Министерство образования и науки Российской Федерации

Калужский филиал
федерального государственного бюджетного образовательного
учреждения высшего образования
**«Московский государственный технический университет
имени Н.Э. Баумана
(национальный исследовательский университет)»**
(КФ МГТУ им. Н.Э. Баумана)

Ю.С. Белов

РАБОТА С МАТРИЦАМИ И ПРЕОБРАЗОВАНИЯМИ В OPENGL

Методические указания к лабораторной работе
по дисциплине «Компьютерная графика»

Калуга, 2018

УДК 004.62
ББК 32.972.5
Б435

Методические указания составлены в соответствии с учебным планом КФ МГТУ им. Н.Э. Баумана по направлению подготовки 09.03.04 «Программная инженерия» кафедры «Программного обеспечения ЭВМ, информационных технологий и прикладной математики».

Методические указания рассмотрены и одобрены:

- Кафедрой «Программного обеспечения ЭВМ, информационных технологий и прикладной математики» (ФН1-КФ) протокол №_7_ от «_21_» __февраля____ 2018 г.

И.о. зав. кафедрой ФН1-КФ _____к.т.н., доцент Ю.Е. Гагарин

- Методической комиссией факультета ФНК протокол №_2_ от «_28_» ____02____ 2018 г.

Председатель методической
комиссии факультета ФНК _____к.х.н., доцент К.Л. Анфилов

- Методической комиссией
КФ МГТУ им.Н.Э. Баумана протокол №_2_ от «_06_» ____03____ 2018 г.

Председатель методической комиссии
КФ МГТУ им.Н.Э. Баумана
_____д.э.н., профессор О.Л. Перерва

Рецензент:
к.т.н., зав. кафедрой ЭИУ2-КФ _____ И.В. Чухраев

Авторы
к.ф.-м.н., доцент кафедры ФН1-КФ _____ Ю.С. Белов

Аннотация

Методические указания по выполнению лабораторной работы по курсу «Компьютерная графика» содержат общие сведения о структуре и применении программного интерфейса OpenGL. В методических указаниях приводятся теоретические сведения матричных преобразованиях и проекциях, реализуемых в OpenGL. Рассмотрен процесс работы со стеком матриц, создания проэцирования и трансформирования сложных сцен, управления положением зрителя в координатном пространстве сцены, а также приведен примеры создания стандартных трезмерных объектов в OpenGL.

Предназначены для студентов 2-го курса бакалавриата КФ МГТУ им. Н.Э. Баумана, обучающихся по направлению подготовки 09.03.04 «Программная инженерия».

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

Настоящие методические указания составлены в соответствии с программой проведения лабораторных работ по курсу «Компьютерная графика» на кафедре «Программное обеспечение ЭВМ, информационные технологии и прикладная математика» факультета фундаментальных наук Калужского филиала МГТУ им. Н.Э. Баумана.

Методические указания, ориентированные на студентов 2-го курса бакалавриата направления подготовки 09.03.04 «Программная инженерия», содержат краткую теоретическую часть, описывающую способы осуществления преобразований матриц средствами программного интерфейса OpenGL, поэтапные примеры создания многокомпонентных сцен и их отображения с учетом выбранных систем проецирования, комментарии и пояснения по вышеназванным этапам, а также задание на лабораторную работу.

Методические указания составлены в расчете на начальное ознакомление студентов с основами работы с программным интерфейсом OpenGL. Для выполнения лабораторной работы студенту необходимо сформировать понимание способа осуществления преобразований пространства с помощью матриц в OpenGL, уметь с различными системами проекций, создавать и редактировать графические примитивы, овладеть терминологией трансформаций.

Программный интерфейс OpenGL, кратко описанный в методических указаниях, может быть использован при создании моделей использующих конвейер трехмерной графики.

ЦЕЛЬ И ЗАДАЧИ РАБОТЫ, ТРЕБОВАНИЯ К РЕЗУЛЬТАТАМ ЕЕ ВЫПОЛНЕНИЯ

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

Основными задачами выполнения лабораторной работы являются: 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.

Результатами работы являются:

- Выполненные преобразования проекций средствами OpenGL
- Реализованные согласно варианту анимаций сцен из множества объектов
- Соблюдение пропорций и размеров объектов
- Подготовленный отчет

ХАРАКТЕРИСТИКА ОБЪЕКТА ИЗУЧЕНИЯ, ИССЛЕДОВАНИЯ

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.

Ключом к преобразованию объектов и координат являются две матрицы, поддерживаемые OpenGL. Между заданием вершин и появлением их на экране проходит три типа преобразований: наблюдения, модели и проектирования. В данном разделе мы исследуем принципы преобразований всех типов, обобщенные в табл. 1.

Таблица 1. Терминология преобразований OpenGL

Преобразование	Использование
Наблюдения	Задает положение наблюдателя или камеры
Модели	Перемещает объекты по сцене
Наблюдения модели	Описывает дуализм преобразований наблюдателя и модели
Проектирования	Обрезает и задает размеры наблюдаемого объекта
Поля просмотра	Псевдопреобразование, масштабирующее конечный результат согласно размерам окна

Координаты наблюдения

Здесь используется важная концепция координат наблюдения. Координаты наблюдения отсчитываются от точки, в которой расположен глаз наблюдателя, независимо от любых возможных преобразований;

эти координаты можно рассматривать как «абсолютные» экранные координаты. Таким образом, координаты системы наблюдения не являются действительными; они представляют виртуальную неподвижную систему координат, которая используется как внешняя системы отсчета. Все преобразования описываются с точки зрения их эффекта в системе координат наблюдателя.

На рис.1 показана система координат наблюдения с двух точек обзора: а - координаты представлены так, как они видятся наблюдателю сцены (т.е. перпендикулярно монитору); б – система координат немного повернута, чтобы лучше была видна ось z. Положительные направления осей x и y идут вправо и вверх, соответственно, с точки зрения наблюдателя. Положительное направление оси z идет от начала координат к пользователю, а отрицательные значения z растут от точки наблюдения вглубь экрана.

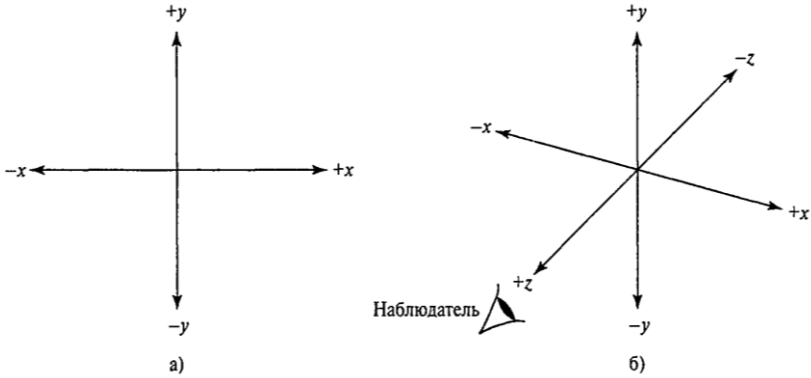


Рис.1 – Координаты наблюдения с двух точек зрения

При рисовании в трехмерном пространстве с помощью OpenGL применяется декартова система координат. При отсутствии преобразований используемая система идентична описанной выше системе координат наблюдения.

Преобразования наблюдения

Первым к сцене будет применено преобразование наблюдения. С его помощью определяется положение камеры, «смотрящей» на сцену. По умолчанию при перспективной проекции точка наблюдения находится в начале координат (0,0,0), а «взгляд» камеры направлен по отрицательному направлению оси z («внутрь» экрана монитора). Чтобы получить выгодное положение камеры, эта точка перемещается относительно системы координат наблюдения. Если точка наблюдения расположена в начале координат, объекты, изображаемые с положительным значением координаты z, располагаются позади наблюдателя.

Преобразование наблюдения позволяет разместить точку наблюдения в любом удобном месте и выбирать любое направление взгляда. Определение преобразования наблюдения подобно расположению и выбору ориентации камеры на сцене.

Преобразование наблюдения нужно задавать до остальных преобразований. Это объясняется тем, что выбор влияет на положение текущей рабочей системы координат относительно системы координат наблюдения. Все последующие преобразования основаны на уже модифицированной системе координат.

Преобразования модели

Преобразования модели позволяют манипулировать моделью и конкретными объектами, ее составляющими. С помощью этих преобразований объекты расставляются по местам, поворачиваются и масштабируются. На рис. 2 иллюстрируются три наиболее распространенные преобразования модели, которые будет применяться к объектам: а – трансляция, когда объект перемещается вдоль указанной оси; б – поворот, когда объект поворачивается вокруг одной из осей; в – эффект масштабирования, когда размеры объекта увеличиваются или уменьшаются на заданную величину.

Масштабирование может быть неравномерным (с разными коэффициентами изменения размеров в разных направлениях), поэтому с его помощью можно растягивать и сжимать объекты. Окончательный

внешний вид сцены или объекта может сильно зависеть от порядка применения преобразований модели.

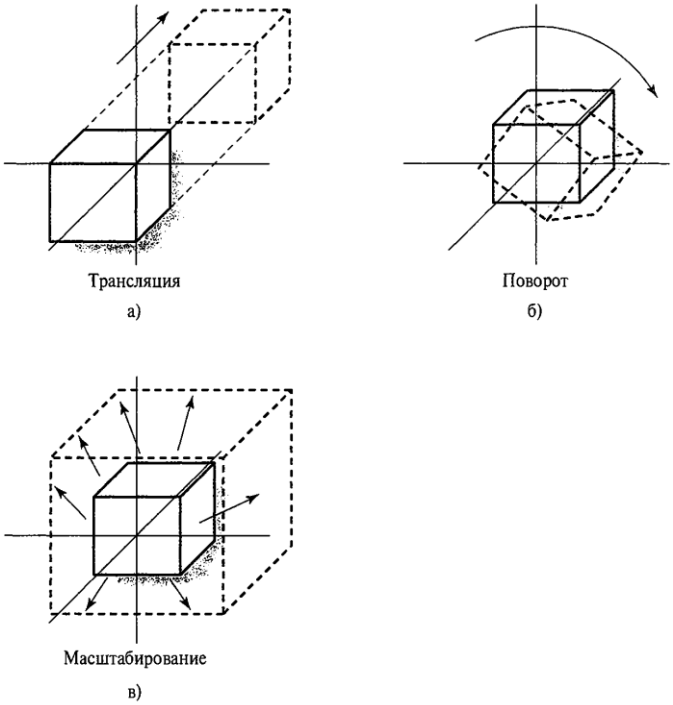


Рис.2 – Преобразования модели

Особенно это справедливо для трансляции и поворота. На рис.3(а) показано, что получается из квадрата, вначале повернутого вокруг оси z, а затем транслированного по положительному направлению новой оси x. На рис. 3(б) тот же квадрат был вначале транслирован по оси x, а затем повернут вокруг оси z. Разница в конечных положениях квадрата объясняется тем, что каждое преобразование выполняется относительно последнего выполненного преобразования. На рис. 3(а) квадрат сперва повернут относительно начала координат. На рис. 3(б) после трансляции квадрата был выполнен поворот вокруг нового начала координат.

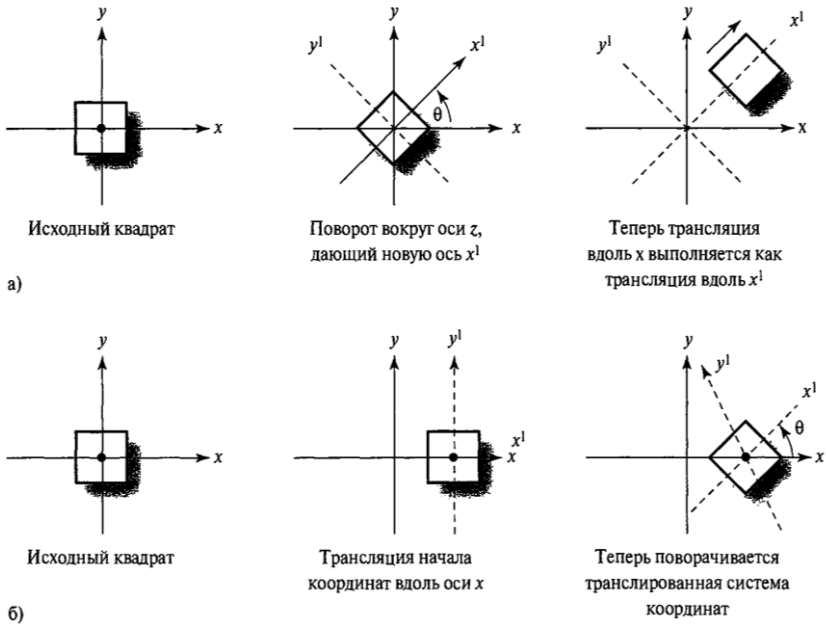


Рис.3 – Преобразования наблюдения модели поворот/трансляция и трансляция/поворот

Дуализм проекции модели

Преобразования наблюдения и модели, по сути, аналогичны с точки зрения их эффекта, а также общего влияния на конечный внешний вид сцены. Эти преобразования различаются исключительно для удобства программиста. Реального различия между движением объекта вперед и движением системы отсчета назад нет; как показано на рис. 4, суммарный эффект получается одинаковым. Термин проекция модели (modelview) означает, что данное преобразование можно считать либо преобразованием модели, либо преобразованием наблюдения (проектирования), но фактически разницы нет; следовательно, это преобразование проекции модели (преобразование наблюдения модели).

Таким образом, преобразование наблюдения – это, по сути, преобразование модели, применяемое к виртуальному объекту (наблюдателю) перед рисованием объектов. Как будет показано ниже, по мере добавления объектов на сцену последовательно задаются новые преобразования. По договоренности исходное преобразование дает точку отсчета для всех остальных преобразований.

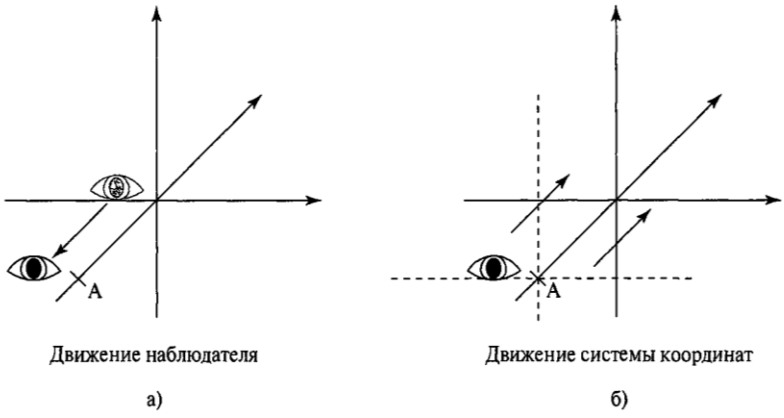


Рис.4 – Два определения концепции преобразования наблюдения

Преобразование проектирования

Преобразование проектирования применяется к вершинам после преобразования наблюдения модели. Данное проектирование в действительности определяет наблюдаемый объем и устанавливает плоскости отсечения. Плоскостями отсечения называются уравнения плоскости в трехмерном пространстве, на основании которых OpenGL определяет, какие геометрические объекты увидит наблюдатель. Говоря более конкретно, преобразование проектирования задает, как законченная сцена (окончательно смоделированная) проектируется в конечное изображение на экране. Рассмотрим два типа проекций: ортографической и перспективной.

В ортографической (или параллельной) проекции все многоугольники, нарисованные на экране, сохраняют заданные точные относи-

тельные размеры. Отрезки и многоугольники отображаются непосредственно на двухмерный экран с помощью параллельных линий, а это означает, что независимо от того, насколько далеко находится объект, он будет изображен с первоначальными размерами (только в виде плоского рисунка на экране). Такой тип проектирования обычно используется в сфере автоматизированного проектирования или для визуализации двухмерных изображений (например, проектов и чертежей), а также такой двухмерной графики, как текст или меню на экране.

При перспективной проекции сцены выглядят более приближенно к реальной жизни, а не к чертежу. Товарным знаком перспективной проекции является ракурс, из-за которого удаленные объекты кажутся меньше более близких объектов такого же размера. Линии, которые были параллельны в трехмерном пространстве, не всегда кажутся параллельными наблюдателю. Рельсы железной дороги, например, параллельны, но, если использовать перспективную проекцию, покажется, что они сходятся в удаленной точке.

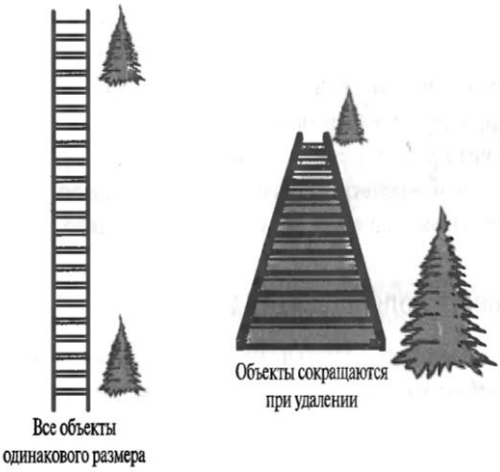


Рис.5 – Ортографическая и перспективная проекции

Достоинством перспективной проекции является то, что не нужно беспокоиться о том, где сходятся линии или насколько малы удаленные объекты. Все, что требуется, - просто задать сцену, используя преобразование наблюдения модели, а затем применить перспективную проекцию, а все остальное сделает OpenGL. Для примера на рис. 5 на двух различных сценах сравниваются ортографическая и перспективная проекции.

