

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**

## **ОСНОВЫ НАЛОЖЕНИЯ ТЕКСТУР В OPENGL**

Методические указания к лабораторной работе  
по дисциплине «Компьютерная графика»

Калуга, 2018

УДК 004.62  
ББК 32.972.5  
Б435

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

Методические указания рассмотрены и одобрены:

- Кафедрой «Программного обеспечения ЭВМ, информационных технологий и прикладной математики» (ФН1-КФ) протокол № 7 от «21» февраля 2018 г.

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

- Методической комиссией факультета ФНК протокол № 2 от «18» а 2018 г.

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

- Методической комиссией КФ МГТУ им.Н.Э. Баумана протокол № 2 от «06» 03 2018 г.

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

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

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

#### Аннотация

Методические указания по выполнению лабораторной работы по курсу «Компьютерная графика» содержат общие сведения о текстурных объектах и способе их применения в программном интерфейсе OpenGL. В методических указаниях приводятся теоретические сведения о текстелях, способах загрузки и наложения текстур, а также о фильтрации, как способе оптимизации текстурных данных. Рассмотрен процесс работы с режимами фильтрации, задания параметров отображения и совмещения текстур с эффектами освещения, а также приведен пример переключения между текстурами в 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, С.А.Глебов, 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 .....	32
ЗАДАНИЕ НА ЛАБОРАТОРНУЮ РАБОТУ .....	124
ТРЕБОВАНИЯ К РЕАЛИЗАЦИИ .....	124
ВАРИАНТЫ ЗАДАНИЙ .....	124
КОНТРОЛЬНЫЕ ВОПРОСЫ И ЗАДАНИЯ .....	125
ФОРМА ОТЧЕТА ПО ЛАБОРАТОРНОЙ РАБОТЕ .....	128
ОСНОВНАЯ ЛИТЕРАТУРА .....	129
ДОПОЛНИТЕЛЬНАЯ ЛИТЕРАТУРА.....	130

## 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 представления текстур с помощью программного интерфейса OpenGL, детальные примеры настройки фильтрации пикселей для создания реалистичных перспективных эффектов, наложения текстур на сложные геометрические формы и применения к ним освещения, пояснения по вышеназванным примерам, а также задание на лабораторную работу.

Методические указания составлены в расчете на начальное ознакомление студентов с основами работы с программным интерфейсом OpenGL. Для выполнения лабораторной работы студенту необходимо понимать принципы задания текстур, уметь работать с параметрами фильтрации, создавать и накладывать текстуры на объекты сложной формы, а также адаптировать их в перспективном сокращении и освещении.

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

## **ЦЕЛЬ И ЗАДАЧИ РАБОТЫ, ТРЕБОВАНИЯ К РЕЗУЛЬТАТАМ ЕЕ ВЫПОЛНЕНИЯ**

Целью выполнения лабораторной работы является формирование практических навыков по работе с текстурами средствами OpenGL, их наложению на освещенные объекты подверженные проекционному сокращению.

Основными задачами выполнения лабораторной работы являются: понимать принципы наложения растровых изображений на геометрические объекты, уметь реализовывать наложение текстур с использованием возможностей OpenGL, научиться использовать наложение множественных текстур, уметь создавать фотореалистичные сцены (корректное освещение), на которых присутствуют текстурированные геометрические объекты

Результатами работы являются:

- Загруженные средствами OpenGL текстуры
- Реализованное согласно варианту наложение текстур на объекты сложных сцен
- Применение освещения кобъектам с наложенной текстурой
- Подготовленный отчет

## **ХАРАКТЕРИСТИКА ОБЪЕКТА ИЗУЧЕНИЯ, ИССЛЕДОВАНИЯ**

### **От растра к текстуре**

Ранее подробно была рассмотрена загрузка растровых изображений в OpenGL. Данные изображения, не модифицированные масштабированием пикселей, обычно характеризуются взаимно однозначным соответствием между пикселями изображения и пикселями экрана. Именно так возник сам термин пиксель (от "pictureelement" — элемент изображения). В данной лабораторной работе рассмотрим наложение изображений на трехмерные примитивы. Когда изображения применяются к геометрическому примитиву, они называются текстурой или картой текстуры. Из рис. 1 видно, насколько разными могут быть изображения, отличающиеся текстурной геометрией. Куб слева закрашен и затенен однородной поверхностью, тогда как куб справа демонстрирует богатство деталей, которые можно получить только с помощью наложения текстуры.

Загружаемое текстурное изображение имеет определенную структуру и упорядочение, как и пиксельные образы, но в этом случае редко существует взаимно однозначное соответствие между текселями (отдельные элементы изображения в текстуре, от "textureelement" — элемент текстуры) и пикселями экрана. Рассмотрим основы загрузки карты текстуры в память и все способы, которыми ее можно отобразить и применить к геометрическим примитивам.

### **Загрузка текстур**

Первым необходимым шагом при наложении карты текстуры на геометрический объект является загрузка текстуры в память. Загруженная текстура становится частью текущего состояния текстуры. Для загрузки данных текстуры из буфера памяти (который, например, считывает информацию из файла на диске) чаще всего используются такие функции OpenGL:

```
void glTexImage1D(GLenum target, GLint level, GLint
internal format, GLsizei width, GLint border, GLenum
format, GLenum type, void *data);
void glTexImage2D(GLenum target, GLint level, GLint
internal format, GLsizei width, GLsizei height, GLint
border, GLenum format, GLenum type, void *data);
```

```
void glTexImage3D(GLenum target, GLint level, GLint  
internal format, GLsizei width, GLsizei height, GLsi-  
zei depth, GLint border, GLenum format, GLenum type,  
void *data);
```

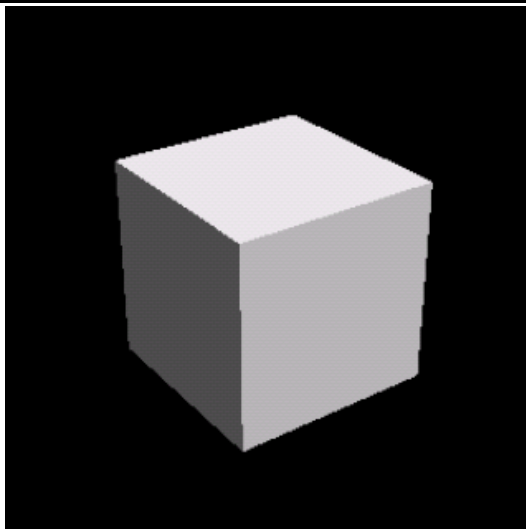


Рис.1. –Сильный контраст между текстурной и нетекстурной геометрией

Эти три сравнительно длинных функции сообщают OpenGL всю информацию, требуемую для интерпретации данных текстуры, на которые указывает параметр `data`.

Первое, что следует знать об этих функциях, — все они, по сути, являются тремя разновидностями одной функции `glTexImage`. OpenGL поддерживает одно-, двух- и трехмерные карты текстуры и использует соответствующие функции для загрузки текстуры и присвоения ей статуса текущей. Также следует знать, что при вызове одной из указанных функций OpenGL копирует информацию о текстуре из `data`. Эти операции копирования данных могут быть довольно ресурсозатратными.

Аргумент `target` данных функций должен иметь значение `GL_TEXTURE_1D`, `GL_TEXTURE_2D` или `GL_TEXTURE_3D` соответственно.

Параметр `level` задает загруженный уровень сокращенной текстуры ([mipmap](#)). Сокращенные текстуры рассмотрены далее. Для несокращенных текстур значение этого параметра всегда устанавливается равным 0.

Таблица 1. Наиболее распространенные внутренние форматы текстуры

Константа	Значение
<code>GL_ALPHA</code>	Записывать тексели как параметры альфа
<code>GL_LUMINANCE</code>	Записывать тексели как коды яркости
<code>GL_LUMINANCE_ALPHA</code>	Записывать тексели с кодом яркости и альфа
<code>GL_RGB</code>	Записывать тексели как красный, зеленый и синий компоненты
<code>GL_RGBA</code>	Записывать тексели как красный, зеленый, синий и альфа-компонент



Далее нужно задать параметр `internalformat` текстурных данных. Это сообщает OpenGL, сколько компонентов цвета на тексель нужно записывать, объем памяти компонентов и/или информацию о том, следует ли сжимать текстуру. В табл. 1 перечислены наиболее распространенные значения этой функции.

Параметры `width`, `height` и `depth` (где они нужны) задают размеры загружаемой текстуры. Важно отметить, что эти размеры должны быть целыми степенями двойки - 1, 2, 4, 8, 16, 32, 64 и т.д. Не существует жесткого требования, чтобы карты текстуры были квадратными (все размеры равны), но текстуры, загруженные с размерами, не являющимися степенями двойки, приведут к неявной деактивизации текстуры. Параметр `border` позволяет задавать ширину границы для карт текстуры. Границы текстуры разрешают расширять ширину, высоту или глубину карты текстуры на дополнительный набор текселей, располагающихся вдоль границы. Границы текстуры играют важную роль в текстурной фильтрации, которая обсуждается ниже. Пока же просто устанавливайте это значение равным 0.

Последние три параметра (`format`, `type` и `data`) идентичны соответствующим аргументам функции `glDrawPixels`, используемой при помещении данных изображения в буфер цвета. Приемлемые значения констант `format` и `type` перечислены в табл. 2.

Загруженные текстуры не применяются к геометрическим объектам, если не активизировано соответствующее состояние текстуры. Чтобы включить или выключить данное состояние текстуры, вызывается функция `glEnable` или `glDisable` с параметром `GL_TEXTURE_1D`, `GL_TEXTURE_2D` или `GL_TEXTURE_3D`. В каждый момент времени может быть активизировано только одно из этих состояний. Тем не менее следует отключать неиспользуемые состояния текстуры. Например, чтобы переключаться между одно- и двухмерным текстурированием, следует вызывать следующие функции:

```
glDisable(GL_TEXTURE_1D);  
glEnable(GL_TEXTURE_2D);
```

Таблица 2. Типы тексельных данных

Константа	Описание
GL_UNSIGNED_BYTE	Все компоненты цвета является 8-битовыми целыми числами без знака
GL_BYTE	8-битовые числа со знаком
GL_BITMAP	Отдельные биты без данных о цвете; то же, что glBitmap
GL_UNSIGNED_SHORT	16-битовые целые числа без знака
GL_SHORT	16-битовые целые числа со знаком
GL_UNSIGNED_INT	32-битовые целые числа без знака
GL_INT	32-битовые целые числа со знаком
GL_FLOAT	Величины с плавающей запятой обычной точности
GL_UNSIGNED_BYTE_3_2_2	Упакованные RGB-коды
GL_UNSIGNED_BYTE_2_3_3_REV	Упакованные RGB-коды
GL_UNSIGNED_SHORT_4_4_4_4	Упакованные RGBA-коды
GL_UNSIGNED_SHORT_5_5_5_1	Упакованные RGBA-коды
GL_UNSIGNED_INT_8_8_8	Упакованные RGBA-коды
GL_UNSIGNED_INT_8_8_8_REV	Упакованные RGBA-коды

И еще одно замечание относительно загрузки текстур: данные текстуры, загруженные функцией `glTexImage`, проходят через тот же конвейер преобразований пикселей и воспроизведения изображений. Это означает, что к загруженным данным текстуры применяются упаковка пикселей, масштабирование пикселей, таблицы цветов, свертки и т.д.

## Использование буфера цвета

Одно- и двухмерные текстуры также можно загрузить, используя данные из буфера цвета. С помощью указанных ниже двух функций изображение можно считать из буфера цвета и использовать его как новую текстуру.

```
void glCopyTexImage1D(GLenum target, GLint level,
GLenum internal format, GLint x, GLint y, GLsizei
width, GLint border);
```

```
void glCopyTexImage2D(GLenum target, GLint level,
GLenum internal format, GLint x, GLint y, GLsizei
width, GLsizei height, GLint border);
```

Данные функции похожи на **`glTexImage`**, но здесь `x` и `y` задают положение в буфере цвета, с которого начинается чтение данных текстуры. Исходный буфер задается с использованием **`glReadBuffer`** и ведет себя так же, как **`glReadPixels`**.

## Обновление текстуры

Многократная загрузка новых текстур может стать критическим параметром производительности в таких приложениях реального времени, как игры или имитаторы. Если загруженная карта текстуры уже не требуется, ее можно заменить целиком или частично. Отметим, что замещение карты текстуры часто можно выполнить гораздо быстрее, чем загрузку новой текстуры с помощью `glTexImage`. В данной ситуации используется функция `glTexSubImage`, которая также имеет три варианта.

```
void glTexSubImage1D(GLenum target, GLint level,
GLint xOffset, GLsizei width, GLenum format, GLenum
type, const GLvoid *data);
```

```
void glTexSubImage2D(GLenum target, GLint level,
GLint xOffset,
GLint yOffset, GLsizei width, GLsizei height, GLenum
format, GLenum type, const GLvoid *data);
void glTexSubImage3D(GLenum target, GLint level,
GLint xOffset,
GLint yOffset, GLint zOffset, GLsizei width, GLsizei
height, GLsizei depth, GLenum format, GLenum type,
const GLvoid *data);
```

Обычно аргументы точно соответствуют параметрам, используемым в функции `glTexImage`. Параметры `xOffset`, `yOffset` и `zOffset` задают смещения в существующей карте текстуры, с которого начинается замещение данных текстуры. Значения `width`, `height` и `depth` задают размеры текстуры, которая "вставляется" в существующую текстуру.

Последний набор функций (разновидности `glCopyTexSubImage`) позволяет комбинировать чтение из буфера цвета с вводом или замещением части текстуры.

```
void glCopyTexSubImage1D(GLenum target, GLint level,
GLint xoffset, GLint x, GLint y, GLsizei width);
void glCopyTexSubImage2D(GLenum target, GLint level,
GLint xoffset, GLint yoffset, GLint x, GLint y, GLsi-
zei width, GLsizei height);
void glCopyTexSubImage3D(GLenum target, GLint lev-
el, GLint xoffset, GLint yoffset, GLint zoffset,
GLint x, GLint y, GLsizei width, GLsizei height)
```

Возможно, вы отметили, что в приведенном списке нет функции `glCopyTexImage3D`. Это объясняется тем, что буфер цвета является двухмерным, поэтому не существует соответствующего способа использовать двухмерное цветное изображение как источник трехмерной текстуры. Тем не менее с помощью `glCopyTexSubImage3D` данные буфера можно использовать для задания плоскости текселей в трехмерной текстуре.

## Отображение текстур на геометрические объекты

Загрузка текстуры и активизация текстурирования указывают OpenGL применять текстуру ко всем примитивам OpenGL. При этом

следует предоставить OpenGL информацию о том, как наложить текстуру на геометрический объект. Для этого задаются *текстурные координаты* всех вершин. Обращение к текстелям карты текстуры выполняется не как к ячейкам памяти (как было для пиксельных образов), а как к абстрактным текстурным координатам (обычно — значения с плавающей запятой). Как правило, текстурные координаты задаются в виде значений с плавающей запятой, принадлежащих диапазону 0-1. Текстурные координаты обозначаются  $s$ ,  $t$ ,  $r$  и  $q$  (подобно координатам вершины  $x$ ,  $y$ ,  $z$  и  $w$ ), поддерживаются одно-, двух- и трехмерные координаты, также предусмотрена возможность масштабирования координат.

На рис. 2 показаны одно-, двух- и трехмерные текстуры, а также их упорядочение относительно текселей.

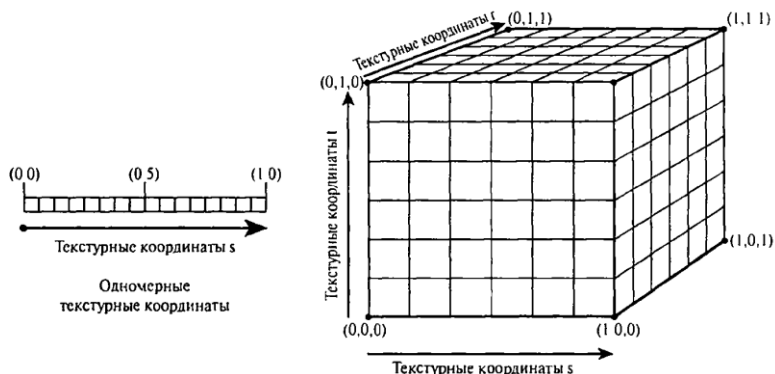


Рис.2. – Адресация текселей с помощью текстурных координат

Поскольку четырехмерных текстур не существует, то возникает вопрос, для чего нужна координата  $q$ ? Координата  $q$  соответствует геометрической координате  $w$ . Это масштабный множитель, применяющийся к остальным текстурным координатам; т.е. реальные значения, используемые в качестве текстурных координат, равны  $s/q$ ,  $t/q$  и  $r/q$ . По умолчанию  $q$  равно 1.0.

Текстурные координаты задаются с помощью функции `glTexCoord`. Весьма похожая на функции координат вершин, нормалей поверхностей и кодов цвета, `glTexCoord` имеет множество разновидностей. Ниже приведены три простейших варианта, используемых в демонстрационных программах.

```
void glTexCoordf(GLfloat s);
```

```
void glTexCoord2f(GLfloat s, GLfloat t);  
void glTexCoord3f(GLfloat s, GLfloat t, GLfloat r);
```

При использовании данных функций со всеми вершинам соотносится одна текстурная координата. Затем OpenGL при необходимости растягивает или сжимает текстуру, чтобы наложить ее на геометрический объект. На рис. 3 приведен пример двухмерной текстуры, отображаемой на примитив GL\_QUAD. Обратите внимание на то, что углы текстуры соответствуют углам квадрата. Важно знать, что точно так же, как другие свойства вершины (материалы, нормали и т.д.), текстурные координаты необходимо задать до определения самой вершины.

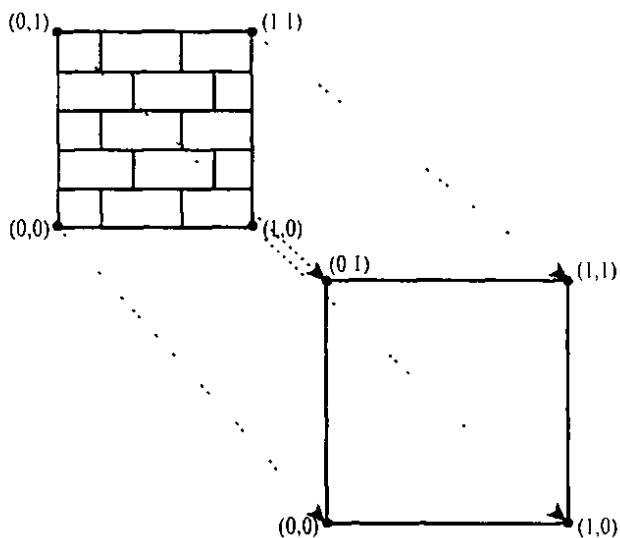


Рис.3. – Наложение двухмерной текстуры на квадрат

К сожалению, вам редко встретится такое прекрасное соответствие, чтобы квадратная текстура накладывалась на квадратный объект. Чтобы помочь вам лучше понять текстурные координаты, на рис. 4 предоставлен другой пример. На данном рисунке также изображена квадратная карта текстуры, однако геометрический объект на этот раз является треугольником. На карту текстуры наложены текстурные

координаты, соответствующие точкам текстуры, переходящим в вершины треугольника.

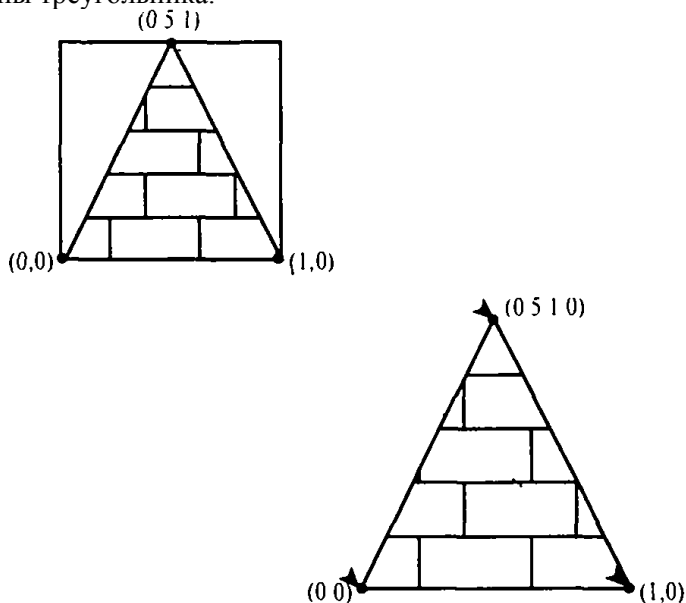


Рис.4. – Применение фрагмента карты текстуры к треугольнику

## Матрица текстуры

Текстурные координаты также можно преобразовывать с помощью матрицы текстуры. Стек матриц текстуры действует точно так же, как стек любых других матриц, обсуждавшихся ранее (наблюдения модели, проекции и цвета). Чтобы сделать матрицу текстуры целью вызова матричной функции, вызывается функция `glMatrixMode` с аргументом `GL_TEXTURE`.

```
glMatrixMode (GL_TEXTURE);
```

В спецификациях OpenGL требуется, чтобы стек матрицы текстуры имел глубину всего два элемента (для использования `glPushMatrix` и `glPopMatrix`). Текстурные координаты можно транслировать, масштабировать и даже поворачивать.

