); // массив из двух переменных типа double

glVertex3dv(array); // массив из трех переменных типа double

Таблица 1.1

Значение mode	Описание
GL_POINTS	Каждый вызов glVertex задает отдельную точку
GL_LINES	Каждая пара вершин задает отрезок
GL_LINE_STRIP	Рисуется ломаная
GL_LINE_LOOP	Рисуется ломаная, причем ее последняя точка соединяется с первой
GL_TRIANGLES	Каждые три вызова glVertex задают треугольник
GL_TRIANGLE_STRIP	Рисуются треугольники с общей стороной
GL_TRIANGLE_FAN	То же самое, но по другому правилу соединяются вершины
GL_QUADS	Каждые четыре вызова glVertex задают четырехугольник
GL_QUAD_STRIP	Четырехугольники с общей стороной
GL_POLYGON	Многоугольник

Лицевые и обратные грани примитива

Любой многоугольник в OpenGL считается составным примитивом, состоящим из двух граней. Под *гранью* понимается одна из сторон много-

угольника. Каждый многоугольник имеет 2 грани: лицевую и обратную. Лицевой считается та сторона, вершины которой обходятся против часовой стрелки. Такое правило установлено по умолчанию. Однако, его можно изменить, установив направление обхода вершин лицевых граней вызовом команды

void glFrontFace (GLenum *mode*)

со значением параметра *mode* равным **GL_CW** (clockwise), а вернуть значение по умолчанию можно, указав **GL_CCW** (counter-clockwise).

Точки

Вы можете нарисовать столько точек, сколько вам нужно. Вызывая glVertex2d, вы устанавливаете новую точку. При этом можно указывать некоторые атрибуты для примитива, такие как размер, цвет и т.п.

Размер точки можно устанавливать с помощью функции:

void glPointSize(GLfloat size);

size – размер объекта в пикселях.

Функция вызывается вне glBegin/glEnd, иначе она будет проигнорирована.

Пример:

```
glPointSize(7);
 glBegin(GL_POINTS);
  glColor3d(1,0,0);
  glVertex3d(-4.5,4,0); // первая точка
  glColor3d(0,1,0);
  glVertex3d(-4,4,0);   // вторая точка
  glColor3d(0,0,1);     // третья
  glVertex3d(-3.5,4,0);
 glEnd();
```

в этом примере будут отрисованы 3 квадратные точки.

При увеличении размера точки будут выглядеть как большие закрашенные квадраты. Но иногда необходимо отрисовывать точки в привычном нам формате, т.е. круглые. И это можно легко сделать, включив специальный режим сглаживания.

Режим сглаживания можно устанавливать вызовом функции

glEnable(GL_POINT_SMOOTH);

Отключение режима выполняется вызовом

glDisable();

с этим же параметром.

Функции так же надо вызывать вне glBegin/glEnd, иначе они будут проигнорированы.

Пример:

```
 glPointSize(7);
 glEnable(GL_POINT_SMOOTH);
  glBegin(GL_POINTS);
   glColor3d(1,0,0);
   glVertex3d(-4.5,4,0); // первая точка
   glColor3d(0,1,0);
   glVertex3d(-4,4,0);   // вторая точка
   glColor3d(0,0,1);     // третья
   glVertex3d(-3.5,4,0);
  glEnd();
glEnable(GL_POINT_SMOOTH);
```

В этом примере будут отрисованы три круглые точки.

Совет: этот прием – увеличить размер точки и включить сглаживание – можно использовать для отрисовки в сцене кругов различных диаметров, потому как отдельного примитива для рисования окружности в OpenGL не предусмотрено.

Линии

Для линий также можно изменять ширину, цвет, размер, устанавливать сглаживание. Если вы зададите разные цвета для начала и конца линии, то ее цвет будет переливающимся, OpenGL по умолчанию делает интерполяцию.

Так же Вы можете рисовать прерывистые линии, делается это путем наложения маски при помощи следующей функции:

void glLineStipple(GLint factor, GLushort pattern);

Параметр *pattern* задает саму маску. Например, если его значение равно 255(0x00FF), то, чтобы вычислить задаваемую маску, воспользуемся калькулятором. В двоичном виде это число выглядит так: 0000000011111111, т.е. всего 16 бит. Старшие восемь установлены в ноль, значит, тут линии не будет. Младшие установлены в единицу, тут будет рисоваться линия.

Параметр *factor* определяет, сколько раз повторяется каждый бит. Например, если его установить равным 2, то накладываемая маска будет выглядеть так:

0000000000000000111111111111111

Перед использованием функции glLineStipple необходимо включить режим, разрешающий рисовать прерывистую линию

glEnable(GL_LINE_STIPPLE);

а после отрисовки этот режим надо выключить

glDisable(GL_LINE_STIPPLE);

Пример:

```
glLineWidth(1);       // ширину линии  устанавливаем 1
 glBegin(GL_LINES);
   glColor3d(1,0,0);     // красный цвет
  glVertex3d(-4.5,3,0); // первая линия
  glVertex3d(-3,3,0);
  glColor3d(0,1,0);     // зеленый
  glVertex3d(-3,3.3,0); // вторая линия
```

```
        glVertex3d(-4,3.4,0);
       glEnd();
        glLineWidth(3);      // ширина 3
       glBegin(GL_LINE_STRIP); // см. ниже
        glColor3d(1,0,0);
        glVertex3d(-2.7,3,0);
        glVertex3d(-1,3,0);
        glColor3d(0,1,0);
        glVertex3d(-1.5,3.3,0);
        glColor3d(0,0,1);
        glVertex3d(-1,3.5,0);
       glEnd();
        glLineWidth(5);
       glEnable(GL_LINE_SMOOTH);
       glEnable(GL_LINE_STIPPLE); // разрешаем рисовать  преры-
вистую линию
       glLineStipple(2,58360);    // устанавливаем маску
       glBegin(GL_LINE_LOOP);
        glColor3d(1,0,0);
        glVertex3d(1,3,0);
        glVertex3d(4,3,0);
        glColor3d(0,1,0);
        glVertex3d(3,2.7,0);
        glColor3d(0,0,1);
        glVertex3d(2.5,3.7,0);
         glEnd();
       glDisable(GL_LINE_SMOOTH);
       glDisable(GL_LINE_STIPPLE);
```

Треугольники

Для треугольника можно задавать те же параметры, что и для линии. Так же можно устанавливать режим растеризации треугольника (да и любого многоугольника) с помощью функции

void glPolygonMode(GLenum *face*,GLenum *mode*);

Первый параметр *face* устанавливает выбор грани для применения режима отрисовки и может принимать значения – GL_FRONT, GL_BACK и GL_FRONT_AND_BACK.

Второй параметр *mode* указывает, как будет рисоваться многоугольник. Он принимает значения:

- GL_POINT – рисуются только точки в вершинах многоугольника,
- GL_LINE – рисуется каркас примитива линиями,

- GL_FILL – рисуем заполненный многоугольник.

Треугольники также можно рисовать, передав **GL_TRIANGLE_STRIP** или **GL_TRIANGLE_FAN** в glBegin. В первом случае, первая, вторая и третья вершины задают первый треугольник. Вторая, третья и четвертая вершины – второй треугольник. Третья, четвертая и пятая вершина – третий треугольник и т.д. То есть вершины n, n+1 и n+2 определят n-й треугольник. Во втором случае, первая, вторая и третья вершины задают первый треугольник. Первая, третья и четвертая вершины задают второй треугольник и т.д. Вершины 1, n+1, n+2 определяют n-й треугольник. Второй способ для отрисовки кругов (см. пример).

Пример:

```
glPolygonMode(GL_FRONT_AND_BACK, GL_FILL);
 glBegin(GL_TRIANGLES);
  glColor3d(1,0,0);      // рисуем треугольник
  glVertex3d(-4,2,0);
  glVertex3d(-3,2.9,0);
  glVertex3d(-2,2,0);
 glEnd();
glLineWidth(2);
 glPolygonMode(GL_FRONT_AND_BACK, GL_LINE); //рисуем  каркасные треугольники
glBegin(GL_TRIANGLE_STRIP); // обратите внимание на порядок вершин
  glColor3d(0,1,0);
  glVertex3d(1,2,0);
  glVertex3d(0,2.9,0);
  glVertex3d(-1,2,0);
  glVertex3d(0,1.1,0);
 glEnd();

//рисуем заполненные треугольники - круг)
glPolygonMode(GL_FRONT_AND_BACK, GL_FILL);
glBegin(GL_TRIANGLE_FAN);
glVertex3d(x0, y0, 0);
for (int i = 0; i <= 360; i += 10)
{
double x = r * cos(i * M_PI / 180);
double y = r * sin(i * M_PI / 180);
glVertex2d(x0 + x, y0 + y);
}
glEnd();
```

Многоугольники

Если в примитиве четыре вершины, то можно воспользоваться вызовом функции glBegin с параметром **GL_QUADS** или **GL_QUAD_STRIP**:

- **GL_QUADS** – каждые четыре вершины определяют свой четырехугольник.
- **GL_QUAD_STRIP** – рисуются связанные четырехугольники. Первая, вторая, третья и четвертая вершины определяют первый четырехугольник. Третья, четвертая, пятая и шестая вершины – второй четырехугольник и т.д. (2n-1), 2n, (2n+1) и (2n+2) вершины задают n-й четырехугольник.

Если в примитиве вершин больше четырех, то речь идет о многоугольники, которые в OpenGL называют полигонами и задают вызовом glBegin с параметром **GL_POLYGON** соответственно. При этом *все* вершины, переданные в внутри связки glBegin//glEnd определяют *один многоугольник*.

Для многоугольников так же можно задавать стили при помощи вышеописанной функции glPolygonMode, толщину линии, толщину точек и цвет.

Массивы вершин

При отрисовке графических сцен, как правило, количество вершин не ограничивается единицами и даже десятками. Поэтому использовать повершинную отрисовку совершенно нерационально. На этот случай в OpenGL есть механизм использования массивов для хранения координат всех вершин объектов, что значительно упрощает и оптимизирует код. Этот метод отправляет на просчёт сразу набор вершинных данных. То есть вершинные параметры, такие как координаты, нормали и цвета вершин, текстурные координаты задаются не отдельными функциями OpenGL для

каждой вершины, а отдельными массивами координат. Используя массив вершин (цветов, нормалей, текстур и т.д.), мы можем предварительно скомпилировать или изменить геометрию в любое время, а затем перенести все эти данные сразу.

Итак, если вершин много, то, чтобы не вызывать для каждой функцию glVertex(), можно объединить координаты вершин в массивы, а затем уже с помощью специального набора функций выбирать их оттуда по необходимости. Для этого необходимо реализовать следующий алгоритм:

1. Загружаем данные геометрии в один или несколько массивов (инициализируем массив координат вершин, а также, при необходимости, массивы цветов и/или нормалей);
2. Передаем OpenGL указатели на область хранения необходимых данных, а также определяем способ хранения и выборки значений из массива;
3. Включаем режим отрисовки объектов через массивы, явно указывая, какой именно тип массива (с какими данными) будем использовать;
4. Реализуем отрисовку примитива (примитивов) одной из доступных функций OpenGL;
5. Выключаем режим отрисовки объектов через массивы.