Ортографические проекции используются чаще всего для двухмерного рисования, когда требуется точное соответствие между пикселями и элементами изображения. Они могут применяться для схематических чертежей, текста или в двухмерных графических приложениях. Кроме того, ортографическая проекция используется для трехмерной визуализации, когда глубина визуализации очень мала по сравнению с расстоянием от точки наблюдения. Перспективные проекции применяются для визуализации сцен, содержащих широкие открытые пространства или объекты, требующие изображения с учетом ракурса. Большей частью в сфере трехмерной графики используются именно перспективные проекции.

Преобразования поля просмотра

После того как все сказано и сделано, вы получаете двухмерную проекцию сцены, которая будет отображаться в окно на экране. Это отображение в физические координаты окна является последним преобразованием, именуемым преобразованием поля просмотра.

Матрицы трехмерной графики

Теперь можно понять простую математику матриц. Рассмотрим, как OpenGL выполняет эти преобразования и функции, которые вызываются для достижения искомого эффекта.

Математика, лежащая в основе данных преобразований, сильно упрощается с введением матричной формы записи. Все описанные выше преобразования можно представить умножением матрицы, содержащей вершины (обычно это не просто вектор), на матрицу, описываю-

щую преобразование. Таким образом, все преобразования, доступные с помощью OpenGL, можно описать как произведение двух или нескольких перемноженных матриц.

Матрица и вектор являются двумя важными терминами, которые часто встречаются в литературе по программированию трехмерной графики. Оперируя этими понятиями, также придется столкнуться с термином скаляр. Скаляр – это просто число, представляющее амплитуду или некоторую величину.

Конвейер преобразований

Чтобы реализовать преобразования, модифицируются две матрицы: матрицу наблюдения модели и матрицу проектирования. OpenGL предлагает несколько функций высокого уровня, которые можно вызвать для выполнения этих преобразований. Овладев основами программного интерфейса OpenGL, несомненно, можно, попытататься использовать более сложные технологии трехмерной визуализации. Только тогда понадобятся низкоуровневые функции, которые в действительности устанавливают значения, содержащиеся в матрицах. Последовательность от данных о вершинах к экранным координатам является долгой и на рис.6 приведена схема этого процесса.

Рис.6 – Конвейер преобразования вершины

Вначале вершины преобразуются в матрицу 1x4, тремя первыми значениями которой являются координаты x,y и z. Четвертое число – это масштабный коэффициент, который можно задавать вручную, используя функции вершин, принимающие четыре значения. Эта координата обозначается w, и по умолчанию ее значение равно 1.0. Менять это значение требуется редко.

После этого вершина умножается на матрицу наблюдения модели, что дает преобразованные координаты системы наблюдения. Затем координаты системы наблюдения множатся на матрицу проекции и дают координаты отсечения. Таким образом, эффективно устраняются данные, не входящие в наблюдаемый объем. Далее координаты отсечения делятся на координату w и дают нормированные координаты устройства. В зависимости от выполненных преобразований значение w может модифицироваться матрицей проекции или матрицей наблюдения модели. Как и ранее, за подробности этого процесса отвечают OpenGL и высокоуровневые матричные функции.

Наконец, с помощью преобразования поля просмотра тройка координат отображается на двухмерную плоскость.

Матрица наблюдения модели

Матрица наблюдения модели – это матрица 4x4, представляющая преобразованную систему координат, в которой вы располагаете и ориентируете объекты. Вершины, указанные вами в примитивах, используются как матрица-столбец и умножаются на матрицу наблюдения модели, в результате чего получаются новые преобразованные координаты относительно системы наблюдения.

На рис. 7 матрица, содержащая данные по одной вершине, множится на матрицу наблюдения модели, в результате чего получаются новые координаты системы наблюдения. Данные по вершине – это в действительности четыре элемента с дополнительным значением w, которое представляет коэффициент масштабирования. По умолчанию это значение установлено равным 1.0, и редко будет меняться.

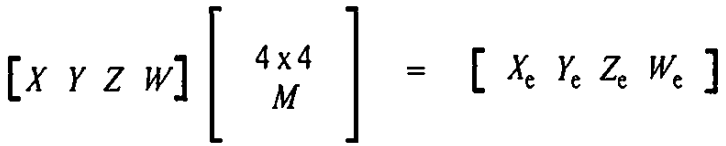


Рис.7 – Матричное уравнение, выражающее применение преобразования наблюдения модели к одной вершине

Трансляция

Рассмотрим пример с модификацией матрицы наблюдения модели. Скажем, требуется нарисовать куб, используя функцию `glutWireCube` библиотеки `GLUT`. Вводите в свою программу строку

```
glutWireCube (10.0f);
```

Создается куб с ребром 10 единиц и центром в начале координат. Чтобы поднять куб по оси y на 10 единиц перед его выводом на экран, вы умножаете матрицу наблюдения модели на матрицу, описывающую трансляцию на 10 единиц по положительному направлению оси y, а затем рисуете объект. «Каркас» кода выглядит следующим образом:

```
//Строится матрица трансляции на 10 единиц в по-
ложительном
// направлении оси y
…
// Эта матрица множится на матрицу наблюдения
модели
…
// Рисуется куб glutWireCube (10.0f);
```

Подобную матрицу очень просто построить, но это требует нескольких строчек кода. К счастью, OpenGL предоставляет высокоуровневые функции, выполняющие это за вас:

```
void glTranslatef (GLfloat x, GLfloat y, GLfloat
z);
```

Функция принимает в качестве параметров величину трансляции по осям x,y и z. После этого она строит подходящую матрицу и вы-

полняет умножение. Соответствующий псевдокод приведен ниже, а общий эффект показан на рис.8.

```
      //Трансляция на 10 единиц в положительном направ-
лении оси y
      glTranslatef (0.0f, 10.0f, 0.0f);
      //Рисуется куб
      glutWireCube (10.0f);
```

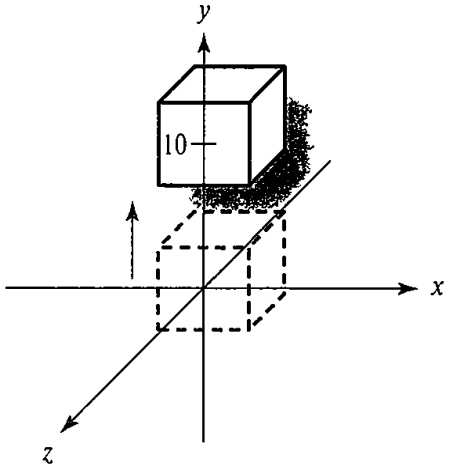


Рис.8 – Куб, транслированный на 10 единиц по положительному направлению оси y

Поворот

Чтобы повернуть объект вокруг одной из трех координатных осей или заданного произвольного вектора, нужно определить матрицу поворота. Здесь нас снова выручает высокоуровневая функция

```
      glRotatef (GLfloat angle, GLfloat x, GLfloat y,
GLfloat z);
```

С помощью этой функции мы выполняем поворот вокруг вектора, определяемого аргументами x, y и z. Угол поворота (против часовой стрелки) измеряется в градусах и задается аргументом `angle`. В простейшем случае поворот выполняется только вокруг одной координатной оси.

Кроме того, можнр выполнить поворот вокруг произвольной оси, задав значения x, y и z направляющего вектора этой оси. Чтобы увидеть ось вращения, можно просто нарисовать линию от начала координат до точки, представленной координатами (x, y, z). В приведенном ниже коде куб поворачивается на 45° вокруг оси, заданной как (1, 1, 1) (соответствующая иллюстрация приведена на рис. 9).

```
// Выполняется преобразование
glRotatef (45.0f, 1.0f, 1.0f, 1.0f);
// Рисуется куб
glWireCube (10.0f);
```

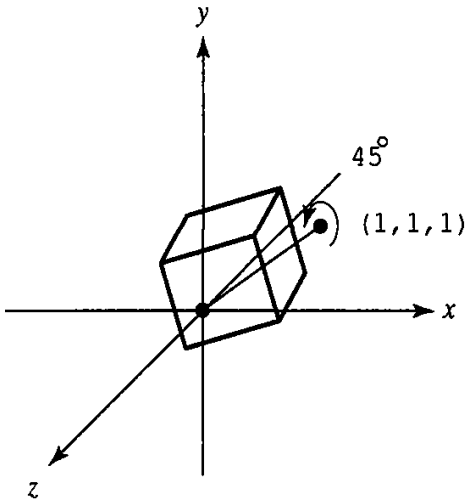


Рис.9 – Куб, повернутый вокруг произвольной оси

Масштабирование

Масштабирование увеличивает размер объекта, отодвигая все его вершины по трем осям от начала координат согласно заданным масштабным коэффициентам. Например, приведенная ниже функция умножает значения x, y и z на заданные масштабные коэффициенты.

```
glScalef (GLgloat x, GLfloat y, GLfloat z);
```

Масштабирование не обязательно должно быть пропорциональным, с его помощью можно растягивать и сжимать объект вдоль разных направлений. Например, следующий код дает куб, вдвое больший вдоль осей x и z, чем кубы, рассмотренные в предыдущем примере, но такой же по величине вдоль оси y. Результат подобного масштабирования показан на рис. 10.

```
// Выполняется преобразование масштабирования
glScalef (2.0f, 1.0f, 2.0f);
// Рисуется куб
glutWireCube (10.0f);
```

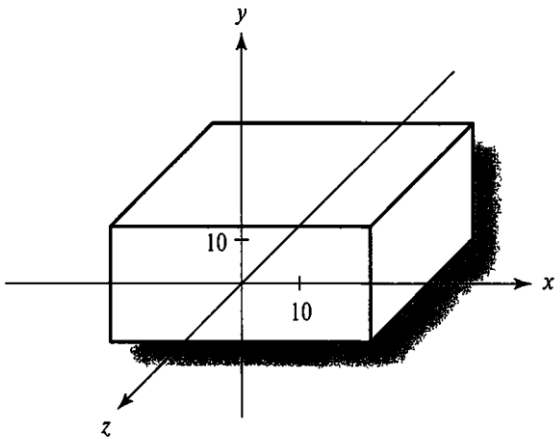


Рис.10 – Непропорциональное масштабирование куба

Единичная матрица

Могкт возникнуть вопросы: зачем вообще работать с матрицами? Почему нельзя просто вызывать указанные функции преобразований, которые будут модифицировать объекты желаемым образом? Действительно ли нужно знать, что мы модифицируем именно матрицу наблюдения модели?

Ответом будет "и да, и нет" (причем только "нет", если вы рисуете на сцене единственный объект). Дело в том, что влияние этих функций кумулятивно. Всякий раз, когда вызывается одну из них, подхо-

дящая матрица строится и умножается на текущую матрицу наблюдения модели. После этого полученная матрица становится текущей матрицей наблюдения модели, на которую будет множиться следующее преобразование и т.д.

Предположим, требуется нарисовать две сферы — одну с центром в точке с координатой y = 10, а другую — с центром в точке x = 10, как показано на рис. 11.

Возможно, для решения этой задачи вы напишете примерно такой код.

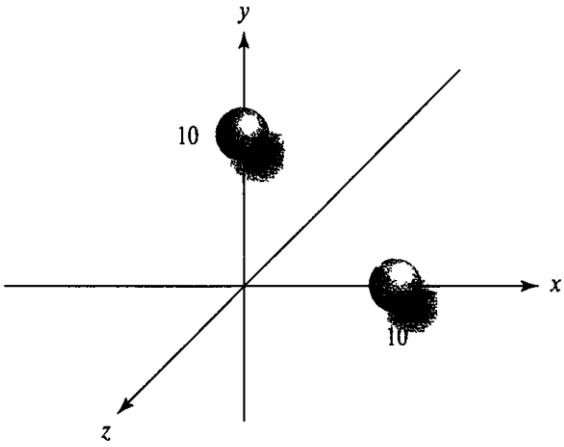


Рис.11 – Сферы, нарисованные на осях x и y

```
// Пройти на 10 единиц вдоль оси y
glTranslatef (0.0f, 10.0f, 0.0f);
// Нарисовать первую сферу
glutSolidSphere (1.0f, 15, 15);
// Пройти на 10 единиц вдоль оси x
glTranslatef (10.0f, 0.0f, 0.0f);
//Нарисовать вторую сферу
glutSolidSphere (1.0f);
```

Однако, учтите, что все вызовы glTranslate накапливаются в матрице наблюдения модели, поэтому второй вызов задает трансляцию в 10 единиц вдоль оси x от положения, полученного при предыдущей

трансляции в положительном направлении оси y. В результате получается изображение, показанное на рис. 12.

Чтобы решить эту проблему, можно вызвать дополнительно функцию `glTranslate` и пройти 10 единиц в отрицательном направлении оси y, но так будет трудно кодировать и отлаживать сложные сцены (не говоря уже об увеличении нагрузки на процессор, связанной с математикой преобразований). Гораздо проще необходимо обновить матрицу наблюдения модели до известного состояния – в нашем случае с центром в начале системы координат наблюдения.

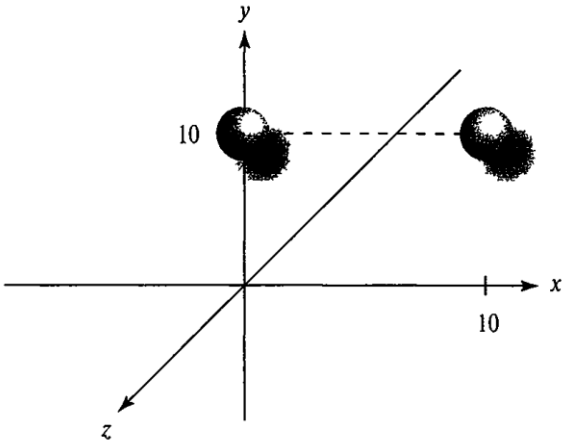


Рис.12 – Результат последовательных трансляций

Подобное обновление заключается в загрузке единичной матрицы вместо текущей матрицы наблюдения модели. Единичная матрица указывает, что преобразование не происходит, сообщая, что все заданные координаты указаны в координатах системы наблюдения. Единичная матрица содержит все нули, исключая диагональ, где расположены единицы. При умножении такой матрицы на любую матрицу вершины результат не отличается от исходной матрицы. Соответствующий пример показан на рис.13.

Загрузка единичной матрицы означает, что вершины не преобразовываются. По сути, таким образом обновляется матрицу наблюдения

модели, она возвращается к состоянию, соответствующему положению в начале координат.

$$\begin{bmatrix} 8{,}0 & 4{,}5 & -2{,}0 & 1{,}0 \end{bmatrix} \begin{bmatrix} 1{,}0 & 0 & 0 & 0 \\ 0 & 1{,}0 & 0 & 0 \\ 0 & 0 & 1{,}0 & 0 \\ 0 & 0 & 0 & 1{,}0 \end{bmatrix} = \begin{bmatrix} 8{,}0 & 4{,}5 & -2{,}0 & 1{,}0 \end{bmatrix}$$

Рис.13 – Умножение вершины на единичную матрицу дает ту же матрицу вершины

Выполнение следующих двух строк кода загружает единичную матрицу в матрицу наблюдения модели:

```
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
```

В первой строке указывается, что текущей обрабатываемой матрицей является матрица наблюдения модели. После того, как установленатекущую обрабатываемую матрицу (матрицу, к которой применяются последующие матричные функции), она остается активной, пока вы ее не измените. Во второй строке вместо текущей матрицы (в данном случае матрицы наблюдения модели) загружается единичная.

Приведем теперь код, дающий результат, показанный на рис. 11.

```
      //Матрица наблюдения модели становится текущей и
обновляется
      glMatrixMode (GL_MODELVIEW);
      glLoadIdentity ();
      //Проходим 10 единиц по положительному
направлению оси y
      glTranslatef (0.0f, 10.0f, 0.0f);
      //Рисуем первую сферу
      glutSolidSphere (1.0f, 15,15);
      //Снова обновляем матрицу наблюдения модели
      glLoatIdentity ();
      //Проходим 10 единиц по положительному
направлению оси x
      glTranslatef (10.0f, 0.0f, 0.0f);
      //Рисуем вторую сферу
```

```
glutSolidSphere (1.0f, 15, 15);
```

Стеки матриц

Обновление матрицы наблюдения модели (превращение ее в единичную) перед помещением на сцену нового объекта не всегда желательно. Часто требуется сохранить текущее состояние преобразования, а затем восстановить его после размещения нескольких объектов. Такой подход удобнее, когда в качестве преобразования наблюдения вы используете исходную преобразованную матрицу наблюдения модели (а следовательно, уже не привязаны к началу координат).

Чтобы облегчить подобную процедуру, OpenGL поддерживает стек матриц, как для матрицы наблюдения модели, так и для матрицы проекции. Стек матриц действует так же, как привычный стек программы. Можно поместить текущую матрицу в стек, запомнив ее, а затем менять текущую матрицу. Выталкивание матрицы из стека восстанавливает ее. Принцип действия стека иллюстрируется на рис. 14.