## Текстурная среда

В программе пирамида рисуется с белыми гранями (свойство материала), а текстура накладывается так, что ее цвета масштабируются согласно окраске освещенной геометрической фигуры. То, как OpenGL объединяет цвета текселей с цветом геометрического объекта, на который накладывается текстура зависит от режима текстурной среды. Данный режим устанавливается с помощью функции `glTexEnv`

```
void glTexEnv(GLenum target, GLenum pname, GLint param);
```

```
void glTexEnvf(GLenum target, GLenum pname, GLfloat param);
```

```
void glTexEnviv(GLenum target, GLenum pname, GLint *param);
```

```
void glTexEnvfv(GLenum target, GLenum pname, GLfloat *param);
```

Функция имеет множество разновидностей и с ее помощью можно управлять более сложными деталями текстурирования. В [листинге 1 эта](#) функция задает режим среды `GL_MODULATE` до применения какой-либо текстуры.

```
glTexEnv(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE);
```

В режиме `MODULATE` цвет текселя умножается на цвет геометрического объекта (после расчета освещения). Именно поэтому проявляется затененность пирамиды, из-за чего текстура кажется затененной. Используя данный режим, можно менять цветовой тон текстуры, накладывая цветные геометрические объекты. Например, черно-белая текстура кирпичной кладки, наложенная на красные, желтые и коричневые геометрические объекты, даст красные, желтые и коричневые кирпичи, хотя при этом будет использована единственная текстура.

Если требуется просто заменить цвет геометрии, на которую накладывается текстура, в качестве режима среды задается `GL_REPLACE`. Таким образом, цвета фрагментов геометрического объекта непосредственно замещаются цветами текселей. Поступив так, вы сведете на нет все влияние объекта на текстуру. Если текстура имеет альфа-канал, можно активизировать смешение; иногда с помощью данного режима создаются прозрачные геометрические объекты, согласно альфа-каналу украшенные узором карты текстуры.



Если текстура не имеет компонента альфа, можно активизировать режим `GL_DECAL`, который ведет себя точно так же, как `GL_REPLACE`. В этом режиме текстура просто "переводится" (decals) поверх геометрии и кодов цвета, рассчитанных для фрагментов. Однако, если текстура имеет компонент альфа, переводной рисунок можно применить так, чтобы там, где коды альфа смешиваются с фрагментами объекта, через него просматривался объект.

Текстуры также можно смешивать с постоянным цветом, используя текстурную среду `GL_BLEND`. Устанавливая данный текстурный режим, также следует указывать цвет текстурной среды.

```
GLfloat fColor[4] = { 1.0f, 0.0f, 0.0f, 0.0f };
glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE,
GL_BLEND); glTexEnvfv(GL_TEXTURE_ENV,
GL_TEXTURE_ENV_COLOR, fColor);
```

Наконец, коды цвета текселей можно просто добавить к фрагменту, на который накладываются текстуры, установив режим среды равным `GL_ADD`. Любым кодам цвета, превышающим 1.0, присваивается значение 1.0, и в результате могут образовываться насыщенные коды цвета (по сути, белый или более близкие к белому цвета, чем вы могли ожидать).

## Параметры текстуры

Наложение текстуры — это не просто перенос изображения на треугольную грань. На правила визуализации и поведение применяющихся карт текстуры влияет множество параметров. Все они задаются с помощью различных вариантов функции `glTexParameter`:

```
void glTexParameterf(GLenum target, GLenum pname,
GLfloat param);
void glTexParameteri(GLenum target, GLenum pname,
GLint param);
void glTexParameterfv(GLenum target, GLenum pname,
GLfloat *params);
void glTexParameteriv(GLenum target, GLenum pname,
GLint *params);
```

Первый аргумент `target` задает, к какому текстурному режиму должен применяться параметр, и его значениями могут быть

GL\_TEXTURE\_1D, GL\_TEXTURE\_2D или GL\_TEXTURE\_3D. Второй аргумент pname задает, какой параметр текстуры устанавливается, и наконец, аргументы param или params устанавливают значение конкретного параметра текстуры.

## Основная фильтрация

В отличие от растровых образов, рисуемых в буфере цвета при применении текстуры к геометрическому объекту, между текселями карты текстуры и пикселями экрана практически никогда не существует взаимно-однозначного соответствия (Рис.5.). Конечно, аккуратный программист может добиться такого результата, но только с помощью текстурной геометрии, тщательно спланированной так, чтобы на экране создавать иллюзии того, что тексели и пиксели выровнены. В реальной же жизни текстурные изображения всегда должны растягиваться или сжиматься, если их применяют к геометрическим поверхностям. Из-за ориентации геометрических объектов данная текстура может даже одновременно растягиваться и сжиматься на поверхности объекта.

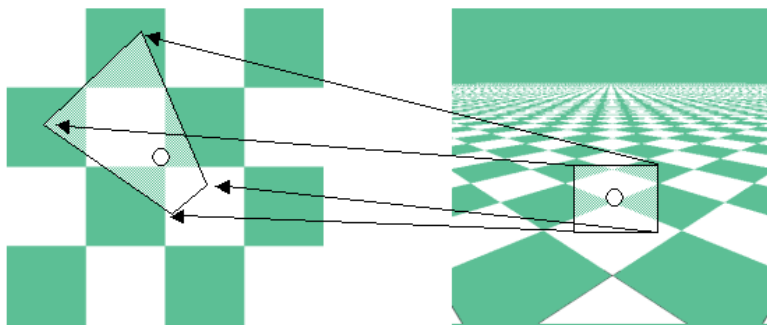


Рис.5. – Область экрана и ее образ в текстур

Процесс расчета цветных фрагментов растянутой или сжатой карты текстуры называется *фильтрацией текстуры*. С помощью функции параметров текстуры OpenGL позволяет устанавливать фильтры *увеличения* и *уменьшения*. Данные фильтры соотнесены с именами параметров GL\_TEXTURE\_MAG\_FILTER и GL\_TEXTURE\_MIN\_FILTER. Меняя значения функции фильтрации,

можно выбрать два базовых текстурных фильтра `GL_NEAREST` и `GL_LINEAR`, которые соответствуют фильтрации по ближайшему соседу и линейной фильтрации

Фильтрация по ближайшему соседу является простейшим (и самым быстрым) методом фильтрации, который можно выбрать в OpenGL. Вначале вычисляются текстурные координаты, а затем строится их зависимость от текселей текстуры — на какой тексель припадет координата, такой цвет и будет использован в качестве цвета фрагмента.

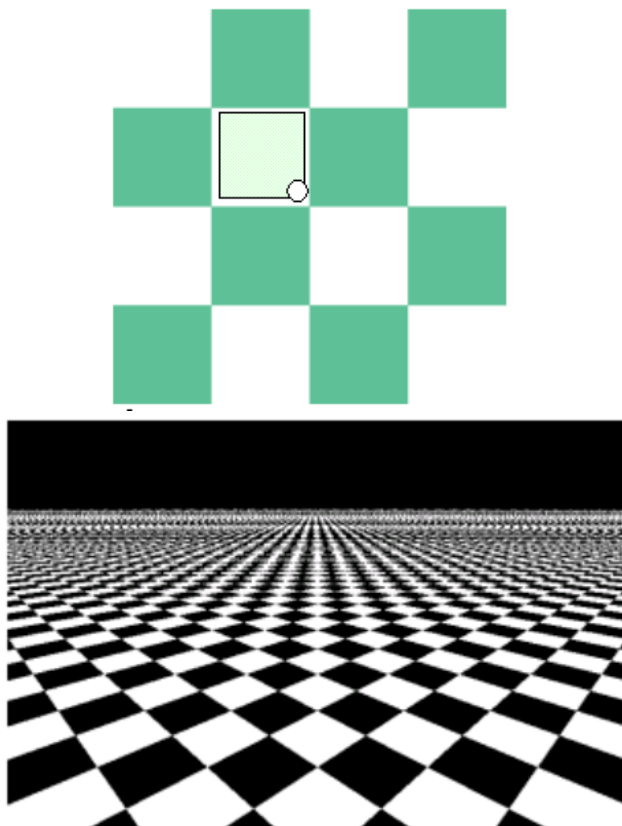


Рис.6. – Изображение фильтрации по ближайшему соседу

Фильтрация по ближайшему "соседу" характеризуется "большими пикселями", поскольку в этом режиме текстура чрезмерно растягивается. Соответствующий пример демонстрируется на рис. 6.

Для выбора в качестве текстурного фильтра двухмерного (GL\_TEXTURE\_2D) фильтра уменьшения или увеличения применяются такие две функции:

```
glTexParameteri(GL_TEXTURE_2D,  
GL_TEXTURE_MAG_FILTER, GL_NEAREST); glTexParameteri(  
GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,  
GL_NEAREST);
```

Линейная фильтрация требует больше работы, чем фильтрация по ближайшему "соседу", но часто эти издержки себя оправдывают. На современном потребительском аппаратном обеспечении дополнительная цена линейной фильтрации пренебрежимо мала. При линейной фильтрации учитывается не тексель, ближайший к текстурной координате, а взвешенное среднее текселей, окружающих текстурную координату, т.е. выполняется линейная интерполяция. Чтобы интерполированный фрагмент точно соответствовал цвету текселя, текстурная координата должна попасть в центр текселя. Линейная фильтрация характеризуется "пористой" (нечеткой) графикой при растягивании текстуры. Однако размытость часто дает более реалистичный и менее искусственный внешний вид, чем зазубренные блоки режима фильтрации по ближайшему "соседу". На рис. 7 демонстрируется пример-противопоставление иллюстрации на рис. 6. Задать линейную фильтрацию (в режиме GL\_TEXTURE\_2D) довольно просто, и для этого используются приведенные ниже команды (также включены в функцию SetupRC [листинга 1.](#)).

```
glTexParameteri(GL_TEXTURE_2D,  
GL_TEXTURE_MAG_FILTER, GL_LINEAR); glTexParameteri(  
GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
```

## Намотка текстуры

Обычно текстурные координаты для отображения текселей карты текстуры принадлежат диапазону от 0.0 до 1.0. Если текстурные координаты выходят из этого диапазона, OpenGL обрабатывает их согласно текущему режиму намотки текстуры. Режим намотки можно установить для каждой координаты отдельно, вызвав функцию `glTexParameteri` с параметром `GL_TEXTURE_WRAP_S`, `GL_TEXTURE_WRAP_T` или `GL_TEXTURE_WRAP_R`. Собственно режим намотки можно установить равным одному из следующих

значений: `GL_REPEAT`, `GL_CLAMP`, `GL_CLAMP_TO_EDGE` или `GL_CLAMP_TO_BORDER`.

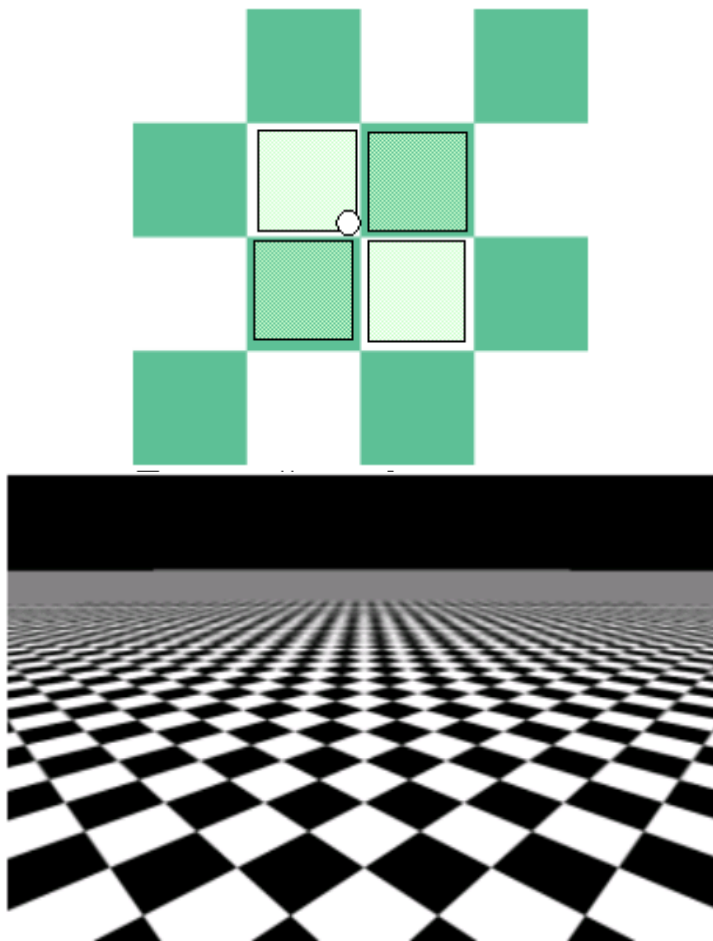


Рис.7. – Изображение линейной фильтрации

Режим `GL_REPEAT` просто вынуждает текстуру повторяться в том направлении, где текстурная координата превысила 1.0, причем текстура повторяется для всех целых текстурных координат. Этот режим очень полезен при наложении небольшой мозаичной плитки на большие геометрические поверхности. Аккуратно разработанные бесшовные текстуры позволяют создать иллюзию гораздо большей

текстуры за счет гораздо меньшего текстурного изображения. Другие режимы не повторяют, а "вжимают" текстуру в требуемый диапазон (отсюда и слово "clamp" в их названиях).

Если бы единственным следствием выбора режима намотки было повторение или не повторение текстуры, требовалось бы всего два режима: повтор и ограничение согласно разрешенному диапазону. Однако кроме этого режим намотки текстуры сильно влияет на выполнение текстурной фильтрации на краях карты текстуры. При фильтрации с параметром `GL_NEAREST` режим намотки не имеет значения, поскольку текстурные координаты всегда относятся к конкретному текстелю карты текстуры. В то же время фильтр `GL_LINEAR` принимает среднее пикселей, окружающих рассчитываемую координату, но при этом возникают проблемы с текселями, лежащими на краях карты текстуры.

Данная проблема решается достаточно аккуратно в режиме `GL_REPEAT`. Выборки текселей просто берутся из следующей строки или столбца, которые в режиме повторения повторяют первые. Данный режим идеально подходит для текстур, оборачивающих объект и встречающихся на другой стороне (рассмотрите, например, наложение текстуры на сферу).

Режимы намотки текстуры с ограничением согласно допустимому диапазону предлагают несколько вариантов обработки краев текстуры. В режиме `GL_CLAMP` требуемые тексели извлекают из границ текстуры или `TEXTURE_BORDER_COLOR` (устанавливается с помощью `glTexParameterfv`). Режим намотки `GL_CLAMP_TO_EDGE` игнорирует выборки текселей, выходящие за край, и не включает их в среднее. Наконец, `GL_CLAMP_TO_BORDER` использует тексели границы везде, где текстурные координаты выходят из диапазона 0.0-1.0.

Типичным приложением режимов ограничения согласно допустимому диапазону является наложение текстуры на большую область, которая требует текстуры, слишком большой, чтобы поместиться в память. В этом случае область разбивается на маленькие "плитки", которые затем располагаются рядом. Если в подобном случае не использовать такой режим, как `GL_CLAMP_TO_EDGE`, на стыках между плитками получим визуальные артефакты фильтрации

Например, в листинге 1. текстурные координаты вдоль основания пирамиды могут породить темный стык, если не указать, что режим намотки ограничивает билинейную фильтрацию пределами карты текстуры

```
glTexParameterf(GL_TEXTURE_2D,  
GL_TEXTURE_WRAP_S,gl_CLAMP_TO_EDGE);  
glTexParameterf(GL_TEXTURE_2D,  
GL_TEXTURE_WRAP_T,gl_CLAMP_TO_EDGE);
```

## Сокращённые текстуры

Применение *сокращённой* (или *множественной*) *текстуры* (mip-mapping) является мощной техникой наложения текстуры, позволяющей повысить и производительность визуализации, и визуальное качество сцены. Эта техника решает две распространённых проблемы стандартного метода наложения текстуры. Первой является эффект *сцинтилляции* (наложения артефактов), проявляющийся на поверхности объектов, визуализированных на очень маленькой площади экрана по сравнению с относительным размером наложенной текстуры. Сцинтилляцию можно рассматривать как разновидность мерцания, которое проявляется, когда область дискретизации карты текстуры движется непропорционально её размерам на экране. Отрицательные эффекты сцинтилляции наиболее заметны при движении камеры или объектов.

Вторая проблема больше связана с производительностью, но вызвана теми же причинами, что и сцинтилляция. Чтобы отобразить небольшое число фрагментов на экране, нужно загрузить в память и обработать (отфильтровать) большой фрагмент текстуры. Из-за этого при увеличении размера текстуры существенно снижается производительность.

Решение обеих проблем заключается в простом использовании меньших карт текстуры. Однако это решение создаёт новую проблему: при приближении камеры к тому же объекту его нужно будет визуализировать с большими размерами, и небольшую карту текстуры придётся растягивать до состояния объекта с безнадёжно расплывшейся или блочной текстурой.

Выходом из создавшейся ситуации является множественное отображение (mipmapping). Данная техника получила название от латинского выражения "multum in parvo" (MIP), означающего "многое в малом". По сути, вы загружаете не одно изображение в одно состояние текстуры, а целый ряд изображений (от наибольшего до наименьшего) в единое "всеобъемлющее" (mipmapped) состояние текстуры. Затем OpenGL с помощью нового набора режимов фильтрации выбирает текстуру или текстуры, наилучшим образом подходящие для данной геометрии. За счет немного увеличившихся требований к памяти (и, возможно, существенно увеличившихся требований к обработке) можно одновременно избавиться от сцинтилляции и издержек, связанных с обработкой далеких объектов, при этом поддерживая версии текстуры с большими разрешениями, которые при необходимости можно будет использовать.

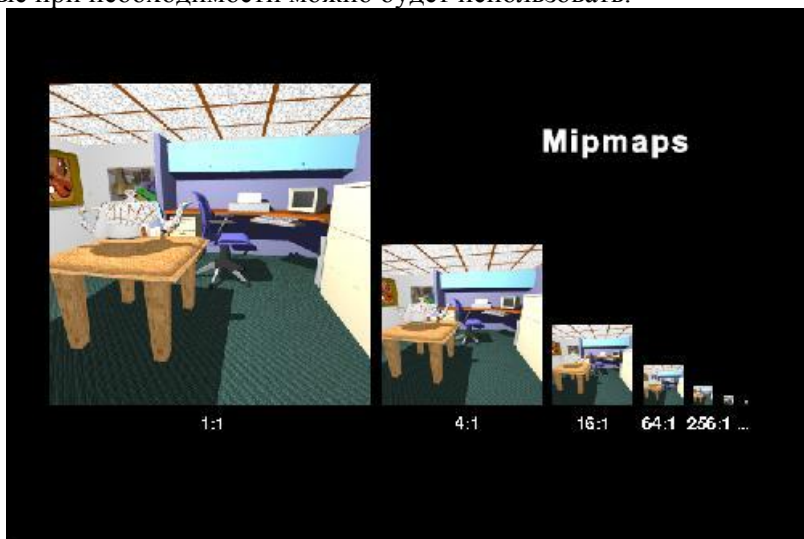


Рис.8. – Ряд изображений с последовательно сокращающейся текстурой

"Всеобъемлющая" текстура состоит из ряда текстурных изображений, каждое последующее из которых вдвое меньше предыдущего. Данный сценарий иллюстрируется на рис. 8. Уровни множественных отображений не обязательно должны быть квадратными, но сокращение размеров вдвое продолжается до тех пор, пока последнее изображение не будет текселем 1x1. После того



как в ходе сокращения один из размеров становится равным 1, далее сокращается только второй размер. Кстати, использование квадратного набора "всеобъемлющих" текстур требует примерно на треть больше памяти, чем применение обычной текстуры.

Уровни множественной текстуры загружаются с помощью `glTexImage`. Теперь в этой команде важен параметр `level`, поскольку он задает, какому уровню текстуры соответствуют предоставленные данные изображения. Первым идет уровень 0, затем следуют уровни 1, 2 и т.д. Если технология множественной текстуры не используется, загружается только уровень 0. Чтобы использовать множественную текстуру по умолчанию, следует заселить все ее уровни. Тем не менее можно задать только базовый и максимальный уровни, указав параметры текстуры `GL_TEXTURE_BASE_LEVEL` и `GL_TEXTURE_MAX_LEVEL`. Например, если нужно загружать только уровни с 0 по 4, функция `glTexParameteri` вызывается дважды.

```
glTexParameteri(GL_TEXTURE_2D,  
GL_TEXTURE_BASE_LEVEL, 0);  
glTexParameteri(GL_TEXTURE_2D,  
GL_TEXTURE_MAX_LEVEL, 4);
```

Хотя `GL_TEXTURE_BASE_LEVEL` и `GL_TEXTURE_MAX_LEVEL` контролируют, какие уровни текстуры загружаются, вы можете явно ограничить диапазон загружаемых уровней, применив вместо данных параметров `GL_TEXTURE_MIN_LOD` и `GL_TEXTURE_MAX_LOD`.