Рассмотрим этот алгоритм подробнее.

1. *Загружаем данные геометрии в один или несколько массивов*

Для этого инициализируем одномерный массив необходимыми значениями, используя средства того алгоритмического языка, на котором пишется программа.

Например, в C++ задаем массив вершин вида:

```
GLfloat pVerts[]= {-0.5f, -0.5f,      0.5f,  0.0f,    -0.5f , 0.5f};
```

здесь последовательно расположено по две координаты для трёх вершин.

2. *Передаем OpenGL указатели на область хранения необходимых данных*

Для этого вызываем функцию OpenGL

void glVertexPointer (GLint *size*, GLenum *type*, GLsizei *stride*, void **ptr*),

которая определяет способ хранения и выборки координат вершин из массива, а так же передает указатель на сам массив.

Аргументы функции:

- *size* – определяет число координат вершины (может быть 2, 3, 4).
- type – определяет тип данных массива (может быть равен GL_SHORT, GL_INT, GL_FLOAT, GL_DOUBLE).
- *stride* – задает смещение от координат одной вершины до координат следующей; если *stride* равен нулю, это значит, что координаты расположены последовательно. Иногда, например, удобно в массиве располагать не только значения координат вершин, но и некоторые сопутствующие атрибуты (текстурные координаты, нормали вершин, значения цвета и т.д.). В этом случае параметр *stride* позволяет "раздвинуть" координаты вершин в массиве и оставить места для дополнительных данных.
- *ptr* – указывает адрес массива, где находятся данные.

Например, для заданного выше массива вершин с учетом отрисовки в двумерном пространстве (на одну вершину – две кординаты) вызов этой функции можно записать так:

glVertexPointer(2, GL_FLOAT, 0, &pVerts);

Первый параметр в этой функции говорит, сколько координат идёт на одну вершину, здесь – 2. Второй параметр отвечает за тип координат. Третий параметр равен шагу в байтах между данными с координатами для каждой вершины – 0 в нашем случае означает, что координаты лежат

плотно друг за другом. Четвёртый параметр – собственно сам указатель на массив с координатами.

Аналогично можно определить массив нормалей, цветов и некоторых других атрибутов вершины, используя команды

void NormalPointer (GLenum *type*, GLsizei *stride*, void **pointer*)

void ColorPointer (GLint *size*, GLenum *type*, GLsizei *stride*, void **pointer*)

3. *Включаем режим отрисовки объектов через массивы*

Для того чтобы эти массивы можно было использовать в дальнейшем, надо вызвать команду

void glEnableClientState (GLenum *array*)

с параметрами *array* GL_VERTEX_ARRAY, GL_NORMAL_ARRAY, GL_COLOR_ARRAY соответственно.

4. *Реализуем отрисовку примитива (примитивов) одной из доступных функций OpenGL*

Отрисовку объектов с использованием массивов можно выполнять разными способами: можно выбирать вершины по одной, поиндексно, и рисовать по принципу повершинной отрисовки (способ 1), а можно указывать сразу весь набор необходимых вершин в массиве и рисовать необходимое количество примитивов, по сути, одной строкой (способ 2).

Способ 1

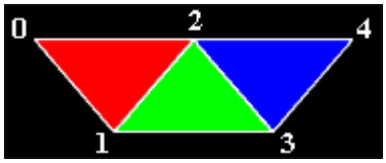
Для отображения содержимого массивов используется команда:

void glArrayElement (GLint *index*),

которая передает OpenGL атрибуты вершины, используя элементы массива с номером *index*. Это аналогично последовательному применению команд вида glColor, glNormal, glVertex с соответствующими параметрами.

В случае если одна вершина входит в несколько примитивов, то вместо дублирования ее координат в массиве удобно использовать ее индекс.

Этот метод позволяет работать с индексным заданием геометрических объектов для вывода на просчёт адаптера.



Эта функция используется внутри пары glBegin() и glEnd().

Собственно ей мы и задаём индексы. А массивы с данными о самих вершинах, как и говорилось, устанавливаются ранее.

Пример:

```
void DrawObjects()
{
//инициализация массива вершин (по 3 координаты на вершину)
  static GLfloat pVerts[]= {-0.5f,  0.0f, 0.0f,
                           -0.25f,-0.4f, 0.0f,
                            0.0f,  0.0f, 0.0f,
                            0.25f,-0.4f, 0.0f,
                            0.5f,  0.0f, 0.0f};
  //включаем режим использования массивов вершин
  glEnableClientState(GL_VERTEX_ARRAY);
//задаем способ хранение и указываем массив
  glVertexPointer(3,GL_FLOAT,0,pVerts);
//рисуем
  glBegin(GL_TRIANGLES);
   glColor3ub(255,0,0); // красный
    glArrayElement(0);
    glArrayElement(1);
    glArrayElement(2);
   glColor3ub(0,255,0); // зеленый
    glArrayElement(1);
    glArrayElement(3);
    glArrayElement(2);
   glColor3ub(0,0,255); // синий
    glArrayElement(2);
    glArrayElement(3);
    glArrayElement(4);
  glEnd();
//выключаем режим использования массивов вершин
  glDisableClientState(GL_VERTEX_ARRAY);
}
```

Этот пример выводит три цветных треугольника.

Способ 2

Используем функцию

void glDrawArrays (GLenum *mode*, GLint *first*, GLsizei *count*),

которая рисует *count* примитивов, определяемых параметром *mode*, используя элементы из массивов с индексами от *first* до *first*+*count*-1. Это эквивалентно вызову команды glArrayElement() с соответствующими индексами.

В нашем случае – для одного треугольника используем 3 вершины.

glDrawArrays(GL_TRIANGLES,0,3);

Обратите внимание, что *эта функция должна вызываться вне пары glBegin() и glEnd()*.

Пример:

```
void DrawObjects()
{
  static GLfloat pVerts[]= {-0.5f, -0.5f, 0.0f,
                             0.5f,  0.0f, 0.0f,
                            -0.5f , 0.5f, 0.0f};
  glEnableClientState(GL_VERTEX_ARRAY);
  glVertexPointer(3,GL_FLOAT,0,pVerts);
  glColor3ub(255,0,0); // красный
  glDrawArrays(GL_TRIANGLES,0,3);
}
```

Способ 3

Так индексы также можно задавать отдельным массивом и выводить объект одним вызовом без пары glBegin() и glEnd().

Для этого надо вызвать команду

void glDrawArrays (GLenum *mode*, GLsizei *count*, GLenum *type*, void **indices*),

где *indices* – это указатель на массив номеров вершин (индексов), которые надо использовать для построения примитивов, *type* определяет тип

элементов этого массива: GL_UNSIGNED_BYTE, GL_UNSIGNED_SHORT, GL_UNSIGNED_INT (обычно под индексы используют unsigned short), а *count* задает количество индексов, берущихся из массива.

Заметьте, что указатель *indices* можно расценить как указатель на элемент, с которого будет происходить расчёт объекта. То есть вы можете поставить указатель на любой элемент в массиве.

Пример:

```
void DrawObjects()
{
  static GLfloat pVerts[]= {-0.5f,  0.0f, 0.0f,
                           -0.25f,-0.4f, 0.0f,
                            0.0f,  0.0f, 0.0f,
                            0.25f,-0.4f, 0.0f,
                            0.5f,  0.0f, 0.0f};
  static GLushort pInds[] =
  { 0,1,2 , 1,3,2 , 2,3,4 };
  glEnableClientState(GL_VERTEX_ARRAY);
  glVertexPointer(3,GL_FLOAT,0,pVerts);
  glDrawElements(GL_TRIANGLES, 9, GL_UNSIGNED_SHORT, pInds);
}
```

Пример выводит те же самые треугольники, поскольку цвет не выставляется, то все треугольники будут белыми.

5. *Выключаем режим отрисовки объектов через массивы*

   После окончания работы с массивом желательно вызвать команду

   void glDisableClientState (GLenum *array*)

   с соответствующим значением параметра *array*.

Методика выполнения лабораторной работы

В течение занятия необходимо:

1) изучить теоретическую информацию, предлагаемую на лекции, в данном пособии и в методических указаниях;

2) сформировать модель решения с учетом использования для отрисовки массивов вершин и цветов;

3) выполнить отрисовку типовых графических примитивов с использованием массивов вершин и цветов, согласно варианту задания.

Варианты заданий

Построить изображение двумерной сцены из нескольких фигур (картину), по согласованию с преподавателем.

Использовать один (!) массив вершин и один (!) массив цветов для ВСЕХ объектов сцены – это гарантия повышенного результирующего балла за лабораторную работу.

Объекты сцены должны состоять из различных примитивов (не менее 5 типов), для их отрисовки должны использоваться разные методы и способы (отрисовка граней, способы закрашивания, толщина линий, цвет и т.д.). Один из объектов (как минимум) должен содержать не менее 25 вершин.

При отрисовке предпочтительно использовать инструменты используемого алгоритмического языка на максимально доступном уровне.

Содержание отчета

Результатом выполнения лабораторной работы должны стать:

1) Отчет (в печатном и электронном вариантах), состоящий из следующих пунктов:
   a. постановка задачи;
   b. блок-схема инициализации;
   c. программный код предлагаемого решения;
   d. скриншоты результатов работы программы.
2) Программная реализация решения

Защита лабораторной работы в устной форме должна быть произведена студентом преподавателю в срок до начала следующего лабораторного занятия. Во время защиты преподавателю необходимо предоставить:

- отчет по лабораторной работе (в печатном виде);
- отчет по лабораторной работе (в электронном виде);
- программную реализацию решения.

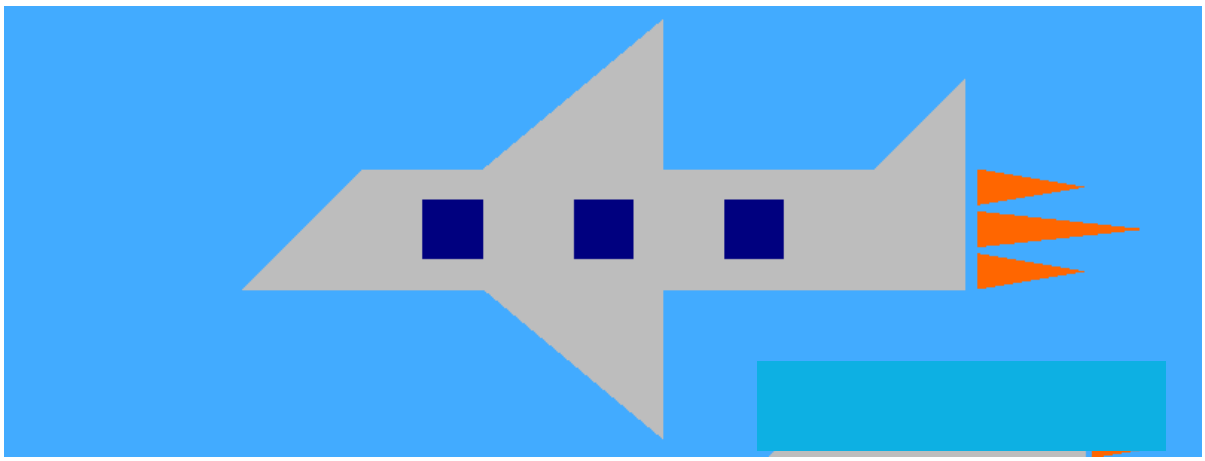
Оценка (в баллах) выставляется за **устный отчет** студента по предоставленным преподавателю материалам.