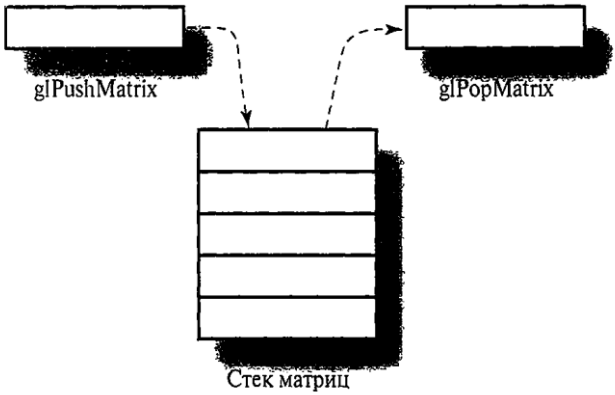


Рис.14 – Стек матриц в действии

Использование проекций

Во всех приведенных выше примерах для размещения в объеме наблюдения точки наблюдения и объектов использовалась матрица на-

блюдения модели. В действительности размер и форму наблюдаемого объема задает матрица проекции.

При создании простой параллельный наблюдаемый объем, использовалась функция `glOrtho` для указания ближней и дальней, левой и правой, верхней и нижней координат отсечения. При загрузке в качестве матрицы проекции единичной матрицы указывается, что плоскости отсечения проходят через точки +1 и - 1 на каждой оси. Если не загрузили матрицу перспективной проекции, сама по себе матрица проекции не корректирует масштаб или проекцию.

Следующая пара программ `ORTHO` и `RESPECT` не рассматривается подробно с точки зрения их исходного кода. В примерах применяется освещение и затенение (которые мы еще не рассматривали), подчеркивающие различия между ортографической и перспективной проекциями.

Нетривиальное умножение матриц

Описанные высокоуровневые преобразования (поворота, масштабирования и трансляции) прекрасно подходят для решения многих простых задач. Реальные же мощь и гибкость получат только те, кто потрудится понять непосредственное использование матриц. Это не так сложно, как кажется, но вначале нужно понять магию этих 16 чисел, составляющих матрицу преобразования размером 4 x 4.

OpenGL представляет матрицу 4 x 4 не как двухмерный массив чисел с плавающей запятой, а как единый массив 16 значений с плавающей запятой. Подход, принятый в OpenGL, отличается от того, что применяется во многих математических библиотеках, "понимающих" двухмерные массивы. Например, из приведенных ниже двух вариантов OpenGL предпочитает первый.

```
      GLfloat matrix[16];  // Прекрасная матрица с точки зрения OpenGL
      GLfloat matrix[4][4];  // Вариант популярный, но не сильно
      //эффективный в OpenGL
```

OpenGL может использовать второй вариант, но первый гораздо эффективнее. Данные 16 элементов представляют матрицу 4x4, как показано на рис.15. Когда элементы массива последовательно проходятся по столбцам матрицы, такой порядок называется развертыванием матрицы по столбцам. В памяти компьютера представление двухмерного массива как матрицы 4 x4 (второй вариант в приведенном выше коде) — это развертывание матрицы по строкам. Говоря математическими терминами, чтобы перевести одну матрицу в другую, ее нужно транспонировать.

$$\begin{bmatrix} a_0 & a_4 & a_8 & a_{12} \\ a_1 & a_5 & a_9 & a_{13} \\ a_2 & a_6 & a_{10} & a_{14} \\ a_3 & a_7 & a_{11} & a_{15} \end{bmatrix}$$

Рис.15 – Развертывание матрицы по столбцам

Эти 16 значений представляют определенную точку в пространстве и ориентацию трех осей относительно системы наблюдения (помните, что это стационарная, неизменяемая система координат, о которой говорили ранее). Интерпретировать эти числа совсем не сложно. Четыре столбца представляют четырехэлементный вектор. Сфокусируем внимание только на трех первых элементах этих векторов. Четвертый вектор-столбец содержит значения x, y и z преобразованной системы координат. При действии функции `glTranslate` на единичную матрицу все, что она делает, — это помещает значения x, y и z в двенадцатую, тринадцатую и четырнадцатую позиции матрицы.

Первые три столбца являются просто направленными векторами, которые представляют ориентацию (здесь векторы указывают направление) осей x, y и z в пространстве. В большинстве случаев эти три вектора всегда образуют друг с другом угол 90°. На рис. 16 показана матрица преобразования 4x4 с обозначенными векторами-столбцами. Обратите внимание на последнюю строку матрицы кроме последней единицы, все элементы равны нулю.

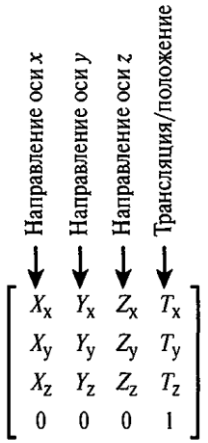


Рис.16 – Как матрица 4x4 представляет положение и ориентацию в трехмерном пространстве

Самым впечатляющим моментом является то, что, если имеется матрица 4x4, которая содержит положение и ориентацию другой системы координат, то, умножив эту матрицу на вершину (матрицу- или вектор-столбец), вы получите новую вершину, преобразованную в новую систему координат. Это означает, что положение в пространстве и любую желаемую ориентацию можно единственным образом определить матрицей 4x4, и, умножив все вершины объекта на эту матрицу, преобразуется весь объект в данную точку пространства с данной ориентацией!

Создание в OpenGL движения с использованием камер и актеров

Чтобы описать положение и ориентацию любого объекта на трехмерной сцене, можно использовать одну матрицу 4x4, представляющую его преобразование. Тем не менее работа непосредственно с матрицами может быть несколько неудобной, поэтому программисты всегда ищут способы более лаконичного представления положения и

ориентации в пространстве. Такие фиксированные объекты, как ландшафт, часто не изменяются, и их вершины обычно точно задают, где в пространстве должны рисоваться геометрические объекты. Объекты, которые движутся по сцене, называются актерами, по аналогии о актерами на реальной сцене.

Актеры имеют собственные преобразования, и часто другие актеры преобразовываются не только относительно внешней системы координат (координаты системы наблюдения), но и относительно других актеров. Говорят, что каждый актер со своими преобразованиями имеет собственную систему отсчета или локальную систему координат объекта. Часто (в геометрических проверках, не связанных с визуализацией) полезно переходить из локальной системы во внешнюю и обратно.

Управление камерой

В OpenGL в действительности нет никакого преобразования камеры. Можно использовать концепцию камеры как полезную метафору, помогающую управлять точкой зрения в определенной трехмерной среде. Если вообразить камеру как объект, расположенный в некоторой точке пространства и имеющий ориентацию, стагнет ясно, что в текущей системе отсчета как актеров, так и камеру можно представить в трехмерной среде.

Чтобы применить преобразование камеры, берем преобразование актеров камеры и преображаем его так, чтобы движение камеры назад было эквивалентно движению всего мира вперед. Подобным образом, поворот налево эквивалентен вращению всего мира вправо. Чтобы визуализировать данную сцену, мы обычно принимаем подход, схематически представленный на рис. 17.

Библиотека GLU содержит функцию, которая формирует преобразование камеры на основе данных, записанных в структуре системы отсчета.

```
void gluLookAt(GLdouble eyex, GLdouble eyey,
GLdouble eyez, GLdouble centerx, GLdouble centery,
```

```
GLdouble centerz, GLdouble upx, GLdouble upy, GLdouble
upz);
```

В качестве аргумента функция принимает положение глаза, точку, расположенную непосредственно перед глазом, и направление вверх. Библиотека `glTools` также содержит сокращенную функцию, выполняющую эквивалентное действие с использованием системы отсчета.

```
void gltApplyCameraTransform(GLTFrame *pCamera);
```

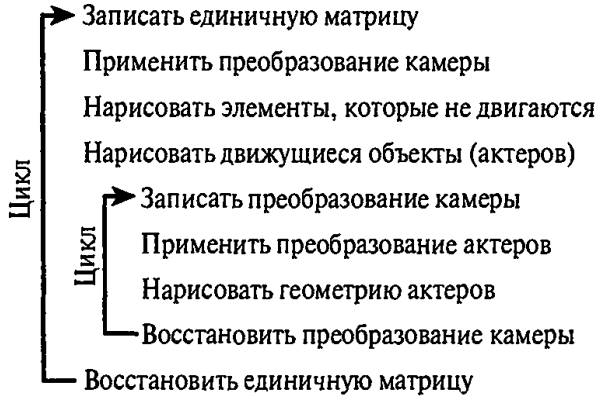


Рис.17 – Типичный цикл визуализации трехмерной среды

Примечание по опросу клавиатуры

Движение камеры в ответ на клавиатурные сообщения может иногда дать меньше, чем наиболее гладкую из возможных анимацию. Это объясняется тем, что скорость нажатия на клавиши обычно не превышает 20 раз в секунду. Для получения наилучших результатов визуализация должна выполняться со скоростью не меньше, чем 30 кадров в секунду (оптимальный вариант – 60 кадров/с), один раз опрашивая клавиатуру для каждого кадра анимации.

ЗАДАЧИ И ПОРЯДОК ВЫПОЛНЕНИЯ РАБОТЫ

Пример ядра

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 1 показана функция `RenderScene` этого примера, а результат выполнения программы ATOM демонстрируется на рис. 18.

Листинг 1 - Функция `RenderScene` из программы ATOM

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

// Величина поворота
static GLfloat xRot = 0.0f;
static GLfloat yRot = 0.0f;


// 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, 0, 0);
        glutSolidSphere(10.0f, 15, 15);

        // Желтые электроны
        glColor3ub(255,255,0);

        // Орбита первого электрона
        // Записываем преобразование наблюдения
        glPushMatrix();

        // Поворачиваем на угол поворота
        glRotatef(fElect1, 0.0f, 1.0f, 0.0f);

        // Трансляция элемента от начала координат на
орбиту
        glTranslatef(90.0f, 0.0f, 0.0f);

        // Рисуем электрон
        glutSolidSphere(6.0f, 15, 15);


        // 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 SpecialKeys(int key, int x, int y)
      {
      if(key == GLUT_KEY_UP)
            xRot-= 5.0f;

      if(key == GLUT_KEY_DOWN)
            xRot += 5.0f;

      if(key == GLUT_KEY_LEFT)
            yRot -= 5.0f;

      if(key == GLUT_KEY_RIGHT)
            yRot += 5.0f;

      if(key > 356.0f)
            xRot = 0.0f;

      if(key < -1.0f)
            xRot = 355.0f;

      if(key > 356.0f)
            yRot = 0.0f;

      if(key < -1.0f)
            yRot = 355.0f;

    // Перерисовывает сцену с новыми координатами
      glutPostRedisplay();
      }

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);
      glutSpecialFunc(SpecialKeys);
      glutDisplayFunc(RenderScene);
        glutTimerFunc(500, TimerFunc, 1);
      SetupRC();
      glutMainLoop();

      return 0;
      }
```

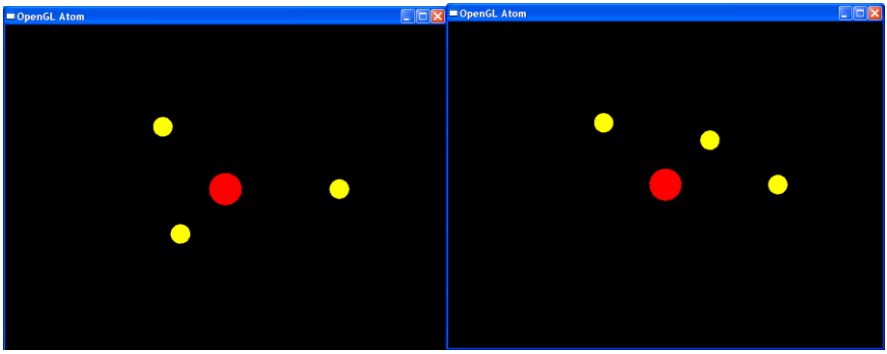


Рис.18 – Результат выполнения программы ATOM

Разберем код, отвечающий за расположение первого электрона. Итак, в первой строке записывается текущая матрица наблюдения модели (в стек заносится текущее преобразование).

```
      // Орбита первого электрона
      // Записываем преобразование наблюдения
      glPushMatrix();
```

Теперь система координат кажется повернутой вокруг оси y на угол `fElect1`.

```
      // Поворот на угол поворота
      glRotatef(fElect1, 0.0f, 1.0f, 0.0f);
```

Путем трансляции повернутой системы координат рисуется электрон.

```
        //Трансляция от начала координат на орбиту
        glTranslatef(90.0f, 0.0f, 0.0f);
```

Затем рисуется электрон (сплошная сфера), и восстанавливается (извлекается из стека) матрица наблюдения модели.

