

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, С.А. Глебов

ОСНОВНЫЕ ПРИМИТИВЫ 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 № 1 от «18» 02 2018 г.

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

- Методической комиссией КФ МГТУ им.Н.Э. Баумана протокол № 1 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 о координатных пространствах и их проекциях, реализуемых в OpenGL. Рассмотрен процесс работы с буфером, отрисовки простейших геометрических форм и решения задач отсечения и масштабирования, а также приведен пример работы с машиной состояний OpenGL.

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

© Калужский филиал МГТУ им. Н.Э. Баумана, 2018 г.
© Ю.С. Белов, 2018 г.

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ	4
ЦЕЛЬ И ЗАДАЧИ РАБОТЫ, ТРЕБОВАНИЯ К РЕЗУЛЬТАТАМ ЕЕ ВЫПОЛНЕНИЯ	5
ХАРАКТЕРИСТИКА ОБЪЕКТА ИЗУЧЕНИЯ, ИССЛЕДОВАНИЯ	6
ЗАДАЧИ И ПОРЯДОК ВЫПОЛНЕНИЯ РАБОТЫ	28
ЗАДАНИЕ НА ЛАБОРАТОРНУЮ РАБОТУ	70
ТРЕБОВАНИЯ К РЕАЛИЗАЦИИ	70
ВАРИАНТЫ ЗАДАНИЙ	71
КОНТРОЛЬНЫЕ ВОПРОСЫ И ЗАДАНИЯ	72
ФОРМА ОТЧЕТА ПО ЛАБОРАТОРНОЙ РАБОТЕ	73
ОСНОВНАЯ ЛИТЕРАТУРА	74
ДОПОЛНИТЕЛЬНАЯ ЛИТЕРАТУРА.....	75

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 задания точек в трехмерном пространстве средствами программного интерфейса OpenGL, поэтапные примеры описания блоков простейших форм и отображения их, 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 графическими примитивами OpenGL. Для выполнения лабораторной работы студенту необходимо уметь ориентироваться в способах описания графических примитивов OpenGL, уметь работать отсечением граней по признаку ориентации, создавать и редактировать графические примитивы.

Программный интерфейс OpenGL, кратко описанный в методических указаниях, может быть применен при создании моделей использующих конвейер трехмерной графики. С помощью простейших графических элементов можно задавать композитные объекты любой сложности.

ЦЕЛЬ И ЗАДАЧИ РАБОТЫ, ТРЕБОВАНИЯ К РЕЗУЛЬТАТАМ ЕЕ ВЫПОЛНЕНИЯ

Целью выполнения лабораторной работы является формирование практических навыков по работе с графическими примитивами OpenGL, а также применения к ним эффектов средствами машины состояний и использования проверки глубины.

Основными задачами выполнения лабораторной работы являются: научиться устанавливать размеры наблюдаемого объема, изучить параметры функции `glVertex`, изучить основные параметры функции `glBegin`, сформировать понимание особенности использования функции `glEnable` с конкретными геометрическими примитивами, выяснить основы построения сплошных объектов.

Результатами работы являются:

- Выполненные средствами OpenGL сцены из простейших геометрических объектов, с использованием различных методов описания
- Сформированное представление об обходе вершин, лицевых и обратных гранях
- Применение отсечений для оптимизации расчетов отображаемой сцены
- Подготовленный отчет

ХАРАКТЕРИСТИКА ОБЪЕКТА ИЗУЧЕНИЯ, ИССЛЕДОВАНИЯ

Объекты и сцены, которые создаются с помощью OpenGL, состоят из маленьких, простых форм, упорядоченных различными способами. Компонентные элементы трехмерных объектов, называются *примитивами*. Все примитивы в OpenGL, начиная от точек и линий и заканчивая сложными многоугольниками, являются одно- или двухмерными объектами.

Рисование точек в трехмерном пространстве

Пиксель — это наименьший элемент на мониторе, в цветной системе пиксели могут раскрашиваться одним из множества доступных цветов. Эта разновидность компьютерной графики является простейшей: нарисовать точку на экране и присвоить ей определенный цвет. Далее на основе этой простейшей концепции можно создавать линии, многоугольники, окружности и другие формы. Возможно, таким средством можно даже создать графический интерфейс пользователя.

При использовании OpenGL рисование на экране компьютера отличается фундаментально. Работа ведется не с физическими экранными координатами и пикселями, а с позиционными координатами в выбранном объеме наблюдения. OpenGL заботится о том, как спроектировать точки, линии и все остальное из заданного трехмерного пространства на двухмерное изображение, формируемое на экране компьютера.

Трехмерная точка: вершина

Чтобы задать нарисованную точку на трехмерной "палитре", используем функцию OpenGL `glVertex`, — наиболее распространенную функцию во всех программных интерфейсах OpenGL. Функция `glVertex` может принимать от одного до четырех параметров любого численного типа (от `byte` до `double`) с учетом договоренностей относительно именования.

В следующей простой строке кода задается точка, в нашей системе координат расположенная на 50 единиц по положительному направлению оси x , на 50 единиц по положительному направлению оси y и на 0 единиц по положительному направлению оси z .

```
glVertex3f(50.0f, 50.0f, 0.0f);
```

Эта точка показана на рис. 1. Здесь координаты представлены как значения с плавающей запятой. Кроме того, использованная форма `glVertex` принимает три аргумента — координаты x , y и z , соответственно.

Две другие формы `glVertex` принимают, соответственно, два и четыре аргумента. Ту же точку, что и на рис. 1, мы можно представить с помощью такого кода:

```
glVertex2f(50.0f, 50.0f);
```

Эта форма `glVertex` принимает только два аргумента, и предполагает, что координата z всегда равна 0.0. В форме `glVertex`, принимающей четыре аргумента (`glVertex4`) последняя координата w (по умолчанию установленная равной 1.0) используется при масштабировании.

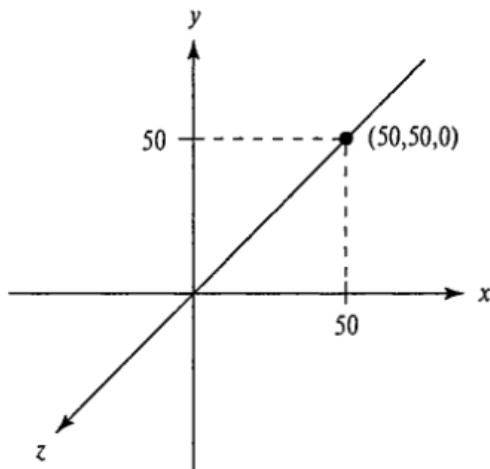


Рис.1 – Точка $(50, 50, 0)$, заданная с помощью команды `glVertex3f(50.0f, 50.0f, 0.0f)`

Согласно геометрическому определению вершины, она не является просто точкой в пространстве, а точкой пересечения двух прямых или кривых линий. Именно это и является сутью примитивов.

Примитив — это просто интерпретация набора или списка вершин в виде некоторой формы, изображенной на экране. В OpenGL существует 10 примитивов — от простой точки, изображенной в пространстве, до замкнутого многоугольника с любым числом сторон. Один из

способов рисования примитивов — с помощью команды `glBegin` сообщить OpenGL, что список вершин нужно интерпретировать как определенный примитив. Затем закрыть список вершин, определяющих примитив, воспользовавшись командой `glEnd`.

Все геометрические примитивы описываются в терминах своих *вершин* — координат, которые определяют сами точки примитива, конечные точки линии или углы полигона. Имеется некоторое представление о математическом смысле терминов точка, линия и полигон. Их смысл в OpenGL почти тот же самый, но не идентичный. Одно из различий проистекает из ограничений на компьютерные расчеты. В любой реализации OpenGL числа с плавающей точкой имеют конечную точность и, следовательно, могут возникать ошибки, связанные с округлением. Как следствие, координаты точек, линий и полигонов в OpenGL страдают этим же недостатком. Более важное различие проистекает из ограничений растровых дисплеев. На таких дисплеях наименьшим отображаемым элементом является пиксель и, хотя пиксель по размеру может быть меньше 1/100 дюйма, он все равно значительно больше, чем математические понятия бесконечно малого элемента (для точек) и бесконечно короткого (для линий). Когда OpenGL производит вычисления, она предполагает, что точки представлены в виде векторов с координатами в формате с плавающей точкой. Однако точка, как правило (но не всегда), отображается на дисплее в виде одного пикселя, и несколько точек, имеющих слегка различающиеся координаты, OpenGL может нарисовать на одном и том же пикселе.

Рисование геометрических примитивов

Точки

Точка определяется набором чисел с плавающей точкой, называемым вершиной. Все внутренние вычисления производятся в предположении, что координаты заданы в трехмерном пространстве. Для вершин, которые пользователь определил как двумерные (то есть задал только x - и y -координаты), OpenGL устанавливает z - координату равной 0. OpenGL работает в *однородных координатах* трехмерной проекционной геометрии, таким образом, во внутренних вычислениях все вершины представлены четырьмя числами с плавающей точкой (x, y, z, w) . Если w отлично от 0, то эти координаты в Евклидовой геометрии соответствуют точке $(x/w, y/w, z/w)$. Можно указывать w -

координату в командах OpenGL, но это делается достаточно редко. Если w-координата не указана, она принимается равной 1.0.

Линии

В OpenGL термин линия относится к сегменту прямой (отрезку), а не к математическому понятию прямой, которая предполагается бесконечной в обоих направлениях. Достаточно просто задавать серию соединенных отрезков или даже закрытую последовательность отрезков. В любом случае отрезки описываются в терминах вершин на их концах.

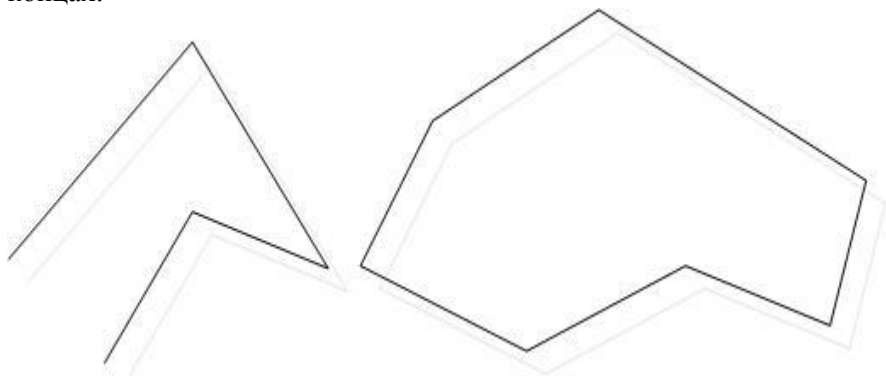


Рис.2 – Открытая и закрытая последовательности отрезков

Полигоны

Полигон (или многоугольник) – это область, ограниченная одной замкнутой ломаной, при этом каждый отрезок ломаной описывается вершинами на его концах (вершинами полигона). Обычно полигоны рисуются заполненными, но можно также рисовать полигоны в виде их границ или в виде точек на концах отрезков, образующих полигон. В общем случае полигоны могут быть достаточно сложны, поэтому OpenGL накладывает очень серьезные ограничения в отношении того, что считается полигональным примитивом. Во-первых, ребра полигонов в OpenGL не могут пересекаться (в математике полигон, удовлетворяющий этому условию, называется *простым*). Во-вторых, полигоны OpenGL должны быть выпуклыми. Полигон является выпуклым, если отрезок, соединяющий две точки полигона (точки могут быть и внутренними, и граничными) целиком лежит внутри полигона (то есть

все его точки также принадлежат полигону). На рис. 3 приведены примеры нескольких допустимых и недопустимых полигонов. OpenGL не накладывает ограничений на количество вершин полигона (и, как следствие, на количество отрезков, определяющих его границу). Необходимо отметить, что полигоны с дырами не могут быть описаны, так как они не являются выпуклыми, и могут быть заданы при помощи одной замкнутой ломаной. Если вы поставите OpenGL перед вопросом о рисовании *невыпуклого* полигона, результат может быть не таким, какой вы ожидаете. Например, на большинстве систем в ответ будет заполнена только выпуклая оболочка полигона (на многих системах не произойдет и этого).

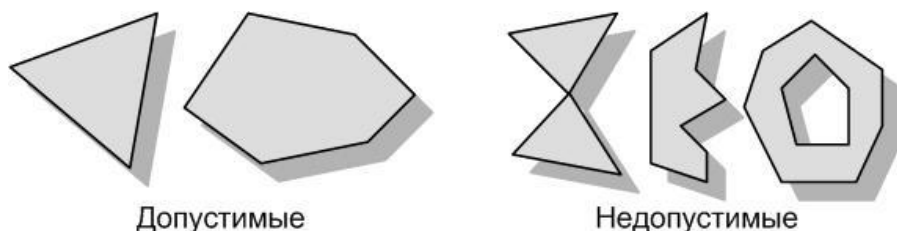


Рис.3 – Допустимые и недопустимые полигоны

Причина, по которой OpenGL накладывает подобные ограничения, заключается в том, что в аппаратуру намного проще заложить функции визуализации некоторого определенного класса полигонов. Простые полигоны визуализировать легко. Для увеличения быстродействия OpenGL предполагает, что все полигоны простые. Многие реальные поверхности состоят из непростых полигонов, невыпуклых полигонов или полигонов с дырами. Поскольку каждый из таких полигонов может быть представлен в виде набора простых выпуклых полигонов, в библиотеке GLU имеется набор методов для построения более сложных объектов. Эти методы принимают описание сложного полигона и выполняют *тесселяцию* (*teselation*), то есть разбиение сложного полигона на группы более простых OpenGL – полигонов, которые в последствие могут быть визуализированы и отображены. Поскольку вершины в OpenGL всегда трехмерны, точки, формирующие границу каждого конкретного полигона, не обязательно лежат на одной плоскости пространства. (Конечно, во многих случаях они лежат на одной плоскости – например, если z-координаты всех точек

равны 0 или если полигон является *треугольником*.) Если вершины полигона не лежат в одной плоскости, то после различных пространственных трансформаций, изменения точки наблюдения и проецирования на экран дисплея может оказаться, что эти точки уже не определяют простой выпуклый полигон. Представим четырехугольник, в котором точки немного отклоняются от единой плоскости, и посмотрим на него со стороны ребра. В таких условиях может оказаться, что полигон не является простым и, как следствие, может быть не визуализирован верно. (Рис. 4) Ситуация не является такой уж необычной в случае если вы аппроксимируете изогнутые поверхности с помощью четырехугольников. Однако всегда можно избежать подобных проблем, применяя треугольники, так как три отдельно взятые точки всегда лежат в одной плоскости.



