ПРАКТИЧЕСКОЕ ЗАНЯТИЕ №3

РАБОТА С МАТРИЦАМИ И ПРЕОБРАЗОВАНИЯМИ В OPENGL

Цели: формирование практических навыков по работе с матричными преобразованиями для изменения сцены в целом и отдельных еѐ объектов, закрепление знаний о способах проецирования, изучения методов работы со стеком матриц средствами OpenGL, создание ряда многообъектных сцен и их преообразование с использованием переноса, поворота и масштабирования.

Задачи: сформировать понимание преобразований наблюдения, модели и проектирования, познакомиться с концепцией матриц преобразования, выяснить назначение единичной матрицы, научиться создавать приложения OpenGL с использованием матриц преобразования и ранее изученных объектов.

Выполнение:

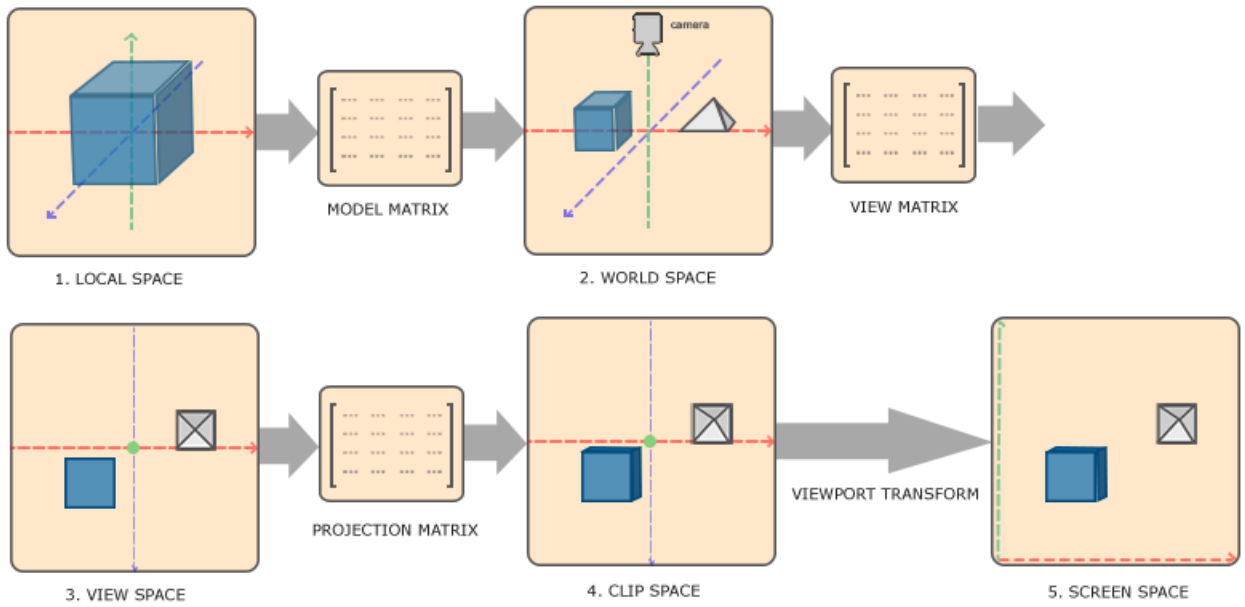
1. Ознакомиться с теоретическим материалом
2. Выполнить основные задания
3. Предоставить отчет, по каждому заданию содержащий: формулировку задания, исходный код программы, скриншоты работающей программы (один или несколько, если необходимо).
4. Ответить на вопросы преподавателя

Возможность располагать объекты на сцене и выбирать их ориентацию является необходимым инструментом для тех, кто занимается программированием трехмерной графики. Размеры объекта удобнее описывать относительно начала координат, а затем перенести объекты в нужное положение.

Преобразование координат в нормализованные, а затем в экранные координаты обычно осуществляется пошагово, и, до окончательного преобразования в экранные координаты, мы переводим вершины объекта в несколько координатных систем. Преимущество преобразования координат через несколько *промежуточных* координатных систем заключается в том, что некоторые операции/вычисления проще выполняются в определённых системах, и это скоро станет очевидно. Всего есть 5 различных координатных систем, которые для нас важны:

Для преобразования координат из одного пространства в другое, мы будем использовать несколько матриц трансформации, среди которых, самыми важными являются матрицы **Модели**, **Вида** и **Проекции**. Координаты наших вершин начинаются в **локальном пространстве** как **локальные координаты**, и в дальнейшем преобразуются в **мировые координаты**, потом в **координаты вида**, **отсечения**, и,

наконец, все заканчивается **экранными координатами**. Следующее изображение показывает эту последовательность, и то, что делает каждое преобразование:



1. Локальные координаты это координаты вашего объекта измеряемые относительно точки отсчета расположенной там, где начинается сам объект

2. На следующем шаге локальные координаты преобразуются в координаты мирового пространства, которые по смыслу являются координатами более крупного мира. Эти координаты измеряются относительно глобальной точки отсчёта, единой для всех других объектов расположенных в мировом пространстве

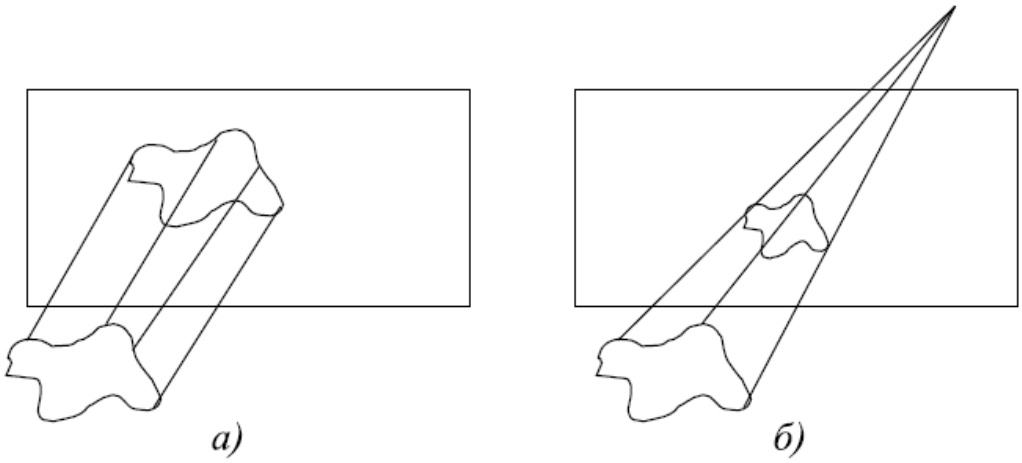
3. Дальше мы трансформируем мировые координаты в координаты пространства Вида таким образом, что каждая вершина становится видна как если бы на неё смотрели из камеры или с точки зрения наблюдателя

4. После того, как координаты были преобразованы в пространство Вида, мы хотим спроецировать их в координаты Отсечения. Координаты Отсечения являются действительными в диапазоне от -1.0 до 1.0 и определяют, какие вершины появятся на экране.

5. И, наконец, в процессе преобразования, который мы назовем **трансформацией области просмотра**, мы преобразуем координаты отсечения от -1.0 до 1.0 в область экранных координат, заданную функцией **glViewport**.

После всего этого, полученные координаты отсылаются растеризатору для превращения их во фрагменты.