```
        //Рисуем электрон
        glutSolidSphere(6.0f, 15, 15);
        //Восстанавливаем преобразование наблюдения
        glPopMatrix();
```

Остальные электроны размещаются аналогично.

Ортографические проекции

Ортографическая проекция, использованная в большинстве рассмотренных ранее примеров, представлена правильным кубом, который со всех сторон имеет вид квадрата. Логическая ширина одинакова на передней, задней, верхней, нижней, левой и правой сторонах. В результате получается параллельная проекция, которая полезна для рисования специфических объектов, при наблюдении которых со стороны ракурс не учитывается. Это удобно, например, в автоматизированном проектировании, представлении такой двухмерной графики, как текст, или в архитектурных рисунках, где желательно представить точные размеры.

Листинг 2 - Полный листинг простой программы ORTHO

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

// Величина поворота
static GLfloat xRot = 0.0f;
static GLfloat yRot = 0.0f;

// Вызывается при изменении размеров окна
void ChangeSize(GLsizei w, GLsizei h)
    {
    GLfloat nRange = 120.0f;
```

```
    // Предотвращает деление на нуль
    if(h == 0)
        h = 1;

    // Устанавливает размеры поля просмотра равны
размерам окна    glViewport(0, 0, w, h);

    // Устанавливаем перспективную систему координат
    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();
    }

// Эта функция выполняет необходимую инициализацию в
контексте
// визуализации. В данном случае устанавливается
// и инициализируется освещение сцены.
void SetupRC()
    {
    // Параметры света
    GLfloat  whiteLight[] = { 0.45f, 0.45f, 0.45f, 1.0f
};
    GLfloat  sourceLight[] = { 0.25f, 0.25f, 0.25f,
1.0f };
```

```
    GLfloat  lightPos[] = { -50.f, 25.0f, 250.0f, 0.0f
};

    glEnable(GL_DEPTH_TEST);  // Удаление скрыьых
поверхностей
    glFrontFace(GL_CCW);      //Многоугольники с обходом
                              против //часовой стрелки
                              направлены наружу

    glEnable(GL_CULL_FACE);   // Расчеты внутри самолета
                              не
                              // выполняются
    // Активизация освещения
    glEnable(GL_LIGHTING);

    // Устанавливается и активизируется источник света
0
    glLightModelfv(GL_LIGHT_MODEL_AMBIENT,whiteLight);
    glLightfv(GL_LIGHT0,GL_AMBIENT,sourceLight);
    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 );
    }

// Реагирует на клавиши со стрелками, двигая систему
отсчета камеры
void SpecialKeys(int key, int x, int y)
      {
      if(key == GLUT_KEY_UP)
```

```
            xRot-= 5.0f;

      if(key == GLUT_KEY_DOWN)
            xRot += 5.0f;

      if(key == GLUT_KEY_LEFT)
            yRot -= 5.0f;

      if(key == GLUT_KEY_RIGHT)
            yRot += 5.0f;

        xRot = (GLfloat)((const int)xRot % 360);
        yRot = (GLfloat)((const int)yRot % 360);

      // Обновляем окно
      glutPostRedisplay();
      }


// Вызывается для рисования сцены
void RenderScene(void)
    {
    float fZ,bZ;

    // Очищаем окно текущим цветом очистки
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    fZ = 100.0f;
    bZ = -100.0f;

    // Записываем состояние матрицы и выполняем поворот
    glPushMatrix();
    glRotatef(xRot, 1.0f, 0.0f, 0.0f);
    glRotatef(yRot, 0.0f, 1.0f, 0.0f);

    // В качестве текущего цвета рисования задает
    красный    glColor3f(1.0f, 0.0f, 0.0f);
```

```
    // Лицевая часть
////////////////////////////////
    glBegin(GL_QUADS);
        // прямо вверх оси z
        glNormal3f(0.0f, 0.0f, 1.0f);

        // Левая сторона
        glVertex3f(-50.0f, 50.0f, fZ);
        glVertex3f(-50.0f, -50.0f, fZ);
        glVertex3f(-35.0f, -50.0f, fZ);
        glVertex3f(-35.0f,50.0f,fZ);

        // Правая сторона
        glVertex3f(50.0f, 50.0f, fZ);
        glVertex3f(35.0f, 50.0f, fZ);
        glVertex3f(35.0f, -50.0f, fZ);
        glVertex3f(50.0f,-50.0f,fZ);

        // Верх
        glVertex3f(-35.0f, 50.0f, fZ);
        glVertex3f(-35.0f, 35.0f, fZ);
        glVertex3f(35.0f, 35.0f, fZ);
        glVertex3f(35.0f, 50.0f,fZ);

        // Низ
        glVertex3f(-35.0f, -35.0f, fZ);
        glVertex3f(-35.0f, -50.0f, fZ);
        glVertex3f(35.0f, -50.0f, fZ);
        glVertex3f(35.0f, -35.0f,fZ);

        // Верхняя длинная часть
////////////////////////////

        //Все нормали для верхней части вверх оси Y

        glNormal3f(0.0f, 1.0f, 0.0f);
        glVertex3f(-50.0f, 50.0f, fZ);
        glVertex3f(50.0f, 50.0f, fZ);
```

```
        glVertex3f(50.0f, 50.0f, bZ);
        glVertex3f(-50.0f,50.0f,bZ);

        // Нижняя секция
        glNormal3f(0.0f, -1.0f, 0.0f);
        glVertex3f(-50.0f, -50.0f, fZ);
        glVertex3f(-50.0f, -50.0f, bZ);
        glVertex3f(50.0f, -50.0f, bZ);
        glVertex3f(50.0f, -50.0f, fZ);

        // Левая секция
        glNormal3f(1.0f, 0.0f, 0.0f);
        glVertex3f(50.0f, 50.0f, fZ);
        glVertex3f(50.0f, -50.0f, fZ);
        glVertex3f(50.0f, -50.0f, bZ);
        glVertex3f(50.0f, 50.0f, bZ);

        // Правая секция
        glNormal3f(-1.0f, 0.0f, 0.0f);
        glVertex3f(-50.0f, 50.0f, fZ);
        glVertex3f(-50.0f, 50.0f, bZ);
        glVertex3f(-50.0f, -50.0f, bZ);
        glVertex3f(-50.0f, -50.0f, fZ);
    glEnd();

    glFrontFace(GL_CW);       // элементы с обходом по
часовой стрелке
                              // смотрят наружу


    glBegin(GL_QUADS);
        // Задняя часть
        // Параллельно оси z
        glNormal3f(0.0f, 0.0f, -1.0f);

        // Левая сторона
        glVertex3f(-50.0f, 50.0f, bZ);
        glVertex3f(-50.0f, -50.0f, bZ);
```

```
        glVertex3f(-35.0f, -50.0f, bZ);
        glVertex3f(-35.0f,50.0f,bZ);

        // Правая сторона
        glVertex3f(50.0f, 50.0f, bZ);
        glVertex3f(35.0f, 50.0f, bZ);
        glVertex3f(35.0f, -50.0f, bZ);
        glVertex3f(50.0f,-50.0f,bZ);

        // Верх
        glVertex3f(-35.0f, 50.0f, bZ);
        glVertex3f(-35.0f, 35.0f, bZ);
        glVertex3f(35.0f, 35.0f, bZ);
        glVertex3f(35.0f, 50.0f,bZ);

        // Низ
        glVertex3f(-35.0f, -35.0f, bZ);
        glVertex3f(-35.0f, -50.0f, bZ);
        glVertex3f(35.0f, -50.0f, bZ);
        glVertex3f(35.0f, -35.0f,bZ);

        // Внутрення часть
////////////////////////////
      glColor3f(0.75f, 0.75f, 0.75f);

        // Нормаль указывает на ось Y
        glNormal3f(0.0f, 1.0f, 0.0f);
        glVertex3f(-35.0f, 35.0f, fZ);
        glVertex3f(35.0f, 35.0f, fZ);
        glVertex3f(35.0f, 35.0f, bZ);
        glVertex3f(-35.0f,35.0f,bZ);

        // Нижняя часть
        glNormal3f(0.0f, 1.0f, 0.0f);
        glVertex3f(-35.0f, -35.0f, fZ);
        glVertex3f(-35.0f, -35.0f, bZ);
        glVertex3f(35.0f, -35.0f, bZ);
        glVertex3f(35.0f, -35.0f, fZ);
```

```
        // Левая часть
        glNormal3f(1.0f, 0.0f, 0.0f);
        glVertex3f(-35.0f, 35.0f, fZ);
        glVertex3f(-35.0f, 35.0f, bZ);
        glVertex3f(-35.0f, -35.0f, bZ);
        glVertex3f(-35.0f, -35.0f, fZ);

        // Правая часть
        glNormal3f(-1.0f, 0.0f, 0.0f);
        glVertex3f(35.0f, 35.0f, fZ);
        glVertex3f(35.0f, -35.0f, fZ);
        glVertex3f(35.0f, -35.0f, bZ);
        glVertex3f(35.0f, 35.0f, bZ);
    glEnd();

    glFrontFace(GL_CCW); // полигоны с обходом против
    часовой стрелки        направлены наружу

 // Восстанавливается состояние матрицы
    glPopMatrix();

    // Буфер-обмена
    glutSwapBuffers();
    }



int main(int argc, char *argv[])
    {
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB |
GLUT_DEPTH);
    glutInitWindowSize(800, 600);
    glutCreateWindow("Orthographic Projection");
    glutReshapeFunc(ChangeSize);
    glutSpecialFunc(SpecialKeys);
    glutDisplayFunc(RenderScene);
```

```
SetupRC();
glutMainLoop();

return 0;
}
```

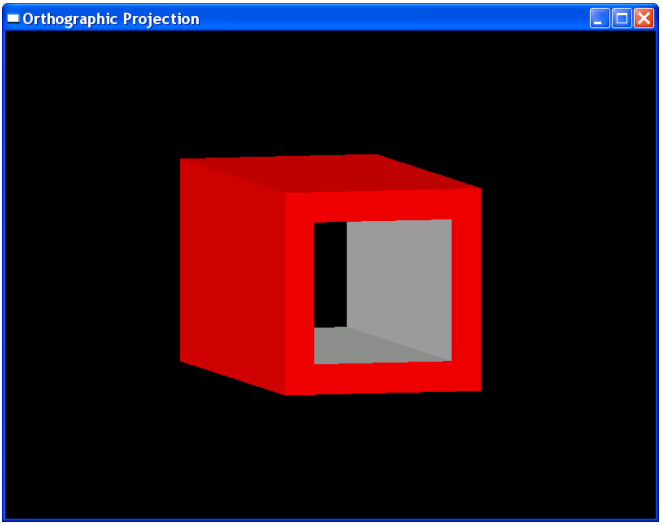


Рис.19 – Полная квадратная труба, показанная с помощью ортографической проекции

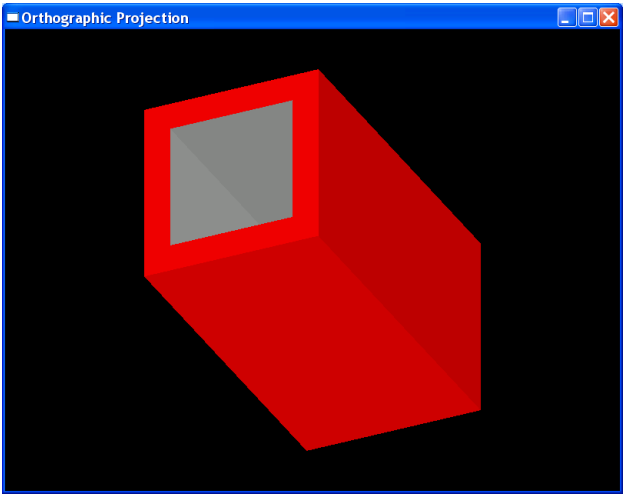


Рис.20 – Вид квадратной трубы сбоку, на котором видна ее длина

На рис. 21 показано, что видно непосредственно в торец трубы. Поскольку труба не сходится на расстоянии, изображение не соответствует тому, что можно наблюдать в реальной жизни. Чтобы учесть перспективу, нужна перспективная проекция.

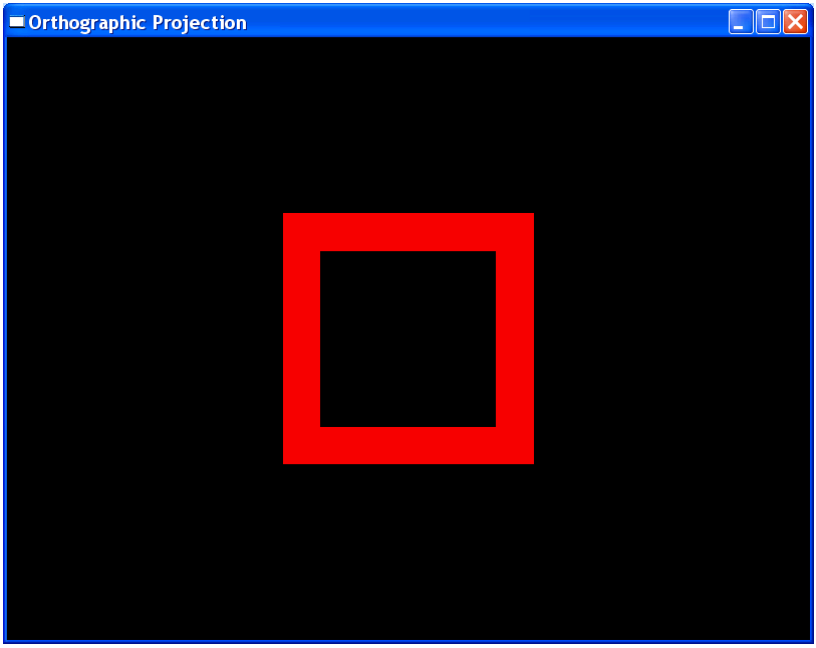


Рис.21 – Наблюдение торца трубы

Перспективная проекция

На перспективной проекции объекты, удаленные от наблюдателя, сокращаются и сжимаются с помощью перспективного деления. Ширина задней части наблюдаемого объема не равна ширине передней после проектирования на экран. Таким образам, объект с одинаковыми физическими размерами кажется больше вблизи передней части наблюдаемого объема, чем вблизи задней.

Геометрическая фигура, использованная в следующем примере, называется усеченной пирамидой. Усеченная пирамида — это усеченный фрагмент пирамиды, наблюдаемый со стороны узкого конца в направлении широкого. Пример усеченной пирамиды с обозначенным наблюдателем показан на рис. 22.

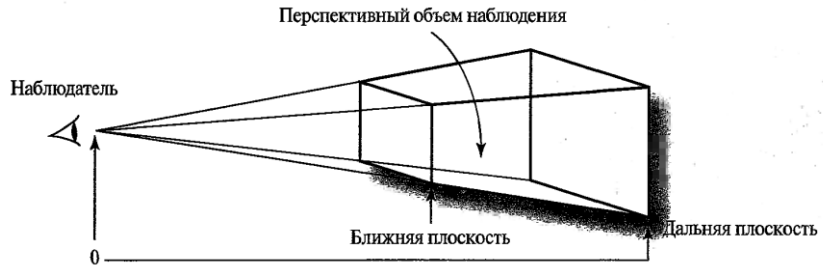


Рис.22 – Перспективная проекция, определенная усеченной пирамидой

Усеченную пирамиду можно определить с помощью функции `glFrustum`. Ее параметрами являются координаты и расстояния между передней и задней отсекающими плоскостями. Однако функция `glFrustum` не совсем понятна интуитивно с точки зрения задания проекции для получения желаемого эффекта. Иногда легче использовать вспомогательную функцию `gluPerspective`.

```
void gluPerspective(GLdouble fovy, GLdouble aspect, GLdouble zNear, GLdouble zFar);
```

Параметрами функции `gluPerspective` является угол обзора, характеристическое отношение высоты к ширине и расстояния до ближней и дальней плоскостей отсечения (см. рис. 23). Чтобы определить характеристическое отношение, высота (w) поля просмотра делится на его ширину (h).

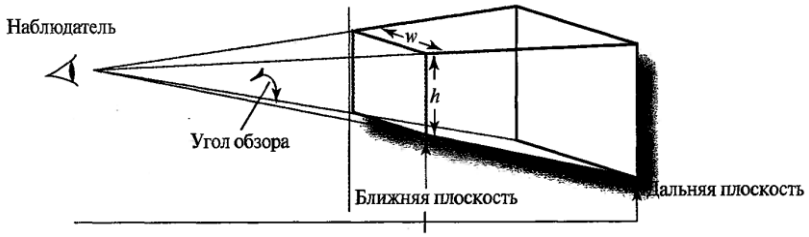


Рис.23 – Усеченная пирамида, определенная функцией `gluPerspective`

В листинге 3 показано, как ортографическая проекция из предыдущего примера меняется на перспективную. Учет ракурса повышает реализм использованной ранее ортографической проекции квадратной трубы (см. рис. 24, 25 и 26). Единственным сделанным существенным изменением в коде листинга 2 является замена `gluOrtho2D` на `gluPerspective`.

Листинг 3 - Установка перспективной проекции в программе PESPECT

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

// Величина поворота
static GLfloat xRot = 0.0f;
static GLfloat yRot = 0.0f;


// Меняет наблюдаемый объем и поле просмотра
// Вызывается при изменении размеров окна
void ChangeSize(GLsizei w, GLsizei h)
    {
    GLfloat fAspect;

    // Предотвращает деление на ноль
    if(h == 0)
        h = 1;
```

```
    // Устанавливает поле просмотра по размерам окна
    glViewport(0, 0, w, h);

    fAspect = (GLfloat)w/(GLfloat)h;

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

    // Генерирует перспективную проекцию
    gluPerspective(60.0f, fAspect, 1.0, 400.0);

    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    }


// Эта функция выполняет необходимую инициализацию в
контексте
// визуализации. В данном случае устанавливается
// и инициализируется освещение сцены.
void SetupRC()
    {
    // Параметры освещения и координаты
    GLfloat  whiteLight[] = { 0.45f, 0.45f, 0.45f, 1.0f
};
    GLfloat  sourceLight[] = { 0.25f, 0.25f, 0.25f,
1.0f };
    GLfloat  lightPos[] = { -50.f, 25.0f, 250.0f, 0.0f
};

    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

    // Активизация освещения
    glEnable(GL_LIGHTING);

    // Устанавливается и активизируется источник света 0
    glLightModelfv(GL_LIGHT_MODEL_AMBIENT,whiteLight);
    glLightfv(GL_LIGHT0,GL_AMBIENT,sourceLight);
    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 );
    }

// Реагирует на клавиши со стрелками, двигая систему
отсчета камеры
void SpecialKeys(int key, int x, int y)
      {
      if(key == GLUT_KEY_UP)
            xRot-= 5.0f;

      if(key == GLUT_KEY_DOWN)
            xRot += 5.0f;

      if(key == GLUT_KEY_LEFT)
            yRot -= 5.0f;

      if(key == GLUT_KEY_RIGHT)
            yRot += 5.0f;
```

```
        xRot = (GLfloat)((const int)xRot % 360);
        yRot = (GLfloat)((const int)yRot % 360);

      // Обновляем окно
      glutPostRedisplay();
      }


// Вызывается для рисования сцены
void RenderScene(void)
    {
    float fZ,bZ;

    // Очищаем окно текущим цветом заливки
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    fZ = 100.0f;
    bZ = -100.0f;

    // Записывается состояние матрицы и выполняются
повороты
    glPushMatrix();
    glTranslatef(0.0f, 0.0f, -300.0f);
    glRotatef(xRot, 1.0f, 0.0f, 0.0f);
    glRotatef(yRot, 0.0f, 1.0f, 0.0f);

    // В качестве текущего цвета рисования задает
красный
    glColor3f(1.0f, 0.0f, 0.0f);

    // Лицевая часть
////////////////////////////////
    glBegin(GL_QUADS);
        // прямо вверх оси z
        glNormal3f(0.0f, 0.0f, 1.0f);

        // Левая сторона
```

```
        glVertex3f(-50.0f, 50.0f, fZ);
        glVertex3f(-50.0f, -50.0f, fZ);
        glVertex3f(-35.0f, -50.0f, fZ);
        glVertex3f(-35.0f,50.0f,fZ);

        // Правая сторона
        glVertex3f(50.0f, 50.0f, fZ);
        glVertex3f(35.0f, 50.0f, fZ);
        glVertex3f(35.0f, -50.0f, fZ);
        glVertex3f(50.0f,-50.0f,fZ);

        // Верх
        glVertex3f(-35.0f, 50.0f, fZ);
        glVertex3f(-35.0f, 35.0f, fZ);
        glVertex3f(35.0f, 35.0f, fZ);
        glVertex3f(35.0f, 50.0f,fZ);

        // Низ
        glVertex3f(-35.0f, -35.0f, fZ);
        glVertex3f(-35.0f, -50.0f, fZ);
        glVertex3f(35.0f, -50.0f, fZ);
        glVertex3f(35.0f, -35.0f,fZ);

        // Верхняя длинная
часть//////////////////////////
        // Нормаль указывает на ось Y
        glNormal3f(0.0f, 1.0f, 0.0f);
        glVertex3f(-50.0f, 50.0f, fZ);
        glVertex3f(50.0f, 50.0f, fZ);
        glVertex3f(50.0f, 50.0f, bZ);
        glVertex3f(-50.0f,50.0f,bZ);

