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

ОСВЕЩЕНИЕ В OPENGL

Методические указания к лабораторной работе
по дисциплине «Компьютерная графика»

Калуга, 2018

УДК 004.62
ББК 32.972.5
Б435

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

Методические указания рассмотрены и одобрены:

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

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

- Методической комиссией факультета ФНК протокол №_2_ от «_28_» _02_____ 2018 г.

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

- Методической комиссией
КФ МГТУ им.Н.Э. Баумана протокол №_2_ от «_06_» _03_____ 2018 г.

Председатель методической комиссии
КФ МГТУ им.Н.Э. Баумана

_____д.э.н., профессор О.Л. Перерва

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

Авторы
к.ф.-м.н., доцент кафедры ФН1-КФ _____ Ю.С. Белов
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 материалах и освещении в программном интерфейсе OpenGL. В методических указаниях приводятся теоретические сведения о цветовых моделях, распространеии света и способах их описания, реализуемых в OpenGL. Рассмотрен процесс смешения цветов, установки источников света и создания эффектов тени и тумана с использование машины состояний OpenGL.

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

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

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

Методические указания, ориентированные на студентов 2-го курса бакалавриата направления подготовки 09.03.04 «Программная инженерия», содержат краткую теоретическую часть, описывающую способы представления материалов и применения освещения с помощью машины состояний программного интерфейса OpenGL, поэтапные примеры создания сцен и отдельных объектов с различными параметрами освещения, комментарии и пояснения по вышеназванным этапам, а также задание на лабораторную работу. Дополнительно рассмотрены способы сглаживания используемые в OpenGL.

Методические указания составлены в расчете на начальное ознакомление студентов с основами работы с программным интерфейсом OpenGL. Для выполнения лабораторной работы студенту необходимо уметь ориентироваться в описании цветов и источников света в OpenGL, уметь применять эффект тумана с различными параметрами, владеть способами задания свойств материалов моделей, а также сглаживания границ растеризуемых графических примитивов.

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 освещением средствами OpenGL, а также созданию настройки свойств материалов объектов и применения эффектов тени и тумана для большей реалистичности изображений.

Основными задачами выполнения лабораторной работы являются: знать что такое затенение и особенности его применения к объектам различной геометрии; уметь работать с различными типами освещения и согласовывать свет с характеристиками материала; уметь устанавливать характеристики источников освещения, а также создавать блики и прожектора; понимать принципы формирования тени объекта. Уметь создавать тень на различных поверхностях; уметь реализовывать эффект отражения; понимать принципы формирования тумана и знать основные характеристики; понимать принципы сглаживания и уметь задавать необходимые в контексте каждой задачи параметры для выполнения эффекта сглаживания.

Результатами работы являются:

- Выполненная настройка свойств материалов средствами OpenGL
- Реализованные согласно варианту освещенные сцены
- Применение эффекта сглаживания к ранее описанным примитивам
- Подготовленный отчет

ХАРАКТЕРИСТИКА ОБЪЕКТА ИЗУЧЕНИЯ, ИССЛЕДОВАНИЯ

Разрешение экрана

Разрешение экранов современных компьютеров может меняться от 640x480 пикселей до 1600x1200 или даже больше. Задавая объем отсечения и поле просмотра, всегда нужно учитывать размер окна. Масштабируя размер рисунка до размера окна, легко учесть комбинации различных разрешений и размеров окна, которые могут встретиться. Качественно написанные графические приложения всегда отображают примерно одинаковое изображение, независимо от разрешения экрана. При увеличении разрешения изображение на экране должно перестраиваться так, чтобы пользователь мог более четко видеть большее число деталей.

Насыщенность цвета

Если увеличение разрешения экрана или числа доступных для рисования пикселей увеличивает детализацию и резкость изображения, качество получаемого изображения должно повышаться и при увеличении числа доступных цветов. Изображение, показанное на компьютере, способном отображать миллионы цветов, должно выглядеть значительно лучше, чем тот же рисунок, показанный всего с 16 цветами. В программировании вам, очевидно, придется учитывать только три степени насыщенности цвета: 4 бит, 8 бит и 24 бит.

4-битовый режим цвета

На слабых машинах программа должна запускаться при 16 цветах — это называется *4-битовый режим,* поскольку информация о цвете каждого пикселя записывается с использованием 4 бит. Эти 4 бит представляют значение от 0 до 15 — индекс в наборе из 16 предопределенных цветов. Когда используется ограниченный набор цветов, доступ к которым осуществляется с помощью индекса, этот набор называется *палитрой.*

8-битовый режим цвета

8-битовый режим цвета поддерживает на экране до 256 цветов. Это существенное улучшение по сравнению с 4-битовым цветом, хотя такой насыщенности цвета все же недостаточно для серьезной работы. Большинство аппаратных ускорителей OpenGL не ускоряет 8-битовый цвет, но при программной визуализации, при определенных условиях, можно получить в Windows удовлетворительные результаты. В этом случае самые важные моменты касаются построения правильной палитры цветов.

24-битовый режим цвета

Наилучшее качество изображений, доступное сейчас на ПК, дает 24-битовый режим цвета. В этом режиме каждому пикселю сопоставляется 24 бит, из которых по 8 бит выделено для хранения информации об одном из составляющих цветов (красный, зеленый и синий). Любой пиксель экрана можно раскрасить одним из 16 миллионов возможных цветов. Наиболее очевидным недостатком этого режима является объем памяти, требуемой для записи информации об экранах с высоким разрешением (более 2 Мбайт при экране 1024x768). Кроме того, перемещение больших участков памяти также происходит гораздо медленнее, например, когда применяется анимация или просто рисуется на экране. К счастью, современные графические адаптеры с ускорителями оптимизированы под операции такого типа и в них вмонтирована большая память, предназначенная для удовлетворения потребностей в дополнительной памяти.

16- и 32-битовые режимы цвета

Чтобы сэкономить память или улучшить производительность, многие графические карты также поддерживают другие режимы цвета. В контексте улучшения производительности был разработан 32-битовый режим цвета, иногда называемый *реалистичным цветовоспроизведением* (true color). В действительности при 32-битовом режиме отображения возможен показ такого же числа цветов, что и при 24-битовом

режиме, но этот режим более выгоден с точки зрения производительности, поскольку каждому пикселю сопоставляется 32-битовый адрес. К сожалению, при этом не используется 8 бит (1 байт) на каждый пиксель. На современных 32-битовых ПК Intel кратность памяти 32 обеспечивает более быстрый доступ к памяти. Современные ускорители OpenGL также поддерживают 32-битовый режим, где 24 бит зарезервированы для записи RGB-цветов, а 8 бит — для хранения значения альфа.

Другой популярный режим отображения (16-битовый цвет) иногда применяется для более эффективного использования памяти. Это позволяет выбирать для раскрашивания пикселя один из 65 536 возможных цветов.

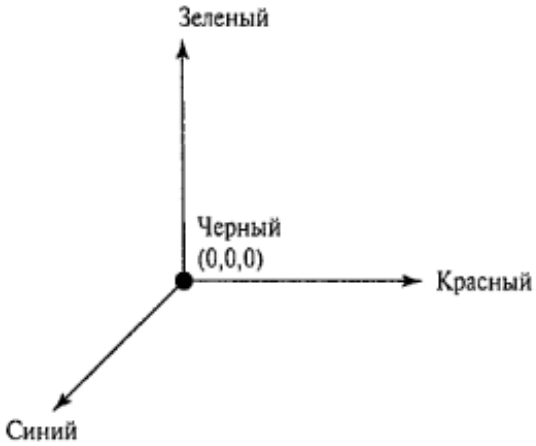

Рис. 1 – Начало координат пространства цветов RGB

Использование цвета в OpenGL

Теперь известно, как OpenGL задает точный цвет через интенсивности красного, зеленого и синего компонентов. Кроме того, известно, что современное аппаратное обеспечение персональных компьютеров может отображать все, почти все возможные комбинации или лишь несколько из них. Таким образом, возникает вопрос: как задать цвет через красный, зеленый и синий компоненты?

Куб цвета

Поскольку цвет задается тремя положительными кодами цвета, доступные цвета можно смоделировать объемом, который называют *пространством цветов RGB.* Данное пространство показано на рис. 1, где обозначено начало координат, и на осях отмечены красный, синий и зеленый цвета. Координаты в этом пространстве задаются так же, как координаты ***x, y*** и ***z.*** В начале координат (0,0,0) относительные интенсивности всех компонентов равны нулю, поэтому получается черный цвет.

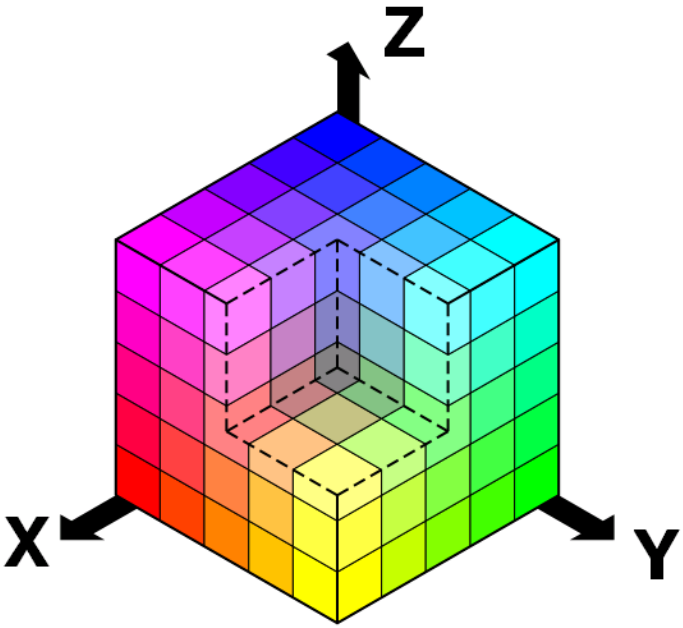

Рис. 2 –Пространство цветов RGB

Для хранения информации о цветах на ПК допускается использование максимум 24 бит (по 8 бит на каждый компонент), поэтому будем считать, что значение 255 (максимальное число, которое можно записать с помощью 8 бит) представляет максимально насыщенный компонент. Таким образом, получаем куб со сторонами 255 единиц. Вершина куба, противоположная началу координат, которое соответ-

ствует черному цвету и представляется концентрациями (0,0,0), соответствует белому цвету и характеризуется относительными концентрациями (255, 255, 255). Точки максимальной насыщенности (255), соответствующие вершинам куба, расположенным на осях, представляют чистые красный, зеленый и синий цвета.

Такой "куб цвета" (рис. 2) содержит все возможные цвета либо на поверхности, либо внутри куба. Например, возможные оттенки серого (между черным и белым) располагаются на диагонали, соединяющей вершины (0,0,0) и (255,255,255).

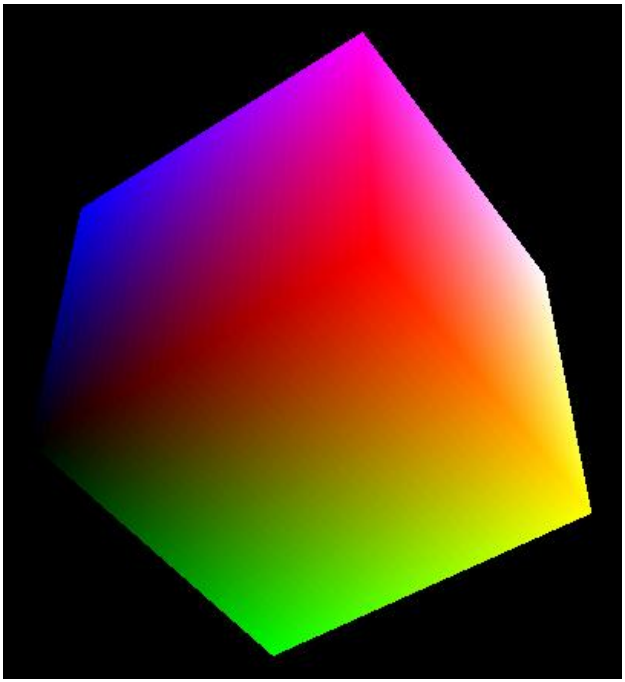

Рис. 3 – Куб цветов

На рис. 3 показан куб цвета. На поверхности этого куба представлены различные цвета, плавно связывающие черный цвет одной вершины с белым цветом противоположной. Красный, синий и зеленый цвета располагаются в соответствующих вершинах, расположенных на расстоянии 255 единиц от начала координат. Кроме того, обозначены желтый, голубой и пурпурный цвета, являющиеся комбинациями

двух из трех основных цветов. Нажимая клавиши со стрелками, можено вращать куб, изучая его со всех сторон.

Задание цвета рисования

Кратко напомним основную информацию о функции `glColor`. Ее прототип можно записать следующим образом.

```
void glColor<x><t>(red, green, blue, alpha);
```

В имени функции <x> представляет число аргументов; это может быть 3 - красный, зеленый и синий или 4 - те же плюс компонент альфа. Компонент альфа задает прозрачность цвета. В дальнейшем будет использоваться вариант функции с тремя аргументами.

Параметр <t> в имени функции задает тип данных аргумента и может иметь одно из следующих значений: `b`, `d`, `f`, `i`, `s`, `ub`, `ui` или `us` от типов данных byte, double, float, integer, short, unsigned byte, unsigned integer и unsigned short соответственно. В другой версии функции к концу названия добавляется буква `v`; данная версия принимает в качестве аргумента массив, содержащий аргументы (`v` — сокращение от "vector").

В большинстве программ OpenGL, с которыми придется столкнуться, используется функция `glColor3f`, и интенсивность всех компонентов задается как величина, принадлежащая диапазону от 0.0 (нулевая интенсивность) до 1.0 (максимальная интенсивность). Если у вас есть опыт программирования в Windows, легче использовать вариант `glColor3ub` этой функции. В качестве аргументов эта версия принимает три байта без знака, представляющие числа от 0 до 255 и задающие интенсивности красного, зеленого и синего компонентов. Использование этой версии функции подобно применению следующего макроса Windows под названием RGB.

```
glColor3ub(0,255,128)  = RGB(0,255,128)
```

Описанный подход может облегчить согласование цветов OpenGL с существующими цветами RGB, которые привычно применять в программах, не использующих OpenGL. Тем не менее, следует отметить,

что внутренне OpenGL представляет коды цвета как величины с плавающей запятой, и из-за необходимости преобразования констант в форму внутреннего представления возможна небольшая потеря производительности. Также может случиться, что в будущем возникнут буферы цвета с большим разрешением (фактически, буферы цвета, в которые заносятся величины с плавающей запятой, уже появились), поэтому коды цвета, заданные через величины с плавающей запятой, могут оказаться более удобными для аппаратного обеспечения.

Цвет в реальном мире

Реальные объекты не выглядят как раскрашенные сплошными или плавно переходящими друг в друга красками, основанными на четко заданных RGB-кодах. Рассмотрим, например, рис.4, где показан результат выполнения программы JET. Это простой самолет, нарисованный вручную с помощью треугольников, причем при рисовании использовались только методы, рассмотренные выше.

Цвета треугольников выбирались так, чтобы подчеркнуть трехмерную структуру самолета. Тем не менее, набор треугольников очень мало походит на что-либо, наблюдаемое в реальной жизни. Предположим, что построена модель этого самолета и раскрасили все плоские поверхности указанными цветами. Модель должна выглядеть глянцевой или матовой в зависимости от типа использованной краски, а цвет всех плоских поверхностей будет меняться в зависимости от угла наблюдения и положения источников света.

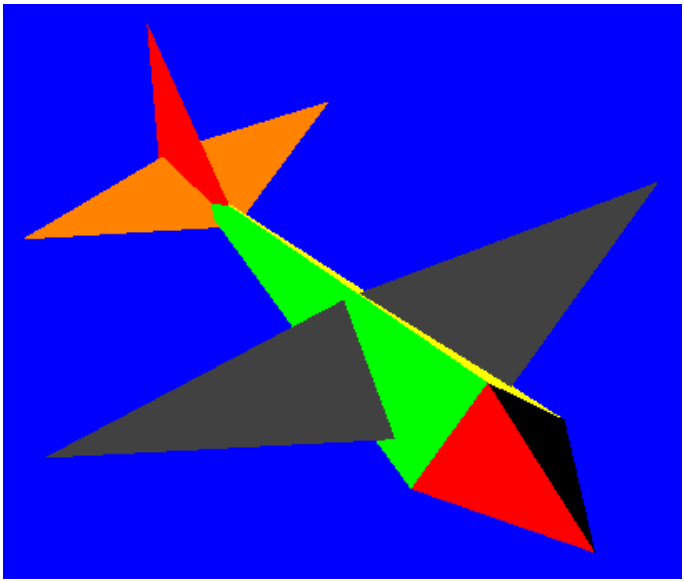

Рис. 4 – Простой самолет, построенный путем присвоения треугольникам различных цветов

К счастью, OpenGL выполняет аппроксимацию предметов реального мира с учетом условий освещения. Если объект сам не излучает света, он освещается светом трех различных категорий: рассеянным (светом окружающей среды), диффузным и отраженным.

Рассеянный свет

Рассеянный свет не поступает из какого-либо определенного направления (поэтому его еще называют светом окружающей среды). Он имеет источник, но лучи его света отражаются от комнаты или сцены и становятся ненаправленными. Объекты, освещенные рассеянным светом, равномерно окрашивают со всех сторон и во всех направлениях. Во всех примерах, приводившихся до этого момента, можно сказать, что представленные объекты освещаются ярким рассеянным светом, поскольку объекты были всегда видимы и равномерно раскрашены (или затенены) вне зависимости от угла поворота или наблюдения. Объект, освещенный рассеянным светом, показан на рис. 5.

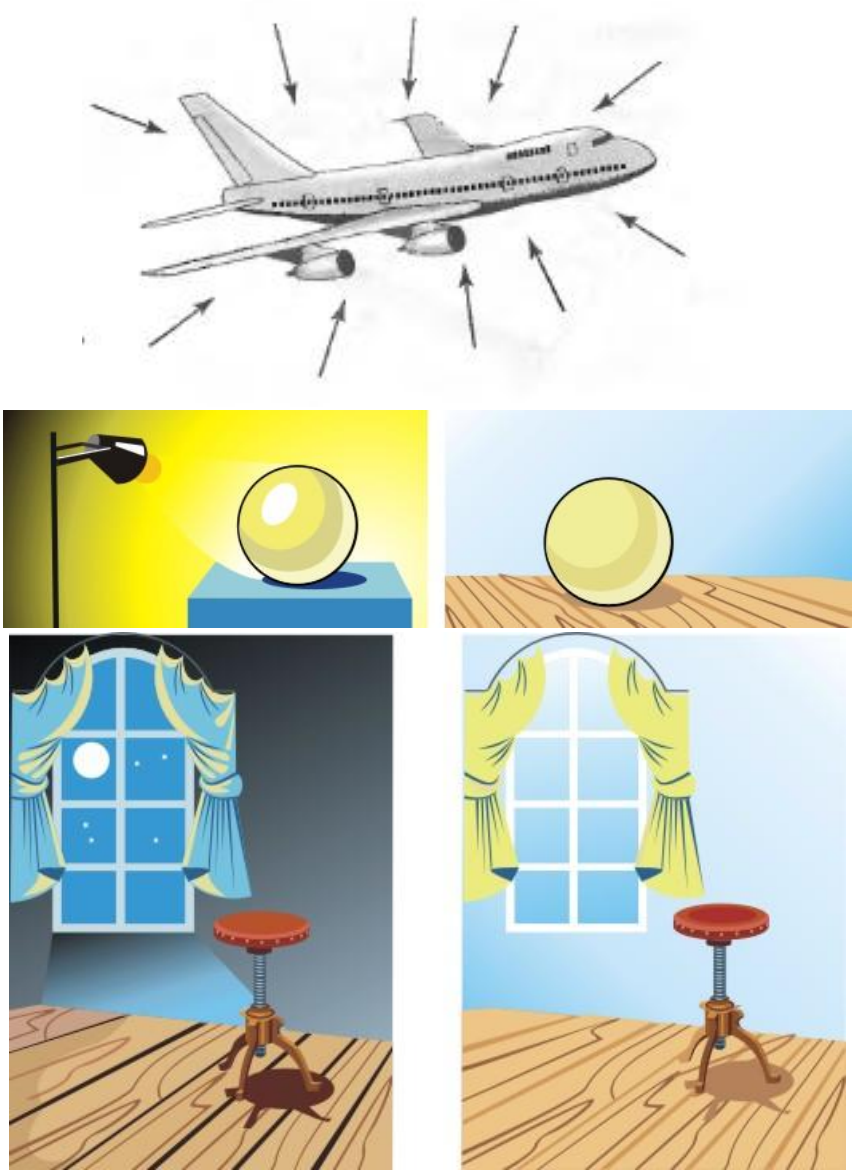

Рис. 5 – Объект, освещенный исключительно рассеянным светом

Диффузный свет

Диффузный свет поступает из определенного направления, но он равномерно отражается от поверхности.

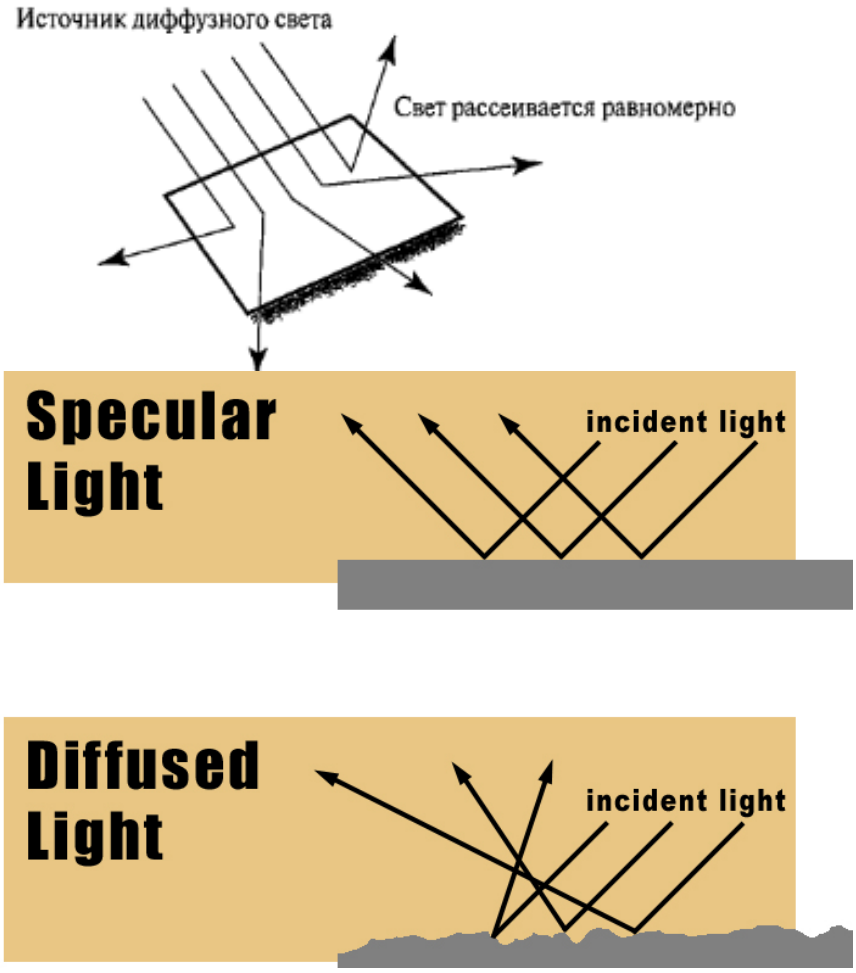

Рис. 6 – Объект, освещенный исключительно источником диффузного света

Даже при том, что свет отражается равномерно, поверхность объекта выглядит ярче, если свет расположен прямо на поверхности, чем если он падает на поверхность под углом. На рис. 6 показан объект, освещенный источником диффузного света.

Отраженный свет

Подобно диффузному свету, приходящему из определенного направления, но равномерно рассеиваемому во всех направлениях, отраженный свет является направленным, но он отражается в ограниченном направлении. Сильный отраженный свет обычно дает яркое пятно на поверхности, именуемое *бликом.* Примерами источников отражаемого света являются прожектор и Солнце. Пример объекта, освещенного только источником отражаемого света, приводится на рис. 7.

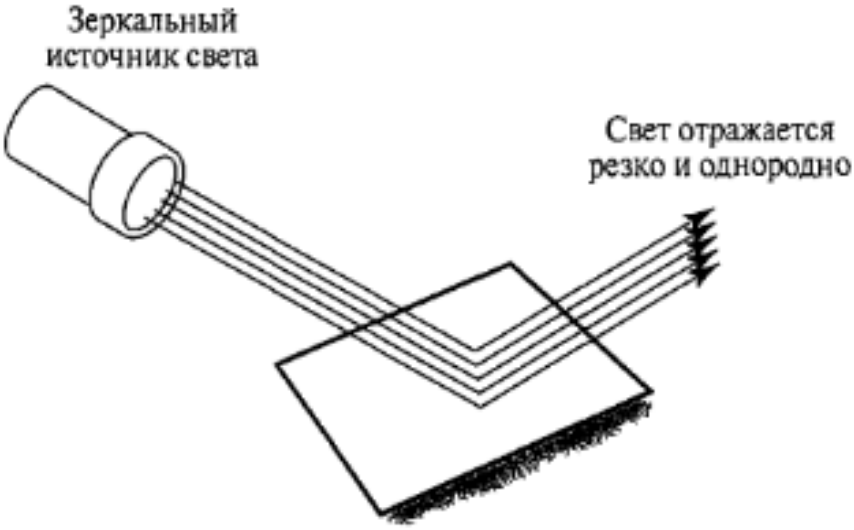

Рис. 7 – Объект, освещенный исключительно источником отражаемого света

Ни один реальный источник света не относится только к одной из указанных категорий — он описывается как смесь нескольких источников разной интенсивности. Например, красный луч лазера в лаборатории образован почти чисто красным отражательным компонентом. Однако частицы дыма или пыли рассеивают луч, поэтому можно видеть, как он проходит через комнату. Такое рассеяние представляет

диффузный компонент света. Если луч достаточно ярок, и нет других источников света, объекты в комнате приобретают красноватый оттенок, и здесь присутствует незначительный рассеянный свет.

Таким образом, говорят, что источник света на сцене имеет три компонента: рассеянный, диффузный и отражаемый свет. Точно так же компоненты цвета, составляющие освещение, определяются RGBA-кодом, описывающим относительные интенсивности красного, зеленого и синего света, составляющего данный компонент. Например, красный свет описанного лазера можно представить через компоненты, указанные в табл. 1.

Обратите внимание на то, что красный луч лазера не содержит зеленого или синего света. Также отметим, что отражаемый, диффузный и рассеянный свет могут представляться значениями из диапазона 0-1. Изучая приведенную таблицу, можно сказать, что красный луч лазера на некоторой сцене имеет очень большой отражаемый компонент, небольшой диффузный компонент и очень маленький рассеянный компонент.