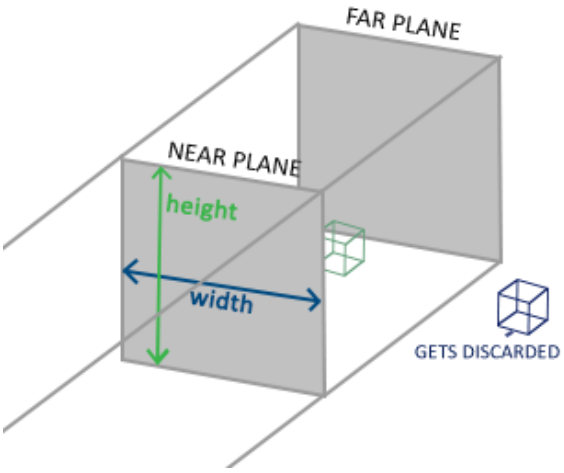
В компьютерной графике наиболее распространены параллельная (рис. *а*) и перспективная (рис. *б*) проекции.



ОРТОГРАФИЧЕСКАЯ ПРОЕКЦИЯ

Ортографической проекцией *называется проекция, в которой картинная плоскость совпадает с одной из координатных плоскостей или параллельна ей.*

Матрица ортографической проекции задает усечённую пирамиду в виде параллелограмма, который является пространством отсечения, где все вершины, находящиеся вне его объема отсекаются. При создании матрицы ортографической проекции мы задаем ширину, высоту и длину видимой пирамиды отсечения. Все координаты, которые после их преобразования матрицей проекции в пространство отсечения попадают в ограниченный пирамидой объем отсечены не будут. Усеченная пирамида выглядит немного похожей на контейнер:



Усеченная пирамида определяет область видимых координат и задается шириной, высотой, **ближней** и **дальней** плоскостями. Любая координата, расположенная перед ближней плоскостью, отсекается, точно также поступают и с координатами, находящимися за задней плоскостью. Ортографическая усеченная пирамида напрямую переводит попадающие в неё координаты в нормализованные координаты устройства, и w-компоненты векторов не используются; если w-компонент равен 1.0, то перспективное деление не изменит значений координат.

Для создания матрицы ортографической проекции мы используем встроенную функцию **ortho**. Она же используется по умолчанию с параметрами координат 1 и -1. В общем же случае:

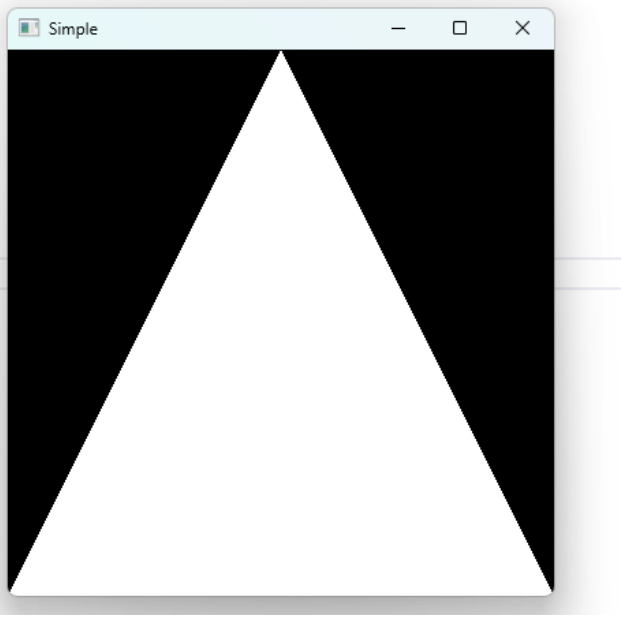
void glOrtho (GLdouble left, GLdouble right, GLdouble bottom, GLdouble top, GLdouble near, GLdouble far).

Первые два параметра определяют левую и правую координаты усеченной пирамиды, а третий и четвертый параметры задают нижнюю и верхнюю границы пирамиды. Эти четыре точки устанавливают размеры ближней и дальней плоскостей, а 5-й и 6-й параметры указывают расстояние между ними. Эта особая матрица проекции преобразует все координаты попадающие в диапазоны значений x, y и z, в нормализованные координаты устройства.

До этого занятия мы работали на плоскости, теперь перейдем в пространство. Рассмотрим основные принципы на простом примере треугольника.

Листинг 1.

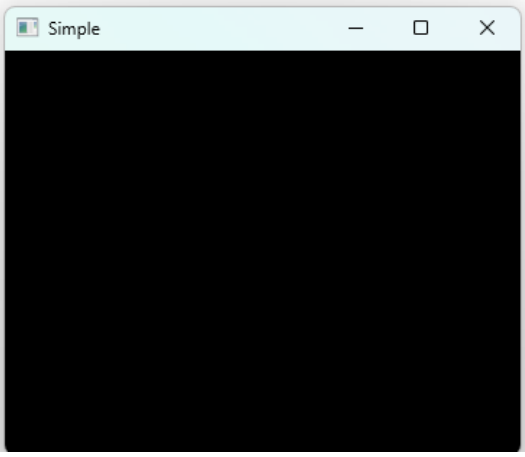
```
#include "glew.h"
#include "glut.h"


void RenderScene(void)
{
    glClear(GL_COLOR_BUFFER_BIT);
    glColor3ub(255, 255,255);
    glBegin(GL_TRIANGLES);
    glVertex3f(-1.f, -1.f, 0.f);
    glVertex3f(0.f, 1.0f, 0.f);
    glVertex3f(1.f, -1.f, 0.f);
    glEnd();
    glFlush();
}

void main(void)
{
    glutInitDisplayMode(GLUT_SINGLE);
```



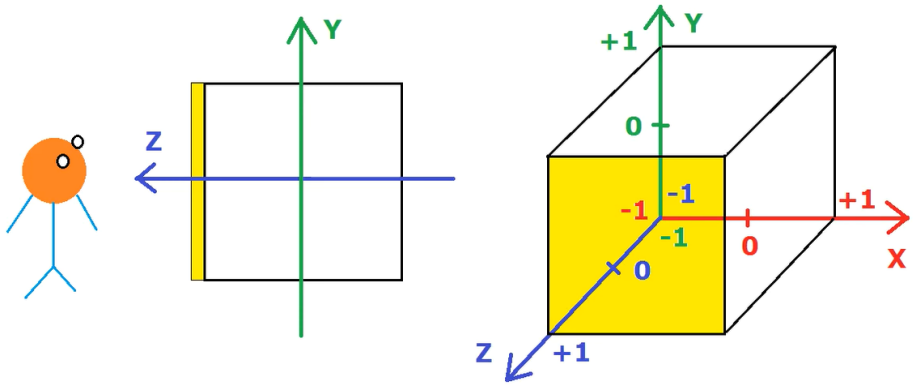
При заданных координатах треугольник строится в плоскости XOY. Теперь изменим координату z на 1. Пока ничего нового не произошло, мы все еще наблюдаем этот треугольник. А теперь на z=2.