        // Низ
        glNormal3f(0.0f, -1.0f, 0.0f);
        glVertex3f(-50.0f, -50.0f, fZ);
        glVertex3f(-50.0f, -50.0f, bZ);
        glVertex3f(50.0f, -50.0f, bZ);
        glVertex3f(50.0f, -50.0f, fZ);
```

```
    // Левая сторона
    glNormal3f(1.0f, 0.0f, 0.0f);
    glVertex3f(50.0f, 50.0f, fZ);
    glVertex3f(50.0f, -50.0f, fZ);
    glVertex3f(50.0f, -50.0f, bZ);
    glVertex3f(50.0f, 50.0f, bZ);

    // Правая сторона
    glNormal3f(-1.0f, 0.0f, 0.0f);
    glVertex3f(-50.0f, 50.0f, fZ);
    glVertex3f(-50.0f, 50.0f, bZ);
    glVertex3f(-50.0f, -50.0f, bZ);
    glVertex3f(-50.0f, -50.0f, fZ);
glEnd();

glFrontFace(GL_CW);       // элементы с обходом по
                          часовой стрелке //смотрят
                          наружу


glBegin(GL_QUADS);
    // Задняя часть
    // Нормали направлены прямо вверх оси z
    glNormal3f(0.0f, 0.0f, -1.0f);

    // Левая сторона
    glVertex3f(-50.0f, 50.0f, bZ);
    glVertex3f(-50.0f, -50.0f, bZ);
    glVertex3f(-35.0f, -50.0f, bZ);
    glVertex3f(-35.0f,50.0f,bZ);

    // Правая сторона
    glVertex3f(50.0f, 50.0f, bZ);
    glVertex3f(35.0f, 50.0f, bZ);
    glVertex3f(35.0f, -50.0f, bZ);
    glVertex3f(50.0f,-50.0f,bZ);
```

```
        // Верх
        glVertex3f(-35.0f, 50.0f, bZ);
        glVertex3f(-35.0f, 35.0f, bZ);
        glVertex3f(35.0f, 35.0f, bZ);
        glVertex3f(35.0f, 50.0f,bZ);

        // Низ
        glVertex3f(-35.0f, -35.0f, bZ);
        glVertex3f(-35.0f, -50.0f, bZ);
        glVertex3f(35.0f, -50.0f, bZ);
        glVertex3f(35.0f, -35.0f,bZ);

        // Внутренняя часть
////////////////////////////
      glColor3f(0.75f, 0.75f, 0.75f);

        // Нормаль направлена вверх оси Y
        glNormal3f(0.0f, 1.0f, 0.0f);
        glVertex3f(-35.0f, 35.0f, fZ);
        glVertex3f(35.0f, 35.0f, fZ);
        glVertex3f(35.0f, 35.0f, bZ);
        glVertex3f(-35.0f,35.0f,bZ);

        // Низ
        glNormal3f(0.0f, 1.0f, 0.0f);
        glVertex3f(-35.0f, -35.0f, fZ);
        glVertex3f(-35.0f, -35.0f, bZ);
        glVertex3f(35.0f, -35.0f, bZ);
        glVertex3f(35.0f, -35.0f, fZ);

        //Левая сторона
        glNormal3f(1.0f, 0.0f, 0.0f);
        glVertex3f(-35.0f, 35.0f, fZ);
        glVertex3f(-35.0f, 35.0f, bZ);
        glVertex3f(-35.0f, -35.0f, bZ);
        glVertex3f(-35.0f, -35.0f, fZ);

        // Правая сторона
```

```
        glNormal3f(-1.0f, 0.0f, 0.0f);
        glVertex3f(35.0f, 35.0f, fZ);
        glVertex3f(35.0f, -35.0f, fZ);
        glVertex3f(35.0f, -35.0f, bZ);
        glVertex3f(35.0f, 35.0f, bZ);
    glEnd();

    glFrontFace(GL_CCW);      // многоугольники с
                              обходом против //часовой
                              стрелки направлены наружу


    // Восстанавливается состояние матрицы
    glPopMatrix();

    // Буфер обмена
    glutSwapBuffers();
    }



int main(int argc, char *argv[])
    {
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB |
GLUT_DEPTH);
    glutInitWindowSize(800, 600);
    glutCreateWindow("Perspective Projection");
    glutReshapeFunc(ChangeSize);
    glutSpecialFunc(SpecialKeys);
    glutDisplayFunc(RenderScene);
    SetupRC();
    glutMainLoop();

    return 0;
    }
```

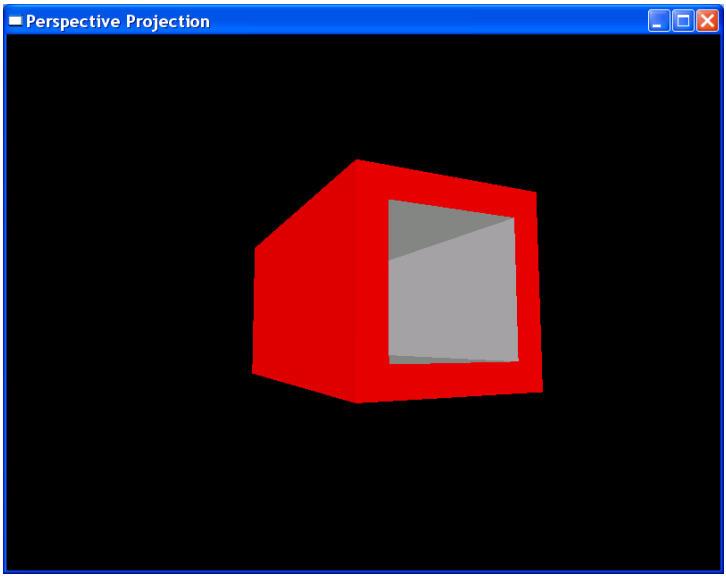


Рис.24 – Квадратная труба в перспективной проекции

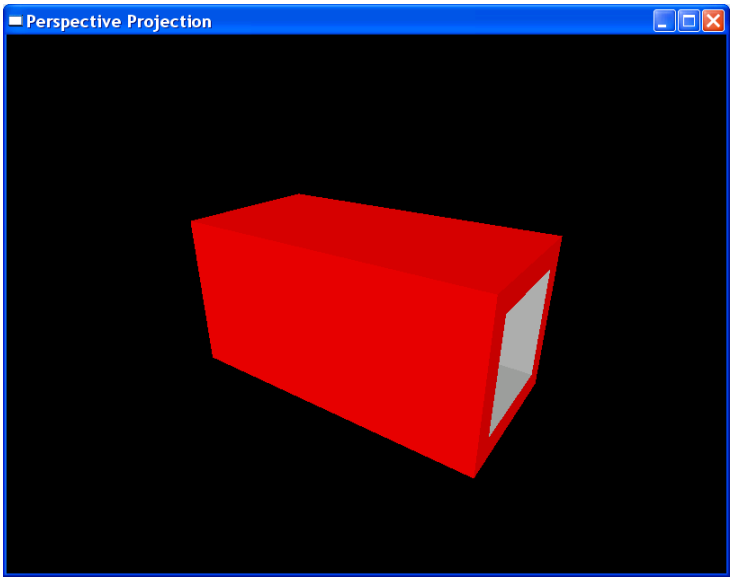


Рис.25 – Вид сбоку с учетом ракурса

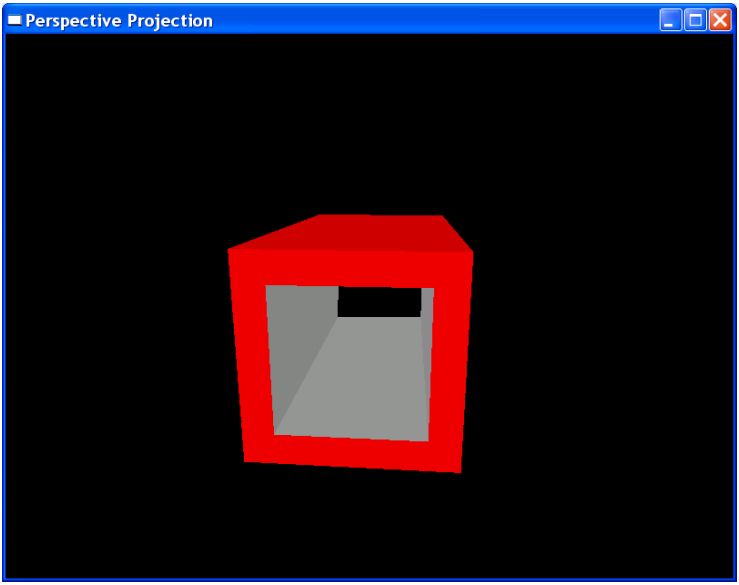


Рис.26 – Вид сквозь трубу при использовании перспективной проекции

Пример

Чтобы создать завершенный пример, демонстрирующий работу с матрицей наблюдения модели и перспективной проекцией, рассмотрим моделирование в программе SOLAR 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 на рис. 27 и 28: когда Земля расположена перед Солнцем, она кажется больше; Земля, находящаяся с противоположной стороны, выглядит меньше.

Листинг 4 - Код системы «Солнце-Земля-Луна»

```
#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
      glLightMo-
delfv(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 glCo-
lor
      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();
```

```
return 0;
}
```

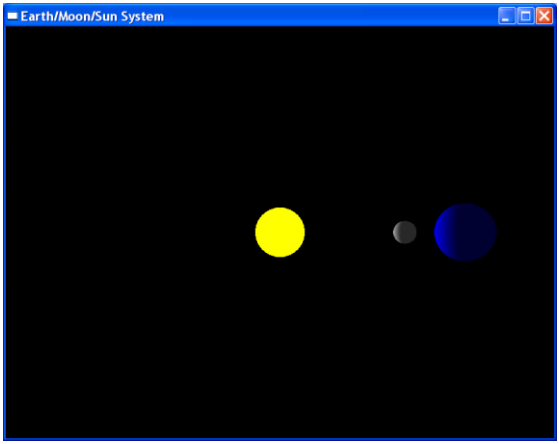


Рис.27 – Система «Солнце-Земля-Луна»: Земля находится ближе к наблюдателю

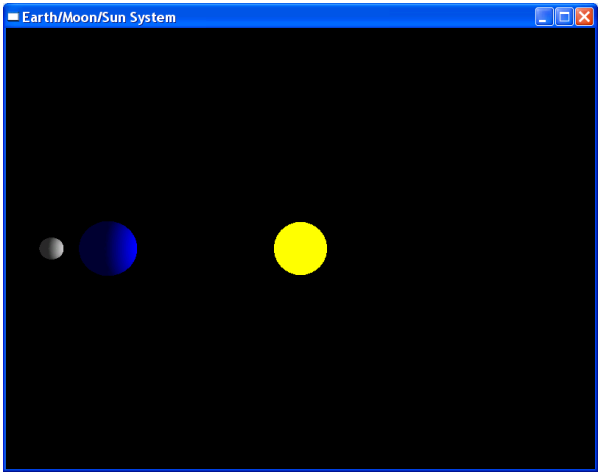


Рис.28 – Система «Солнце-Земля-Луна»: Земля находится ближе к наблюдателю

Выполнение собственных преобразований

Рассмотрим, как создавать и загружать собственные матрицы преобразований. В приведенной программе TRANSFORM мы рисуем тор перед точкой наблюдения и заставляем его вращаться на месте. За всю математику, необходимую для генерации геометрии тора, отвечает функция `DrawTorus`, принимающая в качестве аргумента матрицу преобразования 4x4, которая позже будет действовать на вершины. Чтобы преобразовать тор, создаем матрицу и действуем ею на все вершины. Основная функция визуализации, приведена в листинге 5.

Листинг 5 - Код, задающий при рисовании матрицу преобразования

```
#include "glew.h"
#include "glut.h"
#include <windows.h>
#include <math.h>
// Используемые константы
#define GLT_PI    3.14159265358979323846
#define GLT_PI_DIV_180 0.017453292519943296
#define GLT_INV_PI_DIV_180 57.2957795130823229
#define gltDegToRad(x)  ((x)*GLT_PI_DIV_180)
// Некоторые типы данных
typedef GLfloat GLTVector2[2];//Двукомпонентный вектор
с плавающей запятой
typedef GLfloat GLTVector3[3];//Трехкомпонентный вектор
с плавающей запятой
typedef GLfloat GLTVector4[4];//Четырехкомпонентный
вектор с плавающей //запятой
typedef GLfloat GLTMatrix[16];// Основноц столбец
матрицы 4x4 с плавающей //запятой
void gltLoadIdentityMatrix(GLTMatrix m)
      {
      static GLTMatrix identity = { 1.0f, 0.0f, 0.0f,
0.0f,
                                    0.0f, 1.0f, 0.0f,
0.0f,
                                    0.0f, 0.0f, 1.0f,
0.0f,
```

```
                                        0.0f, 0.0f, 0.0f,
1.0f };

      memcpy(m, identity, sizeof(GLTMatrix));
      };
void gltTransformPoint(const GLTVector3 vSrcVector,
const GLTMatrix mMatrix, GLTVector3 vOut)
    {
    vOut[0] = mMatrix[0] * vSrcVector[0] + mMatrix[4] *
vSrcVector[1] + mMatrix[8] *  vSrcVector[2] + mMa-
trix[12];
    vOut[1] = mMatrix[1] * vSrcVector[0] + mMatrix[5] *
vSrcVector[1] + mMatrix[9] *  vSrcVector[2] + mMa-
trix[13];
    vOut[2] = mMatrix[2] * vSrcVector[0] + mMatrix[6] *
vSrcVector[1] + mMatrix[10] * vSrcVector[2] + mMa-
trix[14];
    };

void gltRotationMatrix(float angle, float x, float y,
float z, GLTMatrix mMatrix)
    {
    float vecLength, sinSave, cosSave, oneMinusCos;
    float xx, yy, zz, xy, yz, zx, xs, ys, zs;

    // Если нулевой вектор проходит, то …
    if(x == 0.0f && y == 0.0f && z == 0.0f)
        {
        gltLoadIdentityMatrix(mMatrix);
        return;
        }

    // Вектор масштабирования
    vecLength = (float)sqrt( x*x + y*y + z*z );

    // Нормализованная матрица вращения
    x /= vecLength;
    y /= vecLength;
```

```
z /= vecLength;

sinSave = (float)sin(angle);
cosSave = (float)cos(angle);
oneMinusCos = 1.0f - cosSave;

xx = x * x;
yy = y * y;
zz = z * z;
xy = x * y;
yz = y * z;
zx = z * x;
xs = x * sinSave;
ys = y * sinSave;
zs = z * sinSave;

mMatrix[0] = (oneMinusCos * xx) + cosSave;
mMatrix[4] = (oneMinusCos * xy) - zs;
mMatrix[8] = (oneMinusCos * zx) + ys;
mMatrix[12] = 0.0f;

mMatrix[1] = (oneMinusCos * xy) + zs;
mMatrix[5] = (oneMinusCos * yy) + cosSave;
mMatrix[9] = (oneMinusCos * yz) - xs;
mMatrix[13] = 0.0f;

mMatrix[2] = (oneMinusCos * zx) - ys;
mMatrix[6] = (oneMinusCos * yz) + xs;
mMatrix[10] = (oneMinusCos * zz) + cosSave;
mMatrix[14] = 0.0f;

mMatrix[3] = 0.0f;
mMatrix[7] = 0.0f;
mMatrix[11] = 0.0f;
mMatrix[15] = 1.0f;
};
```

```
typedef struct{                          // Контейнер
системы отсчета
    GLTVector3 vLocation;
    GLTVector3 vUp;
    GLTVector3 vForward;
    } GLTFrame;

// Прорисовка тора(бублика), с использованием  текущей
структуры 1D для легкого затемнения света
void DrawTorus(GLTMatrix mTransform)
    {
    GLfloat majorRadius = 0.35f;
    GLfloat minorRadius = 0.15f;
    GLint   numMajor = 40;
    GLint   numMinor = 20;
    GLTVector3 objectVertex;          // Вершина,
находящаяся в поле зрения
    GLTVector3 transformedVertex;     // Новая
измененная вершина
    double majorStep = 2.0f*GLT_PI / numMajor;
    double minorStep = 2.0f*GLT_PI / numMinor;
    int i, j;

    for (i=0; i<numMajor; ++i)
        {
        double a0 = i * majorStep;
        double a1 = a0 + majorStep;
        GLfloat x0 = (GLfloat) cos(a0);
        GLfloat y0 = (GLfloat) sin(a0);
        GLfloat x1 = (GLfloat) cos(a1);
        GLfloat y1 = (GLfloat) sin(a1);

        glBegin(GL_TRIANGLE_STRIP);
        for (j=0; j<=numMinor; ++j)
            {
            double b = j * minorStep;
            GLfloat c = (GLfloat) cos(b);
            GLfloat r = minorRadius * c + majorRadius;
```

```
            GLfloat z = minorRadius * (GLfloat) sin(b);

            // Первая точка
            objectVertex[0] = x0*r;
            objectVertex[1] = y0*r;
            objectVertex[2] = z;
            gltTransformPoint(objectVertex, mTransform,
transformedVertex);
            glVertex3fv(transformedVertex);

            // Вторая точка
            objectVertex[0] = x1*r;
            objectVertex[1] = y1*r;
            objectVertex[2] = z;
            gltTransformPoint(objectVertex, mTransform,
transformedVertex);
            glVertex3fv(transformedVertex);
            }
        glEnd();
        }
    }


// Вызывается для рисования сцены
void RenderScene(void)
    {
    GLTMatrix   transformationMatrix;  // Матрица
поворота
    static GLfloat yRot = 0.0f;   // Угол поворота,
задействованный //в //анимации

    yRot += 0.5f;

    // Очистка окна ткущим цветом очистки
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    // Строим матрицу поворота
```

```
    gltRotationMatrix(gltDegToRad(yRot), 0.0f, 1.0f,
0.0f, transformationMatrix);
    transformationMatrix[12] = 0.0f;
    transformationMatrix[13] = 0.0f;
    transformationMatrix[14] = -2.5f;

    DrawTorus(transformationMatrix);

    // Буфер обмена
    glutSwapBuffers();
    }

// Функция выполняет всю необходимую инициализацию в
контексте // визуализации
void SetupRC()
    {
    // Голубой фон
    glClearColor(0.0f, 0.0f, .50f, 1.0f );

    // Все рисуется в каркасном виде
    glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);
    }

////////////////////////////////////////////////////
////
// Вызывается библиотекой GLUT в холостом состоянии
//(окно не меняет размера и не перемещается)
void TimerFunction(int value)
    {
    // Перерисовывает сцену с новыми координатами
glutPostRedisplay();
    glutTimerFunc(33,TimerFunction, 1);
    }



void ChangeSize(int w, int h)
    {
```

```
    GLfloat fAspect;

      // Предотвращает деление на нуль, когда окно
слишком маленькое
      // (нельзя сделать окно нулевой ширины).
    if(h == 0)
        h = 1;

    glViewport(0, 0, w, h);

    fAspect = (GLfloat)w / (GLfloat)h;

    // Система координат обновляется перед модификацией
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();

    // Генерируется перспективная проекция
    gluPerspective(35.0f, fAspect, 1.0f, 50.0f);

    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    }

int main(int argc, char* argv[])
    {
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB |
GLUT_DEPTH);
    glutInitWindowSize(800,600);
    glutCreateWindow("Manual Transformations Demo");
    glutReshapeFunc(ChangeSize);
    glutDisplayFunc(RenderScene);

    SetupRC();
    glutTimerFunc(33, TimerFunction, 1);

    glutMainLoop();
```

```
    return 0;
    }
```

Вначале указываем, где будет храниться матрица

```
GLTMatrix transformationMatrix;  //Матрица поворота
```

Тип данных `GLTMatrix` является просто определением типа, объявленным как массив из 16 элементов с плавающей запятой.