Рис.4 – Пример неплоского полигона, трансформированного в непростой

Квадраты

Поскольку квадраты часто встречаются в графических приложениях, OpenGL предоставляет примитив, рисующий заполненный квадрат и его команду **glRect*()**. Можно нарисовать квадрат как полигон, но в вашей конкретной реализации OpenGL **glRect*()** для квадратов может быть реализована более рационально.

```
void glRect{sifd} (TYPE x1, TYPE y1, TYPE x2, TYPE y2);
void glRect{sifd}v (TYPE *v1, TYPE *v2);
```

Рисует квадрат, определяемый угловыми точками $(x1, y1)$ и $(x2, y2)$. Квадрат лежит в плоскости $z=0$ и его стороны параллельны осям абсцисс и ординат. Если используется векторная форма функции, углы задаются в виде двух указателей на массивы, каждый из которых содержит пару (x, y) . Необходимо заметить, что хотя по умолчанию квадрат имеет определенную ориентацию в трехмерном пространстве, можно изменить это за счет применения вращения или других трансформаций.

Кривые и изогнутые поверхности

Любая плавная кривая или поверхность может быть аппроксимирована с любой степенью точности короткими отрезками или полигональными регионами. Таким образом, достаточное разделение кривых или поверхностей на прямые отрезки или плоские полигоны может привести к тому, что они по-прежнему будут выглядеть плавно изогнутыми (рис. 5).

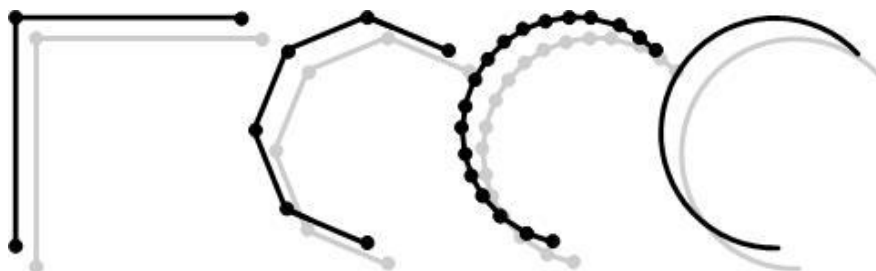


Рис.5 – Аппроксимация кривых

Указание вершин

При использовании OpenGL каждый геометрический объект однозначно описывается в виде упорядоченного набора вершин. Для указания вершины используется команда **glVertex*()**.

```
void glVertex{234}{sifd}(TYPE coords);  
void glVertex{234}{sifd}v(const TYPE *coords);
```

Указывает одну вершину для использования в описании геометрического объекта. Для каждой вершины можно указывать от 2 (x,y) до 4 координат (x,y,z,w), выбирая соответствующую версию команды. Если используется версия, где в явном виде не задаются z или w, то z принимается равным 0, а w – равным 1. Вызовы **glVertex*()** имеют силу только между вызовами команд **glBegin()** и **glEnd()**.

Теперь, когда известно, как задавать вершины, все еще нужно узнать, как заставить OpenGL создать из этих вершин набор точек, линию или полигон. (Рис. 6) Для того чтобы это сделать, следует обработать каждый набор вершин вызовами команд **glBegin()** и **glEnd()**. Аргумент, передаваемый **glBegin()**, определяет, какие графические примитивы создаются на основе вершин.

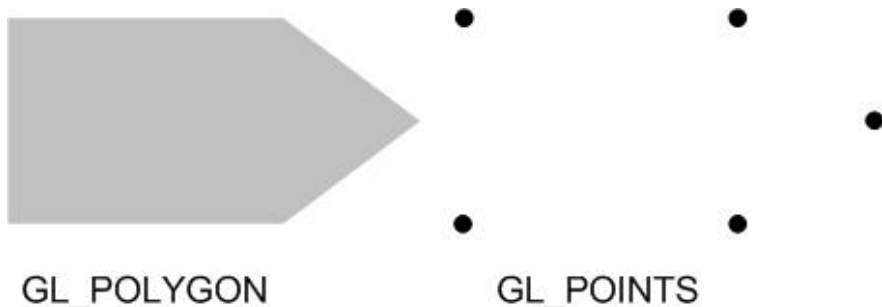


Рис.6 –Полигон или набор точек

Ломаные и замкнутые линии

Следующие два примитива OpenGL построены на основе `GL_LINES` и позволяют задавать список вершин, по которым рисуется линия. С помощью `GL_LINE_STRIP` линия рисуется непрерывно от одной вершины до другой. Приведенный ниже код рисует два отрезка на плоскости $xу$, определяемые тремя точками. Соответствующее изображение показано на рис. 7.

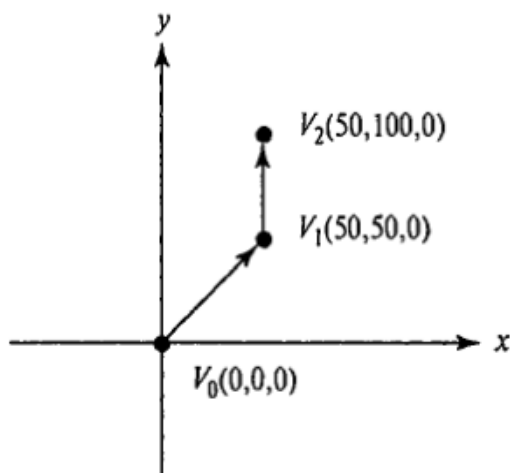


Рис.7 – Пример использования функции `GL_LINE_STRIP` с тремя вершинами

```
glBegin(GL_LINE_STRIP);
    glVertex3f(0.0f, 0.0f, 0.0f);           // V0
    glVertex3f(50.0f, 50.0f, 0.0f);        // V1
    glVertex3f(50.0f, 100.0f, 0.0f);       // V2
glEnd();
```

Последний примитив линий называется `GL_LINE_LOOP`. Этот примитив ведет себя так же, как `GL_LINE_STRIP`, только последний отрезок соединяет последнюю точку с первой. Таким образом можно рисовать замкнутые линии. Пример использования `GL_LINE_LOOP` показан на рис. 7, где задействован тот же набор вершин, что и на рис. 8.

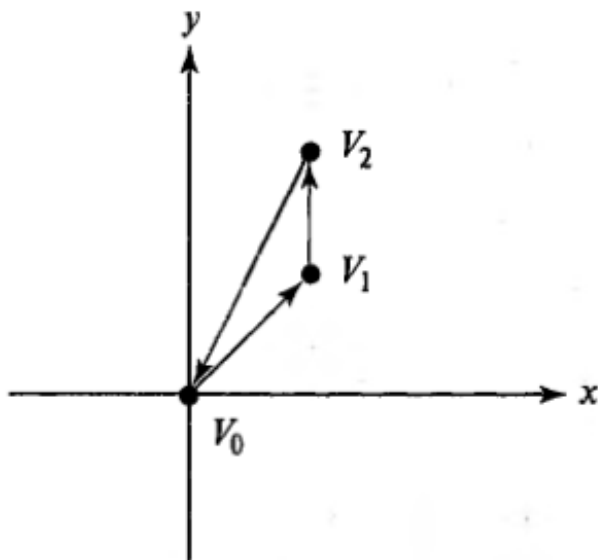


Рис.8 – Те же вершины, что и на рис. 7, используются примитивом `GL_LINE_LOOP`

Рисование треугольников в трехмерном пространстве

Итак, показано, как с помощью `GL_LINE_LOOP` рисовать точки, линии и даже замкнутые многоугольники. Используя только эти примитивы, можно легко изобразить любую возможную форму в трехмерном пространстве. Например, можно нарисовать шесть квадратов, упорядочив их так, чтобы они формировали стороны куба.

Можно заметить, что любые формы, которые создаваемые с помощью этих примитивов, не закрашены никаким цветом; в конечном счете рисуются только линии. Упорядочив в пространстве шесть квадратов, вы получите не сплошной, а каркасный куб. Чтобы нарисовать сплошную поверхность, одних точек и линий мало; нужны многоугольники. *Многоугольник* — это замкнутая форма, которая может быть закрашена текущим выбранным цветом и является основой всех твердотельных композиций в OpenGL.

Треугольники

Простейшим многоугольником является треугольник. Примитив GL_TRIANGLES рисует треугольники, соединяя три вершины. С помощью приведенного ниже кода, например, рисуется два треугольника, показанных на рис. 9.

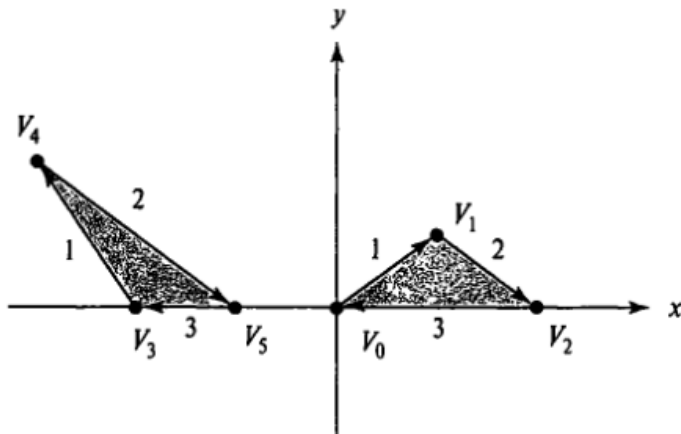


Рис.9 – Два треугольника, изображенных с помощью функции GL_TRIANGLES

```
glBegin(GL_TRIANGLES);
    glVertex2f(0.0f, 0.0f);           // V0
    glVertex2f(25.0f, 25.0f);        // V1
    glVertex2f(50.0f, 0.0f);         // V2
    glVertex2f(-50.0f, 0.0f);        // V3
    glVertex2f(-75.0f, 50.0f);       // V4
    glVertex2f(-25.0f, 0.0f);        // V5
glEnd();
```

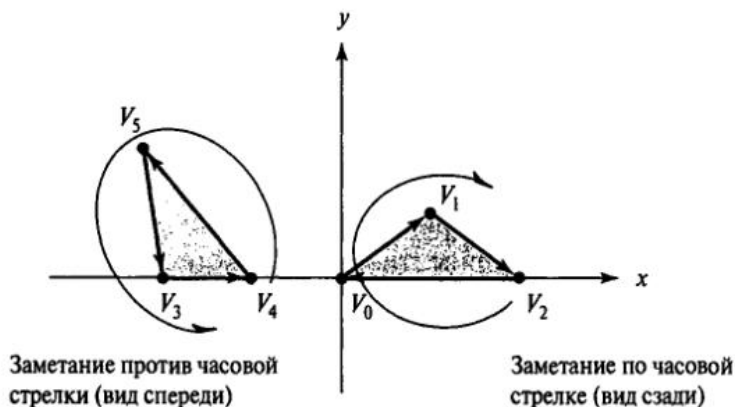


Рис.10 – Два треугольника с различным обходом

Обход

На рис. 9 иллюстрируется важная характеристика любого многоугольного примитива. Обратите внимание на стрелочки на линиях, соединяющих вершины. Когда рисуется первый треугольник, линии идут от V_0 к V_1 , затем — к V_2 , и, наконец, — обратно к V_0 , чтобы замкнуть треугольник. Этот путь проходит в порядке задания вершин, в данном примере — по часовой стрелке. Та же направленная характеристика приведена для второго треугольника.

Комбинация порядка и направления, в котором задаются вершины, называется *обходом* (winding). О треугольниках, подобных изображенным на рис. 9, говорят, что они *обходятся по часовой стрелке*. Если поменять местами положения вершин V_4 и V_5 на левом треугольнике, получим *обход против часовой стрелки*. На рис. 10, например, показаны треугольники с противоположным обходом.

В OpenGL по умолчанию считается, что передняя грань многоугольника обходится **против часовой стрелки**. Это означает, что треугольник на рис. 10 слева — передняя грань треугольника, а справа показана задняя грань треугольника.

Почему это важно? Как вскоре станет ясно, передним и задним граням многоугольников часто нужно приписать различные физические характеристики. Заднюю грань многоугольника можно вообще скрыть или наделить другими отражательными свойствами и цветом.

При этом важно следить за согласованностью всех многоугольников сцены и использовать направленные вперед многоугольники для изображения внешних поверхностей сплошных объектов.

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

```
glFrontFace(GL_CW);
```

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

Ленты треугольников

Для представления многих поверхностей и форм приходится рисовать несколько соединенных треугольников. Можно существенно сэкономить время, рисуя полосы соединенных треугольников с помощью примитива `GL_TRIANGLE_STRIP`. Процесс рисования ленты из трех треугольников, заданных пятью вершинами (от `V0` до `V4`), показан на рис. 11. Здесь видно, что вершины не обязательно обходятся в том порядке, в каком задаются. Это делается для того, чтобы сохранить обход (против часовой стрелки) всех треугольников. Структура процесса выглядит так: `V0, V1, V2`; затем `V2, V1, V3`; после этого `V2, V3, V4`; и т.д.

Далее при обсуждении многоугольных примитивов мы не будем приводить фрагменты кода, демонстрирующие вершины и операторы `glBegin`. Сейчас вы уже должны представлять себе положение вещей. Позже, когда у нас для работы будет хорошая программа, мы возобновим разбор примеров.

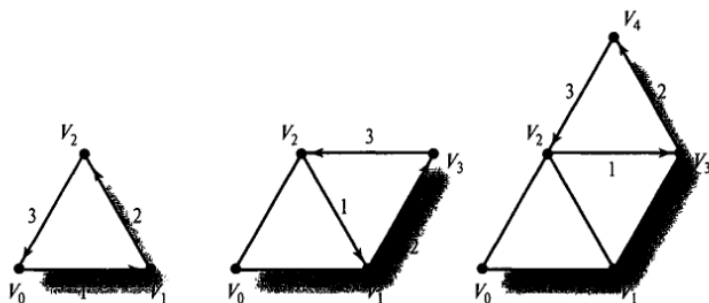


Рис.11 – Ход функции `GL_TRIANGLE_STRIP`

Существует два преимущества использования ленты треугольников перед заданием каждого треугольника отдельно. Прежде всего, задав первые три вершины исходного треугольника, для задания каждого следующего нужно указать всего лишь одну вершину. Если нарисовать нужно много треугольников, такое решение позволяет существенно сэкономить место. Вторым преимуществом является математическая производительность и экономия полосы пропускания. Меньшее количество вершин означает более быструю передачу из памяти компьютера в графическую карту и меньше преобразований вершин.

Вееры треугольников

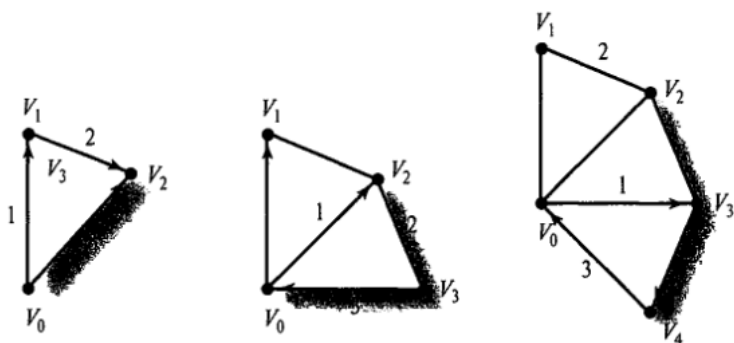


Рис.12 – Ход функции `GL_TRIANGLE_FAN`

Помимо лент треугольников можно, используя функцию `GL_TRIANGLE_FAN`, создавать вееры треугольников, расходящихся из центральной точки. Веер из трех треугольников, заданных четырьмя вершинами, показан на рис. 12. Первая вершина, V_0 , формирует начало веера. Затем на основе первых трех вершин рисуется исходный треугольник, каждая последующая вершина образует треугольник с началом (V_0) и вершиной, непосредственно ей предшествующей (V_{n-1}).