Пример выполнения лабораторной работы

```
#include <Windows.h>
#include <iostream>
#include <GLFW/glfw3.h>
#ifndef M_PI
#define M_PI 3.14159265358979323846
#endif

int main()
{
    if (!glfwInit())
    {
        std::cout << "Initalize" << std::endl;
        return -1;
    }
    GLFWwindow* window = glfwCreateWindow(810, 810,
"AIRPLANE", nullptr, nullptr);
    glfwMakeContextCurrent(window);
    while (!glfwWindowShouldClose(window))
    {
        glfwPollEvents();
        glClearColor(0.26f, 0.67f, 1.0f, 1.0f); //очищение
экрана и заполнение
        glClear(GL_COLOR_BUFFER_BIT);
        int width, height;
        glfwGetFramebufferSize(window, &width, &height);
        glViewport(0, 0, width, height); //область вывода;
//инициализация массива вершин
        POINTFLOAT vertices[] = {
        {-0.4, -0.1}, //нос
        {-0.4, 0.1},{-0.6, -0.1},{-0.4, -0.1}, //корпус
        {-0.4,0.1},{0.45,0.1},{0.45,-0.1},{-0.3, -0.05},//окно
     {-0.3,0.05},{-0.2,0.05},{-0.2,-0.05{-0.05,-0.05},//окно2
     {-0.05,0.05},{0.05,0.05},{0.05,-0.05},{0.2,-0.05},//окно3
        {0.2,0.05},{0.3,0.05},{0.3, -0.05},{0.45, 0.1},//хвост
        {0.45, -0.1},{0.6, -0.1},{0.45, 0.1}, //хвост
        {0.6, -0.1},{0.6, 0.25},{-0.2, 0.1},//крыло
        {0.1, 0.1},{0.1, 0.35},{-0.2, -0.1},//крыло2
        {0.1, -0.1},{0.1, -0.35},{-0.8, 0.7},  //облако
        {-0.9,0.65},{-0.8,0.6},{-0.7,0.65},{0.62,0.1},//пламя
        {0.62, 0.04},{0.8, 0.07},{0.62, 0.03},//пламя 2
        {0.62, -0.03},{0.9, 0.0},{0.62, -0.04},//пламя 3
        {0.62, -0.1},{0.8, -0.07},{-0.8, -0.8},//облако2
        {-0.9, -0.75},{-0.8, -0.7},{-0.7, -0.75},
        };

        glVertexPointer(2, GL_FLOAT, 0, &vertices);
        glEnableClientState(GL_VERTEX_ARRAY);
        glColor3f(0.74f, 0.74f, 0.74f);
```

```
        glDrawArrays(GL_TRIANGLES, 0, 3);

        glColor3f(0.74f, 0.74f, 0.74f);
        glDrawArrays(GL_QUADS, 3, 4); //основа
        glColor3f(0.0f, 0.0f, 0.5f);
        glDrawArrays(GL_QUADS, 7, 12); //окна
        glColor3f(0.74f, 0.74f, 0.74f);
        glDrawArrays(GL_TRIANGLES, 19, 5);// хвост
        glDrawArrays(GL_TRIANGLES, 25, 6);//крылья
        glColor3f(1.0f, 0.4f, 0.0f);
        glDrawArrays(GL_TRIANGLES, 35, 9);//пламя из турбин
      glfwSwapBuffers(window);

        glDisableClientState(GL_VERTEX_ARRAY);
        glfwDestroyWindow(window);
        glfwTerminate();
    return 0;
  }
```

Результат выполнения:



Дополнительный материал

Пример реализации отрисовки графической сцены в C#.

В программе вершины объекта разложены по разным массивам исключительно для удобства чтения листинга. На самом деле, можно было легко обойтись одним общим массивом и, практически, одной единственной функцией отрисовки. Помимо этого, код содержит избыточное количество установочных функций OpenGL.

Задание: оптимизируйте данный код.

Листинг программы:

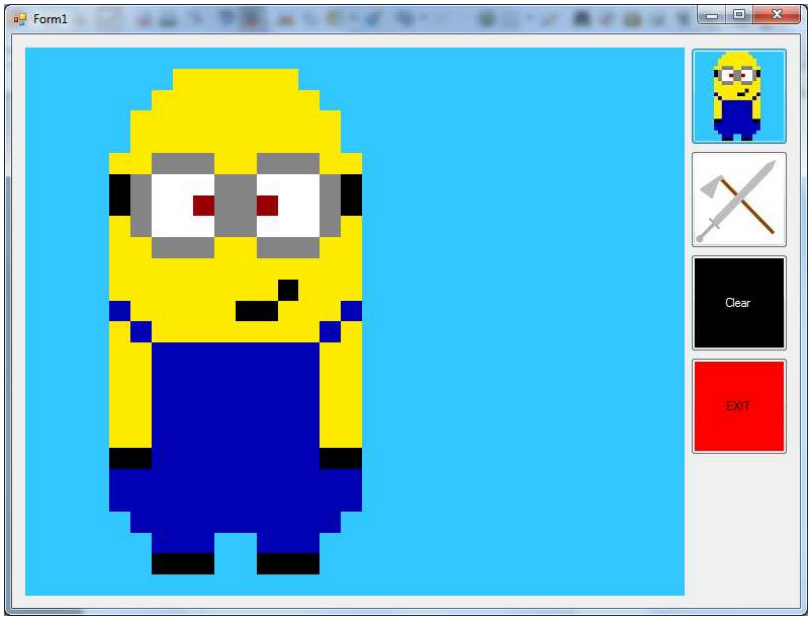
```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;
using Tao.OpenGl;
using Tao.Platform.Windows;

namespace _3labgraf
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
            OpenGlControl1.InitializeContexts();
        }

        private void Form1_Load(object sender, EventArgs e)
        {
            // отчиСтка окна
            Gl.glClearColor(255, 255, 255, 1);

            // установка порта вывода в соответствии с разме-
рами элемента ant
            //Gl.glViewport(0, 0, ant.Width, ant.Height);

            Gl.glMatrixMode(Gl.GL_MODELVIEW);
            Gl.glLoadIdentity();
        }
        private void start_Click(object sender, EventArgs e)
        {
            Gl.glClear(Gl.GL_COLOR_BUFFER_BIT |
Gl.GL_DEPTH_BUFFER_BIT);
            //фон
            Gl.glBegin(Gl.GL_POLYGON);
            Gl.glColor3ub(50, 200, 255);
            Gl.glVertex2d(0, 0);
            Gl.glVertex2d(300, 0);
            Gl.glVertex2d(300, 200);
            Gl.glVertex2d(0, 200);
            Gl.glEnd();

            Gl.glEnableClientState(Gl.GL_VERTEX_ARRAY);
            /************************************/
            float[] yellow = new float[]
```

