Министерство образования Республики Беларусь
Учреждение образования
«Белорусский государственный университет
информатики и радиоэлектроники»

Кафедра электронных вычислительных машин

Д.И. Самаль, В.А. Супонев, В.А. Прытков

МАШИННАЯ ГРАФИКА

Лабораторный практикум
для студентов специальности
1- 40 02 01
“Вычислительные машины, системы и сети“
всех форм обучения

Минск БГУИР 2009

УДК 681.3.01 (076)
ББК 32.973.2 я73
С17

Рецензент
зав. лабораторией машинной графики ОИПИ НАН Беларуси,
канд. техн. наук В.В. Ткаченко

Самаль Д.И.

С17 Машинная графика: лабораторный практикум для студ. спец. 1- 40 02 01 / Д. И. Самаль, В. А. Супонев, В. А. Прытков. – Минск: БГУИР, 2009. – 43 с.: ил.
ISBN 978-985-488-387-8

Лабораторные работы посвящены изучению алгоритмов построения отрезков и окружностей, отсечения невидимых линий и плоскостей, преобразованиям на плоскости и в пространстве.

В настоящем учебном пособии приведены задания по курсу «Машинная графика» для самостоятельного выполнения в рамках лабораторных работ.

Особое внимание уделено использованию открытого стандартного интерфейса к графической аппаратуре – OpenGL.

УДК 681.3.01 (076)
ББК 32.973.2 я73

ISBN 978-985-488-387-8

Содержание

Введение

Машинная графика (МГ) – это совокупность технических, математических и программных средств и приемов, позволяющих осуществить ввод и вывод из ЭВМ графической информации без ручного преобразования информации в числовую или графическую форму. Машинная графика используется во многих научных и инженерных дисциплинах, в бизнесе и кинематографии, в рекламном и издательском деле, в проектировании. Современная вычислительная техника активно использует элементы машинной графики для более комфортной работы пользователя. Практически всё современное программное обеспечение напрямую или опосредованно использует алгоритмы МГ.

Можно выделить три основных этапа формирования изображения электронной вычислительной машиной:

– построение модели объекта или сцены, содержащей несколько объектов (т.е. описание объектов и их связей в рамках евклидовой геометрии);

– подготовка модели к визуализации в зависимости от местонахождения наблюдателя (выполнение геометрических преобразований, удаление невидимых линий);

– визуализация с помощью заданного устройства отображения (отсечение по объёму/окну видимости, формирование растрового представления, наложение текстуры, затенение, добавление дополнительных эффектов, вывод на терминал).

Изображение можно представить в виде множества точек, линий, строк текста и закрашенных областей, называемых примитивами. Для задания многих объектов используют векторные модели. При этом изображение чаще всего описывается набором вершин, рёбер и граней, формирующих в итоге множество многоугольников с заданными атрибутами и представляющих в совокупности один или несколько объектов сцены. Например, квадрат можно описать четырьмя ребрами: E_1, E_2, E_3, E_4. Каждое ребро задается своими вершинами

$E_1=\{P_1,P_2\}$, $E_2=\{P_2,P_3\}$, $E_3=\{P_3,P_4\}$, $E_4=\{P_4,P_1\}$. Каждая вершина задается своими координатами $P_i=(x_i,y_i)$. При описании трехмерных объектов используются три координаты точки – $P_i=(x_i,y_i,z_i)$. Следует отметить, что визуализация изображений, как правило, выполняется на плоскости, т.е. итоговое изображение двухмерно, и, таким образом, при отображении трёхмерных объектов одним из обязательных шагов является проективное преобразование.

Периферийные устройства вывода изображений делятся по своему типу на растровые и векторные. При этом следует отличать «векторное изображение» как исходную для алгоритмов МГ векторную модель объекта или сцены, от изображения, сформированного устройством векторного типа, например, графопостроителем, который чертит изображение с помощью перьев разной толщины и цвета. Так как в настоящее время визуализация изображений производится в большинстве случаев именно растровыми устройствами вывода, то ниже по тексту термин «векторное изображение» будет употребляться как синоним понятия векторной модели, если иного не оговорено.

В растровых устройствах отображения точку заменяет пиксель (от английского *picture element*). Пиксель – это наименьшая часть изображения, с которой может работать алгоритм обработки либо визуализации изображения.

Изображения, отображаемые на растровых устройствах либо хранимые в виде двухмерного массива значений пикселей – растра, называются растровыми. Процесс преобразования векторных моделей в растровые изображения вследствие дискретности и конечности растра имеет определённые особенности. Краеугольным различием векторных и растровых алгоритмов вывода графической информации является невозможность выполнения на растре аксиом евклидовой геометрии. Например, отрезок, заданный двумя вершинами и имеющий бесконечное множество точек в евклидовой геометрии, в растровом представлении всегда состоит из конечного множества пикселей. Так же, например, два пикселя на растре могут принадлежать одновременно более чем одной прямой линии.

Линия – это связное множество точек. На растровом представлении используют два типа связности – четырехсвязность (соседними считаются только смежные по вертикали и горизонтали пиксели) и восьмисвязность (дополнительно «соседями» считаются диагонально смежные пиксели). Растровое представление линии – это связное множество пикселей, центры которых минимально отклоняются от непрерывного представления этой линии. Длиной линии принято считать количество пикселей её растрового представления.

Основные типы растровых изображений – это бинарные (используются только черный и белый цвета, 1 бит/пиксель), монохромные (например, зелёный и чёрный, 1 бит/пиксель), полутоновые (до 256 оттенков от черного до белого, 8 бит/пиксель), цветные (совокупность трех базовых цветов, каждый из которых трактуется как полутоновое изображение, – обычно 24 бит/пиксель). Каждый пиксель задается своими пространственными координатами и значением яркости/цвета согласно используемому типу изображения.

Лабораторный практикум, за исключением первой лабораторной работы, базируется на программном интерфейсе создания трёхмерной (3D) графики – OpenGL. OpenGL является открытым стандартом программного интерфейса к графической аппаратуре, который берёт своё начало от библиотеки графических функций IRIS фирмы Silicon Graphics. С 1 июня 1992 года – с момента выпуска своей первой спецификации – OpenGL контролируется специально созданным Советом наблюдения за архитектурой (Architecture Review Board – ARB) OpenGL. Среди основателей ARB, помимо Silicon Graphics, – DEC, IBM, Intel, Microsoft и другие производители аппаратного обеспечения для ПК.

OpenGL – это не язык программирования наподобие C++ или Java. Строго говоря, «программ OpenGL» не существует – следует говорить о программах, в которых в качестве одного из программных интерфейсов (API) используется OpenGL. Таким образом, писать программы с использованием OpenGL можно на многих языках высокого уровня, поддерживающих вызов функций библиотек OpenGL, которые, в свою очередь, являются кроссплатформенными.

ЛАБОРАТОРНАЯ РАБОТА №1.
АЛГОРИТМЫ БРЕЗЕНХЕМА ДЛЯ ПОСТРОЕНИЯ ОТРЕЗКА И ОКРУЖНОСТИ

Цель работы: освоить алгоритмы формирования растровых представлений произвольного отрезка прямой и окружности.

1.1. Постановка задачи

Сгенерировать двухмерное изображение, содержащее заданное количество отрезков различной длины и заданное количество окружностей различного радиуса с помощью самостоятельно разработанных функций. Для программной реализации функций разложения прямой и окружности в растр разрешается использовать только целочисленные алгоритмы.

1.2. Алгоритм разложения отрезка в растр

Рассмотрим уравнение идеальной прямой, проходящей через две точки,

$$y = y_1 + (x-x_1) *(y_2-y_1)/(x_2-x_1) = y_1 + (x-x_1)*\Delta y/\Delta x,$$

где Δy – разность y-координат концов отрезка,

Δx – разность x-координат.

Если x и y изменяются вдоль прямой дискретно на δ_x и δ_y, тогда $y_i+1=y_i+\delta_y = y_i + \delta_x*\Delta y/\Delta x$. Пусть δ_x и/или δ_y равняется величине пикселя, т.е., 1.