Листинг 2.

```
void RenderScene(void)
{
    glClear(GL_COLOR_BUFFER_BIT);
    glColor3ub(255, 255,255);
    glBegin(GL_TRIANGLES);
    glVertex3f(-1.f, -1.f, 2.f);
    glVertex3f(0.f, 1.0f, 2.f);
    glVertex3f(1.f, -1.f, 2.f);
    glEnd();
    glFlush();
}
```

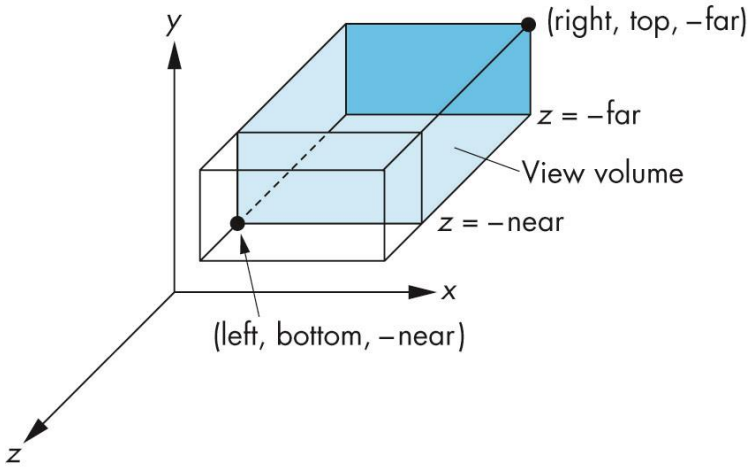


Теперь треугольник находится **вне** поля видимости. Почему так происходит? Изначально камера находится в начале координат и направлена вдоль отрицательного направления оси Oz, а все пространство находится между -1 и 1 трех осей.



В OpenGL рассматривает трехмерное пространство в правосторонней системе координат. В этой системе координат направление «вперед» наблюдателя откладывается в отрицательном направлении оси z. Однако при вызове glOrtho мы указываем не значения на оси z (которые должны были бы быть отрицательными), а «глубину» - расстояние до ближней и дальней плоскости, а это расстояние положительно. Положительные значения глубины для ближней и дальней плоскости соответствуют отрицательному диапазону координат в мировом пространстве.

Таким образом, треугольник будет виден только внутри указанного диапазона [-1;-100].

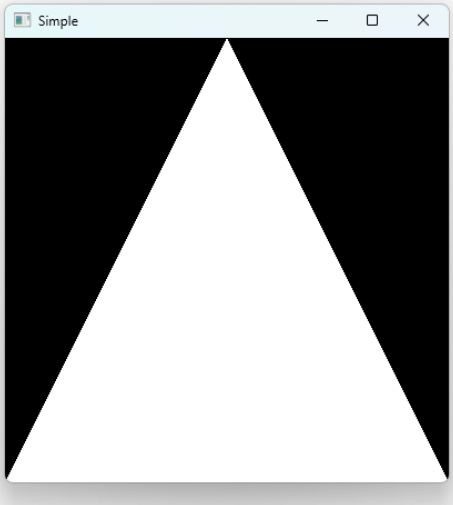


Ортографическая матрица проекции отображает координаты непосредственно на двумерную плоскость, которой является ваш дисплей, но в действительности, прямое проецирование дает нереалистичные результаты, потому что не принимает в расчет **перспективы**. Это удобно, например, в автоматизированном проектировании, представлении такой двухмерной графики, как текст, или в архитектурных рисунках, где желательно представить точные размеры.

Давайте вернемся к нашему треугольнику. Он будет виден для всех z из [-1;100] при этом его размеры меняться не будут.

Листинг 3.

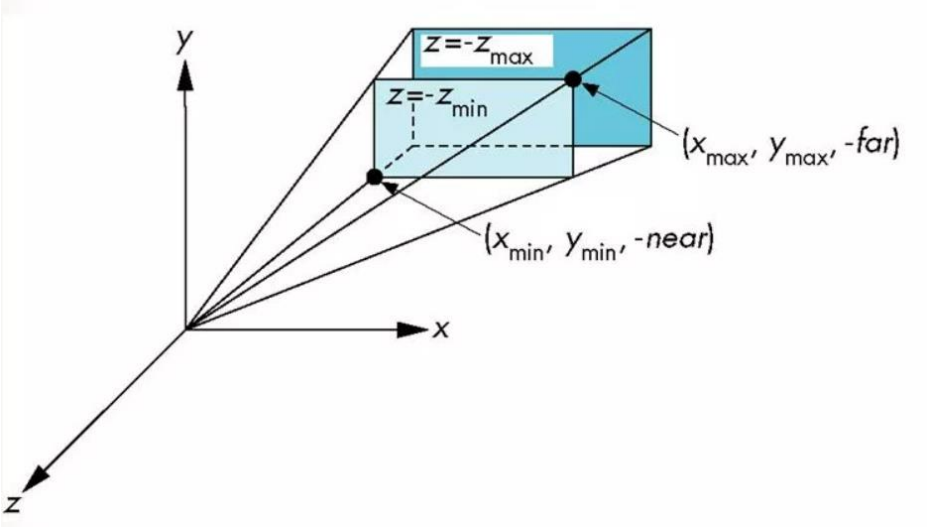
```
void RenderScene(void)
{
    glClear(GL_COLOR_BUFFER_BIT);
    glColor3ub(255, 255,255);
    glBegin(GL_TRIANGLES);
    glVertex3f(-100.f, -100.f, -2.f);
    glVertex3f(0.f, 100.0f, -2.f);
    glVertex3f(100.f, -100.f, -2.f);
    glEnd();
    glFlush();
}

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



ПЕРСПЕКТИВНАЯ ПРОЕКЦИЯ

Теперь рассмотрим ***Перспективную (центральную) проекцию*** — *вид проекции, где лучи проектирования исходят из одного центра (центра проектирования), размещенного на конечном расстоянии от объектов и плоскости проектирования.*

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

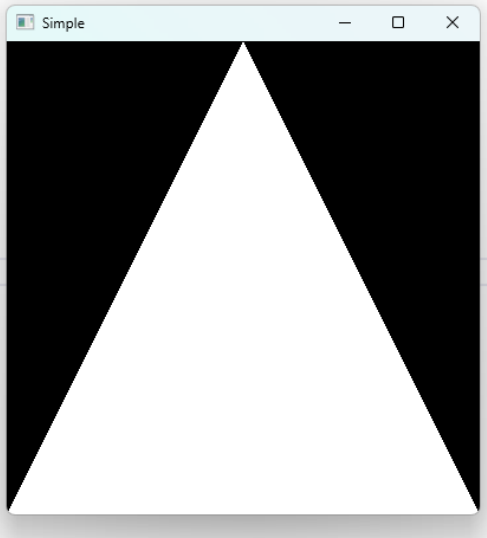


Данный вид проектирования позволяет получить наиболее реалистичные изображения трехмерных объектов из-за перспективных искажений сцены. Перспективная проекция получается путем перспективного преобразования и проецирования на некоторую плоскость наблюдения. Перспективные проекции параллельных прямых, не параллельных плоскости проекции, будут сходиться в точке схода.

Листинг 4

Заменим `glOrtho` на `glFrustum` и поэкспериментируем с координатой z. Пока ничего нового, размеры треугольника те же.

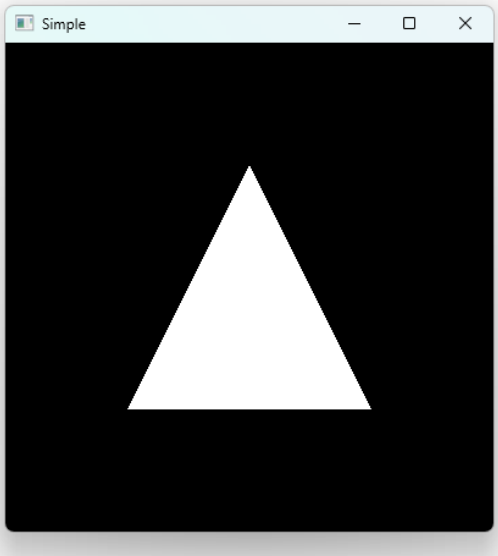
```
void RenderScene(void)
{
    glClear(GL_COLOR_BUFFER_BIT);
    glColor3ub(255, 255,255);
    glBegin(GL_TRIANGLES);
    glVertex3f(-100.f, -100.f, -1.f);
    glVertex3f(0.f, 100.0f, -1.f);
    glVertex3f(100.f, -100.f, -1.f);
    glEnd();
    glFlush();
}