Многоугольники: режимы

Многоугольники не обязательно должны заполняться текущим цветом. По умолчанию многоугольники рисуются как сплошные объекты, но это можно изменить, указав, что многоугольники должны рисоваться в виде контуров или даже точек (изображаются только вершины). Функция `glPolygonMode` позволяет визуализировать многоугольники в форме сплошных объектов, контуров или точек-вершин.

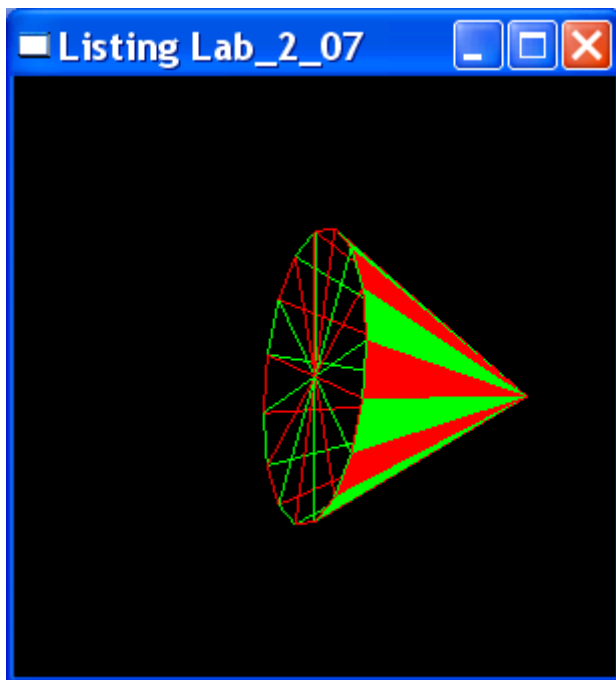


Рис.13 – Использование `glpolygonMode` для визуализации сторон треугольников в виде контуров

Кроме того, можно применить такой режим визуализации к обоим сторонам многоугольников или только к передним или задним. В приведенном ниже коде показано, как в зависимости от состояния булевой переменной `bOutline` устанавливается режим контурного или сплошного представления.

```
// Если установлена метка, рисуем заднюю сторону
// в форме каркаса
if (bOutline)
    glPolygonMode (GL_BACK, GL_LINE);
```

```
else  
    glPolygonMode(GL_BACK, GL_FILL);
```

На рис. 13 показаны задние стороны всех многоугольников, визуализированных в режиме контурного представления. (Чтобы получить это изображение, был отключен отбор; в противном случае внутренние элементы были бы удалены, и контуры не получились бы.) Обратите внимание на то, что основание конуса теперь является не сплошным, а каркасным, и можно видеть внутренние элементы конуса: внутренние стенки также представлены каркасными треугольниками.

Подробнее о полигонах

Обычно полигоны рисуются в виде закрашенной области пикселей, лежащих внутри их границ, но также они могут быть нарисованы в виде самой границы (ломаной) или просто в виде точек, соответствующих их вершинам. Возможна как полная заливка, так и применение различных шаблонов. Если смежные полигоны разделяют ребро или вершину, то пиксели, соответствующие на экране этому ребру или вершине рисуются только один раз, то есть они включаются только в один из полигонов. Это делается для того, чтобы для частично прозрачных полигонов ребра не рисовались дважды – это может привести к тому, что такие ребра будут выглядеть темнее или ярче (в зависимости от их цвета). Заметьте, что это может привести к тому, что узкие полигоны при рисовании будут иметь дыры в одном или более рядов или столбцов пикселей. Антиалиасинг для полигонов более сложен, чем для точек и отрезков.

Закрашенные полигоны, полигоны в виде линий или точек

У полигона две стороны или грани – лицевая и обратная, и он может быть визуализирован по-разному в зависимости от того, которая из сторон видна наблюдателю. Это позволяет создавать изображения таким образом, как если бы можно было заглянуть внутрь объекта у которого лицевые и обратные части рисуются по-разному (например, сфера, разрезанная пополам). Полигон также считается лицевым или обратным в зависимости от того, какой из граней он повернут к наблюдателю. По умолчанию, лицевые и обратные грани изображаются

одинаково. Чтобы это изменить, а также, чтобы рисовать только вершины или границы полигона, используйте команду **glPolygonMode()**.
`void glPolygonMode (GLenum face, GLenum mode);`

Управляет режимом отображения для лицевых и обратных граней полигонов. Параметр *face* указывает, для каких граней изменяется режим отображения и может принимать значения **GL_FRONT_AND_BACK** (режим меняется и для лицевых и для обратных граней), **GL_FRONT** (только для лицевых), **GL_BACK** (только для обратных). Параметр *mode* может быть равен **GL_POINT**, **GL_LINE** или **GL_FILL** в зависимости от желаемого режима отображения: точки, линии или заливка. По умолчанию оба типа граней рисуются в виде заполненных областей пикселей (заливки). Например, если необходимо отображать лицевые грани в виде заливки, а обратные в виде линии по границе, используйте две следующие команды:

```
glPolygonMode(GL_FRONT, GL_FILL);  
glPolygonMode(GL_BACK, GL_LINE);
```

Указание лицевых граней и удаление нелицевых граней

По принятому соглашению, грань полигона (и он сам на экране) считается лицевой, если ее вершины, начиная с первой, отображаются на экране против часовой стрелки. Можно сконструировать поверхность любого «разумного» тела (или ориентируемого тела, например, сфера, торус, чашка – ориентируемы, лист Мёбиуса – нет) из полигонов постоянной ориентации. Иными словами, вы можете сконструировать их с использованием только полигонов, чьи вершины идут по часовой стрелке или только полигонов, чьи вершины идут в обратном направлении (тело, собранное из полигонов постоянной ориентации, и называется ориентируемым).

Представьте, что долгое время описывали модель некоторой ориентируемой поверхности и вдруг обнаружили, что по ошибке разместили полигоны обратными гранями наружу (то есть гранями, вершины которых на экране идут по часовой стрелке). Можно указать OpenGL, что считать лицевыми и обратными гранями с помощью команды **glFrontFace()**.

```
void glFrontFace (GLenum mode);
```

Контролирует то, какие грани OpenGL считает лицевыми. По умолчанию *mode* равняется **GL_CCW**, что соответствует ориентации

против часовой стрелки упорядоченного набора вершин спроецированного полигона. Если в параметре *mode* указать `GL_CW`, то лицевыми будут считаться грани, чьи вершины расположены по часовой стрелке.

Если сконструированная поверхность, состоящая из непрозрачных полигонов постоянной ориентации, скажем лицевых, полностью замкнута, ни одна из обратных граней полигонов никогда не будет видна – они всегда будут закрыты лицевыми гранями. Если наблюдение происходит снаружи поверхности, можно включить механизм удаления нелицевых (обратных) граней, для того, чтобы OpenGL игнорировала и не рисовала грани, определенные ею как обратные. Аналогичным образом, если точка просмотра внутри замкнутой поверхности, видны только обратные грани, и можно заставить OpenGL рисовать только их. Чтобы заставить OpenGL игнорировать лицевые или обратные грани полигонов, используйте команду **glCullFace()** и включите удаление невидимых граней командой **glEnable()**.

```
void glCullFace (GLenum mode);
```

Задаёт, какие полигоны должны игнорироваться и удаляться до их преобразования в экранные координаты. Параметр *mode* может принимать значение `GL_FRONT`, `GL_BACK` или `GL_FRONT_AND_BACK`, что соответствует лицевым, обратным или всем полигонам соответственно. Для того, чтобы команда имела эффект, требуется включить механизм удаления нелицевых граней с помощью параметра `GL_CULL_FACE` и команды **glEnable()**. Чтобы выключить его тот же параметр должен быть передан в команду **glDisable()**.

Другие примитивы

Треугольники являются предпочтительными примитивами при формировании объектов, поскольку большая часть аппаратного обеспечения OpenGL ускоряет обработку треугольников, однако это не единственные доступные примитивы. Некоторое аппаратное обеспечение предлагает ускорение и других форм, а с точки зрения программирования использование универсальных графических примитивов может быть проще. Остальные примитивы OpenGL позволяют быстро задавать четырехугольники или ленты четырехугольников, а также универсальные многоугольники.

Четырехугольники

Если к треугольнику добавить еще одну сторону, получится четырехугольник или фигура с четырьмя сторонами. Для рисования четырехугольников в OpenGL применяется примитив **GL_QUADS**. На рис. 14 по четырем вершинам нарисован квадрат. Обратите внимание на то, что все изображенные здесь квадраты обходятся по часовой стрелке. Важно помнить, что при рисовании квадратов все четыре вершины четырехугольника должны лежать на одной плоскости (квадрат не должен быть изогнутым!).

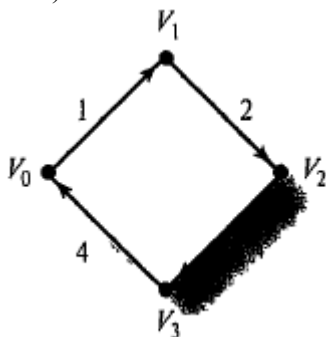


Рис.14 – Пример использования функции GL_QUADS

Ленты четырехугольников

Точно так же, как ленты треугольников, можно задавать ленты четырехугольников, применяя примитив *gl_quad_strip*. На рис. 15 показан процесс построения ленты четырехугольников, заданной шестью вершинами. Обратите внимание на то, что все эти фигуры обходятся по часовой стрелке.

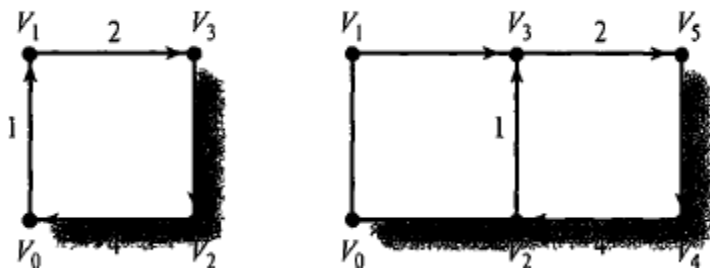


Рис.15 – Процесс построения ленты четырехугольников с помощью функции GL_QUAD_STRIP

Многоугольники общего вида

Последним примитивом OpenGL является `GL_POLYGON`, с его помощью вы можете рисовать многоугольники, имеющие любое число сторон. На рис. 16 показан многоугольник, содержащий пять вершин. Как и четырехугольники, все фигуры, нарисованные с помощью `GL_POLYGON`, должны лежать в одной плоскости. Обойти это правило проще простого — подставить `gl_triangle_fan` вместо `gl_polygon`

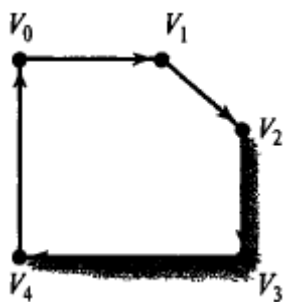


Рис.16 – Процесс изображения `GL_POLYGON`

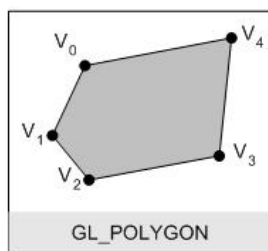
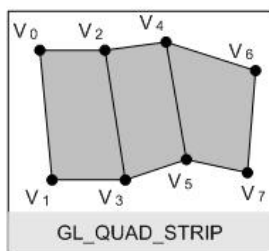
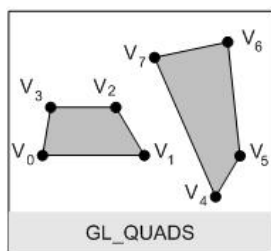
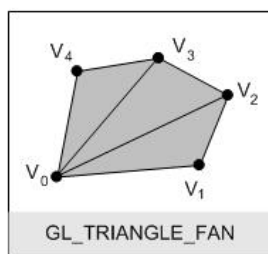
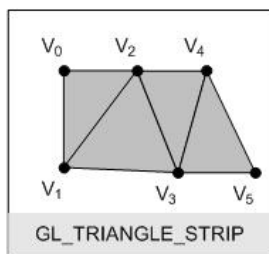
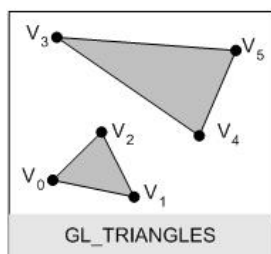
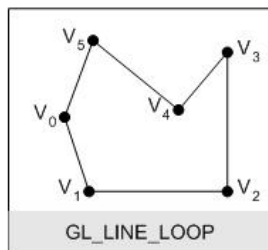
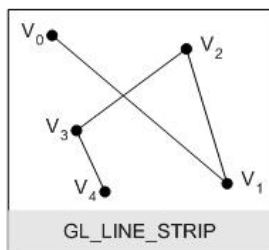
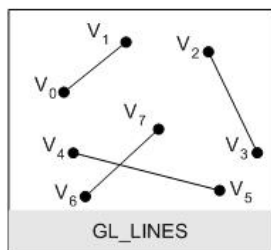
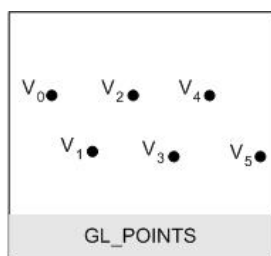
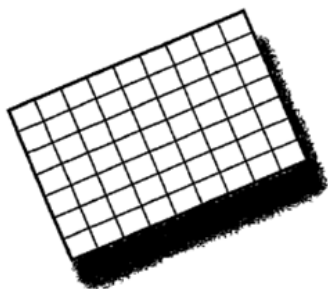


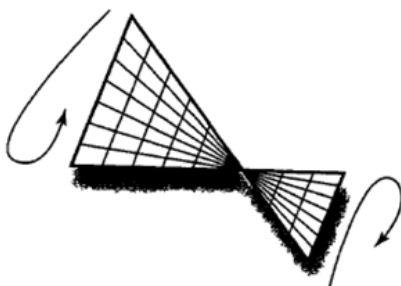
Рис.17 – Параметры для задания геометрических примитивов

Правила построения многоугольников

Когда для построения сложной поверхности вы используете большое число многоугольников, нужно помнить два важных правила.

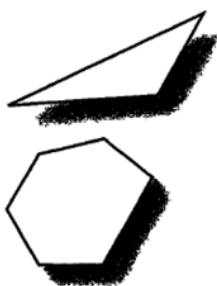


Плоский многоугольник

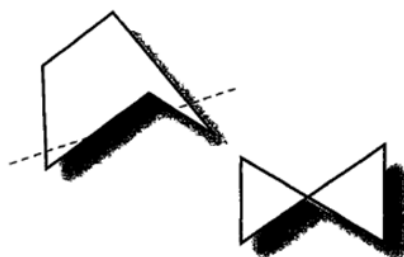


Неплоский многоугольник

Рис.18 – Плоские и неплоские многоугольники



**Приемлемый
многоугольник**



Неприемлемый многоугольник

Рис.19 – Приемлемые и неприемлемые примитивные многоугольники



Рис.20 – Вогнутая четырехконечная звезда, составленная из шести треугольников

Первое правило заключается в том, что все многоугольники должны быть плоскими. Т.е. все вершины многоугольника должны лежать на одной плоскости, как показано на рис. 18. Многоугольник не может перекручиваться или сворачиваться в пространстве.