Таблица 1. Распределение цвета и света в источнике-лазере

	Красный	Зеленый	Синий	Альфа
Отражаемый свет	0,99	0,0	0,0	1,0
Диффузный свет	0,10	0,0	0,0	1,0
Рассеянный свет	0,05	0,0	0,0	1,0

В месте, на которое направлен луч, скорее всего будет видно красное пятно. Кроме того, из-за условий в комнате (дым, пыль и т.д.) диффузный компонент позволяет видеть луч, проходящий через комнату. Наконец, рассеянный свет немного рассеивает свет по всей комнате (также благодаря частицам дыма или пыли). Рассеянный и диффузный компоненты света часто объединяются, поскольку они схожи по своей природе.

Материалы в реальном мире

Свет — это лишь один компонент общей картины. В реальном мире объекты имеют собственный цвет. Синий мяч отражает преимущественно синие фотоны и поглощает большинство других. Это предполагает, что свет, которым освещается мяч, имеет синие фотоны, которые, отражаясь, попадут в глаз наблюдателя. В общем случае большинство сцен в реальном мире освещается белым светом, содержащим равномерную смесь всех цветов. Следовательно, при освещении белым светом большинство объектов предстают в "естественных" цветах. Однако это не всегда так; если поместить синий мяч в темную комнату, которая освещается только желтым светом, наблюдателю мяч будет казаться черным, поскольку весь желтый свет поглощается, а синего света, который можно было бы отразить, нет.

Свойства материалов

Используя освещение, не говорится, что многоугольники имеют цвет, скорее указываются материалы, которые имеют определенные отражательные свойства. Вместо того чтобы сказать, что многоугольник красный, говорится, что многоугольник сделан из материала, отражающего преимущественно красный свет. При этом по-прежнему утверждается, что поверхность красная, но теперь необходимо дополнительно задать отражательные свойства материала для источников рассеянного, диффузного и отраженного света. Материал может быть блестящим и очень хорошо отражать отраженный свет, поглощая большую часть рассеянного или диффузного света. И наоборот, тусклый освещенный объект может поглощать весь отраженный свет и ни при каких обстоятельствах не выглядеть блестящим. Кроме того, нужно задавать еще и излучающие свойства таких объектов, испускающих собственный свет, как задние габаритные фонари автомобиля или часы, светящиеся в темноте.

Добавление света к материалам

Установка освещения и свойств материалов для достижения желаемых эффектов требует некоторой практики. При рисовании объекта OpenGL решает, какой цвет использовать для каждого пикселя объекта. Объект характеризуется отражательными "цветами", а источник света — своими "цветами". Как OpenGL определяет, какие цвета использовать? Понять соответствующие принципы несложно.

Каждой вершине используемых примитивов присваивается RGB-код, основанный на суммарном эффекте рассеянного, диффузного и отражаемого освещения, умноженного на способность материала отражать рассеянный, диффузный и отражательный свет. В результате, поскольку используется плавное сопряжение цветов вершин, достигается иллюзия реального освещения.

Расчет эффектов рассеянного света

Чтобы рассчитать эффекты рассеянного света, вначале потребуется отказаться от понятия цвета, а помнить только об интенсивностях красного, зеленого и синего компонентов. Источнику рассеянного света, дающему свет, состоящий из красного, зеленого и синего компонентов половинной интенсивности, соответствует RGB-код (0.5,0.5,0.5). Если этот рассеянный свет освещает объект, отражательная способность рассеянного света которого задается RGB-кодом (0.5,1.0,0.5), суммарный вклад рассеянного света в "цвет" запишется следующим образом

```
(0.5 * 0.5, 0.5 * 1.0, 0.5 * 0.5) = (0.25, 0.5,
                                     0.25)
```

Таким образом, умножая каждую составляющую света источника на соответствующий компонент свойства материала (рис. 8).

Рис. 8 – Расчет компонента "рассеянный свет" цвета объекта

Следовательно, компоненты цвета материала определяют, какой процент падающего света будет отражен. В примере в рассеянном свете присутствует красный компонент, интенсивность которого равна половине максимальной, а свойство материала, характеризующее способность отражения отражательного света, равно 0,5, т.е. будет отражена половина половины максимальной интенсивности, или одна четвертая — 0,25.

Диффузные и отражательные эффекты

Расчет рассеянного света довольно прост. Диффузный свет описывается интенсивностью RGB-компонентов, точно так же взаимодействующими с материалом, который характеризуется определенными свойствами. Однако диффузный свет является направленным, а интенсивность на поверхности объекта меняется в зависимости от угла между поверхностью и источником света, расстояния до источника света и коэффициентов ослабления (если между источником и поверхностью имеется не полностью прозрачная среда, подобная туману) и т.д. То же можно сказать и об источниках и интенсивностях отраженного света. Суммарный эффект, выраженный через RGB-коды, рассчитывается точно так же, как для рассеянного света, но интенсивности источника света (скорректированные с учетом угла падения) умножаются на параметры отражательных свойств материала. После этого все три полученных набора RGB-кодов суммируются и дают

окончательный цвет объекта. Если какой-то компонент цвета превышает 1,0, он устанавливается равным 1.

В общем случае рассеянный и диффузный компоненты источников света и материалов одинаковы и в основном определяют цвета объекта. Отраженный свет и соответствующие свойства материала обычно имеют характеристики, близкие к светлосерому или белому цвету. Отраженный компонент существенно зависит от угла падения, а блики на поверхности объекта обычно имеют белый цвет.

Добавление света к сцене

Изложенный ниже материал поможет систематизировать все вышесказанное. Рассмотрим несколько примеров кода OpenGL, необходимого для создания освещения; это также поможет закрепить все, что было изучено. Ниже приведено несколько примеров, основанных на программе JET. Исходная версия не содержит кода, генерирующего освещение, — в ней просто рисуются треугольники с активизированным удалением скрытых поверхностей (проверка глубин).

Использование источников света

Манипулирование рассеянным светом используется, но в большинстве приложений, где моделируется мир, приближенный к реальному, следует задавать один или несколько источников света. Помимо интенсивности и цвета эти источники характеризуются положением и/или направлением. Отметим, что размещение таких источников света может очень сильно влиять на вид сцены.

OpenGL поддерживает использование по крайней мере восьми независимых источников света, расположенных в любых точках сцены или даже вне отображаемого объема. Можно также поместить источник света в бесконечности, сделав его лучи параллельными, или заставить близлежащий источник света излучать во все стороны. Кроме того, можно установить точечный источник света, излучающий в пре-

делах заданного конуса света, и приписать ему любые характеристики.

Задавая источник света, сообщаем OpenGL, где он находится и в каком направлении светит. Часто источник света излучает во всех направлениях (тогда он называется *ненаправленным),* но он может быть и направленным. В любом случае, какой бы объект ни рисовали, лучи света от любого источника (отличного от источника рассеянного света) под углом падают на поверхность объекта, образованную множеством многоугольников. Разумеется, если речь идет о направленном свете, не обязательно все поверхности всех многоугольников будут освещены. Отметим, что для расчета эффектов затенения на поверхности многоугольников OpenGL требуется информация, на основании которой можно рассчитать угол.

На рис.9 показано, как на многоугольник (квадрат) падает луч света от некоторого источника. Луч образует угол A с плоскостью, на которую он падает. Затем свет отражается под углом B в сторону наблюдателя (в противном случае его бы не видели). Эти углы используются вместе с параметрами освещения и материалов (см. выше) для расчета кажущегося цвета обозначенной точки. Так получилось, что положения, используемые OpenGL, являются вершинами многоугольника. Поскольку OpenGL рассчитывает кажущиеся цвета всех вершин, а затем создает между ними плавные переходы, создается иллюзия освещения.

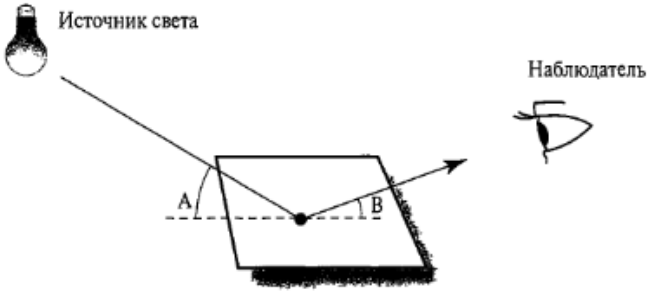

Рис. 9 – Свет отражается от объектов под определенным углом

С точки зрения программирования эти расчеты освещения представляют определенную концептуальную сложность. Каждый много-

угольник создается как набор вершин, представляющих собой просто точки. После этого на каждую вершину под определенным углом падает луч света. Каким же образом OpenGL рассчитывает угол между точкой и линией (лучом света)? Разумеется, нельзя геометрически найти угол между одной точкой и линией в трехмерном пространстве, поскольку общее число возможных результатов бесконечно. Следовательно, с каждой вершиной нужно соотнести определенную информацию, сообщающую направление "вверх" от вершины и "прочь" от поверхности примитива.

Нормали к поверхности

Линия, идущая под прямым углом от вершины в направлении "вверх", начинается на воображаемой плоскости (плоскости многоугольника). Данная линия называется *вектором нормали* (или *нормалью)*. Таким образом, вектор нормали — это линия, указывающая в направлении, образующем угол 90° с поверхностью вашего многоугольника. Примеры двух- и трехмерных векторов нормали представлены на рис. 10.

Рис. 10 – Двух- и трехмерные векторы нормали

Возможно, у вас уже возник вопрос, почему необходимо задавать вектор нормали для каждой вершины. Почему нельзя просто задать нормаль для всего многоугольника и использовать ее для каждой вершины? Можно так сделать, более того, так и будетв последующих примерах. Однако иногда невыгодно, чтобы все нормали были пер-

пендикулярны поверхности многоугольника. Можно заметить, что многие поверхности не являются плоскими! Их можно аппроксимировать плоскими многоугольными фрагментами, но в результате будет получена угловатая фасеточная поверхность.

Задание нормали

Чтобы понять, как задается нормаль для вершины, рассмотрим рис. 11, где изображена плоскость, "плавающая" над плоскостью ***xz*** в трехмерном пространстве. Данный рисунок служит только для иллюстрации концепции. Обратите внимание на линию, проходящую через вершину (1,1,0) перпендикулярно плоскости.

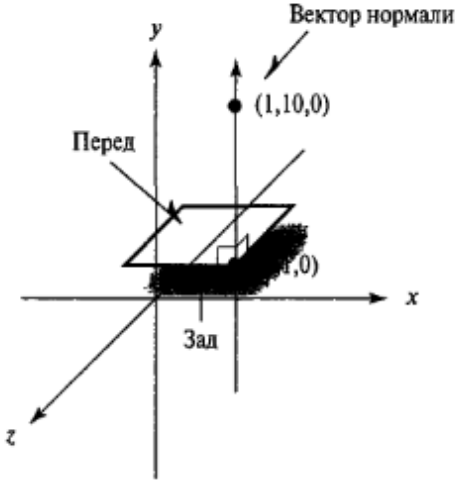

Рис. 11 – Вектор нормали, перпендикулярный поверхности

Если выбрать любую точку на этой линии, скажем, (1,10,0), линия, проходящая от первой точки (1,1,0) до этой второй точки (1,10,0), будет вектором нормали. Вторая заданная точка в действительности указывает, что направление от вершины совпадает с положительным направлением оси ***y.*** Данная договоренность также используется для обозначения передних и задних сторон многоугольников, поскольку векторы направлены вверх и от передней поверхности.

Видно, что эта вторая точка характеризуется числом единиц по положительным направлениям осей ***x, y*** и ***z*** до некоторой точки вектора нормали, указывающего от вершины. Вместо того чтобы задавать две точки для каждого вектора нормали, можно вычесть вершину из второй точки нормали и получить тройку координат, обозначающую расстояния по осям ***x, y*** и ***z*** от вершины до точки. В примере получаем такие три величины.

(1,10,0) - (1,1,0) = (1 - 1,10 - 1,0) = (0,9,0)

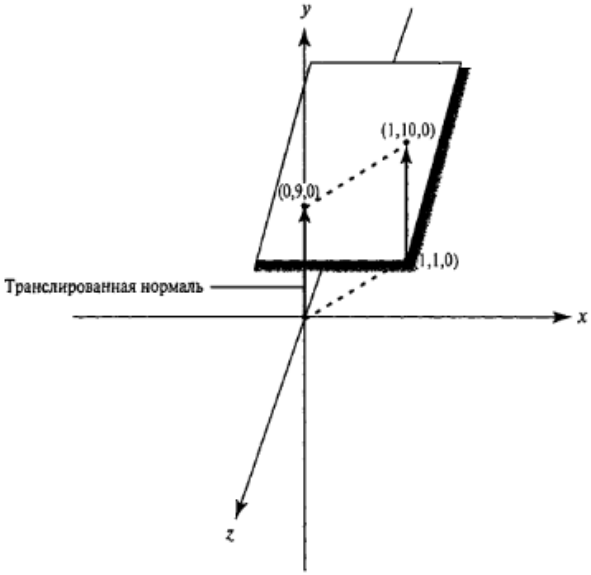

Рис. 12 –Новый транслированный вектор нормали

На данный пример можно посмотреть и по-другому: если транслировать вершину в начало координат, точка, заданная как разность двух исходных точек, по-прежнему будет задавать направление, образующее угол 90° с поверхностью. Такой транслированный вектор нормали показан на рис. 12.

Вектор — это направленная величина, сообщающая OpenGL, в каком направлении располагается лицевая сторона вершины (или мно-

гоугольника). В следующем сегменте кода показан вектор нормали для одного из треугольников,

```
glBegin(GL_TRIANGLES);
    glNormal3f(0.0f,   -1.0f,  0.0f);
    glVertex3f(0.0f,   0.0f,  60.0f);
    glVertex3f(-15.0f,  0.0f,  30.0f);
    glVertex3f(15.0f,0.0f,30.0f);
glEnd();
```

В качестве аргументов функция `glNormal3f` принимает тройку координат, задающих вектор нормали, который указывает направление, перпендикулярное поверхности треугольника. В примере нормали ко всем трем вершинам имеют одинаковую ориентацию — по отрицательному направлению оси ***y.*** Это простой пример, поскольку треугольник целиком лежит на плоскости ***xz,*** и представляет в действительности нижний фрагмент самолета. Позже буде видно, что часто для всех вершин требуется задавать разные нормали.

Перспектива задания нормали для каждой вершины каждого многоугольника, который рисуется, может привести в уныние, особенно, если лишь несколько поверхностей лежат на одной из основных плоскостей. Чуть ниже будет описана полезная функция, которую можно вызывать снова и снова для расчета нормалей за вас.

Обратите особое внимание на порядок вершин в треугольниках самолета. Если наблюдаетяс рисование этого треугольника с направления, в котором указывает вектор нормали, вершины кажутся расположенными вокруг треугольника против часовой стрелки. Это называется *обходом многоугольника.* По умолчанию передняя грань многоугольника определяется как сторона, вершины которой обходятся против часовой стрелки.

Единичные нормали

Если дать OpenGL применить свою магию, все нормали к поверхности будут преобразованы в единичные нормали. *Единичным* называется вектор нормали, длина которого равна 1. Длина нормали на

рис. 12 равна 9. Чтобы найти длину нормали, нужно возвести в квадрат все ее компоненты, сложить их и извлечь из суммы квадратный корень. Если поделить каждый компонент нормали на ее длину, получится вектор, указывающий в том же направлении, но имеющий единичную длину. В этом случае новый вектор нормали будет задаваться как (0,1,0). Описанный процесс получил название *нормировки.* Итак, при расчете освещения все векторы нормали должны *нормироваться.*

Вы можете указать OpenGL автоматически преобразовать имеющиеся нормали в единичные, активизировав нормировку с помощью вызова `glEnable` с параметром `GL_NORMALIZE`.

```
glEnable(GL_NORMALIZE);
```

Следует отметить, что вызовы функции преобразования `glScale` также масштабируют длины нормалей. Если в одной программе используется и `glScale`, и функции освещения, последние могут дать нежелательные эффекты. Если были заданы единичные нормали для всех геометрических объектов и использовали постоянный коэффициент масштабирования в `glScale` (все геометрические объекты масштабируются одинаково), можно использовать новую альтернативу `GL_NORMALIZE` — `GL_RESCALE_NORMALS`. Чтобы активизировать эту возможность, вызывается следующая функция:

```
glEnable(GL_RESCALE_NORMALS);
```

Этот вызов сообщает OpenGL, что нормали не имеют единичной длины, но их можно одинаково масштабировать, чтобы получить единичную длину. Чтобы обработать такую ситуацию, OpenGL исследует матрицу наблюдения модели. В результате получаем меньше математических операций на вершину, чем потребовалось бы в противном случае.

Поскольку перед началом работы OpenGL ему лучше предоставить единичные нормали, библиотека `glTools` содержит функцию, которая принимает любой вектор нормали и нормирует его.

```
void gltNormalizeVector(GLTVector vNormal);
```

Нахождение нормали

На рис.13 представлен многоугольник, не лежащий целиком в одной из осевых плоскостей. Угадать вектор нормали, указывающий от этой поверхности, гораздо сложнее, поэтому нужен удобный способ расчета нормали произвольного многоугольника в трехмерных координатах.

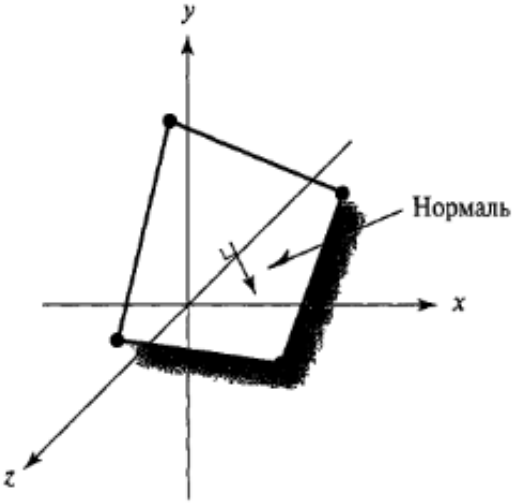

Рис. 13 – Нетривиальная задача нахождения нормали

Вектор нормали любого многоугольника легко рассчитать по трем точкам, на плоскости этого многоугольника. На рис.14 показаны три точки — P_1 P_2 и P_3,- которые можно использовать для определения двух векторов: вектора V_1, направленного от P_1 к P_2, и вектора V_2, направленного от P_1 к P_3. Математически два вектора в трехмерном пространстве определяют плоскость. Если найти векторное произведение двух векторов (математически это записывается как V_1 x V_2), получающийся в результате вектор будет перпендикулярен данной плоскости. На рис.15, например, показан вектор V_3, являющийся векторным произведением V_1 и V_2.

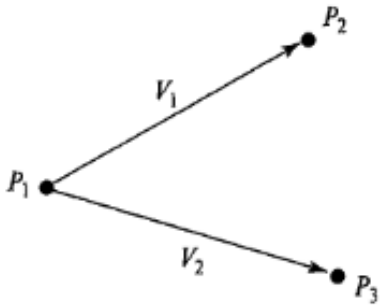

Рис. 14 – Два вектора, определенные тремя точками на плоскости

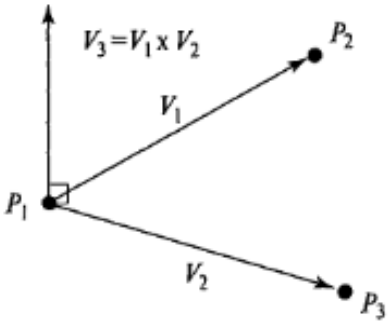

Рис. 15 – Вектор нормали, определенный как векторное произведение двух векторов

Поскольку данный метод полезен и используется очень часто, библиотека `glTools` содержит функцию, рассчитывающую вектор нормали по трем данным точкам многоугольника.

```
  void gltGetNormalVector(GLTVector vP1, GLTVector
vP2,
                          GLTVector vP3,  GLTVec-
tor vNormal);
```

Чтобы использовать эту функцию, ей нужно передать три вектора (каждый из которых представляет собой массив из трех величин с плавающей запятой) из многоугольника или треугольника (заданных

при обходе вершин против часовой стрелки), а на выходе будет получен другой векторный массив, содержащий вектор нормали.

Эффекты освещения

Рассеянного и диффузного света достаточно, чтобы создать иллюзию освещения. Поверхность самолета кажется затененной согласно углу падения света. При вращении модели эти углы меняются, и эффекты освещения видятся меняющимися так, что легко догадаться, откуда поступает свет.

Мы не учитывали отраженный компонент источника света, а также способность материала самолета к отражению. Хотя эффекты освещения достаточно отчетливы, поверхность самолета кажется раскрашенной ровно и уныло. Рассеянного и диффузного освещения и соответствующих свойств материала достаточно, если моделируетяс глина, дерево, картон, одежда или другой ровно окрашенный объект. Для таких металлических поверхностей, как обшивка самолета, желательно немного блеска.

Отраженные блики

Необходимый блеск поверхности ваших объектов добавляют зеркальное отражение и соответствующие свойства материала. Требуемое сияние имеет отбеливающие влияния на цвет объекта и может давать блики на поверхности при остром угле между лучом падения света и направлением на наблюдателя. Зеркальные блики появляются практически всегда, когда свет падает на поверхность объекта и отражается от нее. Хорошим примером блика является белая искорка на блестящем красном шаре, выставленном на солнечный свет.

Отраженный свет

Добавить к источнику света компонент отраженного света достаточно легко. В приведенном ниже коде иллюстрируется установка источника света в предыдущей программе, модифицированной добавлением компонента отраженного света.

```
  // Коды и координаты источников света
  GLfloat ambientLight[] = { 0.3f,  0.3f,  0.3f,
1.0f };
  GLfloat diffuseLight[] = { 0.7f,  0.7f,  0.7f,
1.0f );
  GLfloat  specular[]  =  {  1.0f,  1.0f,   1.0f,
1.0f};
  ...
  ...
  // Активизируется освещение
  glEnable(GL_LIGHTING);

  // Устанавливается и активизируется источник
света 0 glLightfv(GL_LIGHT0,GL_AMBIENT, ambien-
tLight);
glLightfv(GL_LIGHT0,GL_DIFFUSE,diffuseLight);
glLightfv(GL_LlGHT0,GL_SPECULAR,specular);
  glEnable(GL_LIGHT0);
```

Массив `specular[]` задает в качестве компонента отраженного света очень яркий источник белого света. В данном случае целью является моделирование яркого солнечного света. Следующая строка просто прибавляет компонент отраженного света к источнику света `GL_LIGHT0`.

```
  glLightfv(GL_LIGHT0,GL_SPECULAR,specular);
```

Если бы это было единственным изменением в программе, не было бы никакой разницы во внешнем виде самолета —соответствующие отражательнех свойства материала не определены.

Зеркальное отражение

Добавление к свойствам материала коэффициента зеркального отражения выполняется так же просто, как добавление зеркального компонента к источнику света. В следующем фрагменте кода приводятся команды программы, на этот раз модифицированной так, чтобы материал имел ненулевой коэффициент зеркального отражения.

```
// Коды и координаты источников света
GLfloat specref[] = {  1.0f,  1.0f,  1.0f,  1.0f
};
... ...
// Активизируем согласование цветов
glEnable(GL_COLOR_MATERIAL);
// Свойства материалов согласуются с кодами glColor glCo-
lorMaterial(GL_FRONT, GL_AMBIENT_AND_DIFFUSE);
// С этого момента все материалы имеют максимальный коэффи-
циент
// зеркального отражения
glMaterialfv{GL_FRONT,   GL_SPECULAR,  specref};
glMateriali(GL_FRONT,   GL_SHININESS,  128);
```

Как и ранее, так активизировалось согласование цветов, чтобы коэффициенты рассеянного и диффузного отражения материалов соответствовали текущему цвету, заданному функциями `glColor`.

Теперь был добавлен массив `specref[]`, который содержит RGBA-коды коэффициента зеркального отражения. Этот массив, состоящий целиком из единиц, характеризует поверхность, которая отражает практически весь отражаемый свет. В приведенной ниже строке указывается, что материал всех последующих многоугольников будет иметь такой же коэффициент зеркального отражения.

```
glMaterialfv(GL_FRONT, GL_SPECULAR, specref);
```

Поскольку функция `glMaterial` не вызывается еще раз с параметром `GL_SPECULAR`, указанное свойство имеют все материалы. Это сделано специально, поскольку требуется, чтобы весь самолет казался сделанным из металла или блестящих сплавов.

Также в программе задали, что отражательные свойства (диффузный и рассеянный свет) материала всех последующих многоугольников (если не изменять это поведение с помощью вызова `glMaterial` или `glColorMaterial`) меняется так же, как текущий цвет, но свойства зеркального отражения при этом остаются прежними.

Тень

Добавление тени к сценам может существенно повысить их реализм и визуальную эффективность. На рис 16 и 17 показаны освещенные объектыа. Хотя оба варианта окрашены, объект с тенью более убедителен, чем без нее.

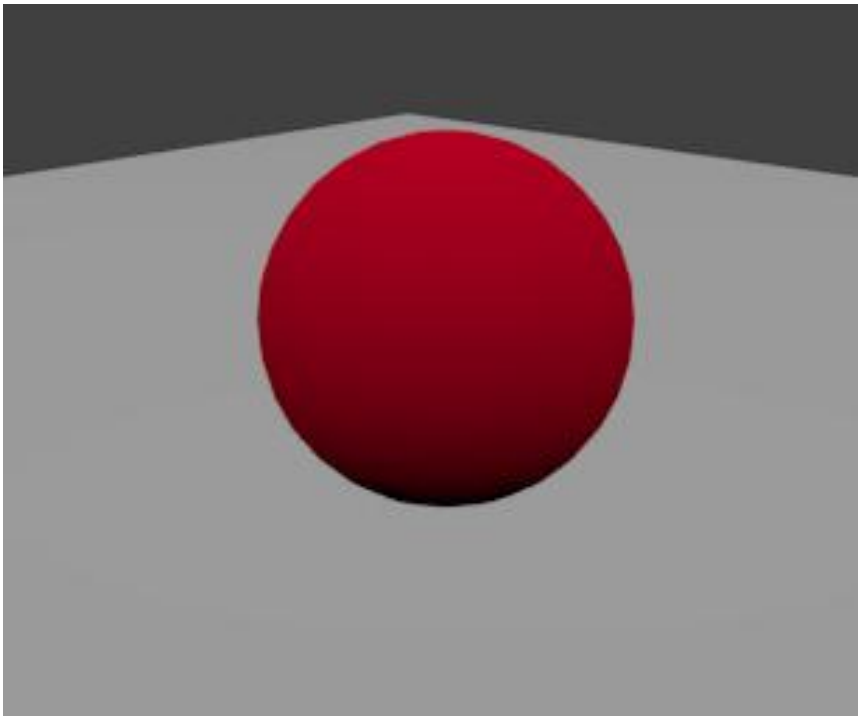

Рис. 16 – Освещенный объект без тени

Что такое тень?

Концептуально рисование тени является достаточно простым. Тень получается тогда, когда объект закрывает свет от источника света и не дает ему попасть на другой объект или поверхность, расположенную за объектом, создающим тень. Область поверхности затененного объ-

екта, на которую накладывается контур затеняющего объекта, кажется темной.