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



Следующий пример демонстрирует треугольник отдаленный от зрителя в два раза.

Листинг 5.

```
void RenderScene(void)
{
    glClear(GL_COLOR_BUFFER_BIT);
    glColor3ub(255, 255,255);
    glBegin(GL_TRIANGLES);
    glVertex3f(-100.f, -100.f, -2.f);
    glVertex3f(0.f, 100.0f, -2.f);
    glVertex3f(100.f, -100.f, -2.f);
    glEnd();
    glFlush();
}

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



Изменяя left, right, bottom и top мы изменяем площадь ближнего прямоугольника, от которого зависит площадь дальнего, а следовательно и объем видимого пространства. Стоит добавить только одно замечание, расстояние до ближней плоскости должно быть строго положительным и не равняться нулю.

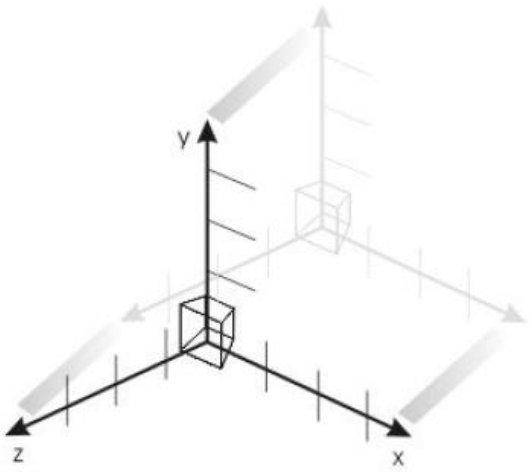
МОДЕЛЬНЫЕ ТРАНСФОРМАЦИИ

Существует три команды OpenGL для модельных преобразований: **glTranslate*()**, **glRotate*()** и **glScale*()**. Как вы можете предположить, эти команды трансформируют объект (или координатную систему – в зависимости от того, как вы предпочитаете думать об этом), перенося, поворачивая, увеличивая, уменьшая или отражая его. Все эти команды эквивалентны созданию соответствующей матрицы переноса, поворота или масштабирования.

Перенос

void **glTranslate**{fd} (TYPE *x*, TYPE *y*, TYPE *z*);
Умножает текущую матрицу на матрицу, передвигающую (переносящую) объект на расстояния *x*, *y*, *z*, переданные в качестве аргументов команды, по соответствующим осям (или перемещает локальную координатную систему на те же расстояния).

На рисунке изображен эффект команды **glTranslate*()**.

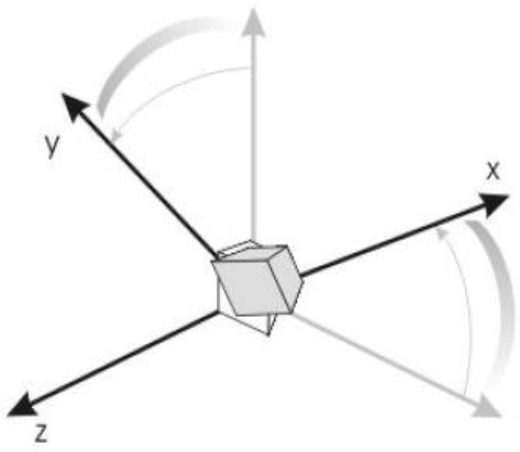


Обратите внимание на то, что использование (*0.0*, *0.0*, *0.0*) в качестве аргумента **glTranslate*()** – это единичная операция, то есть она не влияет на объект или на его координатную систему.

Поворот

void **glRotate**{fd} (TYPE *angle*, TYPE *x*, TYPE *y*, TYPE *z*);

Умножает текущую матрицу на матрицу, которая поворачивает объект (или локальную координатную систему) в направлении против часовой стрелки вокруг луча из начала координат, проходящего через точку (*x*, *y*, *z*). Параметр *angle*задает угол поворота в градусах. Результат выполнения **glRotatef**(*45.0*, *0.0*,*0.0*,*1.0*), то есть поворот на 45 градусов вокруг оси *z*, показан на рисунке.



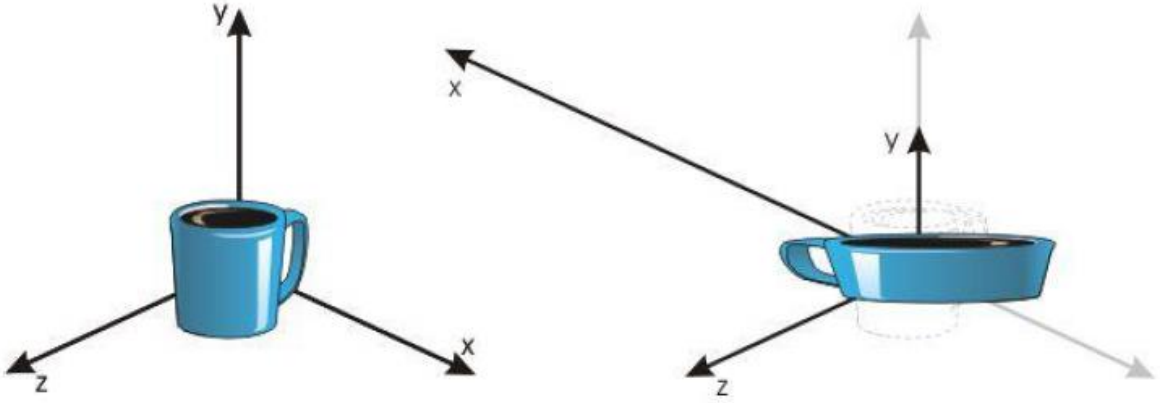
Заметьте, что чем дальше объект от оси вращения, тем больше орбита его поворота и тем заметнее сам поворот. Также обратите внимание на то, что вызов **glRotate*()** с параметром *angle* равным 0 не имеет никакого эффекта.

Масштабирование

void **glScale**{fd} (TYPE *x*, TYPE *y*, TYPE *z*);

Умножает текущую матрицу на матрицу, которая растягивает, сжимает или отражает объект вдоль координатный осей. Каждая *x* -, *y*- и *z*- координата каждой точки объекта будет умножена на соответствующий аргумент *x*, *y*или *z*команды **glScale*()**. При рассмотрении преобразования с точки зрения локальной координатной системы, оси этой системы растягиваются, сжимаются или отражаются с учетом факторов *x*, *y* и *z*, и ассоциированный с этот системой объект меняется вместе с ней.

На рисунке показан эффект команды **glScalef***(-2.0*,*0.5*,*1.0*);

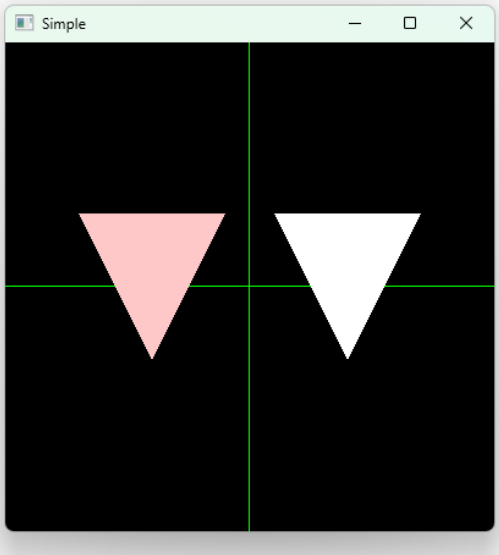


glScale*() – это единственная из трех команд модельных преобразований, изменяющая размер объекта: масштабирование с величинами более *1.0* растягивает объект, использование величин меньше *1.0* сжимает его. Масштабирование с величиной *-1.0* отражает объект относительно оси или осей. Единичными аргументами (то есть аргументами, не имеющими эффекта) являются (*1.0*, *1.0*, *1.0*). Вообще следует ограничивать использование **glScale*()** теми случаями, когда это действительно необходимо. Использование **glScale*()** снижает быстродействие расчетов освещенности, так как вектора нормалей должны быть нормализованы заново после преобразования.

Пример как порядок сдвига и поворота влияют на конечный результат изображения объекта.

Листинг 6.