```
typedef GLfloat GLTMatrix[16]; //Матрица 4 на 4 величин типа GLfloat, развертываемая по столбцам
```

Анимация в этом примере заключается в последовательном увеличении переменной `yRot`, которая представляет поворот вокруг оси y. После очистки буфера цвета и глубины составляем матрицу преобразования:

```
      gltRotationMatrix(gltDegToRad(yRot), 0.0f,
1.0f, 0.0f, transformationMatrix);
      tranformationMatrix[12]=0.0f;
      tranformationMatrix[13]=0.0f;
      tranformationMatrix[14]=-2.5f;
```

Первая строка содержит вызов другой функции `glTools`, именуемой `gltRotationMatrix`. Эта функция принимает в качестве аргументов угол поворота в радианах (это способствует более эффективным расчетам) и три аргумента, задающих вектор, вокруг которого выполняется поворот. Исключая то, что угол задан в радианах, а не в градусах, эта функция не отличается от функции OpenGL `glRotate`. Последний аргумент — это матрица, в которой можно записать получающуюся матрицу поворота. Функция `gltDegToRad` выполняет преобразование градусов в радианы.

Как показано на рис. 30, последние два столбца матрицы представляют трансляцию преобразования. Вместо того чтобы выполнять полное умножение матриц, можно просто ввести трансляцию непосредственно в матрицу. Теперь получающаяся матрица представляет трансляцию в пространстве (точку, в которую помещается тор) и последующий поворот системы координат объекта, выполненный в этой точке.

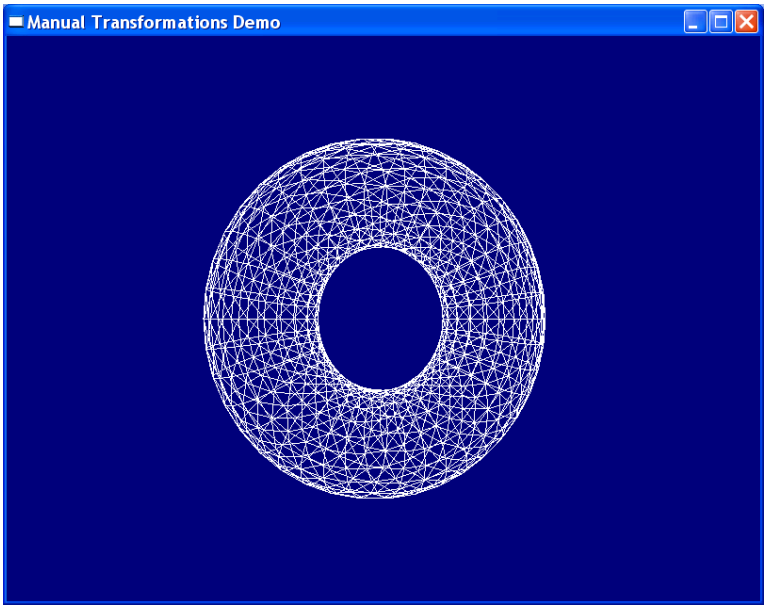


Рис.29 – Вращающийся тор, выполняющий заданные преобразования

Далее матрица преобразования передается функции `DrawTorus`. Вся функция не приводится, но стоит обратить внимание на такие ее строки:

```
        objectVertex[0]=x0*r;

        objectVertex[1]=y0*r;
        objectVertex[2]=z;
        gltTransformPoint(objectVertex, mTransform,
transformedVertex);
        glVertex3fv(transformedVertex);
```

Три компонента вершины загружаются в массив и передаются функции `TransformPoint`. Эта функция `glTools` выполняет умножение вершин на матрицу и возвращает преобразованную вершину в массив `transformedVertex`. После этого используем векторную версию `glVertex` и посылаем OpenGL данные о вершинах. Результатом является вращающийся тор, показанный на рис. 31.

Важно, хотя бы раз увидеть реальную механику преобразования вершин матрицей с использованием подобного детального примера. По мере роста ваших знаний как программиста OpenGL обнаружится, что необходимость преобразовывать точки вручную возникает в таких не связанных прямо с визуализацией задачах, как детектирование столкновений (упругие удары об объекты), отбор по усеченной пирамиде (элементы, которые не видно, отбрасываются и не рисуются) и в некоторых других алгоритмах спецэффектов.

Система актеров

Простым и гибким способом представления системы отсчета является использование структуры данных (или класса в C++), которая содержит точку в пространстве, вектор, указывающий вперед, и вектор, указывающий вверх. Используя эти величины, можно однозначно определить данную точку и ориентацию в пространстве. Приведенный ниже пример взят из библиотеки `glTools` и является структурой данных `GLFrame`, которая может хранить всю эту информацию в одном месте.

```
typedef struct{
GLTVector3f  vLocation;
GLTVector3f vUp;
GLTVector3f vForward;
} GLTFrame;
```

Использование системы отсчета, подобной приведенной, для представления положения и ориентации объекта является очень мощным аппаратом. Для начала можно использовать эти данные непосредственно для создания матрицы преобразования 4x4. Обратимся к рис. 30. Вектор направления «вверх» становится столбцом y матрицы, тогда как вектор "вперед" превращается в вектор столбца z, а точка-положение становится вектором-столбцом трансляции. В результате остается неизвестным только вектор-столбец x, но поскольку известно, что все три оси взаимно перпендикулярны (ортонормальны), можно вычислить вектор-столбец x, найдя векторное произведение векто-

ров x и y. В листинге 6 показана функция `gltGetMatrixFrom-Frame`, которая именно это и делает.

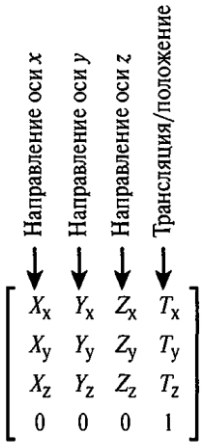


Рис.30 – Как матрица 4x4 представляет положение и ориентацию в трехмерном пространстве

Листинг 6 - Код вычисления матрицы 4x4 по системе отсчета

```
      // Выводит матрицу 4 на 4 из системы отсчета
      void gltGetMatrixFromFrame(GLTFrame *pFrame,
GLTMatrix mMatrix)
      {
      GLTVector3f vXAxis; // Находим ось x
      // Рассчитываем ось x
      gltVectorCrossProduct(pFrame->vUp, pFrame-
>vForward,vXAxis);
      // Заселяем матрицу
      // Вектор-столбец x
      memcpy(mMatrix, vXAxis, sizeof(GLTVector));
      mMatrix[3]=0.0f;
```