## Фильтрация множественной текстуры

Множественная текстура добавляет новый прием к двум базовым режимам текстурной фильтрации `GL_NEAREST` и `GL_LINEAR`, предоставляя четыре перестановки варианта фильтрации множественной текстуры, перечисленных в таблице 3.

Простая загрузка уровней текстуры с помощью функции `glTexImage` сама по себе не активизирует множественную текстуру. Если фильтр текстуры установлен равным `GL_LINEAR` или `GL_NEAREST`, используется только основной уровень текстуры, а все уровни множественной текстуры игнорируются. Чтобы эти загруженные уровни использовались, нужно задать один из фильтров множественной текстуры. Соответствующие константы имеют вид `GL_FILTER_MIPMAP_SELECTOR`, где `FILTER` задает фильтр

текстуры, который будет использоваться на выбранном уровне множественной текстуры SELECTOR указывает, как выбирается уровень множественной текстуры, например, GL\_NEAREST соответствует выбору ближайшего подходящего уровня.

Таблица 3. Фильтры множественной текстуры

GL_NEAREST	Фильтрация по ближайшему "соседу" на основном уровне текстуры
GL_LINEAR	Линейная фильтрация на основном уровне текстуры
GL_NEAREST_MIPMAP_NEAREST	Выбор ближайшего уровня текстуры и выполнение фильтрации по ближайшему «соседу»
GL_NEAREST_MIPMAP_LINEAR	Выполнение линейной интерполяции между уровнями текстуры и выполнение фильтрации по ближайшему "соседу"
GL_LINEAR_MIPMAP_NEAREST	Выбор ближайшего уровня текстуры и выполнение линейной фильтрации
GL_LINEAR_MIPMAP_LINEAR	Выполнение линейной интерполяции между уровнями текстуры и выполнение линейной фильтрации, также называется <i>трилинейной фильтрацией</i> или <i>трилинейным множественным отображением</i>

Использование в качестве SELECTOR константы GL\_LINEAR даст линейную интерполяцию между двумя ближайшими уровнями множественной текстуры, результат которой будет отфильтрован с применением выбранного фильтра текстуры. Указание одного из режимов фильтрации множественной текстуры без загрузки уровней множественной текстуры равнозначно деактивизации наложения текстуры.

Выбор фильтра меняется в зависимости от приложения и существующих требований к производительности GL\_NEAREST\_MIPMAP\_NEAREST, например, даст очень высокую производительность и небольшие артефакты наложения (сцинтилляции), но фильтрация по ближайшим соседям часто неприятна зрительно. В играх для ускорения обработки часто применяется GL\_LINEAR\_MIPMAP\_NEAREST, поскольку это соответствует высококачественной линейной фильтрации, но выбор между доступными уровнями текстуры разных размеров делается быстро.

Применяя для выбора уровня принцип ближайших "соседей", также можно получить нежелательные визуальные артефакты. При наблюдении поверхности под углом на ней часто заметно место перехода от одного уровня текстуры к другому. Данный переход выглядит как искажение или резкий переход от одного уровня детализации к другому. Фильтры GL\_LINEAR\_MIPMAP\_LINEAR и GL\_NEAREST\_MIPMAP\_LINEAR выполняют дополнительную интерполяцию между уровнями множественной текстуры, что позволяет устранить зону перехода, хотя и за счет существенно увеличивающейся обработки. Фильтр GL\_LINEAR\_MIPMAP\_LINEAR называется *трилинейной фильтрацией*, и до недавнего времени он был стандартом фильтрации текстуры, дающим наивысшую точность. Совсем недавно на аппаратном обеспечении OpenGL стала доступна анизотропная фильтрация, дающая лучшее качество, но за счет еще больших издержек.

## Генерация уровней множественной текстуры

Как отмечалось ранее, множественная текстура требует приблизительно на треть больше памяти, чем просто загрузка основного текстурного изображения. Кроме того, нужно, чтобы для загрузки были доступны все меньшие и меньшие версии основного

текстурного изображения. Иногда это бывает неудобно, поскольку изображения с меньшим разрешением не обязательно будут доступны или программисту, или конечному пользователю программного обеспечения. В связи с этим в библиотеку GLU включена функция `gluScaleImage`, которую можно применить для последовательного масштабирования и загрузки изображения, пока не будут загружены все необходимые уровни множественной текстуры. Часто доступна еще более удобная функция, автоматически создающая масштабированные изображения и загружающая их с помощью `glTexImage`. Эта функция - `gluBuildMipmaps` - имеет три варианта и поддерживает одно-, двух- и трехмерные карты текстуры.

```
int gluBuild1DMipmaps(GLenum target, GLint internalFormat, GLint width, GLenum format, GLenum type, const void *data);
```

```
int gluBuild2DMipmaps(GLenum target, GLint internalFormat, GLint width, GLint height, GLenum format, GLenum type, const void *data);
int gluBuild3DMipmaps(GLenum target, GLint internalFormat, GLint width, GLint height, GLint depth, GLenum format, GLenum type, const void *data);
```

С помощью новых версий библиотеки GLU можно на более низком уровне получить контроль над тем, какие уровни множественной текстуры загружаются. Для этого были введены следующие функции.

```
int gluBuild1DMipmapLevels(GLenum target, GLint internalFormat, GLint width, GLenum format, GLenum type, GLint level, GLint base, GLint max, const void *data);
int gluBuild2DMipmapLevels(GLenum target, GLint internalFormat, GLint width, GLint height, GLenum format, GLenum type, GLint wlevel, GLint base, GLint max, const void *data);
int gluBuild3DMipmapLevels(GLenum target, GLint internalFormat, GLint width, GLint height, GLint depth, GLenum format, GLenum type,
```

```
Glint level, GLint base, GLint max, const void  
*data);
```

В этих функциях `level` обозначает уровень множественной текстуры, задаваемый параметром `data`. С помощью предоставленных данных строятся уровни множественной текстуры от `base` до `max`.

## **Аппаратная генерация множественных отображений**

Если вы заранее знаете, что требуется загрузка всех уровней множественной текстуры, можете использовать аппаратное ускорение OpenGL для быстрой генерации всех необходимых уровней множественного отображения. Для этого параметр текстуры `GL_GENERATE_MIPMAP` устанавливается равным `GL_TRUE`:

```
glTexParameteri (GL_TEXTURE_2D, GL_GENERATE_MIPMAP,  
GL_TRUE);
```

После установки данного параметра все вызовы функций `glTexImage` или `glTexSubImage`, обновляющие основную карту текстуры (уровень 0 множественной текстуры), автоматически обновляют все нижние уровни множественной текстуры. Если задействовать графическое аппаратное обеспечение, данная возможность будет реализована гораздо быстрее, чем при использовании `gluBuildMipmaps`. Тем не менее, следует знать, что изначально данная возможность относилась к расширениям и в основной программный интерфейс OpenGL была включена только в версии 1.4.

## **Смещение уровня детализации**

Чтобы определить, какой уровень множественного отображения выбрать, OpenGL использует формулу, в которую входят размеры уровней множественного отображения и области экрана, занимаемой геометрическим объектом. OpenGL старается добиться близкого соответствия между выбранным уровнем множественного отображения и представлением текстуры на экране. Можно указать OpenGL сместить критерий выбора назад (к большим уровням) или вперед (к меньшим уровням множественного отображения). Это

может привести к повышению производительности (использование меньших уровней) или увеличению резкости объектов с наложенной текстурой (использование больших уровней множественного отображения). Как показано ниже, смещение задается с помощью параметра текстурной среды `GL_TEXTURE_LOD_BIAS`.

```
glTexEnvf (GL_TEXTURE_FILTER_CONTROL,  
GL_TEXTURE_LOD_BIAS, -1.5);
```

В приведенном примере степень детализации текстуры немного смещена к большим уровням детализации (меньшего параметра уровня), что дает более резкие текстуры за счет усложнения обработки.

## Текстурные объекты

До этого момента показывалось, как загружать параметры текстуры, влияющие на наложение карт текстуры на геометрические объекты. Рисунок и параметры текстуры задаются с помощью функции `glTexParameter`, содержащей *состояние текстуры*. Загрузка и поддержание состояния текстуры находится на одном из первых мест во многих приложениях OpenGL с интенсивным использованием текстуры (в частности, игр).

Особенно трудоемкими являются вызовы таких функций, как `glTexImage`, `glTexSubImage` и `gluBuildMipmaps`. Эти функции перемещают большие участки памяти и, возможно, требуют изменения формата данных, согласовывающего эти данные с внутренним представлением. Переключение между текстурами или загрузка другого текстурного изображения обычно может оказаться довольно дорогой операцией.

Текстурные объекты позволяют загружать более одного состояния текстуры за раз (включая рисунок текстуры) и быстро переключаться между ними. Состояние текстуры поддерживается связанным в настоящее время текстурным объектом, который идентифицируется целым числом без знака. Для выделения нескольких текстурных объектов применяется следующая функция:

```
void    glGenTextures (GLsizei    n,    GLuint  
*textures);
```

Вызывая эту функцию, вы задаете число текстурных объектов и указатель на массив целых чисел без знака, который будет заселен идентификаторами текстурных объектов. Идентификаторы вы можете

представить себе как обработчики доступных состояний текстуры. Чтобы "привязаться" к одному из этих состояний, вызывается такая функция:

```
void glBindTexture (GLenum target, GLuint texture);
```

Параметр `target` должен иметь значение `GL_TEXTURE_1D`, `GL_TEXTURE_2D` либо `GL_TEXTURE_3D`, а `texture` — это конкретный текстурный объект, с которым связываемся. С этого момента вся загрузка текстур и настройки параметров текстуры влияют только на текущий связанный текстурный объект. Для удаления текстурных объектов применяется следующая функция:

```
void glDeleteTextures (GLsizei n, GLuint *textures);
```

Аргументы данной функции имеют то же значение, что и в функции `glGenTextures`. Отметим, что не нужно генерировать и удалять все текстурные объекты одновременно. Многократный вызов функции `glGenTextures` влечет за собой очень малые издержки. Многократный вызов `glDeleteTextures` может ввести небольшие задержки, но это связано только с тем, что вы освобождаете большие объемы памяти текстуры

Чтобы проверить, насколько справедливы имена объектов (или обработчиков), те узнать, соответствуют ли они текстурным объектам, используется следующая функция:

```
GLboolean glIsTexture (GLuint texture);
```

Эта функция возвращает `GL_TRUE`, если целое число представляет распределенное ранее имя текстурного объекта, и `GL_FALSE` в противном случае.

## ЗАДАЧИ И ПОРЯДОК ВЫПОЛНЕНИЯ РАБОТЫ

### Пример наложения двумерной текстуры

Загрузка текстуры и предоставление текстурных координат являются необходимыми требованиями наложения текстуры. Разберем простой пример применения двумерной текстуры. В приведенном ниже коде используются функции, которые уже рассматривали, и несколько новых. На основе данного примера и опишем указанные дополнительные вопросы наложения текстуры.

В листинге 1 приведен весь код программы, рисующей простую освещенную четырехгранную пирамиду, составленную из треугольников. На каждую грань пирамиды и ее основание наложена текстура камня. Результат выполнения программы показан на рис. 9.

#### Листинг 1 - Исходный код программы

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

// Rotation amounts
static GLfloat xRot = 0.0f;
static GLfloat yRot = 0.0f;

// Define targa header.
#pragma pack(1)
typedef struct
{
    GLbyte identsize;                // Size of ID
    field that follows header (0)
    GLbyte colorMapType;            // 0 = None, 1 =
    paletted
    GLbyte imageType;               // 0 = none, 1 =
    indexed, 2 = rgb, 3 = grey, +8=rle
```



```

        unsigned short    colorMapStart;           //
First colour map entry
        unsigned short    colorMapLength;         //
Number of colors
        unsigned char     colorMapBits;          // bits per
palette entry
        unsigned short     xstart;                // im-
age x origin
        unsigned short     ystart;                // im-
age y origin
        unsigned short     width;                 //
width in pixels
        unsigned short     height;                //
height in pixels
        GLbyte bits;                          // bits per pixel
(8 16, 24, 32)
        GLbyte descriptor;                      // image descrip-
tor
    } TGAHEADER;
#pragma pack(8)

```

```

////////////////////////////////////
////////////////////////////////////
// Allocate memory and load targa bits. Returns poin-
ter to new buffer,
// height, and width of texture, and the OpenGL for-
mat of data.
// Call free() on buffer when finished!
// This only works on pretty vanilla targas... 8, 24,
or 32 bit color
// only, no palettes, no RLE encoding.
GLbyte *gltLoadTGA(const char *szFileName, GLint
*iWidth, GLint *iHeight, GLint *iComponents, GLenum
*eFormat)
{

```

```

    FILE *pFile;                // File pointer
    TGAHEADER tgaHeader;        // TGA file header
    unsigned long lImageSize;    // Size in
bytes of image
    short sDepth;               // Pixel depth;
    GLbyte *pBits = NULL;       // Pointer to
bits

    // Default/Failed values
    *iWidth = 0;
    *iHeight = 0;
    *eFormat = GL_BGR_EXT;
    *iComponents = GL_RGB8;

    // Attempt to open the file
    pFile = fopen(szFileName, "rb");
    if(pFile == NULL)
        return NULL;

    // Read in header (binary)
    fread(&tgaHeader, 18/* sizeof(TGAHEADER)*/, 1,
pFile);

    // Do byte swap for big vs little endian
#ifdef __APPLE__
    BYTE_SWAP(tgaHeader.colorMapStart);
    BYTE_SWAP(tgaHeader.colorMapLength);
    BYTE_SWAP(tgaHeader.xstart);
    BYTE_SWAP(tgaHeader.ystart);
    BYTE_SWAP(tgaHeader.width);
    BYTE_SWAP(tgaHeader.height);
#endif

    // Get width, height, and depth of texture
    *iWidth = tgaHeader.width;

```

```

    *iHeight = tgaHeader.height;
    sDepth = tgaHeader.bits / 8;

    // Put some validity checks here. Very simply, I
    only understand
    // or care about 8, 24, or 32 bit targa's.
    if(tgaHeader.bits != 8 && tgaHeader.bits != 24 &&
    tgaHeader.bits != 32)
        return NULL;

    // Calculate size of image buffer
    lImageSize = tgaHeader.width * tgaHeader.height *
    sDepth;

    // Allocate memory and check for success
    pBits = (GLbyte*)malloc(lImageSize * si-
    zeof(GLbyte));
    if(pBits == NULL)
        return NULL;

    // Read in the bits
    // Check for read error. This should catch RLE or
    other
    // weird formats that I don't want to recognize
    if(fread(pBits, lImageSize, 1, pFile) != 1)
    {
        free(pBits);
        return NULL;
    }

    // Set OpenGL format expected
    switch(sDepth)
    {
    case 3:        // Most likely case
        *eFormat = GL_BGR_EXT;
        *iComponents = GL_RGB8;

```

```

        break;
    case 4:
        *eFormat = GL_BGRA_EXT;
        *iComponents = GL_RGBA8;
        break;
    case 1:
        *eFormat = GL_LUMINANCE;
        *iComponents = GL_LUMINANCE8;
        break;
};

// Done with File
fclose(pFile);

// Return pointer to image data
return pBits;
}

// Change viewing volume and viewport. Called when
window is resized
void ChangeSize(int w, int h)
{
    GLfloat fAspect;

    // Prevent a divide by zero
    if(h == 0)
        h = 1;

    // Set Viewport to window dimensions
    glViewport(0, 0, w, h);

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

    // Reset coordinate system

```

```

glMatrixMode(GL_PROJECTION);
glLoadIdentity();

// Produce the perspective projection
gluPerspective(35.0f, fAspect, 1.0, 40.0);

glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
}

// This function does any needed initialization on
the rendering
// context. Here it sets up and initializes the
lighting for
// the scene.
void SetupRC()
{
    GLubyte *pBytes;
    GLint iWidth, iHeight, iComponents;
    GLenum eFormat;

    // Light values and coordinates
    // GLfloat  whiteLight[] = { 0.05f, 0.05f, 0.05f,
1.0f };
    GLfloat  whiteLight[] = { 0.5f, 0.5f, 0.5f,
1.0f };
    GLfloat  sourceLight[] = { 0.75f, 0.75f, 0.75f,
1.0f };
    GLfloat  lightPos[] = { -10.f, 5.0f, 5.0f, 1.0f
};

    glEnable(GL_DEPTH_TEST); // Hidden surface removal
    glFrontFace(GL_CCW);      // Counter clock-
wise polygons face out

```

```

    glEnable(GL_CULL_FACE);          // Do not calculate
inside of jet

    // Enable lighting
    glEnable(GL_LIGHTING);

    // Setup and enable light 0
    glLightModelfv(GL_LIGHT_MODEL_AMBIENT, whiteLight);
    glLightfv(GL_LIGHT0, GL_AMBIENT, sourceLight);
    glLightfv(GL_LIGHT0, GL_DIFFUSE, sourceLight);
    glLightfv(GL_LIGHT0, GL_POSITION, lightPos);
    glEnable(GL_LIGHT0);

    // Enable color tracking
    glEnable(GL_COLOR_MATERIAL);

    // Set Material properties to follow glColor values
    glColorMaterial(GL_FRONT,
GL_AMBIENT_AND_DIFFUSE);

    // Black blue background
    glClearColor(0.0f, 0.0f, 0.0f, 1.0f );

    // Load texture
    glPixelStorei(GL_UNPACK_ALIGNMENT, 1);
    pBytes = (GLubyte *)glTLoadTGA("liq1.tga",
&iWidth, &iHeight, &iComponents, &eFormat);
    glTexImage2D(GL_TEXTURE_2D, 0, iComponents,
iWidth, iHeight, 0, eFormat, GL_UNSIGNED_BYTE,
pBytes);
    free(pBytes);

    glTexParameteri(GL_TEXTURE_2D,
GL_TEXTURE_MIN_FILTER, GL_LINEAR);

```

```

        glTexParameteri(GL_TEXTURE_2D,
GL_TEXTURE_MAG_FILTER, GL_LINEAR);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S,
GL_CLAMP_TO_EDGE);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T,
GL_CLAMP_TO_EDGE);

        glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE,
GL_MODULATE);
        glEnable(GL_TEXTURE_2D);
    }

// Respond to arrow keys
void SpecialKeys(int key, int x, int y)
{
    if(key == GLUT_KEY_UP)
        xRot -= 5.0f;

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

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

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

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

    // Refresh the Window
    glutPostRedisplay();
}

typedef GLfloat GLTVector3[3];           // Three compo-
nent floating point vector

```

```

// Subtract one vector from another
void gltSubtractVectors(const GLTVector3 vFirst,
const GLTVector3 vSecond, GLTVector3 vResult)
{
    vResult[0] = vFirst[0] - vSecond[0];
    vResult[1] = vFirst[1] - vSecond[1];
    vResult[2] = vFirst[2] - vSecond[2];
}

// Scales a vector by a scalar
void gltScaleVector(GLTVector3 vVector, const GLfloat
fScale)
{
    vVector[0] *= fScale; vVector[1] *= fScale; vVec-
tor[2] *= fScale;
}

// Gets the length of a vector squared
GLfloat gltGetVectorLengthSqr(const GLTVector3 vVec-
tor)
{
    return (vVector[0]*vVector[0]) + (vVec-
tor[1]*vVector[1]) + (vVector[2]*vVector[2]);
}

// Gets the length of a vector
GLfloat gltGetVectorLength(const GLTVector3 vVector)
{
    return
    (GLfloat)sqrt(gltGetVectorLengthSqr(vVector));
}

// Scales a vector by it's length - creates a unit
vector
void gltNormalizeVector(GLTVector3 vNormal)
{

```



```

        GLfloat fLength = 1.0f / glmGetVector-
Length(vNormal);
        glmScaleVector(vNormal, fLength);
    }
// Calculate the cross product of two vectors
void glmVectorCrossProduct(const glmVector3 vU, const
glmVector3 vV, glmVector3 vResult)
    {
        vResult[0] = vU[1]*vV[2] - vV[1]*vU[2];
        vResult[1] = -vU[0]*vV[2] + vV[0]*vU[2];
        vResult[2] = vU[0]*vV[1] - vV[0]*vU[1];
    }