```
void RenderScene(void)
{


    glClear(GL_COLOR_BUFFER_BIT);
        //Задаем оси
    glBegin(GL_LINES);
    glColor3f(0.0, 1.0, 0.0);
    glVertex3f(-1.0, 0.0, 0.0);
    glVertex3f(1.0, 0.0, 0);
    glVertex3f(0.0, -1, 0.0);
    glVertex3f(0.0, 1, 0);
    glEnd();

    // Обновляет стек матрицы проектирования
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();

    //задается перспективная система координат
    glFrustum(-100.0, 100.0, -100.0, 100.0, 1.0, 100.0);

    //Обновляем матрицу наблюдения модели
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();

    //создаем перемещение первого треугольника
    glPushMatrix();
    //сдвиг по оси X
    glTranslatef(40, 0, 0);
    //поворот
    glRotatef(180, 0, 0, 1);

    glColor3ub(255, 255,255);
    glBegin(GL_TRIANGLES);
    glVertex3f(-30.f, -30.f, -1.f);
    glVertex3f(0.f, 30.0f, -1.f);
    glVertex3f(30.f, -30.f, -1.f);
    glEnd();
    glPopMatrix();
    //создаем перемещение второго треугольника
    glPushMatrix();
    //поворот
    glRotatef(180, 0, 0, 1);
    //сдвиг по оси X
    glTranslatef(40, 0, 0);

    glColor3ub(255, 200, 200);
    glBegin(GL_TRIANGLES);
    glVertex3f(-30.f, -30.f, -1.f);
    glVertex3f(0.f, 30.0f, -1.f);
    glVertex3f(30.f, -30.f, -1.f);
    glEnd();
    glPopMatrix();
    glutSwapBuffers();
}
```



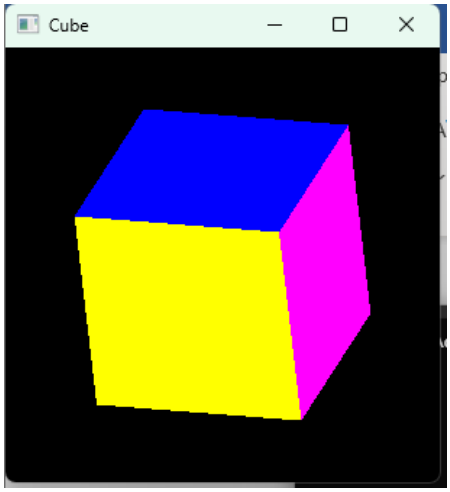
В этой программе отображаются два треугольника. Белый(правый) перемещается на 40 по оси X, а только затем поворачивается на 180 градусов относительно оси Z. Розовый (левый) треугольник выполняет те же действия, но в обратном порядке. Поэтому для него сместиться на 40, это переместиться влево относительно наблюдателя.
Процедура glPushMatrix служит для помещения текущей матрицы в стек , для извлечения матрицы из стека служит процедура glPopMatrix. Поэтому все преобразования переноса и поворота (которые находятся между glPushMatrix и glPopMatrix) для первого треугольника не влияют на второй.

В OpenGL существуют две матрицы, последовательно применяющиеся в преобразовании координат. Одна из них – матрица моделирования (modelview matrix), а другая – матрица проецирования (projection matrix). Первая служит для задания положения объекта и его ориентации, вторая отвечает за выбранный способ проецирования. Существует набор различных процедур, умножающих текущую матрицу (моделирования или проецирования) на матрицу выбранного геометрического преобразования.

Текущая матрица задается при помощи процедуры glMatrixMode(GLenum mode). Параметр mode может принимать значения GL_MODELVIEW, GL_TEXTURE или GL_PROJECTION, позволяя выбирать в качестве текущей матрицы матрицу моделирования (видовую матрицу), матрицу проецированияили матрицу преобразования текстуры.

Процедура glLoadldenity() устанавливает единичную текущую матрицу.Обычно задание соответствующей матрицы начинается с установки единичной матрицы и последовательного применения матриц геометрических преобразований. Если указано несколько преобразований, то текущая матрица в результате будет последовательно умножена на соответствующие матрицы.

Пример «Куб».



Листинг 7.

```
#include "glew.h"
#include "glut.h"


double rotate_y = 0;
double rotate_x = 0;


void display() {

    //  Очищаем буферы
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    // Поворачиваем на угол
    glRotatef(rotate_x, 1.0, 0.0, 0.0);
    glRotatef(rotate_y, 0.0, 1.0, 0.0);

     //Задаем грань куба 1
    glBegin(GL_POLYGON);
    glColor3f(1.0, 1.0, 0.0);
    glVertex3f(0.5, -0.5, -0.5);
    glVertex3f(0.5, 0.5, -0.5);
    glVertex3f(-0.5, 0.5, -0.5);
    glVertex3f(-0.5, -0.5, -0.5);
    glEnd();

    //Задаем грань куба 2
    glBegin(GL_POLYGON);
    glColor3f(1.0, 1.0, 1.0);
    glVertex3f(0.5, -0.5, 0.5);
    glVertex3f(0.5, 0.5, 0.5);
    glVertex3f(-0.5, 0.5, 0.5);
    glVertex3f(-0.5, -0.5, 0.5);
    glEnd();