В связи с этим можно сформулировать еще одну причину для того, чтобы использовать треугольники. Ни один треугольник нельзя пере-крутить так, чтобы три его вершины не лежали на одной плоскости, — математически для задания плоскости нужно именно три точки. (Если вы сможете нарисовать “неправильный” треугольник, то вас уже давно разыскивает Комитет по Нобелевским премиям!)

Второе правило построения многоугольников заключается в том, что стороны многоугольника не должны пересекаться, и сам он должен быть выпуклым. Многоугольник самопересекается, если пересекаются две любых его стороны. Немного сложнее проверка выпуклости: через многоугольник проводятся линии, и если какая-то из них входит и выходит из многоугольника несколько раз, многоугольник является вогнутым. Примеры “хороших” и “плохих” многоугольников приведены на рис. 19.

ЗАДАЧИ И ПОРЯДОК ВЫПОЛНЕНИЯ РАБОТЫ

Установка трехмерной канвы

На рис. 21 показан простой наблюдаемый объем. Область, обособленная этим объемом, является декартовым координатным пространством, простирающимся от -100 до +100 по всем трем осям — x , y и z . Наблюдаемый объем удобно воспринимать как трехмерную канву, в которой вы рисуете с помощью команд и функций OpenGL.

Объем устанавливается с помощью вызова `glOrtho`. В листинге 1 приведен код функции `changeSize`, которая вызывается при изменении размеров окна (в том числе в момент его первоначального создания).

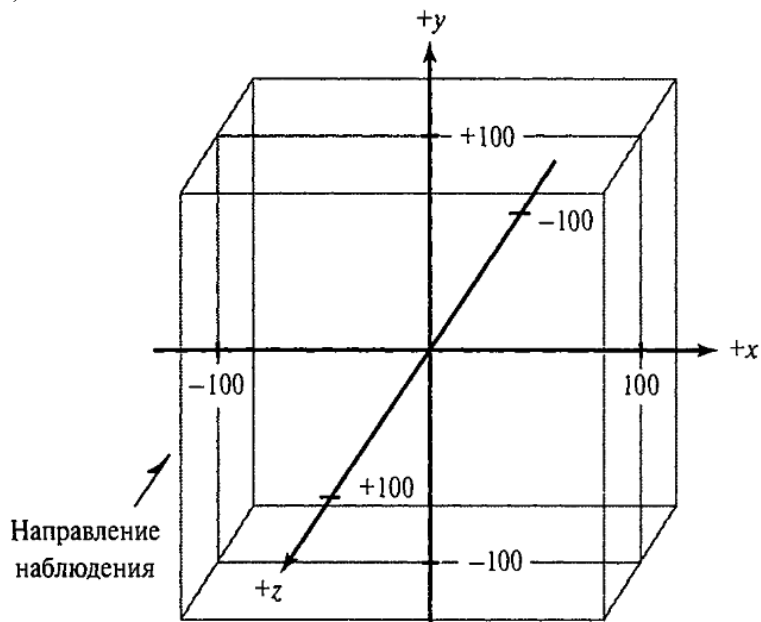


Рис.21. – Декартов наблюдаемый объем размером 100 x 100 x 100

Листинг 1 - Код, устанавливающий наблюдаемый объем (Рис.1)

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

```

{
    GLfloat nRange = 100.0f; // Предотвращает деление
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 отсе-
кающих
    // плоскостей (левая, правая, нижняя, верхняя,
    // ближняя, дальняя)
    if (w <= h)
        glOrtho (-nRange, nRange, -nRange*h/w,
nRange*h/w, -nRange, nRange);
    else
        glOrtho (-nRange*w/h, nRange*w/h, -nRange,
nRange, -nRange, nRange);
    // Обновляется стек матриц проекции модели
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
}

```

Изучив любой исходный код, можно заметить возможности функций `RenderScene: glRotate, glPushMatrix` и `glPopMatrix`. В этих функциях реализованы важные механизмы. Функции позволяют рисовать в трехмерном пространстве и помогают легко визуализировать внешний вид рисунков под различными углами.

Первый пример

Код, представленный в листинге 2, рисует несколько точек в трехмерной среде. С помощью простой тригонометрии изображается ряд точек, формирующих спиралевидную траекторию вдоль оси z . Обратите внимание на то, что в функции `SetupRC` в качестве текущего цвета рисования задан зеленый.

Листинг 2 - Код визуализации, дающий спиралеподобную точечную траекторию

```

#include "glew.h" // System and OpenGL Stuff
#include "glut.h"
#include <math.h>

// Определяется константа со значением "пи"
#define GL_PI 3.1415f

// Значения углов
static GLfloat xRot = 0.0f;
static GLfloat yRot = 0.0f;

// Вызывается для рисования сцены
void RenderScene(void)
{
    GLfloat x,y,z,angle; // Здесь хранятся координаты
и углы
    // Окно очищается текущим цветом очистки
    glClear(GL_COLOR_BUFFER_BIT);

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

    // Вызываем один раз для всех оставшихся точек
    glBegin(GL_POINTS);

    z = -50.0f;
    for(angle = 0.0f; angle <= (2.0f*GL_PI)*3.0f; an-
gle += 0.1f)
    {
        x = 50.0f*sin(angle);
        y = 50.0f*cos(angle);

        // Задаем точку и немного смещаем значение
z
        glVertex3f(x, y, z);
        z += 0.5f;
    }

    // Рисуем точки
    glEnd();
}

```

```

// Восстанавливаем преобразования
glPopMatrix();

// Очищаем стек команд преобразования
glutSwapBuffers();
}

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

    // Цвет рисования выбирается зеленым
    glColor3f(0.0f, 1.0f, 0.0f);
}

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;
}

```

```

    // Обновляем окно
    glutPostRedisplay();
}

void ChangeSize(int w, int h)
{
    GLfloat nRange = 100.0f;

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

    // Устанавливаем размеры поля просмотра равными
    // размерам окна
    glViewport(0, 0, w, h);

    // Устанавливаем перспективную систему координат
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();

    // Устанавливает объем отсечения с помощью отсе-
    // кающих
    // плоскостей (левая, правая, нижняя, верхняя,
    // ближняя, дальняя)

    if (w <= h)
        glOrtho (-nRange, nRange, -nRange*h/w,
        nRange*h/w, -nRange, nRange);
    else
        glOrtho (-nRange*w/h, nRange*w/h, -nRange,
        nRange, -nRange, nRange);

    // Обновляется стек матриц проекции модели
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
}

int main(int argc, char* argv[])
{
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB |
    GLUT_DEPTH);
    glutCreateWindow("Points Example");

```



```

glutReshapeFunc (ChangeSize);
glutSpecialFunc (SpecialKeys);
glutDisplayFunc (RenderScene);
SetupRC ();
glutMainLoop ();

return 0;
}

```

Сейчас нас интересует только код между функциями `glBegin` и `glEnd`. В этом коде рассчитываются координаты x и y угла, трижды проходящего от 0° до 360° . В программе этот угол выражается не в градусах, а в радианах. При каждом изображении точки значение z немного увеличивается. При запуске данной программы видна только окружность из точек, поскольку первоначально смотрите вдоль оси z . Эффект этого проиллюстрирован на рис. 22.

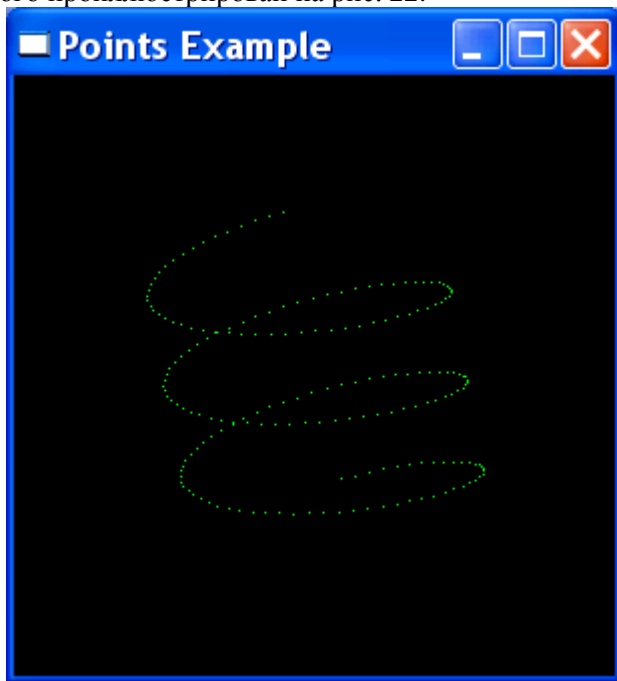


Рис.22 – Результат выполнения программы POINTS

На рисунке, представленном ниже, приведена окружность на плоскости xy . Отрезок, проходящий от начала координат $(0,0)$ к любой

точке на окружности, образует угол α с осью x . Для любого угла тригонометрические функции синус и косинус дают значения x и y точки на окружности. Пошагово меняя переменную, представляющую y , и выполнив полный оборот вокруг начала координат, можно рассчитать все точки на окружности. Обратите внимание на то, что функции времени выполнения `sin` и `cos` принимают значения углов, измеряемый в радианах, а не в градусах.

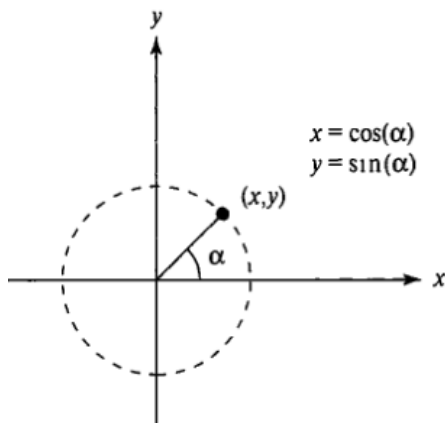


Рис.23 – Отсчет точек на окружности

Рисование точек

Начнем с первого и простейшего примитива: точек. Рассмотрим приведенный ниже код.

```
glBegin(GL_POINTS); // Выбираем примитив "точки"
glVertex3f(0.0f, 0.0f, 0.0f); // Задаем точку
glVertex3f(50.0f, 50.0f, 50.0f); // Задаем другую точку
glEnd(); // Рисуем точки
```

Аргумент `glBegin(GL_POINTS)` сообщает OpenGL, что следующие далее вершины нужно интерпретировать и рисовать как точки. В нашем примере перечислены две вершины, которые транслируются как две отдельные точки, точно так же они и рисуются.

Из данного примера можно понять один важный момент относительно `glBegin` и `glEnd`: можно перечислить несколько примитивов

в одном вызове, если они принадлежат к одному типу примитивов. Таким образом, с помощью одной последовательности glBegin/glEnd можно включить столько примитивов, сколько хотите. В следующем фрагменте кода ресурсы расходуются неэкономно, и он будет выполняться медленнее, чем код, приведенный выше.

```
glBegin(GL_POINTS);      // Задаем точку
    glVertex3f(0.0f, 0.0f, 0.0f);
glEnd();
glBegin(GL_POINTS);      // Задаем другую точку
    glVertex3f(50.0f, 50.0f, 50.0f);
glEnd()
```

Задание размера точки

Когда рисуется одна точка, ее размер по умолчанию равен одному пикселю. Изменить величину точки можно с помощью функции glPointSize.

```
void glPointSize(GLfloat size);
```

Функция glPointSize принимает один параметр, задающий приблизительный диаметр в пикселях рисуемой точки. Поддерживаются не все размеры, поэтому следует проверять, доступен ли размер, который задан для точки. Чтобы найти диапазон размеров и наименьший интервал между ними, применяется следующий код:

```
GLfloat sizes[2]; // Записываем диапазон размеров
поддерживаемых точек
GLfloat step;      // Записываем поддержи-
ваемый инкремент размеров точек
GLfloat curSize;   // Записываем размер текущих
точек
```

```
// Очищаем окно текущим цветом очистки
glClear(GL_COLOR_BUFFER_BIT);
```

```
// Записывается состояние матрицы и выполняются
повороты
```

```
glPushMatrix();
glRotatef(xRot, 1.0f, 0.0f, 0.0f);
glRotatef(yRot, 0.0f, 1.0f, 0.0f);
```

```
// Получаем диапазон размеров поддерживаемых то-
чек и размер шага
```

```
glGetFloatv(GL_POINT_SIZE_RANGE, sizes);
```

```
glGetFloatv(GL_POINT_SIZE_GRANULARITY, &step);  
  
// Задаем исходный размер точки  
curSize = sizes[0];
```



Рис.24 – Результат выполнения программы POINTSZ

Здесь массив размеров будет содержать два элемента — наименьшее и наибольшее возможное значение `glPointSize`. Кроме того, шаг переменной будет равен наименьшему шагу, возможному между размерами точек. Спецификация OpenGL требует поддержки только одного размера точек — 1,0. Программная реализация OpenGL от Microsoft, например, позволяет менять размер точек от 0,5 до 10,0 с минимальным размером шага 0,125. Задание размера, не входящего в диапазон, не интерпретируется как ошибка. Вместо этого используется наибольший или наименьший поддерживаемый размер, ближайший к заданному значению.

Точки, в отличие от других геометрических объектов, не меняются при делении на коэффициент перспективы. Т.е. они не становятся меньше при удалении от точки наблюдения, и не становятся больше при приближении к наблюдателю. Точки всегда являются квадратными. Даже используя `glPointSize` для увеличения размера точек, просто получите большие квадраты! Чтобы увидеть круглые точки, нужно использовать технику защиты от наложения.

Листинг 3 – Создание спирали из точек постепенно увеличивающегося размера

```
void RenderScene(void)
{
    GLfloat x,y,z,angle; // Место хранения координат
и углов
    GLfloat sizes[2]; //Запоминаем диапазон размеров
поддерживаемых точек

    GLfloat step;          // Запоминаем поддержи-
ваемый инкремент размеров точек
    GLfloat curSize;      // Записываем размер текущих
точек

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

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

    // Получаем диапазон размеров поддерживаемых то-
чек и размер шага

    glGetFloatv(GL_POINT_SIZE_RANGE,sizes);
    glGetFloatv(GL_POINT_SIZE_GRANULARITY,&step);

    // Задаем исходный размер точки
    curSize = sizes[0];

    // Задаем начальную координату z
    z = -50.0f;

    // Циклический проход по окружности три раза
    for(angle = 0.0f; angle <= (2.0f*3.1415f)*3.0f;
angle += 0.1f)
    {
        // Расчет значений x и y точек окружности
        x = 50.0f*sin(angle);
```

```

        y = 50.0f*cos(angle);

        // Задаем размер точки перед указанием при-
матива
        glPointSize(curSize);

        // Рисуем точку
        glBegin(GL_POINTS);
            glVertex3f(x, y, z);
        glEnd();

        // Увеличиваем значение z и размер точки
        z += 0.5f;
        curSize += step;
    }

    // Восстанавливаем преобразования
    glPopMatrix();

    // Очищаем стек команд рисования
    glutSwapBuffers();
}

```

Пример иллюстрирует несколько важных моментов. Для начала обратите внимание на то, что функцию `glPointSize` нужно вызывать вне пары операторов `glBegin/glEnd`. В такой "обложке" приемлемы не все функции OpenGL. Хотя `glPointSize` влияет на все точки, рисуемые после нее, программа не начинает рисовать точки, до вызова функции `glBegin (GL_POINTS)`.