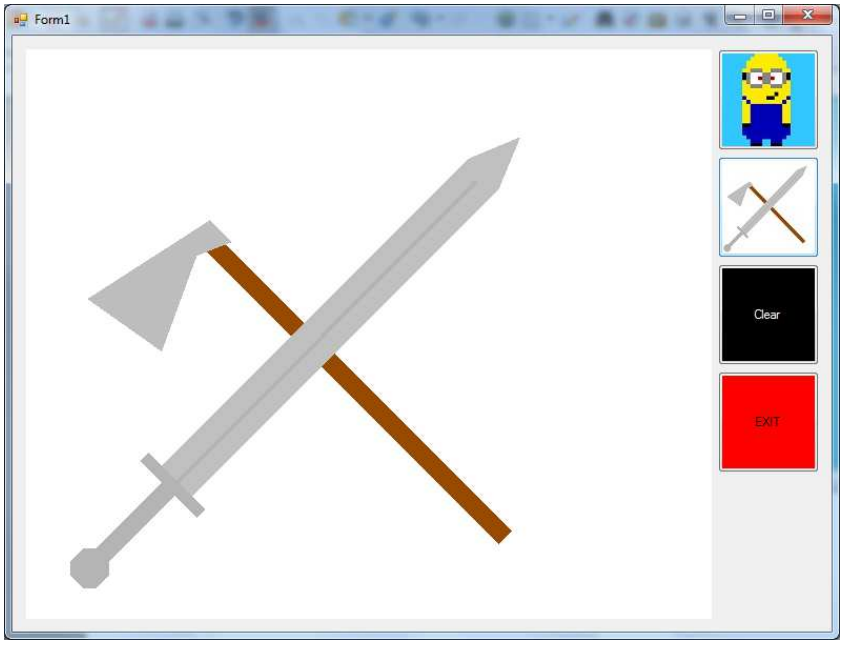
```
 { 25, 105, 25, 115, 75, 115, 75, 105, 20, 105, 80, 105, 80,
100, 20, 100, 30, 120, 70, 120, 70, 115, 30, 115, 35, 125, 65,
125, 65, 120, 35, 120, 20, 90, 80, 90, 80, 35, 20, 35
};
            Gl.glColor3ub(255, 235, 0);
            Gl.glVertexPointer(2, Gl.GL_FLOAT, 0, yellow);
            Gl.glDrawArrays(Gl.GL_QUADS, 0, 20);
            /*************************************/

            float[] eyes = new float[]
{30, 100, 70, 100, 70, 85, 30, 85, 55, 95, 60, 95, 60, 90, 55,
90, 40, 95, 45, 95, 45, 90, 40, 90
};
            Gl.glColor3ub(255, 255, 255);
            Gl.glVertexPointer(2, Gl.GL_FLOAT, 0, eyes);
            Gl.glDrawArrays(Gl.GL_QUADS, 0, 8);
            /*************************************/

            float[] glases = new float[]
 {30, 100, 30, 105, 45, 105, 45, 100, 55, 100, 55, 105, 70,
105, 70, 100, 25, 100, 30, 100, 30, 85, 25, 85, 45, 100, 55,
100, 55, 85, 45, 85, 70, 100, 75, 100, 75, 85, 70, 85, 30, 80,
30, 85, 45, 85, 45, 80, 55, 80, 55, 85, 70, 85, 70, 80
};
            Gl.glColor3ub(133, 133, 133);
            Gl.glVertexPointer(2, Gl.GL_FLOAT, 0, glases);
            Gl.glDrawArrays(Gl.GL_QUADS, 0, 28);
            /*************************************/

            float[] blue = new float[]
 {30, 60, 70, 60, 70, 15, 30, 15, 30, 15, 45, 15, 45, 10, 30,
10, 55, 15, 70, 15, 70, 10, 55, 10, 20, 30, 80, 30, 80, 20,
20, 20, 25, 20, 75, 20, 75, 15, 25, 15, 20, 70, 25, 70, 25,
65, 20, 65, 25, 65, 30, 65, 30, 60, 25, 60, 75, 70, 80, 70,
80, 65, 75, 65, 70, 65, 75, 65, 75, 60, 70, 60
};
            Gl.glColor3ub(0, 0, 180);
            Gl.glVertexPointer(2, Gl.GL_FLOAT, 0, blue);
            Gl.glDrawArrays(Gl.GL_QUADS, 0, 36);
            /*************************************/

            float[] black = new float[]
 {20, 100, 25, 100, 25, 90, 20, 90, 75, 100, 80, 100, 80, 90,
75, 90, 20, 35, 30, 35, 30, 30, 20, 30, 70, 35, 80, 35, 80,
30, 70, 30, 50, 70, 60, 70, 60, 65, 50, 65, 60, 75, 65, 75,
65, 70, 60, 70, 30, 10, 45, 10, 45, 5, 30, 5, 55, 10, 70, 10,
70, 5, 55, 5
};
            Gl.glColor3ub(0, 0, 0);
            Gl.glVertexPointer(2, Gl.GL_FLOAT, 0, black);
            Gl.glDrawArrays(Gl.GL_QUADS, 0, 32);
```

```
            OpenGlControl1.Invalidate();
        }

        private void start2_Click(object sender, EventArgs e)
        {
            Gl.glClear(Gl.GL_COLOR_BUFFER_BIT |
Gl.GL_DEPTH_BUFFER_BIT);
            //топор
            //рукоять
            Gl.glBegin(Gl.GL_POLYGON);
            Gl.glColor3ub(150, 75, 0);
            Gl.glVertex2d(108, 17);
            Gl.glVertex2d(38, 87);
            Gl.glVertex2d(41, 90);
            Gl.glVertex2d(111, 20);
            Gl.glEnd();
            //лезвие
            Gl.glBegin(Gl.GL_QUADS);
            Gl.glColor3ub(190, 190, 190);
            Gl.glVertex2d(31, 61);
            Gl.glVertex2d(14, 73);
            Gl.glVertex2d(47, 86);
            Gl.glVertex2d(42, 91);
            Gl.glEnd();
            /*************************************/
            //меч
            //навершие
            Gl.glBegin(Gl.GL_POLYGON);
            Gl.glColor3ub(180, 180, 180);
            Gl.glVertex2d(10, 10);
            Gl.glVertex2d(10, 13);
            Gl.glVertex2d(13, 16);
            Gl.glVertex2d(16, 16);
            Gl.glVertex2d(19, 13);
            Gl.glVertex2d(19, 10);
            Gl.glVertex2d(16, 7);
            Gl.glVertex2d(13, 7);
            Gl.glEnd();
            //рукоять
            Gl.glBegin(Gl.GL_QUADS);
            Gl.glColor3ub(180, 180, 180);
            Gl.glVertex2d(16, 16);
            Gl.glVertex2d(31, 31);
            Gl.glVertex2d(34, 28);
            Gl.glVertex2d(19, 13);
            Gl.glEnd();
            //гарда
            Gl.glBegin(Gl.GL_QUADS);
            Gl.glColor3ub(180, 180, 180);
            Gl.glVertex2d(26, 36);
```

```
            Gl.glVertex2d(28, 38);
            Gl.glVertex2d(41, 25);
            Gl.glVertex2d(39, 23);
            Gl.glEnd();
            //лезвие
            Gl.glBegin(Gl.GL_POLYGON);
            Gl.glColor3ub(192, 192, 192);
            Gl.glVertex2d(31, 35);
            Gl.glVertex2d(101, 105);
            Gl.glVertex2d(113, 110);
            Gl.glVertex2d(108, 98);
            Gl.glVertex2d(38, 28);
            Gl.glEnd();
            Gl.glLineWidth(5);
            Gl.glBegin(Gl.GL_LINES);
            Gl.glColor3ub(180, 180, 180);
            Gl.glVertex2d(31.5, 28);
            Gl.glVertex2d(103, 100);
            Gl.glEnd();

            OpenGlControl1.Invalidate();
        }

        private void clear_Click(object sender, EventArgs e)
        {
            Gl.glClear(Gl.GL_COLOR_BUFFER_BIT |
Gl.GL_DEPTH_BUFFER_BIT);
            Gl.glBegin(Gl.GL_POLYGON);
            Gl.glColor3ub(0, 0, 0);
            Gl.glVertex2d(0, 0);
            Gl.glVertex2d(300, 0);
            Gl.glVertex2d(300, 200);
            Gl.glVertex2d(0, 200);
            Gl.glEnd();
            OpenGlControl1.Invalidate();
        }

        private void button1_Click(object sender, EventArgs e)
        {
            Application.Exit();
        }
    }
}
```

Скриншоты работы программы:





Контрольные вопросы

1) Перечислите способы отрисовки треугольников и опишите принципы их работы.
2) Объясните понятия лицевой и обратной грани многоугольника.
3) Опишите алгоритм отрисовки примитивов с использованием массива вершин.
4) Опишите алгоритм отрисовки примитивов с использованием массива цветов.
5) Напишите программный код для отрисовки 5 треугольников разного цвета с использованием одного массива вершин.
6) Объясните способ отрисовки прерывистых линий.
7) Нарисуйте круг с помощью функций OpenGL.

ЛАБОРАТОРНАЯ РАБОТА №4

OpenGL: Исследование и реализация алгоритмов трансляции, поворота и сдвига двумерных объектов

Цель работы

Изучить алгоритмы преобразований видовой матрицы, ознакомиться с функционалом OpenGL для реализации смещения, поворота и масштабирования объектов сцены.

Теоретическая информация

Матрицы преобразований в OpenGL

Для задания различных преобразований объектов сцены в OpenGL используются операции над матрицами, при этом *различают три типа матриц*:

- видовая,
- проекций,
- текстуры.

Все они имеют размер 4x4. Хотя некоторые 3D API позволяют задавать модельную и видовую матрицу раздельно, OpenGL объединяет их в одной матрице называемой модельно-видовой или видовой матрицей (modelview matrix). Эта матрица определяет преобразование координат объектного пространства в видовое пространство. Видовая матрица определяет преобразования объекта в мировых координатах, такие как параллельный перенос, изменение масштаба и поворот.

Матрица проекций задает как будут проецироваться трехмерные объекты на плоскость экрана (в оконные координаты), а матрица текстуры определяет наложение текстуры на объект.

Довольно часто приложение начинает с загрузки видового преобразования в видовую матрицу, а затем добавляет необходимые модельные преобразования. Позиции источников света в OpenGL сохраняются в видовых координатах. Нужное преобразование должно быть загружено в видовую матрицу перед указанием позиции источника света, или вы не получите ожидаемого светового эффекта. Расчет освещенности в OpenGL происходит на по-вершинном базисе в видовой координатной системе. Чтобы вычислить отражение позиции источников света и нормали к поверхностям должны быть в одной координатной системе. Поскольку реализации OpenGL часто расчитывают освещенность в видовых координатах, нормали также должны быть приведены к видовым координатам. Это делается с помощью обратно транспонированной видовой матрицы. После этого для определения освещенности каждой вершины могут быть применены формулы OpenGL.

После преобразования координат в видовое пространство, следующим шагом является определение объема видимости. Это – область трехмерной сцены, которая будет видимой в финальном изображении. Преобразование, которое переводит объекты, находящиеся в объеме видимости, в усеченное пространство (clip space) (это пространство удобное для отсечения) называется проекционным (projection transformation) (рис. 1).

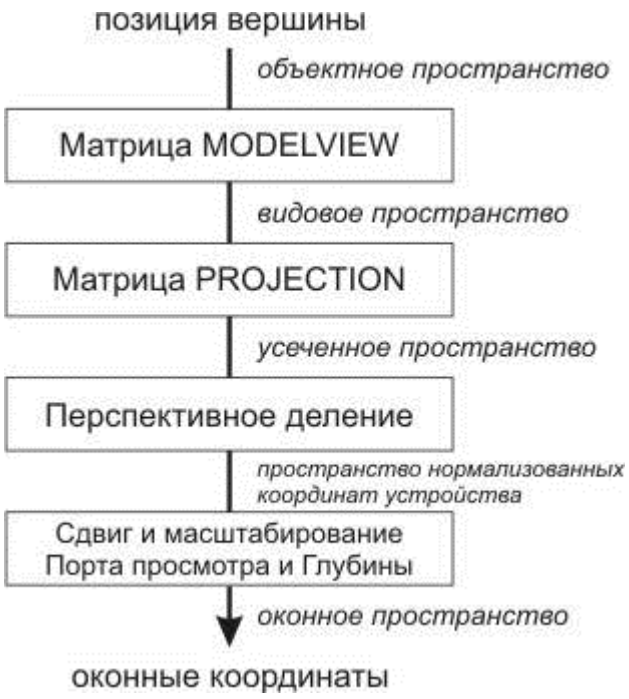


Рисунок 1. Конвейер координатных преобразований

Команды управления текущими матрицами преобразования

Выбор требуемого режима осуществляется командой

glMatrixMode (GLenum mode),

которая с помощью значения перечисления *mode* определяет, какую из матриц преобразования размером 4x4 сделать текущей и вносить в нее изменения последующими командами:

- mode = **GL_MODELVIEW**: Модельные преобразования (модельно-видовая матрица) – применяются к размещению объектов на сцене.
- mode = **GL_PROJECTION**: Видовые преобразования (проекционная матрица) – применяются к размещению и ориентации точки обзора (настройка фотокамеры).
- mode = **GL_TEXTURE**: Текстурные преобразования (текстурная матрица) – применяются для управления текстурами заполнения

объектов. В данной лабораторной работе мы не будем применять текстуры, поэтому и останавливаться здесь на них не будем.

В начальном состоянии по умолчанию любая из этих матриц преобразования является единичной

Для определения элементов матрицы текущего типа вызывается команда

void glLoadMatrix [f d] (GLtype *m),

где *m* указывает на массив из 16 элементов типа float или double в соответствии с названием команды, при этом сначала в нем должен быть записан первый столбец матрицы, затем второй, третий и четвертый.

Команда

void glLoadIdentity (void)

заменяет текущую матрицу на единичную (матрицей с единицами по главной диагонали и равными нулю всеми остальными элементами).

Часто нужно *сохранить содержимое текущей матрицы* для дальнейшего использования, для чего используют команды

void glPushMatrix (void)

void glPopMatrix (void)

Они записывают и восстанавливают текущую матрицу из стека, причем для каждого типа матриц стек свой. Для видовых матриц глубина стека равна как минимум 32, а для двух оставшихся типов как минимум 2.