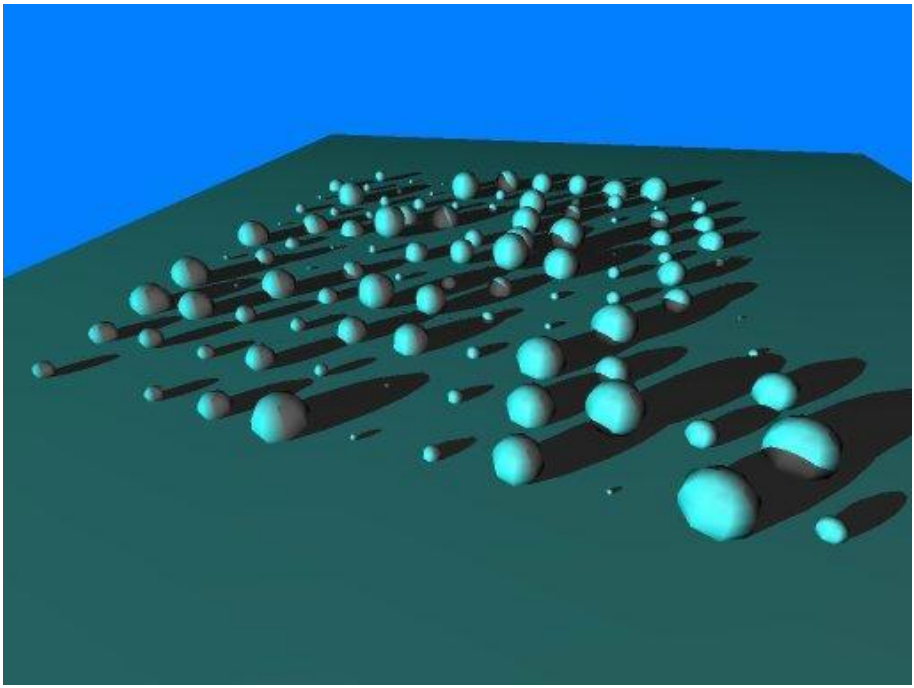

Рис. 17 – Освещенный объект с тенью

Тень можно создать программными средствами, спроектировав исходный объект на плоскость поверхности, содержащей объект. Затем объект рисуется черным (или другим темным цветом), возможно, частично прозрачным. Существует множество методов и алгоритмов рисования теней (в том числе достаточно сложные). Данной лабораторной работе будет продемонстрирован один из простейших методов, хорошо работающий при наложении теней на плоскую поверхность (такую, как земля). Процесс подобного проектирования иллюстрируется на рис. 18.

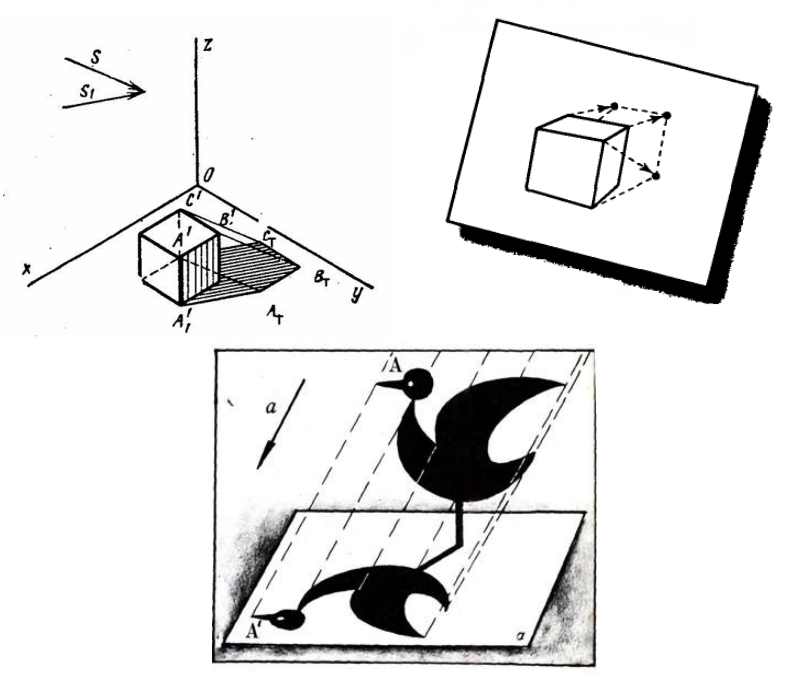

Рис. 18 – Проектирование объекта с целью создания тени

Для наложения проекции объекта на другую поверхность используются нетривиальные действия с матрицами. Сейчас можно попытаться свести процесс к максимально простым концепциям и действиям.

Многие эффекты достигаются за счет смешения цветов. Такие прозрачные объекты, как витражное стекло или пластиковые бутылки, позволяют видеть объекты, расположенные за ними, но свет таких объектов смешивается с цветом прозрачного материала, через который смотрит наблюдатель. В OpenGL подобная прозрачность достигается за счет следующей схемы. Вначале рисуются фоновые объекты, а затем цвет объектов переднего плана смешивается с цветами, уже присутствующими в буфере цветов. Чтобы овладеть этой техникой, нужно рассмотреть четвертый компонент цвета, который пока не изучен, – величина *альфа*.

Смешение

Вы уже знаете, что процедуры визуализации OpenGL при нормальных обстоятельствах помещают коды цвета в буфер цветов. Кроме того, известно, что в буфер глубины помещаются параметры глубины каждого фрагмента. При отключенной проверке глубины новые коды цвета просто записываются поверх значений, уже присутствующих в буфере цветов. Если включить (активизировать) проверку глубины, новые цветные фрагменты будут замещать существующие только в том случае, если они располагаются ближе к ближней плоскости отсечения, чем значения, записанные в буфере. Разумеется, все сказанное относится к нормальным обстоятельствам, и сформулированные правила теряют силу в тот момент, когда включается смешение цветов OpenGL.

```
glEnable(GL_BLENDING);
```

При активизированном смешении цветов новый цвет объединяется с кодом цвета, уже присутствующим в буфере цветов. Используя различные способы объединения цветов, можно получить интересные специальные эффекты.

Объединение цветов

Вначале необходимо ввести более строгую терминологию поступающих кодов цвета и кодов, уже присутствующих в буфере. Код, уже записанный в буфере цвета, называется целевым, и содержит три отдельных компонента (соответствующих красному, зеленому и синему цветам) и (необязательно) записанное значение альфа. Код цвета, поступающий в результате выполнения команд визуализации и взаимодействующий или не взаимодействующий с целевым цветом, называется исходным. Исходный цвет также содержит три или четыре компонента (красный, зеленый, синий и, возможно, альфа).

Объединение исходного и целевого цветов при смешении происходит согласно уравнению смешивания. По умолчанию уравнение смешения выглядит следующим образом.

$C_f = (C_s * S) + (C_d * D)$

Здесь C_f — конечный цвет, C_s — исходный цвет, C_d — целевой цвет, а S и D — коэффициенты смешения источника и цели. Коэффициенты смешения задаются с помощью следующей функции

```
glBlendFunc(GLenum S, GLenum D);
```

Видно, что S и D относятся к типу перечисляемых, а не физических величин, которые задаются непосредственно. В табл. 2 перечислены возможные значения смешивающих функций. Нижние индексы обозначают “исходный цвет” (source — s), “целевой цвет” (destination — d) и “цвет смеси” (color — C, рассматривается ниже) R, G, B и A обозначают красный, зеленый, синий и альфа-компонент, соответственно.

Помните, что цвета представляются величинами с плавающей запятой, поэтому их сложение, вычитание и даже умножение относятся к разрешенным операциям. Табл. 1 может показаться слишком запутанной, поэтому рассмотрим типичную функцию смешения.

```
glBlendFunc(GL_SRC_ALPHA,
GL_ONE_MINUS_SRC_ALPHA);
```

Приведенная функция сообщает OpenGL взять исходный (поступающий) цвет и умножить цвет (RGB-значения) на значение альфа. Полученный результат нужно прибавить к результату умножения целевого цвета на один минус альфа-фактор источника Предположим, например, что в буфере цвета уже имеется красный цвет (l.0f, 0.0f, 0.0f, 0.0f). Это целевой цвет (Cd) Если поверх рисуется что-либо синего цвета с альфа-фактором 0.5 (0.0f, 0.0f, l.0f, 0.5f), конечный цвет можно рассчитать следующим образом.

Cd, = целевой цвет = (1.0f, 0.0f, 0.0f, 0.0f)

Cs = исходный цвет = (0.0f, 0.0f, 1.0f, 0.5f)

S = альфа-фактор источника = 0.5

D = единица минус альфа-фактор источника = 1.0 - 0.5 = 0.5

Теперь уравнение

$C_f = (C_s * S) + (C_d * D)$

превращается в такое:

C_f = (синий * 0.5) + (красный * 0.5)

Таблица 2. Функции смешивания OpenGL

Функция	Коэффициенты смешения (RGB)	Коэффициент смешения (альфа)
GL_ZERO	(0, 0, 0)	0
GL_ONE	(1, 1, 1)	1
GL_SRC_COLOR	(R_s, G_s, B_s)	A_s
GL_ONE_MINUS_SRC_COLOR	(1, 1, 1)-(R_s, G_s, B_s)	1 - A_s
GL_DST_COLOR	(R_d,G_d,B_d)	A_d
GL_ONE_MINUS_DST_COLOR	(1,1,1) - (R_d, G_d, B_d)	1 -A_d
GL_SRC_ALPHA	(A_s, A_s, A_s)	A_s
GL_ONE_MINUS_SRC_ALPHA	(1,1,1)- (A_s, A_s, A_s)	1-A_s
GL_DST_ALPHA	(A_d, A_d, A_d)	A_d
GL_ONE_MINUS_DST_ALPHA	(l, l, l)-(A_d, A_d, A_d)	1-A_d
GL_CONSTANT_COLOR	(R_c, G_c, B_c)	A_c
GL_ONE_MINUS_CONSTANT_COLOR	(1,1,1) - (R_c, G_c, B_c)	1 - A_c
GL_CONSTANT_ALPHA	(A_c, A_c, A_c)	A_c
GL_ONE_MINUS_CONSTANT_ALPHA	(1,1,1)-(A_c, A_c, A_c)	1 -A_c

GL_SRC_ALPHA_SATURATE	(f, f, f)*	1
	f = min(A_s, 1 - A_d)	

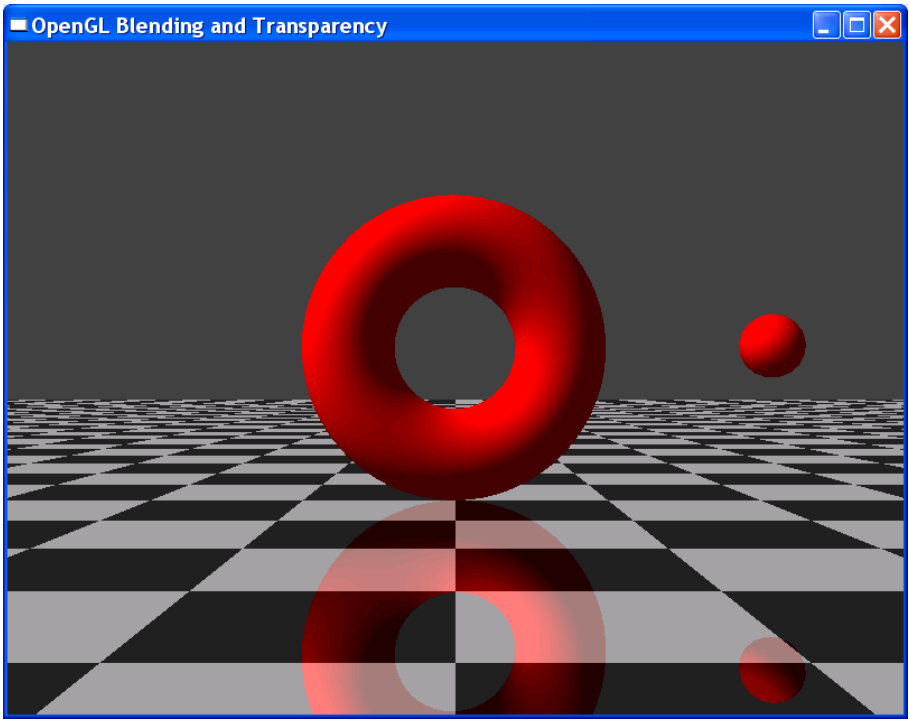

Рис. 19 – Использование смешивания для получения ложного эффекта отражения

Изменение уравнения смешивания

Приведенное ранее уравнение смешения

$Cf = (_{C}s * S) + (C_d * D)$

является *уравнением по умолчанию*. В действительности можно выбирать из пяти различных уравнений смешения, перечисленных в табл. 3 и задаваемых с помощью следующей функции:

```
void glBlendEquation(GLenum mode);
```

Таблица 3. Доступные режимы (уравнения) смешивания

Режим	Функция
GL_FUNC_ADD (режим по умолчанию)	Cf = (Cs * S) + (Cd * D)
GL_FUNC_SUBTRACT	Cf = (Cs * S) – (Cd * D)
GL_FUNC_REVERSE_SUBTRACT	Cf = (Cd * D) – (Cs * S)
GL_MIN	Cf = min(Cs, Cd)
GL_MAX	Cf = max(Cs, Cd)

Если гибкости glBlendFunc недостаточно, можно применять следующую функцию

```
void glBlendFuncSeparate(GLenum srcRGB, GLenum dstRGB, GLenum srcAlpha, GLenum dstAlpha);
```

Если glBlendFunc задает функции смешивания исходного и целевого RGBA-кодов, то glBlendFuncSeparate позволяет по отдельности задавать функции смешивания для компонентов RGB и альфа

Наконец, как показано в таблице 2, значения GL_CONSTANT_COLOR, GL_ONE_MINUS_CONSTANT_COLOR, GL_CONSTANT_ALPHA и GL_ONE_MINUS_CONSTANT_ALPHA позволяют вводить в уравнение смешивания постоянный цвет-компонент. Этот постоянный цвет изначально (по умолчанию) является черным (0.0f, 0.0f, 0.0f, 0.0f), но его можно изменить с помощью приведенной ниже функции

```
void glBlendColor(GLclampf red, GLclampf green,
 Glclampf blue, GLclampf alpha);
```

ЗАДАЧИ И ПОРЯДОК ВЫПОЛНЕНИЯ РАБОТЫ

Затенение

Функция `glColor` устанавливает текущий цвет, который используется для рисования всех вершин, заданных после этой команды. До этого момента во всех примерах рисовались каркасные объекты или сплошные тела, каждая грань которых изображалась своим цветом. А каким будет цвет объекта, если вершины примитива (точки, отрезка или многоугольника) задать разноцветными?

Рассмотрим вначале точки. Точка имеет только одну вершину, поэтому какой бы цвет не задали для этой вершины, точка будет изображена одним этим цветом. Все просто.

Рис. 20 – Закрашивание отрезка с черной и белой вершинами отражения

Отрезок имеет две вершины, которые могут иметь разные цвета. Цвет отрезка зависит от модели затенения. *Затенение* определяется

просто как гладкий переход одного цвета в другой. С помощью прямого отреза можно соединить любые две точки пространства цветов RGB (см. рис. 2).

При *гладком затенении* цвета вдоль линии меняются так же, как при проходе куба цвета от точки, соответствующей одному цвету, к точке, соответствующей другому. На рис. 20 показан куб цветов с обозначенными черной и белой вершинами. Под ним изображена линия с двумя вершинами — черной и белой. Цвета, выбираемые для точек отрезка, соответствуют цветам, расположенным на диагонали куба, соединяющей черную и белую вершины. В результате получается отрезок, плавно переходящий от черного цвета к белому через различные оттенки серого.

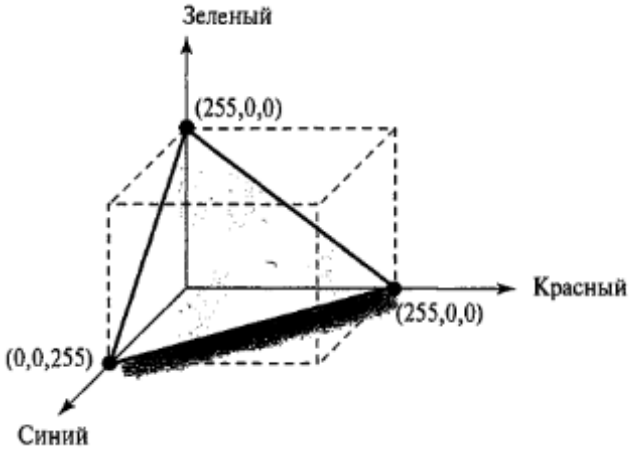

Рис. 21 – Треугольник в пространстве цветов RGB

Затенение можно описать математически, найдя уравнение линии, соединяющей две точки в трехмерном пространстве цветов RGB. После этого можно просто "пройти" от одного конца линии к другому, извлекая по пути координаты, определяющие цвет всех пикселей на экране. Во многих книгах по компьютерной графике объясняется алгоритм, позволяющий достичь такого эффекта, масштабировать цветной отрезок в физическую линию на экране и т.д.

Для многоугольников затенение становится немного сложнее. Треугольник, например, также можно представить как плоскость, принадлежащую кубу цвета. На рис. 21 показан треугольник, вершины которого — максимально насыщенные красный, зеленый и синий цвета. Код отображения этого треугольника на экране приводится в листинге 1.

Выбор модели затенения

В листинге 1 задается модель затенения OpenGL, которая будет использоваться для плавного преобразования одного цвета в другой. Это модель затенения по умолчанию, но чтобы гарантировать, что ваша программа работает так, как предполагалось, указанную функцию стоит всегда вызывать явно.

Кроме указанной модели затенения можно задать модель плоского затенения, использовав `GL_FLAT` в качестве аргумента функции `glshadeModel`. Плоское затенение означает, что внутри примитива никаких расчетов промежуточных цветов не выполняется. В общем случае при плоском затенении цветом внутренней части примитива является цвет последней заданной вершины. Единственным исключением является примитив `GL_POLYGON`, где цвет внутренней части совпадает с цветом первой вершины.

Далее в коде, приведенном в листинге 1, задается красная верхняя вершина треугольника, зеленая правая нижняя вершина и синяя левая нижняя. Поскольку указано плавное затенение, внутренняя часть треугольника затеняется, и формируются плавные переходы цветов один в другой.

Результат выполнения программы TRIANGLE показан на рис. 22. Отметим, что данный результат представляет плоскость, показанную на рис. 21.

Разные цвета вершин могут быть и у многоугольников, более сложных, чем треугольники. В таких случаях логика затенения может быть более сложной. Не имеет значения, насколько сложными явля-

ются многоугольники, — OpenGL успешно выполнит затенение внутренних точек между любыми вершинами.

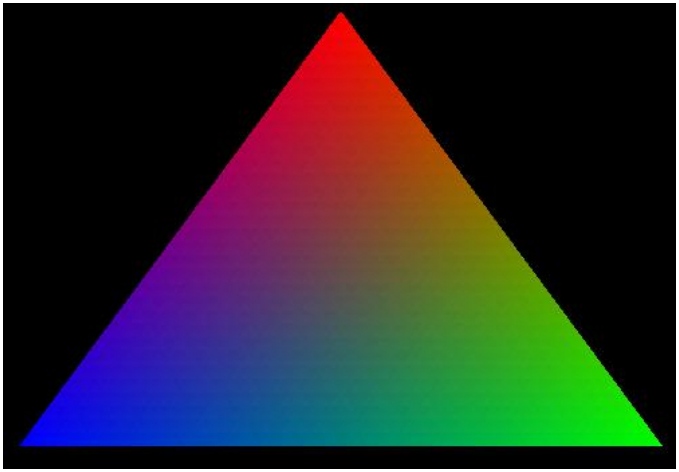

Рис. 22 –Результат выполнения программы TRIANGLE

Листинг 1 - Плавное затенение треугольника, имеющего красную, синюю и зеленую вершины

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

// Вызывается для рисования сцены
void RenderScene(void)
      {
      // Очистка окна текущим цветом очистки
      glClear(GL_COLOR_BUFFER_BIT);

      // Восстанавливаем нормальное состояние
рисования
      glShadeModel(GL_SMOOTH);

      // Рисуем треугольник
      glBegin(GL_TRIANGLES);
            // Красная вершина
```

```
            glCo-
lor3ub((GLubyte)255,(GLubyte)0,(GLubyte)0);
            glVertex3f(0.0f,200.0f,0.0f);

            // Зеленый на правом нижнем углу
            glCo-
lor3ub((GLubyte)0,(GLubyte)255,(GLubyte)0);
            glVertex3f(200.0f,-70.0f,0.0f);

            // Голубой на левом нижнем углу
            glCo-
lor3ub((GLubyte)0,(GLubyte)0,(GLubyte)255);
            glVertex3f(-200.0f, -70.0f, 0.0f);
      glEnd();

      // Очищает очередь текущих команд
      glutSwapBuffers();
      }
// Эта функция выполняет необходимую инициализацию в
контексте
// визуализации
void SetupRC()
      {
      // Черный фон
      glClearColor(0.0f, 0.0f, 0.0f, 1.0f );
      }

void ChangeSize(int w, int h)
      {
      GLfloat windowHeight,windowWidth;

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

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

      // Обновление системы координат перед модифика-
циями
    glLoadIdentity();

      // Высота окна больше ширины
      if (w <= h)
            {
            windowHeight = 250.0f*h/w;
            windowWidth = 250.0f;
            }
    else
            {
            // Ширина окна больше высоты
            windowWidth = 250.0f*w/h;
            windowHeight = 250.0f;
            }

      // Установить объем отсечения
      glOrtho(-windowWidth, windowWidth, -
windowHeight, windowHeight, 1.0f, -1.0f);
      }

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

```
glutMainLoop();
return 0;
}
```

Активизация освещения

Чтобы указать OpenGL рассчитывать освещение, вызывается функция `glEnable` с параметром `GL_LIGHTING`.

```
glEnable(GL_LIGHTING);
```

Данная команда сообщает OpenGL, что для определения цвета каждой вершины на сцене нужно использовать свойства материалов и параметры освещения. Однако, если свойства материалов или параметры освещения не заданы, объект останется темным и неосвещенным, как показано на рис. 23.

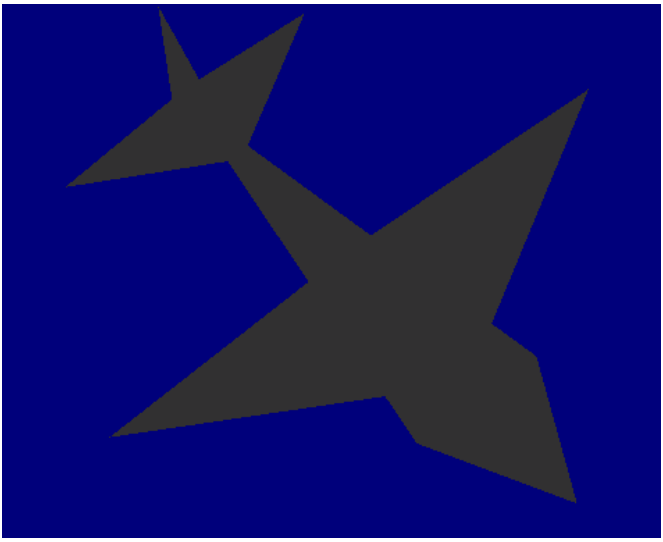

Рис. 23 – Неосвещенный самолет, не отражающий света т.е. отсутствует glEnable(GL_COLOR_MATERIAL);

Функция `SetupRC` вызывается непосредственно после формирования контекста визуализации. Именно в этот момент выполняется инициализация параметров освещения.

Настройка модели освещения

После того как активизирован расчет освещения, первое, что нужно сделать, — задать модель освещения. На модель освещения влияют три параметра, которые указываются с помощью функции `glLightModel`.

Первый из этих параметров — `GL_LIGHT_MODEL_AMBIENT` — используется в следующем примере. Он позволяет задавать глобальный рассеянный свет, равномерно со всех сторон освещающий объекты на сцене. В приведенном ниже коде, например, задается яркий белый свет.

```
// Яркий белый свет (RGB-коды максимальной интенсивности)
GLfloat ambientLight[] = { 1.0f,  1.0f,  1.0f,  1.0f };
// Активизируется освещение
glEnable(GL_LIGHTING);
// В модели освещения задается использование рассеянного света,
// заданного в функции ambientLight[]
glLightModelfv(GL_LIGHT_MODEL_AMBIENT,ambientLight);
```

Использованный в данном случае вариант функции `glLightModel (glLightModelfv)` в качестве первого параметра принимает параметр модели освещения, который задается или модифицируется, и массив RGBA-кодов, описывающий свет. Используемые по умолчанию значения `(0.2, 0.2, 0.2, 1.0)` дают довольно тусклое освещение. Параметры же модели освещения позволяют легко определять, освещаются ли передние, задние или обе части многоугольников, и видеть, как рассчитываются углы отраженного освещения.

Установка свойств материалов

Теперь, когда у нас есть источник рассеянного света, следует установить часть свойств материала, чтобы многоугольники отражали свет, и можно было видеть самолетик. Свойства материала можно за-

дать двумя способами. Первый — использовать функцию glMaterial перед заданием каждого многоугольника или набора многоугольников. Рассмотрим следующий фрагмент кода.

```
Glfloat gray[] = { 0.75f,  0.75f,  0.75f,  1.0f };
...
...
glMaterialfv(GL_FRONT,   GL_AMBIENT_AND_DIFFUSE,
gray) ; glBegin(GL_TRIANGLES);
          glVertex3f(-15.0f,0.0f,30.0f);
    glVertex3f(0.0f,   15.0f,  30.0f);
    glVertex3f(0.0f,   0.0f,  -56.0f);
glEnd();
```

Первый параметр функции glMaterialfv задает, какие стороны поверхности получают заданные свойства — передняя, задняя или обе (GL_FRONT, GL_BACK или GL_FRONT_AND_BACK). Второй параметр указывает, какие свойства устанавливаются; в приведенном примере задаются одинаковые отражательные способности рассеянного и диффузного света. Последним параметром является массив, содержащий RGBA-коды, формирующие указанные свойства. Ко всем примитивам, заданным после вызова функции glMaterial, применяются последние установленные значения, пока не будет вызвана следующая функция glMaterial.

В большинстве случаев рассеянный и диффузный компоненты одинаковы, и если не требуются эффекты зеркального отражения (искрящиеся, сияющие пятна), определять отражательные свойства не нужно. Но даже в этом случае требуется кропотливая работа по определению массива для каждого цвета объекта и вызов функции glMaterial перед каждым многоугольником или группой многоугольников.