Растровое представление отрезка прямой – это связное множество пикселей, имеющих наименьшее отклонение от идеальной прямой. При этом может использоваться 4- и 8-связность. Точное растровое представление (4- и 8-связное) можно построить только для вертикального и горизонтального отрезка. Для них $\delta x = 1$, $\delta y = 0$, либо $\delta x = 0$, $\delta y = 1$. Для отрезка, расположенного под углом 45° (135°) к горизонтальной оси, можно построить точное 8-связное растровое представление. Для него $\delta x = \delta y = 1$. Растровое представление всех остальных отрезков выглядит в виде ступенчатой последовательности пикселей (рис. 1.1).

В 1965 г. Дж. Брезенхем (J. Bresenham) опубликовал целочисленный алгоритм формирования растрового представления произвольного отрезка прямой. Пусть для растрового представления отрезка выбран некоторый пиксель, например имеющий координаты (x_1, y_1). Как выбрать следующий соседний пиксель? Алгоритм использует понятие функции, оценивающей величину отклонения выбираемого пикселя от идеальной прямой. Пусть $0<\Delta y<\Delta x$, тогда фиксируем $\delta x = 1$ (т. е. $x_i = x_i+1$) и оцениваем, пиксель с какой у-координатой ($y_i+1=y_1$ или $y_i+1=y_1+1$) расположен ближе к идеальной прямой. Выбирается тот пиксель, который имеет меньшее (из двух) значение отклонения, т.е. оценочной функции. При этом отклонение ближайшего пикселя не превышает 1/2.

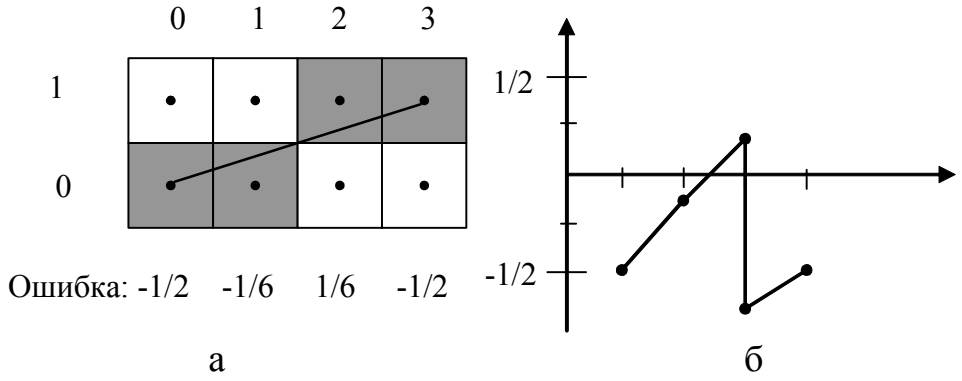


Рис.1.1. Отрезок с вершинами (0,0) и (3,1).

а) – его растровое представление.

б) – график функции «ошибки» растрового представления отрезка по отношению к его идеальному положению (при инициализации нулевого значения ошибки значением $f_0 = -1/2$).

Основа алгоритма – это движение вдоль основной оси на один пиксель и поддержание текущего отклонения от идеальной прямой в заданных пределах. Если текущее отклонение превышает норму, то шаг делается по неосновной оси, соответственно отклонение уменьшается на единицу. Идея алгоритма основывается на использовании понятия «ошибки». В каноническом случае, т.е. при построении отрезка в первом октанте с началом в центре системы коорди-

нат, процесс рисования 8-связной произвольной прямой линии можно закодировать последовательностью следующего вида: sdssd…, где s – горизонтальное смещение, а d – диагональное смещение от пикселя к пикселю. «Ошибкой» в таком случае будет называться расстояние между значением ординаты (при фиксированном значении x) идеального отрезка, который алгоритм стремится изобразить, и значением ординаты «центра» ближайшего пикселя (с тем же фиксированным значением x), который и будет в результате закрашен.

Алгоритм реализован таким образом, что в случае построения 8-связной линии, одна координата (в каноническом случае – x) изменяется на единицу на каждом шаге, а другая либо остаётся без изменений, либо также изменяется на единицу. Пример подобного построения приведён на рисунке 1.1 а), перемещение вдоль растрового построения можно закодировать как – «sds» .

Так как значение «ошибки» на каждом шаге не может превышать 1/2 расстояния между центрами соседних пикселей, которые претендуют на право быть закрашенными на текущем шаге, то можно изначально «сместить» значение «ошибки» на −1/2. Это позволит принимать решение – оставлять вторую координату без изменений либо инкрементировать её – на основе проверки знака текущего значения «ошибки», вместо сравнения её значения с 1/2.

С целью увеличения быстродействия алгоритма можно использовать целочисленную арифметику и исключить операции деления. Для этого значение ошибки, вычисляемой по формуле

$$f = \Delta y/\Delta x - 0{,}5 \,, \tag{1.1}$$

надо умножить на $2\Delta x$, тогда

$$2\Delta x * f = 2\Delta y - \Delta x \,. \tag{1.2}$$

Обозначив $F = 2\Delta x * f$, получим

$$F = 2\Delta y - \Delta x \,. \tag{1.3}$$

С помощью полученного выражения можно вычислить начальное значение ошибки. В цикле прорисовки отрезка в первом октанте её значение будет вычисляться в соответствии с выражением 1.4

$$F_{i+1} = F_i + 2\Delta y. \quad (1.4)$$

Целочисленный алгоритм Брезенхема для канонического случая построения отрезка можно сформулировать в виде последовательностей операций:

1. Инициализация нулевого значения ошибки в соответствии с (1.3).

2. Цикл по значению x:

2.1. Отображение пикселя с текущими координатами.

2.2. Модификация значения ошибки в соответствии с (1.4), инкрементация x.

2.3. В случае, если $F \geq 0$, коррекция $F = F - 2\Delta x$ и инкрементация y.

3. Если x равен значению конца отрезка – x_2, то конец алгоритма, отрезок отображён, если иначе – повторить шаг 2.

Следует отметить, что в общем случае не все отрезки, изображаемые на экране, начинаются в центре системы координат и лежат целиком в первом октанте. Соответственно, приведённый алгоритм не является универсальным. Для построения отрезка с любыми конечными координатами можно воспользоваться геометрическими преобразованиями на плоскости – переносом и отражением, либо разработать общий алгоритм, который будет учитывать положение отрезка в заданной системе координат. Общий алгоритм должен выбирать основную ось (т.е. ту, координаты которой будут изменяться на каждом шаге в зависимости от величины углового коэффициента), а также учитывать направление движения, инкрементируя либо декрементируя координаты по соответствующим осям. Общий вариант целочисленного алгоритма Брезенхема предлагается реализовать в программном виде самостоятельно на основе материалов лекций и приведенной литературы.

1.3. Алгоритм растеризации окружности

Окружность задается уравнением

$$X^2 + Y^2 = R^2, \quad (1.5)$$

отсюда

$$Y = \pm\sqrt{R^2 - X^2}. \quad (1.6)$$

Использование выражения (1.6) для вычисления координаты Y неэффективно, т. к. приходится использовать арифметику с плавающей запятой и извлечение квадратного корня. Кроме этого, производя приращение *x* координаты на фиксированную дельту, получить непрерывное представление окружности не получится (рис. 1.2). Нужный эффект достигается при использовании алгоритма Брезенхема для построения окружности.