Для умножения текущей матрицы слева на другую матрицу используется команда

void glMultMatrix [f d] (GLtype *m),

где *m* должен задавать матрицу размером 4x4 в виде массива с описанным расположением данных. Однако обычно для изменения матрицы того или иного типа удобно использовать специальные команды, которые по значениям своих параметров создают нужную матрицу и перемножают ее с текущей.

Видовое (модельное) преобразование

К видовым преобразованиям будем относить *перенос, поворот и изменение масштаба* вдоль координатных осей. Для проведения этих операций достаточно умножить на соответствующую матрицу каждую вершину объекта и получить измененные координаты этой вершины:

$(x', y', z', 1)^T = M * (x, y, z, 1)^T$,

где M – матрица видового преобразования.

В OpenGL *текущая матрица преобразований является произведением двух матриц – матрицы модели и матрицы проецирования,* при этом формируется единая матрица преобразования, которая применяется ко всем вершинам всех геометрических объектов.

Матрица модели – glMatrixMode (GL_MODELVIEW) связана с координатами объектов. Это матрица в базисе видовых координат, она используется для построения картинки в том виде как её видит наблюдатель.

Матрица проецирования – glMatrixMode (GL_PROJECTION). Матрица в системе координат устройства. Вычисляет нормализованные координаты, которые преобразуются в экранные после трансформаций, связанных с областью вывода.

В OpenGL все объекты рисуются в начале координат, т.е. в точке (0,0,0).

Для того чтобы изобразить объект в точке (x1,y1,z1), надо переместить начало координат в эту точку, т.е. перейти к новым координатам. Перспективное преобразование и проецирование производится аналогично.

Сама итоговая матрица преобразований может быть создана с помощью следующих команд:

void glTranslate [f d] (GLtype x, GLtype y, GLtype z)

void glRotate [f d] (GLtype angle, GLtype x, GLtype y, GLtype z)

void glScale [f d] (GLtype x, GLtype y, GLtype z)

Перенос (трансляция)

Трансляция – это сдвиг объекта в пространстве без изменения его формы.

Представьте треугольник вокруг центра нашей матрицы (0, 0, 0). Вершины пусть будут: (0,-1, 0); (-1,-1, 0); (1,-1, 0);

Есть два способа сдвинуть треугольник вправо. Первый состоит в сдвиге вершин треугольника: (1,-1, 0); (0, -1, 0); (2, -1, 0); Другой способ – надо сдвинуть всю матрицу вправо и перерисовать треугольник вокруг нового центра матрицы. Вот это и есть трансляция.

Трансляции осуществляются при помощи функции

glTranslate [f d] (Dx, Dy, Dz) ,

которая сдвигает начало координат на (Dx, Dy, Dz)

Следует задать 3 параметра, значения сдвига матрицы по x/ y/ z. Новый центр матрицы будет в точке с этими координатами.

Пример:

```
glLoadIdentity();
 glTranslatef(0.9,0.0,0.0);
 glBegin( GL_TRIANGLES );
   glColor3f(1.0, 0.0, 0.0);
    glVertex3f(0.0, 1.0, 0.0);
   glColor3f(0.0, 1.0, 0.0);
```

```
    glVertex3f(-1.0, -1.0, 0.0);
   glColor3f(0.0, 0.0, 1.0);
    glVertex3f(1.0, -1.0, 0.0);
glEnd();
```

С помощью glLoadIdentity() мы выполняем сброс матрицы, располагая ее в первоначальных координатах. Если удалить эту строку, матрица каждый раз будет сдвигаться (а раз – это шаг в главном цикле).

Чтобы этого не происходило, можно так же воспользоваться функциями:

glPushMatrix()

glPopMatrix()

Поворот

Поворот в OpenGL реализуется функцией

glRotate [f d] (ϕ, x,y,z) ,

которая поворачивает систему координат на угол ϕ (в градусах) против часовой стрелки вокруг вектора (x, y, z).

Угол – некоторое число (обычно переменная), которое задает размер поворота в градусах вокруг выбранной далее оси.

(x, y, z) – если один из параметров равен 1.0f, OpenGL будет вращать объект вдоль соответствующей оси. Поэтому если Вы имеете glRotatef(10.0f,0.0f,1.0f,0.0f), объект будет поворачиваться на 10 градусов по оси Y. Если glRotatef (5.0f,1.0f,0.0f,1.0f), объект будет поворачиваться на 5 градусов по обеим осям X и Z.

Чтобы лучше понять вращения по осям X, Y и Z, разберем это на примерах.

Ось X: предположим, Вы работаете за токарным станком. Заготовка перемещается слева направо (также как ось X в OpenGL). Заготовка вра-

щается вокруг оси перемещения – так же мы вращаем что-то вокруг оси X в OpenGL.

Ось Y: представьте, что Вы стоите посреди поля. Огромный торнадо приближается к Вам. Центр торнадо перемещается от земли в небо (верх и низ, подобно оси Y в OpenGL). Предметы захваченные торнадо кружатся вдоль центра торнадо слева направо или справа налево. Когда вы вращаете что-то вокруг оси Y в OpenGL, это что-то будет вращаться так же.

Ось Z: Вы смотрите на переднюю часть вентилятора. Передняя часть вентилятора ближе к Вам, а дальняя часть дальше от Вас (также как ось Z в OpenGL). Лопасти вентилятора вращаются вокруг центра вентилятора по часовой или против часовой стрелки. Когда Вы вращаете что-то вокруг оси Z в OpenGL, это что-то будет вращаться так же.

Угол поворота можно задавать в некоторой глобальной переменной, например:

GLfloat ugol; // Угол для треугольника

Тогда вызов функции поворота можно записать следующим образом

glRotatef(ugol,0.0f,1.0f,0.0f); // Вращение треугольника по оси Y

Пример:

```
Здесь будет нарисован закрашенный сглаженный треугольник. Треугольник будет нарисован с левой стороны экрана, и будет вращаться по оси Y слева направо.
glBegin(GL_TRIANGLES); //Начало рисования треугольника
glColor3f(1.0f,0.0f,0.0f);      // Верхняя точка - красная
glVertex3f(0.0f,1.0f,0.0f);  //Первая точка
glColor3f(0.0f,1.0f,0.0f);      //Левая точка - зеленная
glVertex3f(-1.0f,-1.0f,0.0f);  //Вторая
glColor3f(0.0f,0.0f,1.0f);      //Правая - синия
glVertex3f(1.0f,-1.0f,0.0f);  //Третья
 glEnd();  // Конец рисования
glLoadIdentity();
```

Масштабирование

Преобразование масштабирования визуально увеличивает или уменьшает размеры объекта. Но реально воздействует на масштаб системы координат.

Команда масштабирования

glScale (arg1, arg2, arg3)

где *arg1, arg2, arg3* – коэффициенты масштабирования (масштабные множители) по каждой из осей.

Если масштабные множители больше единицы объект растягивается в заданном направлении, если меньше – объект сжимается. Масштабные множители могут иметь отрицательные значения, тогда изображение переворачивается по соответствующей оси. При двумерных построениях значение коэффициента по оси Z игнорируется.

ВАЖНО: После применения любых команд видовых преобразований следует восстановить нормальный масштаб, чтобы каждое следующее обращение к обработчику перерисовки экрана не приводило бы к последовательному изменению изображения. Контроль за итоговым видом сцены полностью возлагается на плечи разработчика, поэтому необходимо внимательно следить и за последовательностью применения функций преобразования и за восстановлением исходного состояния модельно-видовой матрицы.

Совет: *В случае если надо, например, повернуть один объект сцены, а другой оставить неподвижным, удобно сначала сохранить текущую видовую матрицу в стеке командой glPushMatrix(), затем вызвать glRotate() с нужными параметрами, описать примитивы, из которых состоит этот объект, а затем восстановить текущую матрицу командой glPopMatrix().*

Методика выполнения лабораторной работы

В течение занятия необходимо:

1) изучить теоретическую информацию, предлагаемую на лекции, в данном пособии и в методических указаниях;
2) сформировать модель решения с учетом использования преобразований модельно-видовой матрицы, согласно варианту задания;
3) реализовать программное решение, согласно варианту задания.

Варианты заданий

1. Написать программу, отрисовывающую два плоских геометрических объекта, состоящих из нескольких примитивов (минимум 3-х), расположенных параллельно друг другу, относительно центра экрана. Реализовать меню, с помощью которого организовать выполнение следующих действий:
    a. Поворот правой фигуры на заданный угол вокруг оси x;
    b. Смещение левой фигуры на заданное расстояние от центра;
    c. Увеличение одного примитива из правой фигуры (по выбору) в два раза.
2. Написать программу, отрисовывающую два плоских геометрических объекта, состоящих из нескольких примитивов (минимум 3-х), расположенных параллельно друг другу, относительно центра экрана. Реализовать меню, с помощью которого организовать выполнение следующих действий:
    a. Поворот левой фигуры на заданный угол вокруг оси z;
    b. Смещение правой фигуры на заданное расстояние вниз от центра;
    c. Уменьшение одного примитива фигуры по выбору в два раза.

3. Написать программу, отрисовывающую два плоских геометрических объекта, состоящих из нескольких примитивов (минимум 3-х), расположенных параллельно друг другу, относительно центра экрана. Реализовать меню, с помощью которого организовать выполнение следующих действий:
    a. Поворот правой фигуры на заданный угол вокруг оси y;
    b. Смещение левой фигуры на заданное расстояние вверх от центра;
    c. Уменьшение одного примитива из правой фигуры (по выбору) в два раза.

4. Написать программу, отрисовывающую два плоских геометрических объекта, состоящих из нескольких примитивов (минимум 3-х), расположенных параллельно друг другу, относительно центра экрана. Реализовать меню, с помощью которого организовать выполнение следующих действий:
    a. Смещение заданной пользователем фигуры в заданном направлении на заданное расстояние;
    b. Поворот заданной фигуры вокруг заданной оси.

5. Написать программу, отрисовывающую два плоских геометрических объекта, состоящих из нескольких примитивов (минимум 3-х), расположенных параллельно друг другу, относительно центра экрана. Реализовать меню, с помощью которого организовать выполнение следующих действий:
    a. Поворот правой фигуры на заданный угол вокруг оси x;
    b. Смещение левой фигуры на заданное расстояние от центра;
    c. Увеличение одного примитива из правой фигуры (по выбору) в два раза.

6. Написать программу, отрисовывающую два плоских геометрических объекта, состоящих из нескольких примитивов (минимум 3-х), расположенных параллельно друг другу, относительно центра экрана. Реализовать меню, с помощью которого организовать выполнение следующих действий:
    a. Поворот левой фигуры на заданный угол вокруг оси y;
    b. Смещение правой фигуры на заданное расстояние вверх от центра;
    c. Уменьшение одного примитива из правой фигуры (по выбору) в два раза.