Теперь можно рассмотреть второй (предпочтительный) вариант задания свойств материала — *согласования цветов* (color tracking). При согласовании цветов OpenGL указано задать свойства материалов, вызывая только функцию glColor. Чтобы активизировать согласование

цветов, вызывается функция glEnable с параметром GL_COLOR_MATERIAL.

```
glEnable(GL_COLOR_MATERIAL);
```

Затем с помощью функции glColorMaterial задаются параметры материала, соответствующие значениям, заданным с помощью glColor. Например, чтобы задать рассеивающие и диффузные свойства передних частей многоугольников, соответствующие цветам, заданным с помощью glColor, вызывается такая функция

```
glColorMaterial ( GL _FRONT , GL_AMBIENT_AND
_DIFFUSE);
```

В приведенном ранее фрагменте кода задаются свойства последующих материалов. Подход, который рассмотрен ниже, выглядит более громоздким, но в действительности при увеличении числа различных цветных многоугольников он экономит большое число строк кода и выполняется быстрее.

```
// Активизируется согласование цветов
glEnable(GL_COLOR_MATERIAL);
// Рассеянный и диффузный цвета передней стороны объ-
ектов
// соответствуют тому, что указано в glColor
glColorMaterial(GL_FRONT,GL_AMBIENT_AND_DIFFUSE);
...
...
glcolor3f(0.75f,  0.75f,  0.75f);
glBegin(GL_TRIANGLES);
    glVertex3f(-15.0f,0.0f,30.0f);
    glVertex3f(0.0f,  15.0f,  30.0f);
    glVertex3f(0.0f,   0.0f,  -56.0f);
glEnd();
```

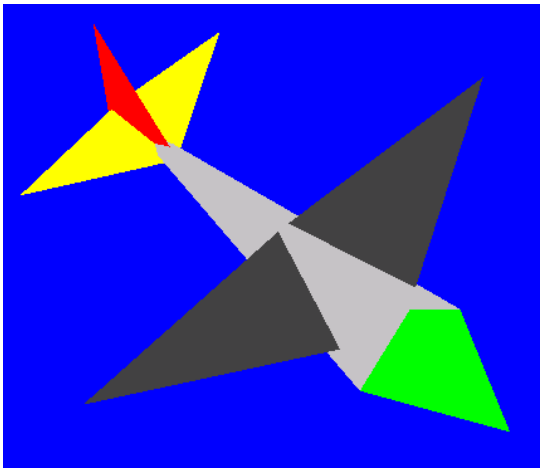

Рис. 24 – Результат выполнения завершенной программы

Листинг 2 содержит код, который с помощью функции `SetupRC` добавляет в пример с моделью самолета и который устанавливает яркий источник рассеянного света и так задает свойства материала, чтобы объект мог отражать свет и был видимым. Кроме того, изменились цвета самолета, чтобы разные цвета имели не все многоугольники, а все участки поверхности. Конечный результат выполнения программы, показанный на рис. 24, не сильно отличается от изображения, получавшегося до применения освещения. Тем не менее, если вполовину уменьшить рассеянный свет, получено более приемлемое изображение, показанное на рис. 25. В этом случае RGBA-коды рассеянного света устанавливались следующим образом.

```
GLfloat ambientLight[] = { 0.5f,  0.5f,  0.5f,   1.0f
};
```

Теперь видно, как уменьшить рассеянный свет на сцене, чтобы получить более тусклое изображение. Данная возможность полезна при моделировании сцен, на которые постепенно опускаются сумерки или заслоняется определенный источник света (один объект выходит в тень другого).

Листинг 2 – Настройка условий рассеянного освещения

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


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


//Вызывается для рисования сцены
void RenderScene(void)
    {
    // Очищаем окно текущим цветом очистки
    glClear(GL_COLOR_BUFFER_BIT |
GL_DEPTH_BUFFER_BIT);

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

      // Носовой конус ////////////////////////////
      // Ярко-зеленый
      glColor3ub(0, 255, 0);
      glBegin(GL_TRIANGLES);
            glVertex3f(0.0f, 0.0f, 60.0f);
            glVertex3f(-15.0f, 0.0f, 30.0f);
            glVertex3f(15.0f,0.0f,30.0f);

            glVertex3f(15.0f,0.0f,30.0f);
            glVertex3f(0.0f, 15.0f, 30.0f);
            glVertex3f(0.0f, 0.0f, 60.0f);

            glVertex3f(0.0f, 0.0f, 60.0f);
            glVertex3f(0.0f, 15.0f, 30.0f);
            glVertex3f(-15.0f,0.0f,30.0f);
```

```
            // Тело самолета ////////////////////////
            // Светло-серый
            glColor3ub(192,192,192);
            glVertex3f(-15.0f,0.0f,30.0f);
            glVertex3f(0.0f, 15.0f, 30.0f);
            glVertex3f(0.0f, 0.0f, -56.0f);

            glVertex3f(0.0f, 0.0f, -56.0f);
            glVertex3f(0.0f, 15.0f, 30.0f);
            glVertex3f(15.0f,0.0f,30.0f);

            glVertex3f(15.0f,0.0f,30.0f);
            glVertex3f(-15.0f, 0.0f, 30.0f);
            glVertex3f(0.0f, 0.0f, -56.0f);



////////////////////////////////////////////////
            // Левое крыло
            // Темно-серый
            glColor3ub(64,64,64);
            glVertex3f(0.0f,2.0f,27.0f);
            glVertex3f(-60.0f, 2.0f, -8.0f);
            glVertex3f(60.0f, 2.0f, -8.0f);

            glVertex3f(60.0f, 2.0f, -8.0f);
            glVertex3f(0.0f, 7.0f, -8.0f);
            glVertex3f(0.0f,2.0f,27.0f);

            glVertex3f(60.0f, 2.0f, -8.0f);
            glVertex3f(-60.0f, 2.0f, -8.0f);
            glVertex3f(0.0f,7.0f,-8.0f);


            // Другие секции верхнего крыла
```

```
            glVertex3f(0.0f,2.0f,27.0f);
            glVertex3f(0.0f, 7.0f, -8.0f);
            glVertex3f(-60.0f, 2.0f, -8.0f);

            // Хвостовая
часть//////////////////////////////
            // Нижняя часть стабилизатора
            glColor3ub(255,255,0);
            glVertex3f(-30.0f, -0.50f, -57.0f);
            glVertex3f(30.0f, -0.50f, -57.0f);
            glVertex3f(0.0f,-0.50f,-40.0f);

            // верхняя часть левой стороны
            glVertex3f(0.0f,-0.5f,-40.0f);
            glVertex3f(30.0f, -0.5f, -57.0f);
            glVertex3f(0.0f, 4.0f, -57.0f);

            // верхняя часть правой стороны
            glVertex3f(0.0f, 4.0f, -57.0f);
            glVertex3f(-30.0f, -0.5f, -57.0f);
            glVertex3f(0.0f,-0.5f,-40.0f);

            // задняя нижняя часть хвоста
            glVertex3f(30.0f,-0.5f,-57.0f);
            glVertex3f(-30.0f, -0.5f, -57.0f);
            glVertex3f(0.0f, 4.0f, -57.0f);


            // Верхняя левая часть хвоста
            glColor3ub(255,0,0);
            glVertex3f(0.0f,0.5f,-40.0f);
            glVertex3f(3.0f, 0.5f, -57.0f);
            glVertex3f(0.0f, 25.0f, -65.0f);

            glVertex3f(0.0f, 25.0f, -65.0f);
            glVertex3f(-3.0f, 0.5f, -57.0f);
```

```
            glVertex3f(0.0f,0.5f,-40.0f);


            // Задняя часть горизонтального участка
            glVertex3f(3.0f,0.5f,-57.0f);
            glVertex3f(-3.0f, 0.5f, -57.0f);
            glVertex3f(0.0f, 25.0f, -65.0f);
      glEnd();

    glPopMatrix();

    // Отобразить результаты
    glutSwapBuffers();
    }

// Функция выполняет необходимую инициализацию
// в контексте визуализации
void SetupRC()
    {
    // Параметры света
    // Яркий белый свет
   // GLfloat ambientLight[] = { 1.0f, 1.0f, 1.0f,
1.0f };

      GLfloat ambientLight[] = { 0.5f,  0.5f,  0.5f,
1.0f };
    glEnable(GL_DEPTH_TEST);// Удаление скрытых час-
тей
    glEnable(GL_CULL_FACE); //Расчеты внутри самолета
не выполняются
glFrontFace(GL_CCW);// Многоугольники с обходом про-
тив часовой стрелки
                        //направлены наружу

    // Освещение материала
```

```
    glEnable(GL_LIGHTING);                  // Активиза-
ция освещения

    // В модели освещения задается использование рас-
    сеянного света,  //заданного в функции
    ambientLight[]

    glLightMo-
delfv(GL_LIGHT_MODEL_AMBIENT,ambientLight);

    glEnable(GL_COLOR_MATERIAL);  // Активизируется
согласование цветов

      //Рассеянный и диффузный цвета передней стороны
объектов
      //соответствуют тому, что указано в glColor

    glColorMaterial(GL_FRONT,GL_AMBIENT_AND_DIFFUSE);

    // Светло-синий цвет
    glClearColor(0.0f, 0.0f, 05.f,1.0f);
    }

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

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

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

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

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

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

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

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

      // Обновляется окно
    glutPostRedisplay();
    }


void ChangeSize(int w, int h)
    {
    GLfloat nRange = 80.0f;
    // Предотвращение деления на ноль
    if(h == 0)
        h = 1;

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

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

      // Устанавливает объем отсечения с помощью от-
секающих
      // плоскостей (левая, правая, нижняя, верхняя,
      // ближняя, дальняя)
```

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

    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    }

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

    return 0;
    }
```

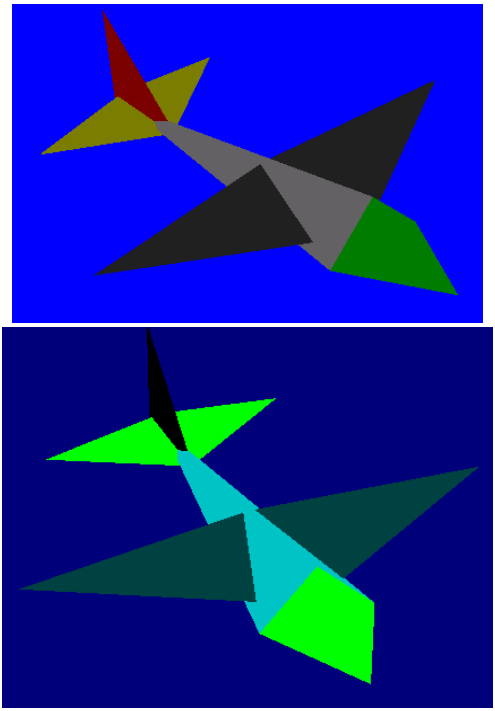

Рис. 25 – Результат выполнения при уменьшенной вдвое интенсивности источника света и задания других составляющих света 0.0f, 1.0f, 1.0f

Установка источника света

Теперь, когда ясны требования по "настройке" многоугольников, чтобы они получали свет и взаимодействовали с источником света, пришла пора включить свет. В листинге 3 приведен полный код программы. Часть процесса установки этой простой программы посвящена созданию источника света, который помещается слева вверху, немного позади наблюдателя. Источник света `GL_LIGHT0` имеет рассеянный и диффузный компонент, интенсивности которых заданы в массивах `amblentLight []` и `diffuseLight []`. В результате получается умеренно мощный белый источник света.

```
  GLfloat ambientLight[] = { 0.3f, 0.3f, 0.3f, 1.0f
};
  GLfloat diffuseLight[] = { 0.7f, 0.7f,  0.7f,
1.0f };
  ...
  ...
  // Устанавливается и активизируется источник света
0 glLightfv(GL_LIGHT0,GL_AMBIENT,ambientLight);
glLightfv(GL_LIGHT0,GL_DIFFUSE,diffuseLight);
  Наконец, включается источник света GL_LIGHT0.
  glEnable(GL_LIGHT0);
  С помощью приведенного ниже кода (находится в
функции ChangeSize) источник света размешается в про-
странстве.
  GLfloat lightPos[] = { -50.f,   50.0f,   100.0f,
1.0f };
  ...
  ...
  glLightfv(GL_LIGHT0,GL_POSITION,lightPos);
```

Здесь функция `lightPos []` содержит положение источника света. Последнее значение массива — 1.0; оно сообщает, что указанные координаты являются положением источника света. Если последнее значение массива равно 0.0, значит, источник света расположен в бесконечности в направлении, которое указано вектором, приведенным в этом массиве. Пока же отметим, что источники света подобны геометрическим объектам в том, что их можно перемещать по сцене с помощью матрицы наблюдения модели. Задействовав положение источника света в преобразовании наблюдения, гарантируется, что свет приходит с правильного направления вне зависимости от того, как преобразовыванп геометрия.

Листинг 3 –Установка контекста освещения и визуализации

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

```
#include <math.h>

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

typedef GLfloat GLTVector3[3];
// Значение скалярного вектора
void gltScaleVector(GLTVector3 vVector, const GLfloat
fScale)
    {
    vVector[0] *= fScale; vVector[1] *= fScale;
vVector[2] *= fScale;
    }

// Получает квадрат длины вектора
GLfloat gltGetVectorLengthSqrd(const GLTVector3
vVector)
    {
    return (vVector[0]*vVector[0]) +
(vVector[1]*vVector[1]) + (vVector[2]*vVector[2]);
    }

// Получает длину вектора
GLfloat gltGetVectorLength(const GLTVector3 vVector)
    {
    return
(GLfloat)sqrt(gltGetVectorLengthSqrd(vVector));
    }
// Нормируется вектор источника света
void gltNormalizeVector(GLTVector3 vNormal)
    {
    GLfloat fLength = 1.0f /
gltGetVectorLength(vNormal);
    gltScaleVector(vNormal, fLength);
    }
```

```
// Вычислить векторное произведение двух векторов
void gltVectorCrossProduct(const GLTVector3 vU, const
GLTVector3 vV, GLTVector3 vResult)
      {
      vResult[0] = vU[1]*vV[2] - vV[1]*vU[2];
      vResult[1] = -vU[0]*vV[2] + vV[0]*vU[2];
      vResult[2] = vU[0]*vV[1] - vV[0]*vU[1];
      }
// Вычесть один вектор из другого
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];
    }
// Даны три точки на плоскости направлены против ча-
совой стрелки, вычислить // нормаль
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);
    }


// Вызывается для рисования сцены
void RenderScene(void)
      {
```

```
      GLTVector3 vNormal;    // Память для вычисления
нормали к поверхности
      // Очистка окна текущим цветом очистки
      glClear(GL_COLOR_BUFFER_BIT |
GL_DEPTH_BUFFER_BIT);

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


      // Носовой конус - прямо вниз
        // Устанавливается цвет материала
      glColor3ub(128, 128, 128);
      glBegin(GL_TRIANGLES);
                glNormal3f(0.0f, -1.0f, 0.0f);
           glNormal3f(0.0f, -1.0f, 0.0f);
           glVertex3f(0.0f, 0.0f, 60.0f);
           glVertex3f(-15.0f, 0.0f, 30.0f);
           glVertex3f(15.0f,0.0f,30.0f);


                // Вершины для этой панели
                {
                GLTVector3 vPoints[3] = {{ 15.0f,
0.0f,  30.0f},
                                         { 0.0f,
15.0f, 30.0f},
                                         { 0.0f,
0.0f,  60.0f}};

                // Рассчет нормали поверхности
           gltGetNormalVector(vPoints[0], vPoints[1],
vPoints[2], vNormal);
```

```
            glNormal3fv(vNormal);
            glVertex3fv(vPoints[0]);
            glVertex3fv(vPoints[1]);
            glVertex3fv(vPoints[2]);
                 }


                 {
                 GLTVector3 vPoints[3] = {{ 0.0f,
0.0f, 60.0f },
                                          { 0.0f,
15.0f, 30.0f },
                                          { -15.0f,
0.0f, 30.0f }};

             gltGetNormalVector(vPoints[0],
vPoints[1], vPoints[2], vNormal);
                 glNormal3fv(vNormal);
            glVertex3fv(vPoints[0]);
            glVertex3fv(vPoints[1]);
            glVertex3fv(vPoints[2]);
                 }


                 // Тело поверхности
//////////////////////
                 {
                 GLTVector3 vPoints[3] = {{ -15.0f,
0.0f, 30.0f },
                                          { 0.0f,
15.0f, 30.0f },
                                  { 0.0f, 0.0f, -56.0f
}};

                 gltGetNormalVector(vPoints[0],
vPoints[1], vPoints[2], vNormal);
```

```
                glNormal3fv(vNormal);
            glVertex3fv(vPoints[0]);
            glVertex3fv(vPoints[1]);
            glVertex3fv(vPoints[2]);
                }

                {
                GLTVector3 vPoints[3] = {{ 0.0f,
0.0f, -56.0f },
                                         { 0.0f,
15.0f, 30.0f },
                                         {
15.0f,0.0f,30.0f }};

                gltGetNormalVector(vPoints[0],
vPoints[1], vPoints[2], vNormal);
                glNormal3fv(vNormal);
            glVertex3fv(vPoints[0]);
            glVertex3fv(vPoints[1]);
            glVertex3fv(vPoints[2]);
                }


            glNormal3f(0.0f, -1.0f, 0.0f);
            glVertex3f(15.0f,0.0f,30.0f);
            glVertex3f(-15.0f, 0.0f, 30.0f);
            glVertex3f(0.0f, 0.0f, -56.0f);


////////////////////////////////////////////
                // Left wing
                // Large triangle for bottom of wing
                {
                GLTVector3 vPoints[3] = {{
0.0f,2.0f,27.0f },
```

```
                                        { -60.0f,
2.0f, -8.0f },
                                        { 60.0f,
2.0f, -8.0f }};

                gltGetNormalVector(vPoints[0],
vPoints[1], vPoints[2], vNormal);
                glNormal3fv(vNormal);
           glVertex3fv(vPoints[0]);
           glVertex3fv(vPoints[1]);
           glVertex3fv(vPoints[2]);
                }


                {
                GLTVector3 vPoints[3] = {{ 60.0f,
2.0f, -8.0f},
                           {0.0f, 7.0f, -8.0f},
                           {0.0f,2.0f,27.0f }};

                gltGetNormalVector(vPoints[0],
vPoints[1], vPoints[2], vNormal);
                glNormal3fv(vNormal);
           glVertex3fv(vPoints[0]);
           glVertex3fv(vPoints[1]);
           glVertex3fv(vPoints[2]);
                }

                {
                GLTVector3 vPoints[3] = {{60.0f,
2.0f, -8.0f},
                           {-60.0f, 2.0f, -8.0f},
                           {0.0f,7.0f,-8.0f }};

                gltGetNormalVector(vPoints[0],
vPoints[1], vPoints[2], vNormal);
```

```
                glNormal3fv(vNormal);
            glVertex3fv(vPoints[0]);
            glVertex3fv(vPoints[1]);
            glVertex3fv(vPoints[2]);
                }

                {
                GLTVector3 vPoints[3] =
{{0.0f,2.0f,27.0f},
                                            {0.0f, 7.0f,
-8.0f},
                                            {-60.0f,
2.0f, -8.0f}};

                gltGetNormalVector(vPoints[0],
vPoints[1], vPoints[2], vNormal);
                glNormal3fv(vNormal);
            glVertex3fv(vPoints[0]);
            glVertex3fv(vPoints[1]);
            glVertex3fv(vPoints[2]);
                }


                // Хвостовая
часть//////////////////////////////
                // Нижняя часть стабилизатора
            glNormal3f(0.0f, -1.0f, 0.0f);
            glVertex3f(-30.0f, -0.50f, -57.0f);
            glVertex3f(30.0f, -0.50f, -57.0f);
            glVertex3f(0.0f,-0.50f,-40.0f);

                {
                GLTVector3 vPoints[3] = {{ 0.0f,-
0.5f,-40.0f },
                                {30.0f, -0.5f, -57.0f},
                                {0.0f, 4.0f, -57.0f }};
```

```
                gltGetNormalVector(vPoints[0],
vPoints[1], vPoints[2], vNormal);
                glNormal3fv(vNormal);
            glVertex3fv(vPoints[0]);
            glVertex3fv(vPoints[1]);
            glVertex3fv(vPoints[2]);
                }


                {
                GLTVector3 vPoints[3] = {{ 0.0f,
4.0f, -57.0f },
                                { -30.0f, -0.5f, -57.0f
},
                                { 0.0f,-0.5f,-40.0f }};

                gltGetNormalVector(vPoints[0],
vPoints[1], vPoints[2], vNormal);
                glNormal3fv(vNormal);
            glVertex3fv(vPoints[0]);
            glVertex3fv(vPoints[1]);
            glVertex3fv(vPoints[2]);
                }


                {
                GLTVector3 vPoints[3] = {{ 30.0f,-
0.5f,-57.0f },
                                { -30.0f, -0.5f, -57.0f
},
                                { 0.0f, 4.0f, -57.0f }};

                gltGetNormalVector(vPoints[0],
vPoints[1], vPoints[2], vNormal);
                glNormal3fv(vNormal);
```

```
            glVertex3fv(vPoints[0]);
            glVertex3fv(vPoints[1]);
            glVertex3fv(vPoints[2]);
                 }

                 {
                 GLTVector3 vPoints[3] = {{
0.0f,0.5f,-40.0f },
                              { 3.0f, 0.5f, -57.0f },
                              { 0.0f, 25.0f, -65.0f
}};

                 gltGetNormalVector(vPoints[0],
vPoints[1], vPoints[2], vNormal);
                 glNormal3fv(vNormal);
            glVertex3fv(vPoints[0]);
            glVertex3fv(vPoints[1]);
            glVertex3fv(vPoints[2]);
                 }


                 {
                 GLTVector3 vPoints[3] = {{ 0.0f,
25.0f, -65.0f },
                              { -3.0f, 0.5f, -57.0f},
                              { 0.0f,0.5f,-40.0f }};

                 gltGetNormalVector(vPoints[0],
vPoints[1], vPoints[2], vNormal);
                 glNormal3fv(vNormal);
            glVertex3fv(vPoints[0]);
            glVertex3fv(vPoints[1]);
            glVertex3fv(vPoints[2]);
                 }

                 {
```

```
                GLTVector3 vPoints[3] = {{
3.0f,0.5f,-57.0f },
                                { -3.0f, 0.5f, -57.0f },
                                { 0.0f, 25.0f, -65.0f
}};

                gltGetNormalVector(vPoints[0],
vPoints[1], vPoints[2], vNormal);
                glNormal3fv(vNormal);
            glVertex3fv(vPoints[0]);
            glVertex3fv(vPoints[1]);
            glVertex3fv(vPoints[2]);
                }


        glEnd();

      // Восстановление состояния матрицы
      glPopMatrix();
      // Отобразить результаты
      glutSwapBuffers();
      }

// Эта функция выполняет необходимую инициализацию в
контексте
// визуализации.
void SetupRC()
    {
    // Координаты
    GLfloat  ambientLight[] = { 0.3f, 0.3f, 0.3f,
1.0f };
    GLfloat  diffuseLight[] = { 0.7f, 0.7f, 0.7f,
1.0f };

    glEnable(GL_DEPTH_TEST);// Удалить скрытых по-
верхности
```

```
    glFrontFace(GL_CCW);// Многоугольники с обходом
против часовой стрелки
                        // направлены вперед

    glEnable(GL_CULL_FACE);// Внутри самолета расчеты
не производятся

    // Активизируется освещение
    glEnable(GL_LIGHTING);

    // Устанавливается и активизируется источник све-
та 0
    glLightfv(GL_LIGHT0,GL_AMBIENT,ambientLight);
    glLightfv(GL_LIGHT0,GL_DIFFUSE,diffuseLight);
    glEnable(GL_LIGHT0);

    // Активизирует согласование цветов
    glEnable(GL_COLOR_MATERIAL);

    // Свойства материалов соответствуют кодам
glColor
    glColorMaterial(GL_FRONT,
GL_AMBIENT_AND_DIFFUSE);

    // Светло-голубой цвет фона
    glClearColor(0.0f, 0.0f, 1.0f, 1.0f );
    }

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

    if(key == GLUT_KEY_DOWN)
```

```
        xRot += 5.0f;

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

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

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

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

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

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

    // Обновление окна
    glutPostRedisplay();
    }


//////////////////////////////////////////////////////
/////
// Обновляется проекция и положение источника света
void ChangeSize(int w, int h)
    {
    GLfloat fAspect;
    GLfloat lightPos[] = { -50.f, 50.0f, 100.0f, 1.0f
};

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

```
        h = 1;

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

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

    fAspect = (GLfloat) w / (GLfloat) h;
    gluPerspective(45.0f, fAspect, 1.0f, 225.0f);

    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();

    glLightfv(GL_LIGHT0,GL_POSITION,lightPos);
    glTranslatef(0.0f, 0.0f, -150.0f);
    }

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

    return 0;
    }
```

Устанавливаем свойства материала

Внимательно изучая листинг 3, можно заметить, что активизировано согласование цвета, причем согласовываются рассеивающее и диффузное свойства лицевой поверхности многоугольника.