Задав размер точки, больший того, что возвращает переменная размера, также можно наблюдать (это зависит от аппаратного обеспечения), что OpenGL использует наибольший доступный размер точки, но не увеличивает ее на изображении. Этот момент является общим для всех параметров функций OpenGL, имеющих диапазон приемлемых значений. Значения, не попадающие в этот диапазон, принудительно вводятся в него. Слишком маленькие значения превращаются в наименьшее приемлемое значение, а слишком большие — в наибольшее.

Наиболее очевидным моментом, который, возможно, отметить при запуске программы `POINTS.Z`, является то, что точки большего размера представляются просто большими кубиками. Это поведение по

умолчанию, но обычно во многих приложениях оно нежелательно. Кроме того, может возникнуть вопрос, что произойдет, если увеличить размер точки на значение, большее единицы. Если величина 1.0 представляет один пиксель, то как нарисовать меньше одного пикселя или, скажем, 2.5 пикселя?

Размер, заданный в `glPointSize`, не является точным размером точки в пикселях, а приблизительным диаметром окружности, содержащей все пиксели, используемые для рисования точки. Он указывает OpenGL рисовать точки как улучшенные (т.е. маленькие, закрашенные окружности), разрешая их сглаживание. Эта технология вместе со сглаживанием линий относится к категории *защиты от наложения* (antialiasing).

Рисование линий в трехмерном пространстве

Примитив `GL_POINTS` для каждой заданной вершины рисует точку. Следующий логический этап — задать две вершины и нарисовать отрезок между ними. Именно для этого предназначен примитив `GL_LINES`. Приведенный ниже короткий фрагмент кода рисует отрезок, соединяющий точки (0,0,0) и (50,50, 50).

```
glBegin(GL_LINES);  
    glVertex3f(0.0f, 0.0f, 0.0f);  
    glVertex3f(50.0f, 50.0f, 50.0f);  
glEnd();
```

Обратите внимание на то, что две вершины задают один примитив. Для двух заданных вершин рисуется одна линия. Если задать в `GL_LINES` нечетное число вершин, последняя из них будет проигнорирована. В листинге 4 приведен более сложный пример, в котором рисуется ряд линий, веером расходящихся из одной точки. Каждой точке в этом примере соответствует парная ей на противоположной стороне окружности. Результат выполнения программы показан на рис. 25.

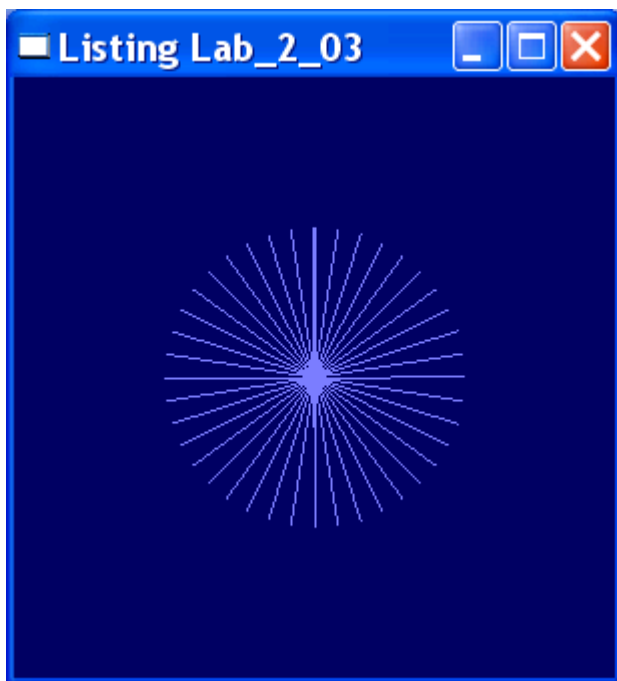


Рис.25 – Результат выполнения программы LINES

Листинг 4 – Код программы LINES, которая отображает набор линий, веером расходящихся по окружности

```
glBegin(GL_LINES);  
// Все линии принадлежат плоскости xy  
z = 0.0f;  
for(angle = 0.0f; angle <= GL_PI; angle +=  
(GL_PI/20.0f))  
{  
    // Верхняя половина окружности  
    x = 50.0f*sin(angle);  
    y = 50.0f*cos(angle);  
    glVertex3f(x, y, z);           // Первая конечная  
точка отрезка  
    // Нижняя половина окружности  
    x = 50.0f*sin(angle + GL_PI);  
    y = 50.0f*cos(angle + GL_PI);
```



```

        glVertex3f(x, y, z);    // Вторая конечная точка
отрезка
    }
    // Рисуются точки
glEnd();

```

Аппроксимация кривых прямолинейными отрезками

Программа POINTS, результат выполнения которой приводился на рис. 21, демонстрирует, как выстроить точки вдоль спиралеобразной траектории. Можно помещать точки ближе (уменьшая шаг по углу), создавая гладкую спиральную кривую вместо разорванных точек, которые только аппроксимируют ее форму. Отметим, что такая операция возможна, но для более сложных кривых из тысяч точек она будет выполняться очень медленно.

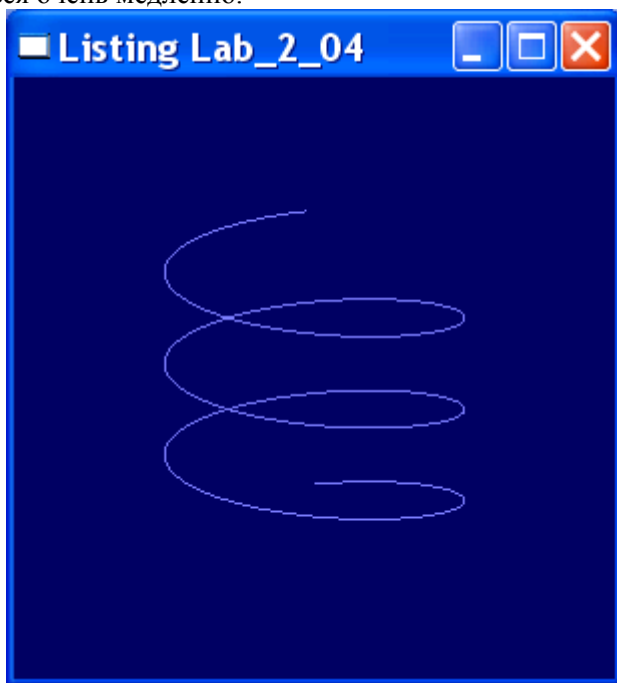


Рис.26 – Результат выполнения программы LSTRIPS, аппроксимирующей гладкую кривую

Гораздо лучшим способом аппроксимации кривой является имитация промежуточных точек с помощью функции `GL_LINE_STRIP`. При сближении точек получится более плавная кривая, и вам не придется явно задавать все ее точки. В листинге 5 показан код из листинга 2, в котором вместо `GL_POINTS` используется функция `GL_LINE_STRIP`. Результат выполнения этой новой программы `LSTRIPS` показан на рис. 26. Видно, что аппроксимация кривой достаточно хороша, так что данная удобная техника практически незаменима при работе с OpenGL.

Листинг 5 – Ломанные линии

```
// Вызывается один раз для всех точек
glBegin(GL_LINE_STRIP);
z = -50.0f;
for(angle = 0.0f; angle <= (2.0f*GL_PI)*3.0f; angle +=
0.1f)
{
    x = 50.0f*sin(angle);
    y = 50.0f*cos(angle);
    // Задаем точку и немного смещаем значение z
    glVertex3f(x, y, z);
    z += 0.5f;
}
// Рисуем точки
glEnd();
```

Задание ширины линии

Точно так же, как задавались различные размеры точек, при рисовании с помощью функции `glLineWidth` можно указывать различную ширину линий:

```
void glLineWidth(GLfloat width);
```

Функция `glLineWidth` принимает один параметр, задающий приблизительную ширину в пикселях изображаемой линии. Подобно размерам точек, поддерживается не любая ширина линий, и нужно убедиться, что необходимая ширина доступна. Чтобы определить диапазон ширин линий и наименьший интервал между ними, используйте следующий код.

```
GLfloat sizes[2];           // Записывает диапазон поддержи-
ваемой ширины линий
```

```

GLfloat step;          // Записывает поддерживаемый ин-
кремент ширины линий
// Получает диапазон и шаг поддерживаемой ширины линий
GLfloatv(GL_LINE_WIDTH_RANGE, sizes);          glGet-
Floatv(GL_LINE_WIDTH_GRANULARITY, &step);

```

Здесь массив размеров будет содержать два элемента — наименьшее и наибольшее приемлемое значение `glLineWidth`. Кроме того, переменная `step` будет содержать наименьший размер шага, допустимый между шириной линий. Спецификация OpenGL требует, чтобы поддерживалась только одна ширина линий — 1.0. Реализация OpenGL, выполненная Microsoft, позволяет использовать ширину линий от 0.5 до 10.0 с наименьшим размером шага — 0.125.

В листинге 6 приведен код более интересного примера использования `glLineWidth`. Он взят из программы `LINESW`, и при его выполнении рисуется 10 линий переменной ширины. Выполнение программы начинается с низа окна при -90 по оси *y* и каждая следующая линия располагается на 20 единиц выше и имеет ширину на 1 больше. Результат выполнения этой программы приведен на рис. 27.

Листинг 6 - Рисование линий различной ширины

```

void RenderScene(void)
{
    // Вызывается один раз для всех оставшихся точек
    GLfloat x,y,z,angle; // Здесь хранятся координаты
и углы
    GLfloat fSizes[2]; // Метрики диапазона ширины
линий
    GLfloat fCurrSize; // Запись текущего состояния

    // Очищаем окно текущим цветом очистки
    glClear(GL_COLOR_BUFFER_BIT);
    glPushMatrix();
    GLfloatv(GL_LINE_WIDTH_RANGE, fSizes);
    fCurrSize = fSizes[0];
    // Пошаговый проход оси y по 20 единиц за раз
    for (y = -90.0f; y < 90.0f; y += 20.0f)
    {
        // Задается ширина линии
        glLineWidth(fCurrSize);
        // Рисуется линия
        glBegin(GL_LINES);

```

```

        glVertex2f(-80.0f, y);
        glVertex2f(80.0f, y);
    glEnd();
    // Увеличивается ширина линии
    fCurrSize += 1.0f;
}
// Восстанавливаем преобразования
glPopMatrix();

// Очищаем стек команд рисования
glutSwapBuffers();
}

```

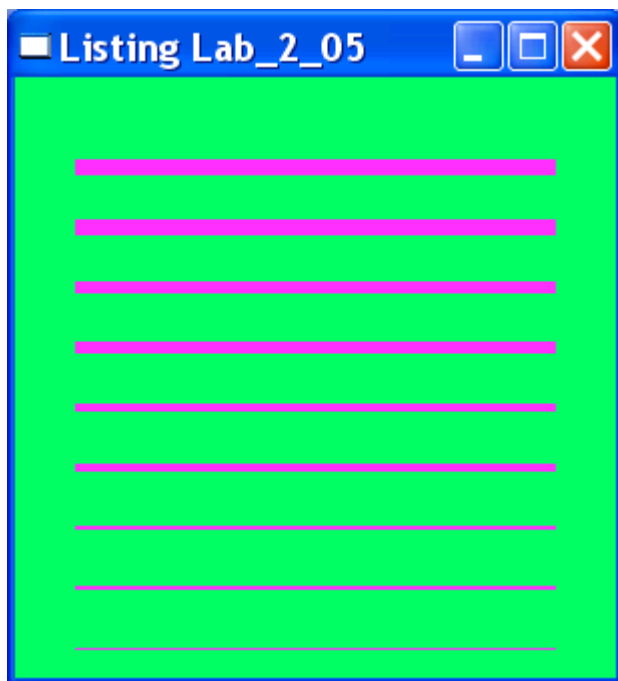


Рис.27 – Результат выполнения программы для рисования линий разной толщины

Обратите внимание на то, что, задавая координаты линий, мы использовали `glVertex2f` вместо `glVertex3f`. Как говорилось ранее, это просто договоренность, поскольку мы рисуем все объекты на плоскости xy при значении координаты z равном 0. Чтобы убедиться, что по-прежнему рисуются линии в трех измерениях, поверните рису-

нок с помощью клавиш со стрелками. Сразу заметно, что все линии лежат в одной плоскости.

Фактура линии

Помимо изменения ширины отрезков можно создавать линии со штриховыми или пунктирными узорами, называемыми *фактурой* (stippling). Чтобы использовать фактуру линий, вначале ее нужно активизировать с помощью следующей команды:

```
glEnable(GL_LINE_STIPPLE);
```

После этого функция `glLineStipple` устанавливает структуру, которая будет применяться при рисовании:

```
void glLineStipple(GLint factor, GLushort pattern);
```

Шаблон = 0X00FF = 225

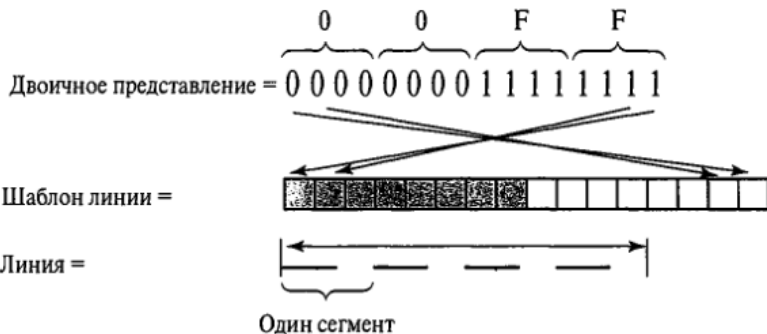


Рис.28 – Шаблон фактуры, используемый для построения отрезка

Параметр `pattern` — это 16-битовое значение, задающее шаблон, который нужно использовать при рисовании линии. Каждый бит представляет участок линии, включенный либо выключенный. По умолчанию каждый бит соответствует одному пикселю, а параметр `factor` используется как множитель, увеличивающий ширину шаблона. Например, если установить `factor` равным 5, каждый бит шаблона будет представлять пять включенных или выключенных пикселей подряд. Более того, вначале для задания линии используется нулевой бит (самый младший разряд) шаблона. Пример применения битового шаблона к отрезку показан на рис. 28.

Может возникнуть вопрос, почему при рисовании линии шаблоны фактуры используются задом наперед. Это объясняется тем, что для OpenGL гораздо быстрее смещать шаблон на одну позицию влево всякий раз, когда требуется следующее значение маски. Поскольку OpenGL разрабатывался для высокопроизводительной графики, подобные трюки встречаются в нем довольно часто.