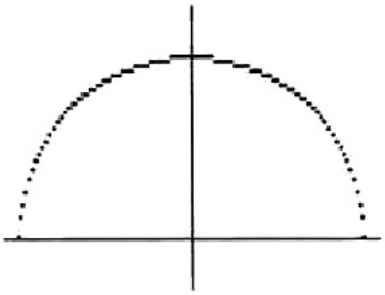
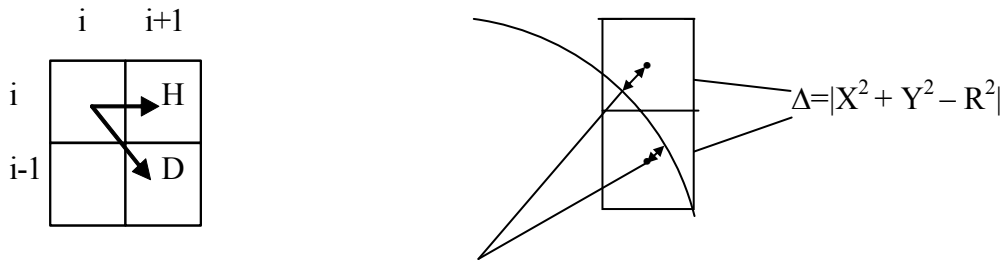


Рис. 1.2. Результат построения растрового представления окружности в соответствии с выражением 1.6

Для прорисовки окружности достаточно построить только 1/8 её часть. Остальные части получаются зеркальным отражением.

Алгоритм Брезенхема вычисляет часть окружности, лежащую в первом квадранте, однако можно рассмотреть упрощённый вариант построения дуги, лежащей во втором октанте. В этом алгоритме используется понятие «ошибки» – расстояния от центра пикселя до действительного положения окружности (рис. 1.3).



а б

Рис. 1.3. Отрисовка окружности: а – Пиксели-кандидаты на прорисовку на следующей итерации, б – значения ошибки (для пикселей-кандидатов)

Вычисление координат очередного пикселя базируется на координатах ранее вычисленного пикселя. Пусть вычисленный пиксель имел координаты (x_i, y_i). Тогда в случае построения дуги окружности с началом в точке с координатами *(0, R)* и движением по часовой стрелке, «кандидатами» на прорисовку на следующем шаге будут два пикселя – H и D (рис. 1.3).

Реальная окружность может быть расположена относительно центров пикселей-кандидатов, обозначенных точками T и S, одним из способов 1 – 5.

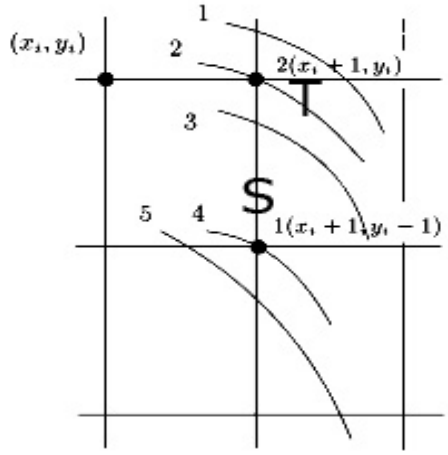


Рис. 1.4. Варианты прорисовки окружности

Если мы выбираем точку S, то тем самым считаем, что $(x_i+1)^2+(y_i-1)^2 \approx R^2$. Если же выбираем точку T, то допускаем, что $(x_i+1)^2+(y_i)^2 \approx R^2$. Рассмотрим две погрешности Δ^i_1 и Δ^i_2:

$$\Delta^i_1 = (x_i+1)^2+(y_i-1)^2-R^2 \; ; \qquad \Delta^i_2 = (x_i+1)^2+(y_i)^2-R^2 \; ; \qquad (1.7)$$

и контрольную величину $\Delta^i = \Delta^i_1+\Delta^i_2$.

При выборе точки, следующей за (x_i, y_i), станем руководствоваться следующим критерием: если $\Delta^i > 0$, выберем точку S; если $\Delta^i \le 0$, выберем точку T.

Основанием для использования подобного критерия являются знаки погрешностей Δ^i_1 *и* Δ^i_2 в каждом из пяти возможных случаев:

– для положения 1: $\Delta^i_1 < 0, \Delta^i_2 < 0 \Rightarrow \Delta^i_1 + \Delta^i_2 < 0 \Rightarrow$ выбирается T;

– для положения 2: $\Delta^i_1 < 0, \Delta^i_2 = 0 \Rightarrow \Delta^i_1 + \Delta^i_2 < 0 \Rightarrow$ выбирается T;

– для положения 3 возможны варианты (учитывая, что $\Delta^i_1 < 0, \Delta^i_2 > 0$):

а) $|\Delta^i_1| \ge |\Delta^i_2| \Rightarrow \Delta^i_1 + \Delta^i_2 \le 0 \Rightarrow$ выбирается T;

б) $|\Delta^i_1| < |\Delta^i_2| \Rightarrow \Delta^i_1 + \Delta^i_2 > 0 \Rightarrow$ выбирается S;

– для положения 4: $\Delta^i_1 = 0,\ \Delta^i_2 > 0 \Rightarrow\ \Delta^i_1 + \Delta^i_2 > 0 \Rightarrow$ выбирается S;

– для положения 5: $\Delta^i_1 > 0,\ \Delta^i_2 > 0 \Rightarrow\ \Delta^i_1 + \Delta^i_2 > 0 \Rightarrow$ выбирается S.

Контрольная величина Δ^i может быть вычислена в соответствии с выражением

$$\Delta^i = \Delta^i_1+\Delta^i_2 = (x_i+1)^2+(y_i-1)^2-R^2+(x_i+1)^2+(y_i)^2-R^2 = 2x_i^2+2y_i^2+4x_i-2y_i+3-2R^2. \quad (1.8)$$

Выражение для Δ^{i+1} существенным образом зависит от выбора следующего закрашиваемого пикселя из двух возможных кандидатов – H и D. Соответственно, подставляя $y_{i+1} = y_i$ в выражение 1.8 в случае выбора пикселя H, получим

$$\Delta^{i+1}\ [при\ y_{i+1} = y_i] = 2(x_i+1)^2+2y_i^2+4(x_i+1)-2y_i+3-2R^2 = \Delta^i+4x_i+6\ . \quad (1.9)$$

В случае закраски на следующем шаге пикселя D – $y_{i+1} = y_i - 1$, тогда:

$$\Delta^{i+1}\ [при\ y_{i+1} = y_i-1] = 2(x_i+1)^2+2(y_i-1)^2+4(x_i+1)-2(y_i-1)+3-2R^2\ ,$$

$$\Delta^{i+1}\ [при\ y_{i+1} = y_i-1] = \Delta^i+4(x_i-y_i)+10\ . \quad (1.10)$$

Если обозначить $\boldsymbol{u} = 4x_i+6$, и $\boldsymbol{v} = 4(x_i-y_i)+10$, то алгоритм можно записать в следующем виде:

Выбор T: если $(\Delta^i \le 0)$, то $x=x+1$;

Выбор S: если $(\Delta^i > 0)$, то $x=x+1,\ y= y-1$.

Вычисление ошибки на следующем шаге:

T: $\boldsymbol{u} = 4x_i+6$, $\quad \Delta^{i+1} = \Delta^i + \boldsymbol{u}$;

S: $\boldsymbol{v} = 4(x_i-y_i)+10$, $\quad \Delta^{i+1} = \Delta^i + \boldsymbol{v}$.

Значение ошибки для первого шага можно вычислить, если вспомнить, что $x_1 = 0$, $y_1 = R$, тогда

$$\Delta^1_1 = (0+1)^2+(R-1)^2-R^2 = 2-2R,$$

$$\Delta^1_2 = (0+1)^2+R^2-R^2 = 1,$$

$$\Delta^1 = \Delta^1_1+\Delta^i_2. = 3-2R.$$

Итоговая блок-схема алгоритма построения 1/8 части окружности приведена на рис. 1.5.