    //Задаем грань куба 3
    glBegin(GL_POLYGON);
    glColor3f(1.0, 0.0, 1.0);
    glVertex3f(0.5, -0.5, -0.5);
    glVertex3f(0.5, 0.5, -0.5);
    glVertex3f(0.5, 0.5, 0.5);
    glVertex3f(0.5, -0.5, 0.5);
    glEnd();

    //Задаем грань куба 4
    glBegin(GL_POLYGON);
    glColor3f(0.0, 1.0, 0.0);
    glVertex3f(-0.5, -0.5, 0.5);
    glVertex3f(-0.5, 0.5, 0.5);
    glVertex3f(-0.5, 0.5, -0.5);
    glVertex3f(-0.5, -0.5, -0.5);
    glEnd();

    //Задаем грань куба 5
    glBegin(GL_POLYGON);
    glColor3f(0.0, 0.0, 1.0);
    glVertex3f(0.5, 0.5, 0.5);
    glVertex3f(0.5, 0.5, -0.5);
    glVertex3f(-0.5, 0.5, -0.5);
    glVertex3f(-0.5, 0.5, 0.5);
    glEnd();

    //Задаем грань куба 6
    glBegin(GL_POLYGON);
    glColor3f(1.0, 0.0, 0.0);
```

```
    glVertex3f(0.5, -0.5, -0.5);
    glVertex3f(0.5, -0.5, 0.5);
    glVertex3f(-0.5, -0.5, 0.5);
    glVertex3f(-0.5, -0.5, -0.5);
    glEnd();


    glutSwapBuffers();

}

void specialKeys(int key, int x, int y) {

    //  Меняем угол на 5 градусов
    if (key == GLUT_KEY_RIGHT)
        rotate_y += 5;
    else if (key == GLUT_KEY_LEFT)
        rotate_y -= 5;
    else if (key == GLUT_KEY_UP)
        rotate_x += 5;
    else if (key == GLUT_KEY_DOWN)
        rotate_x -= 5;

    glutPostRedisplay();
}

int main() {
    glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH);
    glutCreateWindow("Cube");
    glEnable(GL_DEPTH_TEST);
    glutDisplayFunc(display);
    glutSpecialFunc(specialKeys);
    glutMainLoop();
  }
```

В этой программе можно менять угол поворота куба с помощью четырех клавиш-стрелок на клавиатуре. А почему какие-то грани куба (лицевые) видны наблюдателю, а некоторые (задние) не видны? В данной программе задействован алгоритм отбрасывания лицевых и тыловых граней. В OpenGL по умолчанию считается, что передняя грань многоугольника обходится **против часовой стрелки**. Как вскоре станет ясно, передним и задним граням многоугольников часто нужно приписать различные физические характеристики. Заднюю грань многоугольника можно вообще скрыть или наделить другими отражательными свойствами и цветом. При этом важно следить за согласованностью всех многоугольников сцены и использовать направленные вперед многоугольники для изображения внешних поверхностей сплошных объектов.

Если поведение OpenGL по умолчанию требуется изменить на противоположное, вызывается следующая функция.

glFrontFace(GL_CW);

Параметр GL_CW сообщает OpenGL, что передней гранью нужно считать многоугольники с обходом по часовой стрелке. Чтобы вернуться к обходу против часовой стрелки, следует указать параметр GL_CCW.

Вторая возможность скрыть невидимые части объекта — это тест глубины, который использует буфер глубины. Для того, чтобы разрешить тест глубины необходимо задействовать два момента:

1. разрешить его вызовом glEnable с параметром GL_DEPTH_TEST

2. настроить параметры теста, а именно правило, по которому новый фрагмент будет проходить тест или будет отбрасываться.

Настройка теста глубины производится командой:

void glDepthFunc (GLenum func)

где в качестве параметра указывает один из стандартных идентификаторов, каждый из которых определяет случай, в котором новый фрагмент будет проходить тест и попадать в буфер кадра:

GL_LESS – если его глубина меньше глубины фрагмента в буфере

GL_GREATER — если его глубина больше глубины фрагмента в буфере

GL_EQUAL — если его глубина равна губине фрагмента в буфере

GL_LEQUAL — если его глубина меньше либо равна глубине фрагмента ф буфере

GL_GEQUAL — если его глубина больше либо равна глубине фрагмента в буфере

GL_NOTEQUAL — если его глубина не равна глубине фрагмента в буфере

Для очистки буфера глубины необходимо вызвать glClear с параметром

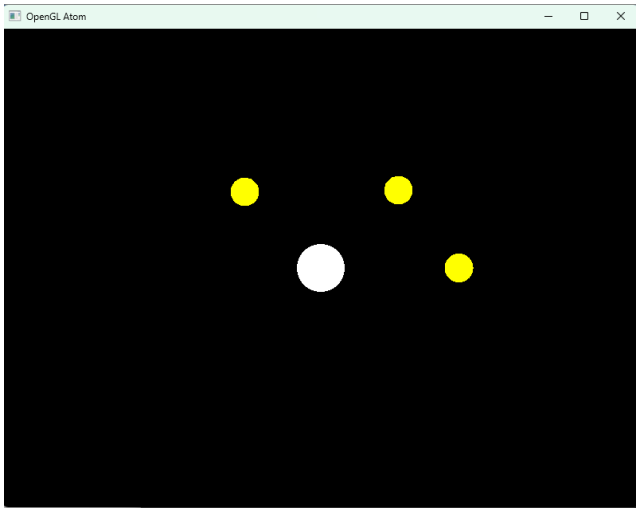
GL_DEPTH_BUFFER_BIT. Так как этот параметр флаговый, мы можем совместить его с уже имеющимся GL_COLOR_BUFFER_BIT и очистить оба буфера одновременно:

```
glClear ( GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT );
```

Пример «Атом».

Воспользуемся полученными знаниями. В следующем примере построим грубую анимированную модель атома. Атом имеет сферу в центре, представляющую ядро, и три электрона на орбите вокруг атома. Как и ранее, используем ортографическую проекцию.

В программе ATOM задействован механизм GLUT обратного вызова таймера, с помощью которого сцена перерисовывается примерно 10 раз в секунду. При каждом вызове функции RenderScene увеличивается угол поворота элемента относительно ядра. Кроме того, каждый электрон находится на отдельной плоскости. В листинге 8 показан результат выполнения программы ATOM.



Листинг 8.

```
#include "glew.h"
#include "glut.h"
#include <math.h>


// Вызывается для рисования сцены
void RenderScene(void)
{
	// Угол поворота вокруг ядра
	static GLfloat fElect1 = 0.0f;
	// Очищаем окно текущим цветом очистки
	glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
	//Обновляем матрицу наблюдения модели
	glMatrixMode(GL_MODELVIEW);
	glLoadIdentity();
	//Транслируем всю сцену в поле зрения
	//Это исходное преобразование наблюдения
	glTranslatef(0.0f, 0.0f, -100.0f);
	// Красное ядро
	glColor3ub(255, 255, 255);
	glutSolidSphere(10.0f, 150, 150);
	// Желтые электроны
	glColor3ub(255, 255, 0);
	// Орбита первого электрона
	// Записываем преобразование наблюдения
	glPushMatrix();

	glRotatef(fElect1, 0.0f, 1.0f, 0.0f);

	glTranslatef(90.0f,  0.0f,  0.0f);
	// Рисуем электрон
	glutSolidSphere(6.0f, 150, 150);
	// Восстанавливаем преобразование наблюдения
	glPopMatrix();
	//Орбита второго электрона
	glPushMatrix();
	glRotatef(45.0f, 0.0f, 0.0f, 1.0f);
	glRotatef(fElect1, 0.0f, 1.0f, 0.0f);
	glTranslatef(-70.0f, 0.0f, 0.0f);
	glutSolidSphere(6.0f, 15, 15);
	glPopMatrix();
	// Орбита третьего электрона
	glPushMatrix();
	glRotatef(360.0f - 45.0f, 0.0f, 0.0f, 1.0f);
	glRotatef(fElect1, 0.0f, 1.0f, 0.0f);
	glTranslatef(0.0f, 0.0f, 60.0f);
	glutSolidSphere(6.0f, 15, 15);
	glPopMatrix();
```