7. Написать программу, отрисовывающую два плоских геометрических объекта, состоящих из нескольких примитивов (минимум 3-х), расположенных параллельно друг другу, относительно центра экрана. Реализовать меню, с помощью которого организовать выполнение следующих действий:
    a. Поворот правой фигуры на заданный угол вокруг оси z;
    b. Смещение левой фигуры на заданное расстояние вниз от центра;
    c. Уменьшение одного примитива фигуры по выбору в два раза.
8. Написать программу, отрисовывающую два плоских геометрических объекта, состоящих из нескольких примитивов (минимум 3-х), расположенных параллельно друг другу, относительно центра экрана. Реализовать меню, с помощью которого организовать выполнение следующих действий:
    a. Смещение заданной пользователем фигуры в заданном направлении на заданное расстояние;

b. Поворот заданной фигуры вокруг заданной оси.

9. Написать программу, отрисовывающую два плоских геометрических объекта, состоящих из нескольких примитивов (минимум 3-х), расположенных параллельно друг другу, относительно центра экрана. Реализовать меню, с помощью которого организовать выполнение следующих действий:
    a. Поворот левой фигуры на заданный угол вокруг оси z;
    b. Смещение правой фигуры на заданное расстояние вниз от центра;
    c. Уменьшение одного примитива фигуры по выбору в два раза.

10. Написать программу, отрисовывающую два плоских геометрических объекта, состоящих из нескольких примитивов (минимум 3-х), расположенных параллельно друг другу, относительно центра экрана. Реализовать меню, с помощью которого организовать выполнение следующих действий:
    a. Поворот левой фигуры на заданный угол вокруг оси x;
    b. Смещение правой фигуры на заданное расстояние от центра;
    c. Увеличение одного примитива из правой фигуры (по выбору) в два раза.

11. Написать программу, отрисовывающую два плоских геометрических объекта, состоящих из нескольких примитивов (минимум 3-х), расположенных параллельно друг другу, относительно центра экрана. Реализовать меню, с помощью которого организовать выполнение следующих действий:

a. Поворот правой фигуры на заданный угол вокруг оси y;
b. Смещение левой фигуры на заданное расстояние вверх от центра;
c. Уменьшение одного примитива из правой фигуры (по выбору) в два раза.

12. Написать программу, отрисовывающую два плоских геометрических объекта, состоящих из нескольких примитивов (минимум 3-х), расположенных параллельно друг другу, относительно центра экрана. Реализовать меню, с помощью которого организовать выполнение следующих действий:
    a. Поворот правой фигуры на заданный угол вокруг оси z;
    b. Смещение левой фигуры на заданное расстояние вниз от центра;
    c. Уменьшение одного примитива фигуры по выбору в два раза.

13. Написать программу, отрисовывающую два плоских геометрических объекта, состоящих из нескольких примитивов (минимум 3-х), расположенных параллельно друг другу, относительно центра экрана. Реализовать меню, с помощью которого организовать выполнение следующих действий:
    a. Поворот левой фигуры на заданный угол вокруг оси z;
    b. Смещение правой фигуры на заданное расстояние вниз от центра;
    c. Увеличение одного примитива фигуры по выбору в два раза.

14. Написать программу, отрисовывающую два плоских геометрических объекта, состоящих из нескольких примитивов (минимум 3-х), расположенных параллельно друг другу, относительно центра экрана. Реализовать меню, с помощью которого организовать выполнение следующих действий:
    a. Поворот правой фигуры на заданный угол вокруг оси x;
    b. Смещение левой фигуры на заданное расстояние от центра;
    c. Увеличение одного примитива из правой фигуры (по выбору) в два раза.

15. Написать программу, отрисовывающую два плоских геометрических объекта, состоящих из нескольких примитивов (минимум 3-х), расположенных параллельно друг другу, относительно центра экрана. Реализовать меню, с помощью которого организовать выполнение следующих действий:
    a. Смещение заданной пользователем фигуры в заданном направлении на заданное расстояние;
    b. Поворот заданной фигуры вокруг заданной оси.

Содержание отчета

Результатом выполнения лабораторной работы должен стать отчет (в печатном и электронном вариантах), состоящий из следующих пунктов:

1) постановка задачи;
2) перечень функций Opengl, использованных в предлагаемом решении;
3) блок-схема алгоритма отрисовки сцены;
4) программный код предлагаемого решения;
5) скриншоты результатов работы программы.

Отчет по лабораторной работе должен быть произведен студентом преподавателю в срок до начала следующего лабораторного занятия. Во время занятия преподавателю необходимо предоставить:

- отчет по лабораторной работе (в печатном виде);
- программную реализацию решения.

Оценка (в баллах) выставляется за **устный отчет** студента по предоставленным преподавателю материалам.

Пример выполнения лабораторной работы

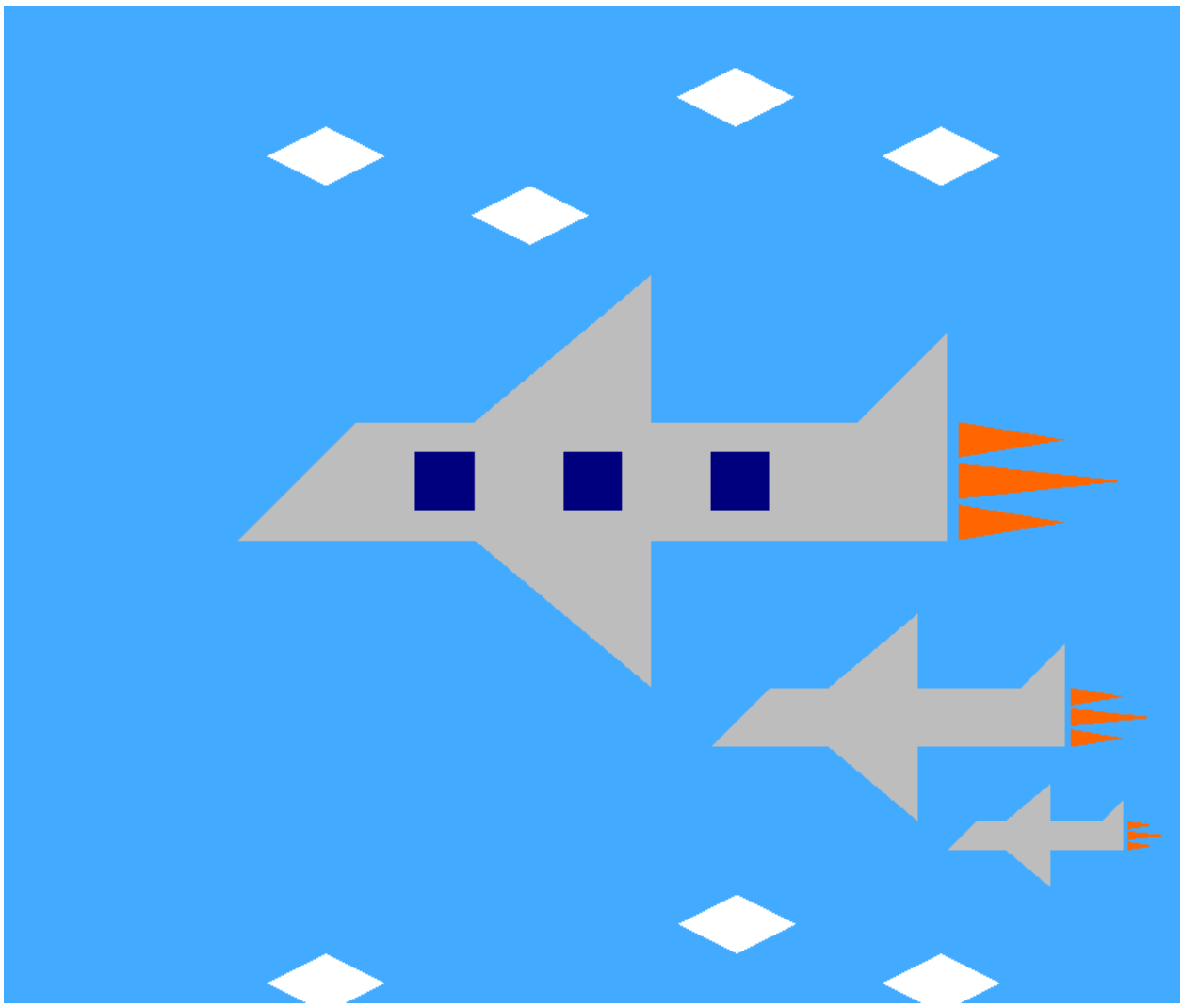
Начало – в примере к лабораторной работе №3.

```
glPushMatrix(); // клонирование самолетов
      for (int i = 0; i < 2; i++)
      {
          glScalef(0.5f, 0.5f, 0);
          glTranslatef(1, -0.8f, 0);
          glColor3f(0.74f, 0.74f, 0.74f);
          glDrawArrays(GL_TRIANGLES, 0, 3);
          glColor3f(0.74f, 0.74f, 0.74f);
          glDrawArrays(GL_QUADS, 3, 4); //основа
          glColor3f(0.74f, 0.74f, 0.74f);
          glDrawArrays(GL_TRIANGLES, 19, 3);//часть хвоста
          glColor3f(0.74f, 0.74f, 0.74f);
          glDrawArrays(GL_TRIANGLES, 22, 3);//часть хвоста
          glColor3f(0.74f, 0.74f, 0.74f);
          glDrawArrays(GL_TRIANGLES, 25, 3);//крыло
          glColor3f(0.74f, 0.74f, 0.74f);
          glDrawArrays(GL_TRIANGLES, 28, 3);//крыло2
          glColor3f(1.0f, 0.4f, 0.0f);
       glDrawArrays(GL_TRIANGLES, 35, 3);//пламя из турбин
       glDrawArrays(GL_TRIANGLES, 38, 3);//пламя из турбин2
       glDrawArrays(GL_TRIANGLES, 41, 3);//пламя из турбин3
       glColor3f(1.0f, 1.0f, 0.0f);
     };
      glPopMatrix();
      glPushMatrix(); //клонирование облаков
      static float alfa = 0;
      for (int j = 0; j < 4; j++)
      {
          if (j == 2)
          {
              glTranslatef(0.0001 + alfa, 0, 0);
              glTranslatef(0.0, +0.2f, 0);
              glColor3f(1.0f, 1.0f, 1.0f);
              glDrawArrays(GL_QUADS, 31, 4); //облако
```

```
            alfa += 0.0001;
        }
        else
        {
            glTranslatef(0.0001 + alfa, 0, 0);
            glTranslatef(0.0, -0.1f, 0);
            glColor3f(1.0f, 1.0f, 1.0f);
            glDrawArrays(GL_QUADS, 31, 4); //облако
            alfa += 0.0001;
        }
    };
    glDisableClientState(GL_VERTEX_ARRAY);
    glEnableClientState(GL_VERTEX_ARRAY);
    glPopMatrix();
    glPushMatrix();
    for (int j = 0; j < 4; j++)
    {
        if (j == 2)
        {
            glTranslatef(0.0001 + alfa, 0, 0);
            glTranslatef(0.0, +0.2f, 0);
            glColor3f(1.0f, 1.0f, 1.0f);
            glDrawArrays(GL_QUADS, 44, 4); //облако
            alfa += 0.0001;
        }
        else
        {
            glTranslatef(0.0001 + alfa, 0, 0);
            glTranslatef(0.0, -0.1f, 0);
            glColor3f(1.0f, 1.0f, 1.0f);
            glDrawArrays(GL_QUADS, 44, 4); //облако
            alfa += 0.0001;
        }
    };
    glEnd();
    glPopMatrix();
    glfwSwapBuffers(window);
}
```

В результате работы был продублирован два раза самолет. Реализовано одностороннее движение облаков(да, это облака).