```
  // Активизируем согласование цвета
  glEnable(GL_COLOR_MATERIAL);
  // Свойства материалов согласуются с кодами
glColor                    glColorMaterial(GL_FRONT,
GL_AMBIENT_AND_DIFFUSE);
```

Устанавливаем многоугольники

Код визуализации из первых двух примеров существенно изменился, чтобы поддерживать новую модель освещения. В листинге 4 приведена выборка из функции (программа LITJET).

Листинг 4 – Фрагмент кода, в котором устанавливаются цвета и вычисляются нормали и многоугольники

```
GLTVector vNormal;
// Здесь хранятся рассчитанные нормали к поверхностям
...
...
//Устанавливается цвет материала
glColor3ub(128,   128,  128);
glBegin(GL_TRIANGLES);
    glNormal3f(0.0f,  -1.0f, 0.0f);
    glNormal3f(0.0f,  -1.0f, 0.0f);
    glVertex3f(0.0f,   0.0f, 60.0f);
    glVertex3f(-15.0f, 0.0f, 30.0f);
    glVertex3f(15.0f,  0.0f, 30.0f);
    // Вершины панели
{    GLTVector vPoints[3]  = {{15.0f, 0.0f, 30.0f),
                              { 0.0f, 15.0f, 30.0f},
                              { 0.0f, 0.0f, 60.0f}};
// Рассчитывается нормаль плоскости
```

```
gltGetNormalVector(vPoints[0],vPoints[1].vPoints[2],v
Normal); glNormal3fv(vNormal);
glVertex3fv(vPoints(0]); glVertex3fv(vPoints[1]);
glVertex3fv(vPoints[2]);
}
{    GLTVector vPoints[3]  = {{  0.0f, 0.0f,  60.0f),
                              {  0.0f, 15.0f, 30.0f},
                              {-15.0f, 0.0f,
30.0f}};
gltGetNormalVector(vPoints[0],vPoints[1},vPoints[2],v
Normal);
glNormal3fv(vNormal);
glVertex3fv(vPoints[0]) ;
glVertex3fv(vPoints[1]);
glVertex3fv(vPoints[2]);
}
```

Обратите внимание на то, что вектор нормали рассчитывается, используя функцию `gltGetNormalVector` из `glTools`. Кроме того, свойства материалов теперь соответствуют цветам, установленным с помощью `glColor`. Еще один момент, на который стоит обратить внимание: не все треугольники взяты в блок из функций `glBegin/glEnd`. Можно один раз указать, что рисуете треугольники, и любые три вершины, которые будут приведены после этого, будут расцениваться как вершины нового треугольника, пока это не изменено с помощью функции `glEnd`. При очень большом числе многоугольников данная техника может существенно повысить производительность, устранив множество ненужных вызовов процедур и настроек групп примитивов.

На рис. 26 показан результат выполнения завершенной программы. Теперь весь самолет составлен из треугольников одного оттенка серого, а не из фигур разных цветов. Цвет изменен, чтобы на поверхности было легче наблюдать эффекты освещения. Несмотря на то, что поверхность имеет один сплошной цвет, благодаря освещению попрежнему можно видеть форму. Поворачивая самолетик с помощью клавиш со стрелками, можно наблюдать различные эффекты затене-

ния, меняющиеся по мере того, как модель движется и взаимодействует со светом. Наиболее очевидным способом повышения производительности данного кода является заблаговременный расчет всех векторов нормали и запись их для последующего использования в функции `RenderScene`.

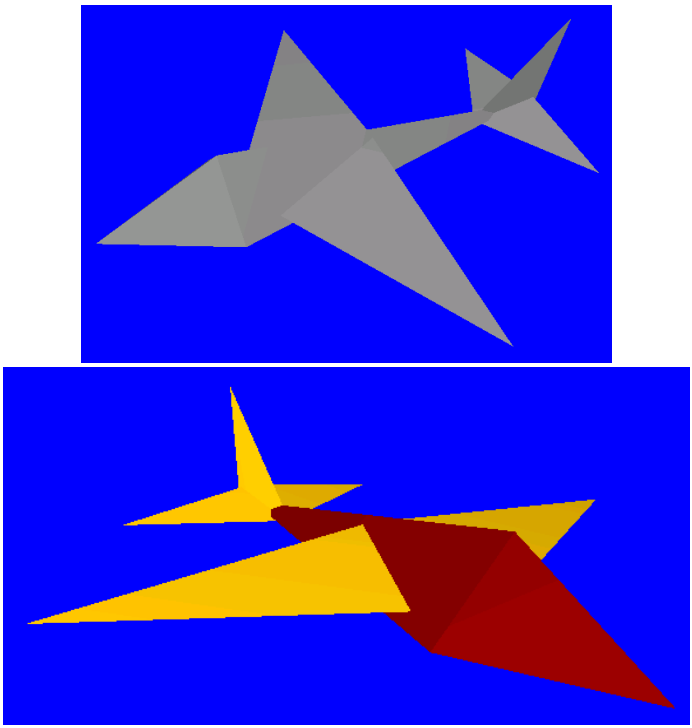

Рис. 26 – Результат выполнения программы листинг 3.

Коэффициент зеркального отражения

Как отмечалось ранее, сильный отраженный свет и высокая отражательная способность делают цвета объекта светлее. В примере сильный (максимальной интенсивности) зеркальный свет и очень большая (максимальная) отражательная способность привели к тому, что самолет кажется белым или серым за исключением точек поверхности, удаленных от источника света (эти точки кажутся черными и

неосвещенными). Чтобы смягчить этот эффект, после задания компонента зеркального отражения введем в код следующую строку:

```
glMateriali(GL_FRONT,GL_SHININESS,128);
```

Свойство `GL_SHININESS` задает коэффициент зеркального отражения материала — насколько маленьким и сфокусированным является блик. Значение 0 соответствует несфокусированному "зайчику", и именно так получается равномерное осветление цветов всего многоугольника. Заданием этого значения уменьшается размер и сильнее фокусируется яркое пятно света. Чем больше значение, тем ярче и блестящее поверхность. Во всех реализациях OpenGL диапазон значений этого параметра составляет 1-128.

В листинге 5 показан новый код процедуры `SetupRC` в контексте предыдущей программы. Результат выполнения данной программы показан на рис. 27.

Листинг 5 – Код настройки из программы, дающий "зайчики" на поверхности самолета

```
// Эта функция выполняет необходимую инициализацию в
контексте
// визуализации. В данном случае она устанавливает и
инициализирует
// освещение на сцене
void SetupRC()
    {
    // Коды и координаты источников света
    GLfloat ambientLight[] = { 0.3f, 0.3f,  0.3f,
1.0f };
    GLfloat diffuseLight[] = { 0.7f, 0.7f,  0.7f,
1.0f };
    GLfloat specular[] = { 1.0f,  1.0f,  1.0f,
1.0f};
    GLfloat specref[] = { 1.0f,   1.0f,  1.0f,   1.0f
};
```

```
    glEnable(GL_DEPTH_TEST);    // Удаление скрытых
поверхностей

     glFrontFace(GL_CCW);
          // Многоугольники с обходом против часовой
стрелки
          // направлены вперед
          glEnable(GL_CULL_FACE);
// Внутри самолета расчеты не производятся
// Активизируется освещение
glEnable(GL_LIGHTING);
// Устанавливается и активизируется источник света 0
     glLightfv(GL_LIGHT0,GL_AMBIENT,ambientLight);
     glLightfv(GL_LIGHT0,GL_DIFFUSE,diffuseLight) ;
     glLightfv(GL_LIGHT0, GL__SPECULAR, specular) ;
     glEnable(GL_LIGHT0);
// Активизируем согласование цвета
     glEnable(GL_COLOR_MATERIAL);
// Свойства материалов соответствуют кодам glColor
     glColorMaterial(GL_FRONT,
GL_AMBIENT_AND_DIFFUSE);
// С этого момента все материалы получают способность
// отражать блики
     glMaterialfv(GL_FRONT, GL_SPECULAR, specref);
     glMateriali(GL_FRONT,GL_SHININESS,128);
// Светло-синий фон
     glClearColor(0.0f,   0.0f,  1.0f,  1.0f );
     }

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

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

```
typedef GLfloat GLTVector3[3];
// Значение скалярного вектора
void gltScaleVector(GLTVector3 vVector, const GLfloat
fScale)
    {
    vVector[0] *= fScale; vVector[1] *= fScale;
vVector[2] *= fScale;
    }

// Получает квадрат длины вектора
GLfloat gltGetVectorLengthSqrd(const GLTVector3
vVector)
    {
    return (vVector[0]*vVector[0]) +
(vVector[1]*vVector[1]) + (vVector[2]*vVector[2]);
    }

// Получает длину вектора
GLfloat gltGetVectorLength(const GLTVector3 vVector)
    {
    return
(GLfloat)sqrt(gltGetVectorLengthSqrd(vVector));
    }
// Нормируется вектор источника света
void gltNormalizeVector(GLTVector3 vNormal)
    {
    GLfloat fLength = 1.0f /
gltGetVectorLength(vNormal);
    gltScaleVector(vNormal, fLength);
    }
// Вычислить векторное произведение двух векторов
void gltVectorCrossProduct(const GLTVector3 vU, const
GLTVector3 vV, GLTVector3 vResult)
     {
     vResult[0] = vU[1]*vV[2] - vV[1]*vU[2];
```

```
      vResult[1] = -vU[0]*vV[2] + vV[0]*vU[2];
      vResult[2] = vU[0]*vV[1] - vV[0]*vU[1];
      }
// Вычесть один вектор из другого
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];
    }
// Даны три точки на плоскости направлены против ча-
совой стрелки, вычислить // нормаль
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);
    }

// Вызывается для рисования сцены
void RenderScene(void)
      {
      GLTVector3 vNormal;    // Память для вычисления
нормали к поверхности

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

```
      // Записываем состояние матрицы и выполняем по-
ворот
      glPushMatrix();
      glRotatef(xRot, 1.0f, 0.0f, 0.0f);
      glRotatef(yRot, 0.0f, 1.0f, 0.0f);


      // Носовой конус - прямо вниз
        // Устанавливается цвет материала
      glColor3ub(128, 128, 128);
      glBegin(GL_TRIANGLES);
                glNormal3f(0.0f, -1.0f, 0.0f);
           glNormal3f(0.0f, -1.0f, 0.0f);
           glVertex3f(0.0f, 0.0f, 60.0f);
           glVertex3f(-15.0f, 0.0f, 30.0f);
           glVertex3f(15.0f,0.0f,30.0f);

                     // Вершины для этой панели
                {
                GLTVector3 vPoints[3] = {{ 15.0f,
0.0f,  30.0f},
                                         { 0.0f,
15.0f, 30.0f},
                                         { 0.0f,
0.0f,  60.0f}};

                // Рассчет нормали поверхности
                gltGetNormalVector(vPoints[0],
vPoints[1], vPoints[2], vNormal);
           glNormal3fv(vNormal);
           glVertex3fv(vPoints[0]);
           glVertex3fv(vPoints[1]);
           glVertex3fv(vPoints[2]);
                }
```

```
                {
                GLTVector3 vPoints[3] = {{ 0.0f,
0.0f, 60.0f },
                                         { 0.0f,
15.0f, 30.0f },
                                         { -15.0f,
0.0f, 30.0f }};

                gltGetNormalVector(vPoints[0],
vPoints[1], vPoints[2], vNormal);
                glNormal3fv(vNormal);
            glVertex3fv(vPoints[0]);
            glVertex3fv(vPoints[1]);
            glVertex3fv(vPoints[2]);
                }


                // Тело поверхности
//////////////////////
                {
                GLTVector3 vPoints[3] = {{ -15.0f,
0.0f, 30.0f },
                                         { 0.0f,
15.0f, 30.0f },
                                 { 0.0f, 0.0f, -56.0f
}};

                gltGetNormalVector(vPoints[0],
vPoints[1], vPoints[2], vNormal);
                glNormal3fv(vNormal);
            glVertex3fv(vPoints[0]);
            glVertex3fv(vPoints[1]);
            glVertex3fv(vPoints[2]);
                }

                {
```

```
                GLTVector3 vPoints[3] = {{ 0.0f,
0.0f, -56.0f },
                                         { 0.0f,
15.0f, 30.0f },
                                         {
15.0f,0.0f,30.0f }};

                gltGetNormalVector(vPoints[0],
vPoints[1], vPoints[2], vNormal);
                glNormal3fv(vNormal);
           glVertex3fv(vPoints[0]);
           glVertex3fv(vPoints[1]);
           glVertex3fv(vPoints[2]);
                }


           glNormal3f(0.0f, -1.0f, 0.0f);
           glVertex3f(15.0f,0.0f,30.0f);
           glVertex3f(-15.0f, 0.0f, 30.0f);
           glVertex3f(0.0f, 0.0f, -56.0f);


////////////////////////////////////////////
                // Левое крыло
                // Большой треугольник для нижней
части крыла                {
                GLTVector3 vPoints[3] = {{
0.0f,2.0f,27.0f },
                                         { -60.0f,
2.0f, -8.0f },
                                         { 60.0f,
2.0f, -8.0f }};

                gltGetNormalVector(vPoints[0],
vPoints[1], vPoints[2], vNormal);
                glNormal3fv(vNormal);
```

```
            glVertex3fv(vPoints[0]);
            glVertex3fv(vPoints[1]);
            glVertex3fv(vPoints[2]);
                 }


                 {
                 GLTVector3 vPoints[3] = {{ 60.0f,
2.0f, -8.0f},
                                {0.0f, 7.0f, -8.0f},
                                {0.0f,2.0f,27.0f }};

                 gltGetNormalVector(vPoints[0],
vPoints[1], vPoints[2], vNormal);
                 glNormal3fv(vNormal);
            glVertex3fv(vPoints[0]);
            glVertex3fv(vPoints[1]);
            glVertex3fv(vPoints[2]);
                 }

                 {
                 GLTVector3 vPoints[3] = {{60.0f,
2.0f, -8.0f},
                                {-60.0f, 2.0f, -8.0f},
                                {0.0f,7.0f,-8.0f }};

                 gltGetNormalVector(vPoints[0],
vPoints[1], vPoints[2], vNormal);
                 glNormal3fv(vNormal);
            glVertex3fv(vPoints[0]);
            glVertex3fv(vPoints[1]);
            glVertex3fv(vPoints[2]);
                 }

                 {
```

```
                GLTVector3 vPoints[3] =
{{0.0f,2.0f,27.0f},
                                                {0.0f, 7.0f,
-8.0f},
                                                {-60.0f,
2.0f, -8.0f}};

                gltGetNormalVector(vPoints[0],
vPoints[1], vPoints[2], vNormal);
                glNormal3fv(vNormal);
            glVertex3fv(vPoints[0]);
            glVertex3fv(vPoints[1]);
            glVertex3fv(vPoints[2]);
                }


                // Хвостовая
часть//////////////////////////////
                // Нижняя часть стабилизатора
            glNormal3f(0.0f, -1.0f, 0.0f);
            glVertex3f(-30.0f, -0.50f, -57.0f);
            glVertex3f(30.0f, -0.50f, -57.0f);
            glVertex3f(0.0f,-0.50f,-40.0f);

                {
                GLTVector3 vPoints[3] = {{ 0.0f,-
0.5f,-40.0f },
                                {30.0f, -0.5f, -57.0f},
                                {0.0f, 4.0f, -57.0f }};

                gltGetNormalVector(vPoints[0],
vPoints[1], vPoints[2], vNormal);
                glNormal3fv(vNormal);
            glVertex3fv(vPoints[0]);
            glVertex3fv(vPoints[1]);
            glVertex3fv(vPoints[2]);
```

```
                }


                {
                GLTVector3 vPoints[3] = {{ 0.0f,
4.0f, -57.0f },
                                { -30.0f, -0.5f, -57.0f
},
                                { 0.0f,-0.5f,-40.0f }};

                gltGetNormalVector(vPoints[0],
vPoints[1], vPoints[2], vNormal);
                glNormal3fv(vNormal);
           glVertex3fv(vPoints[0]);
           glVertex3fv(vPoints[1]);
           glVertex3fv(vPoints[2]);
                }


                {
                GLTVector3 vPoints[3] = {{ 30.0f,-
0.5f,-57.0f },
                                { -30.0f, -0.5f, -57.0f
},
                                { 0.0f, 4.0f, -57.0f }};

                gltGetNormalVector(vPoints[0],
vPoints[1], vPoints[2], vNormal);
                glNormal3fv(vNormal);
           glVertex3fv(vPoints[0]);
           glVertex3fv(vPoints[1]);
           glVertex3fv(vPoints[2]);
                }

                {
```

```
                GLTVector3 vPoints[3] = {{
0.0f,0.5f,-40.0f },
                                { 3.0f, 0.5f, -57.0f },
                                { 0.0f, 25.0f, -65.0f
}};

                gltGetNormalVector(vPoints[0],
vPoints[1], vPoints[2], vNormal);
                glNormal3fv(vNormal);
           glVertex3fv(vPoints[0]);
           glVertex3fv(vPoints[1]);
           glVertex3fv(vPoints[2]);
                }


                {
                GLTVector3 vPoints[3] = {{ 0.0f,
25.0f, -65.0f },
                                { -3.0f, 0.5f, -57.0f},
                                { 0.0f,0.5f,-40.0f }};

                gltGetNormalVector(vPoints[0],
vPoints[1], vPoints[2], vNormal);
                glNormal3fv(vNormal);
           glVertex3fv(vPoints[0]);
           glVertex3fv(vPoints[1]);
           glVertex3fv(vPoints[2]);
                }

                {
                GLTVector3 vPoints[3] = {{
3.0f,0.5f,-57.0f },
                                { -3.0f, 0.5f, -57.0f },
                                { 0.0f, 25.0f, -65.0f
}};
```

```
                gltGetNormalVector(vPoints[0],
vPoints[1], vPoints[2], vNormal);
                glNormal3fv(vNormal);
           glVertex3fv(vPoints[0]);
           glVertex3fv(vPoints[1]);
           glVertex3fv(vPoints[2]);
                }

        glEnd();

      // Восстановление состояния матрицы
      glPopMatrix();
      // Отобразить результаты
      glutSwapBuffers();
      }

// Эта функция выполняет необходимую инициализацию в
контексте
// визуализации.
void SetupRC()
    {
    // Координаты иосвещение
    GLfloat  ambientLight[] = { 0.3f, 0.3f, 0.3f,
1.0f };
    GLfloat  diffuseLight[] = { 0.7f, 0.7f, 0.7f,
1.0f };
    GLfloat  specular[] = { 1.0f, 1.0f, 1.0f, 1.0f };
    GLfloat  specref[] = { 1.0f, 1.0f, 1.0f, 1.0f };

    glEnable(GL_DEPTH_TEST);// Удалить скрытых по-
верхности
    glFrontFace(GL_CCW); // Многоугольники с обходом
против часовой стрелки
                         // направлены вперед
    glEnable(GL_CULL_FACE); // Внутри самолета рас-
четы не производятся
```

```
    // Активизируется освещение
    glEnable(GL_LIGHTING);

    // Устанавливается и активизируется источник све-
та 0
    glLightfv(GL_LIGHT0,GL_AMBIENT,ambientLight);
    glLightfv(GL_LIGHT0,GL_DIFFUSE,diffuseLight);
    glLightfv(GL_LIGHT0,GL_SPECULAR, specular);
    glEnable(GL_LIGHT0);

    // Активизирует согласование цветов
    glEnable(GL_COLOR_MATERIAL);

// Свойства материалов соответствуют кодам glColor
glColorMaterial(GL_FRONT, GL_AMBIENT_AND_DIFFUSE);

    // Все материалы в дальнейшем имеют полное зер-
    кальное отражение с высоким // блеском
    glMaterialfv(GL_FRONT, GL_SPECULAR, specref);
    glMateriali(GL_FRONT, GL_SHININESS, 128);

    // Светло-голубой фон
    glClearColor(0.0f, 0.0f, 1.0f, 1.0f );
    }

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

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

    if(key == GLUT_KEY_LEFT)
```

```
        yRot -= 5.0f;

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

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

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

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

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

    // Refresh the Window
    glutPostRedisplay();
    }


////////////////////////////////////////////////////
/////
// Обновляется проекция и положение источника света
void ChangeSize(int w, int h)
    {
    GLfloat fAspect;
    GLfloat lightPos[] = { -50.f, 50.0f, 100.0f, 1.0f
};

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

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

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

    fAspect = (GLfloat) w / (GLfloat) h;
    gluPerspective(45.0f, fAspect, 1.0f, 225.0f);

    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();

    glLightfv(GL_LIGHT0,GL_POSITION,lightPos);
    glTranslatef(0.0f, 0.0f, -150.0f);
    }

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

    return 0;
    }
```

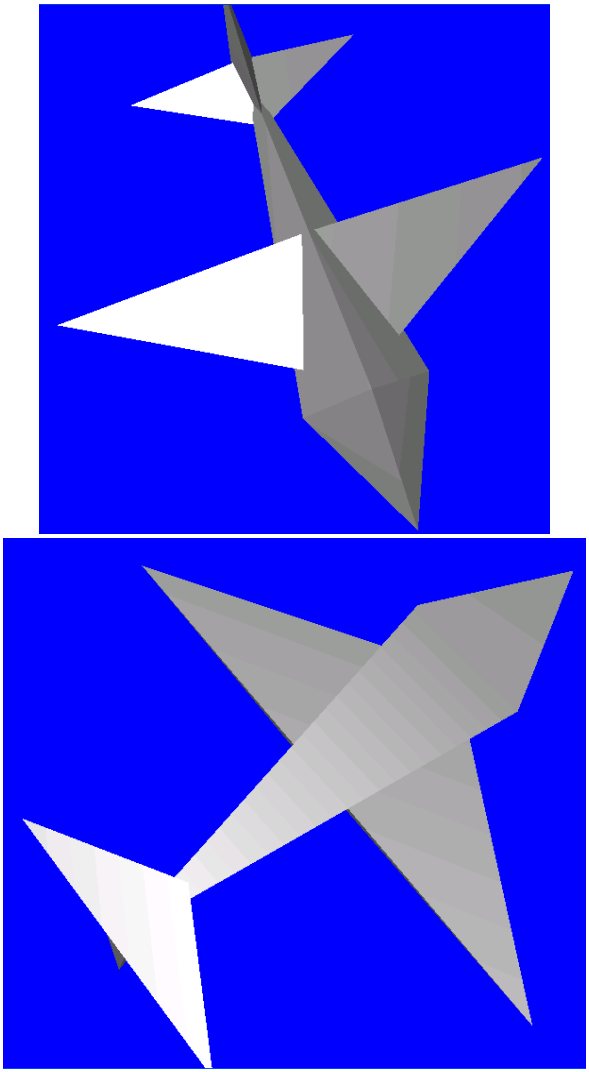

Рис. 27 – Результат выполнения программы

Собираем все вместе

В приложении будет продемонстрировано использование нормалей для создания иллюзии гладкой поверхности, перемещение источника света по сцене и создание точечного источника света.

Все сказанное реализовано в программе-примере. С помощью функции glutSolidSphere создана сплошная сфера в центре отображаемого объема. На этой сфере зажгли источник света, который можно перемещать. Кроме того, изменили “гладкость" нормалей и продемонстрировали некоторые ограничения модели освещения OpenGL.

До этого момента задавали положение источника света (с помощью функции glLight) так

```
// Массив, задающий положение
GLfloat lightPos[] = { 0.0f, 150.0f, 150.0f,
1.0f };
… …
// Устанавливается положение источника света
glLightfv(GL_LIGHT0, GL_POSITION, lightPos);
```

Массив lightPos[] содержит координаты x, y и z реального положения источника света на сцене или направления, с которого поступает свет. Последнее значение (в данном случае - 1.0) указывает, что в этой точке действительно расположен источник света. По умолчанию свет равномерно излучается по всем направлениям, но это можно изменить, задав эффект прожекторного освещения.

Создание прожектора

Создание прожектора не отличается от создания любого другого локального источника света. В коде, приведенном в листинге 6. показана функция SetupRC. Здесь в центре окна помещается синяя сфера и создастся прожектор, который можно перемещать вертикально с помощью клавиш со стрелками вверх и вниз и горизонтально — с помощью стрелок вправо и влево. При удалении прожектора от поверхности сферы за ним по поверхности следует яркий блик.

Листинг 6 – Настройка освещения

```
// Коды и координаты источников света
```

```
    GLfloat lightPos[] = { 0.0f, 0.0f, 75.0f, 1.0f
};
    GLfloat specular[] = { 1.0f, 1.0f, 1.0f, 1.0f);
    GLfloat specref[] = { 1.0f, 1.0f, 1.0f, 1.0f };
    GLfloat ambientLight[] = { 0.5f, 0.5f, 0.5f,
1.0f);
    GLfloat spotDir[] = { 0.0f, 0.0f, -1.0f );
    // Эта функция выполняет необходимую инициализа-
цию
    // в контексте визуализации. В данном случае она
    // устанавливает и инициализирует освещение на
сцене
    void SetupRC()
    {
    glEnable(GL_DEPTH_TEST);
    //Удаление скрытых поверхностей
    glFrontFace(GL_CCW);
    // Многоугольники с обходом против часовой
стрелки
    // направлены вперед

    glEnable(GL_CULL_FACE);
    // Задние поверхности не отображаются

    // Активизируется освещение
    glEnable(GL_LIGHTING);
    // Устанавливается и активизируется источник
света 0
    // Создается слабое рассеянное освещение, чтобы
    // можно было видеть объекты
    glLightModelfv(GL_LlGHT_MODEL_AMBIENT, ambien-
tLight);
    // Источник света имеет только диффузный и отра-
жательный
    // компоненты
    glLightfv(GL_LIGHT0,GL_DIFFUSE, ambientLight);
```

```
    glLightfv(GL_LIGHTO,GL_SP£CULAR,specular);
    glLightfv(GL_LIGHT0,GL_POSITION,lightPos);
    // Прожекторные эффекты
    // Угол конуса освещения составляет 60 градусов
    glLightf(GL_LIGHT0,GL_SPOT_CUTOFF,60.0f);
    // Активизируется источник света
    glEnable(GL_LIGHT0);
    // Активизируем согласование цветов
    glEnable(GL_COLOR_MATERlAL);
    // Свойства материалов соответствуют кодам
glColor
    glColorMaterial (GL_FRONT,
GL_AMBIENT_AND_DIFFUSE) ;
    // С этого момента все материала получают спо-
собность
    // отражать блики
    glMaterialfv(GL_FRONT, GL_SPECULAR, specref) ;
    glMaterial (GL_FRONT, GL_SHININESS, 128);
    // Черный фон
    glClearColor(0.0f, 0.0f, 0.0f, l.0f );
    }
```

Прожектором источник света делает следующая строка кода

```
// Прожекторные эффекты
// Угол конуса освещения составляет 60 градусов
glLightf(GL_LIGHT0,GL_SPOT_CUTOFF,60.0f);
```

Значение `GL_SPOT_CUTOFF` задаст угол конуса света, излучаемого прожектором (угол между осью и краем конуса). Для нормального локального источника света это значение равно 180°, так что свет не образует конуса. Фактически при создании прожекторных эффектов допускаются только углы от 0 до 90°. Прожектор излучает конус света, а объекты вне этого конуса не освещаются. На рис. 28 показано, как угол связан с шириной основания конуса.

Рисование прожектора

Если на сцене помещается луч прожектора, нужна точка, из которой исходит этот луч. То, что в некоторой точке пространства имеется источник света, не означает, что в этой точке яркое пятно. В программе SPOT в месте, где располагается источник-прожектор, поместили красный конус, обозначив положение этого источника света. Внутрь конуса поместили яркую желтую сферу, имитирующую электрическую лампочку.

В данном примере имеется всплывающее меню, которое использовали для демонстрации нескольких моментов. Меню содержит позиции выбора плоского и гладкого затенения и создания сферы с низкой, средней и высокой степенью "мозаичности" (малое, среднее или большое число многоугольников, составляющих поверхность).

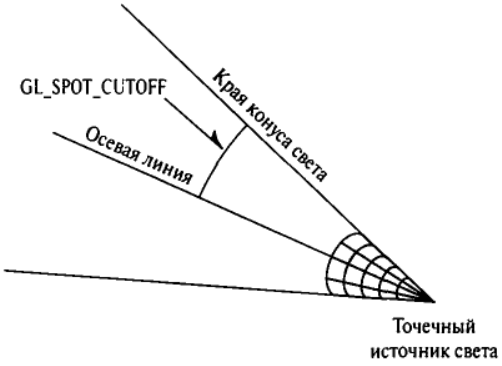

Рис. 28 – Угол конуса света (прожектор)

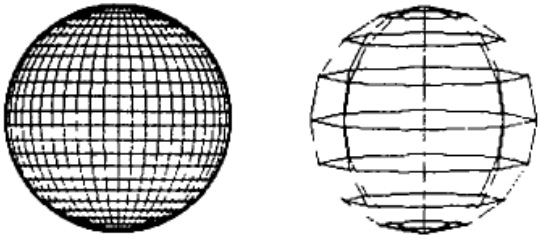

Рис. 29 – Слева показана сфера, собранная из мелких фрагментов, справа — сфера, полученная всего из нескольких многоугольников

Мозаичное представление заключается в разбиении сетки многоугольников на более мелкую сетку (больше вершин). На рис. 29 показано каркасное представление сферы с очень мелкой мозаикой рядом со сферой, составленной всего из нескольких многоугольников

На рис. 30 пример показан в исходном состоянии с прожектором, немного смещенным в одну сторону. Сфера состоит из нескольких многоугольников с плоским затенением. В Windows для открывания всплывающего меню используйте правую кнопку мыши, а с помощью этого меню можно переключаться между плоским затенением и низким, средним и высоким мозаичным представлением сферы. Полный код визуализации сцены приводится в листинге 7.

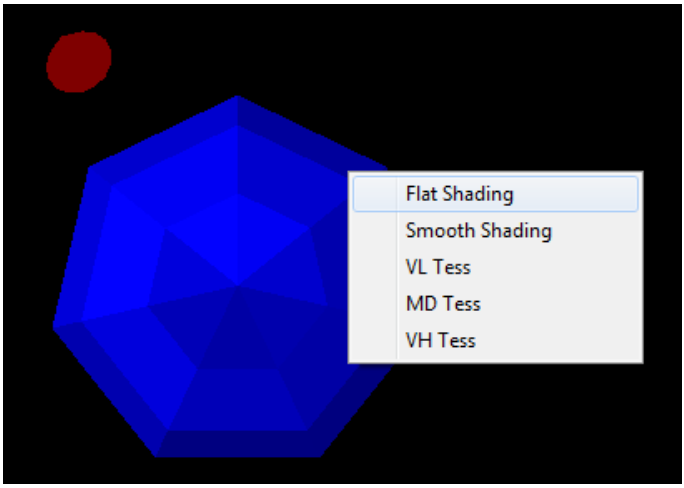

Рис. 30 – Программа SPOT – малое число вершин, плоское затенение

Листинг 7 – Функция визуализации, отвечающая за движение прожектора