В листинге 7 приведен пример использования шаблона фактуры, являющегося просто последовательностью чередующихся включенных и выключенных бит (01010101010101). Данный код взят из программы LSTIPPLE, которая снизу вверх изображает 10 линий, параллельных оси x . Фактура каждой линии представлена шаблоном 0x5555, но для каждой следующей линии множитель шаблона увеличивается на 1. Влияние множителя на уширяющийся фактурный шаблон можно видеть на рис. 29.

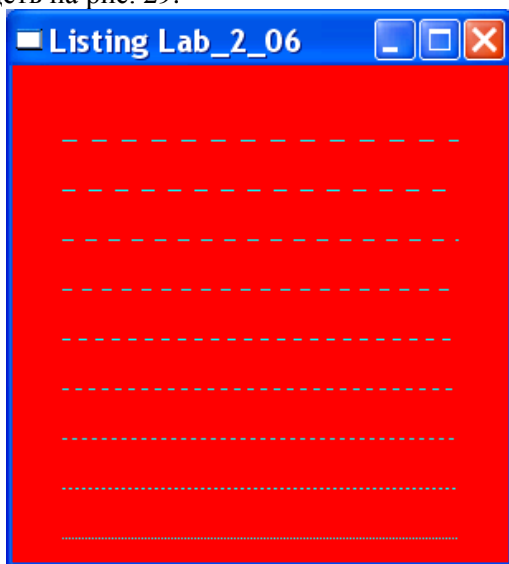


Рис.29 – Результат выполнения программы LSTIPPLE

Листинг 7 - Влияние множителя на битовый шаблон

```
void RenderScene(void)
{
    GLfloat y;          // Здесь хранятся меняющиеся
    координаты y
    GLint factor =1;    // Множитель фактуры
```

```

GLushort pattern = 0x5555; // Шаблон фактуры
glClear(GL_COLOR_BUFFER_BIT);

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

// Активизируется фактура

glEnable(GL_LINE_STIPPLE);
// Пошаговый проход оси у по 20 единиц за раз
for(y = -90.0f; y < 90.0f; y += 20.0f)
{
    // Обновляется множитель повтора и шаблон
    glLineStipple(factor, pattern);
    // Рисуются линия
    glBegin(GL_LINES);
        glVertex2f(-80.0f, y);
        glVertex2f(80.0f, y);
    glEnd();
    factor++;
}
glPopMatrix();

// Очищаем стек команд рисования
glutSwapBuffers();
}

```

Чтобы задать шаблон отрезка (например, для получения пунктирных или штриховых отрезков) следует использовать команду **glLineStipple()** и затем включить шаблонирование командой **glEnable()**.

```

glLineStipple(1, 0x3F07);
glEnable(GL_LINE_STIPPLE);
void glLineStipple (Glint factor, GLushort pattern);

```

Устанавливает текущий шаблон для отрезка. Аргумент *pattern* – это 16-битная серия из нулей и единиц, определяющая, как будет рисоваться отрезок. Она повторяется по необходимости для шаблонирования всего отрезка. Единица означает, что соответствующая точка отрезка будет нарисована на экране, ноль означает, что точка нарисована не будет (на попиксельной основе). Шаблон применяется, начи-

ная с младшего бита аргумента *pattern*. Шаблон может быть растянут с учетом значения фактора повторения *factor*. Каждый бит шаблона при наложении на отрезок расценивается как *factor* битов того же значения, идущих друг за другом. Например, если в шаблоне встречаются подряд три единицы, а затем два нуля и *factor* равен 3, то шаблон будет трактоваться как содержащий 9 единиц и 6 нулей. Допустимые значения аргумента *factor* ограничены диапазоном от 1 до 256. Шаблонирование должно быть включено передачей аргумента `GL_LINE_STIPPLE` в функцию `glEnable()`. Оно блокируется передачей того же аргумента в `glDisable()`.

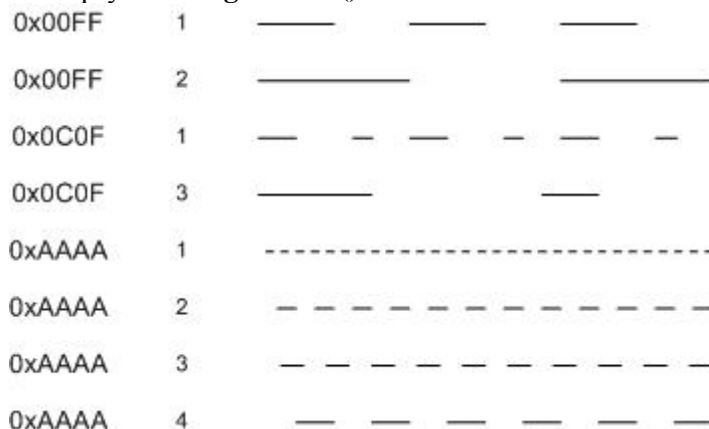


Рис.30 – Шаблонированные отрезки

Итак, в предыдущем примере с шаблоном равным `0x3F07` (что в двоичной системе счисления соответствует записи `0011111100000111`) отрезок будет выведен на экран, начинаясь (по порядку) с 3 нарисованных пикселей, 5 отсутствующих, 6 нарисованных и 2 отсутствующих (если вам кажется, что мы применили шаблон задом – наперед вспомните, что он применяется, начиная с младшего бита). Если длина отрезка на экране больше 16 пикселей, начиная с 17-го, шаблон будет применен заново и так далее до конца отрезка. Если бы *factor* был равен 2, шаблон был бы растянут, и отрезок выглядел бы следующим образом: вначале 6 нарисованных пикселей, затем 10 отсутствующих, 12 нарисованных и 4 отсутствующих.

На рис. 30 показаны отрезки, нарисованные с применением различных шаблонов и факторов повторения шаблона. Если шаблонирование заблокировано, все отрезки рисуются таким же образом, как

если бы шаблон был установлен в 0xFFFF, а фактор повторения в 1. Обратите внимание, что шаблонирование может применяться в комбинации с линиями различной толщины.

Если рисуется ломаная (с помощью GL_LINE_STRIP или GL_LINE_LOOP) то шаблон накладывается на нее непрерывно, независимо от того, где кончается один сегмент и начинается другой. В противовес этому для каждой индивидуальной линии (рисуемой с помощью GL_LINES) шаблон начинается заново, даже если все команды указания вершин вызываются внутри одного блока **glBegin()** – **glEnd()**.

Листинг 8 иллюстрирует результаты экспериментов с различными шаблонами и значениями толщины отрезков. Он также показывает разницу между рисованием ломаной и отдельных отрезков. Результаты работы программы показаны на рис. 31.

Листинг 8 – Эксперименты с ломанными и отрезками

```
#include "glew.h" // System and OpenGL Stuff
#include "glut.h"
#include <math.h>

#define drawOneLine(x1,y1,x2,y2) glBegin(GL_LINES);glVertex2f((x1),(y1));glVertex2f((x2),(y2));glEnd();
void init(void)
{
    glClearColor(1.0,0.0,1.0,0.0);
    glShadeModel(GL_FLAT);
}
void display(void)
{
    int i;
    glClear(GL_COLOR_BUFFER_BIT);
    //Черный цвет для всех линий
    glColor3f(0.0,1.0,0.0);
    glEnable(GL_LINE_STIPPLE);
    //В первом ряду три линии с разными шаблонами
    glLineWidth(1.0);
    glLineStipple(1,0x0101); //Пунктирная
    drawOneLine(50.0,125.0,150.0,125.0);
    glLineStipple(1,0x00FF); //Штриховая
    drawOneLine(150.0,125.0,250.0,125.0);
```

```

    glLineStipple(1,0x1C47); //Штрих-пунктирная
    drawOneLine(250.0,125.0,350.0,125.0);
    //Во втором ряду три толстые линии с аналогичными
шаблонами
    glLineWidth(5.0);
    glLineStipple(1,0x0101); //Пунктирная
    drawOneLine(50.0,100.0,150.0,100.0);
    glLineStipple(1,0x00FF); //Штриховая
    drawOneLine(150.0,100.0,250.0,100.0);
    glLineStipple(1,0x1C47); //Штрих-пунктирная
    drawOneLine(250.0,100.0,350.0,100.0);
    //В третьем ряду шесть штрих-пунктирных линий,
объединенных в ломаную
    glLineWidth(1.0);
    glLineStipple(1,0x1C47); //Штрих-пунктирная
    glBegin(GL_LINE_STRIP);
    for (i=0;i<7;i++)
        glVertex2f(50.0+((GLfloat)i*50.0),75.0);
    glEnd();
    //В четвертом ряду шесть независимых линий того
же шаблона
    for (i=0;i<6;i++)
    {
        drawOne-
Line(50.0+((GLfloat)i*50.0),50.0,50.0+((GLfloat)(i+1)*5
0.0),50.0);
    }
    //В пятом ряду 1 штрих-пунктирная линия с факто-
ром повторения=5
    glLineStipple(5,0x1c47);
    drawOneLine(50.0,25.0,350.0,25.0);
    glDisable(GL_LINE_STIPPLE);
    glFlush();
}
void reshape(int w,int h)
{
    glViewport(0,0,(GLsizei) w, (GLsizei) h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluOrtho2D(0.0,400.0,0.0,150.0);
}
int main(int argc, char **argv)
{
    glutInit(&argc,argv);

```

```

glutInitDisplayMode(GLUT_SINGLE|GLUT_RGB);
glutInitWindowSize(400,150);
glutInitWindowPosition(100,100);
glutCreateWindow("Line Stipple Patterns");
init();
glutDisplayFunc(display);
glutReshapeFunc(reshape);
glutMainLoop();
return 0;
}

```

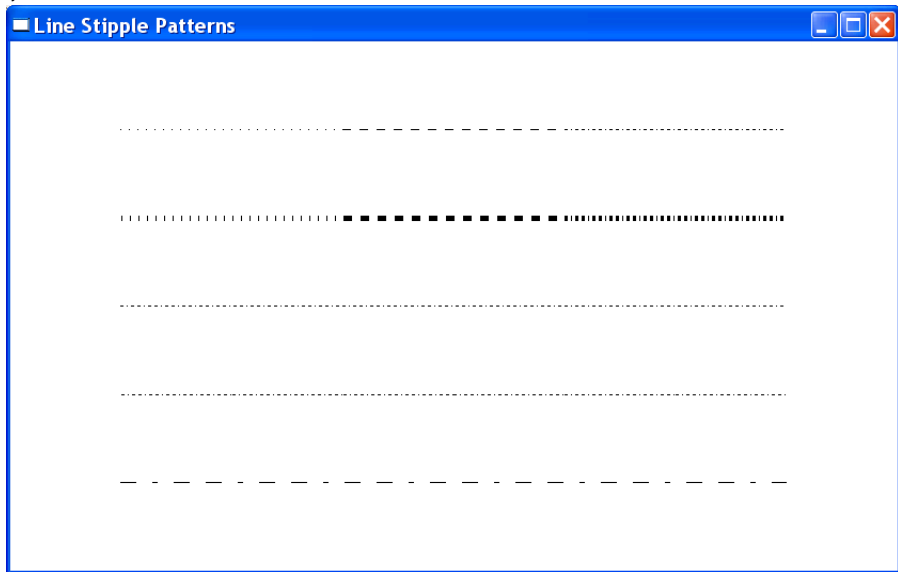


Рис.31 – Результаты экспериментов с различными шаблонами и значениями толщины отрезков

Построение сплошных объектов

Составление сплошных объектов из треугольников (или любых других многоугольников) включает не только сборку наборов вершин в трехмерном координатном пространстве. Рассмотрим, например, простую программу TRIANGLE, в которой с помощью двух веев треугольников создается конус в наблюдаемом объеме. Первый веев задает форму конуса, используя первую точку в качестве вершины конуса, а все остальные — как точки на окружности, удаленной от

наблюдателя в направлении оси z . Второй веер формирует окружность и целиком лежит в плоскости xy , образуя основание конуса.

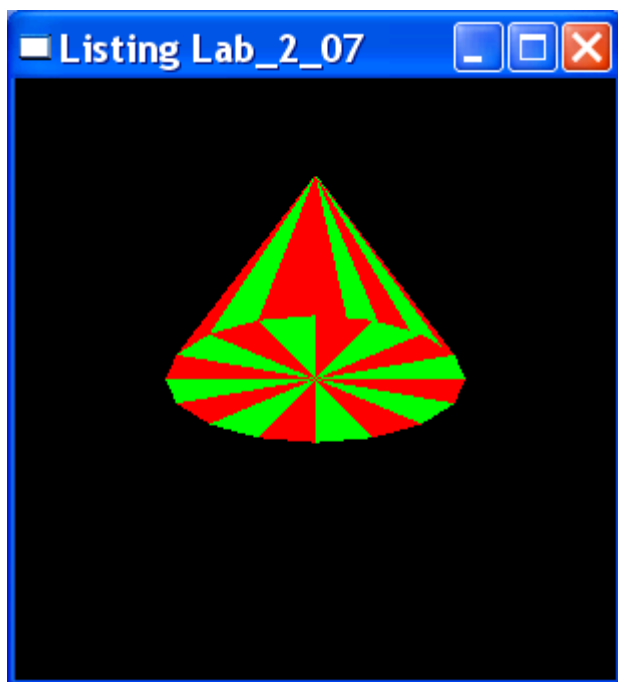


Рис.32 – Первоначальный результат выполнения программы TRIANGLE

Результат выполнения программы TRIANGLE показан на рис. 32. Здесь вы смотрите вдоль оси z и можете видеть только окружность, составленную из веера треугольников. Отдельные треугольники выделены цветом, поэтому при запуске программы мы наблюдаем чередующиеся фрагменты зеленого и красного цвета.

Код функций SetupRC и RenderScene продемонстрирован в листинге 9. На примере данной программы демонстрируется несколько аспектов составления трехмерных объектов. Щелкая правой кнопкой на окне, вы получите меню Effects, с помощью которого можно активизировать и деактивизировать определенные особенности трехмерного рисунка, что поможет исследовать некоторые характеристики создания трехмерных объектов. Эти особенности разберем ниже.