Скриншот работы программы:



Контрольные вопросы

1) Почему в OpenGL используются матрицы для расчета геометрических преобразований?
2) Что такое «модельно-видовая» матрица?
3) Объясните принцип расчета переноса.
4) Объясните принцип расчета масштабирования.
5) Объясните принцип расчета поворота.
6) Зачем в процессе реализации геометрических преобразований рекомендуется использовать стек?
7) Объясните назначение функции glLoadIdentity() .
8) Какие функции OpenGL используются для обращения к матрицам геометрических преобразований?
9) Решите математически (с помощью матричных расчетов) задачу:

Стив потерялся и ему нужно найти свой дом ☹. Он знает свои координаты – (15, 4, 5). Чтобы узнать координаты своего дома, ему нужно подойти к картографу в деревне, которая находится на (10, 6, 5).

Когда Стив дошёл до деревни, он узнал, что ему нужно сделать, чтобы попасть домой – повернуться вокруг оси Z на 90 градусов, сместиться по Z на 5, повернуться по очереди на 90 градусов вокруг оси X и Z, сместиться по X на -3 и У на 10 одновременно.

10) Решите математически (с помощью матричных расчетов) задачу:

У машины отвалилось колесо и укатилось. Координата оси, на которую нужно прикрепить колесо [8, 4, 3]. Колесо упало на координате [1,1,1], так еще и завалилось на бок. Нужно быстрей крепить колесо, пока зомби не успели догнать нас!

ЛАБОРАТОРНАЯ РАБОТА №5

OpenGL: Исследование и реализация методов визуализации трехмерной сцены

Цель работы

Изучение методов создания трехмерных объектов средствами OpenGL. Исследование свойств плоских проекций трехмерных объектов.

Теоретическая информация

Проективные преобразования в OpenGL

Основная проблема 3D – это конвертация 3-мерных объектов в соответствующие пиксели на 2-мерную плоскость экрана. При этом производятся следующие основные операции:

1. Проекционные преобразования – ограничение области видимости объемной сцены. Для этого используется матрица проекций. Результатом этого преобразования является преобразование координат вершин видимой области сцены для каждой из осей координат таким образом, чтобы они находились в интервале от –1 до 1.

2. Клиппинг – т.е. отсекание тех объектов или их частей, которые не попадают в прямоугольник, ограниченный рамками монитора.

3. Растеризация, т.е. приведение 2D-координат в соответствие с оконными координатами устройства. Выполняется преобразование мировой (правосторонней) системы координат в видовую (левостороннюю) систему координат.

В составе OpenGL имеются две функции для задания перспективных проекций и одна для задания параллельных проекций. Каждая из функций определяет зону видимости – пирамиду или параллелепипед. Объекты, не попадающие в эту зону, отсекаются и не включаются в отображаемую сцену.

Перспективные преобразования в Opengl

Параметры пирамиды видимости задаются функцией glFrustum(), смысл аргументов которой поясняет рисунок 2.

void ***glFrustum*** (GLdouble *left*, GLdouble *right*, GLdouble *bottom*, GLdouble *top*, GLdouble z*near*, GLdouble z*far*)

Функция **glFrustum** умножает текущую матрицу на матрицу перспективы. Первые четыре аргумента задают координаты для обрезки плоскости отсечения с четырех сторон. Значения аргументов near и far, задающих положение передней и задней отсекающих плоскостей, должны быть положительными и отсчитываться от центра проецирования вдоль оси проецирования.

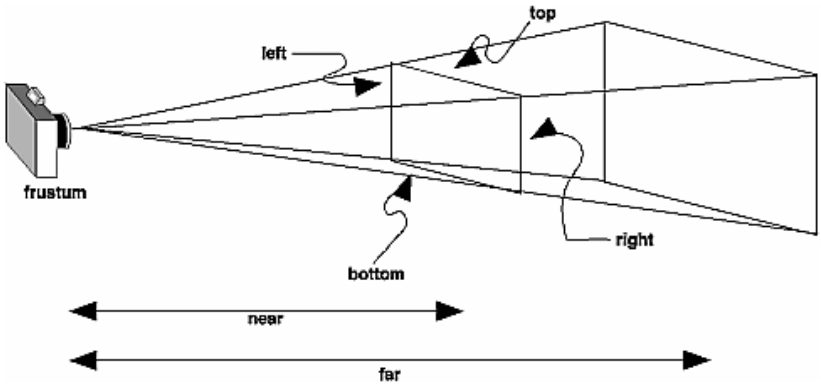


Рисунок 2. Пирамида видимости при расчете перспективных преобразований

Поскольку матрица проецирования умножается на текущую матрицу, сначала нужно задать режим работы с этой матрицей. Типичная последовательность операций представлена ниже.

```
glMatrixMode (GL_PROJECTION);
glLoadIdentity ();
glFrustum (xmin, xmax, ymin, ymax, near, far);
```

Проекционная матрица показана на рисунке 3.

$$\begin{pmatrix} \frac{2\,near}{right-left} & 0 & A & 0 \\ 0 & \frac{2\,near}{top-bottom} & B & 0 \\ 0 & 0 & C & D \\ 0 & 0 & -1 & 0 \end{pmatrix}$$

$$A = \frac{right+left}{right-left} \qquad C = -\frac{far+near}{far-near}$$

$$B = \frac{top+bottom}{top-bottom} \qquad D = -\frac{2\,far\,near}{far-near}$$

Рисунок 3 Проекционная матрица и расчет перспективных преобразований

Параметры (left, bottom, near) и (right, top, near) определяют точки на "ближней" плоскости отсечения, которые проецируются на нижний левый и правый верхний углы окна соответственно, предполагается, что "глаз" (точка, откуда смотрят) расположена в точке с координатами (0,0,0). Параметр far определяет положение "дальней" плоскости отсечения. Точность буфера глубины зависит от значений, указанных для near и far. Чем больше соотношение far к near, тем менее эффективным буфер глубины будет различать поверхности, расположенные рядом друг с другом.

Еще раз напомним, что параметры far и near должны всегда быть положительными.

Можно так же использовать **glPushMatrix** и **glPopMatrix** для сохранения и восстановления текущего стека матриц.

Во многих приложениях предпочтительнее задавать не линейные параметры, характеризующие положение углов усеченной пирамиды видимости, а угол и поле зрения. Однако если картинная плоскость является прямоугольником, а не квадратом, то нужно задавать пару углов зрения: один в вертикальной плоскости, другой – в горизонтальной (рис. 4).

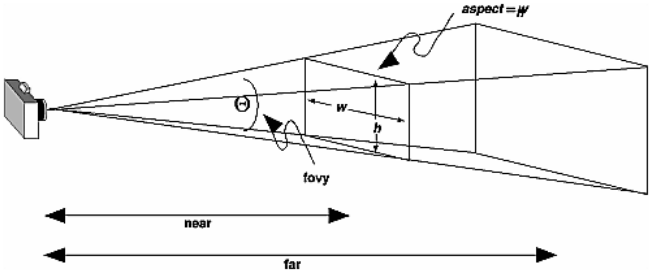


Рисунок 4. Пирамида видимости при расчете перспективных преобразований gluPerspective

Для задания вышеуказанных параметров используется функция:

void ***gluPerspective*** (GLdouble *fovy*, GLdouble *aspect*, GLdouble *near*, GLdouble *far*);

Аргументы этой функции имеют следующий смысл:

- fovy – угол зрения в вертикальной плоскости;
- aspect – отношение ширины окна картинной плоскости к его высоте;
- near и far – расстояние от центра проецирования до передней и задней отсекающих плоскостей.

Параллельное проецирование в Opengl

В составе OpenGL имеется только одна функция для задания параметров параллельного проецирования, которая формирует ортогональную проекцию. Зона видимости при этом превращается в параллелепипед (рис. 5).

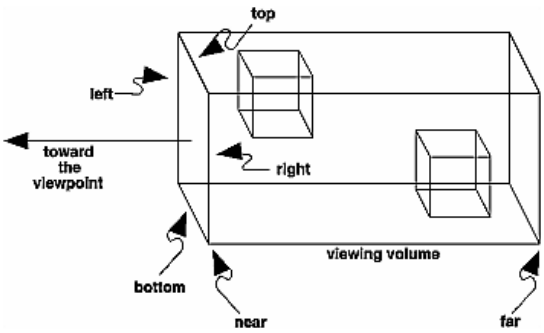


Рисунок 5. Параллелепипед видимости при ортогональном проецировании

Общая форма этой функции:

void ***glOrtho*** (GLdouble *left*, GLdouble *right*, GLdouble *bottom*, GLdouble *top*, GLdouble *near*, GLdouble *far*);

Аргументы вызова имеют тот же геометрический смысл, что и одноименные аргументы функции glFrustum().

Задание положения и ориентации камеры

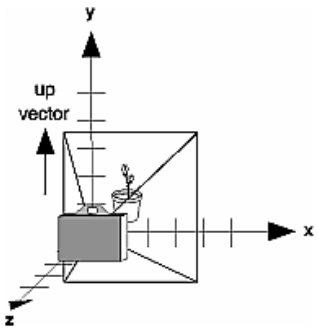


Рис. 6

В составе OpenGL имеется функция gluLookAt(), которая позволяет задать положение и ориентацию камеры (рис. 6).

void ***gluLookAt*** (GLdouble *eyex*, GLdouble *eyey*, GLdouble *eyez*, GLdouble *centerx*, GLdouble *centery*, GLdouble *centerz*, GLdouble *upx*, GLdouble *upy*, GLdouble *upz*)

Аргументы функции имеют следующий вид:

- eyex, eyey, eyez – координаты точки наблюдения;
- centerx, centery, centerz – координаты контрольной точки объекта, указывающей центр сцены;
- upx, upy, upz- компоненты точки, которая задает положительное направления оси Y сцены.

Установка области видимости

Viewport – это область окна, в которой будет отображаться результаты нашей работы. Для установки области видимости надо использовать функцию OpenGL

glViewport (GLint *x*, GLint *y*, GLsizei *width*, GLsizei *height*)

Назначение параметров: x и y – координаты левого верхнего угла области видимости, width и height – размеры. По умолчанию, OpenGL устанавливает область видимости равную размерам окна приложения в момент инициализации. Наиболее часто эту команду используют в обработчике изменения размеров окна.