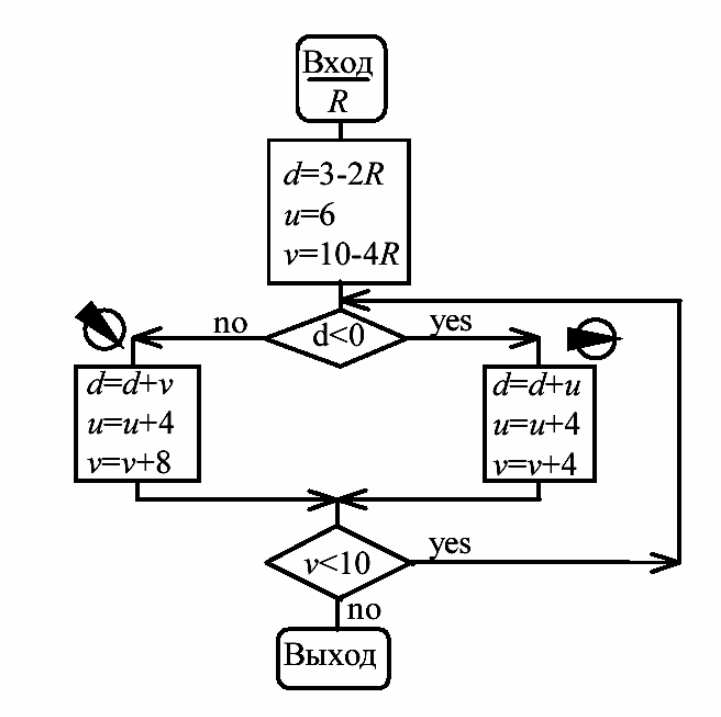


Рис. 1.5. Блок-схема алгоритма построения 1/8 части окружности

1.4. Задание к лабораторной работе

В рамках лабораторной работы выполнить следующую последовательность операций:

1. Программно реализовать целочисленные алгоритмы Брезенхема для растеризации отрезка (в общем случае) и окружности.

2. Используя разработанные функции, сгенерировать и вывести на экран осмысленную двухмерную сцену, содержащую как минимум 30 – 40 отрезков и 10 – 15 окружностей различного диаметра.

1.5. Контрольные вопросы

1. В чем состоит идея алгоритма Брезенхема для построения растрового представления отрезка/окружности?

2. Каковы достоинства алгоритма Брезенхема? Как можно улучшить алгоритм?

3. Всегда ли совпадают растровые представления отрезков, заданных координатами $(x_1, y_1) – (x_2, y_2)$ и $(x_2, y_2) – (x_1, y_1)$?

4. Будут ли отличаться растровые представления отрезка $(x_1, y_1) – (x_2, y_2)$ и отрезков а) $(x_1+e, y_1) – (x_2+e, y_2)$;

б) $(x_1+e, y_1+e) – (x_2+e, y_2+e)$;

в) $(x_1, y_1-e)- (x_2, y_2-e)$,

где e – константа $0<e<1$?

5. Накапливается ли ошибка в процессе выполнения алгоритма Брезенхема?

6. С какой точностью можно восстановить уравнение непрерывной прямой по растровому представлению отрезка?

7. Всегда ли совпадают растровые представления окружностей, заданных координатами $(x_1, y_1) – (x_2, y_2)$ и $(x_2, y_2) – (x_1, y_1)$?

8. Что проще в вычислительном плане: сформировать растровое представление окружности с помощью а) алгоритма Брезенхема для вычерчивания отрезка путем отображения правильного N-угольника, аппроксимирующего окружность с нужной точностью, или б) алгоритма Брезенхема для построения растрового представления окружности?

9. Будут ли различаться растровые представления окружностей, если:

а) их радиусы отличаются на e;,

б) координаты их центров отличаются на e,

где e – константа $0<e<1$?

1.6. Литература по теме работы

1. Роджерс, Д. Алгоритмы машинной графики/ Д. Роджерс. – М.: Мир, 1989, §§2.1 – 2.6.

2. Фоли, Дж. Основы интерактивной машинной графики/ Дж. Фоли, А. вэн Дэм. – М.: Мир, 1985, т. 2, §§11.2, 11.4.

3. Херн, Д. Микрокомпьютерная графика и стандарт OpenGL/ Д. Херн, М.Бейкер. – М.: “Вильямс”, 2005, §§3.4, 3.5, 3.9.

4. Лабораторный практикум по курсу «Машинная графика»/ Р.Х. Садыхов [и др.]. – Минск: БГУИР, 2002.

ЛАБОРАТОРНАЯ РАБОТА №2.
ГЕНЕРАЦИЯ ДВУХМЕРНЫХ СЦЕН С ПОМОЩЬЮ OpenGL

Цель работы: освоить функции инициализации, построения примитивов, визуализации программного интерфейса OpenGL.

2.1. Постановка задачи

Сгенерировать двухмерное изображение осмысленной сцены заданной сложности с помощью функций программного интерфейса OpenGL.

2.2. Базовые функции OpenGL

Основные функции интерфейса OpenGL изначально были представлены в виде основной платформонезависимой библиотеки GL (от Graphics Library – графическая библиотека) и набора библиотек AGL, GLX, WGL, отвечающих за интерфейс с различными оконными системами – Apple, X-Window, Windows соответственно, на которых производилось выполнение итоговых программ. Впоследствии к интерфейсу OpenGL добавилась библиотека GLU – OpenGL Utility Library, расширяющая графические возможности базовой библиотеки, а набор специализированных под упомянутые системы библиотек был трансформирован в единый переносимый оконный интерфейс – GLUT (OpenGL Utility Toolkit). Следует отметить, что функции основных библиотек OpenGL поддерживаются Windows на системном уровне и устанавливать библиотеки при написании Windows программ, использующих интерфейс OpenGL, нет необходимости (справедливо для функций стандарта OpenGL v1.1). В таком случае, однако, текст программы на C/C++/C# должен начинаться с подключения соответствующих заголовочных файлов:

```
#include <GL/gl.h>
#include <GL/glu.h>
```

В случае, если разрабатывается переносимый вариант графической программы, то для визуализации необходимо использовать функции библиотеки GLUT (её можно без затруднений скачать из Интернета, например, отсюда –

[см. 2.4 п.3]). Программа, соответственно, должна начинаться с подключения заголовочного файла библиотеки:

```
#include <GL/glut.h>.
```

Заголовочные файлы «gl.h» и «glu.h» явно подключать необязательно – при использовании «glut.h» они будут подключены автоматически.

Типы данных, используемые на разных системах, могут существенно различаться. Для обеспечения переносимости программ в OpenGL определены собственные перенумерованные типы, которые будут оставаться неизменяемыми при выполнении программы на каждой из требуемых систем:

GLbyte – *8-битовое целое, соответствует типу* ***signed char*** *в C/C++,*
GLshort – *16-битовое целое, соответствует типу* ***short*** *в C/C++,*
GLint, GLsizei – *32-битовое целое –* ***long*** *в C/C++,*
GLfloat, GLclampf – *32-битовое вещественное –* ***float*** *в C/C++,*
GLdouble, GLclampd – *64-битовое вещественное –* ***double*** *в C/C++,*
GLubyte, GLboolean – *8-битовое целое без знака –* ***unsigned char,***
GLushort – *16-битовое целое, –* ***unsigned short*** *в C/C++,*
GLuint, GLenum – *32-битовое целое без знака –* ***unsigned long,***
GLbitfield – *поля двоичных разрядов*.

Все типы данных, используемые в программе, должны начинаться с заглавных символов GL. Типы GLsizei и GLclamp(f/d) введены для более удобной «читаемости» исходных кодов программ, использующих библиотеки OpenGL. GLsizei принято использовать для переменных, содержащих целочисленные значения размеров или глубины. Имена GLclampf/GLclampd «подсказывают», что соответствующие переменные будут ограничены согласно допустимому диапазону 0,0 – 1,0 (clamped – «зажато», англ.).

Массивы и указатели обозначаются аналогично нотации C/C++:

```
GLint  ndigits[10]; // Массив из 10 переменных типа GLint;
GLdouble *doubles[10]; // Массив из 10 указателей на переменные типа GLdouble.
```

Названия функций OpenGL представляют собой составные конструкции, которые начинаются с прописных символов имени библиотеки, к которой они относятся, – *gl, glu, glut, glt, aux и т.д.*, и имеют следующий формат: *<Префикс библиотеки> <Имя команды> <Необязательный счётчик аргументов> <Необязательный тип аргументов>* (рис. 2.1).