// Called to draw scene
// Given three points on a plane in counter clockwise
order, calculate the unit normal
void glmGetNormalVector(const glmVector3 vP1, const
glmVector3 vP2, const glmVector3 vP3, glmVector3
vNormal)
    {
        glmVector3 vV1, vV2;

        glmSubtractVectors(vP2, vP1, vV1);
        glmSubtractVectors(vP3, vP1, vV2);

        glmVectorCrossProduct(vV1, vV2, vNormal);
        glmNormalizeVector(vNormal);
    }

void RenderScene(void)
    {
        glmVector3 vNormal;
        glmVector3 vCorners[5] = { { 0.0f, .80f, 0.0f },
// Top
        0

```

```

                                { -0.5f, 0.0f, -.50f },
// Back left      1
                                { 0.5f, 0.0f, -0.50f },
// Back right    2
                                { 0.5f, 0.0f, 0.5f },
// Front right   3
                                { -0.5f, 0.0f, 0.5f }};
// Front left    4

    // Clear the window with current clearing color
    glClear(GL_COLOR_BUFFER_BIT |
GL_DEPTH_BUFFER_BIT);

    // Save the matrix state and do the rotations
    glPushMatrix();
        // Move object back and do in place rotation
        glTranslatef(0.0f, -0.25f, -4.0f);
        glRotatef(xRot, 1.0f, 0.0f, 0.0f);
        glRotatef(yRot, 0.0f, 1.0f, 0.0f);

    // Draw the Pyramid
    glColor3f(1.0f, 1.0f, 1.0f);
    glBegin(GL_TRIANGLES);
        // Bottom section - two triangles
        glNormal3f(0.0f, -1.0f, 0.0f);
        glTexCoord2f(1.0f, 1.0f);
        glVertex3fv(vCorners[2]);

        glTexCoord2f(0.0f, 0.0f);
        glVertex3fv(vCorners[4]);

        glTexCoord2f(0.0f, 1.0f);
        glVertex3fv(vCorners[1]);

        glTexCoord2f(1.0f, 1.0f);

```

```

        glVertex3fv(vCorners[2]);

        glTexCoord2f(1.0f, 0.0f);
        glVertex3fv(vCorners[3]);

        glTexCoord2f(0.0f, 0.0f);
        glVertex3fv(vCorners[4]);

        // Front Face
        glGetNormalVector(vCorners[0], vCorners[4], vCorners[3], vNormal);
        glNormal3fv(vNormal);
        glTexCoord2f(0.5f, 1.0f);
        glVertex3fv(vCorners[0]);
        glTexCoord2f(0.0f, 0.0f);
        glVertex3fv(vCorners[4]);
        glTexCoord2f(1.0f, 0.0f);
        glVertex3fv(vCorners[3]);

        // Left Face
        glGetNormalVector(vCorners[0], vCorners[1], vCorners[4], vNormal);
        glNormal3fv(vNormal);
        glTexCoord2f(0.5f, 1.0f);
        glVertex3fv(vCorners[0]);
        glTexCoord2f(0.0f, 0.0f);
        glVertex3fv(vCorners[1]);
        glTexCoord2f(1.0f, 0.0f);
        glVertex3fv(vCorners[4]);

        // Back Face
        glGetNormalVector(vCorners[0], vCorners[2], vCorners[1], vNormal);
        glNormal3fv(vNormal);
        glTexCoord2f(0.5f, 1.0f);
        glVertex3fv(vCorners[0]);

```

```

        glTexCoord2f(0.0f, 0.0f);
        glVertex3fv(vCorners[2]);

        glTexCoord2f(1.0f, 0.0f);
        glVertex3fv(vCorners[1]);

        // Right Face
        gltGetNormalVector(vCorners[0], vCorners[3], vCorners[2], vNormal);
        glNormal3fv(vNormal);
        glTexCoord2f(0.5f, 1.0f);
        glVertex3fv(vCorners[0]);
        glTexCoord2f(0.0f, 0.0f);
        glVertex3fv(vCorners[3]);
        glTexCoord2f(1.0f, 0.0f);
        glVertex3fv(vCorners[2]);
    glEnd();

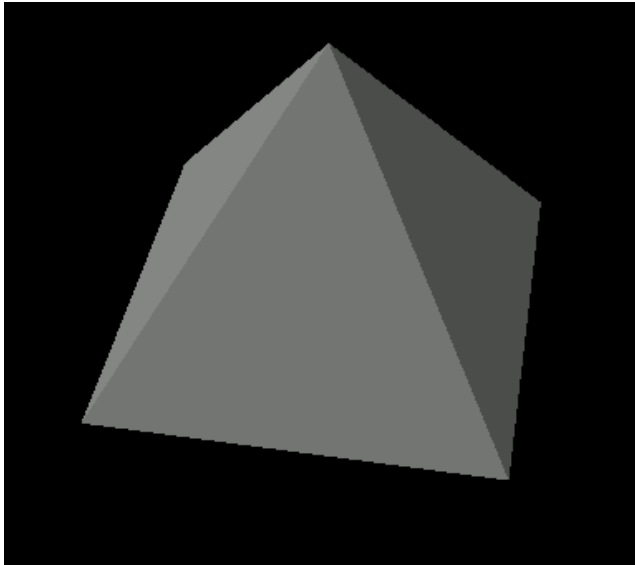
    // Restore the matrix state
    glPopMatrix();

    // Buffer swap
    glutSwapBuffers();
}

int main(int argc, char *argv[])
{
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH);
    glutInitWindowSize(800, 600);
    glutCreateWindow("Textured Pyramid");

```

```
glutReshapeFunc (ChangeSize);  
glutSpecialFunc (SpecialKeys);  
glutDisplayFunc (RenderScene);  
SetupRC ();  
glutMainLoop ();  
  
return 0;  
}
```



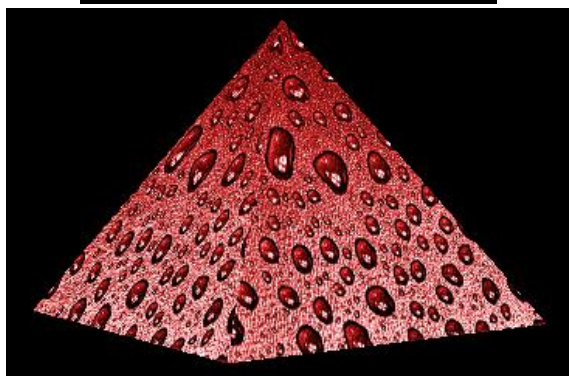


Рис.9. – Результат выполнения программы Листинг 1.

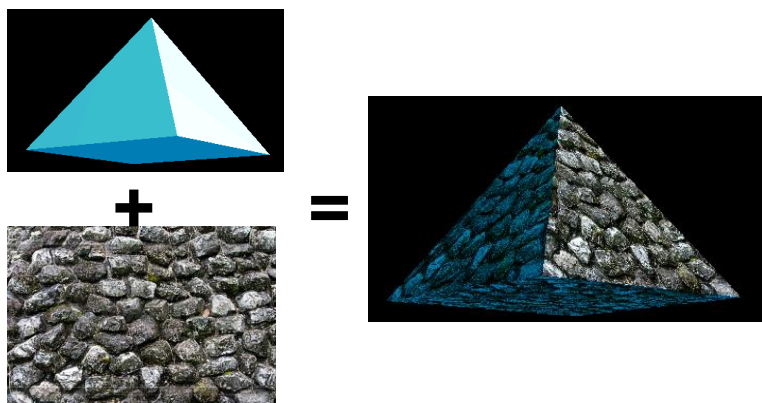


Рис.10. – Освещенная геометрия + текстура = затененная текстура.

Вся необходимая инициализация выполняется функцией `SetupRC`, в том числе загрузка текстуры с использованием функции `glLoadTGA` и предоставление битов функции [`glTexImage2D`](#)

```
// Load texture
glPixelStorei(GL_UNPACK_ALIGNMENT, 1);
pBytes = (GLubyte *)glLoadTGA("liq1.tga",
&iWidth, &iHeight, &iComponents, &eFormat);
glTexImage2D(GL_TEXTURE_2D, 0, iComponents,
iWidth, iHeight, 0, eFormat, GL_UNSIGNED_BYTE,
pBytes);
free(pBytes);
```

и конечно включается режим наложения текстуры  
`glEnable(GL_TEXTURE_2D);`

Функция `RenderScene` рисует пирамиду в виде ряда текстурных треугольников. В приведенном ниже фрагменте кода показано построение одной грани с помощью нормали (рассчитана с использованием угловых вершин), после чего следуют три текстурных координаты и координаты вершин.

// передняя грань

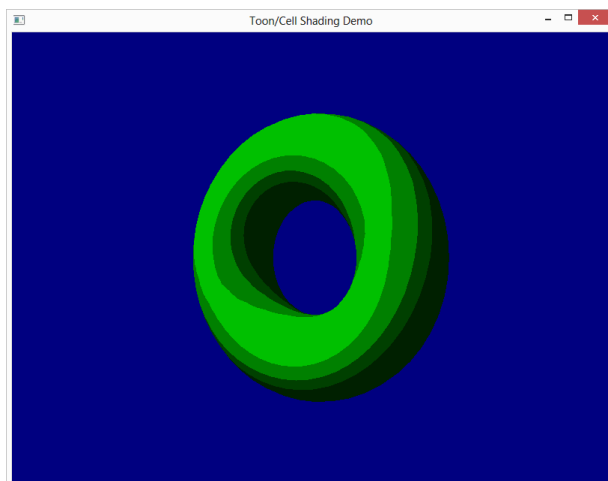
```
glGetNormalVector(vCorners[0], vCorners[4], vCorners[3], vNormal);
glNormal3fv(vNormal);
glTexCoord2f(0.5f, 1.0f);
glVertex3fv(vCorners[0]);
glTexCoord2f(0.0f, 0.0f);
glVertex3fv(vCorners[4]);
glTexCoord2f(1.0f, 0.0f);
glVertex3fv(vCorners[3]);
```

## Мультфильмы с текстурами

В первом примере использовались двухмерные текстуры, поскольку обычно они простые и их легче понять. Рассмотрим теперь наложение одномерной текстуры, широко используемое в компьютерных играх для визуализации геометрии, которая при отображении на экране создает иллюзию мультипликации. В мультипликационном, или келевом затенении в качестве таблицы

соответствий применяется одномерная карта текстуры, на основе которой геометрические объекты заполняются сплошным цветом (с помощью [GL\\_NEAREST](#)).

В основе подхода лежит идея использования нормали поверхности из геометрического описания объекта и нормали источника света для нахождения интенсивности света, падающего на поверхность модели. Скалярное произведение данных векторов дает значение между 0.0 и 1.0, которое используется как одномерная текстурная координата. В программе, представленной в листинге 2, с помощью данной техники рисуется зеленый тор. Результат выполнения программы показан на рис. 11.





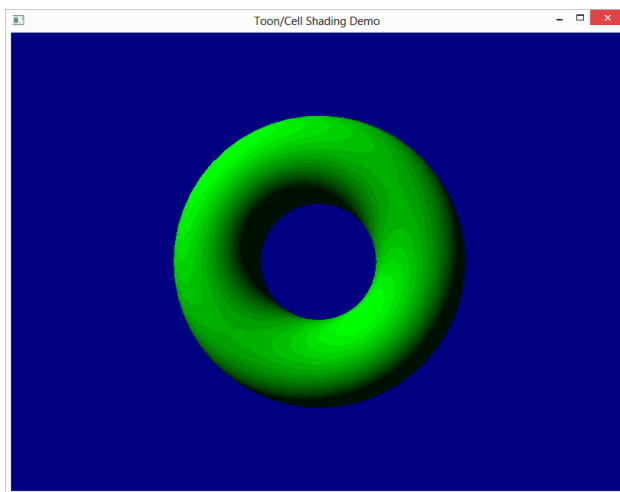


Рис.11. – Тор с келевым затенением

## Листинг 2 - Исходный код программы

```
//Рисуются тор (баранка); для затенения используется
текущая
//одномерная текстура
Void toonDrawTorus (GLfloat majorRadius, GLfloat mi-
norRadius, int numMajor, int numMinor, GLTVector3
vLightDir)
{
    GLTMatrix mModelViewMatrix;
    GLTVector3 vNormal vTransformedNormal;
    Double majorStep = 2.0f*GLT_PI / numMajor;
    Double minorStep = 2.0f*GLT_PI / numMinor;
    Int I, j;
    // Получается матрица наблюдения модели
    glGetFloatv (GL_MODELVIEW_MATRIX, mModelViewMa-
trix);
    // Нормируется вектор источника света
    gltNormalizeVector (vLightDir);
    // С помощью лент треугольников рисуется тор
    for (int i=0; i<numMajor; i++)
```

```

{
    Double a0=1 * majorStep;
    Double a1=a0 + majorStep;
    GLfloat x0 = (GLfloat) cos (a0);
    GLfloat y0 = (GLfloat) sin (a0);
    GLfloat x1 = (GLfloat) cos (a1);
    GLfloat y1 = (GLfloat) sin (a1);
    glBegin (GL_TRIANGLE_STRIP);
    for (j=0; j<=numMinor; ++j)
    {
        double b = j * minorStep;
        GLfloat c = (GLfloat) cos(b);
        GLfloat r = minorRadius * c + major-
Radius;
        GLfloat z = minorRadius * (GLfloat)
sin(b);
// Первая точка
        vNormal [0] = x0*c;
        vNormal [1] = y0*c;
        vNormal [2] = z/minorRadius;
        gltNormalizeVector(vNormal);
        gltRotateVector (vNormal, mModel-
ViewMatrix,
vTransformedNor-
mal);
// Текстурные координаты устанавливаются согласно
// интенсивности света
        glTexCoordlf (gltVectorDotProduct
(vLightDir, vTrans-
formedNormal));
        glVertex3f(x0*r, y0*r, z);
// Вторая точка
        vNormal [0] = x1*c;
        vNormal [1] = y1*c;
        vNormal [2] = z/minorRadius;
        gltNormalizeVector(vNormal);

```

```

        glRotateVector (vNormal, mModel-
                        ViewMatrix,
                        vTransformedNor-
                        mal);
// Текстурные координаты устанавливаются согласно
// интенсивности света
        glTexCoordf (glVectorDotProduct
                    (vLightDir, vTrans-
                    formedNormal));
        glVertex3f(x1*r, y1*r, z);
    }
    glEnd();
}
// Вызывается для рисования сцены
void RenderScene (void)
{
    // Угол поворота
    Static GLfloat yRot = 0.0f;
// Откуда приходит свет
    GLTVector3 vLightDir = { -1.0f, 1.0f,
    1.0f },
// Очищаем окно текущим цветом очистки
    glClear (GL_COLOR_BUFFER_BIT |
    GL_DEPTH_BUFFER_BIT);
    glPushMatrix ();
    glTranslatef (0.0f, 0.0f, -2.5f);
    glRotatef (yRot, 0.0f, 1.0f, 0.0f);
    toonDrawTorus (0.35f, 0.15f, 50, 25,
    vLightDir);
    glPopMatrix ();
// Переключает буферы
    glutSwapBuffers ();
// Каждый кадр поворачивается на ½ градуса
    yRot += 0.5f;
}

```

```

// Эта функция выполняет необходимую инициализацию в
контексте визуализации
void SetupRC ()
{
// Загружается одномерная текстура с кодом
// мультипликационного затенения
// Зелёный, более зелёный...
    GLbyte toonTable [4] [3] = { { 0, 32, 0 },
                                   { 0, 64, 0 },
                                   { 0, 128, 0 },
                                   { 0, 192, 0 } };

// Синеватый фон
    glClearColor (0.0f, 0.0f, 0.5f, 1.0f);
    glEnable (GL_DEPTH_TEST);
    glEnable (GL_CULL_FACE);
    glTexEnvf (GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE,
GL_DECAL);
    glTexParameterf (GL_TEXTURE_1D,
                     GL_TEXTURE_MAG_FILTER,
                     GL_NEAREST);
    glTexParameterf (GL_TEXTURE_1D,
                     GL_TEXTURE_MIN_FILTER,
                     GL_NEAREST);
    glTexParameterf (GL_TEXTURE_1D,
GL_TEXTURE_WRAP_S, GL_CLAMP);
    glPixelStoref (GL_UNPACK_ALIGNMENT, 1);
    glTexImage1D (GL_TEXTURE_1D, 0, GL_RGB, 4, 0,
                  GL_RGB, GL_UNSIGNED_BYTE, toon-
                  Table);
    glEnable (GL_TEXTURE_1D);
}

// Вызывается библиотекой GLUT в холостом режиме (ок-
но не
// двигается и не меняет размеров)
void TimerFunction (int value)
{

```

```

        // Сцена перерисовывается с новыми координатами
        glutPostRedisplay ();
        glutTimeFunc (33, TimerFunction, 1);
    }
void ChangeSize (int w, int h)
{
    GLfloat fAspect;
    // Предотвращает деление на ноль, когда окно
слишком
//маленькое
    // (нельзя сделать окно нулевой ширины).
    If (h==0)
        h=1;
    glViewport (0, 0, w,h);
    fAspect = (GLfloat) w / (GLfloat) h;
// Система координат обновляется перед модификацией
    glMatrixMode (GL_PROJECTION);
    glLoadIdentity ();
// Устанавливается объём сечения
    gluPerspective (35.0f, fAspect, 1.0f, 50.0f);
    glMatrixMode (GL_MODELVIEW);
    glLoadIdentity ();
}
// Точка входа программы
Int main (int argc, char* argv[])
{
    glutInit (&argc, argv);
    glutInitDisplayMode (GLUT_DOUBLE | GLUT_RGB |
GLUT_DEPTH);
    glutInitWindowSize (800, 600);
    glutCreateWindow ("Shading Demo");
    glutReshapeFunc (ChangeSize);
    glutDisplayFunc (RenderScene);
    glutTimerFunc (33, TimerFunction, 1);
    SetupRC ();
    glutMainLoop ();
}

```

```
    return 0;
}
```

## Полный код листинга 6.2.

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

#define GLT_PI    3.14159265358979323846

////////////////////////////////////
//
// Some data types
typedef GLfloat GLTVector2[2];      // Two component
floating point vector
typedef GLfloat GLTVector3[3];      // Three compo-
nent floating point vector
typedef GLfloat GLTVector4[4];      // Four component
floating point vector
typedef GLfloat GLTMatrix[16];      // A column major
4x4 matrix of type GLfloat

// Adds two vectors together
void gltAddVectors(const GLTVector3 vFirst, const
GLTVector3 vSecond, GLTVector3 vResult) {
    vResult[0] = vFirst[0] + vSecond[0];
    vResult[1] = vFirst[1] + vSecond[1];
    vResult[2] = vFirst[2] + vSecond[2];
}

// Subtract one vector from another
void gltSubtractVectors(const GLTVector3 vFirst,
const GLTVector3 vSecond, GLTVector3 vResult)
{
    vResult[0] = vFirst[0] - vSecond[0];
    vResult[1] = vFirst[1] - vSecond[1];
```

```

    vResult[2] = vFirst[2] - vSecond[2];
}

// Scales a vector by a scalar
void gltScaleVector(GLTVector3 vVector, const GLfloat
fScale)
{
    vVector[0] *= fScale; vVector[1] *= fScale; vVec-
tor[2] *= fScale;
}

// Gets the length of a vector squared
GLfloat gltGetVectorLengthSqr(const GLTVector3 vVec-
tor)
{
    return (vVector[0]*vVector[0]) + (vVec-
tor[1]*vVector[1]) + (vVector[2]*vVector[2]);
}

// Gets the length of a vector
GLfloat gltGetVectorLength(const GLTVector3 vVector)
{
    return
    (GLfloat)sqrt(gltGetVectorLengthSqr(vVector));
}

// Scales a vector by it's length - creates a unit
vector
void gltNormalizeVector(GLTVector3 vNormal)
{
    GLfloat fLength = 1.0f / gltGetVector-
Length(vNormal);
    gltScaleVector(vNormal, fLength);
}
// Transform a point by a 4x4 matrix

```

```

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

// Rotates a vector using a 4x4 matrix. Translation
column is ignored
void glRotateVector(const GLTVector3 vSrcVector,
const GLTMatrix mMatrix, GLTVector3 vOut)
{
    vOut[0] = mMatrix[0] * vSrcVector[0] + mMatrix[4]
* vSrcVector[1] + mMatrix[8] * vSrcVector[2];
    vOut[1] = mMatrix[1] * vSrcVector[0] + mMatrix[5]
* vSrcVector[1] + mMatrix[9] * vSrcVector[2];
    vOut[2] = mMatrix[2] * vSrcVector[0] + mMatrix[6]
* vSrcVector[1] + mMatrix[10] * vSrcVector[2];
}

// Get the dot product between two vectors
GLfloat glVectorDotProduct(const GLTVector3 vU,
const GLTVector3 vV)
{
    return vU[0]*vV[0] + vU[1]*vV[1] + vU[2]*vV[2];
}

// Draw a torus (doughnut), using the current 1D tex-
ture for light shading