Листинг 9 – Код функций SetupRC и RenderScene

```
void SetupRC()
{
    // Черный фон
    glClearColor(0.0f, 0.0f, 0.0f, 1.0f );
    // Цвет рисования выбирается зеленым
    glColor3f(0.0f, 1.0f, 0.0f);
    // Цвет модели затенения выбирается неструктури-
    рованным
    glShadeModel(GL_FLAT);
    // Многоугольники с обходом по часовой стрелке
    считаются
    // направленными вперед; поведение изменено на
    обратное,
    // поскольку мы используем веры треугольников
    glFrontFace(GL_CW);
}

// Вызывается для рисования сцены
void RenderScene(void)
{
    GLfloat x,y,angle; // Здесь хранятся координаты
    и углы
    int iPivot =1;      // Используется, чтобы от-
    мечать
                        // чередующиеся цвета
    // Очищаем окно и буфер глубины
    glClear(GL_COLOR_BUFFER_BIT |
    GL_DEPTH_BUFFER_BIT) ;
    // Включаем отбор, если установлена метка
    if(bCull)
        glEnable(GL_CULL_FACE);
    else
        glDisable(GL_CULL_FACE);
    // Если установлена метка, активизируем проверку
    глубины
    if(bDepth)
        glEnable(GL_DEPTH_TEST) ;
    else
        glDisable(GL_DEPTH_TEST);
    // Если установлена метка, рисуем заднюю сторону
    // в форме каркаса
```

```

    if(bOutline)
        glPolygonMode(GL_BACK, GL_LINE);
    else
        glPolygonMode(GL_BACK, GL_FILL);
    // Записываем состояние матрицы и выполняем пово-
рот
    glPushMatrix();
    glRotatef(xRot, 1.0f, 0.0f, 0.0f);
    glRotatef(yRot, 0.0f, 1.0f, 0.0f);
    // Начинаем веер треугольников
    glBegin(GL_TRIANGLE_FAN);
    // Вершина конуса является общей вершиной веера.
Перемещаясь
    // вверх по оси z, вместо окружности получаем ко-
нус
    glVertex3f(0.0f, 0.0f, 75.0f);
    // По циклу проходим окружность и задаем четные
точки вдоль
    // окружности как вершины веера треугольников
    for(angle = 0.0f; angle < (2.0f*GL_PI); angle +=
(GL_PI/8.0f))
    {
        // Рассчитываем положения x и y следующей
вершины
        x = 50.0f*sin(angle);
        y = 50.0f*cos(angle);
        // Чередуем красный и зеленый цвет
        if((iPivot %2) == 0)
            glColor3f(0.0f, 1.0f, 0.0f);
        else
            glColor3f(1.0f, 0.0f, 0.0f);
        // Увеличиваем pivot на 1, чтобы в следую-
щий раз
        // изменить цвет
        iPivot++;
        // Задаем следующую вершину веера треуголь-
ников
        glVertex2f(x, y);
    }
    // Рисуем веер, имитирующий конус
    glEnd();
    // Начинаем новый веер треугольников, имитирующий
основание
    // конуса

```

```

    glBegin(GL_TRIANGLE_FAN);
    // Центром веера является начало координат
    glVertex2f(0.0f, 0.0f);
    for(angle = 0.0f; angle < (2.0f*GL_PI); angle +=
(GL_PI/8.0f))
    {
        // Рассчитываем координаты x и y следующей
вершины
        x = 50.0f*sin(angle);
        y = 50.0f*cos(angle);
        // Чередуем красный и зеленый цвета
        if((iPivot %2) == 0)
            glColor3f(0.0f, 1.0f, 0.0f);
        else
            glColor3f(1.0f, 0.0f, 0.0f);
        // Увеличиваем pivot на единицу, чтобы в
следующий раз
        // поменять цвета
        iPivot++;
        // Задаем следующую вершину веера треуголь-
ников
        glVertex2f(x, y);
    }
    // Рисуем веер, имитирующий основание конуса
    glEnd();
    // Восстанавливаем преобразования
    glPopMatrix();
    // Очищаем стек команд рисования
    glutSwapBuffers();
}

```

Установка цвета многоугольника

До этого момента мы устанавливали текущий цвет только один раз и рисовали единственную форму. Теперь, когда работа производится с несколькими многоугольниками, ситуация становится интереснее. Требуется использовать несколько различных цветов, чтобы результаты работы выглядели нагляднее. В действительности цвета задаются для вершин, а не для многоугольников. Согласно модели затенения, многоугольник может закрашиваться сплошным цветом (используется текущий цвет, выбранный при задании последней вершины) или глад-

ко затеняться с переходом друг в друга цветов, заданных для различных вершин.

Строка

```
glShadeModel(GL_FLAT);
```

сообщает OpenGL заполнить многоугольники сплошным цветом, который был текущим на момент задания последней вершины многоугольника.

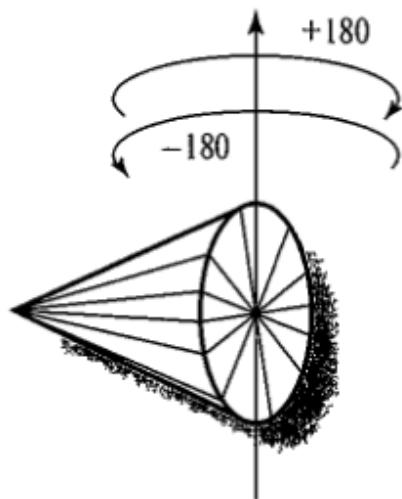


Рис.33 – Вращающийся конус кажется качающимся взад-вперед

Именно поэтому мы могли так просто менять текущий цвет на красный или зеленый перед указанием новой вершины веера треугольников. С другой стороны, строка

```
glShadeModel(GL_SMOOTH);
```

указывает OpenGL затенить треугольники плавно, пытаясь интерполировать цвета точек, находящихся между вершинами заданного цвета.

Удаление скрытых поверхностей

Нажмите одну из клавиш со стрелкой, чтобы начать вращение конуса, и больше не выбирайте ничего из меню *Effects*. Можно заметить, что конус качается взад-вперед на плюс и минус 180° , а основание конуса всегда смотрит на зрителя, и не поворачивается на 360° . Более отчетливо эффект виден на рис. 33.

Эффект качки создается из-за того, что основание конуса рисуется после боковых граней. Не имеет значения, как ориентирован конус, основание рисуется поверх него, создавая иллюзию "качки". Если нарисовано несколько объектов, причем один расположен перед другим (с позиции наблюдателя), последний нарисованный объект кажется расположенным перед изображенными ранее.

Это можно исправить, добавив проверку глубины. *Проверка глубины* — это эффективная технология удаления скрытых (или невидимых) поверхностей, и в OpenGL имеются соответствующие функции, выполняющие необходимые уточнения рисунка. Принцип прост: когда пиксель рисуется, ему присваивается значение (глубина z), характеризующее расстояние от положения наблюдателя. Когда позже в этой же точке нужно будет нарисовать другой пиксель, глубина z этого нового пикселя сравнивается с записанным значением. Если новое значение больше, значит, точка расположена ближе к наблюдателю, т.е. перед предыдущей точкой, а следовательно, старая точка затеняется новой. Если новое значение меньше, соответствующая точка расположена позади существующего пикселя, а следовательно, не наблюдается. Данный "маневр" выполняет внутренний *буфер глубины*, где хранятся глубины всех пикселей экрана.

Чтобы активизировать проверку глубины, нужно вызывать такую функцию:

```
glEnable(GL_DEPTH_TEST);
```

В программе, приведенной в листинге 9 проверка глубины была активизирована при присвоении переменной `bDepth` значения `True` и деактивизирована при присвоении той же функции `bDepth` значение `False`.

```
// Если установлена метка, активизируем проверку глубины
```

```
    if (bDepth)
        glEnable(GL_DEPTH_TEST) ;
    else
        glDisable(GL_DEPTH_TEST);
```

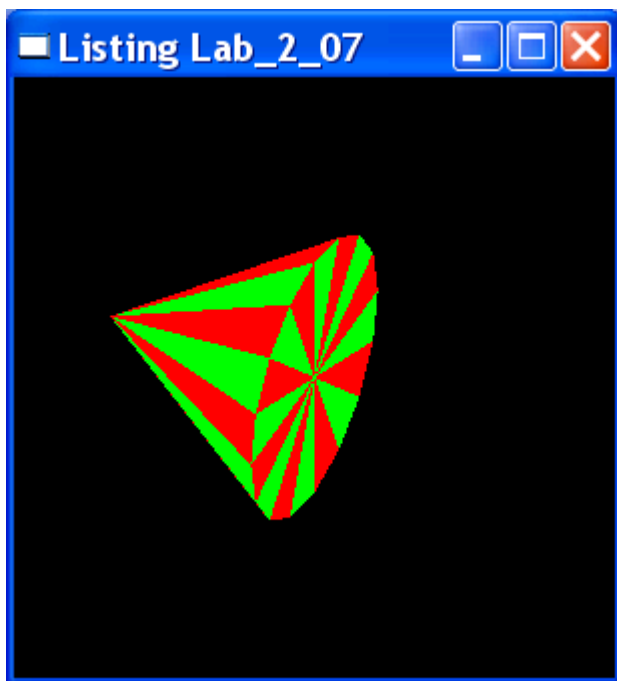


Рис.34 – При указанной ориентации основание конуса теперь правильно помещается за сторонами

Нужное значение переменной `bDepth` устанавливается при выборе команды `Depth Test` из меню `Effects`. Кроме того, при каждой визуализации сцены нужно очищать буфер глубины. Этот буфер глубины является аналогом буфера цвета в том смысле, что содержит информацию о расстоянии пикселей от наблюдателя. На основе этой информации определяется, не скрыты ли какие-то пиксели другими точками, расположенными ближе к наблюдателю.

```
// Очищаем окно и буфер глубины
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT)
;
```

Щелчок правой кнопкой мыши открывает меню, позволяющее включать и выключать проверку глубины. На рис. 34 показан результат выполнения программы `TRIANGLE` с активизированной проверкой глубины. Здесь также видно, как основание конуса правильно заслоняется боковыми гранями. Из приведенного примера видно, что

проверка глубины просто необходима при создании трехмерных объектов из сплошных многоугольников.

Отбор: повышение производительности за счет скрытых поверхностей

Из сказанного должно быть понятно, что с точки зрения визуального восприятия не нужно рисовать поверхность, заслоняемую другой. Тем не менее даже в этом случае получаются некоторые издержки, поскольку каждый нарисованный пиксель должен сравниваться со значением z предыдущего пикселя. В то же время иногда известно, что поверхность ни при каких условиях не будет рисоваться, так зачем ее вообще задавать? *Отбором* (culling) называется технология удаления геометрических элементов, которые зритель никогда не увидит. Не передавая эти элементы аппаратному обеспечению и драйверу OpenGL, мы существенно повышаем производительность. Одной из разновидностей отбора является отбор задних граней — удаление задних сторон поверхности.

В рассматриваемом примере конус является замкнутой поверхностью, и зритель никогда не увидит его внутреннюю часть. В действительности OpenGL (внутренне) рисует задние стенки дальней стороны конуса, а затем — передние стенки многоугольников, направленных на нас. Следовательно, на основании сравнений значений в буфере глубины либо игнорируется дальняя сторона конуса, либо поверх нее что-то рисуется. На рис. 35 показан конус с определенной ориентацией с включенной (а) и выключенной (б) проверкой глубины. Обратите внимание на то, что при активизированной проверке глубины меняются зеленые и красные треугольники, составляющие конус. Без проверки глубины просматриваются стороны треугольников дальней грани конуса.

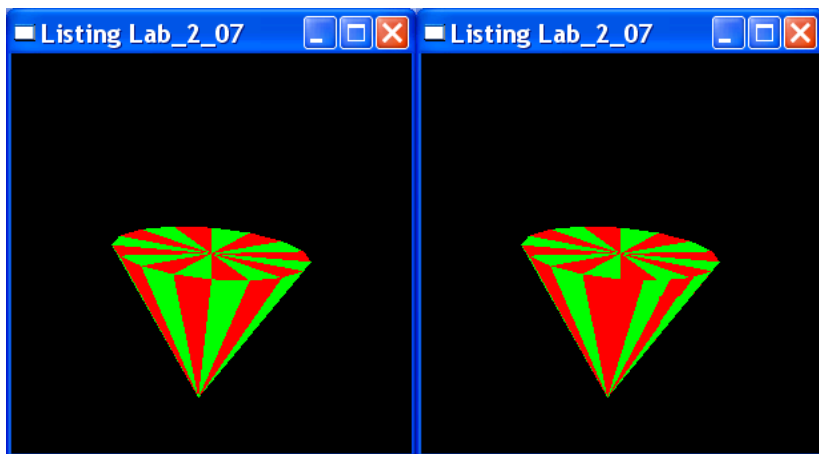


Рис.35 – Пирамида: с проверкой и без проверки глубины

Ранее объяснялось, как OpenGL с помощью обхода определяет передние и задние стороны многоугольников, и отмечалось, что важно поддерживать согласованность многоугольников, определяющих внешнюю сторону объектов. Согласованность позволяет указывать OpenGL, что визуализировать нужно только переднюю часть, только часть или обе стороны многоугольника. Удаляя задние стороны многоугольников, можно существенно уменьшить объем обработки при визуализации изображения. Хотя вследствие проверки глубины внутренняя часть объектов будет в любом случае удалена, при обработке OpenGL должен их учитывать, если мы явно не укажем, что этого делать не нужно.

Отбор задних граней активизируется или деактивируется с помощью следующего кода ([см. листинг 9](#)).

```
void SetupRC()
{
    // Черный фон
    glClearColor(0.0f, 0.0f, 0.0f, 1.0f );
    // Цвет рисования выбирается зеленым
    glColor3f(0.0f, 1.0f, 0.0f);
    // Цвет модели затенения выбирается неструктури-
    рованным
    glShadeModel(GL_FLAT);
    // Многоугольники с обходом по часовой стрелке
    считаются
```

```

        // направленными вперед; поведение изменено на
        обратное,
        // поскольку мы используем вееры треугольников
        glFrontFace(GL_CW);
    }

.....
// Включаем отбор, если установлена метка
    if(bCull)
        glEnable(GL_CULL_FACE);
    else
        glDisable(GL_CULL_FACE);

```

Обратите внимание на то, что вначале меняется определение направленных вперед многоугольников, предполагая обход по часовой стрелке (так как все рассматриваемые вееры треугольников обходятся по часовой стрелке).

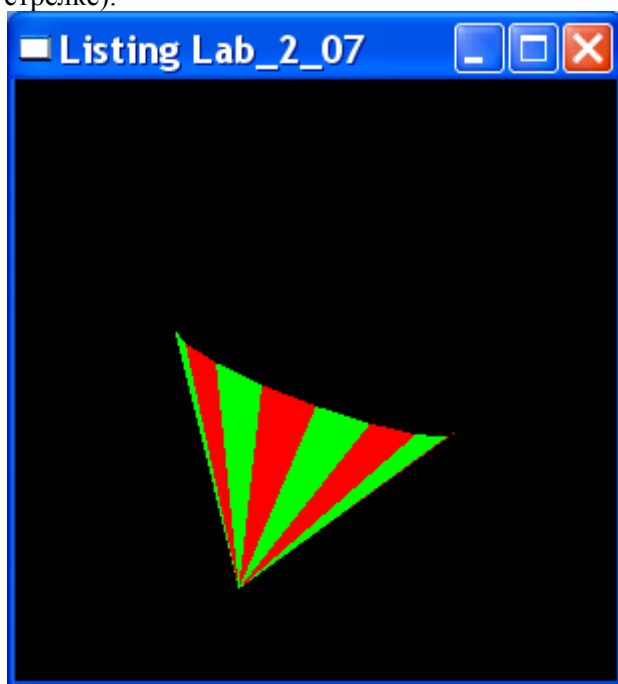


Рис.36 – Основание конуса отбирается, поскольку треугольники, направленные вперед, находятся внутри



Рис.37 – Как конус собирается из двух треугольных вееро

На рис. 36 показано, что при активизированном отборе основание конуса исчезает. Причина такого поведения кроется в том, что мы не придерживаемся собственного правила, что все поверхностные многоугольники обходятся в одном направлении. Веер треугольников, образующий основание конуса, обходится по часовой стрелке, подобно вееру, образующему стороны конуса, но в таком случае передняя сторона фрагмента основания конуса направлена внутрь (см. рис. 37).