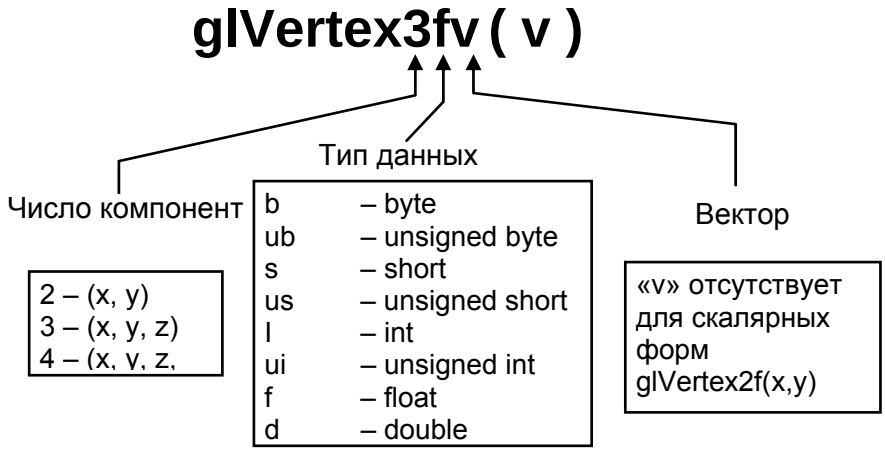


Рис. 2.1. Пример расшифровки названия функции OpenGL

Все отображаемые с помощью OpenGL сцены состоят из ограниченного набора примитивов: GL_POINTS, GL_LINE_STRIP, GL_LINE_LOOP, GL_LINES, GL_TRIANGLES, GL_TRIANGLE_STRIP, GL_TRIANGLE_FAN, GL_QUADS, GL_QUAD_STRIP, GL_POLYGON (рис. 2.2).

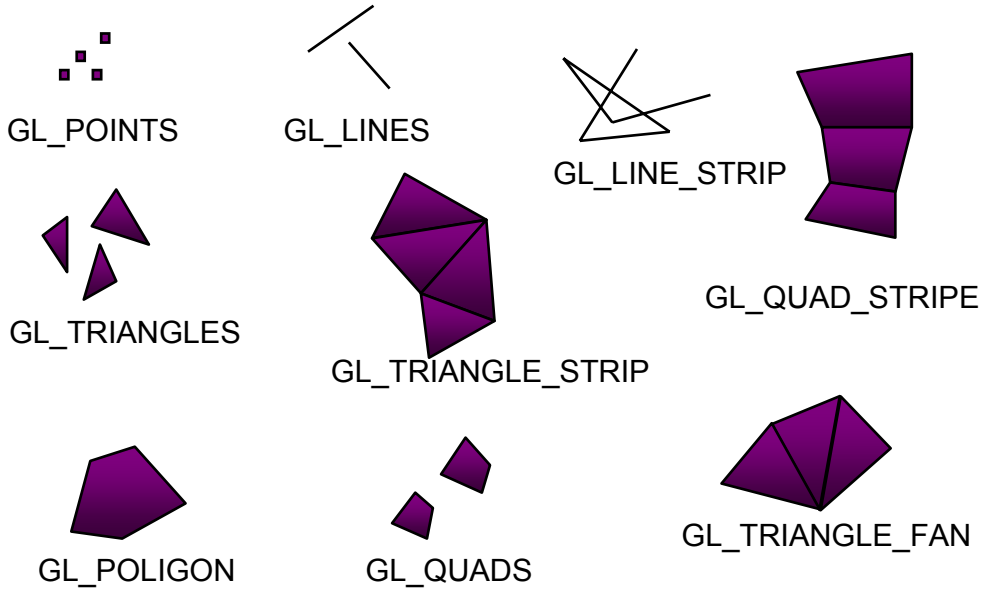


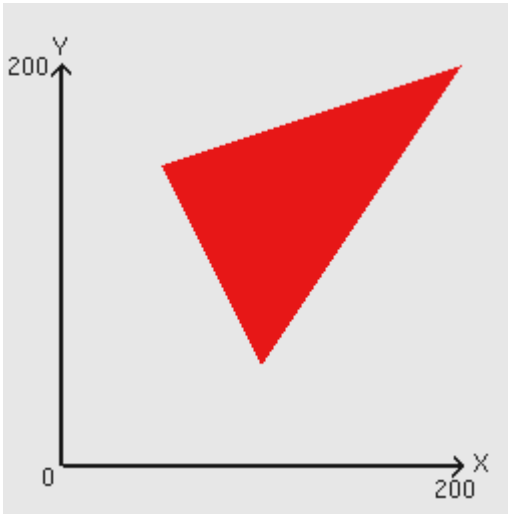
Рис. 2.2. Примеры наиболее часто используемых примитивов OpenGL

Для задания примитивов используется конструкция

```
glBegin( PRIMITIVE_TYPE );
// список вершин
glEnd();
```

где *PRIMITIVE_TYPE* – константа из указанного выше перечня, определяющая последовательность соединения вершин.

```
glBegin(GL_TRIANGLES);
  glColor3f( 1.0, 0.0, 0.0);
  glVertex2f(150.0f, 50.0f);
  glVertex2f(50.0f, 150.0f);
  glVertex2f(200.0f, 200.0f);
glEnd();
```



а б

Рис. 2.3. Пример рисования треугольника.

а) – часть кода программы, б) – отображение треугольника

Если рассматривать минимальный состав консольной программы ОС Windows, использующей функции библиотеки OpenGL, то код функции main на языке C/C++ будет иметь вид

```
void main (void)
{   glutInitDisplayMode(GLUT_SINGLE, GLUT_RGB);
    glutCreateWindow("Пример");
    glutDisplayFunc(MyDrawFunction);
    MyInitFunction();
    glutMainLoop();
}
```

Из приведённых выше функций библиотеки GLUT первая отвечает за инициализацию режима отображения – в данном случае с использованием одного пиксельного буфера (GLUT_SINGLE) и цветовой системы координат RGB (GLUT_RGB). Функция *glutCreateWindow("Пример")* создаёт окно с заголовком «Пример», в котором будет производиться вывод сцены.

Функция *glutDisplayFunc()* устанавливает с помощью механизма обратного вызова указатель на авторскую функцию *MyDrawFunction()* для её вызова и выполнения при каждой необходимости отрисовки окна. Именно в функции *MyDrawFunction()* производится задание элементов сцены, которые будут отображены. Дополнительной авторской функцией является *MyInitFunction()*,

вызываемая однократно для инициализации состояния OpenGL. В принципе, команды инициализации OpenGL, содержащиеся в этой функции, не обязательно оформлять в виде отдельного функционального модуля, однако с точки зрения хорошего стиля программирования лучше их всё же обособить.

Последней командой должна являться *glutMainLoop()*, вызываемая однократно и выполняемая до завершения программы. Она запускает все механизмы OpenGL, обрабатывает все сообщения ОС, нажатий клавиш и т.п.

В свою очередь, функция инициализации может содержать минимум команд:

```
void MyInitFunction(void)
{
  glClearColor(0.0f, 0.0f, 1.0f, 1.0f);
  gluOrtho2D (0.0, 400.0, 0.0, 300.0);
}
```

В данном случае с помощью *glClearColor()* при инициализации задаётся фоновый цвет экрана – так называемый «цвет очистки экрана». Функция *gluOrtho2D()* задаёт план проекции и диапазоны значений X и Y в этой проекции – без функций подобного назначения сцена будет оставаться «непривязанной» к окну отображения, со всеми вытекающими последствиями.

Тело функции *MyDrawFunction()* должно содержать как минимум вызовы двух функций OpenGL – *glClear()* и *glFlush()*, между которыми производится задание объектов сцены:

```
void MyDrawFunction(void)
{
  glClear(GL_COLOR_BUFFER_BIT); //Очистка – т.е., заливка
              // пиксельного буфера цветом очистки экрана
  glBegin(GL_TRIANGLES);
    glColor3f( 1.0, 0.0, 0.0);
    glVertex2f(150.0f, 50.0f);
    glVertex2f(50.0f, 150.0f);
    glVertex2f(200.0f, 200.0f);
  glEnd();
  glFlush();  // Указание выполнения всех невыполненных конвейером
              // OpenGL команд
}
```

Если собрать приведённые выше части программы в единый файл, скомпилировать её и запустить на выполнение, то в окне программы будет выведен треугольник, отображённый на рис. 2.3, б.