```



```

void toonDrawTorus(GLfloat majorRadius, GLfloat minorRadius,
                  int numMajor, int numMinor, GLVector3 vLightDir)
{
    GLTMatrix mModelViewMatrix;
    GLTVector3 vNormal, vTransformedNormal;
    double majorStep = 2.0f*GLT_PI / numMajor;
    double minorStep = 2.0f*GLT_PI / numMinor;
    int i, j;

    // Get the modelview matrix
    glGetFloatv(GL_MODELVIEW_MATRIX, mModelViewMatrix);

    // Normalize the light vector
    gltNormalizeVector(vLightDir);

    // Draw torus as a series of triangle strips
    for (i=0; i<numMajor; ++i)
    {
        double a0 = i * majorStep;
        double a1 = a0 + majorStep;
        GLfloat x0 = (GLfloat) cos(a0);
        GLfloat y0 = (GLfloat) sin(a0);
        GLfloat x1 = (GLfloat) cos(a1);
        GLfloat y1 = (GLfloat) sin(a1);

        glBegin(GL_TRIANGLE_STRIP);
        for (j=0; j<=numMinor; ++j)
        {
            double b = j * minorStep;
            GLfloat c = (GLfloat) cos(b);
            GLfloat r = minorRadius * c + majorRadius;
            glNormal3f(x0*c, y0*c, r);
            glVertex3f(x0*c, y0*c, r);
            glNormal3f(x1*c, y1*c, r);
            glVertex3f(x1*c, y1*c, r);
            glNormal3f(x0*c, y0*c, r);
            glVertex3f(x0*c, y0*c, r);
            glNormal3f(x1*c, y1*c, r);
            glVertex3f(x1*c, y1*c, r);
        }
    }
}

```

```

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

        // First point
        vNormal[0] = x0*c;
        vNormal[1] = y0*c;
        vNormal[2] = z/minorRadius;
        gltNormalizeVector(vNormal);
        gltRotateVector(vNormal, mModelViewMa-
trix, vTransformedNormal);

        // Texture coordinate is set by intensity
of light
        glTex-
Coord1f(gltVectorDotProduct(vLightDir, vTransformed-
Normal));
        glVertex3f(x0*r, y0*r, z);

        // Second point
        vNormal[0] = x1*c;
        vNormal[1] = y1*c;
        vNormal[2] = z/minorRadius;
        gltNormalizeVector(vNormal);
        gltRotateVector(vNormal, mModelViewMa-
trix, vTransformedNormal);

        // Texture coordinate is set by intensity
of light
        glTex-
Coord1f(gltVectorDotProduct(vLightDir, vTransformed-
Normal));
        glVertex3f(x1*r, y1*r, z);
    }
    glEnd();
}
}

```

```

// Called to draw scene
void RenderScene(void)
{
    // Rotation angle
    static GLfloat yRot = 0.0f;

    // Where is the light coming from
    GLTVector3 vLightDir = { -1.0f, 1.0f, 1.0f };

    // Clear the window with current clearing color
    glClear(GL_COLOR_BUFFER_BIT |
GL_DEPTH_BUFFER_BIT);
    glPushMatrix();
        glTranslatef(0.0f, 0.0f, -2.5f);
        glRotatef(yRot, 0.0f, 1.0f, 0.0f);
        toonDrawTorus(0.35f, 0.15f, 50, 25,
vLightDir);
        glPopMatrix();

    // Do the buffer Swap
    glutSwapBuffers();

    // Rotate 1/2 degree more each frame
    yRot += 0.5f;
}

// This function does any needed initialization on
the rendering
// context.
void SetupRC()
{
    // Load a 1D texture with toon shaded values
    // Green, greener...
    GLbyte toonTable[4][3] = { { 0, 32, 0 },

```

```

                                { 0, 64, 0 },
                                { 0, 128, 0 },
                                { 0, 192, 0 } };

// Bluish background
glClearColor(0.0f, 0.0f, .50f, 1.0f );
glEnable(GL_DEPTH_TEST);
glEnable(GL_CULL_FACE);

glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE,
GL_DECAL);
glTexParameteri(GL_TEXTURE_1D,
GL_TEXTURE_MAG_FILTER, GL_NEAREST);
glTexParameteri(GL_TEXTURE_1D,
GL_TEXTURE_MIN_FILTER, GL_NEAREST);
glTexParameteri(GL_TEXTURE_1D, GL_TEXTURE_WRAP_S,
GL_CLAMP);
glPixelStorei(GL_UNPACK_ALIGNMENT, 1);
glTexImage1D(GL_TEXTURE_1D, 0, GL_RGB, 4, 0,
GL_RGB, GL_UNSIGNED_BYTE, toonTable);

glEnable(GL_TEXTURE_1D);
}

////////////////////////////////////
/////
// Called by GLUT library when idle (window not being
// resized or moved)
void TimerFunction(int value)
{
    // Redraw the scene with new coordinates
    glutPostRedisplay();
    glutTimerFunc(33,TimerFunction, 1);
}

```

```

void ChangeSize(int w, int h)
{
    GLfloat fAspect;

    // Prevent a divide by zero, when window is too
short
    // (you cant make a window of zero width).
    if(h == 0)
        h = 1;

    glViewport(0, 0, w, h);

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

    // Reset the coordinate system before modifying
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();

    // Set the clipping volume
    gluPerspective(35.0f, fAspect, 1.0f, 50.0f);

    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
}

////////////////////////////////////
// Program entry point
int main(int argc, char* argv[])
{
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB |
GLUT_DEPTH);
    glutInitWindowSize(800,600);
    glutCreateWindow("Toon/Cell Shading Demo");
    glutReshapeFunc(ChangeSize);
}

```

```
glutDisplayFunc(RenderScene);  
glutTimerFunc(33, TimerFunction, 1);  
  
SetupRC();  
glutMainLoop();  
  
return 0;  
}
```

## Управление несколькими текстурами

В общем случае текстурные объекты применяются для загрузки нескольких текстур в ходе инициализации программы и быстрого переключения между ними в процессе визуализации. Когда программа завершается, текстурные объекты удаляются. При запуске приводимой ниже программы загружаются три текстуры, которые затем переключаются при визуализации туннеля. Туннель имеет кирпичные стены и пол и потолок из разных материалов. Результат выполнения программы показан на рис. 12.

Программа также иллюстрирует множественное отображение и различные режимы фильтрации текстуры множественного отображения. Нажимая клавиши со стрелками вверх и вниз, вы перемещаете точку наблюдения назад-вперед по туннелю, а контекстное меню (вызывается щелчком правой кнопки мыши) позволяет переключаться между шестью различными режимами фильтрации и сравнивать их влияние на визуализацию изображения. Исходный код программы приводится в листинге 3.

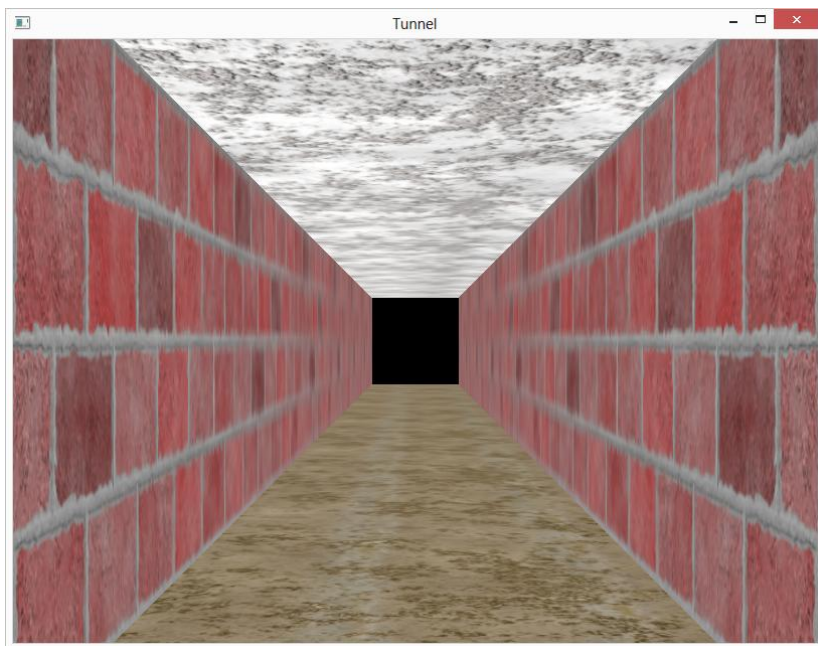


Рис.12. – Туннель, визуализированный с тремя различными текстурами

### Листинг 3 - Исходный код программы

```
// Величины поворота
static GLfloat zPos = -60.0f;
// Текстурные объекты
#define TEXTURE_BRICK 0
#define TEXTURE_FLOOR 1
#define TEXTURE_CEILING 2
#define TEXTURE_COUNT 3
GLuint textures [TEXTURE_COUNT];
const char *szTextureFiles[TEXTURE_COUNT] = {
"brick.tga", "floor.tga", "ceiling.tga" };
////////////////////////////////////
////////// Меняется текстурный фильтр для каждого
текстурного объекта
```

```

void ProcessMenu (int value)
{
    GLint iLoop;
    for (iLoop = 0; iLoop < TEXTURE_COUNT; iLoop++)
    {
        glBindTexture (GL_TEXTURE_2D, textures[iLoop]);
        switch(value)
        {
            case 0:
                glTexParameteri
                    (GL_TEXTURE_2D,
                     GL_TEXTURE_MIN_FILTER,
                     GL_LINEAR,
                     GL_NEAREST);

                break;
            case 1:
                glTexParameteri
                    (GL_TEXTURE_2D,
                     GL_TEXTURE_MIN_FILTER,
                     GL_LINEAR);

                break;
            case 2:
                glTexParameteri
                    (GL_TEXTURE_2D,
                     GL_TEXTURE_MIN_FILTER,
                     GL_LINEAR,
                     GL_NEAREST_MIPMAP_NEAREST);

                break;
            case 3:
                glTexParameteri
                    (GL_TEXTURE_2D,
                     GL_TEXTURE_MIN_FILTER,
                     GL_LINEAR,

```



GL\_NEAREST\_MIPMAP  
LINEAR);

```
        break;
    case 4:
        glTexParameteri
            (GL_TEXTURE_2D,
             GL_TEXTURE_MIN_FI
             LTER,
             GL_LINEAR_MIPMAP_
             NEAREST);

        break;
    case 5:
    default:
        glTexParameteri
            (GL_TEXTURE_2D,
             GL_TEXTURE_MIN_FI
             LTER,
             GL_LINEAR_MIPMAP_
             LINEAR);

        break;
    }

    }

    // Иницируется перерисовывание
    glutPostRedisplay ();
}

////////////////////////////////////
////////////////////////////////////
// Эта функция выполняет необходимую инициализацию в
// контексте // визуализации. Здесь задаются и инициали-
// зируются текстурные
// объекты
void SetupRC ()
{
    GLubyte *pBytes;
    GLint iWidth, lHeight, iComponents;
    GLenum eFormat;
```

```

GLint iLoop;
    // Черный фон
glClearColor(0.0f, 0.0f, 0.0f, 1.0f);
    // Текстуры применяются как переводные
    рисунки, без эффектов освещения или
    // окрашивания
glEnable (GL_TEXTURE_2D);
glTexEnvf (GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE,
GL_DECAL);
    // Загружаются текстуры
glGenTextures (TEXTURE_COUNT, textures);
for(iLoop = 0; iLoop < TEXTURE_COUNT; iLoop++)
{
    // Связывание со следующим текстурным
    объектом
    glBindTexture (GL_TEXTURE_2D, tex-
    tures[iLoop]);
    // Загружается текстура, устанавливаются
    режимы
    // фильтрации и намотки
    pBytes = gltLoadTGA (szTexture-
                        Files[iLoop], SiWidth,
                        SiHeight, SiComponents,
                        SeFormat);
    gluBuild2DMipmaps (GL_TEXTURE_2D, iCompo-
    nents, iWidth, lHeight,
    eFormat,
    GL_UNSIGNED_BYTE,
    pBytes);
    glTexParameterf (GL_TEXTURE_2D,
GL_TEXTURE_MAG_FILTER, GL_LINEAR);
    glTexParameterf (GL_TEXTURE_2D,
                        GL_TEXTURE_MIN_FILTER,
                        GL_LINEAR_MIPMAP_LINEAR);

```

```

        glTexParameteri (GL_TEXTURE_2D,
                           GL_TEXTURE_WRAP_S,
                           GL_CLAMP_TO_EDGE);
        glTexParameteri (GL_TEXTURE_2D,
                           GL_TEXTURE_WRAP_T,
                           GL_CLAMP_TO_EDGE);
        // Первоначальные данные текстуры уже не нужны
        Free (pBytes);
    }

}

////////////////////////////////////
////////////////////////////////////
// Выключается контекст визуализации. Просто удаляются
// текстурные объекты
void ShutdownRC (void)
{
    glDeleteTextures (TEXTURE_COUNT, textures);
}

////////////////////////////////////
////////////////////////////////////
// В ответ на нажатия клавиш со стрелками точка на-
// блюдения
// перемещается взад-вперед
void SpecialKeys (int key, int x, int y)
{
    If (key == GLUT_KEY_UP)
        zPos += 1.0f;
    if (key == GLUT_KEY_DOWN)
        zPos -= 1.0f;
    // Обновляет окно
    glutPostRedisplay () ;
}

////////////////////////////////////
////////////////////////////////////
// Меняет наблюдаемый объем и поле просмотра.

```

```

// Вызывается при изменении размеров окна
void ChangeSize (int w, int h)
{
    GLfloat fAspect;
    // Предотвращает деление на нуль
    If (h == 0)
        h = 1;
    // Размер поля просмотра устанавливается равным
размеру окна
    glViewport (0, 0, w, h);
    fAspect = (GLfloat)w/(GLfloat)h;
    // Обновляется система координат
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity ();
    // Генерируется перспективная проекция
    gluPerspective (90.0f, fAspect, 1, 120);
    glMatrixMode (GL_MODELVIEW);
    glLoadIdentity ();
}
////////////////////////////////////
////////////////////////////////////
// Вызывается для рисования сцены
void RenderScene (void)
{
    GLfloat z;
    // Очищаем окно текущим цветом очистки
    glClear (GL_COLOR_BUFFER_BIT);
    // Записывается состояние матрицы и выполняются
повороты
    glPushMatrix ();
    // Объекты перемещаются назад, и выполняется
поворот на
    // месте
    glTranslatef(0.0f, 0.0f, zPos);
    // Пол
    for(z = 60. 0f; z >= 0.0f; z -= 10)

```

```

{
glBindTexture (GL_TEXTURE_2D, textures[TEXTURE_FLOOR]);
glBegin (GL_QUADS);
    glTexCoord2f (0.0f, 0.0f);
    glVertex3f (-10.0f, -10.0f, z);
    glTexCoord2f (1.0f, 0.0f);
    glVertex3f (10.0f, -10.0f, z);
    glTexCoord2f (1.0f, 1.0f);
    glVertex3f (10.0f, -10.0f, z -
10.0f);
    glTexCoord2f (0.0f, 1.0f);
    glVertex3f (-10.0f, -10.0f, z -
10.0f);
glEnd ();
// Потолок
glBindTexture (GL_TEXTURE_2D, textures[TEXTURE_CEILING]);
glBegin (GL_QUADS);
    glTexCoord2f (0.0f, 1.0f);
    glVertex3f (-10.0f, 10.0f, z -
10.0f);
    glTexCoord2f (1.0f, 1.0f);
    glVertex3f (10.0f, 10.0f, z -
10.0f);
    glTexCoord2f (1.0f, 0.0f);
    glVertex3f (10.0f, 10.0f, z);
    glTexCoord2f (0.0f, 0.0f);
    glVertex3f (-10.0f, 10.0f, z);
glEnd ();
// Левая стена
glBindTexture (GL_TEXTURE_2D, textures[TEXTURE_BRICK]);
glBegin (GL_QUADS);
    glTexCoord2f (0.0f, 0.0f);
    glVertex3f (-10.0f, -10.0f, z);

```

```

        glTexCoord2f (1.0f, 0.0f);
        glVertex3f (-10.0f, -10.0f, z -
10.0f);
        glTexCoord2f (1.0f, 1.0f);
        glVertex3f (-10.0f, 10.0f, z -
10.0f);
        glTexCoord2f (0.0f, 1.0f);
        glVertex3f (-10.0f, 10.0f, z);
glEnd ();
// Правая стена
glBegin(GL_QUADS);
        glTexCoord2f (0.0f, 1.0f);
        glVertex3f (10.0f, 10.0f, z);
        glTexCoord2f (1.0f, 1.0f);
        glVertex3f (10.0f, 10.0f, z -
10.0f);
        glTexCoord2f (1.0f, 0.0f);
        glVertex3f (10.0f, -10.0f, z -
10.0f);
        glTexCoord2f (0.0f, 0.0f);
        glVertex3f (10.0f, -10.0f, z);
        glEnd ();
    }

    // Восстанавливается состояние матрицы
    glPopMatrix ();
    // Переключение буферов
    glutSwapBuffers ();
}

////////////////////////////////////
////////////////////////////////////
// Точка входа программы
int main (int argc, char *argv[])
{
    // Стандартный набор инициализации
    glutInit (&argc, argv);
    glutInitDisplayMode (GLUT_DOUBLE | GLUT_RGB);

```

```

    glutImtWindowSize (800, 600);
    glutCreateWindow ("Tunnel");
    glutReshapeFunc (ChangeSize);
    glutSpecialFunc (SpecialKeys);
    glutDisplayFunc (RenderScene);
    // Добавляются позиции меню для изменения
    фильтра
    glutCreateMenu (ProcessMenu);
    glutAddMenuEntry ("GL_NEAREST",0);
    glutAddMenuEntry ("GL_LINEAR",1);
    glutAddMenuEntry ("GL_NEAREST_MIPMAP_NEAREST",
2) ;
    glutAddMenuEntry ("GL_NEAREST_MIPMAP_LINEAR",
3);
    glutAddMenuEntry ("GL_LINEAR_MIPMAP_NEAREST",
4);
    glutAddMenuEntry ("GL_LINEAR_MIPMAP_LINEAR",
5);
    glutAttachMenu (GLUT_RIGHT_BUTTON);
    // Запуск, основной цикл, выключение
    SetupRC ();
    glutMainLoop ();
    ShutdownRC ();
    return 0;
}

```

## Полный код листинга

```

#include "glew.h"
#include "glut.h"
#include <math.h>
#include <stdio.h>
#include <stdlib.h>

// Rotation amounts

```

```

static GLfloat zPos = -60.0f;

// Texture objects
#define TEXTURE_BRICK    0
#define TEXTURE_FLOOR   1
#define TEXTURE_CEILING 2
#define TEXTURE_COUNT    3
GLuint textures[TEXTURE_COUNT];
const char *szTextureFiles[TEXTURE_COUNT] = {
    "brick.tga", "floor.tga", "ceiling.tga" };
// Define targa header.
#pragma pack(1)
typedef struct
{
    GLbyte identsize;           // Size of ID
    field that follows header (0)
    GLbyte colorMapType;       // 0 = None, 1 =
    paletted
    GLbyte imageType;          // 0 = none, 1 =
    indexed, 2 = rgb, 3 = grey, +8=rle
    unsigned short colorMapStart; //
    First colour map entry
    unsigned short colorMapLength; //
    Number of colors
    unsigned char colorMapBits; // bits per
    palette entry
    unsigned short xstart;      // im-
    age x origin
    unsigned short ystart;      // im-
    age y origin
    unsigned short width;       //
    width in pixels
    unsigned short height;      //
    height in pixels
    GLbyte bits;                // bits per pixel
    (8 16, 24, 32)

```



```

        GLbyte descriptor;                // image descrip-
tor
    } TGAHEADER;
#pragma pack(8)

////////////////////////////////////
////////////////////////////////////
// Allocate memory and load targa bits. Returns poin-
ter to new buffer,
// height, and width of texture, and the OpenGL for-
mat of data.
// Call free() on buffer when finished!
// This only works on pretty vanilla targas... 8, 24,
or 32 bit color
// only, no palettes, no RLE encoding.
GLbyte *gltLoadTGA(const char *szFileName, GLint
*iWidth, GLint *iHeight, GLint *iComponents, GLenum
*eFormat)
{
    FILE *pFile;                // File pointer
    TGAHEADER tgaHeader;        // TGA file header
    unsigned long lImageSize;    // Size in
bytes of image
    short sDepth;                // Pixel depth;
    GLbyte *pBits = NULL;       // Pointer to
bits

    // Default/Failed values
    *iWidth = 0;
    *iHeight = 0;
    *eFormat = GL_BGR_EXT;
    *iComponents = GL_RGB8;

    // Attempt to open the fil