```
      // Вектор-столбец y
      memcpy(mMatrix+4,  pFrame->vUp, si-
zeof(GLTVector));
      mMatrix[7]=0.0f;
      // Вектор-столбец z
      memcpy(mMatrix+8,  pFrame->vForward, si-
zeof(GLTVector));
      mMatrix[11]=0.0f;
      // Вектор трансляции/положения
      memcpy(mMatrix+12,  pFrame->vLocation, si-
zeof(GLTVector));
      mMatrix[15]=1.0f;
      }
```

Применение преобразования актеров так же просто, как вызов аргументом, равным получающейся в результате матрице.

Финальная сборка

Разберем теперь последний пример, связывающий в единое целое все концепции, которые здесь обсуждали. В программе SPHEREWORLD создается мир, населенный сферами, которые располагаются в случайных местах. Каждая сфера представляется отдельной структурой `GLTFrame`, отвечающей за ее положение и ориентацию. Также используется система отсчета, чтобы представить камеру, которую можно двигать вокруг мира сфер с помощью клавиш с изображением стрелки. В центра мира сфер с помощью простой высокоуровневой процедуры преобразования создадим вращающийся тор со сферой на его орбите.

В данном примере собраны все идеи, рассмотренные выше, и показано, как они работают вместе. В дополнение к основному исходному файлу `sphereworld.c` проект также содержит модули `torus.c`, `matrixmath.c` и `framemath.c` из библиотеки `glTools`, имеющиеся в папке `\common`. Программа целиком гн приводится, поскольку ее «скелет» GLUT такой же, как и в других примерах, но наиболее важные функции и фрагменты представлены в листинге 7.

Листинг 7 - Основные функции программы SPHEREWORLD

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

//////////////////////////////////////////////////////
// Используемые константы
#define GLT_PI    3.14159265358979323846
#define GLT_PI_DIV_180 0.017453292519943296
#define GLT_INV_PI_DIV_180 57.2957795130823229

//////////////////////////////////////////////////////
//////////////////////////
// Полезные клавиши быстрого вызова и макросы
// Измерения в радианах, но нам нужен способ, чтобы
поменять обратно
#define gltDegToRad(x)  ((x)*GLT_PI_DIV_180)
#define gltRadToDeg(x)  ((x)*GLT_INV_PI_DIV_180)
// Некоторые типы данных
typedef GLfloat GLTVector2[2];//Двукомпонентный вектор
с плавающей запятой
typedef GLfloat GLTVector3[3];//Трехомпонентный вектор
с плавающей запятой
typedef GLfloat GLTVector4[4];//Четырехкомпонентный
вектор с плавающей //запятой
typedef GLfloat GLTMatrix[16];// Основноц столбец
матрицы 4x4 с плавающей //запятой


typedef struct{                      // Рамка для
контейнера ссылок    GLTVector3 vLocation;
    GLTVector3 vUp;
    GLTVector3 vForward;
    } GLTFrame;

#define NUM_SPHERES      50
```

```
GLTFrame    spheres[NUM_SPHERES];
GLTFrame    frameCamera;
// Вычислить векторное произведение двух векторов
void gltVectorCrossProduct(const GLTVector3 vU, const
GLTVector3 vV, GLTVector3 vResult)
      {
      vResult[0] = vU[1]*vV[2] - vV[1]*vU[2];
      vResult[1] = -vU[0]*vV[2] + vV[0]*vU[2];
      vResult[2] = vU[0]*vV[1] - vV[0]*vU[1];
      }
void gltLoadIdentityMatrix(GLTMatrix m)
      {
      static GLTMatrix identity = { 1.0f, 0.0f, 0.0f,
0.0f,
                                    0.0f, 1.0f, 0.0f,
0.0f,
                                    0.0f, 0.0f, 1.0f,
0.0f,
                                    0.0f, 0.0f, 0.0f,
1.0f };

      memcpy(m, identity, sizeof(GLTMatrix));
      }
// Создает матрицу вращения 4x4, используются радианы,
а не в градусы
void gltRotationMatrix(float angle, float x, float y,
float z, GLTMatrix mMatrix)
    {
    float vecLength, sinSave, cosSave, oneMinusCos;
    float xx, yy, zz, xy, yz, zx, xs, ys, zs;

    // Если нулевой вектор проходит, то...
    if(x == 0.0f && y == 0.0f && z == 0.0f)
        {
        gltLoadIdentityMatrix(mMatrix);
        return;
        }
```

```
// Масштабирование вектора
vecLength = (float)sqrt( x*x + y*y + z*z );

// Нормарованная матрица вращения
x /= vecLength;
y /= vecLength;
z /= vecLength;

sinSave = (float)sin(angle);
cosSave = (float)cos(angle);
oneMinusCos = 1.0f - cosSave;

xx = x * x;
yy = y * y;
zz = z * z;
xy = x * y;
yz = y * z;
zx = z * x;
xs = x * sinSave;
ys = y * sinSave;
zs = z * sinSave;

mMatrix[0] = (oneMinusCos * xx) + cosSave;
mMatrix[4] = (oneMinusCos * xy) - zs;
mMatrix[8] = (oneMinusCos * zx) + ys;
mMatrix[12] = 0.0f;

mMatrix[1] = (oneMinusCos * xy) + zs;
mMatrix[5] = (oneMinusCos * yy) + cosSave;
mMatrix[9] = (oneMinusCos * yz) - xs;
mMatrix[13] = 0.0f;

mMatrix[2] = (oneMinusCos * zx) - ys;
mMatrix[6] = (oneMinusCos * yz) + xs;
mMatrix[10] = (oneMinusCos * zz) + cosSave;
mMatrix[14] = 0.0f;

mMatrix[3] = 0.0f;
```

```
    mMatrix[7] = 0.0f;
    mMatrix[11] = 0.0f;
    mMatrix[15] = 1.0f;
    }

// Поворот вектора с использованием матрицы 4x4.
Перевод столбца игнорируется
void gltRotateVector(const GLTVector3 vSrcVector, const
GLTMatrix mMatrix, GLTVector3 vOut)
    {
    vOut[0] = mMatrix[0] * vSrcVector[0] + mMatrix[4] *
vSrcVector[1] + mMatrix[8] *  vSrcVector[2];
    vOut[1] = mMatrix[1] * vSrcVector[0] + mMatrix[5] *
vSrcVector[1] + mMatrix[9] *  vSrcVector[2];
    vOut[2] = mMatrix[2] * vSrcVector[0] + mMatrix[6] *
vSrcVector[1] + mMatrix[10] * vSrcVector[2];
    }
////////////////////////////////////////////////////
//
//Это простое перемещение вперед вдоль переднего
вектора.
void gltInitFrame(GLTFrame *pFrame)
    {
    pFrame->vLocation[0] = 0.0f;
    pFrame->vLocation[1] = 0.0f;
    pFrame->vLocation[2] = 0.0f;

    pFrame->vUp[0] = 0.0f;
    pFrame->vUp[1] = 1.0f;
    pFrame->vUp[2] = 0.0f;

    pFrame->vForward[0] = 0.0f;
    pFrame->vForward[1] = 0.0f;
    pFrame->vForward[2] = -1.0f;
    }

////////////////////////////////////////////////////
////////////
```

```
// Выводит матрицу преобразования 4x4 от системы
отсчета
void gltGetMatrixFromFrame(GLTFrame *pFrame, GLTMatrix
mMatrix)
    {
    GLTVector3 vXAxis;       // Производные оси X

    // Рассчитать оси X
    gltVectorCrossProduct(pFrame->vUp, pFrame-
>vForward, vXAxis);

    // Просто заполнения матрицы столбцом вектора X
    memcpy(mMatrix, vXAxis, sizeof(GLTVector3));
    mMatrix[3] = 0.0f;

    // столбец вектора Y
    memcpy(mMatrix+4, pFrame->vUp, sizeof(GLTVector3));
    mMatrix[7] = 0.0f;

    // Столбец вектора Z
    memcpy(mMatrix+8, pFrame->vForward, si-
zeof(GLTVector3));
    mMatrix[11] = 0.0f;

    // Перевод/Расположение вектора
    memcpy(mMatrix+12, pFrame->vLocation, si-
zeof(GLTVector3));
    mMatrix[15] = 1.0f;
    }

////////////////////////////////////////////////////
/////////////
// Использовать действующее лицо, учитывая его
преобразования системы отсчета
void gltApplyActorTransform(GLTFrame *pFrame)
    {
    GLTMatrix mTransform;
    gltGetMatrixFromFrame(pFrame, mTransform);
```

```
    glMultMatrixf(mTransform);
    }

////////////////////////////////////////////////////////
///////////
// Применение камеры преобразования данной системы
отсчета. Это в //значительной степени альтернативная
реализация gluLookAt использующую тип //float вместо
double и имеющую прямой вектор вместо указателя.
void gltApplyCameraTransform(GLTFrame *pCamera)
    {
    GLTMatrix mMatrix;
    GLTVector3 vAxisX;
    GLTVector3 zFlipped;

    zFlipped[0] = -pCamera->vForward[0];
    zFlipped[1] = -pCamera->vForward[1];
    zFlipped[2] = -pCamera->vForward[2];

    // Вывести X вектор
    gltVectorCrossProduct(pCamera->vUp, zFlipped, vAx-
isX);

    // Заполнение матрицы, обратите внимание, это
поворот и перенос
    mMatrix[0] = vAxisX[0];
    mMatrix[4] = vAxisX[1];
    mMatrix[8] = vAxisX[2];
    mMatrix[12] = 0.0f;

    mMatrix[1] = pCamera->vUp[0];
    mMatrix[5] = pCamera->vUp[1];
    mMatrix[9] = pCamera->vUp[2];
    mMatrix[13] = 0.0f;

    mMatrix[2] = zFlipped[0];
    mMatrix[6] = zFlipped[1];
    mMatrix[10] = zFlipped[2];
```

```
    mMatrix[14] = 0.0f;

    mMatrix[3] = 0.0f;
    mMatrix[7] = 0.0f;
    mMatrix[11] = 0.0f;
    mMatrix[15] = 1.0f;

    // Делаем первый поворот
    glMultMatrixf(mMatrix);

    // Обратные преобразования
    glTranslatef(-pCamera->vLocation[0], -pCamera-
>vLocation[1], -pCamera->vLocation[2]);
    }
void gltMoveFrameForward(GLTFrame *pFrame, GLfloat
fStep)
    {
    pFrame->vLocation[0] += pFrame->vForward[0] *
fStep;
    pFrame->vLocation[1] += pFrame->vForward[1] *
fStep;
    pFrame->vLocation[2] += pFrame->vForward[2] *
fStep;
    }

////////////////////////////////////////////////////
//
// Перемещение системы отсчета вверх оси Y
void gltMoveFrameUp(GLTFrame *pFrame, GLfloat fStep)
    {
    pFrame->vLocation[0] += pFrame->vUp[0] * fStep;
    pFrame->vLocation[1] += pFrame->vUp[1] * fStep;
    pFrame->vLocation[2] += pFrame->vUp[2] * fStep;
    }

////////////////////////////////////////////////////
/
// Перемещение системы отсчета вдоль оси X
```

```
void gltMoveFrameRight(GLTFrame *pFrame, GLfloat fStep)
    {
    GLTVector3 vCross;

    gltVectorCrossProduct(pFrame->vUp, pFrame-
>vForward, vCross);
    pFrame->vLocation[0] += vCross[0] * fStep;
    pFrame->vLocation[1] += vCross[1] * fStep;
    pFrame->vLocation[2] += vCross[2] * fStep;
    }

////////////////////////////////////////////////////////
//
// Перевод рамки в мировых координатах
void gltTranslateFrameWorld(GLTFrame *pFrame, GLfloat
x, GLfloat y, GLfloat z)
    { pFrame->vLocation[0] += x; pFrame->vLocation[1]
+= y; pFrame->vLocation[2] += z; }

////////////////////////////////////////////////////////
//
// Перевод рамки в локальных координатах
void gltTranslateFrameLocal(GLTFrame *pFrame, GLfloat
x, GLfloat y, GLfloat z)
    {
    gltMoveFrameRight(pFrame, x);
    gltMoveFrameUp(pFrame, y);
    gltMoveFrameForward(pFrame, z);
    }

////////////////////////////////////////////////////////
//
// Поворот рамки вокруг локальной оси Y
void gltRotateFrameLocalY(GLTFrame *pFrame, GLfloat
fAngle)
    {
    GLTMatrix mRotation;
    GLTVector3 vNewForward;
```

```
    gltRotationMatrix((float)gltDegToRad(fAngle), 0.0f,
1.0f, 0.0f, mRotation);
    gltRotationMatrix(fAngle, pFrame->vUp[0], pFrame-
>vUp[1], pFrame->vUp[2], mRotation);

    gltRotateVector(pFrame->vForward, mRotation, vNew-
Forward);
    memcpy(pFrame->vForward, vNewForward, si-
zeof(GLTVector3));
    }

////////////////////////////////////////////////////////
///
//  Поворот рамки вокруг локальной оси X
void gltRotateFrameLocalX(GLTFrame *pFrame, GLfloat
fAngle)
    {
    GLTMatrix mRotation;
    GLTVector3 vCross;

    gltVectorCrossProduct(vCross, pFrame->vUp, pFrame-
>vForward);
    gltRotationMatrix(fAngle, vCross[0], vCross[1],
vCross[2], mRotation);

    GLTVector3 vNewVect;
    // Встроеную матрицу 3x3 умножаем для вращения
    vNewVect[0] = mRotation[0] * pFrame->vForward[0] +
mRotation[4] * pFrame->vForward[1] + mRotation[8] *
pFrame->vForward[2];
    vNewVect[1] = mRotation[1] * pFrame->vForward[0] +
mRotation[5] * pFrame->vForward[1] + mRotation[9] *
pFrame->vForward[2];
    vNewVect[2] = mRotation[2] * pFrame->vForward[0] +
mRotation[6] * pFrame->vForward[1] + mRotation[10] *
pFrame->vForward[2];
```

```
    memcpy(pFrame->vForward, vNewVect, si-
zeof(GLfloat)*3);

    // Обновление вектора, направленного вверх
    vNewVect[0] = mRotation[0] * pFrame->vUp[0] + mRo-
tation[4] * pFrame->vUp[1] + mRotation[8] *  pFrame-
>vUp[2];
    vNewVect[1] = mRotation[1] * pFrame->vUp[0] + mRo-
tation[5] * pFrame->vUp[1] + mRotation[9] *  pFrame-
>vUp[2];
    vNewVect[2] = mRotation[2] * pFrame->vUp[0] + mRo-
tation[6] * pFrame->vUp[1] + mRotation[10] * pFrame-
>vUp[2];
    memcpy(pFrame->vUp, vNewVect, sizeof(GLfloat) * 3);
    }

////////////////////////////////////////////////////////
//////
// Поворот рамки вокруг локальной оси Z
void gltRotateFrameLocalZ(GLTFrame *pFrame, GLfloat
fAngle)
    {
    GLTMatrix mRotation;

    // Повернуть требуется только вектор, направленный
вверх
    gltRotationMatrix(fAngle, pFrame->vForward[0],
pFrame->vForward[1], pFrame->vForward[2], mRotation);

    GLTVector3 vNewVect;
    vNewVect[0] = mRotation[0] * pFrame->vUp[0] + mRo-
tation[4] * pFrame->vUp[1] + mRotation[8] *  pFrame-
>vUp[2];
    vNewVect[1] = mRotation[1] * pFrame->vUp[0] + mRo-
tation[5] * pFrame->vUp[1] + mRotation[9] *  pFrame-
>vUp[2];
```

```
    vNewVect[2] = mRotation[2] * pFrame->vUp[0] + mRo-
tation[6] * pFrame->vUp[1] + mRotation[10] * pFrame-
>vUp[2];
    memcpy(pFrame->vUp, vNewVect, sizeof(GLfloat) * 3);
    }
// Скалярное масштабирование вектора
void gltScaleVector(GLTVector3 vVector, const GLfloat
fScale)
    {
    vVector[0] *= fScale; vVector[1] *= fScale; vVec-
tor[2] *= fScale;
    }

// Возвращает длину вектора в квадрате
GLfloat gltGetVectorLengthSqrd(const GLTVector3 vVec-
tor)
    {
    return (vVector[0]*vVector[0]) + (vVec-
tor[1]*vVector[1]) + (vVector[2]*vVector[2]);
    }

// Возвращает длину вектора
GLfloat gltGetVectorLength(const GLTVector3 vVector)
    {
    return
(GLfloat)sqrt(gltGetVectorLengthSqrd(vVector));
    }
// Масштабирование вектора по длине - создание
единичсного вектора
void gltNormalizeVector(GLTVector3 vNormal)
    {
    GLfloat fLength = 1.0f / gltGetVector-
Length(vNormal);
    gltScaleVector(vNormal, fLength);
    }
void gltDrawTorus(GLfloat majorRadius, GLfloat minorRa-
dius, GLint numMajor, GLint numMinor)
    {
```

```
    GLTVector3 vNormal;
    double majorStep = 2.0f*GLT_PI / numMajor;
    double minorStep = 2.0f*GLT_PI / numMinor;
    int i, j;

    for (i=0; i<numMajor; ++i)
            {
            double a0 = i * majorStep;
            double a1 = a0 + majorStep;
            GLfloat x0 = (GLfloat) cos(a0);
            GLfloat y0 = (GLfloat) sin(a0);
            GLfloat x1 = (GLfloat) cos(a1);
            GLfloat y1 = (GLfloat) sin(a1);

            glBegin(GL_TRIANGLE_STRIP);
            for (j=0; j<=numMinor; ++j)
                    {
                    double b = j * minorStep;
                    GLfloat c = (GLfloat) cos(b);
                    GLfloat r = minorRadius * c + ma-
jorRadius;
                    GLfloat z = minorRadius * (GLfloat)
sin(b);

                    // First point
                    glTex-
Coord2f((float)(i)/(float)(numMajor),
(float)(j)/(float)(numMinor));
                    vNormal[0] = x0*c;
                    vNormal[1] = y0*c;
                    vNormal[2] = z/minorRadius;
                    gltNormalizeVector(vNormal);
                    glNormal3fv(vNormal);
                    glVertex3f(x0*r, y0*r, z);

                    glTex-
Coord2f((float)(i+1)/(float)(numMajor),
(float)(j)/(float)(numMinor));
```

```
                    vNormal[0] = x1*c;
                    vNormal[1] = y1*c;
                    vNormal[2] = z/minorRadius;
                    glNormal3fv(vNormal);
                    glVertex3f(x1*r, y1*r, z);
                    }
            glEnd();
            }
      }

////////////////////////////////////////////////////////
///////////
// Эта функция выполняет необходимую инициализацию в
контексте
// визуализации
void SetupRC()
    {
    int iSphere;

    // Голубоватый фон
    glClearColor(0.0f, 0.0f, .50f, 1.0f );

    // Все рисуется в каркасном виде
    glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);

    gltInitFrame(&frameCamera);  // Инициализируется
камера

    // Случайным образом размещаются жители-сферы
    for(iSphere = 0; iSphere < NUM_SPHERES; iSphere++)
        {
        gltInitFrame(&spheres[iSphere]);    //
Инициализируется камера
        // Выбираются случайные положения между -20 и 20
с шагом 0.1
        spheres[iSphere].vLocation[0] = (float)((rand()
% 400) - 200) * 0.1f;
        spheres[iSphere].vLocation[1] = 0.0f;
```

```
        spheres[iSphere].vLocation[2] = (float)((rand()
% 400) - 200) * 0.1f;
        }
    }


////////////////////////////////////////////////////////
////
// Рисуется земля с координатной сеткой
void DrawGround(void)
    {
    GLfloat fExtent = 20.0f;
    GLfloat fStep = 1.0f;
    GLfloat y = -0.4f;
    GLint iLine;

    glBegin(GL_LINES);
       for(iLine = -fExtent; iLine <= fExtent; iLine +=
fStep)
          {
          glVertex3f(iLine, y, fExtent);    // Рисуются
z-дорожки
          glVertex3f(iLine, y, -fExtent);

          glVertex3f(fExtent, y, iLine);
          glVertex3f(-fExtent, y, iLine);
          }

    glEnd();
    }


// Вызывается для рисования сцены
void RenderScene(void)
    {
    int i;
      static GLfloat yRot = 0.0f; // Используемый в
анимации угол поворота
```

```
    yRot += 0.5f;

    // Очищаем окно текущим цветом очистки
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    glPushMatrix();
        gltApplyCameraTransform(&frameCamera);

        // Рисуем землю
        DrawGround();

        // Рисуем случайным образом расположенные сферы
        for(i = 0; i < NUM_SPHERES; i++)
            {
            glPushMatrix();
            gltApplyActorTransform(&spheres[i]);
            glutSolidSphere(0.1f, 13, 26);
            glPopMatrix();
            }

        glPushMatrix();
            glTranslatef(0.0f, 0.0f, -2.5f);

            glPushMatrix();
                glRotatef(-yRot * 2.0f, 0.0f, 1.0f,
0.0f);
                glTranslatef(1.0f, 0.0f, 0.0f);
                glutSolidSphere(0.1f, 13, 26);
            glPopMatrix();

            glRotatef(yRot, 0.0f, 1.0f, 0.0f);
            gltDrawTorus(0.35, 0.15, 40, 20);
        glPopMatrix();
    glPopMatrix();

    // Переключение буферов
    glutSwapBuffers();
```

```
    }


// Реагирует на клавиши со стрелками, двигая систему
отсчета камеры
void SpecialKeys(int key, int x, int y)
    {
    if(key == GLUT_KEY_UP)
        gltMoveFrameForward(&frameCamera, 0.1f);

    if(key == GLUT_KEY_DOWN)
        gltMoveFrameForward(&frameCamera, -0.1f);

    if(key == GLUT_KEY_LEFT)
        gltRotateFrameLocalY(&frameCamera, 0.1);

    if(key == GLUT_KEY_RIGHT)
        gltRotateFrameLocalY(&frameCamera, -0.1);

    // Обновление окна
    glutPostRedisplay();
    }


//////////////////////////////////////////////////////
////
//Вызывается библиотекой GLUT в холостом состоянии
(окно не меняет
// размера и не перемещается)
void TimerFunction(int value)
    {
    // Перерисовывает сцену с новыми координатами
    glutPostRedisplay();
    glutTimerFunc(3,TimerFunction, 1);
    }

void ChangeSize(int w, int h)
```

```
    {
    GLfloat fAspect;

    // Предотвращает деление на нуль, когда окно
слишком маленькое
// (нельзя сделать окно нулевой ширины).
    if(h == 0)
        h = 1;

    glViewport(0, 0, w, h);

    fAspect = (GLfloat)w / (GLfloat)h;

    // Система координат обновляется перед модификацией
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();

    // Устанавливается объём сечения
    gluPerspective(35.0f, fAspect, 1.0f, 50.0f);

    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 SphereWorld Demo");
    glutReshapeFunc(ChangeSize);
    glutDisplayFunc(RenderScene);
    glutSpecialFunc(SpecialKeys);

    SetupRC();
    glutTimerFunc(33, TimerFunction, 1);
```

```
    glutMainLoop();

    return 0;
    }
```

Первые несколько строк содержат макрос, определяющий число сферических «жителей» равным 50. После этого объявляется массив систем отсчета и еще одна система, представляющая камеру.

```
      #define NUM_SPHERES 50
      GLTFrame spheres[NUM_SPHERES];
      GLTFrame frameCamera;
```

Функция `SetupRC` вызывает функцию `glTools` под названием `gltInitFrame` с целью инициализации в начале координат камеры, смотрящей в отрицательном направлении оси z (ориентация OpenGL по умолчанию).

```
      gltInitFrame(&frameCamera);   //   Инициализируется
камера
```

С помощью этой функции можно инициализировать любую каркасную структуру, также можно самостоятельно инициализировать структуру, имеющую желаемое положение и ориентацию. После этого в программе идет цикл, инициализирующий массив сферических каркасов и выбирающий случайные координаты x и z точек расположения этих сфер.

```
      // Случайным образом размещаются жители-сферы
      for(iSphere=0;iSphere<NUM_SPHERES;iSphere++)
      {
      gltInitFrame(&spheres[iSphere]);//
Инициализируется система отсчета
      // Выбираются случайные положения между -20 и 20
с шагом 0,1
      spheres[iSphere].vLocation[0]=(float)((rand()%400
)-200)*0.1f;
      spheres[iSphere].vLocation[1]=0.0f;
      spheres[iSphere].vLocation[2]=(float)((rand()%400
)-200)*0.1f;
      }
```

Затем функция DrawGround с помощью отрезков GL_LINE рисует землю как ряд перекрещивающихся линий.

```
// Рисуется земля с координатной сеткой
void DrawGround(void)
{
GLfloat fExtent=20.0f;
GLfloat fStep=1.0f;
GLfloat y=-0.4f;
Glint iLine;
glBegin(GL_LINES);
for(iLine=-fExtent; iLine<=fExtent; iLine+=fStep)
{
glVertex3f(iLine, y, fExtent);// Рисуются z-дорожки
glVertex3f(iLine, y, -fExtent);
glVertex3f(fExtent, y, iLine);
glVertex3f(-fExtent, y, iLine);
}
glEnd();
}
```

Функция RenderScene рисует мир с нашей точки наблюдения. Обратите внимание на то, что вначале записывается единичную матрицу, а затем с помощью вспомогательной функции glTools gltApplyCameraTransform применяем преобразование камеры. Земля статична и преобразовывается камерой только для того, чтобы создать иллюзию движения.

```
glPushMatrix();
gltApplyCameraTransform(&frameCamera);
// Рисуем землю
DrawGround();
```

После этого рисуются все случайным образом расположение сферы. По системе отсчета функция gltApplyActorTransform создает матрицу преобразования и умножает ее на текущую матрицу (а это матрица камеры). Каждой сфере должно сопоставляться собственное преобразование относительно камеры, поэтому матрица камеры записывается при каждом вызове функции glPushMatrix и снова вос-

станавливается с помощью `glPopMatrix`, готовой к обработке новой сферы или новому преобразованию.