Метод обратного вызова часто применяется в OpenGL, в частности:

– *glutReshapeFunc(MyChangeSizeFunction)* позволяет вызвать в случае изменения размеров окна авторскую функцию *MyChangeSizeFunction()*;

– *glutKeyboardFunc(MyKeyProcessingFunction)* вызовет в случае нажатия ASCII-клавиши клавиатуры обработчик события – *MyKeyProcessingFunction();*

– *glutMouseFunc(MyMouseProcessingFunction)* в случае поступления сообщения от «мыши» вызовет функцию *MyMouseProcessingFunction();*

– *glutSpecialFunc(MySpecialKeyProcessingFunction)* при нажатии функциональной клавиши вызовет функцию *MySpecialKeyProcessingFunction()*;

– *glutTimerFunc(50, MyTimerProcessingFunction, 1)* регистрирует функцию MyTimerProcessingFunction(), которая будет вызвана по прошествии 50 ms.

В OpenGL возможны два способа добавления динамики в сцену: 1) изменение координат точки (и/или других параметров) наблюдения и 2) анимация – изменение взаимного расположения объектов сцены. В простейшем случае подобные изменения можно производить с помощью динамического пересчёта координат отображаемого объекта и вызова функции перерисовки изображения по прошествии заданного интервала времени.

Принудительное обновление текущего окна осуществляется с помощью вызова функции *glutPostRedisplay()*. Для многократного вызова функции, пересчитывающей координаты отображаемых объектов, необходимо использовать рекурсивный вызов функции *glutTimerFunc()*, так как таймер в OpenGL срабатывает при вызове однократно. Пример подобной рекурсии:

```
void   MyTimerProcessingFunction (int value)
{
    x1 ++; y1 ++;  // Инкрементация координат вершин треугольника
    x2 ++; y2 ++;
    glutPostRedisplay();
    glutTimerFunc(50, MyTimerProcessingFunction,1);
}
```

Соответственно, первый вызов *glutTimerFunc(50, MyTimerProcessingFunction,1)* должен находиться в основной функции программы – *main()*.

Для более плавного отображения анимации динамические сцены рекомендуется отображать с использованием двойной буферизации, для чего константу GLUT_SINGLE следует заменить на GLUT_DOUBLE, а функцию *glFlush()* – на *glutSwapBuffers()*.

2.3. Примеры типовых заданий к лабораторной работе

1. Разработать программу, использующую функции библиотек OpenGL, отображения в заданном окне движущихся по произвольным траекториям различных фигур – треугольников, квадратов, ломаных линий, лент из треугольников и четырёхугольников.

2. Используя функции библиотек OpenGL, разработать программу, аналогичную стандартному скринсейверу Windows, в котором множество ломаных линий (замкнутых и/или не замкнутых) двигается в границах окна по единой траектории с заданным шагом/задержкой, отражаясь от границ окна.

3. Разработать программу отображения произвольной осмысленной двухмерной сцены, содержащей не менее 50 разных примитивов OpenGL, часть которых должна изменять своё положение с течением времени.

2.4. Литература по теме работы

1. Херн, Д. Микрокомпьютерная графика и стандарт OpenGL/ Д. Херн, М.Бейкер. – М.: “Вильямс”, 2005, §§3.2 - 3.4.

2. Райт-мл., Р.С. OpenGL суперкнига/ Р.С. Райт-мл., Б. Липчак. – М.: “Вильямс”, 2005, §§2-3.

3. The Industry's Foundation for High Performance Graphics. OpenGL. [Электронный ресурс]. – 1992–2009. – Режим доступа: http://www.opengl.org/resources/libraries/glut/

ЛАБОРАТОРНАЯ РАБОТА №3. ГЕОМЕТРИЧЕСКИЕ ПРЕОБРАЗОВАНИЯ В ОДНОРОДНЫХ КООРДИНАТАХ. ПРОЕЦИРОВАНИЕ

Цель работы: научиться выполнять сдвиг, поворот, симметричное отражение, масштабирование и комбинированные преобразования объектов на плоскости и в пространстве с использованием функций библиотек OpenGL.

3.1. Постановка задачи

Дано координатное описание объекта. Требуется выполнить его геометрическое преобразование на плоскости или в пространстве.

3.2. Теоретические основы

Пусть M – произвольная точка плоскости с координатами *X* и *Y*, вычисленными относительно заданной прямоугольной координатной системы. Однородными координатами этой точки называется любая тройка одновременно неравных нулю чисел *x, y, w*, связанных с заданными числами *X* и *Y* следующими соотношениями:

$$\frac{x}{w} = X\,, \quad \frac{y}{w} = Y\,. \tag{3.1}$$

При решении задач компьютерной графики произвольной точке плоскости M*(x, y)*, заданной декартовыми координатами *(X,Y)*, обычно ставится в соответствие точка M' *(x, y,1)* в пространстве большей размерности (рис. 3.1).

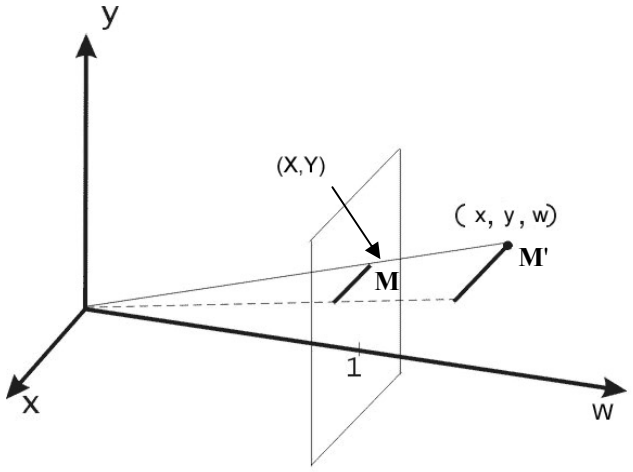


Рис. 3.1. Однородные координаты точки M *(X,Y)*

В случае, если точка M определена в пространстве – *(X,Y,Z)*, то ей будет соответствовать точка с координатами *(x,y,z,1)*. Тройки/четвёрки чисел – *(x,y,w)* и *(x,y,z,w)* называются однородными координатами. В общем случае значение *w*, которое называют масштабирующим множителем, может быть и отличным от 1, однако чаще всего используют $w = 1$, т. к. в этом случае в соответствии с выражением (3.2) $X = x$, $y = Y$ и, по аналогии, $Z = z$, т.е. преобразования из однородных координат в декартовы производятся без вычислений.

Основным применением однородных координат в компьютерной графике является оптимизация вычислений геометрических преобразований. При помощи однородных координат становится возможным задание в виде единой матрицы любого аффинного преобразования на плоскости и в пространстве.

Матрицы, описывающие базовые преобразования на плоскости:

Матрица вращения:

$$[\mathrm{R}] = \begin{bmatrix} \cos\varphi & \sin\varphi & 0 \\ -\sin\varphi & \cos\varphi & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad (3.2)$$

Матрица масштабирования:

$$[\mathrm{D}] = \begin{bmatrix} \alpha & 0 & 0 \\ 0 & \delta & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad (3.3)$$

Матрица отражения:

$$[\mathrm{M}] = \begin{bmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad (3.4)$$

Матрица переноса:

$$[\mathrm{T}] = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ \lambda & \mu & 1 \end{bmatrix} \quad (3.5)$$

В случае необходимости произвести комбинацию преобразований следует лишь перемножить матрицы в порядке нужной последовательности преобразований:

$$[\mathrm{T}_{-A}][\mathrm{R}_{\varphi}][\mathrm{T}_{A}]. \quad (3.6)$$

Так, например, выражение (3.6) задаёт поворот относительно произвольной точки (перенос – поворот – обратный перенос), и в результате искомое преобразование (в матричной записи) будет выглядеть следующим образом:

$$(x',y',1) = (x,y,1)\begin{bmatrix} \cos\varphi & \sin\varphi & 0 \\ -\sin\varphi & \cos\varphi & 0 \\ -a*\cos\varphi + b*\sin\varphi + a & -a*\sin\varphi - b*\cos\varphi + b & 1 \end{bmatrix} \qquad (3.7)$$

Матрицы преобразований могут задаваться в форматах векторов-строк и векторов-столбцов. В (3.2) – (3.7) приведены матрицы в формате для векторов-строк, взаимосвязь между форматами матриц иллюстрирует рис. 3.2.