```

```

pFile = fopen(szFileName, "rb");
if(pFile == NULL)
    return NULL;

// Read in header (binary)
fread(&tgaHeader, 18/* sizeof(TGAHEADER)*/, 1,
pFile);

// Do byte swap for big vs little endian
#ifdef __APPLE__
    BYTE_SWAP(tgaHeader.colorMapStart);
    BYTE_SWAP(tgaHeader.colorMapLength);
    BYTE_SWAP(tgaHeader.xstart);
    BYTE_SWAP(tgaHeader.ystart);
    BYTE_SWAP(tgaHeader.width);
    BYTE_SWAP(tgaHeader.height);
#endif

// Get width, height, and depth of texture
*iWidth = tgaHeader.width;
*iHeight = tgaHeader.height;
sDepth = tgaHeader.bits / 8;

// Put some validity checks here. Very simply, I
only understand
// or care about 8, 24, or 32 bit targa's.
if(tgaHeader.bits != 8 && tgaHeader.bits != 24 &&
tgaHeader.bits != 32)
    return NULL;

// Calculate size of image buffer
lImageSize = tgaHeader.width * tgaHeader.height *
sDepth;

// Allocate memory and check for success

```

```

    pBits = (GLbyte*)malloc(lImageSize * sizeof(GLbyte));
    if(pBits == NULL)
        return NULL;

    // Read in the bits
    // Check for read error. This should catch RLE or
other
    // weird formats that I don't want to recognize
    if(fread(pBits, lImageSize, 1, pFile) != 1)
    {
        free(pBits);
        return NULL;
    }

    // Set OpenGL format expected
    switch(sDepth)
    {
        case 3:        // Most likely case
            *eFormat = GL_BGR_EXT;
            *iComponents = GL_RGB8;
            break;
        case 4:
            *eFormat = GL_BGRA_EXT;
            *iComponents = GL_RGBA8;
            break;
        case 1:
            *eFormat = GL_LUMINANCE;
            *iComponents = GL_LUMINANCE8;
            break;
    };

    // Done with File
    fclose(pFile);

```

```

// Return pointer to image data
return pBits;
}

////////////////////////////////////
////////////////////////////////////
// Change texture filter for each texture object
void ProcessMenu(int value)
{
    GLint iLoop;

    for(iLoop = 0; iLoop < TEXTURE_COUNT; iLoop++)
    {
        glBindTexture(GL_TEXTURE_2D, textures[iLoop]);

        switch(value)
        {
            case 0:
                glTexParameteri(GL_TEXTURE_2D,
GL_TEXTURE_MIN_FILTER, GL_NEAREST);
                break;

            case 1:
                glTexParameteri(GL_TEXTURE_2D,
GL_TEXTURE_MIN_FILTER, GL_LINEAR);
                break;

            case 2:
                glTexParameteri(GL_TEXTURE_2D,
GL_TEXTURE_MIN_FILTER, GL_NEAREST_MIPMAP_NEAREST);
                break;

            case 3:

```

```

        glTexParameteri(GL_TEXTURE_2D,
GL_TEXTURE_MIN_FILTER, GL_NEAREST_MIPMAP_LINEAR);
        break;

```

```

        case 4:
            glTexParameteri(GL_TEXTURE_2D,
GL_TEXTURE_MIN_FILTER, GL_LINEAR_MIPMAP_NEAREST);
            break;

```

```

        case 5:
        default:
            glTexParameteri(GL_TEXTURE_2D,
GL_TEXTURE_MIN_FILTER, GL_LINEAR_MIPMAP_LINEAR);
            break;
        }
    }

```

```

// Trigger Redraw
glutPostRedisplay();
}

```

```

////////////////////////////////////
////////////////////////////////////

```

```

// This function does any needed initialization on
the rendering
// context. Here it sets up and initializes the tex-
ture objects.

```

```

void SetupRC()
{
    GLubyte *pBytes;
    GLint iWidth, iHeight, iComponents;
    GLenum eFormat;
    GLint iLoop;

```

```

    // Black background

```

```

        glClearColor(0.0f, 0.0f, 0.0f,1.0f);

        // Textures applied as decals, no lighting or coloring effects
        glEnable(GL_TEXTURE_2D);
        glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_DECAL);

        // Load textures
        glGenTextures(TEXTURE_COUNT, textures);
        for(iLoop = 0; iLoop < TEXTURE_COUNT; iLoop++)
        {
            // Bind to next texture object
            glBindTexture(GL_TEXTURE_2D, textures[iLoop]);

            // Load texture, set filter and wrap modes
            pBytes = (GLubyte*)glTLoadTGA(szTextureFiles[iLoop],&iWidth,
            &iHeight, &iComponents, &eFormat);
            gluBuild2DMipmaps(GL_TEXTURE_2D, iComponents, iWidth, iHeight, eFormat, GL_UNSIGNED_BYTE, pBytes);
            glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
            glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR_MIPMAP_LINEAR);
            glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
            glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);

            // Don't need original texture data any more
            free(pBytes);
        }
    }

```

```

////////////////////////////////////
// Shutdown the rendering context. Just deletes the
// texture objects
void ShutdownRC(void)

```

```

{
    glDeleteTextures(TEXTURE_COUNT, textures);
}

```

```

////////////////////////////////////
// Respond to arrow keys, move the viewpoint back
// and forth

```

```

void SpecialKeys(int key, int x, int y)

```

```

{
    if(key == GLUT_KEY_UP)
        zPos += 1.0f;

    if(key == GLUT_KEY_DOWN)
        zPos -= 1.0f;

    // Refresh the Window
    glutPostRedisplay();
}

```

```

////////////////////////////////////
////////////////////////////////////

```

```

// Change viewing volume and viewport. Called when
// window is resized

```

```

void ChangeSize(int w, int h)

```

```

{
    GLfloat fAspect;

    // Prevent a divide by zero
    if(h == 0)
        h = 1;
}

```

```

// Set Viewport to window dimensions
glViewport(0, 0, w, h);

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

// Reset coordinate system
glMatrixMode(GL_PROJECTION);
glLoadIdentity();

// Produce the perspective projection
gluPerspective(90.0f, fAspect, 1, 120);


glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
}

////////////////////////////////////
//
// Called to draw scene
void RenderScene(void)
{
    GLfloat z;

    // Clear the window with current clearing color
    glClear(GL_COLOR_BUFFER_BIT);

    // Save the matrix state and do the rotations
    glPushMatrix();
        // Move object back and do in place rotation
        glTranslatef(0.0f, 0.0f, zPos);

    // Floor
    for(z = 60.0f; z >= 0.0f; z -= 10)
    {

```



```

        glBindTexture(GL_TEXTURE_2D, textures[TEXTURE_FLOOR]);
        glBegin(GL_QUADS);
            glTexCoord2f(0.0f, 0.0f);
            glVertex3f(-10.0f, -10.0f, z);

            glTexCoord2f(1.0f, 0.0f);
            glVertex3f(10.0f, -10.0f, z);

            glTexCoord2f(1.0f, 1.0f);
            glVertex3f(10.0f, 10.0f, z -
10.0f);

            glTexCoord2f(0.0f, 1.0f);
            glVertex3f(-10.0f, 10.0f, z -
10.0f);

        glEnd();

        // Ceiling
        glBindTexture(GL_TEXTURE_2D, textures[TEXTURE_CEILING]);
        glBegin(GL_QUADS);
            glTexCoord2f(0.0f, 0.0f);
            glVertex3f(-10.0f, -10.0f, z -
10.0f);

            glTexCoord2f(1.0f, 0.0f);
            glVertex3f(10.0f, -10.0f, z - 10.0f);

            glTexCoord2f(1.0f, 1.0f);
            glVertex3f(10.0f, 10.0f, z);

            glTexCoord2f(0.0f, 1.0f);
            glVertex3f(-10.0f, 10.0f, z);
        glEnd();

```

```

        // Left Wall
        glBindTexture(GL_TEXTURE_2D, textures[TEXTURE_BRICK]);
        glBegin(GL_QUADS);
            glTexCoord2f(0.0f, 0.0f);
            glVertex3f(-10.0f, -10.0f, z);

            glTexCoord2f(1.0f, 0.0f);
            glVertex3f(-10.0f, -10.0f, z -
10.0f);

            glTexCoord2f(1.0f, 1.0f);
            glVertex3f(-10.0f, 10.0f, z -
10.0f);

            glTexCoord2f(0.0f, 1.0f);
            glVertex3f(-10.0f, 10.0f, z);
        glEnd();

        // Right Wall
        glBegin(GL_QUADS);
            glTexCoord2f(0.0f, 1.0f);
            glVertex3f(10.0f, 10.0f, z);

            glTexCoord2f(1.0f, 1.0f);
            glVertex3f(10.0f, 10.0f, z - 10.0f);

            glTexCoord2f(1.0f, 0.0f);
            glVertex3f(10.0f, -10.0f, z -
10.0f);

            glTexCoord2f(0.0f, 0.0f);
            glVertex3f(10.0f, -10.0f, z);
        glEnd();

```

```

    }

    // Restore the matrix state
    glPopMatrix();

    // Buffer swap
    glutSwapBuffers();
}

////////////////////////////////////
/
// Program entry point
int main(int argc, char *argv[])
{
    // Standard initialization stuff
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB);
    glutInitWindowSize(800, 600);
    glutCreateWindow("Tunnel");
    glutReshapeFunc(ChangeSize);
    glutSpecialFunc(SpecialKeys);
    glutDisplayFunc(RenderScene);

    // Add menu entries to change filter
    glutCreateMenu(ProcessMenu);
    glutAddMenuEntry("GL_NEAREST", 0);
    glutAddMenuEntry("GL_LINEAR", 1);
    glutAddMenuEntry("GL_NEAREST_MIPMAP_NEAREST", 2);
    glutAddMenuEntry("GL_NEAREST_MIPMAP_LINEAR", 3);
    glutAddMenuEntry("GL_LINEAR_MIPMAP_NEAREST", 4);
    glutAddMenuEntry("GL_LINEAR_MIPMAP_LINEAR", 5);
    glutAttachMenu(GLUT_RIGHT_BUTTON);

    // Startup, loop, shutdown
    SetupRC();
}

```

```

glutMainLoop();
ShutdownRC();

return 0;
}

```

В данном примере вначале создаются идентификаторы трех текстурных объектов. Массив `textures` содержит три целых числа, которые представляют макросы `TEXTURE_BRICK`, `TEXTURE_FLOOR` и `TEXTURE_CEILING`. С целью увеличения гибкости также создается макрос, определяющий максимальное число текстур, которые будут загружены, и массив строк символов, содержащий имена файлов карт текстуры.

```

// Текстурные объекты
#define TEXTURE_BRICK 0
#define TEXTURE_FLOOR 1
#define TEXTURE_CEILING 2
#define TEXTURE_COUNT 3
GLuint textures [TEXTURE_COUNT];
const char *szTextureFiles[TEXTURE_COUNT] ={
"brick.tga", "floor.tga", "ceiling.tga" };

```

Текстурные объекты распределяются в функции `SetupRC`.

```
glGenTextures (TEXTURE_COUNT, textures);
```

Далее с каждым текстурным объектом связывается простой цикл и в состояние текстуры этого объекта загружается изображение текстуры и параметры ее нанесения.

```

for (iLoop = 0; iLoop < TEXTURE_COUNT; iLoop++)
{
    // Связывание со следующим текстурным объектом
    glBindTexture (GL_TEXTURE_2D, textures[iLoop]);
    // Загружается текстура, устанавливаются режимы
    // фильтрации и
    // намотки
    pBytes = gltLoadTGA (szTexture-
                        Files[iLoop], SiWidth, Si-
                        Height, SiComponents, SeFor-
                        mat);
}

```

```

gluBuild2DMipmaps (GL_TEXTURE_2D, iComponents,
                  iWidth, iHeight, eFormat,
                  GL_UNSIGNED_BYTE, pBytes);
glTexParameteri (GL_TEXTURE_2D,
                  GL_TEXTURE_MAG_FILTER,
                  GL_LINEAR);
glTexParameteri (GL_TEXTURE_2D,
                  GL_TEXTURE_MIN_FILTER,
                  GL_LINEAR_MIPMAP_LINEAR);
glTexParameteri (GL_TEXTURE_2D,
                  GL_TEXTURE_WRAP_S,
                  GL_CLAMP_TO_EDGE);
glTexParameteri (GL_TEXTURE_2D,
                  GL_TEXTURE_WRAP_T,
                  GL_CLAMP_TO_EDGE);
// Первоначальные данные текстуры уже не
// нужны
free(pBytes);
}

```

Завершив инициализацию всех трех текстурных объектов, можно переключаться между ними в процессе визуализации, меняя текстуры.

```

glBindTexture(GL_TEXTURE_2D, textures[TEXTURE_FLOOR]);
glBegin(GL_QUADS);
glTexCoord2f (0.0f, 0.0f);
glVertex3f (-10.0f, -10.0f, z);
glTexCoord2f (1.0f, 0.0f);
glVertex3f (10.0f, -10.0f, z);

```

Наконец, программа завершается, для полной очистки осталось только удалить текстурные объекты.

```

////////////////////////////////////
////////////////////////////////////
// Выключается контекст визуализации. Просто удаляются
// текстурные объекты
void ShutdownRC(void)

```

```
{
glDeleteTextures (TEXTURE_COUNT, textures);
}
```

Обратите также внимание на то, что при установке фильтра множественной текстуры в программе он выбирается только для уменьшающего фильтра

```
glTexParameteri (GL_TEXTURE_2D,
                  GL_TEXTURE_MAG_FILTER,
                  GL_LINEAR);
glTexParameteri (GL_TEXTURE_2D,
                  GL_TEXTURE_MIN_FILTER,
                  GL_LINEAR_MIPMAP_LINEAR);
```

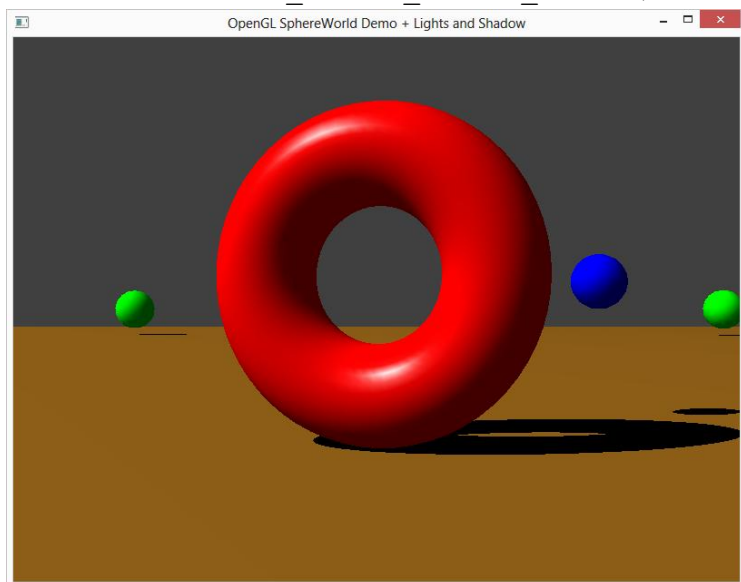


Рис.13. – Исходный топ SPHEREWORLD с зеркальными бликами

Данная ситуация является характерной, поскольку после того как OpenGL выберет наибольший доступный уровень множественного отображения, более высоких уровней, из которых можно было бы выбирать текстуру, уже нет. Можно сказать, что после прохождения

определенного порога применяется наибольшее доступное текстурное изображение, и нет дополнительных уровней множественной текстуры, из которых можно было бы выбрать больший рисунок.

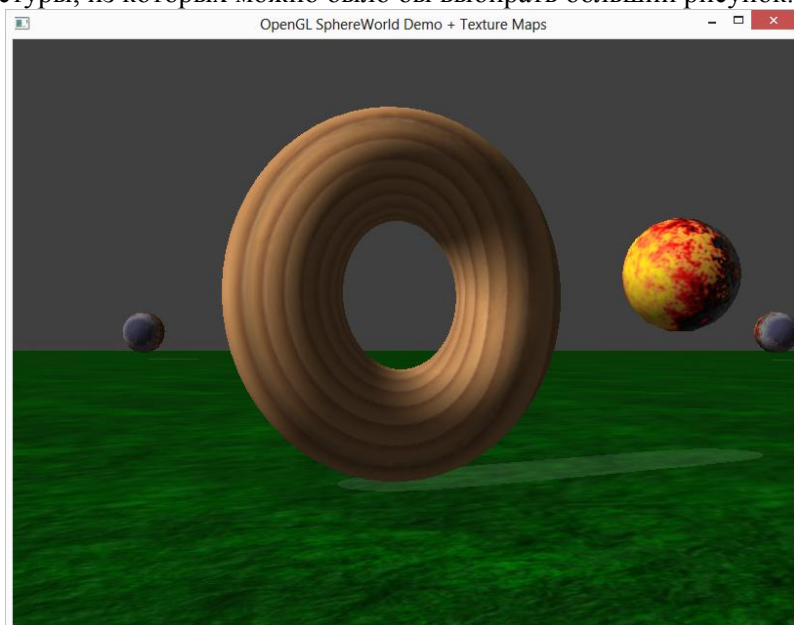


Рис.14. – Текстурный тор с приглушёнными бликами.

## Дополнительные возможности освещения

Наложение текстуры на геометрический объект дает после применения освещения скрытый и часто нежелательный побочный эффект. В общем случае текстурная среда устанавливается равной `GL_MODULATE`, что приводит к такому объединению освещенной геометрии с картой текстуры, что текстурная геометрия также кажется освещенной. Обычно OpenGL рассчитывает освещение и цвета отдельных фрагментов согласно стандартной модели освещения. Затем цвета фрагментов объединяются с отфильтрованными текселями, которые налагаются на геометрические объекты. Однако данный процесс обладает побочным эффектом: существенно уменьшается видимость зеркальных "зайчиков" на поверхности.

На рис. 13 показано исходное изображение мира сфер. На этом рисунке отчетливо видны блики на поверхности тора. На рис. 14

показан результат выполнения программы SPHEREWORLD. На этом рисунке можно наблюдать эффекты наложения текстуры после добавления освещения.

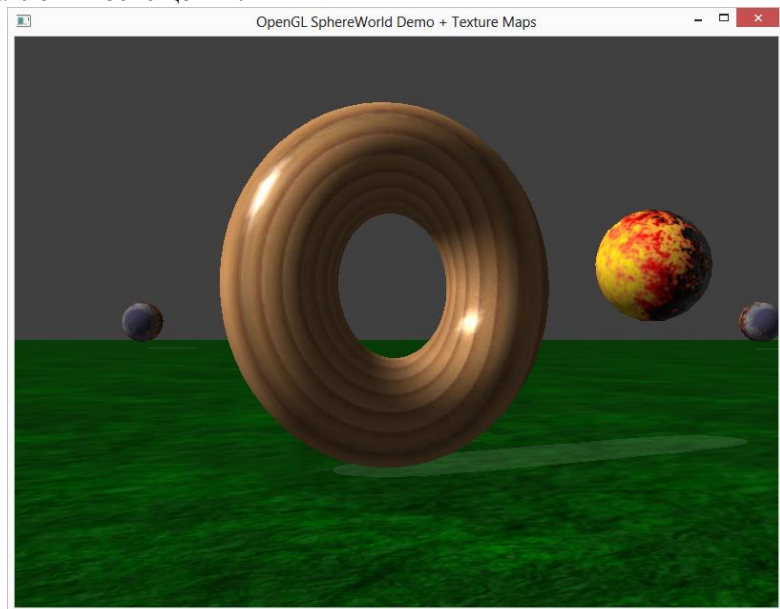


Рис.15. – Блики, восстановленные на текстурном торе

Решением данной проблемы является применение зеркальных бликов после наложения текстуры. Данную модель, получившую название *вторичного зеркального цвета*, можно применять вручную или автоматически (в рамках модели освещения). Обычно при этом выбирается стандартная модель освещения OpenGL, просто включаемая с помощью `glLightModel`.

```
glLightModeli (GL_LIGHT_MODEL_COLOR_CONTROL,  
               GL_SEPARATE_SPECULAR_COLOR) ;
```

Чтобы позже вернуться к нормальной модели освещения, можно указать `GL_SINGLE_COLOR` в качестве параметра модели света.

```
glLightModeli (GL_LIGHT_MODEL_COLOR_CONTROL,  
               GL_COLOR_SINGLE) ;
```

На рис. 15 показан результат выполнения программы SPHEREWORLD (очередная версия) с восстановленными зеркальными



бликами на торе. В данном листинге программы всего лишь добавлена указанная выше строчка кода.