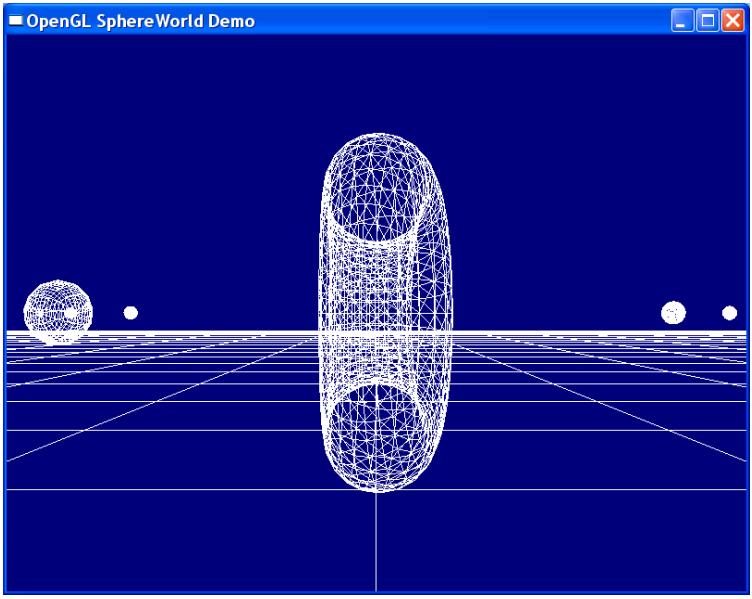


Рис.31 – Результат выполнения программы SPHEREWORLD

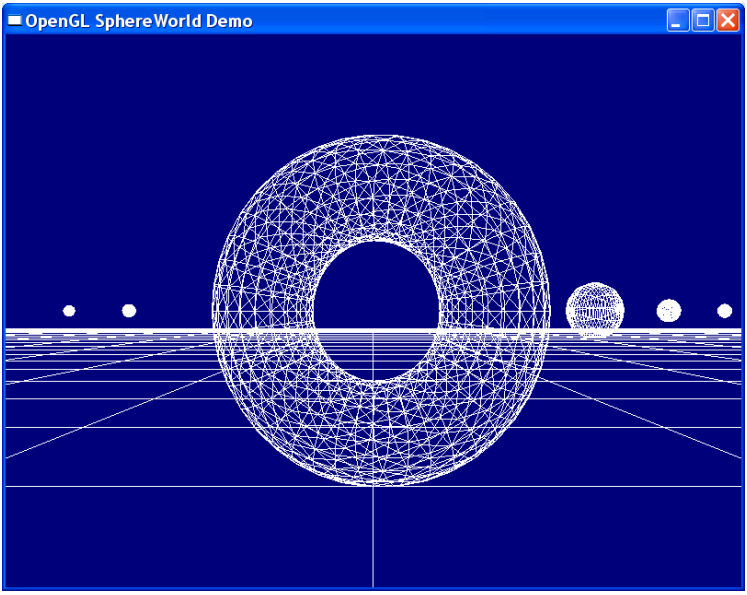


Рис.32 – Результат выполнения программы SPHEREWORLD

```
// Рисуем случайным образом расположение сферы
for(i=0;i<NUM_SPHERES;i++)
{
glPushMatrix();
gltApplyActorTransform(&spheres[i]);
glutSolidSphere(0.1f,13,26);
glPopMatrix();
}
```

А вот теперь пришло время активных действий! Вначале перемещаем систему координат немного вниз по оси z, чтобы можно было увидеть, что следует рисовать дальше. Записываем эту точку и выполняем поворот, затем трансляцию и рисуем сферу. Из-за применения данного эффекта сфера кажется вращающейся вокруг начала координат прямо перед нами. Затем восстанавливаем матрицу преобразования, но только так, чтобы положение начала координат было в точке z=-2,5. После этого, но перед рисование тора, выполняется еще один поворот. В результате кажется, что тор вращается.

```
glPushMatrix();
glTranslatef(0.0f, 0.0f, -2.5f);
```

Наконец, при нажатии любой клавиши со стрелкой вызывается функция `SpecialKeys`. Клавиши со стрелками вверх и вниз вызывают функцию `glTools gltMoveFrameForward`, которая просто перемещает систему отсчета вдоль линии обзора. В ответ на нажатие клавиш со стрелками влево и вправо функция `gltRotateFrameLocalY` поворачивает систему отсчета вокруг локальной оси y (независимо от ориентации).

```
glPushMatrix();
glRotatef(-yRot*2.0f, 0.0f, 1.0f, 0.0f);
glTranslatef(1.0f, 0.0f, 0.0f);
glutSolidSphere(0.1f, 13, 26);
glPopMatrix();
glRotatef(yRot, 0.0f, 1.0f, 0.0f);
gltDrawTorus(0.35, 0.15, 40, 20);
glPopMatrix();
```

```
    glPopMatrix();
```

В сумме все рассмотренные функции дают следующий эффект: мы видим сетку на земле со множеством сфер, разбросанным по случайным местам. Перед нами находится вращающийся тор, по орбите которого быстро движется сфера (рис. 31 и .32).

```
void SpecialKeys(int key, int x, int y)
{
if(key==GLUT_KEY_UP)
gltMoveFrameForward(&rframeCamera, 0.1f);
if(key==GLUT_KEY_DOWN)
gltMoveFrameForward(&rframeCamera, -0.1f);
if(key==GLUT_KEY_LEFT)
gltRotateFrameLocalY(&rframeCamera, 0.1);
if(key==GLUT_KEY_RIGHT)
gltRotateFrameLocalY(&rframeCamera, -0.1);
// Обновляет окно
glutPostRedisplay();
}
```

ЗАДАНИЕ НА ЛАБОРАТОРНУЮ РАБОТУ

Воспроизвести результаты, представленные в теоретическом обзоре, применить различные преобразования матриц для создания сцен, согласно варианту, полученному у преподавателя.

ТРЕБОВАНИЯ К РЕАЛИЗАЦИИ

Задание выполняется согласно варианту. По завершении готовится отчет.

1) Листинг 1 преобразовать согласно варианту.
2) С помощью Листинга 2 и Листинга 3 изобразить фигуру согласно варианту с разным цветом внутренних и внешних граней (возможен разный цвет всех внешних граней). Поменять цвет фона.
3) Для Листинга 4 реализовать изменения согласно варианту.
4) Для Листинга 5 реализовать изменения согласно варианту.
5) Для Листинга 7 реализовать изменения согласно варианту.

Задания 4 и 5 обязательны только для студентов, претендующих на оценку «отлично».

ВАРИАНТЫ ЗАДАНИЙ

Задание 1.

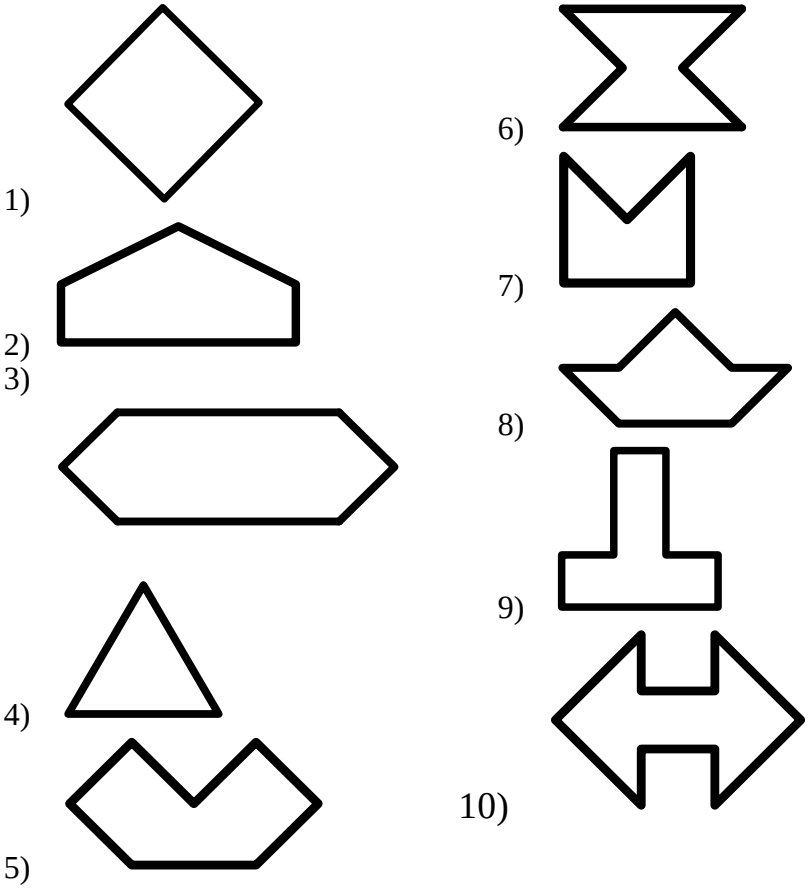
1) Расположить объекты правее на экране.
2) Добавить еще 2 атома к существующей системе в разные плоскости вращения.
3) Изменить круговые орбиты на произвольные замкнутые.
4) Изменить цвет всех объектов.
5) Осуществить небольшие круговые движения для центрального атома.
6) Поменять плоскость вращения всех атомов и изменить их скорость и направление движения.
7) На каждую существующую орбиту добавить по одному атому другого цвета.

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 2.



Задание 3.

1) Изменить плоскость вращения Земли и Луны, скорость и направление вращения Земли.
2) Изменить плоскость вращения Луны, направление вращения и скорость.
3) Добавить одну планету без спутника в плоскости и траектории Земля-Луна
4) Добавить одну планету без спутника вне плоскости Земля-Луна на любой траектории.
5) Добавить одну планету со спутником в плоскости и траектории Земля-Луна
6) Добавить одну планету со спутником в плоскости и на другой траектории Земля-Луна
7) Добавить к Земле еще одни спутник меньшего размера, находящийся на другой траектории и с вращением в противоположную сторону.
8) Добавить в систему две планеты без спутников, находящиеся в разных плоскостях (одна ближе к Солнцу, другая дальше от Солнца).
9) Добавить в систему две планеты без спутников, находящиеся в разных плоскостях (одна ближе к Солнцу, другая дальше от Солнца).
10) Добавить в систему две планеты одна из них со спутником. Планеты находятся в разных плоскостях (одна ближе к Солнцу, другая дальше от Солнца).

Задание 4.
1) Реализовать хаотичное изменение скорости вращения тора
2) Реализовать хаотичное изменение плоскости вращения тора
3) Закрасить тор и изменить направление вращения.
4) Добавить сферу и реализовать вращение сферы вокруг тора по часовой стрелке
5) Добавить еще три вращающихся в разных плоскостях и направлениях тора. Расположить их по координатным четвертям.
6) Добавить слева от тора вращающийся чайник Юта вокруг одной из своих осей и одновременное вращение вокруг тора.

Задание 5.

1) Добавить к вращающемуся тору еще две вращающиеся сферы в разных плоскостях.

2) В произвольном месте пространства вблизи поверхности расположить 5 вращающихся сфер.

3) Изменить размеры пространства (уменьшить), расположить 40 сфер произвольного цвета на разном уровне по оси z.

4) Выше ключевых сферы и тора, добавить сферу и тор, вращающиеся в другом направлении с меньшей скорость.

5) В пространство добавить не менее 10 вращающихся кубов и не менее 5 вращающихся цветных чайников Юта.

6) Создать не менее 10 клонов вращающейся сферы и тора. Объекты должны иметь разный цвет, размер, положение по оси z и направление вращения.

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 мартриц.
8. Раскройте назначение единичной матрицы.
9. Опишите область применения стека матриц и алгоритм егог работы.
10. Сформулируйте суть нетривиального умножения матриц и приемущества его использования.
11. Изложите концепцию использовавшейся в работе функции таймера.
12. Дайте пример описания в GLU функции управления камерой.
13. Приведите алгоритм ортогонального проецирования в OpenGL.
14. Предложите альтернативу функции `gluPerspective` при создании перспективной проекции.
15. Раскройте значение термина *актеры* в контексте построения сцен.

ФОРМА ОТЧЕТА ПО ЛАБОРАТОРНОЙ РАБОТЕ

На выполнение лабораторной работы отводится 2 занятия (4 академических часа: 3 часа на выполнение и сдачу практического задания и 1 час на подготовку отчета).

Номер варианта студента назначается индивидуально преподавателем.

В отчете должны быть представлены:

1) Текст задания для лабораторной работы и номер варианта.
2) Все полные листинги программ, для обязательных заданий. Претендующие на оценку отлично указывают также листинги для заданий 4 и 5. При необходимости листинги программ можно сокращать, если повторные части кода присутствуют в ранее указанных листингах. Во всех листингах программ должны быть подробные комментарии к основным функциям приложения. Выполнение задания должно сопровождаться снимками экрана.

Отчет по каждому новому заданию начинать с новой страницы. В выводах отразить затруднения при ее выполнении и достигнутые результаты.

Отчет на защиту предоставляется в печатном виде.

Отчет содержит: Структура отчета (на отдельном листе(-ах)): титульный лист, формулировка задания (вариант), этапы выполнения работы, результаты выполнения работы, выводы.

ОСНОВНАЯ ЛИТЕРАТУРА

1. Боресков А.В. Основы работы с технологией CUDA / А.В. Боресков, А.А. Харламов - Издательство "ДМК Пресс", 2010. - 232 с. - ISBN 978-5-94074-578-5; ЭБС «Лань». - URL: https://e.lanbook.com/book/1260#book_name (23.12.2017).
2. Васильев С.А. OpenGL. Компьютерная графика : учебное пособие / С.А. Васильев. — Электрон. текстовые данные. — Тамбов: Тамбовский государственный технический университет, ЭБС АСВ, 2012. — 81 с. — 2227-8397. — Режим доступа: http://www.iprbookshop.ru/63931.html — ЭБС «IPRbooks», по паролю
3. Вольф Д. OpenGL 4. Язык шейдеров. Книга рецептов/ Вольф Д. - Издательство "ДМК Пресс", 2015. - 368 с. - 978-5-97060-255-3; ЭБС «Лань». - URL: https://e.lanbook.com/book/ 73071#book_name (23.12.2017).
4. Гинсбург Д. OpenGL ES 3.0. Руководство разработчика/Д. Гинсбург, Б. Пурномо. - Издательство "ДМК Пресс", 2015. - 448 с. - ISBN 978-5-97060-256-0; ЭБС «Лань». - URL: https://e.lanbook.com/book/82816#book_name (29.12.2017).
5. Лихачев В.Н. Создание графических моделей с помощью Open Graphics Library / В.Н. Лихачев. — Электрон. текстовые данные. — М. : Интернет-Университет Информационных Технологий (ИНТУИТ), 2016. — 201 с. — 2227-8397. — Режим доступа: http://www.iprbookshop.ru/39567.html
6. Забелин Л.Ю. Основы компьютерной графики и технологии трехмерного моделирования : учебное пособие/ Забелин Л.Ю., Конюкова О.Л., Диль О.В.— Новосибирск: Сибирский государственный университет телекоммуникаций и информатики, 2015.— 259 с.— Режим доступа: http://www.iprbookshop.ru/54792.— ЭБС «IPRbooks», по паролю
7. Папуловская Н.В. Математические основы программирования трехмерной графики : учебно-методическое пособие / Н.В. Папу-

ловская. — Электрон. текстовые данные. — Екатеринбург: Уральский федеральный университет, 2016. — 112 c. — 978-5-7996-1942-8. — Режим доступа: http://www.iprbookshop.ru/68345.html
8. Перемитина, Т.О. Компьютерная графика : учебное пособие / Т.О. Перемитина ; Министерство образования и науки Российской Федерации, Томский Государственный Университет Систем Управления и Радиоэлектроники (ТУСУР). - Томск : Эль Контент, 2012. - 144 с. : ил.,табл., схем. - ISBN 978-5-4332-0077-7 ; - URL: http//biblioclub.ru/index.php?page=book&id=208688 (30.11.2017).

ДОПОЛНИТЕЛЬНАЯ ЛИТЕРАТУРА

Электронные ресурсы:

1. https://open.gl/transformations - Математические основы матричных преобразований и их связь с системой координат
2. https://www.intuit.ru/studies/courses/2313/613/lecture/13303?page=2 – OpenGL. Матричные преобразования
3. https://habrahabr.ru/post/324968 - Системы координат OpenGl
4. http://www.gamedev.ru/code/articles/openglcamera - Камера в OpenGL