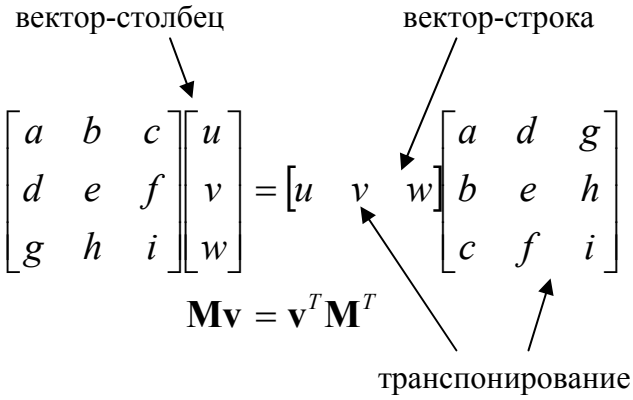


Рис. 3.2. Форматы вектора-столбца и вектора-строки

Ниже приводится пример вычисления итоговой матрицы последовательности преобразований, аналогичной матрице, используемой в (3.7), но применительно к пространственным преобразованиям, а именно: поворот в плоскости XY на θ градусов против часовой стрелки относительно точки P:

$$\mathbf{M} = \mathbf{T}(P)\mathbf{R}(\theta)\mathbf{T}(-P) = \begin{bmatrix} 1 & 0 & 0 & P_x \\ 0 & 1 & 0 & P_y \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}\begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}\begin{bmatrix} 1 & 0 & 0 & -P_x \\ 0 & 1 & 0 & -P_y \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} =$$

$$= \begin{bmatrix} \cos\theta & -\sin\theta & 0 & P_x - P_x\cos\theta + P_y\sin\theta \\ \sin\theta & \cos\theta & 0 & P_y - P_x\sin\theta - P_y\cos\theta \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}. \qquad (3.8)$$

3.3. Работа с матрицами в OpenGL

Преобразования координат в OpenGL можно производить несколькими способами – вызовами специальных функций и/или с помощью соответствую-

щих матриц преобразования. В первом случае OpenGL избавляет программиста от необходимости непосредственного вычисления и оперирования матрицами, однако они всё равно используются, хоть и не явно. При необходимости достижения особой гибкости вычислений матрицами можно оперировать и явно.

Как уже отмечалось выше, возможны два вида преобразований – изменения координат точки наблюдения и координат точек модели. В OpenGL существует логическое разделение этих двух типов преобразований. С точки зрения наблюдателя, различий между ними нет – суммарный эффект от движения объекта вперёд или движения системы отсчёта назад будет одинаковым. Иными словами, «преобразование наблюдения» – это преобразование координат модели виртуального объекта – «наблюдателя».

Так как понятие «наблюдатель» в определённом смысле неявно, то и так называемая «матрица наблюдения» (также в литературе встречается название «матрица модели»), задающая преобразования его координат, обычно используется в неявном виде. Изначально положение наблюдателя относительно системы координат соответствует единичной матрице наблюдения, что означает, что оси *OX* и *OY* расположены параллельно сторонам устройства отображения, а ось *OZ* проходит перпендикулярно экрану (её направление зависит от выбранной системы координат – правой или левой). Применение преобразований приводит к изменению текущих значений матрицы наблюдения и, соответственно, изменению ориентации системы координат сцены относительно экрана. Очевидно, что в случае длинных цепочек преобразований восстанавливать исходное её состояние обратной последовательностью действий не всегда удобно и тем более неэффективно с точки зрения вычислительных затрат.

Для хранения промежуточных состояний матрицы наблюдения используется специальный стек матриц. Запись текущей матрицы в стек производится командой *glPushMatrix()*, а извлечение – *glPopMatrix()*. Так как в OpenGL предусмотрено четыре стека матриц – для матриц наблюдения, проекции, текстуры и цвета, то с целью устранения дублирования команд работы с матрицами пре-

дусмотрен механизм переключения между указанными четырьмя вариантами с помощью функции *glMatrixMode()*, в качестве аргумента которой задаётся одна из четырёх констант: GL_MODELVIEW, GL_PROJECTION, GL_TEXTURE, GL_COLOR соответственно.

При переключении на необходимую матрицу, все команды работы с матрицами будут затрагивать только состояние активной матрицы (до следующей аналогичной команды переключения матриц). Одна из таких команд – *glLoadIdentity()* – приводит к установке элементов текущей матрицы в состояние, эквивалентное единичной матрице, т.е. в её исходное состояние. Команда *glLoadMatrix*[*]*()*, в свою очередь, приводит к установке значений элементов текущей матрицы равными значениям элементов матрицы, передаваемой в качестве аргумента этой команды. Часто бывает целесообразным не устанавливать значения элементов текущей матрицы «напрямую», а перемножать текущую матрицу на задаваемую (в умножении она будет стоять справа), используя команду *glMultMatrix*[*]*(),* – таким способом также можно задать явное изменение элементов нужной матрицы (наблюдения, проекции, текстуры или цвета).

К командам неявного изменения текущей матрицы можно отнести базовые геометрические преобразования OpenGL: *glRotate*[*]*()* – поворот текущей матрицы согласно задаваемым параметрам, *glScale*[*]*()* – масштабирование текущей матрицы согласно заданным параметрам, *glTranslate*[*]*()* – перенос координат на заданные *dx, dy, dz*.

Пример кода программы, иллюстрирующей основы работы с матрицей наблюдения:

```
void MyDrawFunction(void)
{
  glClear(GL_COLOR_BUFFER_BIT); //Очистка – т.е. заливка
              // пиксельного буфера цветом очистки экрана
  glMatrixMode(GL_MODELVIEW); // Переключение стека матриц
                              // на матрицу наблюдения
  glLoadIdentity();  // Загрузка единичной матрицы
  glTranslatef(0.0f, 0.0f, -100.0f);     //Перенос системы
  // координат на 100 единиц по оси Z для обзора всей сцены
```

[*] В зависимости от типа аргумента у функции различается окончание – либо ***d,*** либо ***f***.

```
  … … … …

  glColor3ub(255, 0, 0);       //Установка красного цвета
  glutSolidSphere(10.0f, 15, 15); // Отрисовка красного шарика
  glPushMatrix();    // Сохранение текущего состояния матрицы
                     // наблюдения в стеке
  glRotatef(fElect1, 0.0f, 1.0f, 0.0f); // Поворот системы
  // координат относительно оси OY на заданный в градусах угол –
  // fElect1
  glTranslatef(90.0f, 0.0f, 0.0f); // Перенос положения системы
                          // координат на 90 единиц вдоль оси OX
  glColor3ub(255,255,0);       // Установка жёлтого цвета
  glutSolidSphere(6.0f, 15, 15); // Отрисовка жёлтого шарика
  glPopMatrix(); // Восстановление из стека начальных значений
    // активной матрицы – т.е. матрицы наблюдения
  // Аналогичные операции для отрисовки орбиты другого шарика
  glPushMatrix();
  glRotatef(45.0f, 0.0f, 0.0f, 1.0f);
  glRotatef(fElect1, 0.0f, 1.0f, 0.0f);
  glTranslatef(-70.0f, 0.0f, 0.0f);
  glutSolidSphere(6.0f, 15, 15);
  glPopMatrix();
  … … … …
}
```

Вторая часть приведённого выше примера иллюстрирует хороший стиль программирования в OpenGL – обрамление команд изменения текущего состояния матрицы функциями *glPushMatrix()* и *glPopMatrix(),* даже если перед этим матрица была восстановлена из стека и не изменялась. Подобный стиль помогает избегать трудно уловимых ошибок визуализации сцены.