Кроме того, вызвав функцию `glSecondaryColor`, можно непосредственно задать вторичный цвет после наложения текстуры, если вы не используете освещение (освещение деактивизировано). Эта функция, как и `glColor`, имеет множество разновидностей. Следует также отметить, что, задав вторичный цвет, вы также должны явно активизировать его использование.

```
glEnable (GL_COLOR_SUM) ;
```

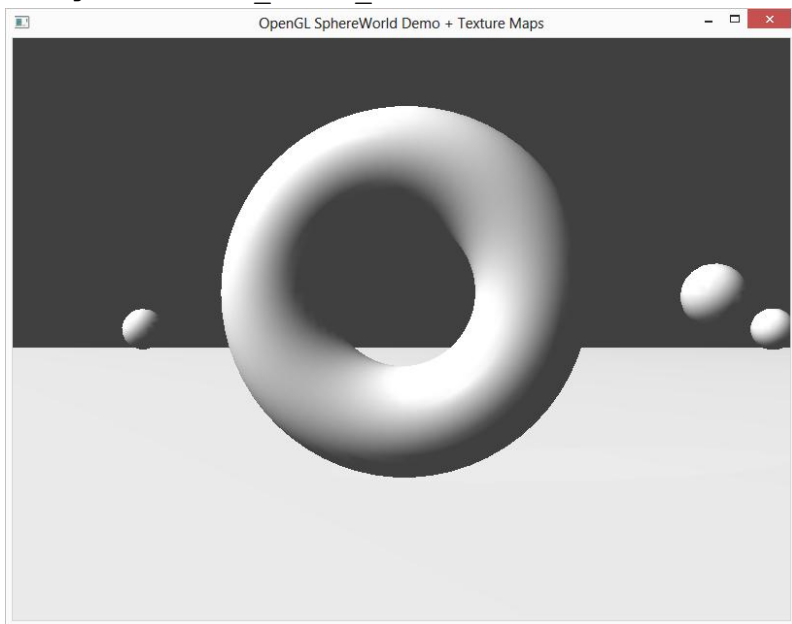


Рис.16. – Блики, восстановленные на торе и других объектов без наложения текстуры.

#### Листинг 4 - Исходный код программы

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

```

#include <windows.h>

////////////////////////////////////
//
// Useful constants
#define GLT_PI    3.14159265358979323846
#define GLT_PI_DIV_180  0.017453292519943296
#define GLT_INV_PI_DIV_180  57.2957795130823229

////////////////////////////////////
////////////////////////////////////
// Useful shortcuts and macros
// Radians are king... but we need a way to swap back
and forth
#define gltDegToRad(x) ((x)*GLT_PI_DIV_180)
#define gltRadToDeg(x) ((x)*GLT_INV_PI_DIV_180)

////////////////////////////////////
//
// Some data types
typedef GLfloat GLTVector2[2];      // Two component
floating point vector
typedef GLfloat GLTVector3[3];      // Three compo-
nent floating point vector
typedef GLfloat GLTVector4[4];      // Four component
floating point vector
typedef GLfloat GLTMatrix[16];      // A column major
4x4 matrix of type GLfloat

typedef struct{                      // The Frame of
reference container
    GLTVector3 vLocation;
    GLTVector3 vUp;
    GLTVector3 vForward;

```

```

    } GLTFrame;

#define NUM_SPHERES      30
GLTFrame    spheres[NUM_SPHERES];
GLTFrame    frameCamera;

// Define targa header.
#pragma pack(1)
typedef struct
{
    GLbyte identsize;                // Size of ID
field that follows header (0)
    GLbyte colorMapType;            // 0 = None, 1 =
paletted
    GLbyte imageType;              // 0 = none, 1 =
indexed, 2 = rgb, 3 = grey, +8=rle
    unsigned short    colorMapStart;        //
First colour map entry
    unsigned short    colorMapLength;       //
Number of colors
    unsigned char     colorMapBits;        // bits per
palette entry
    unsigned short    xstart;              // im-
age x origin
    unsigned short    ystart;              // im-
age y origin
    unsigned short    width;               //
width in pixels
    unsigned short    height;              //
height in pixels
    GLbyte bits;                // bits per pixel
(8 16, 24, 32)
    GLbyte descriptor;          // image descrip-
tor
} TGAHEADER;
#pragma pack(8)

```

```

////////////////////////////////////
////////////////////////////////////
// Allocate memory and load targa bits. Returns pointer to new buffer,
// height, and width of texture, and the OpenGL format of data.
// Call free() on buffer when finished!
// This only works on pretty vanilla targas... 8, 24, or 32 bit color
// only, no palettes, no RLE encoding.
GLbyte *gltLoadTGA(const char *szFileName, GLint *iWidth, GLint *iHeight, GLint *iComponents, GLenum *eFormat)
{
    FILE *pFile;           // File pointer
    TGAHEADER tgaHeader;    // TGA file header
    unsigned long lImageSize; // Size in
bytes of image
    short sDepth;           // Pixel depth;
    GLbyte *pBits = NULL;   // Pointer to
bits

    // Default/Failed values
    *iWidth = 0;
    *iHeight = 0;
    *eFormat = GL_BGR_EXT;
    *iComponents = GL_RGB8;

    // Attempt to open the file
    pFile = fopen(szFileName, "rb");
    if(pFile == NULL)
        return NULL;

```

```

    // Read in header (binary)
    fread(&tgaHeader, 18/* sizeof(TGAHEADER)*/, 1,
pFile);

    // Do byte swap for big vs little endian
#ifdef __APPLE__
    BYTE_SWAP(tgaHeader.colorMapStart);
    BYTE_SWAP(tgaHeader.colorMapLength);
    BYTE_SWAP(tgaHeader.xstart);
    BYTE_SWAP(tgaHeader.ystart);
    BYTE_SWAP(tgaHeader.width);
    BYTE_SWAP(tgaHeader.height);
#endif

    // Get width, height, and depth of texture
    *iWidth = tgaHeader.width;
    *iHeight = tgaHeader.height;
    sDepth = tgaHeader.bits / 8;

    // Put some validity checks here. Very simply, I
only understand
    // or care about 8, 24, or 32 bit targa's.
    if(tgaHeader.bits != 8 && tgaHeader.bits != 24 &&
tgaHeader.bits != 32)
        return NULL;

    // Calculate size of image buffer
    lImageSize = tgaHeader.width * tgaHeader.height *
sDepth;

    // Allocate memory and check for success
    pBits = (GLbyte *)malloc(lImageSize * si-
zeof(GLbyte));
    if(pBits == NULL)
        return NULL;

```

```

    // Read in the bits
    // Check for read error. This should catch RLE or
other
    // weird formats that I don't want to recognize
    if(fread(pBits, lImageSize, 1, pFile) != 1)
    {
        free(pBits);
        return NULL;
    }

    // Set OpenGL format expected
    switch(sDepth)
    {
        case 3:        // Most likely case
            *eFormat = GL_BGR_EXT;
            *iComponents = GL_RGB8;
            break;
        case 4:
            *eFormat = GL_BGRA_EXT;
            *iComponents = GL_RGBA8;
            break;
        case 1:
            *eFormat = GL_LUMINANCE;
            *iComponents = GL_LUMINANCE8;
            break;
    };

    // Done with File
    fclose(pFile);

    // Return pointer to image data
    return pBits;
}

```

```

// Adds two vectors together
void gltAddVectors(const GLTVector3 vFirst, const
GLTVector3 vSecond, GLTVector3 vResult) {
    vResult[0] = vFirst[0] + vSecond[0];
    vResult[1] = vFirst[1] + vSecond[1];
    vResult[2] = vFirst[2] + vSecond[2];
}

// Subtract one vector from another
void gltSubtractVectors(const GLTVector3 vFirst,
const GLTVector3 vSecond, GLTVector3 vResult)
{
    vResult[0] = vFirst[0] - vSecond[0];
    vResult[1] = vFirst[1] - vSecond[1];
    vResult[2] = vFirst[2] - vSecond[2];
}

// Scales a vector by a scalar
void gltScaleVector(GLTVector3 vVector, const GLfloat
fScale)
{
    vVector[0] *= fScale; vVector[1] *= fScale; vVec-
tor[2] *= fScale;
}

// Gets the length of a vector squared
GLfloat gltGetVectorLengthSqr(const GLTVector3 vVec-
tor)
{
    return (vVector[0]*vVector[0]) + (vVec-
tor[1]*vVector[1]) + (vVector[2]*vVector[2]);
}

// Gets the length of a vector
GLfloat gltGetVectorLength(const GLTVector3 vVector)
{

```

```

        return
        (GLfloat)sqrt (gltGetVectorLengthSqr(vVector));
    }

// Scales a vector by it's length - creates a unit
vector
void gltNormalizeVector(GLTVector3 vNormal)
{
    GLfloat fLength = 1.0f / gltGetVector-
Length(vNormal);
    gltScaleVector(vNormal, fLength);
}

// Copies a vector
void gltCopyVector(const GLTVector3 vSource, GLTVec-
tor3 vDest)
{
    memcpy(vDest, vSource, sizeof(GLTVector3));
}

// Get the dot product between two vectors
GLfloat gltVectorDotProduct(const GLTVector3 vU,
const GLTVector3 vV)
{
    return vU[0]*vV[0] + vU[1]*vV[1] + vU[2]*vV[2];
}

// Calculate the cross product of two vectors
void gltVectorCrossProduct(const GLTVector3 vU, const
GLTVector3 vV, GLTVector3 vResult)
{
    vResult[0] = vU[1]*vV[2] - vV[1]*vU[2];
    vResult[1] = -vU[0]*vV[2] + vV[0]*vU[2];
    vResult[2] = vU[0]*vV[1] - vV[0]*vU[1];
}

```



```

// Given three points on a plane in counter clockwise
order, calculate the unit normal
void gltGetNormalVector(const GLTVector3 vP1, const
GLTVector3 vP2, const GLTVector3 vP3, GLTVector3
vNormal)
{
    GLTVector3 vV1, vV2;

    gltSubtractVectors(vP2, vP1, vV1);
    gltSubtractVectors(vP3, vP1, vV2);

    gltVectorCrossProduct(vV1, vV2, vNormal);
    gltNormalizeVector(vNormal);
}

// Transform a point by a 4x4 matrix
void gltTransformPoint(const GLTVector3 vSrcVector,
const GLTMatrix mMatrix, GLTVector3 vOut)
{
    vOut[0] = mMatrix[0] * vSrcVector[0] + mMatrix[4]
* vSrcVector[1] + mMatrix[8] * vSrcVector[2] + mMa-
trix[12];
    vOut[1] = mMatrix[1] * vSrcVector[0] + mMatrix[5]
* vSrcVector[1] + mMatrix[9] * vSrcVector[2] + mMa-
trix[13];
    vOut[2] = mMatrix[2] * vSrcVector[0] + mMatrix[6]
* vSrcVector[1] + mMatrix[10] * vSrcVector[2] + mMa-
trix[14];
}

// Rotates a vector using a 4x4 matrix. Translation
column is ignored

```

```

void gltRotateVector(const GLTVector3 vSrcVector,
const GLTMatrix mMatrix, GLTVector3 vOut)
{
    vOut[0] = mMatrix[0] * vSrcVector[0] + mMatrix[4]
* vSrcVector[1] + mMatrix[8] * vSrcVector[2];
    vOut[1] = mMatrix[1] * vSrcVector[0] + mMatrix[5]
* vSrcVector[1] + mMatrix[9] * vSrcVector[2];
    vOut[2] = mMatrix[2] * vSrcVector[0] + mMatrix[6]
* vSrcVector[1] + mMatrix[10] * vSrcVector[2];
}

```

// Gets the three coefficients of a plane equation  
given three points on the plane.

```

void gltGetPlaneEquation(GLTVector3 vPoint1, GLTVec-
tor3 vPoint2, GLTVector3 vPoint3, GLTVector3 vPlane)

```

```

{
    // Get normal vector from three points. The nor-
mal vector is the first three coefficients
    // to the plane equation...
    gltGetNormalVector(vPoint1, vPoint2, vPoint3,
vPlane);

```

```

    // Final coefficient found by back substitution
    vPlane[3] = -(vPlane[0] * vPoint3[0] + vPlane[1]
* vPoint3[1] + vPlane[2] * vPoint3[2]);
}

```

// Determine the distance of a point from a plane,  
given the point and the  
// equation of the plane.

```

GLfloat gltDistanceToPlane(GLTVector3 vPoint, GLTVec-
tor4 vPlane)

```

```

{
    return vPoint[0]*vPlane[0] + vPoint[1]*vPlane[1]
+ vPoint[2]*vPlane[2] + vPlane[3];
}

```

```

}

// For best results, put this in a display list
// Draw a sphere at the origin
void gltDrawSphere(GLfloat fRadius, GLint iSlices,
GLint iStacks)
{
    GLfloat drho = (GLfloat)(3.141592653589) /
(GLfloat) iStacks;
    GLfloat dtheta = 2.0f * (GLfloat)(3.141592653589)
/ (GLfloat) iSlices;
    GLfloat ds = 1.0f / (GLfloat) iSlices;
    GLfloat dt = 1.0f / (GLfloat) iStacks;
    GLfloat t = 1.0f;
    GLfloat s = 0.0f;
    GLint i, j;        // Looping variables

    for (i = 0; i < iStacks; i++)
    {
        GLfloat rho = (GLfloat)i * drho;
        GLfloat srho = (GLfloat)(sin(rho));
        GLfloat crho = (GLfloat)(cos(rho));
        GLfloat srhodrho = (GLfloat)(sin(rho +
drho));
        GLfloat crhodrho = (GLfloat)(cos(rho +
drho));

        // Many sources of OpenGL sphere drawing code
uses a triangle fan
        // for the caps of the sphere. This however
introduces texturing
        // artifacts at the poles on some OpenGL im-
plementations
        glBegin(GL_TRIANGLE_STRIP);

```

```

        s = 0.0f;
        for ( j = 0; j <= iSlices; j++)
        {
            GLfloat theta = (j == iSlices) ?
0.0f : j * dtheta;
            GLfloat stheta = (GLfloat)(-
sin(theta));
            GLfloat ctheta =
(GLfloat)(cos(theta));

            GLfloat x = stheta * srho;
            GLfloat y = ctheta * srho;
            GLfloat z = crho;

            glTexCoord2f(s, t);
            glNormal3f(x, y, z);
            glVertex3f(x * fRadius, y * fRadius, z *
fRadius);

            x = stheta * srhodrho;
            y = ctheta * srhodrho;
            z = crhodrho;
            glTexCoord2f(s, t - dt);
            s += ds;
            glNormal3f(x, y, z);
            glVertex3f(x * fRadius, y * fRadius, z *
fRadius);
        }
        glEnd();

        t -= dt;
    }
}

// Initialize a frame of reference.

```

```

// Uses default OpenGL viewing position and orienta-
tion
void gltInitFrame(GLTFrame *pFrame)
{
    pFrame->vLocation[0] = 0.0f;
    pFrame->vLocation[1] = 0.0f;
    pFrame->vLocation[2] = 0.0f;

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

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

////////////////////////////////////
////////////////////////////////////
// Derives a 4x4 transformation matrix from a frame
of reference
void gltGetMatrixFromFrame(GLTFrame *pFrame, GLTMatrix mMatrix)
{
    GLTVector3 vXAxis;          // Derived X Axis

    // Calculate X Axis
    gltVectorCrossProduct(pFrame->vUp, pFrame->vForward, vXAxis);

    // Just populate the matrix
    // X column vector
    memcpy(mMatrix, vXAxis, sizeof(GLTVector3));
    mMatrix[3] = 0.0f;

    // y column vector

```

```

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

        // z column vector
        memcpy(mMatrix+8, pFrame->vForward, sizeof(GLTVector3));
        mMatrix[11] = 0.0f;

        // Translation/Location vector
        memcpy(mMatrix+12, pFrame->vLocation, sizeof(GLTVector3));
        mMatrix[15] = 1.0f;
    }

    //////////////////////////////////////
    //////////////////////////////////////
    // Apply an actors transform given it's frame of reference
    void gltApplyActorTransform(GLTFrame *pFrame)
    {
        GLTMatrix mTransform;
        gltGetMatrixFromFrame(pFrame, mTransform);
        glMultMatrixf(mTransform);
    }

    //////////////////////////////////////
    //////////////////////////////////////
    // Apply a camera transform given a frame of reference. This is
    // pretty much just an alternate implementation of gluLookAt using
    // floats instead of doubles and having the forward vector specified
    // instead of a point out in front of me.
    void gltApplyCameraTransform(GLTFrame *pCamera)

```

```

{
    GLTMatrix mMatrix;
    GLTVector3 vAxisX;
    GLTVector3 zFlipped;

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

    // Derive X vector
    gltVectorCrossProduct(pCamera->vUp, zFlipped,
vAxisX);

    // Populate matrix, note this is just the rota-
tion and is transposed
    mMatrix[0] = vAxisX[0];
    mMatrix[4] = vAxisX[1];
    mMatrix[8] = vAxisX[2];
    mMatrix[12] = 0.0f;

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

    mMatrix[2] = zFlipped[0];
    mMatrix[6] = zFlipped[1];
    mMatrix[10] = zFlipped[2];
    mMatrix[14] = 0.0f;

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

    // Do the rotation first

```

```

    glMultMatrixf(mMatrix);

    // Now, translate backwards
    glTranslatef(-pCamera->vLocation[0], -pCamera->
vLocation[1], -pCamera->vLocation[2]);
}

////////////////////////////////////
////
// March a frame of reference forward. This simply
moves
// the location forward along the forward vector.
void gltMoveFrameForward(GLTFrame *pFrame, GLfloat
fStep)
{
    pFrame->vLocation[0] += pFrame->vForward[0] *
fStep;
    pFrame->vLocation[1] += pFrame->vForward[1] *
fStep;
    pFrame->vLocation[2] += pFrame->vForward[2] *
fStep;
}

////////////////////////////////////
////
// Move a frame of reference up it's local Y axis
void gltMoveFrameUp(GLTFrame *pFrame, GLfloat fStep)
{
    pFrame->vLocation[0] += pFrame->vUp[0] * fStep;
    pFrame->vLocation[1] += pFrame->vUp[1] * fStep;
    pFrame->vLocation[2] += pFrame->vUp[2] * fStep;
}

////////////////////////////////////
////
// Move a frame of reference along it's local X axis

```



```

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

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

////////////////////////////////////
////
// Translate a frame in world coordinates
void gltTranslateFrameWorld(GLTFrame *pFrame, GLfloat
x, GLfloat y, GLfloat z)
{ pFrame->vLocation[0] += x; pFrame->vLocation[1]
+= y; pFrame->vLocation[2] += z; }

////////////////////////////////////
////
// Translate a frame in local coordinates
void gltTranslateFrameLocal(GLTFrame *pFrame, GLfloat
x, GLfloat y, GLfloat z)
{
    gltMoveFrameRight(pFrame, x);
    gltMoveFrameUp(pFrame, y);
    gltMoveFrameForward(pFrame, z);
}

// Creates a shadow projection matrix out of the
plane equation
// coefficients and the position of the light. The
return value is stored
// in destMat

```

```

void gltMakeShadowMatrix(GLTVector3 vPoints[3],
GLTVector4 vLightPos, GLTMatrix destMat)
{
    GLTVector4 vPlaneEquation;
    GLfloat dot;

    gltGetPlaneEquation(vPoints[0], vPoints[1],
vPoints[2], vPlaneEquation);

    // Dot product of plane and light position
    dot =    vPlaneEquation[0]*vLightPos[0] +
            vPlaneEquation[1]*vLightPos[1] +
            vPlaneEquation[2]*vLightPos[2] +
            vPlaneEquation[3]*vLightPos[3];

    // Now do the projection
    // First column
    destMat[0] = dot - vLightPos[0] * vPlaneEqua-
tion[0];
    destMat[4] = 0.0f - vLightPos[0] * vPlaneEqua-
tion[1];
    destMat[8] = 0.0f - vLightPos[0] * vPlaneEqua-
tion[2];
    destMat[12] = 0.0f - vLightPos[0] * vPlaneEqua-
tion[3];

    // Second column
    destMat[1] = 0.0f - vLightPos[1] * vPlaneEqua-
tion[0];
    destMat[5] = dot - vLightPos[1] * vPlaneEqua-
tion[1];
    destMat[9] = 0.0f - vLightPos[1] * vPlaneEqua-
tion[2];
    destMat[13] = 0.0f - vLightPos[1] * vPlaneEqua-
tion[3];

```

```

        // Third Column
        destMat[2] = 0.0f - vLightPos[2] * vPlaneEqua-
tion[0];
        destMat[6] = 0.0f - vLightPos[2] * vPlaneEqua-
tion[1];
        destMat[10] = dot - vLightPos[2] * vPlaneEqua-
tion[2];
        destMat[14] = 0.0f - vLightPos[2] * vPlaneEqua-
tion[3];

        // Fourth Column
        destMat[3] = 0.0f - vLightPos[3] * vPlaneEqua-
tion[0];
        destMat[7] = 0.0f - vLightPos[3] * vPlaneEqua-
tion[1];
        destMat[11] = 0.0f - vLightPos[3] * vPlaneEqua-
tion[2];
        destMat[15] = dot - vLightPos[3] * vPlaneEqua-
tion[3];
    }

////////////////////////////////////
////////////////////////////////////
// Load a matrix with the Identity matrix
void gltLoadIdentityMatrix(GLTMatrix m)
{
    static GLTMatrix identity = { 1.0f, 0.0f, 0.0f,
0.0f,
                                0.0f, 1.0f,
0.0f, 0.0f,
                                0.0f, 0.0f,
1.0f, 0.0f,
                                0.0f, 0.0f,
0.0f, 1.0f };
};