Чтобы исправить эту ошибку, можно изменить правило обхода `glFrontFace(GL_CCW)` ;

непосредственно перед рисованием второго веера треугольников. Однако в данном примере требовалось показать отбор в действии и подготовить вас к следующему этапу обработки многоугольников.

Заполнение многоугольников, или возвращаясь к фактуре

Существует два метода наложения узора на сплошные многоугольники. Обычно применяется наложение текстуры, когда изображение отображается на поверхность многоугольника. Другой способ заключается в задании фактурного узора, как это было для линий. Фактурный узор многоугольника — это не более, чем монохромное растровое изображение 32 x 32, используемое как узор-заполнитель. Чтобы

активизировать заполнение многоугольника фактурой, нужно вызывать следующую функцию: `glEnable(GL_POLYGON_STIPPLE);`

После этого вызывается такая функция:

`glPolygonStipple (pBitmap);`

`pBitmap` — это указатель на область данных, содержащую узор-заполнитель. Далее все многоугольники заполняются с помощью узора, заданного функцией `pBitmap (GLubyte *)`. Этот узор подобен используемому в наложении фактуры на линию, только в этот раз буфер должен быть достаточно большим, чтобы вместить узор 32 x 32. Кроме того, биты считываются начиная со старшего разряда, т.е. в обратном, по сравнению с линиями, порядке. На рис. 38 показан растровый образ костра, который используется как узор-заполнитель.

Чтобы построить маску, представляющую данный узор, записывают его снизу вверх по одной строке. К счастью, в отличие от узоров-заполнителей линий данные по умолчанию интерпретируются так, как записаны: первым читается самый старший бит. Таким образом, слева направо можно прочесть все байты и сохранить их в массиве величин типа `GLubyte`, достаточно большом, чтобы вместить 32 строки по 4 байт в каждой.

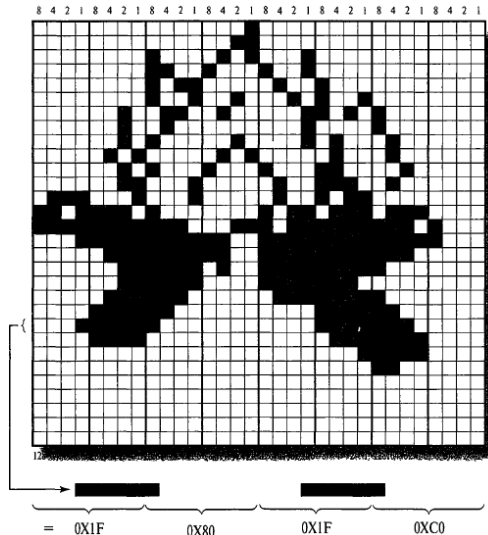


Рис.38 – Построение узора-заполнителя многоугольника

В листинге 10 приведен код, использованный для записи указанного узора. Каждая строка массива представляет строку рис. 38. Первая

строка массива является последней строкой рисунка, а последняя строка массива — первой строкой рисунка.

Листинг 10 - Определение маски костра

```
// Определяется константа со значением "пи"
#define GL_PI 3.1415f

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

// Растровый образ костра
GLubyte fire[128] = { 0x00, 0x00, 0x00, 0x00,
                      0x00, 0x00, 0x00, 0x00,
                      0x00, 0x00, 0x00, 0x00,
                      0x00, 0x00, 0x00, 0x00,
                      0x00, 0x00, 0x00, 0x00,
                      0x00, 0x00, 0x00, 0xc0,
                      0x00, 0x00, 0x01, 0xf0,
                      0x00, 0x00, 0x07, 0xf0,
                      0x0f, 0x00, 0x1f, 0xe0,
                      0x1f, 0x80, 0x1f, 0xc0,
                      0x0f, 0xc0, 0x3f, 0x80,
                      0x07, 0xe0, 0x7e, 0x00,
                      0x03, 0xf0, 0xff, 0x80,
                      0x03, 0xf5, 0xff, 0xe0,
                      0x07, 0xfd, 0xff, 0xf8,
                      0x1f, 0xfc, 0xff, 0xe8,
                      0xff, 0xe3, 0xbf, 0x70,
                      0xde, 0x80, 0xb7, 0x00,
                      0x71, 0x10, 0x4a, 0x80,
                      0x03, 0x10, 0x4e, 0x40,
                      0x02, 0x88, 0x8c, 0x20,
                      0x05, 0x05, 0x04, 0x40,
                      0x02, 0x82, 0x14, 0x40,
                      0x02, 0x40, 0x10, 0x80,
                      0x02, 0x64, 0x1a, 0x80,
                      0x00, 0x92, 0x29, 0x00,
                      0x00, 0xb0, 0x48, 0x00,
                      0x00, 0xc8, 0x90, 0x00,
                      0x00, 0x85, 0x10, 0x00,
```



```

                                0x00, 0x03, 0x00, 0x00,
                                0x00, 0x00, 0x10, 0x00 };

//Вызывается для рисования сцены
void RenderScene(void)
{
    // Очищаем окно, используя текущий цвет очистки
    glClear(GL_COLOR_BUFFER_BIT);

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

    // Начинает рисовать форму знака "Стоп",
    // используя для простоты правильный восьмиуголь-
ник
    glBegin(GL_POLYGON);
        glVertex2f(-20.0f, 50.0f);
        glVertex2f(20.0f, 50.0f);
        glVertex2f(50.0f, 20.0f);
        glVertex2f(50.0f, -20.0f);
        glVertex2f(20.0f, -50.0f);
        glVertex2f(-20.0f, -50.0f);
        glVertex2f(-50.0f, -20.0f);
        glVertex2f(-50.0f, 20.0f);
    glEnd();

    // Восстанавливаем преобразования
    glPopMatrix();

    // Очищаем стек команд рисования
    glutSwapBuffers();
}

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

```

```

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

        // Активизируется заполнение многоугольника
        glEnable(GL_POLYGON_STIPPLE);

        // Задается узор-заполнитель
        glPolygonStipple(fire);
    }

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;

    // Обновляем окно
    glutPostRedisplay();
}

```

```

void ChangeSize(int w, int h)
{
    GLfloat nRange = 100.0f;

    // Предотвращает деление на нуль, когда окно
слишком маленькое
    if(h == 0)
        h = 1;

    // Размер поля просмотра устанавливается равным
размеру окна
    glViewport(0, 0, w, h);

    // Обновляется система координат
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();

    // Устанавливает объем отсечения с помощью отсе-
кающих
    // плоскостей (левая, правая, нижняя, верхняя,
    // ближняя, дальняя)
    if (w <= h)
        glOrtho (-nRange, nRange, -nRange*h/w,
nRange*h/w, -nRange, nRange);
    else
        glOrtho (-nRange*w/h, nRange*w/h, -nRange,
nRange, -nRange, nRange);

    // Обновляется стек матриц проекции модели
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
}

int main(int argc, char* argv[])
{
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB |
GLUT_DEPTH);
    glutCreateWindow("Polygon Stippling");
    glutReshapeFunc(ChangeSize);
    glutSpecialFunc(SpecialKeys);
    glutDisplayFunc(RenderScene);
    SetupRC();
}

```

```
glutMainLoop();

return 0;
}
```

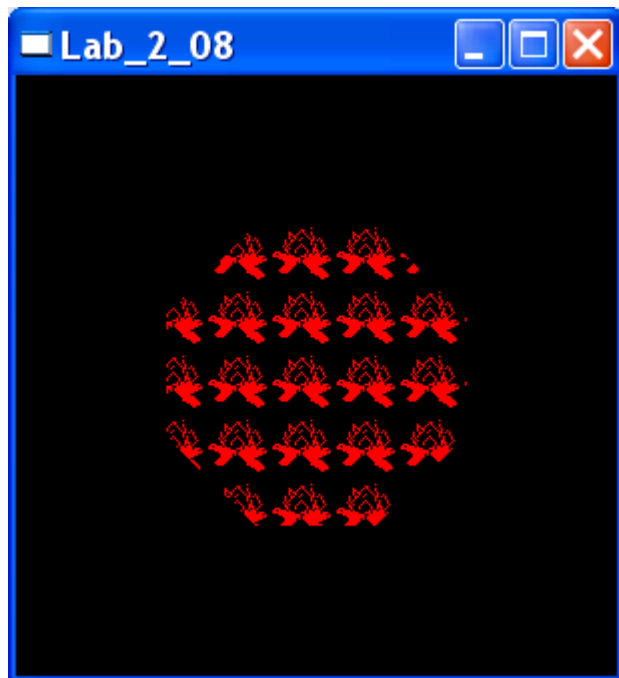


Рис.39 – Результат выполнения программы PSTIPPLE

Чтобы использовать узор-заполнитель, нужно вначале активизировать заполнение многоугольников, а затем указать этот узор в качестве узора-заполнителя. Все это делается в начале программы ***pstipple***, после чего с помощью узора-заполнения рисуется восьмиугольник. Результат выполнения программы ***pstipple*** показан на рис. 39.

На рис. 40 показан немного повернутый восьмиугольник. Обратите внимание на то, что узор-заполнитель все еще используется, но он не поворачивается вместе с многоугольником. Узор-заполнитель применяется только для простого заполнения многоугольников на экране.

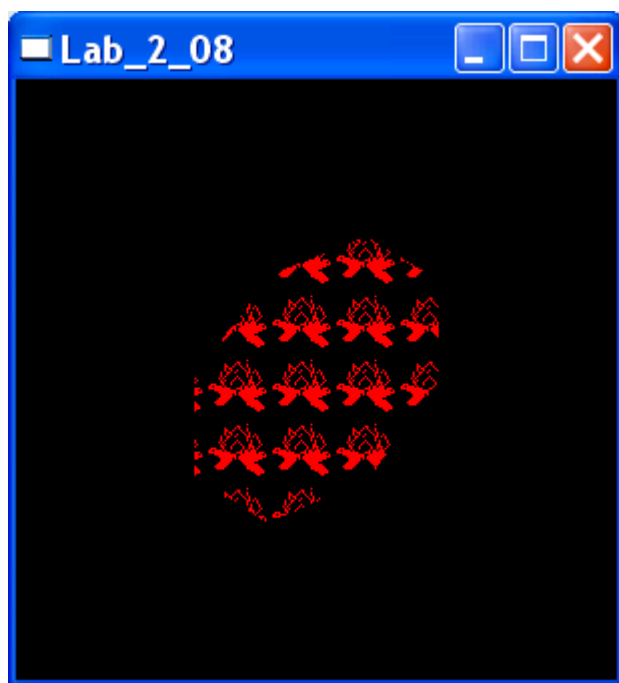


Рис.40 – Результат выполнения функции PSTIPPLE с повернутым многоугольником, из которого видно, что узор-заполнитель не поворачивается вместе с объектом

ЗАДАНИЕ НА ЛАБОРАТОРНУЮ РАБОТУ

Воспроизвести результаты, представленные в теоретическом обзоре, применить различные способы описания графических примитивов в трехмерном пространстве, научиться изменять настройки помещения точек в конвейер обработки.

ТРЕБОВАНИЯ К РЕАЛИЗАЦИИ

Задание выполняется согласно варианту. По завершении готовится отчет.

- 1) Для каждого приложения установить размеры наблюдаемого объема.
- 2) Для [Листинга 1](#) задать с использованием тригонометрических преобразований (функций \sin и \cos) произвольное множество точек (не менее 30 точек) в трехмерном пространстве. Изучить принципы вращения экрана с помощью клавиш курсора и способы подключения меню к программе.
- 3) Для [Листинга 2](#) задать произвольное множество точек разного размера и цвета в трехмерном пространстве.
- 4) Для [Листинга 3](#) изобразить не менее 10 линий с разными параметрами (цвет, толщина, начертание по шаблону) в трехмерном пространстве.
- 5) Для [Листинга 4](#) выполнить аппроксимацию произвольной кривой линии посредством коротких прямых. На 3 примерах наглядно продемонстрировать точность аппроксимации. По возможности использовать меню.
- 6) Для [Листинга 5](#), используя `GL_LINE_STRIP` и `GL_LINE_LOOP` нарисовать трехмерный объект из ломанных линий содержащих не менее 15 точек. Точки задавать, используя цикл.
- 7) Для [Листинга 6](#) используя `GL_TRIANGLES` вывести на экран произвольную трехмерную геометрическую фигуру состоя-

щую из треугольников. Понимать принципы обхода точек треугольника и полигонов и знать на что он влияет.

- 8) Для [Листинга 7](#) продемонстрировать работу директив [GL_TRIANGLE_STRIP](#) и [GL_TRIANGLE_FAN](#).
- 9) Для [Листинга 8](#) для произвольно заданной фигуры составленной из не менее чем 6 треугольников продемонстрировать работу функции `glShadeModel`.
- 10) Для [Листинга 9](#) на наглядном примере продемонстрировать работу проверки глубины для технологии удаления скрытых поверхностей. На наглядном примере продемонстрировать работу технологии отбора задних граней.

ВАРИАНТЫ ЗАДАНИЙ

Для всех вариантов задание одинаковое, внутри подгруппы реализации каждого из пунктов должны различаться.

КОНТРОЛЬНЫЕ ВОПРОСЫ И ЗАДАНИЯ

1. Классифицируйте известные виды геометрических примитивов.
2. Дайте определение тесселяции и раскройте её назначение.
3. Предложите механизм описания криволинейных структур средствами OpenGL.
4. Приведите механизм описания точки.
5. Перечислите основные свойства лицевой грани многоугольника в OpenGL.
6. Раскройте область применения вееров и лент в контексте описания множества полигонов.
7. Классифицируйте многоугольники из соображений их допустимости.
8. Характеризуйте концепцию декартова объёма и способы его задания.
9. Приведите механизм установки размера точки и ширины линии.
10. Предложите способ перехода от определения отдельных точек к многоугольникам.
11. Опишите сущность и назначение шаблонирования линии.
12. Оцените влияние отсечений на производительность.
13. Сформулируйте способ задания цвета многоугольника.
14. Раскройте значение термина узор-заполнитель и опишите способ его задания.
15. Раскройте область применения и роль механизма проверки глубины.

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).

В отчете должны быть представлены:

1. Текст задания для лабораторной работы.
2. В отчете должны быть представлены все листинги программ.

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.

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 с. — 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://learnopengl.com/#!Advanced-OpenGL/Depth-testing> -Тест глубины
2. <https://habrahabr.ru/post/111175/> - Знакомство с OpenGL
3. http://www.opengl-master.ru/view_func.php - Справочник функций OpenGL
4. <https://habrahabr.ru/post/342610/> - Примеры использования теста глубины в OpenGL и граничных эффектов.