Эта функция определяет, каким образом будут преобразовываться реальные x и y координаты в оконные координаты. Преобразования происходят следующим образом:

$$x_w = (x_{nd} + 1)\left(\frac{width}{2}\right) + x$$

$$y_w = (y_{nd} + 1)\left(\frac{height}{2}\right) + y$$

где x_w и y_w – конечные оконные координаты, x_{nd}, y_{nd} – реальные координаты, x, y, width, height – задаются в функции glViewport().

Мы также можем установить, каким образом проецируется z координата из реальных координат в оконные при помощи функции

glDepthRange (GLclampd *near*, GLclampd *zar*)

Параметр znear проецирует "ближнюю" плоскость отсечения в оконные координаты, zfar – проецирует "дальнюю" плоскость в оконные координаты. После отсечения z координата находится в диапазоне от -1.0 до 1.0 соответствующие "ближней" и "дальней" плоскостям отсечения. Функция glDepthRange() описывает, каким образом координаты в этом диапазоне преобразуются в оконные. По умолчанию параметры znear и zfar равны 0 и

1 соответственно, что позволяет полностью использовать весь диапазон значений буфера глубины.

Буфер глубины

Буфер глубины (Z-buffer, depth buffer) – двумерный массив данных, дополняющий двумерное изображение (буфер кадра), где каждому пикселю (фрагменту) изображения ставится в соответствии «глубина», т.е. расстояние от наблюдателя до соответствующей точки поверхности отрисовываемого объекта. Если пиксели двух рисуемых объектов перекрываются, то их значения глубины сравниваются и рисуется тот, который ближе, а его значение глубины сохраняется в Z-буфере. Поддержка буфера осуществляется на аппаратном уровне. Для работы с буфером используются следующие команды:

- glutDisplayMode (GLUT_DEPTH) // создание
- glClear (GL_DEPTH_BUFFER_BIT) // очистка
- glEnable (GL_DEPTH_TEST) // включение
- glDisable (GL_DEPTH_TEST) // выключение

Пример: Рисование трехмерного куба

Куб следует рассматривать как шесть многоугольников, которые определяют его грани. Массив вершин куба может быть представлен в следующем виде:

```
GLfloat vertices[][3]={{-1.0,-1.0,-1.0},{1.0,-1.0,-1.0},
{1.0,1.0,-1.0},{-1.0,1.0,-1.0},{-1.0,-1.0,1.0},
{1.0,-1.0,1.0},{1.0,1.0,1.0},{-1.0,1.0,1.0}};
```

Для определения граней куба можно использовать список точек – элементов массива вершин. Например, одна грань куба в тексте программы определяется следующим образом:

```
glBegin(GL_POLYGON);
     glVertex3fv(vertices[0]);
```

```
        glVertex3fv(vertices[3]);
        glVertex3fv(vertices[2]);
        glVertex3fv(vertices[1]);
    glEnd();
```

Другие пять граней определяются аналогично. При определении трехмерных многогранников порядок перечисления вершин имеет большое значение. Следует учитывать, что многоугольник имеет две стороны – внутреннюю и внешнюю. Будем называть грань внешней, если при взгляде с внешней стороны объекта на эту грань ее вершины «обходятся» против часовой стрелки. Этот метод известен как «правило правой руки», поскольку, если расположить четыре согнутых пальца правой руки вдоль направления обхода контура, большой палец будет указывать наружную сторону грани.

Список вершин можно использовать и для хранения информации, необходимой для раскрашивания куба. С вершинами в данном примере будут ассоциироваться чистые цвета вершин цветового куба (черный, белый, красный, зеленый, синий, голубой, фиолетовый, желтый):

```
GLfloat colors[][3]={{0.0,0.0,0.0},{1.0,0.0,0.0},
{1.0,1.0,0.0},{0.0,1.0,0.0},{0.0,0.0,1.0},
{1.0,0.0,1.0},{1.0,1.0,1.0},{0.0,1.0,1.0}};
```

Для управления режимом интерполяции цветов используется команда void glShadeModel(GLenummode), вызов которой с параметром GL_SMOOTH включает интерполяцию (установка по умолчанию), а с GL_FLAT отключает.

Функция quad() вычерчивает четырехугольник, заданный точками в списке вершин, а функция colorcube() задает шесть граней таким образом, чтобы все они были внешними.

```
GLfloat vertices[][3]={{-1.0,-1.0,-1.0},{1.0,-1.0,-1.0},
{1.0,1.0,-1.0},{-1.0,1.0,-1.0},{-1.0,-1.0,1.0},
{1.0,-1.0,1.0},{1.0,1.0,1.0},{-1.0,1.0,1.0}};
GLfloat colors[][3]={{0.0,0.0,0.0},{1.0,0.0,0.0},
```

```
{1.0,1.0,0.0},{0.0,1.0,0.0},{0.0,0.0,1.0},
{1.0,0.0,1.0},{1.0,1.0,1.0},{0.0,1.0,1.0}};
void polygon(int a, int b, int c, int d)
{
     glBegin(GL_POLYGON);
     glColor3fv(colors[a]);
     glVertex3fv(vertices[a]);
     glColor3fv(colors[b]);
     glVertex3fv(vertices[b]);
     glColor3fv(colors[c]);
     glVertex3fv(vertices[c]);
     glColor3fv(colors[d]);
     glVertex3fv(vertices[d]);
     glEnd();
}
void colorcube()
{
     polygon(0,3,2,1);
     polygon(2,3,7,6);
     polygon(0,4,7,3);
     polygon(1,2,6,5);
     polygon(4,5,6,7);
     polygon(0,1,5,4);
}
```

Методика выполнения лабораторной работы

В течение занятия необходимо:

1) изучить теоретическую информацию, предлагаемую на лекции, в данном пособии и в методических указаниях;
2) сформировать модель решения с учетом использования преобразований модельно-видовой матрицы и матрицы проекций, согласно варианту задания;
3) реализовать программное решение, согласно варианту задания;
4) получить перспективную и параллельную проекцию объекта;
5) организовать перемещение камеры вокруг трехмерного объекта, изменяя координаты точки наблюдения – eyex, eyey, eyez. Для перемещения камеры использовать клавиатуру.

Варианты заданий

Отрисовать 3D-объект:

1) одноэтажный дом с окном, трубой, дверью и ступеньками перед этой дверью; дверь открывается; в окне «загорается» свет (использовать работу с цветом);
2) трехэтажный дом с трубой, окнами и балконами (не менее 3); трубу и балконы уносит «ураган»;
3) классический стул, которому можно по клику «мыши» менять форму ножек и спинки;
4) песочница с «зонтиком»; по клику мыши можно изменить форму песочницы;
5) лавочка и стопка книг на ней; по клику «мыши» книги перемещаются: либо падают, либо передвигаются по сиденью;
6) ноутбук, у которого можно поднять крышку;
7) грузовик, у которого поднимается кузов;
8) грузовик, у которого вращаются колеса;
9) легковой автомобиль с фарами, которые меняют цвет;
10) стол с настольной лампой, которую можно перемещать по столу;
11) системный блок компьютера, в котором можно заменить детали (не менее двух);
12) дом из кубиков-кирпичей с окнами и дверью; реализовать анимацию сборки дома;
13) светофор, который переключает цвета и ломается по клику «мыши»;
14) стол и фоторамка из объемных элементов (не менее трех), которая собирается по клику «мыши»;
15) пульт от телевизора с имитацией нажатия клавиш (не менее двух);

Содержание отчета

Результатом выполнения лабораторной работы должен стать отчет (в печатном и электронном вариантах), состоящий из следующих пунктов:

1) постановка задачи;
2) перечень функций Opengl, использованных в предлагаемом решении;
3) блок-схема алгоритма отрисовки сцены;
4) программный код предлагаемого решения;
5) скриншоты результатов работы программы.

Отчет лабораторной работы должен быть произведен студентом преподавателю в срок до начала следующего лабораторного занятия. Во время занятия преподавателю необходимо предоставить:

- отчет о лабораторной работе (в печатном виде);
- программную реализацию решения.

Оценка (в баллах) выставляется за **устный отчет** студента по предоставленным преподавателю материалам.

Контрольные вопросы

1) Почему в OpenGL используются матрицы для расчета геометрических преобразований?
2) Что такое матрица проекций?
3) Объясните принцип расчета параллельной проекции.
4) Объясните принцип расчета перспективной проекции.
5) Объясните понятие «объем видимости».
6) Зачем в процессе реализации геометрических преобразований рекомендуется использовать стек?
7) Объясните последовательность действий при построении трехмерных сцен.
8) Какие функции OpenGL используются для управления камерой наблюдения?
9) Объясните принцип работы буфера глубины.
10) Объясните возможности OpenGL для установки области видимости.

СПИСОК ЛИТЕРАТУРЫ

1) Тарасов И. Opengl в России. – Режим доступа : http://www.helloworld.ru/texts/comp/games/opengl/opengl2/index.html; (скачать : http://www.read.in.ua/book120840/?razdel=11&p=47).

2) OpenGL. – Режим доступа : https://www.opengl.org.

3) Верма Р. Д. Введение в Opengl . – Москва : Горячая линия – Телеком, 2015. – 303 с.

4) Баяковский Ю.М., Игнатенко А.В., Фролов А.И. Графическая библиотека OpenGL.: Учебно-методическое пособие. – Москва : ВМиК МГУ, 2003. – 132 с.

5) Гайдуков С. Профессиональное программирование трехмерной графики на C++. – Санкт-Петербург : БХВ-Петербург, 2004. – 736 с.

6) Боресков А. Графика трехмерной компьютерной игры на основе OpenGL. – Москва : Диалог-МИФИ, 2005.

7) Херн Д., Бейкер М. Компьютерная графика и стандарт OpenGL. – Москва : ИД «Вильямс», 2005.

8) Программирование компьютерной графики средствами OpenG : Документация, статьи, советы. – Режим доступа : opengl.gamedev.ru.

9) Основы реализации перемещения в пространстве с использованием библиотеки GLUT – Режим доступа: https://gamedev.ru/articles/free_movement

10) Линейная динамика твердого тела с OpenGL – Режим доступа: https://translated.turbopages.org/proxy_u/en-ru.ru.6f139509-641f1602-6b77a00d-74722d776562/https/www.codeproject.com/Articles/626336/Linear-Rigid-Body-Dynamics-with-OpenGL

Оглавление

Электронное учебное издание

Оксана Федоровна **Абрамова**

Компьютерная графика: лабораторный практикум Часть 2

Учебное-методическое пособие

Электронное издание сетевого распространения

Редактор Матвеева Н.И.

Темплан 2023 г. Поз. № 5.
Подписано к использованию 07.06.2023. Формат 60х84 1/16.
Гарнитура Times. Усл. печ. л. 4,0.

Волгоградский государственный технический университет.
400005, г. Волгоград, пр. Ленина, 28, корп. 1.

ВПИ (филиал) ВолгГТУ.
404121, г. Волжский, ул. Энгельса, 42а.