```
// Вызывается для рисования сцены
void RenderScene(void)
{
if(iShade == MODE_FLAT)
      glShadeModel(GL_FLAT);
else // iShade = MODE_SMOOTH;
```

```
      glShadeModel(GL_SMOOTH);
// Очищает окно текущим цветом очистки
glClear(GL_COLOR_BUFFER_BIT |
GL_DEPTH_BUFFER_BIT);
// Вначале помещается источник света
// Записывается координатное преобразование
glPushMatrix();
      // Поворачивается система координат
      sfglRotatef(yRot, 0.0f, 1.0f, 0.0f);
      glRotatef(xRot, 1.0f, 0.0f, 0.0f);
      // Задается новое положение и направление
в
      // повернутых координатах
      glLightfv(GL_LIGHT0,GL_POSITION,lightPos)
;
      glLightfv(GL_LIGHT0,GL_SPOT_DIRECTION,spo
tDir);
      // Рисуется красный конус, окружающий ис-
точник света
      glColor3ub(255,0,0);
      // Транслируется начало координат, чтобы
переместить
      // конус в место расположения источника
света
      glTranslatef(lightPos[0],lightPos[1],ligh
tPos[2]);
      glutSolidCone(4.0f,6.0f,15,15);
      // Рисуется немного смещенная сфера, обо-
значающая
      // электрическую лампочку
      // Сохраняются переменные состояния осве-
щения
      glPushAttrib(GL_LIGHTING_BIT);
      // Выключается свет, и задается яркая
желтая сфера
      glDisable(GL_LIGHTING);
```

```
                glColor3ub(255,255,0);
            glutSolidSphere(3.0f, 15, 15);
            // Восстанавливаются переменные состояния
      освещения
            glPopAttrib();
// Восстанавливаются координатные преобразования
glPopMatrix();
// Устанавливается цвет материала, и рисуется сфера
glColor3ub(0, 0, 255);
if(iTess == MODE_VERYLOW)
      glutSolidSphere(30.0f, 7, 7);
      else
            if(iTess == MODE_MEDIUM)
                  glutSolidSphere(3 0.0 f, 15, 15);
else // iTess = MODE_MEDIUM;
      glutSolidSphere(30.0f, 50, 50);
      // Отображаются результаты glutSwapBuffers();
```

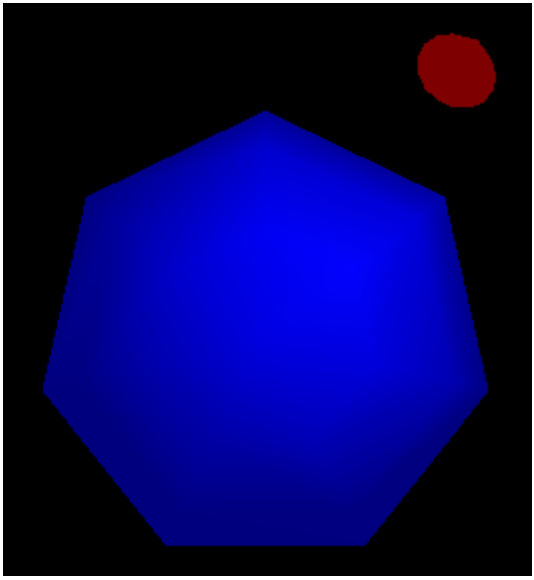

Рис. 31 – Гладкое затенение, но недостаточное число вершин

Переменные iTess и iMode устанавливаются обработчиком меню GLUT и контролируют, на сколько участков разбивается сфера и какое затенение применяется — плоское или гладкое. Обратите внимание на то, что источник света размещается до визуализации геометрических объектов.

На рис.30 можно видеть, что сфера раскрашена грубо и явно заметна каждая плоская грань. Как следует из рис. 31, переключение на гладкое затенение не очень помогает.

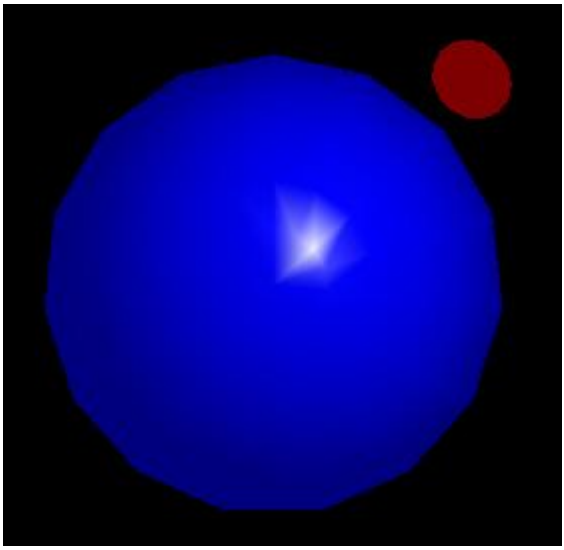

Рис. 32 –Выбор более мелкой сетки многоугольников дает лучшее освещение вершин

Как видно из рис.32, увеличение числа вершин дает лучшие результаты, однако при перемещении источника света вокруг сцены все еще наблюдаются раздражающие артефакты. Эти артефакты являются одним из недостатков схемы освещения OpenGL. Лучше всего охарактеризовать эту ситуацию можно, сказав, что данные артефакты являются недостатком схемы освещения вершин (не обязательно OpenGL!). Освещая вершины, а затем интерполируя полученные значения, получаем грубую аппроксимацию освещения. В большинстве случаев этого подхода достаточно, но, как видно из примера, в опре-

деленных ситуациях его явно мало. Если переключиться на очень мелкое мозаичное представление и подвигать источник света, можно видеть, что освещение портит все, кроме тех мест, где оно отсутствует.

Отметим, что по мере того, как аппаратные ускорители OpenGL начинали ускорять преобразования и эффекты освещения, а процессоры становились мощнее, стало реальным более мелкое мозаичное представление геометрических объектов и лучшие эффекты освещения OpenGL.

Осталось сделать последнее заключение относительно примера SPOT, касающееся мозаичного представления со средней степенью детализации и плоского затенения. Как показано на рис.33, каждая грань сферы окрашивается плоско равномерно. Все вершины имеют одинаковый цвет, который, впрочем, корректируется с учетом нормали и света. При плоском затенении каждый многоугольник имеет цвет последней заданной вершины, и гладкая интерполяция значений вершин не производится.

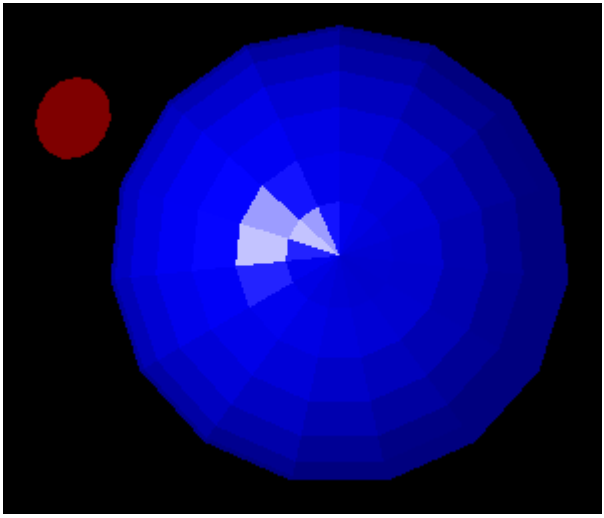

Рис. 33 – Многогранная сфера

Полный листинг приложения сфера - прожектор

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


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

// Коэффициенты освещения и координаты
GLfloat  lightPos[] = { 0.0f, 0.0f, 75.0f, 1.0f };
GLfloat  specular[] = { 1.0f, 1.0f, 1.0f, 1.0f};
GLfloat  specref[] =  { 1.0f, 1.0f, 1.0f, 1.0f };
GLfloat  ambientLight[] = { 0.5f, 0.5f, 0.5f, 1.0f};
GLfloat  spotDir[] = { 0.0f, 0.0f, -1.0f };

// Флаги эффектов
#define MODE_FLAT 1
#define MODE_SMOOTH  2
#define MODE_VERYLOW 3
#define MODE_MEDIUM  4
#define MODE_VERYHIGH 5

int iShade = MODE_FLAT;
int iTess = MODE_VERYLOW;

////////////////////////////////////////////////////
////////////////////////// В ответ на выбор позиции
меню должным образом устанавливаются // флаги
void ProcessMenu(int value)
    {
    switch(value)
        {
        case 1:
            iShade = MODE_FLAT;
            break;
```

```
        case 2:
            iShade = MODE_SMOOTH;
            break;

        case 3:
            iTess = MODE_VERYLOW;
            break;

        case 4:
            iTess = MODE_MEDIUM;
            break;

        case 5:
        default:
            iTess = MODE_VERYHIGH;
            break;
        }

    glutPostRedisplay();
    }

// Вызывается для рисования сцены

void RenderScene(void)
    {
    if(iShade == MODE_FLAT)
        glShadeModel(GL_FLAT);
    else //      iShade = MODE_SMOOTH;
        glShadeModel(GL_SMOOTH);

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

      // Вначале помещается источник света
```

```
      // Записывается координатное преобразование
    glPushMatrix();

      // Поворачивается система координат
        glRotatef(yRot, 0.0f, 1.0f, 0.0f);
        glRotatef(xRot, 1.0f, 0.0f, 0.0f);

      // Задается новое положение и направление в
      // повернутых координатах
        glLightfv(GL_LIGHT0,GL_POSITION,lightPos);

glLightfv(GL_LIGHT0,GL_SPOT_DIRECTION,spotDir);

      // Рисуется красный конус, окружающий источник
света
        glColor3ub(255,0,0);

      // Транслируется начало координат, чтобы пере-
местить
      // конус в место расположения источника света

glTranslatef(lightPos[0],lightPos[1],lightPos[2]);
        glutSolidCone(4.0f,6.0f,15,15);

      // Рисуется немного смещенная сфера, обозначаю-
щая
      // электрическую лампочку
      // Сохраняются переменные состояния освещения
        glPushAttrib(GL_LIGHTING_BIT);

            // Выключается свет, и задается яркая
      желтая сфера
            glDisable(GL_LIGHTING);
            glColor3ub(255,255,0);
            glutSolidSphere(3.0f, 15, 15);
```

```
        // Восстанавливаются переменные состояния ос-
вещения
        glPopAttrib();

// Восстанавливаются координатные преобразования
    glPopMatrix();

    // Устанавливается цвет материала, и рисуется
сфера
    glColor3ub(0, 0, 255);

    if(iTess == MODE_VERYLOW)
      glutSolidSphere(30.0f, 7, 7);
    else
        if(iTess == MODE_MEDIUM)
            glutSolidSphere(30.0f, 15, 15);
        else //  iTess = MODE_MEDIUM;
            glutSolidSphere(30.0f, 50, 50);

      // Отображаются результаты
    glutSwapBuffers();
    }

// Эта функция выполняет необходимую инициализацию
// в контексте визуализации
void SetupRC()
    {
        glEnable(GL_DEPTH_TEST); //Удаление скрытых
поверхностей
        glFrontFace(GL_CCW);      // Многоугольники с
                                  обходом против часовой
                                  // стрелки направлены
                                  вперед
        glEnable(GL_CULL_FACE);   // Задние поверхно-
сти не отображаются
```

```
    // Активизируется освещение
    glEnable(GL_LIGHTING);

    // Устанавливается и активизируется источник
света 0
    // Создается слабое рассеянное освещение, чтобы
    // можно было видеть объекты
    glLightModelfv(GL_LIGHT_MODEL_AMBIENT, ambien-
tLight);

    // Источник света имеет только диффузный и отра-
жательный
    // компоненты
    glLightfv(GL_LIGHT0,GL_DIFFUSE,ambientLight);
    glLightfv(GL_LIGHT0,GL_SPECULAR,specular);
    glLightfv(GL_LIGHT0,GL_POSITION,lightPos);

    // Прожекторные эффекты
    // Угол конуса освещения составляет 60 градусов
    glLightf(GL_LIGHT0,GL_SPOT_CUTOFF,50.0f);

    // Активизируется источник света
    glEnable(GL_LIGHT0);

    // Активизируем согласование цветов
    glEnable(GL_COLOR_MATERIAL);

      // Свойства материалов соответствуют кодам
      glColor    glColorMaterial(GL_FRONT,
      GL_AMBIENT_AND_DIFFUSE);

    // С этого момента все материала получают спо-
собность
    // отражать блики
    glMaterialfv(GL_FRONT, GL_SPECULAR,specref);
```

```
    glMateriali(GL_FRONT, GL_SHININESS,128);


    // Черный фон
    glClearColor(0.0f, 0.0f, 0.0f, 1.0f );
    }

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

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

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

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

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

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

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

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

    // Перерисовка окна
    glutPostRedisplay();
    }
```

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

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

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

  // Обновляется система координат
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    fAspect = (GLfloat) w / (GLfloat) h;
    gluPerspective(35.0f, fAspect, 1.0f, 500.0f);

    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    glTranslatef(0.0f, 0.0f, -250.0f);
    }

int main(int argc, char* argv[])
    {
    int nMenu;

    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB |
GLUT_DEPTH);
    glutInitWindowSize(800, 600);
    glutCreateWindow("Spot Light");

  // Создается меню
    nMenu = glutCreateMenu(ProcessMenu);
```

```
glutAddMenuEntry("Flat Shading",1);
glutAddMenuEntry("Smooth Shading",2);
glutAddMenuEntry("VL Tess",3);
glutAddMenuEntry("MD Tess",4);
glutAddMenuEntry("VH Tess",5);
glutAttachMenu(GLUT_RIGHT_BUTTON);

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

return 0;
}
```

Код наложения проекции

Итак, нужно так “сплющить” спроектированную матрицу наблюдения модели, чтобы все представленные в ней объекты рисовались в двухмерном виде. Вне зависимости от ориентации объекта он сплющивается на плоскость, в которой лежит его тень. Еще следует учесть два параметра: расстояние до источника света и направление его излучения. Направление лучей источника света определяет форму тени и влияет на ее размер.

Приведенная в листинге 8 функция `gltMakeShadowMatrix` из библиотеки `glTools` принимает в качестве аргументов три точки плоскости, на которой желательно видеть тень (эти три точки не должны лежать на одной прямой), положение источника света и указатель на матрицу преобразования, которую строит эта функция. Следует знать, что эта функция выводит коэффициенты уравнения плоскости, на которой будет располагаться тень, и с их помощью, учитывая положение источника света, строит матрицу преобразования. Если эту матрицу умножить на текущую матрицу наблюдения модели, все

нарисованные впоследствии объекты будут спроектированы на эту плоскость.

Листинг 8 –Функции, дающие матрицу “теневого преобразования"

```
      //По коэффициентам уравнения плоскости и поло-
      жению источника света
      // создается матрица отбрасывания тени. Полу-
      ченное
      // значение сохраняется в destMat
      void gltMakeShadowMatrix(GLTVector3 vPoints[3],
                      GLTVector4 vLightPos,
                      GLTMatrix destMat)
      {
      GLTVector4 vPlaneEquation;
      GLfloat dot;
      gltGetPlaneEquation(vPoints[0], vPoints[l],
      vPoints[2], vPlaneEquation);
// Скалярное произведение направляющего вектора
// плоскости и положения источника света
      dot = vPlaneEquation[0]*vLightPos[0] +
            vPlaneEquation[1]*vLightPos[1] +
            vPlaneEquation[2]*vLightPos[2] +
            vPlaneEquation[3]*vLightPos[3];
// Выполняется проектирование
// Первый столбец
      destMat[0] = dot - vLightPos[0] *
      vPlaneEquation[0] ; destMat[4] = 0.0f -
      vLightPos[0] * vPlaneEquation[1]; destMat[8] =
      0.0f - vLightPos[0] * vPlaneEquation[2];
      destMat[12] = 0.0f - vLightPos[0] *
      vPlaneEquation[3];
// Второй столбец
      destMat[1] = 0.0f - vLightPos[l] *
      vPlaneEquation[0] ; destMat[5] = dot -
      vLightPos[l] * vPlaneEquation[1]; destMat[9] =
```

```
      0.0f - vLightPos[l] * vPlaneEquation[2];
      destMat[13] = 0.0f - vLightPos[l] *
      vPlaneEquation[3] ;
// Третий столбец
      destMat[2] = 0.0f - vLightPos[2] *
      vPlaneEquation[0]; destMat[6] = 0.0f -
      vLightPos[2] * vPlaneEquation[1]; destMat[10] =
      dot - vLightPos[2] * vPlaneEquation[2] ;
      destMat[14] = 0.0f - vLightPos[2] *
      vPlaneEquation[3];
// Четвертый столбец
      destMat[3] = 0.0f - vLightPos[3] *
      vPlaneEquation(0]; destMat[7] = 0.0f -
      vLightPos[3] * vPlaneEquation[1] ; destMat[11]
      = 0.0f - vLightPos[3] * vPlaneEquation[2];
      destMat[15] = dot - vLightPos[3] *
      vPlaneEquation[3] ;
      )
```

Пример тени

Чтобы продемонстрировать использование функции, приведенной в листинге 8, подвесим самолет в воздухе над землей. Источник света будет помещен выше и немного левее самолета. С помощью клавиш со стрелками самолет можно вращать, при этом на земле будет соответствующим образом меняться его тень. Результат выполнения соответствующей программы показан на рис 34.

В коде, приведенном в листинге 9, показано, как была создана матрица проекции тени. Обратите внимание, что создали матрицу один раз в функции SetupRC и записали ее в глобальной переменной.

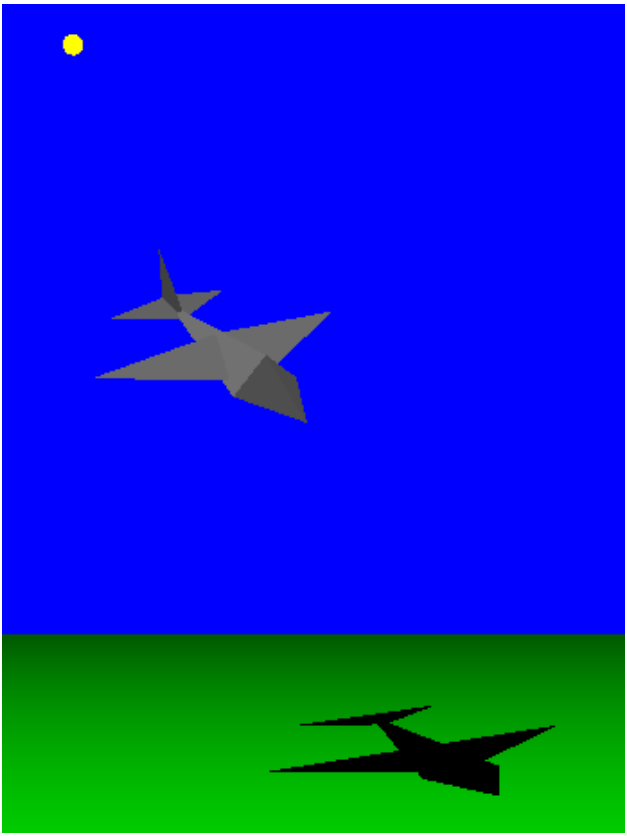

Рис. 34 – Результат выполнения программы

Листинг 9 – Установка матрицы проекции тени

```
GLfloat lightPos[] = { -75.0f, 150.0f, -50.0f,
0.0f };
…
…
// Матрица преобразования, дающая проекцию тени
GLTMatrix shadowMat;
…
…
```

```
// Эта функция выполняет необходимую инициали-
зацию
// в контексте визуализации. В данном случае
она
// устанавливает и инициализирует освещение на
сцене
void SetupRC()
      {
// Любые три точки на поверхности (заданы в по-
рядке
// против часовой стрелки)
      GLTVector3 points[3] = {{-30.0f, -149.0f,
-20.0f },
                                    {-30.0f, -
                                    149.0f, 20.0f
                                    },
                                    { 40.0f, -
                                    149.0f,
                                    20.0f}};
glEnable(GL_DEPTH_TEST);
// Удаление скрытых поверхностей
glFrontFace(GL_CCW);
// Многоугольники с обходом против часовой
стрелки
// направлены вперед glEnable(GL_CULL_FACE);
// Внутри самолета расчеты не производятся
// Активизируется освещение
glEnable(GL_LIGHTING);
      …
// Код настройки освещения и т.п.
…
// Светло-синий фон
      glClearColor(0.0f, 0.0f, l.0f, l.0f );
// Расчет матрицы проекции для рисования тени
// на земле
```

```
        gltMakeShadowMatrix(points, lightPos,
        shadowMat);
    }
```

В листинге 10 приводится код визуализации системы “самолетик с тенью”. Вначале рисуем землю. Затем изображаем самолетик (как обычно), восстанавливаем матрицу наблюдения модели и умножаем ее на матрицу тени. Так получаем искомую матрицу отбрасывания тени. После этого снова рисуем самолет. (Мы модифицировали код, и теперь он принимает метку, сообщающую функции `DrawJet` проводить цветную или черно-белую визуализацию). Восстановив еще раз матрицу наблюдения модели, рисуем небольшую желтую сферу, аппроксимирующую положение источника света. Обратите внимание на то, что перед рисованием плоскости под самолетиком отключена проверка глубины.

Данный прямоугольник лежит в той же плоскости, в которой нарисована тень, и необходимо гарантировать, что тень рисуется. Еще не обсуждалось, что произойдет, если рисовать два объекта или плоскости в одном месте. Впрочем, рассматривалась проверка глубины как средство, позволяющее определить порядок рисования объектов. Если два объекта занимают одно и то же место, обычно показывается только последняя нарисованная фигура. Иногда, впрочем, из-за слабого разрешения значений по глубине возникает полная неразбериха — на экране вперемешку отображаются пиксели, принадлежащие разным объектам (z-fighting)

Листинг 10 – Визуализация самолета и его тени

```
    //Вызывается для рисования сцены
    void RenderScene(void)
    { // Очищает окно текущим цветом очистки
    glClear(GL_COLOR_BUFFER_BIT |
GL_DEPTH_BUFFER_BIT);
    // Рисуется земля; чтобы создать иллюзию глуби-
ны, мы вручную
```

```
      // создаем темно-зеленую тень на заднем плане
glBegin(GL_QUADS);
      glColor3ub(0,32,0);
      glVertex3f(400-0f, -150.0f, -200 0f);
      glVertex3f(-400 0f, -150.0f, -200.0f);
      glColor3ub(0,255,0);
      glVertex3f(-400.0f, -150.0f, 200.0f);
      glVertex3f{400.0f, -150 0f, 200.0f);
      glEndl);
      // Записывается состояние матрицы и выполняются
повороты
      glPushMatrix();
      // Рисуется самолет с новой ориентацией;
      // перед его вращением в нужном месте помещает-
ся
      // источник света
      glEnable(GL_LIGHTING);
      glLightfv(GL_L1GHT0,GL_POSITION,lightPos);
      glRotatef(xRot, l.0f, 0.0f, 0.0f);
      glRotatef(yRot, 0.0f, l.0f, 0.0f);
      DrawJet(FALSE);
      // Восстанавливается исходное состояние матрицы
      glPopMatnx() ;
      // Мы готовы рисовать тень и землю
      // Вначале деактивизируется освещение и записы-
вается
      // состояние проекции
      glDisable(GL_DEPTH_TEST);
      gIDisable(GL_LIGHTING);
      glPushMatnx();
      // Текущая матрица множится на матрицу проекции
тени
      glMultMatrixf((GLfloat *)shadowMat);
      // Самолетик поворачивается в новом плоском
пространстве
      glRotatef(xRot, l.0f, 0.0f, 0.0f);
```

```
      glRotatef(yRot, 0.0f, 1.0f, 0.0f);
      // Чтобы обозначить рисование тени, функции пе-
редается
      // значение TRUE
      DrawJet(TRUE);
      // Восстанавливается нормальная проекция
      glPopMatrix();
      // Рисуется источник света
      glPushMatrix();
      glTranslatef(lightPos[0}, lightPos[1],
lightPos[2]);
      glColor3ub(255,255,0);
      glutSolidSphere(5.0f,10,10);
      glPopMatrix();
      // Восстанавливаются переменные состояния осве-
щения
      glEnable(GL_DEPTH_TEST);
      // Отображаются результаты
      glutSwapBuffers();
```

Полный код для Листингов 8 – 10

```
// Тени
#include <glew.h>
#include <glut.h>
#include <math.h>

// Параметры поворотов
static GLfloat xRot = 0.0f;
static GLfloat yRot = 0.0f;

// Глобальные переменные
// Коды и координаты источников света
GLfloat  ambientLight[] = { 0.3f, 0.3f, 0.3f, 1.0f };
GLfloat  diffuseLight[] = { 0.7f, 0.7f, 0.7f, 1.0f };
GLfloat  specular[] = { 1.0f, 1.0f, 1.0f, 1.0f};
```

```
GLfloat     lightPos[] = { -75.0f, 150.0f, -50.0f,
0.0f };
GLfloat  specref[] =  { 1.0f, 1.0f, 1.0f, 1.0f };
typedef GLfloat GLTVector3[3];
typedef GLfloat GLTVector2[2];//Двукомпонентный век-
тор с плавающей запятой
typedef GLfloat GLTVector3[3];//Трехомпонентный век-
тор с плавающей запятой
typedef GLfloat GLTVector4[4];//Четырехкомпонентный
вектор с плавающей //запятой
typedef GLfloat GLTMatrix[16];// Основноц столбец
матрицы 4x4 с плавающей //запятой

 // Матрица преобразования, дающая проекцию тени
GLTMatrix
shadowMat;
// Масштабирование скалярного вектора
void gltScaleVector(GLTVector3 vVector, const GLfloat
fScale)
    {
    vVector[0] *= fScale; vVector[1] *= fScale;
vVector[2] *= fScale;
    }
// Возвращает длину вектора в квадрате
GLfloat gltGetVectorLengthSqrd(const GLTVector3
vVector)
    {
    return (vVector[0]*vVector[0]) +
(vVector[1]*vVector[1]) + (vVector[2]*vVector[2]);
    }
// Возвращает длину вектора
GLfloat gltGetVectorLength(const GLTVector3 vVector)
    {
    return
(GLfloat)sqrt(gltGetVectorLengthSqrd(vVector));
    }
```

```
// Вычитание одного вектора из другого
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];
    }
// Вычислить векторное произведение двух векторов
void gltVectorCrossProduct(const GLTVector3 vU, const
GLTVector3 vV, GLTVector3 vResult)
     {
     vResult[0] = vU[1]*vV[2] - vV[1]*vU[2];
     vResult[1] = -vU[0]*vV[2] + vV[0]*vU[2];
     vResult[2] = vU[0]*vV[1] - vV[0]*vU[1];
     }
// Масштабирование вектора по длине - создание еди-
ничсного вектора
void gltNormalizeVector(GLTVector3 vNormal)
    {
    GLfloat fLength = 1.0f /
gltGetVectorLength(vNormal);
    gltScaleVector(vNormal, fLength);
    }
// Три точки на плоскости расположены против часовой
стрелки, вычисление
// нормали
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);
    }