```

```

        memcpy(m, identity, sizeof(GLTMatrix));
    }
    //////////////////////////////////////
    //////////////////////////////////////
    // Creates a 4x4 rotation matrix, takes radians NOT
    degrees
    void gltRotationMatrix(float angle, float x, float y,
    float z, GLTMatrix mMatrix)
    {
        float vecLength, sinSave, cosSave, oneMinusCos;
        float xx, yy, zz, xy, yz, zx, xs, ys, zs;

        // If NULL vector passed in, this will blow up...
        if(x == 0.0f && y == 0.0f && z == 0.0f)
        {
            gltLoadIdentityMatrix(mMatrix);
            return;
        }

        // Scale vector
        vecLength = (float)sqrt( x*x + y*y + z*z );

        // Rotation matrix is normalized
        x /= vecLength;
        y /= vecLength;
        z /= vecLength;

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

        xx = x * x;
        yy = y * y;
        zz = z * z;
        xy = x * y;
        yz = y * z;

```

```

zx = z * x;
xs = x * sinSave;
ys = y * sinSave;
zs = z * sinSave;

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

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

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

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

////////////////////////////////////
////
// Rotate a frame around it's local Y axis
void gltRotateFrameLocalY(GLTFrame *pFrame, GLfloat
fAngle)
{
    GLTMatrix mRotation;
    GLTVector3 vNewForward;

```

```

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

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

//////////////////////////////////////
/////
// Rotate a frame around it's local X axis
void gltRotateFrameLocalX(GLTFrame *pFrame, GLfloat
fAngle)
{
    GLTMatrix mRotation;
    GLTVector3 vCross;

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

    GLTVector3 vNewVect;
    // Inline 3x3 matrix multiply for rotation only
    vNewVect[0] = mRotation[0] * pFrame->vForward[0]
+ mRotation[4] * pFrame->vForward[1] + mRotation[8] *
pFrame->vForward[2];
    vNewVect[1] = mRotation[1] * pFrame->vForward[0]
+ mRotation[5] * pFrame->vForward[1] + mRotation[9] *
pFrame->vForward[2];
    vNewVect[2] = mRotation[2] * pFrame->vForward[0]
+ mRotation[6] * pFrame->vForward[1] + mRotation[10]
* pFrame->vForward[2];

```

```

        memcpy(pFrame->vForward, vNewVect, sizeof(GLfloat)*3);

        // Update pointing up vector
        vNewVect[0] = mRotation[0] * pFrame->vUp[0] +
mRotation[4] * pFrame->vUp[1] + mRotation[8] *
pFrame->vUp[2];
        vNewVect[1] = mRotation[1] * pFrame->vUp[0] +
mRotation[5] * pFrame->vUp[1] + mRotation[9] *
pFrame->vUp[2];
        vNewVect[2] = mRotation[2] * pFrame->vUp[0] +
mRotation[6] * pFrame->vUp[1] + mRotation[10] *
pFrame->vUp[2];
        memcpy(pFrame->vUp, vNewVect, sizeof(GLfloat) *
3);
    }

//////////////////////////////////////
//////////
// Rotate a frame around it's local Z axis
void gltRotateFrameLocalZ(GLTFrame *pFrame, GLfloat
fAngle)
{
    GLTMatrix mRotation;

    // Only the up vector needs to be rotated
    gltRotationMatrix(fAngle, pFrame->vForward[0],
pFrame->vForward[1], pFrame->vForward[2], mRotation);

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

```

```

        vNewVect[2] = mRotation[2] * pFrame->vUp[0] +
mRotation[6] * pFrame->vUp[1] + mRotation[10] *
pFrame->vUp[2];
        memcpy(pFrame->vUp, vNewVect, sizeof(GLfloat) *
3);
    }

// Light and material Data
GLfloat fLightPos[4]    = { -100.0f, 100.0f, 50.0f,
1.0f }; // Point source
GLfloat fNoLight[] = { 0.0f, 0.0f, 0.0f, 0.0f };
GLfloat fLowLight[] = { 0.25f, 0.25f, 0.25f, 1.0f };
GLfloat fBrightLight[] = { 1.0f, 1.0f, 1.0f, 1.0f };

GLTMatrix mShadowMatrix;

#define GROUND_TEXTURE  0
#define TORUS_TEXTURE   1
#define SPHERE_TEXTURE  2
#define NUM_TEXTURES    3
GLuint  textureObjects[NUM_TEXTURES];

const char *szTextureFiles[] = {"grass.tga",
"wood.tga", "orb.tga"};

////////////////////////////////////
////////////////////////////////////
// This function does any needed initialization on
the rendering
// context.
void SetupRC()
{
    GLTVector3 vPoints[3] = {{ 0.0f, -0.4f, 0.0f },
                             { 10.0f, -0.4f, 0.0f },
                             { 5.0f, -0.4f, -5.0f }};

```



```

int iSphere;
int i;

// Grayish background
glClearColor(fLowLight[0], fLowLight[1], fLowLight[2], fLowLight[3]);

// Clear stencil buffer with zero, increment by one whenever anybody
// draws into it. When stencil function is enabled, only write where
// stencil value is zero. This prevents the transparent shadow from drawing
// over itself
glStencilOp(GL_INCR, GL_INCR, GL_INCR);
glClearStencil(0);
glStencilFunc(GL_EQUAL, 0x0, 0x01);

// Cull backs of polygons
glCullFace(GL_BACK);
glFrontFace(GL_CCW);
glEnable(GL_CULL_FACE);
glEnable(GL_DEPTH_TEST);
glEnable(GL_MULTISAMPLE_ARB);

// Setup light parameters
glLightModelfv(GL_LIGHT_MODEL_AMBIENT, fNoLight);
// glLightModeli(GL_LIGHT_MODEL_COLOR_CONTROL, GL_SEPARATE_SPECULAR_COLOR);
glLightfv(GL_LIGHT0, GL_AMBIENT, fLowLight);
glLightfv(GL_LIGHT0, GL_DIFFUSE, fBrightLight);
glLightfv(GL_LIGHT0, GL_SPECULAR, fBrightLight);
glEnable(GL_LIGHTING);
glEnable(GL_LIGHT0);

// Calculate shadow matrix

```

```

    gltMakeShadowMatrix(vPoints, fLightPos, mShadow-
Matrix);

    // Mostly use material tracking
    glEnable(GL_COLOR_MATERIAL);
    glColorMaterial(GL_FRONT,
GL_AMBIENT_AND_DIFFUSE);
    glMateriali(GL_FRONT, GL_SHININESS, 128);

    gltInitFrame(&frameCamera); // Initialize the
camera

    // Randomly place the sphere inhabitants
    for(iSphere = 0; iSphere < NUM_SPHERES; iS-
phere++)
    {
        gltInitFrame(&spheres[iSphere]); // In-
italize the frame

        // Pick a random location between -20 and 20
at .1 increments
        spheres[iSphere].vLocation[0] =
(float)((rand() % 400) - 200) * 0.1f;
        spheres[iSphere].vLocation[1] = 0.0f;
        spheres[iSphere].vLocation[2] =
(float)((rand() % 400) - 200) * 0.1f;
    }

    // Set up texture maps
    glEnable(GL_TEXTURE_2D);
    glGenTextures(NUM_TEXTURES, textureObjects);
    glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE,
GL_MODULATE);

    for(i = 0; i < NUM_TEXTURES; i++)

```

```

    {
        GLubyte *pBytes;
        GLint iWidth, iHeight, iComponents;
        GLenum eFormat;

        glBindTexture(GL_TEXTURE_2D, textureOb-
jects[i]);

        // Load this texture map
        pBytes = (GLubyte
*)glLoadTGA(szTextureFiles[i], &iWidth, &iHeight,
&iComponents, &eFormat);
        gluBuild2DMipmaps(GL_TEXTURE_2D, iComponents,
iWidth, iHeight, eFormat, GL_UNSIGNED_BYTE, pBytes);
        free(pBytes);

        glTexParameteri(GL_TEXTURE_2D,
GL_TEXTURE_MAG_FILTER, GL_LINEAR);
        glTexParameteri(GL_TEXTURE_2D,
GL_TEXTURE_MIN_FILTER, GL_LINEAR_MIPMAP_LINEAR);
        glTexParameteri(GL_TEXTURE_2D,
GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
        glTexParameteri(GL_TEXTURE_2D,
GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
    }

}

////////////////////////////////////
////////////////////////////////////
// Do shutdown for the rendering context
void ShutdownRC(void)
{
    // Delete the textures
    glDeleteTextures(NUM_TEXTURES, textureObjects);
}

```

```

// For best results, put this in a display list
// Draw a torus (doughnut) at z = fZVal... torus is
in xy plane
void gltDrawTorus(GLfloat majorRadius, GLfloat minor-
Radius, GLint numMajor, GLint numMinor)
{
    GLTVector3 vNormal;
    double majorStep = 2.0f*GLT_PI / numMajor;
    double minorStep = 2.0f*GLT_PI / numMinor;
    int i, j;

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

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

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

            // First point

```

```

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

        glTex-
Coord2f((float) (i+1)/(float) (numMajor),
(float) (j)/(float) (numMinor));
        vNormal[0] = x1*c;
        vNormal[1] = y1*c;
        vNormal[2] = z/minorRadius;
        glNormal3fv(vNormal);
        glVertex3f(x1*r, y1*r, z);
    }
    glEnd();
}

}

////////////////////////////////////
/////
// Draw the ground as a series of triangle strips
void DrawGround(void)
{
    GLfloat fExtent = 20.0f;
    GLfloat fStep = 1.0f;
    GLfloat y = -0.4f;
    GLint iStrip, iRun;
    GLfloat s = 0.0f;
    GLfloat t = 0.0f;
    GLfloat texStep = 1.0f / (fExtent * .075f);

```

```

        glBindTexture(GL_TEXTURE_2D, textureOb-
jects[GROUND_TEXTURE]);
        glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S,
GL_REPEAT);
        glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T,
GL_REPEAT);

        for(iStrip = -fExtent; iStrip <= fExtent; iStrip
+= fStep)
        {
            t = 0.0f;
            glBegin(GL_TRIANGLE_STRIP);

                for(iRun = fExtent; iRun >= -fExtent;
iRun -= fStep)
                {
                    glTexCoord2f(s, t);
                    glNormal3f(0.0f, 1.0f, 0.0f);    //
All Point up
                    glVertex3f(iStrip, y, iRun);

                    glTexCoord2f(s + texStep, t);
                    glNormal3f(0.0f, 1.0f, 0.0f);    //
All Point up
                    glVertex3f(iStrip + fStep, y, iRun);

                    t += texStep;
                }
            glEnd();
            s += texStep;
        }
    }

////////////////////////////////////
////////////////////////////////////

```

```

// Draw random inhabitants and the rotating to-
rus/sphere duo
void DrawInhabitants(GLint nShadow)
{
    static GLfloat yRot = 0.0f;          // Rotation
angle for animation
    GLint i;

    if(nShadow == 0)
    {
        yRot += 0.5f;
        glColor4f(1.0f, 1.0f, 1.0f, 1.0f);
    }
    else
        glColor4f(1.0f, 1.0f, 1.0f, .10f); // Shadow
color

    // Draw the randomly located spheres
    glBindTexture(GL_TEXTURE_2D, textureOb-
jects[SPHERE_TEXTURE]);
    for(i = 0; i < NUM_SPHERES; i++)
    {
        glPushMatrix();
        gltApplyActorTransform(&spheres[i]);
        gltDrawSphere(0.3f, 21, 11);
        glPopMatrix();
    }

    glPushMatrix();
    glTranslatef(0.0f, 0.1f, -2.5f);

    glPushMatrix();
        glRotatef(-yRot * 2.0f, 0.0f, 1.0f,
0.0f);
        glTranslatef(1.0f, 0.0f, 0.0f);

```

```

        gltDrawSphere(0.1f, 21, 11);
glPopMatrix();

if(nShadow == 0)
{
    // Torus alone will be specular
    glMaterialfv(GL_FRONT, GL_SPECULAR,
fBrightLight);
}

    glRotatef(yRot, 0.0f, 1.0f, 0.0f);
    glBindTexture(GL_TEXTURE_2D, textureOb-
jects[TORUS_TEXTURE]);
    gltDrawTorus(0.35, 0.15, 61, 37);
    glMaterialfv(GL_FRONT, GL_SPECULAR, fNo-
Light);
    glPopMatrix();
}

// Called to draw scene
void RenderScene(void)
{
    // Clear the window with current clearing color
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT
| GL_STENCIL_BUFFER_BIT);

    glPushMatrix();
        gltApplyCameraTransform(&frameCamera);

        // Position light before any other transfor-
mations
        glLightfv(GL_LIGHT0, GL_POSITION, fLightPos);

        // Draw the ground
        glColor3f(1.0f, 1.0f, 1.0f);

```



```

    DrawGround();

    // Draw shadows first
    glDisable(GL_DEPTH_TEST);
    glDisable(GL_LIGHTING);
    glDisable(GL_TEXTURE_2D);
    glEnable(GL_BLEND);
    glBlendFunc(GL_SRC_ALPHA,
GL_ONE_MINUS_SRC_ALPHA);
    glEnable(GL_STENCIL_TEST);
    glPushMatrix();
        glMultMatrixf(mShadowMatrix);
        DrawInhabitants(1);
    glPopMatrix();
    glDisable(GL_STENCIL_TEST);
    glDisable(GL_BLEND);
    glEnable(GL_LIGHTING);
    glEnable(GL_TEXTURE_2D);
    glEnable(GL_DEPTH_TEST);

    // Draw inhabitants normally
    DrawInhabitants(0);

    glPopMatrix();

    // Do the buffer Swap
    glutSwapBuffers();
}

// Respond to arrow keys by moving the camera frame
of reference
void SpecialKeys(int key, int x, int y)
{
    if(key == GLUT_KEY_UP)

```

```

        gltMoveFrameForward(&frameCamera, 0.1f);

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

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

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

    // Refresh the Window
    glutPostRedisplay();
}

////////////////////////////////////
/////
// Called by GLUT library when idle (window not being
// resized or moved)
void TimerFunction(int value)
{
    // Redraw the scene with new coordinates
    glutPostRedisplay();
    glutTimerFunc(3,TimerFunction, 1);
}

void ChangeSize(int w, int h)
{
    GLfloat fAspect;

    // Prevent a divide by zero, when window is too
    short
    // (you cant make a window of zero width).
    if(h == 0)
        h = 1;

```

```

glViewport(0, 0, w, h);

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

// Reset the coordinate system before modifying
glMatrixMode(GL_PROJECTION);
glLoadIdentity();

// Set the clipping volume
gluPerspective(35.0f, fAspect, 1.0f, 50.0f);

glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
}

int main(int argc, char* argv[])
{
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB |
GLUT_DEPTH | GLUT_STENCIL);
    glutInitWindowSize(800,600);
    glutCreateWindow("OpenGL SphereWorld Demo + Tex-
ture Maps");
    glutReshapeFunc(ChangeSize);
    glutDisplayFunc(RenderScene);
    glutSpecialFunc(SpecialKeys);

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

    glutMainLoop();

    ShutdownRC();
    return 0;
}

```

## ЗАДАНИЕ НА ЛАБОРАТОРНУЮ РАБОТУ

Воспроизвести результаты, представленные в теоретическом обзоре, освоить наложение текстур и работу с множеством текстурных объектов, согласно варианту, полученному у преподавателя, наложить текстуры на объекты сцены.

### ТРЕБОВАНИЯ К РЕАЛИЗАЦИИ

Задание выполняется согласно варианту. По завершении готовится отчет.

- 1) Для [Листинга 1](#) на заданный в варианте геометрический объект наложить произвольную текстуру из файла в формате tga.
- 2) Используя [Листинг 2](#), для геометрических объектов реализовать вращение и показать действие одномерной текстуры. Цвет меняется плавно от темного до светлого оттенка. Количество градаций цвета не менее 12.
- 3) Используя [Листинг 3](#) создать коридор по форме указанной в варианте. Для каждой грани использовать свою собственную текстуру т.е. отдельный файл в формате tga..
- 4) Для [Листинга 4](#), используя листинг 6 (листинг 7) из лабораторной работы №3, для исходного тора, сферы и поверхности реализовать наложение текстуры с зеркальными бликами (текстуры указаны в таблицах). Для своих собственных объектов добавить произвольные текстуры из набора: капли на стекле, орнамент, арбуз, каменная стена, кирпичная стена, асфальт, треснутая земля, листва зеленая. реализовать следующие изменения согласно варианту.

Задание 4 обязательно только для студентов, претендующих на оценку «отлично».

### ВАРИАНТЫ ЗАДАНИЙ

Задание 1

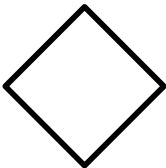
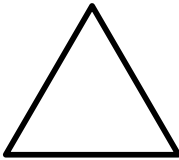
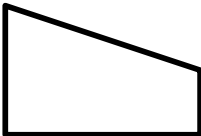

- 1) Сфера

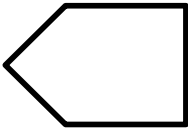
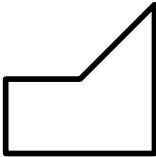
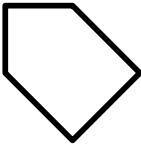



- 2) Тор
- 3) Куб
- 4) Цилиндр
- 5) Чайник Юта
- 6) Призма четырехгранная (не куб)
- 7) Параллелепипед
- 8) Тетраэдр
- 9) Конус
- 10) Додекаэдр

### Задание 2

- 1) Синий цвет, две совмещенные сферы т.е. частично пересекаются
- 2) Зеленый цвет, цилиндр
- 3) Желтый цвет, тетраэдр
- 4) Красный цвет, три цилиндра образуют букву «Н»
- 5) Коричневый цвет, цилиндр и большая сфера
- 6) Серый цвет, цилиндры соединенные в треугольник
- 7) Случайный цвет, три цилиндра образуют букву «П»
- 8) Плавные переходы по цветам радуги, чайник Юта
- 9) Фиолетовый цвет, скрещенный цилиндры, знак «+»
- 10) Розовый цвет, икосаэдр

### Задание 3

<p>1.</p> 	<p>6.</p> 
<p>2.</p> 	<p>7.</p> 

3. 	8. 
4. 	9. 
5. 	10. 

#### Задание 4

Вариант	Тор	Сфера	Поверхность
1	камень	листва	трава
2	капли воды	песок	мех
3	песок	мрамор	лунный пейзаж
4	ткань	трава	текст газеты
5	кожа	треснутое стекло	огонь
6	снег	зерно	дерево

## КОНТРОЛЬНЫЕ ВОПРОСЫ И ЗАДАНИЯ

1. Сформулируйте понятие текселя и его характеристики.
2. Классифицируйте функции для загрузки текстур и параметры для их работы.
3. Оцените, какая операция является узким местом при работе с текстурами.
4. Приведите функцию, используемую для отображения текстуры на геометрические объекты.
5. Дайте определение фильтрации и поясните её роль.
6. Раскройте роль механизма намотки.
7. Изложите назначение множественной (сокращенной) текстуры.
8. Покажите, по каким осям происходит фильтрация множественной текстуры.
9. Опишите работу функции генерации уровней множественной текстуры.
10. Как можно управлять используемым уровнем детализации при работе с множественной текстурой?
11. Раскройте значение термина текстурный объект.
12. Раскройте термин *келевое затенение* и его связь с эффектом мультипликации.
13. Приведите алгоритм применения к объекту с наложенной текстурой эффектов освещения.
14. Классифицируйте дополнительные эффекты освещения, описанные в работе.
15. Покажите методологию работы с несколькими текстурами на одной сцене.

## ФОРМА ОТЧЕТА ПО ЛАБОРАТОРНОЙ РАБОТЕ

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

Номер варианта студента назначается индивидуально преподавателем.

В отчете должны быть представлены:

- 1) Текст задания для лабораторной работы и номер варианта.
- 2) В отчете должны быть представлены все полные 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. Выполнение задания должно сопровождаться снимками экрана.

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

Отчет на защиту предоставляется в печатном виде.

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



## ОСНОВНАЯ ЛИТЕРАТУРА

1. Боресков А.В. Основы работы с технологией CUDA / А.В. Боресков, А.А. Харламов - Издательство "ДМК Пресс", 2010. - 232 с. - ISBN 978-5-94074-578-5; ЭБС «Лань». - URL: [https://e.lanbook.com/book/1260#book\\_name](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](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](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://habrahabr.ru/post/315294/> - Текстуры в OpenGL
2. <https://open.gl/textures> - Текстурные объекты и их параметры OpenGL
3. [http://opengl-master.ru/view\\_post.php?id=73](http://opengl-master.ru/view_post.php?id=73) – Параметры текстуры в OpenGL
4. [http://www.codenet.ru/progr/opengl/opengl\\_05.php](http://www.codenet.ru/progr/opengl/opengl_05.php) – Наложение текстуры на тор в OpenGL