```
	// Увеличиваем угол поворота
	fElect1 += 10.0f;
	if (fElect1 > 360.0f)
		fElect1 = 0.0f;
	// Показываем построенное изображение
	glutSwapBuffers();
}
// Функция выполняет необходимую инициализацию
// в контексте визуализации
void SetupRC()
{
	glEnable(GL_DEPTH_TEST); // Удаление скрытых поверхностей
	glFrontFace(GL_CCW); // Полигоны с обходом против
	//часовой стрелки направлены наружу
	glEnable(GL_CULL_FACE); //Внутри пирамиды расчеты не //производятся
	// Черный фон
	glClearColor(0.0f, 0.0f, 0.0f, 1.0f);
}

void TimerFunc(int value)
{
	glutPostRedisplay();
	glutTimerFunc(100, TimerFunc, 1);
}
void ChangeSize(int w, int h)
{
	GLfloat nRange = 100.0f;
	// Предотвращение деления на ноль
	if (h == 0)
		h = 1;
	// Устанавливает поле просмотра по размерам окна
	glViewport(0, 0, w, h);
	// Обновляет стек матрицы проектирования
	glMatrixMode(GL_PROJECTION);
	glLoadIdentity();
	// Устанавливает объем отсечения с помощью отсекающих
	// плоскостей (левая, правая, нижняя, верхняя,
	// ближняя, дальняя)
	if (w <= h)
		glOrtho(-nRange, nRange, nRange * h / w, -nRange * h / w, -nRange * 2.0f, nRange *
2.0f);
	else
		glOrtho(-nRange * w / h, nRange * w / h, nRange, -nRange, -nRange * 2.0f, nRange *
2.0f);
	glMatrixMode(GL_MODELVIEW);
	glLoadIdentity();
}
int main(int argc, char* argv[])
{
	glutInit(&argc, argv);
	glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH);
	glutInitWindowSize(800, 600);
	glutCreateWindow("OpenGL Atom");
	glutReshapeFunc(ChangeSize);
	glutDisplayFunc(RenderScene);
	glutTimerFunc(500, TimerFunc, 1);
	SetupRC();
	glutMainLoop();
	return 0;
}
```

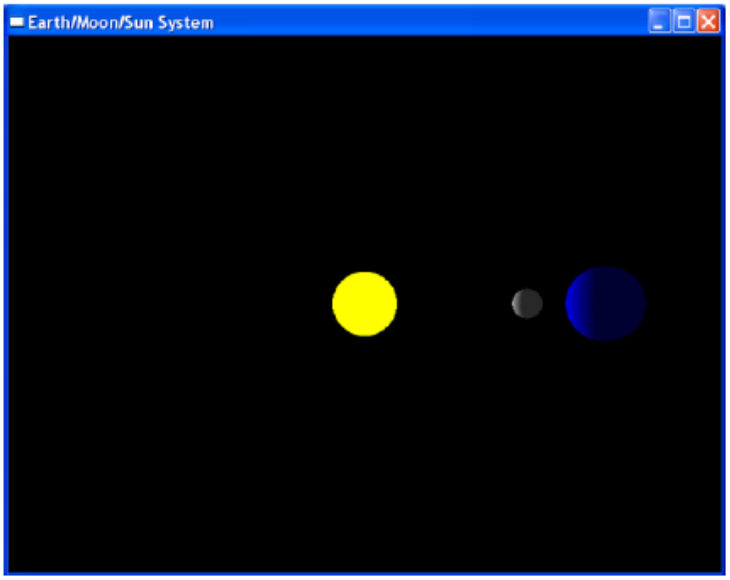
Пример система «Солнце-Земля-Луна».

Чтобы создать завершенный пример, демонстрирующий работу с матрицей наблюдения модели и перспективной проекцией, рассмотрим моделирование в программе вращение системы "Солнце — Земля — Луна". Это классический пример вложенных преобразований, когда объекты преобразовываются относительно друг друга с

использованием стека матриц. Чтобы сделать пример более эффектным, добавлены функции освещения и затенения.

В данной модели Земля движется вокруг Солнца, а Луна — вокруг Земли. Источник света находится в центре Солнца, которое нарисовано без освещения, создавая иллюзию сияющего источника света. Пример демонстрирует, насколько просто с помощью OpenGL получать сложные эффекты.

В листинге 4 приводится код, задающий проекцию, и код визуализации, отвечающий за движение системы. Таймер инициирует перерисовывание окна 10 раз в секунду, поддерживая активной функцию RenderScene. Обратите внимание, что когда Земля расположена перед Солнцем, она кажется больше; Земля, находящаяся с противоположной стороны, выглядит меньше.



Листинг 9.

```
#include "glew.h"
#include "glut.h"
#include <math.h>
// Параметры освещения
GLfloat whiteLight[] = { 0.2f, 0.2f, 0.2f, 1.0f };
GLfloat sourceLight[] = { 0.8f, 0.8f, 0.8f, 1.0f };
GLfloat lightPos[] = { 0.0f, 0.0f, 0.0f, 1.0f };
// Вызывается для рисования сцены
void RenderScene(void)
{
	// Угол поворота системы Земля/Луна
	static float fMoonRot = 0.0f;
	static float fEarthRot = 0.0f;
	// Очищаем окно текущим цветом очистки
	glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
	// Save the matrix state and do the rotations
	glMatrixMode(GL_MODELVIEW);
	glPushMatrix();
	// Транслируем всю сцену в поле зрения
	glTranslatef(0.0f, 0.0f, -300.0f);
	// Устанавливаем цвет материала красным
	// Солнце
	glDisable(GL_LIGHTING);
	glColor3ub(255, 255, 0);
	glutSolidSphere(15.0f, 30, 17);
	glEnable(GL_LIGHTING);
	// Движение источника света, после прорисовки солнца!
	glLightfv(GL_LIGHT0, GL_POSITION, lightPos);
	// Поворот системы координат
```