// Полученные три коэффициента уравнения плоскости
дают три точки на поверхности
void gltGetPlaneEquation(GLTVector3 vPoint1,
GLTVector3 vPoint2, GLTVector3 vPoint3, GLTVector3
vPlane)
    {
    // Получение нормали из трех точек. Нормаль –
первые три коэффициента
      // уравнения плоскости
    gltGetNormalVector(vPoint1, vPoint2, vPoint3,
vPlane);

    // Итоговый коэффициент находится обратной под-
становкой
    vPlane[3] = -(vPlane[0] * vPoint3[0] + vPlane[1]
* vPoint3[1] + vPlane[2] * vPoint3[2]);
    }
// Создание матрицы теневой проекции из коэффициентов
уравнения плоскости и // положение света. Возвращае-
мое значение хранится в destMat
void gltMakeShadowMatrix(GLTVector3 vPoints[3],
GLTVector4 vLightPos, GLTMatrix destMat)
    {
    GLTVector4 vPlaneEquation;
    GLfloat dot;

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

```
    // Скалярное произведение положение самолета и
света
    dot =   vPlaneEquation[0]*vLightPos[0] +
            vPlaneEquation[1]*vLightPos[1] +
            vPlaneEquation[2]*vLightPos[2] +
            vPlaneEquation[3]*vLightPos[3];


    // Проецируем
    // Первый столбец
    destMat[0] = dot - vLightPos[0] *
vPlaneEquation[0];
    destMat[4] = 0.0f - vLightPos[0] *
vPlaneEquation[1];
    destMat[8] = 0.0f - vLightPos[0] *
vPlaneEquation[2];
    destMat[12] = 0.0f - vLightPos[0] *
vPlaneEquation[3];

    // Второй столбец
    destMat[1] = 0.0f - vLightPos[1] *
vPlaneEquation[0];
    destMat[5] = dot - vLightPos[1] *
vPlaneEquation[1];
    destMat[9] = 0.0f - vLightPos[1] *
vPlaneEquation[2];
    destMat[13] = 0.0f - vLightPos[1] *
vPlaneEquation[3];

    // Третий столбец
    destMat[2] = 0.0f - vLightPos[2] *
vPlaneEquation[0];
    destMat[6] = 0.0f - vLightPos[2] *
vPlaneEquation[1];
    destMat[10] = dot - vLightPos[2] *
vPlaneEquation[2];
```

```
    destMat[14] = 0.0f - vLightPos[2] *
vPlaneEquation[3];

    // Четвертый столбец
    destMat[3] = 0.0f - vLightPos[3] *
vPlaneEquation[0];
    destMat[7] = 0.0f - vLightPos[3] *
vPlaneEquation[1];
    destMat[11] = 0.0f - vLightPos[3] *
vPlaneEquation[2];
    destMat[15] = dot - vLightPos[3] *
vPlaneEquation[3];
    }

////////////////////////////////////////////////
// Функция, специально прорисовывающая самолет
void DrawJet(int nShadow)
     {
     GLTVector3 vNormal;    // Память для расчетов
нормали к поверхности

     // Носовой конус ////////////////////////////
     // Установите цвет материала, обратите внима-
ние, черный цвет
     // используется только для тени
     if(nShadow == 0)
          glColor3ub(128, 128, 128);
     else
           glColor3ub(0,0,0);


     // Носовой конус - Прямо вниз
     glBegin(GL_TRIANGLES);
               glNormal3f(0.0f, -1.0f, 0.0f);
          glNormal3f(0.0f, -1.0f, 0.0f);
          glVertex3f(0.0f, 0.0f, 60.0f);
```

```
            glVertex3f(-15.0f, 0.0f, 30.0f);
            glVertex3f(15.0f,0.0f,30.0f);


                // Вершин для этой панели
                {
                GLTVector3 vPoints[3] = {{ 15.0f,
0.0f,  30.0f},
                                         { 0.0f,
15.0f, 30.0f},
                                         { 0.0f,
0.0f,  60.0f}};

                // Рассчитывается нормаль плоскости
                gltGetNormalVector(vPoints[0],
vPoints[1], vPoints[2], vNormal);
            glNormal3fv(vNormal);
            glVertex3fv(vPoints[0]);
            glVertex3fv(vPoints[1]);
            glVertex3fv(vPoints[2]);
                }


                {
                GLTVector3 vPoints[3] = {{ 0.0f,
0.0f, 60.0f },
                                         { 0.0f,
15.0f, 30.0f },
                                         { -15.0f,
0.0f, 30.0f }};

                gltGetNormalVector(vPoints[0],
vPoints[1], vPoints[2], vNormal);
                glNormal3fv(vNormal);
            glVertex3fv(vPoints[0]);
            glVertex3fv(vPoints[1]);
```

```
            glVertex3fv(vPoints[2]);
                 }


                 // Тело плоскости
//////////////////////
                 {
                 GLTVector3 vPoints[3] = {{ -15.0f,
0.0f, 30.0f },
                                          { 0.0f,
15.0f, 30.0f },
                                  { 0.0f, 0.0f, -56.0f
}};

                 gltGetNormalVector(vPoints[0],
vPoints[1], vPoints[2], vNormal);
                 glNormal3fv(vNormal);
            glVertex3fv(vPoints[0]);
            glVertex3fv(vPoints[1]);
            glVertex3fv(vPoints[2]);
                 }

                 {
                 GLTVector3 vPoints[3] = {{ 0.0f,
0.0f, -56.0f },
                                          { 0.0f,
15.0f, 30.0f },
                                          {
15.0f,0.0f,30.0f }};

                 gltGetNormalVector(vPoints[0],
vPoints[1], vPoints[2], vNormal);
                 glNormal3fv(vNormal);
            glVertex3fv(vPoints[0]);
            glVertex3fv(vPoints[1]);
            glVertex3fv(vPoints[2]);
```

```
                }


            glNormal3f(0.0f, -1.0f, 0.0f);
            glVertex3f(15.0f,0.0f,30.0f);
            glVertex3f(-15.0f, 0.0f, 30.0f);
            glVertex3f(0.0f, 0.0f, -56.0f);


////////////////////////////////////////////
                // Левое крыло
                // Большой треугольник для нижней
части крыла
                {
                GLTVector3 vPoints[3] = {{
0.0f,2.0f,27.0f },
                                         { -60.0f,
2.0f, -8.0f },
                                         { 60.0f,
2.0f, -8.0f }};

                gltGetNormalVector(vPoints[0],
vPoints[1], vPoints[2], vNormal);
                glNormal3fv(vNormal);
            glVertex3fv(vPoints[0]);
            glVertex3fv(vPoints[1]);
            glVertex3fv(vPoints[2]);
                }


                {
                GLTVector3 vPoints[3] = {{ 60.0f,
2.0f, -8.0f},
                                {0.0f, 7.0f, -8.0f},
                                {0.0f,2.0f,27.0f }};
```

```
                gltGetNormalVector(vPoints[0],
vPoints[1], vPoints[2], vNormal);
                glNormal3fv(vNormal);
           glVertex3fv(vPoints[0]);
           glVertex3fv(vPoints[1]);
           glVertex3fv(vPoints[2]);
                }

                {
                GLTVector3 vPoints[3] = {{60.0f,
2.0f, -8.0f},
                              {-60.0f, 2.0f, -8.0f},
                              {0.0f,7.0f,-8.0f }};

                gltGetNormalVector(vPoints[0],
vPoints[1], vPoints[2], vNormal);
                glNormal3fv(vNormal);
           glVertex3fv(vPoints[0]);
           glVertex3fv(vPoints[1]);
           glVertex3fv(vPoints[2]);
                }

                {
                GLTVector3 vPoints[3] =
{{0.0f,2.0f,27.0f},
                                        {0.0f, 7.0f,
-8.0f},
                                        {-60.0f,
2.0f, -8.0f}};

                gltGetNormalVector(vPoints[0],
vPoints[1], vPoints[2], vNormal);
                glNormal3fv(vNormal);
           glVertex3fv(vPoints[0]);
           glVertex3fv(vPoints[1]);
           glVertex3fv(vPoints[2]);
```

```
                }


                // Хвостовая
часть//////////////////////////////
                // Нижняя часть стабилизатора
            glNormal3f(0.0f, -1.0f, 0.0f);
            glVertex3f(-30.0f, -0.50f, -57.0f);
            glVertex3f(30.0f, -0.50f, -57.0f);
            glVertex3f(0.0f,-0.50f,-40.0f);

                {
                GLTVector3 vPoints[3] = {{ 0.0f,-
0.5f,-40.0f },
                                {30.0f, -0.5f, -57.0f},
                                {0.0f, 4.0f, -57.0f }};

                gltGetNormalVector(vPoints[0],
vPoints[1], vPoints[2], vNormal);
                glNormal3fv(vNormal);
            glVertex3fv(vPoints[0]);
            glVertex3fv(vPoints[1]);
            glVertex3fv(vPoints[2]);
                }


                {
                GLTVector3 vPoints[3] = {{ 0.0f,
4.0f, -57.0f },
                                { -30.0f, -0.5f, -57.0f
},
                                { 0.0f,-0.5f,-40.0f }};

                gltGetNormalVector(vPoints[0],
vPoints[1], vPoints[2], vNormal);
                glNormal3fv(vNormal);
```

```
            glVertex3fv(vPoints[0]);
            glVertex3fv(vPoints[1]);
            glVertex3fv(vPoints[2]);
                 }


                 {
                 GLTVector3 vPoints[3] = {{ 30.0f,-
0.5f,-57.0f },
                              { -30.0f, -0.5f, -57.0f
},
                              { 0.0f, 4.0f, -57.0f }};

                 gltGetNormalVector(vPoints[0],
vPoints[1], vPoints[2], vNormal);
                 glNormal3fv(vNormal);
            glVertex3fv(vPoints[0]);
            glVertex3fv(vPoints[1]);
            glVertex3fv(vPoints[2]);
                 }

                 {
                 GLTVector3 vPoints[3] = {{
0.0f,0.5f,-40.0f },
                              { 3.0f, 0.5f, -57.0f },
                              { 0.0f, 25.0f, -65.0f
}};

                 gltGetNormalVector(vPoints[0],
vPoints[1], vPoints[2], vNormal);
                 glNormal3fv(vNormal);
            glVertex3fv(vPoints[0]);
            glVertex3fv(vPoints[1]);
            glVertex3fv(vPoints[2]);
                 }
```

```
                {
                GLTVector3 vPoints[3] = {{ 0.0f,
25.0f, -65.0f },
                                { -3.0f, 0.5f, -57.0f},
                                { 0.0f,0.5f,-40.0f }};

                gltGetNormalVector(vPoints[0],
vPoints[1], vPoints[2], vNormal);
                glNormal3fv(vNormal);
           glVertex3fv(vPoints[0]);
           glVertex3fv(vPoints[1]);
           glVertex3fv(vPoints[2]);
                }

                {
                GLTVector3 vPoints[3] = {{
3.0f,0.5f,-57.0f },
                                { -3.0f, 0.5f, -57.0f },
                                { 0.0f, 25.0f, -65.0f
}};

                gltGetNormalVector(vPoints[0],
vPoints[1], vPoints[2], vNormal);
                glNormal3fv(vNormal);
           glVertex3fv(vPoints[0]);
           glVertex3fv(vPoints[1]);
           glVertex3fv(vPoints[2]);
                }


        glEnd();
     }

// Вызывается окно для рисования
void RenderScene(void)
```

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

      // Рисуется земля; чтобы создать иллюзию глуби-
ны, мы вручную
      // создаем темно-зеленую тень на заднем плане
      glBegin(GL_QUADS);
        glColor3ub(0,32,0);
        glVertex3f(400.0f, -150.0f, -200.0f);
        glVertex3f(-400.0f, -150.0f, -200.0f);
        glColor3ub(0,255,0);
        glVertex3f(-400.0f, -150.0f, 200.0f);
        glVertex3f(400.0f, -150.0f, 200.0f);
    glEnd();

    // Записывается состояние матрицы и выполняются
повороты
    glPushMatrix();

      // Рисуется самолет с новой ориентацией;
      // перед его вращением в нужном месте помещает-
ся
      // источник света
    glEnable(GL_LIGHTING);
    glLightfv(GL_LIGHT0,GL_POSITION,lightPos);
    glRotatef(xRot, 1.0f, 0.0f, 0.0f);
    glRotatef(yRot, 0.0f, 1.0f, 0.0f);

    DrawJet(0);

      // Восстанавливается исходное состояние матрицы
    glPopMatrix();
```

```
      // Мы готовы рисовать тень и землю
      // Вначале деактивизируется освещение и записы-
вается
      // состояние проекции
    glDisable(GL_DEPTH_TEST);
    glDisable(GL_LIGHTING);
    glPushMatrix();

      // Текущая матрица множится на матрицу проекции
тени
    glMultMatrixf((GLfloat *)shadowMat);

      // Самолетик поворачивается в новом плоском
пространстве
    glRotatef(xRot, 1.0f, 0.0f, 0.0f);
    glRotatef(yRot, 0.0f, 1.0f, 0.0f);

      // Чтобы обозначить рисование тени, функции пе-
редается
      // значение 1
    DrawJet(1);

      // Восстанавливается нормальная проекция
    glPopMatrix();

      // Рисуется источник света
    glPushMatrix();
    glTranslatef(lightPos[0],lightPos[1],
lightPos[2]);
    glColor3ub(255,255,0);
    glutSolidSphere(5.0f,10,10);
    glPopMatrix();

      // Восстанавливаются переменные состояния осве-
щения
    glEnable(GL_DEPTH_TEST);
```

```
      // Отображаются результаты
    glutSwapBuffers();
    }

// This function does any needed initialization on
the rendering
// context.
void SetupRC()
    {
          // Любые три точки на поверхности (заданы
      в порядке
          // против часовой стрелки)

    GLTVector3 points[3] = {{ -30.0f, -149.0f, -20.0f
},
                            { -30.0f, -149.0f, 20.0f
},
                            { 40.0f, -149.0f, 20.0f
}};

          glEnable(GL_DEPTH_TEST); // Удаление скры-
      тых поверхностей
          glFrontFace(GL_CCW); // Многоугольники с
      обходом против часовой стрелки направлены впе-
      ред
      glEnable(GL_CULL_FACE);     // Внутри самолета
      расчеты не производятся
    // Устанавливается и активизируется источник све-
    та 0
    glLightfv(GL_LIGHT0,GL_AMBIENT,ambientLight);
    glLightfv(GL_LIGHT0,GL_DIFFUSE,diffuseLight);
    glLightfv(GL_LIGHT0,GL_SPECULAR,specular);
    glLightfv(GL_LIGHT0,GL_POSITION,lightPos);
    glEnable(GL_LIGHT0);
```

```
    // Активизирует согласование цветов
    glEnable(GL_COLOR_MATERIAL);

    // Свойства материалов соответствуют кодам
glColor
    glColorMaterial(GL_FRONT,
GL_AMBIENT_AND_DIFFUSE);

     // С этого момента все материалы получают спо-
собность
     // отражать блики
    glMaterialfv(GL_FRONT, GL_SPECULAR,specref);
    glMateriali(GL_FRONT,GL_SHININESS,128);

    // Светло-синий фон
    glClearColor(0.0f, 0.0f, 1.0f, 1.0f );

      // Расчет матрицы проекции для рисования тени
на земле
    gltMakeShadowMatrix(points, lightPos, shadowMat);
    }

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

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

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

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

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

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

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

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

    // Перерисовка окна
    glutPostRedisplay();
    }


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

    // Предотвращение деления на ноль
    if(h == 0)
        h = 1;


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

    // Обновляет систему координат
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();

    fAspect = (GLfloat)w/(GLfloat)h;
    gluPerspective(60.0f, fAspect, 200.0, 500.0);
```

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

    // Перемшаемся вдоль оси Z в поле зрения
    glTranslatef(0.0f, 0.0f, -400.0f);
    glLightfv(GL_LIGHT0,GL_POSITION,lightPos);
    }

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

    return 0;
    }
```

Конечный цвет является масштабированной комбинацией исходного кода красного цвета и поступающего кода синего цвета. Чем выше входной альфа-фактор, тем больше поступающего и меньше старого цвета будет в конечном цвете.

Описанная функция смешения часто используется для создания иллюзии прозрачного объекта, за которым находится что-то непрозрачное. Тем не менее, данная технология требует, чтобы вначале нарисован фоновый объект (или объекты), а затем поверх них изображены прозрачные объекты. Эффект может быть достаточно впечатляющим. Например, в программе REFLECTION использовалась прозрач-

ность для создания иллюзии отражения от зеркальной поверхности. Вначале создан крутящийся тор, вокруг которого вращается сфера. Ниже тора и сферы находится пол из отражающих плиток. Результат выполнения данной программы показан на рис. 19, а код, отвечающий за рисование, приводится в листинге 11.

Листинг 11 – Функция визуализации программы REFLECTION

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

// Вызывается для рисования сцены

void RenderScene(void)

{

// Очищаем окно текущим цветом очистки

glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

glPushMatrix();

// Источник света помещается под полом и освещает

// "отраженный" мир

      glLightfv(GL_LIGHT0, GL_POSITION,
      fLightPosMirror); glPushMatrix();

glFrontFace(GL_CW);

// Зеркально отображается геометрия, инвертируется

// ориентация

glScalef(1.0f, -1.0f, 1.0f);

DrawWorld();

glFrontFace(GL_CCW);
```

```
glPopMatrix();
// Над "отраженными" объектами рисуется
// прозрачная земля
glDisable(GL_LIGHTING);
glEnable(GL_BLEND);
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
DrawGround();
glDisable(GL_BLEND);
glEnable(GL_LIGHTING);
// Восстанавливается правильное освещение и
// правильно рисуется мир
glLightfv(GL_LIGHT0, GL_POSITION, fLightPos);
DrawWorld();
glPopMatrix();
// Переключает буферы
glutSwapBuffers();
}
```

Объединение цветов

Суть алгоритма, порождающего искомый эффект, заключается в рисовании сцены наоборот. Чтобы нарисовать сцену, используем функцию `DrawWorld()`, но чтобы изобразить ее наоборот, масштабируем все с коэффициентом -1 (инверсия относительно оси y), меняем обход многоугольников на обратный и помещаем источник света под наблюдателем. После того как перевернутый мир нарисован, изображаем землю, однако с помощью смешения пол, расположенный

поверх обращенного мира, становится прозрачным. Наконец, отключаем смешение, помещаем источника света над наблюдателем и рисуем мир нормально.

Полный код листинга 11

```
// Reflection.c
#include <glew.h>
#include <glut.h>
#include <math.h>
#include <stdio.h>

#define GLT_PI   3.14159265358979323846
#define GLT_PI_DIV_180 0.017453292519943296
#define GLT_INV_PI_DIV_180 57.2957795130823229
// Данные о свете и материале
GLfloat fLightPos[4]   = { -100.0f, 100.0f, 50.0f,
1.0f };  // Point source
GLfloat fLightPosMirror[4] = { -100.0f, -100.0f,
50.0f, 1.0f };
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 };

static GLfloat yRot = 0.0f;//Угол поворота, задейст-
вованный в анимации
typedef GLfloat GLTVector3[3]; // Трехомпонентный
вектор с плавающей запятой


// Для достижения наилучших результатов, поместим его
в списке отображения
// Рисуем тор (бублик) при z = fZVal...  в плоскости
XY
// Скалярное масштабирование вектора
```

```
void gltScaleVector(GLTVector3 vVector, const GLfloat
fScale)
    {
    vVector[0] *= fScale; vVector[1] *= fScale;
vVector[2] *= fScale;
    }
// Возвращает длину вектора в квадрате
GLfloat gltGetVectorLengthSqrd(const GLTVector3
vVector)
    {
    return (vVector[0]*vVector[0]) +
(vVector[1]*vVector[1]) + (vVector[2]*vVector[2]);
    }
// Возвращает длину вектора
GLfloat gltGetVectorLength(const GLTVector3 vVector)
    {
    return
(GLfloat)sqrt(gltGetVectorLengthSqrd(vVector));
    }
// Масштабирование вектора по длине - создание еди-
ничсного вектора
void gltNormalizeVector(GLTVector3 vNormal)
    {
    GLfloat fLength = 1.0f /
gltGetVectorLength(vNormal);
    gltScaleVector(vNormal, fLength);
    }
void gltDrawTorus(GLfloat majorRadius, GLfloat
minorRadius, 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 +
majorRadius;
                    GLfloat z = minorRadius *
(GLfloat) sin(b);

                    // Первая точка

glTexCoord2f((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);


glTexCoord2f((float)(i+1)/(float)(numMajor),
(float)(j)/(float)(numMinor));
                    vNormal[0] = x1*c;
                    vNormal[1] = y1*c;
                    vNormal[2] = z/minorRadius;
                    glNormal3fv(vNormal);
```

```
                    glVertex3f(x1*r, y1*r, z);
                    }
            glEnd();
            }
      }

//////////////////////////////////////////////////////
/////////////
// Эта функция выполняет необходимую инициализацию в
контексте
// визуализации
void SetupRC()
    {
    // Серый фон
    glClearColor(fLowLight[0], fLowLight[1],
fLowLight[2], fLowLight[3]);

    // Задние части многоугольников отбрасываются
    glCullFace(GL_BACK);
    glFrontFace(GL_CCW);
    glEnable(GL_CULL_FACE);
    glEnable(GL_DEPTH_TEST);

    // Устанавливаются параметры источника света:
glLightModelfv(GL_LIGHT_MODEL_AMBIENT, fNoLight);
    glLightfv(GL_LIGHT0, GL_AMBIENT, fLowLight);
    glLightfv(GL_LIGHT0, GL_DIFFUSE, fBrightLight);
    glLightfv(GL_LIGHT0, GL_SPECULAR, fBrightLight);
    glEnable(GL_LIGHTING);
    glEnable(GL_LIGHT0);

    // Преимущественно используется согласование
свойств материала      glEnable(GL_COLOR_MATERIAL);
    glColorMaterial(GL_FRONT,
GL_AMBIENT_AND_DIFFUSE);
    glMateriali(GL_FRONT, GL_SHININESS, 128);
```

```
    }


////////////////////////////////////////////////////////
//////
// Рисуем землю как стороны треугольников. Модели за-
тенения и цветные модели // установлены так, что мы в
конечном итоге получим черно-белую шахматную
// доску.
void DrawGround(void)
    {
    GLfloat fExtent = 20.0f;
    GLfloat fStep = 0.5f;
    GLfloat y = 0.0f;
    GLfloat fColor;
    GLfloat iStrip, iRun;
    GLint iBounce = 0;

    glShadeModel(GL_FLAT);
    for(iStrip = -fExtent; iStrip <= fExtent; iStrip
+= fStep)
        {
        glBegin(GL_TRIANGLE_STRIP);
            for(iRun = fExtent; iRun >= -fExtent;
iRun -= fStep)
                {
                if((iBounce %2) == 0)
                    fColor = 1.0f;
                else
                    fColor = 0.0f;

                glColor4f(fColor, fColor, fColor,
0.5f);
                glVertex3f(iStrip, y, iRun);
                glVertex3f(iStrip + fStep, y, iRun);
```

```
                iBounce++;
                }
        glEnd();
        }
    glShadeModel(GL_SMOOTH);
    }

////////////////////////////////////////////////////
/////////////////
// Рисуем случайных обитателей и вращающихся
тор/сферу
void DrawWorld(void)
    {
    glColor3f(1.0f, 0.0f, 0.0f);
    glPushMatrix();
        glTranslatef(0.0f, 0.5f, -3.5f);

        glPushMatrix();
            glRotatef(-yRot * 2.0f, 0.0f, 1.0f,
0.0f);
            glTranslatef(1.0f, 0.0f, 0.0f);
            glutSolidSphere(0.1f, 17, 9);
        glPopMatrix();


        glRotatef(yRot, 0.0f, 1.0f, 0.0f);
        gltDrawTorus(0.35, 0.15, 61, 37);

    glPopMatrix();
    }


////////////////////////////////////////////////////
/////////////////
// Вызывается для рисования сцены
```

```
void RenderScene(void)
    {
    // Очищаем окно текущим цветом очистки
    glClear(GL_COLOR_BUFFER_BIT |
GL_DEPTH_BUFFER_BIT);

    glPushMatrix();
// Источник света помещается под полом и освещает
// "отраженный" мир
        glLightfv(GL_LIGHT0, GL_POSITION,
fLightPosMirror);
        glPushMatrix();
            glFrontFace(GL_CW); // Зеркально отобра-
жается геометрия,
// инвертируется ориентация
            glScalef(1.0f, -1.0f, 1.0f);
            DrawWorld();
            glFrontFace(GL_CCW);
        glPopMatrix();

       // Над "отраженными" объектами рисуется
      // прозрачная земля

        glDisable(GL_LIGHTING);
        glEnable(GL_BLEND);
        glBlendFunc(GL_SRC_ALPHA,
GL_ONE_MINUS_SRC_ALPHA);
        DrawGround();
        glDisable(GL_BLEND);
        glEnable(GL_LIGHTING);

// Восстанавливается правильное освещение и
// правильно рисуется мир

        glLightfv(GL_LIGHT0, GL_POSITION, fLightPos);
        DrawWorld();
```

```
    glPopMatrix();

    // Переключение буферов
    glutSwapBuffers();
    }



////////////////////////////////////////////////////////
//////
//Вызывается библиотекой GLUT в холостом состоянии
(окно не меняет //размера и не перемещается)
void TimerFunction(int value)
    {
    yRot += 1.0f;   // Обновление коэффициента пово-
рота

    // Перерисовка сцены
    glutPostRedisplay();

    // Сброс таймера
    glutTimerFunc(1,TimerFunction, 1);
    }

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

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

    glViewport(0, 0, w, h);
```

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

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

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

    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    glTranslatef(0.0f, -0.4f, 0.0f);
    }

////////////////////////////////////////////////////
////////
// Точка входа программы
int main(int argc, char* argv[])
    {
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB |
GLUT_DEPTH);
    glutInitWindowSize(800,600);
    glutCreateWindow("OpenGL Blending and
Transparency");
    glutReshapeFunc(ChangeSize);
    glutDisplayFunc(RenderScene);
    glutTimerFunc(10, TimerFunction, 1);

    SetupRC();
    glutMainLoop();

    return 0;
    }
```

Туман

Другим легким в использовании специальным эффектом, который поддерживает OpenGL, является туман. После завершения всех расчетов по цвету OpenGL позволяет смешивать заданный цвет тумана с цветом геометрических объектов. Доля цвета тумана, смешиваемого с цветом объектов, зависит от расстояния этих объектов до начала координат (камеры) Таким образом удается создавать трехмерные сцены, имитирующие присутствие тумана. Туман может быть полезным для медленного затенения объектов, “исчезающих” в фоне, что позволяет создавать мощные и реалистичные подсказки о глубине (depth cue) На рис. 35 показан результат выполнения программы FOGGED, как видите, это все тот же мир сфер с включенным туманом. В листинге 12 показано несколько строк кода, добавление которых в функцию `SetupRC` дает изображенный эффект.

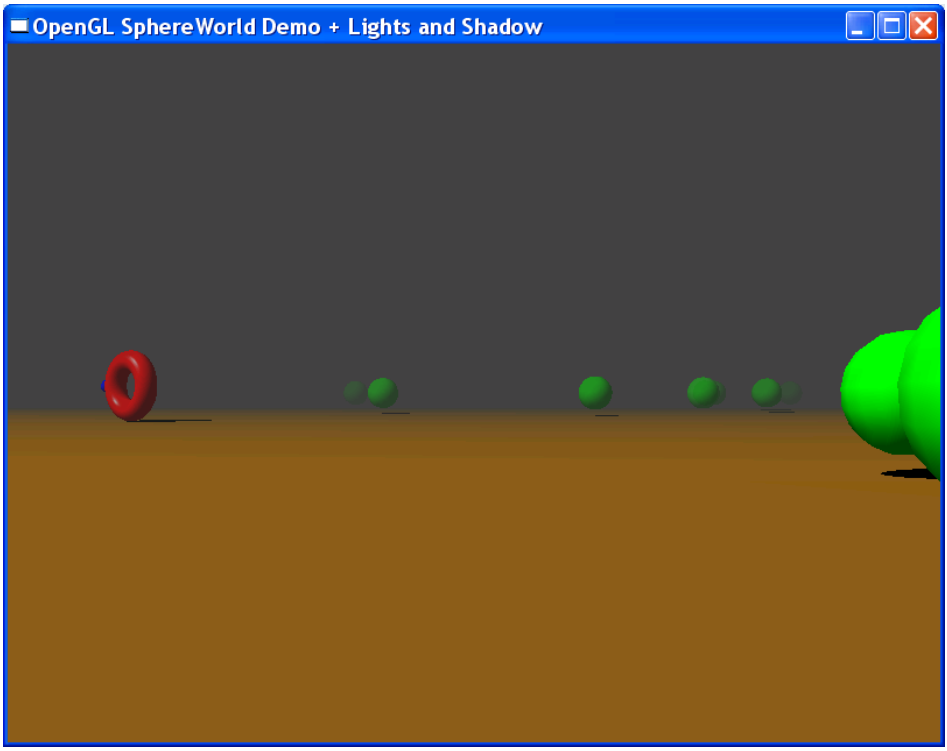

Рис. 35 – Мир сфер с туманом

Листинг 12 – Установка тумана в мире сфер

```
      // Сероватый фон
      glClearColor(fLowLight[0],fLowLight[1],fLowLigh
                  t[2], fLowLight[3]);
      // Установка параметров тумана
      glEnable(GL_FOG);           // Включается туман
      glFogfv(GL_FOG_COLOR, fLowLight);
      // Цвет тумана соответствует фону

      glFogf(GL_FOG_START, 5.0f); //    Насколько да-
леко
      //    начинается туман
      glFogf(GL_FOG_END, 30.0f);  //    Насколько да-
леко
      //    заканчивается туман
      glFogi(GL_FOG_MODE, GL_LINEAR); //     Какое
уравнение
      //    тумана используется
```

Для включения/выключения тумана применяется следующие функции: `glEnable/glDisable(GL_FOG);`

Для изменения параметров тумана (поведения тумана) используется функция `glFog`. Существует несколько вариантов этой функции.

```
void glFogi(GLenum pname, GLint param);
void glFogf(GLenum pname, GLfloat param);
void glFogiv(GLenum pname, GLint* params);
void glFogfv(GLenum pname, GLfloat* params);
```

Первым из проиллюстрированных вариантов является следующий:

```
glFogfv(GL_FOG_COLOR, fLowLight);
// Цвет тумана соответствует фону
```

При использовании с параметром `GL_FOG_COLOR` данная функция ожидает указатель на массив величин с плавающей запятой, который задает цвет тумана. В данном случае использовали туман цвета фона. Если цвет тумана не соответствует цвету фона, то объекты, спрятав-

шиеся в тумане, становятся силуэтами цвета тумана, проявляющимися на заданном фоне.

Таблица 4. Три уравнения тумана, поддерживаемых OpenGL

Режим тумана	Уравнение тумана
GL_LINEAR	f = (end - c)/(end - start)
GL_EXP	f = exp(-d * c)
GL_EXP2	$f = \exp(-(d * c)^2)$

Следующие две строки позволяют задать, насколько должен удалиться объект, чтобы на него начал действовать туман и объект полностью скрылся в тумане (приобретает цвет тумана)

```
glFogf(GL_FOG_START, 5.0f);
// Насколько далеко начинается туман
glFogf(GL_FOG_END, 30.0f);
// Насколько далеко заканчивается туман
```

Параметр GL_FOG_START задает, как далеко от глаза начинает проявляться эффект тумана, а GL_FOG_END — это расстояние от глаза, когда цвет тумана полностью подавляет цвет объекта. Переход от начала к концу тумана контролируется уравнением тумана, которое в данном случае задано с параметром GL_LINEAR.

```
glFogi(GL_FOG_MODE,GL_LINEAR);
//Какое уравнение тумана используется
```

Согласно данному уравнению, рассчитывается “коэффициент тумана”, который меняется от 0 до 1 на заданном участке (от начала тумана до конца) OpenGL поддерживает три уравнения тумана, которые приведены в табл. 3 - GL_LINEAR, GL_EXP и GL_EXP2.

В этих уравнениях

c — расстояние фрагмента от наблюдателя,

end — расстояние GL_FOG_END, а

start — расстояние GL_FOG_START

Значение d представляет собой плотность тумана.

Как правило, плотность тумана задается с помощью функции glFogf

```
glFogf(GL_FOG_DENSITY, 0.5f);
```

Сглаживание

Еще одной сферой, где применяются возможности смешения OpenGL, является сглаживание. В большинстве случаев отдельные визуализированные фрагменты отображаются в отдельные пиксели экрана компьютер.

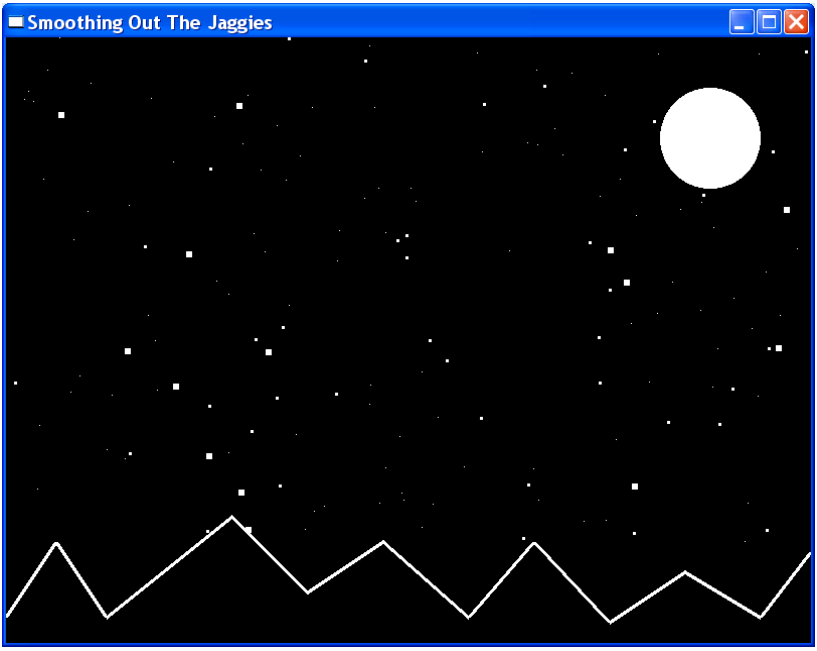

Рис. 36 – Результат выполнения программы

Эти пиксели квадратные (или прямоугольные), и обычно можно довольно отчетливо видеть границу раздела двух цветов. Эти неровности притягивают внимание и разрушают иллюзию естественного изображения. Они являются страшными предателями, сообщающими, что изображения сгенерированы компьютером, а ведь во многих задачах визуализации желательно получить максимальный реализм. На

рис. 36 показан результат выполнения программы, а на рис 37 приведено увеличенное изображение участка ломаной и нескольких точек-звезд, где отчетливо видны зазубренные края.

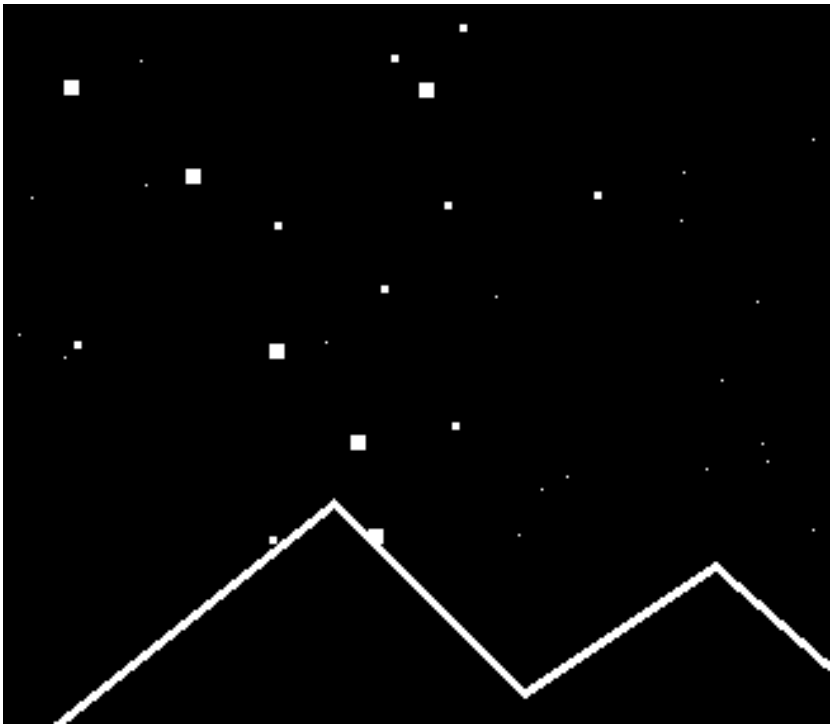

Рис. 37 – Увеличенное изображение зубцеобразных краев

Чтобы справиться с проблемой зазубренных краев примитивов, в OpenGL используется смешение и объединение цвета фрагмента с целевым цветом пикселя и его окружения. По сути, цвета пикселей слегка размываются на соседние пиксели вдоль краев примитивов.

Включить сглаживание достаточно просто. Вначале нужно активизировать смешение и указать, что функция смешения та же, что использовалась в предыдущем разделе для имитации прозрачности

```
glEnable(GL_BLEND);
  glBlendFunc(GL_SRC_ALPHA,
GL_ONE_MINUS_SRC_ALPHA);
```

Кроме того, нужно убедиться, что уравнение смешения задано с параметром GL_ADD, но поскольку это значение по умолчанию и оно является наиболее распространенным уравнением смешения, не будем приводить соответствующий код (кроме того, изменение уравнения смешения поддерживается не во всех реализациях OpenGL) После активизации смешения и выбора подходящей функции смешения, с помощью вызова glEnable можно выбрать сглаживание точек, линий и/или многоугольников (любых сплошных примитивов)

```
glEnable(GL_POINT_SMOOTH); // Сглаживание точек
glEnable(GL_LINE_SMOOTH);      // Сглаживание линий
glEnable(GL_POLYGON_SMOOTH);//Сглаживание краев многоугольников
```

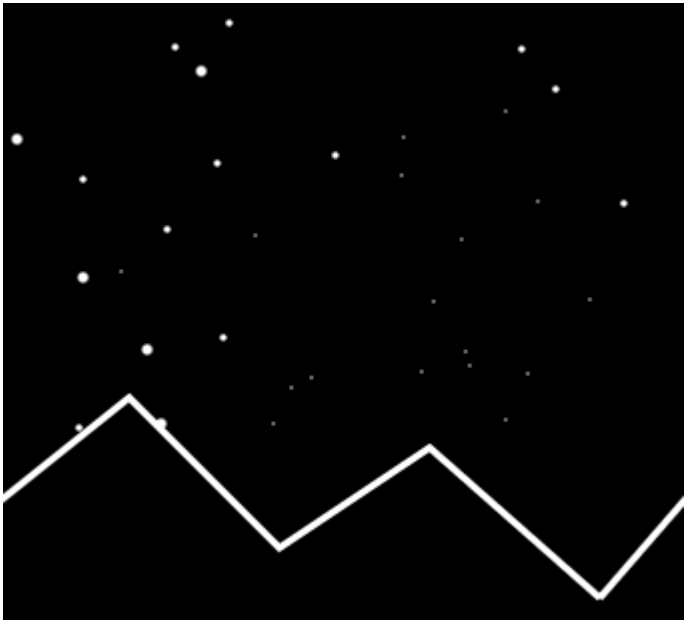

Рис. 38 – Зубцеобразных краев больше нет!

Однако следует отметить, что параметр GL_POLYGON_SMOOTH поддерживается не во всех реализациях OpenGL. В листинге 13

приведен код из программы, реагирующей на метку во всплывающем меню, которое позволяет пользователю переключаться между режимами визуализации с сглаживанием и без него. При запуске программы с активизированным сглаживанием точки и линии кажутся более гладкими (смазанными). На рис. 38 показана та же область, что и на рис. 37, но теперь с несколько уменьшенной зазубренностью краев.

Листинг 13 – Переключение между визуализацией с сглаживанием и обычной

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


// Массив небольших звезд
#define SMALL_STARS 100
typedef GLfloat GLTVector2[2]; // Двукомпонентный век-
тор с плавающей запятой
GLTVector2  vSmallStars[SMALL_STARS];

#define MEDIUM_STARS   40
GLTVector2 vMediumStars[MEDIUM_STARS];

#define LARGE_STARS 15
GLTVector2 vLargeStars[LARGE_STARS];

#define SCREEN_X    800
#define SCREEN_Y    600

////////////////////////////////////////////////////
////////////////
// В ответ на выбор позиции меню устанавливаются
// соответствующие метки
void ProcessMenu(int value)
```

```
    {
    switch(value)
        {
        case 1:
            // Включается сглаживание и дается подсказ-
      ка
            // обеспечить наивысшее качество
            glBlendFunc(GL_SRC_ALPHA,
GL_ONE_MINUS_SRC_ALPHA);
            glEnable(GL_BLEND);
            glEnable(GL_POINT_SMOOTH);
            glHint(GL_POINT_SMOOTH_HINT, GL_NICEST);
            glEnable(GL_LINE_SMOOTH);
            glHint(GL_LINE_SMOOTH_HINT, GL_NICEST);
            glEnable(GL_POLYGON_SMOOTH);
            glHint(GL_POLYGON_SMOOTH_HINT, GL_NICEST);
            break;

        case 2:
            // Выключается смешивание и сглаживание
            glDisable(GL_BLEND);
            glDisable(GL_LINE_SMOOTH);
            glDisable(GL_POINT_SMOOTH);
            glDisable(GL_POLYGON_SMOOTH);
            break;

        default:
            break;
        }

    // Активизируем перерисовывание изображения
    glutPostRedisplay();
    }


//////////////////////////////////////////////////
// Вызывается для перерисовки сцены
void RenderScene(void)
```

```
    {
    int i;                  // Счетчик цикла
    GLfloat x = 700.0f;     // Координаты и радиус луны
    GLfloat y = 500.0f;
    GLfloat r = 50.0f;
    GLfloat angle = 0.0f;   // Другая переменная для
цикла

    // Очистка окна
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    // Закрашиваем все белым цветом
    glColor3f(1.0f, 1.0f, 1.0f);

    // Рисуем небольшие звезды
    glPointSize(1.0f);
    glBegin(GL_POINTS);
        for(i = 0; i < SMALL_STARS; i++)
            glVertex2fv(vSmallStars[i]);
    glEnd();

    // Рисуем звезды среднего размера
    glPointSize(3.05f);
    glBegin(GL_POINTS);
        for(i = 0; i< MEDIUM_STARS; i++)
            glVertex2fv(vMediumStars[i]);
    glEnd();

    // Рисуем большие звезды
    glPointSize(5.5f);
    glBegin(GL_POINTS);
        for(i = 0; i < LARGE_STARS; i++)
            glVertex2fv(vLargeStars[i]);
    glEnd();

    // Рисуем Луну
    glBegin(GL_TRIANGLE_FAN);
        glVertex2f(x, y);
```

```
        for(angle = 0; angle < 2.0f * 3.141592f; angle
+= 0.1f)
            glVertex2f(x + (float)cos(angle) * r, y +
(float)sin(angle) * r);
            glVertex2f(x + r, y);
    glEnd();

    // Рисуем горизонт
    glLineWidth(3.5);
    glBegin(GL_LINE_STRIP);
        glVertex2f(0.0f, 25.0f);
        glVertex2f(50.0f, 100.0f);
        glVertex2f(100.0f, 25.0f);
        glVertex2f(225.0f, 125.0f);
        glVertex2f(300.0f, 50.0f);
        glVertex2f(375.0f, 100.0f);
        glVertex2f(460.0f, 25.0f);
        glVertex2f(525.0f, 100.0f);
        glVertex2f(600.0f, 20.0f);
        glVertex2f(675.0f, 70.0f);
        glVertex2f(750.0f, 25.0f);
        glVertex2f(800.0f, 90.0f);
    glEnd();

    // Переключение буферов
    glutSwapBuffers();
    }

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

    // Заселяем список звезд
    for(i = 0; i < SMALL_STARS; i++)
        {
```

```
        vSmallStars[i][0] = (GLfloat)(rand() %
SCREEN_X);
        vSmallStars[i][1] = (GLfloat)(rand() %
(SCREEN_Y - 100))+100.0f;
        }

    // Заселяем список звезд
    for(i = 0; i < MEDIUM_STARS; i++)
        {
        vMediumStars[i][0] = (GLfloat)(rand() %
SCREEN_X * 10)/10.0f;
        vMediumStars[i][1] = (GLfloat)(rand() %
(SCREEN_Y - 100))+100.0f;
        }

    // Заселяем список звезд
    for(i = 0; i < LARGE_STARS; i++)
        {
        vLargeStars[i][0] = (GLfloat)(rand() %
SCREEN_X*10)/10.0f;
        vLargeStars[i][1] = (GLfloat)(rand() %
(SCREEN_Y - 100)*10.0f)/ 10.0f +100.0f;
        }

    // Черный фон
    glClearColor(0.0f, 0.0f, 0.0f, 1.0f );

    // Устанавливаем белый цвет для рисования
    glColor3f(0.0f, 0.0f, 0.0f);
    }
void ChangeSize(int w, int h)
    {
    // Предотвращаем деление на ноль
    if(h == 0)
        h = 1;

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

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

      // Устанавливает объем отсечения с помощью отсе-
кающих
      // плоскостей (левая, правая, нижняя, верхняя,
      // ближняя, дальняя)
    gluOrtho2D(0.0, SCREEN_X, 0.0, SCREEN_Y);


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

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

      // Создаем меню
      glutCreateMenu(ProcessMenu);
      glutAddMenuEntry("Antialiased Rendering",1);
      glutAddMenuEntry("Normal Rendering",2);
      glutAttachMenu(GLUT_RIGHT_BUTTON);

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

      return 0;
      }
```

Существует множество алгоритмов и подходов, позволяющих получить сглаженные примитивы. В конкретной реализации OpenGL может выбираться любой из этих подходов; возможна поддержка даже двух вариантов. OpenGL можно запросить, поддерживается ли несколько алгоритмов сглаживания, и выбрать из них самый быстрый (`GL_FASTEST`) или наиболее точный (`GL_NICEST`).

ЗАДАНИЕ НА ЛАБОРАТОРНУЮ РАБОТУ

Воспроизвести результаты, представленные в теоретическом обзоре, применить различные виды освещения моделей, согласно варианту, полученному у преподавателя.

ТРЕБОВАНИЯ К РЕАЛИЗАЦИИ

Задание выполняется согласно варианту. По завершении готовится отчет.

1) Выполнить затенение согласно варианту.
2) Продемонстрировать на 6 примерах различные варианты работы функций glLightModelfv и glColorMaterial. Модели выбрать согласно варианту (Примечание: для создания эффекта реалистичности изображения рекомендуется получить координаты полигонов объектов из сторонних приложений.)
3) На основе объекта из задания 2 продемострировать установку источников света согласно варианту (Примечание: для тех кто претендует на оценку «отлично» предусмотреть включение-выключение источника света через меню.)
4) На основе объекта из задания 2 продемонстрировать создание бликов на объекте.
5) На основе объекта из задания 2 создать прожектор согласно варианту
6) На основе объекта из задания 2 создать тень согласно варианту
7) На основе листинга 6 (7) согласно варианту из лабораторной работы №3 продемонстрировать эффект отражения.
8) На основе листинга 6 и 7 согласно варианту из лабораторной работы №3 создать туман с разными параметрами. Должен посредством меню меняться цвет, режим и уравнение тумана.
9) На основе листингов 2, 3, 4 и 5 лабораторной работы №2 продемонстрировать работу механизма сглаживания.

ВАРИАНТЫ ЗАДАНИЙ

Задание 1.

1) Выполнить затенение прямоугольника с двумя одинаковыми по цвету смежными вершинами.
2) Выполнить затенение пирамиды с треугольным основанием.
3) Выполнить затенение пирамиды с квадратным основанием
4) Выполнить затенение квадрата
5) Выполнить затенение пятиугольника
6) Выполнить затенение куба
7) Выполнить затенение параллелепипеда
8) Выполнить затенение призмы с произвольным четырехугольным основанием
9) Выполнить затенение призмы с треугольным основанием
10) Выполнить затенение призмы с пятиугольным основанием

Задание 2.

1) Снеговик
2) Домик
3) Легковой автомобиль
4) Грузовой автомобиль
5) Животное на четырех ногах
6) Цветок типа «ромашка»
7) Велосипед детский трехколесный
8) Поезд
9) Рыба
10) Кисть руки человека

Задание 3.

1) Установить два источника света (находятся в противоположных октантах) одного цвета.
2) Установить три источника света (находятся в противоположных октантах) одного цвета.
3) Установить четыре источника света (два источника света находятся близко друг к другу, другие два в противоположных октантах) одного цвета.
4) Установить три источника света (противоположных) разного цвета.

5) Установить четыре источника света (противоположных) разного цвета.
6) Установить пять источников света (противоположных) разного цвета.
7) Установить три источника света (противоположных) разного цвета.
8) Установить пять источников света (находятся в трех октантах) одного цвета разной интенсивности.
9) Установить три источника света (находятся в смежных октантах) одного цвета разной интенсивности.
10) Установить три источника света (находятся в противоположных октантах) разного серого цвета.

Задание 5.

1) Создать один прожектор с углом 45: из меню выбирать не менее 5 цветов.
2) Создать 2 прожектора. Угол между прожекторами 45. Углы прожекторов 20 и 70 градусов: из меню выбирать не менее 5 цветов для каждого прожектора.
3) Создать 3 прожектора в разных октантах по 20, 45 и 60 градусов. Цвет должен быть разный в тонах серый.
4) Создать 3 прожектора разного цвета: углы 10, 15 и 40 градусов.
5) Создать 4 прожектора разного цвета: углы 10, 20, 30 и 45 градусов.
6) Создать 3 прожектора с углом между ними 120 градусов: по 40 градусов каждый: из меню выбирать не менее 5 цветов для каждого прожектора..
7) Создать 3 прожектора с углом между ними 60 градусов: по 30 градусов.
8) Создать 4 прожектора расположенных по сторонам квадрата: из меню выбирать не менее 3 разных углов для каждого прожектора.

9) Создать 3 прожектора с углами по 30 градусов, находящихся на разной высоте: из меню выбирать не менее 5 цветов для каждого прожектора.
10) Создать 4 прожектора с углами по 15 градусов, расположенных на разной высоте: из меню выбирать не менее 3 цветов для каждого прожектора

Задание 6.

1) Создать два источника света и отобразить две тени. Цвет земли изменить.
2) Создать два источника света. Плоскость земли состоит из двух плоскостей с углом между ними 90 градусов.
3) Создать три источника света. Земля плоская и имеет произвольный угол наклона.
4) Создать один источник света. Плоскость земли представляет сферу.
5) Создать три источника света. Объект заключен в куб. Отобразить тени на стороны куба.
6) Создать два источника света. Отобразить тени на плоскости с нормалью (1,1,0) и (-1,1,0).
7) Создать два источника света. Отобразить тень объекта на исходную плоскость: из меню выбирать включение и выключение каждого источника света.
8) Создать один источник света. Тень отображать на плоскость, выбираемую из меню. Продемонстрировать 3 разные плоскости.
9) Создать два источника света разные по интенсивности. Тень отобразить на исходную плоскость.
10) Создать вращающийся по окружности (эллипсу) в плоскость xz один источник света.

КОНТРОЛЬНЫЕ ВОПРОСЫ И ЗАДАНИЯ

1. Перечислите способы описания цветов в OpenGL.
2. Раскройте понятие *куб цветов* и покажите, какому цветовому пространству он соответствует.
3. Покажите, какой функцией OpenGL задается цвет объекта.
4. Опишите виды света, которыми может быть освещен неизлучающий объект.
5. Классифицируйте источники света, доступные в OpenGL.
6. Перечислите свойства материала объекта, существенные при его освещении.
7. Приведите алгоритм расчета нормали к поверхности, и раскройте роль этой величины.
8. Обоснуйте необходимость нормирования нормали (приведения к единичному виду).
9. Охарактеризуйте математический процесс используемый при моделировании тени.
10. Раскройте значение термина *смешение цветов*, и перечислите режимы смешения.
11. Приведите механизм, используемый в OpenGL для управления освещением.
12. Дайте определение коэффициент зеркального отражения материала.
13. Изложите концепцию прожектора и опишите его средствами OpenGL.
14. Приведите известные уравнения тумана.
15. Сформулируйте механизм и задачи сглаживания.

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) Листинги программ для задний 1-9, привести согласно варианту и продемонстрировать не менее 2-х снимков экрана для каждой реализации.

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

Отчет на защиту предоставляется в печатном виде.

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

ОСНОВНАЯ ЛИТЕРАТУРА

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

ловская. — Электрон. текстовые данные. — Екатеринбург: Уральский федеральный университет, 2016. — 112 c. — 978-5-7996-1942-8. — Режим доступа: http://www.iprbookshop.ru/68345.html

8. Перемитина, Т.О. Компьютерная графика : учебное пособие / Т.О. Перемитина ; Министерство образования и науки Российской Федерации, Томский Государственный Университет Систем Управления и Радиоэлектроники (ТУСУР). - Томск : Эль Контент, 2012. - 144 с. : ил.,табл., схем. - ISBN 978-5-4332-0077-7 ; - URL: http//biblioclub.ru/index.php?page=book&id=208688 (30.11.2017).

ДОПОЛНИТЕЛЬНАЯ ЛИТЕРАТУРА

Электронные ресурсы:

1. http://grafika.me/node/540 - Освещение Jcdtotybt OpenGL - примеры использования
2. https://www.youtube.com/watch?v=J2B9qKsit_o – Видеоурок по освещению OpenGL
3. http://compgraphics.info/OpenGL/lighting/light_sources.php - Источники света в OpenGl
4. http://www.codingclub.net/Articles/OpenGL/Materiali_i_osveschenie_v_OpenGL - Материалы и освещение в OpenGL