```
	glRotatef(fEarthRot, 0.0f, 1.0f, 0.0f);
	// Прорисовка Земли
	glColor3ub(0, 0, 255);
	glTranslatef(105.0f, 0.0f, 0.0f);
	glutSolidSphere(15.0f, 30, 17);
	// Поворот в системе координат, связанной с Землей
	// и изображение Луны
	glColor3ub(200, 200, 200);
	glRotatef(fMoonRot, 0.0f, 1.0f, 0.0f);
	glTranslatef(30.0f, 0.0f, 0.0f);
	fMoonRot += 15.0f;
	if (fMoonRot > 360.0f)
		fMoonRot = 0.0f;
	glutSolidSphere(6.0f, 30, 17);
	// Восстанавливается состояние матрицы
	glPopMatrix(); // Матрица наблюдения модели
	// Шаг по орбите Земли равен пяти градусам
	fEarthRot += 5.0f;
	if (fEarthRot > 360.0f)
		fEarthRot = 0.0f;
	// Показывается построенное изображение
	glutSwapBuffers();
}
// Функция выполняет всю необходимую инициализацию в контексте
//визуализации
void SetupRC()
{
	// Параметры света и координаты
	glEnable(GL_DEPTH_TEST); // Удаление скрытых поверхностей
	glFrontFace(GL_CCW); //Многоугольники с обходом против часовой стрелки направлены наружу
	glEnable(GL_CULL_FACE); //Расчеты внутри самолета не выполняются
	// Активация освещения
	glEnable(GL_LIGHTING);
	// Устанавливается и активизируется источник света 0
	glLightModelfv(GL_LIGHT_MODEL_AMBIENT, whiteLight);
	glLightfv(GL_LIGHT0, GL_DIFFUSE, sourceLight);
	glLightfv(GL_LIGHT0, GL_POSITION, lightPos);
	glEnable(GL_LIGHT0);
	// Активизирует согласование цветов
	glEnable(GL_COLOR_MATERIAL);
	// Свойства материалов соответствуют кодам glColor
	glColorMaterial(GL_FRONT, GL_AMBIENT_AND_DIFFUSE);
	// Темно-синий фон
	glClearColor(0.0f, 0.0f, 0.0f, 1.0f);
}
void TimerFunc(int value)
{
	glutPostRedisplay();
	glutTimerFunc(100, TimerFunc, 1);
}
void ChangeSize(int w, int h)
{
	GLfloat fAspect;
	// Предотвращает деление на ноль
	if (h == 0)
		h = 1;
	// Размер поля просмотра устанавливается равным размеру окна
	glViewport(0, 0, w, h);
	// Расчет соотношения сторон окна
	fAspect = (GLfloat)w / (GLfloat)h;
	// Устанавливаем перспективную систему координат
	glMatrixMode(GL_PROJECTION);
	glLoadIdentity();
	// Поле обзора равно 45 градусов, ближняя и дальняя плоскости
	// проходят через 1 и 425
	gluPerspective(45.0f, fAspect, 1.0, 425.0);
	// Обновляем матрицу наблюдения модели
	glMatrixMode(GL_MODELVIEW);
	glLoadIdentity();
}
int main(int argc, char* argv[])
```

```
{
      glutInit(&argc, argv);
      glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH);
      glutInitWindowSize(800, 600);
      glutCreateWindow("Earth/Moon/Sun System");
      glutReshapeFunc(ChangeSize);
      glutDisplayFunc(RenderScene);
      glutTimerFunc(250, TimerFunc, 1);
      SetupRC();
      glutMainLoop();
  }
```

ЗАДАНИЕ НА ЛАБОРАТОРНУЮ РАБОТУ

Задание 1.

Создать приложение, выводящее на экран вращающуюся пирамиду, каждая грань которой имеет свой собственный цвет.

Задание 2.

Для Листинга 8 реализовать изменения согласно варианту:

1) Клонировать атомную систему и расположить одну слева окна, другую – справа окна.

2) Добавить еще 2 электрона к существующей системе в разные плоскости вращения.

3) Изменить круговые орбиты на произвольные замкнутые.

4) Добавить один электрон на максимально большое расстояние от ядра.

5) Осуществить небольшие круговые движения для центрального атома.

6) Поменять плоскость вращения всех электронов.

7) Изменить скорость и направление движения для одного электрона.

7) Добавить еще один атом с одним электроном на орбите.

8) Осуществить движение всей атомной системы с небольшой амплитудой.

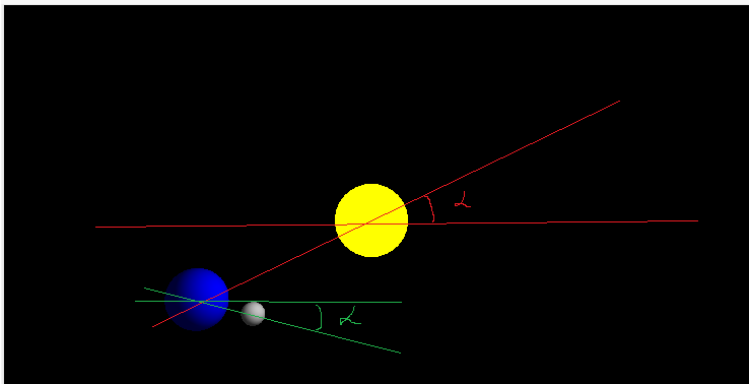
9) Клонировать атомную систему и расположить одну вверху окна, другую – внизу окна.

10) Изменить скорости вращения, цвет и направление вращения каждого атома.

Задание 3.

Для Листинга 9 реализовать изменения согласно варианту:

Определим угол альфа как угол между горизонтальной плоскостью и новой плоскостью вращения (будущей плоскостью вращения) объекта в том виде, в котором мы видим его на экране.



1. Изменить α плоскостей вращения Земли и Луны: 45^0 и 30^0
2. Изменить α плоскостей вращения Земли и Луны: 30^0 и 45^0
3. Изменить α плоскостей вращения Земли и Луны: 60^0 и 45^0
4. Изменить плоскость вращения Земли и Луны, скорость и направление вращения Земли.
5. Добавить одну планету со спутником вне плоскости движения Земли с α между плоскостями Земли и планеты -30^0 . Новая планета должна двигаться с более медленной скоростью, чем Земля.
6. Добавить одну планету со спутником вне плоскости движения Земли с α между плоскостями Земли и планеты 60^0 . Новая планета должна двигаться с более быстрой скоростью, чем Земля.
7. Добавить одну планету без спутника на другую траекторию. Новая планета должна двигаться с более быстрой скоростью, чем Земля, но периоды обращения Земли и новой планеты должны быть одинаковыми.
8. Добавить к Земле еще один спутник меньшего размера, находящийся на другой траектории и с вращением в противоположную сторону.
9. Добавить к Земле еще один спутник большего размера, находящийся на другой траектории и с вращением в противоположную сторону.
10. Добавить в систему две планеты одна из них со спутником. Планеты находятся в разных плоскостях (одна ближе к Солнцу, другая дальше от Солнца).

КОНТРОЛЬНЫЕ ВОПРОСЫ И ЗАДАНИЯ

1. Раскройте терминологию преобразований OpenGL.

2. Дайте определение координат наблюдения и опишите их связь с трехмерной декартовой системой.

3. Поясните, в чем заключается дуализм преобразования проекции модели.

4. Покажите, как работает конвейер преобразований, и перечислите его этапы.

5. Приведите способы непропорционального масштабирования.

6. Охарактеризуйте, как средствами OpenGL описывается преобразование поворота.

7. Приведите описание трансляции на уровне матриц.

8. Раскройте назначение единичной матрицы.

9. Опишите область применения стека матриц и алгоритм его работы.

10. Сформулируйте суть нетривиального умножения матриц и преимущества его использования.

11. Изложите концепцию использовавшейся в работе функции таймера.

12. Раскройте значение термина *актеры* в контексте построения сцен.