При изменении отдельных элементов текущей матрицы следует учитывать тот факт, что внутреннее представление матриц в OpenGL реализовано в виде одномерного массива из 16 элементов заданного программистом типа – либо *GLfloat,* либо *GLdouble*. Порядок обхода матрицы при записи в одномерный массив демонстрирует рис. 3.3.

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

Рис. 3.3. Соответствие элементов одномерного массива *a[0]-a[15]* внутреннему представлению матриц в OpenGL

Подобная программная реализация матриц является более эффективной с вычислительной точки зрения по сравнению с двухмерными массивами типа

```
GLfloat  matrix [4][4];
```

В качестве примера, противоположного преобразованию матрицы наблюдения, можно привести фрагменты программы поворота тора [см. 3.6. п.2], производящей поворот путём пересчёта координат вершин составляющих тор полигонов в «статической» с точки зрения виртуального наблюдателя системе координат. Функция вычисления текущих координат вершин тора в соответствии с получаемой на входе матрицей преобразования будет иметь следующий вид:

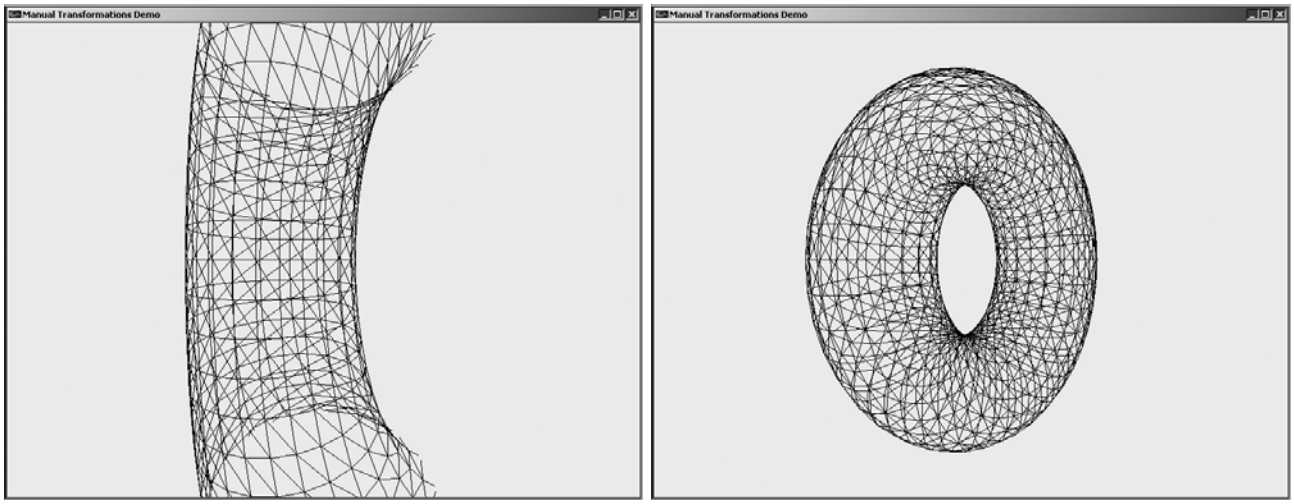
```
void DrawTorus(GLTMatrix mTransform)
{
  GLfloat majorRadius = 0.35f;
  GLfloat minorRadius = 0.15f;
  GLint   numMajor = 40;
  GLint   numMinor = 20;
  GLTVector3 objectVertex;      // Вершина до преобразования
  GLTVector3 transformedVertex; // Вершина после преобразования
  double majorStep = 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);

        objectVertex[0] = x0*r; // Пересчёт координат первой точки
        objectVertex[1] = y0*r;
        objectVertex[2] = z;
gltTransformPoint(objectVertex, mTransform, transformedVertex);
        glVertex3fv(transformedVertex);
```

```
        objectVertex[0] = x1*r; // Пересчёт координат второй точки
        objectVertex[1] = y1*r;
        objectVertex[2] = z;
gltTransformPoint(objectVertex, mTransform, transformedVertex);
        glVertex3fv(transformedVertex);
       }
    glEnd();
  }
}
```

Программный код отображения тора, можно записать в следующем виде:

```
void MyDrawFunction(void)
{
  GLTMatrix   transformationMatrix; // Матрица преобразования
  static GLfloat yRot = 0.0f;       // Угол поворота для анимации
  yRot += 0.5f;

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

  // Построение матрицы поворота
  gltRotationMatrix(gltDegToRad(yRot), 0.0f, 1.0f, 0.0f, transfor-
mationMatrix);
  transformationMatrix[12] = 0.0f;  // Перенос тора вдоль оси
  transformationMatrix[13] = 0.0f;  // OZ – вглубь экрана
  transformationMatrix[14] = -2.5f; // от наблюдателя для охвата
                          //взглядом всей сцены вращающегося тора

  DrawTorus(transformationMatrix); // Изменение координат вершин
                //тора в соответствии с матрицей преобразования

  glutSwapBuffers(); // Переключение буферов кадра
}
```

В приведённой выше функции *void MyDrawFunction()*, вызываемой каждый раз при перерисовке сцены, производится вычисление матрицы поворота на изменяемый с шагом в 0,5 градусов угол относительно оси OY командой *gltRotationMatrix ()*, с последующей модификацией возвращаемой матрицы прямой записью в её ячейки смещений *dx =0.0, dy = 0.0 u dz = –2,5*. Непосредственное изменение ячеек матрицы в данном случае будет эквивалентно её умножению на матрицу переноса вдоль оси OZ на –2.5 единицы. На рис. 3.4 приведены результаты работы программы при различных значениях *dz*. Так как при выводе тора на экран использовалось перспективное преобразование, то смещение объекта вдоль оси OZ эквивалентно его масштабированию.



а б

Рис. 3.4. Отображение тора при значениях:

а – transformationMatrix[14] = –1.0f, б – transformationMatrix[14] = –2.0f

Несмотря на корректность программного кода функций *DrawTorus()* и *MyDrawFunction()*, преобразование координат вершин тора не является оптимальным вариантом визуализации данного объекта по двум причинам:

– пересчёт координат вершин в данном случае будет производиться центральным процессором (ЦП), в ином случае при изменении матрицы наблюдения командами OpenGL были бы задействованы ресурсы специализированных процессоров видеокарты, и вычисления проводились бы быстрее, а ЦП мог бы параллельно выполнять другие задачи;

– при визуализации сцены координаты вершин объектов умножаются на матрицу наблюдения (в данном случае – единичную), соответственно в функции *MyDrawFunction()* неявно присутствует бесполезная операция умножения каждой вершины тора на единичную матрицу.

Программный код оптимального (с вычислительной точки зрения) варианта функции вращения тора будет выглядеть следующим образом:

```
void MyDrawFunction(void)
{
  static GLfloat yRot = 0.0f; // Угол поворота для анимации
  yRot += 0.5f;
  // Очистка окна текущим цветом
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
```

```
  glPushMatrix();
    glTranslatef(0.0f,0.0f,-2.5f);
    glRotatef(yRot, 0.0f, 1.0f, 0.0f);
    gltDrawTorus(0.35f, 0.15f, 40, 20); // Построение тора
  glPopMatrix();
  glutSwapBuffers();// Переключение буферов кадра
}
```

В данном варианте функции отображения все расчёты производятся над матрицей наблюдения (процессорами видеокарты), при этом достигается дополнительный положительный момент – функция вычисления координат тора не зависит от матрицы преобразования, следовательно, можно разработать и использовать для любых преобразований одну универсальную функцию вычисления координат вершин тора нужного размера, толщины и т.п. Такая функция – *gltDrawTorus()* – имеется и включена в библиотеку glTools.

3.4. Проективные преобразования

В общем случае под проективными преобразованиями, или проецированием, подразумевают преобразование точек из координатной системы размерности n в точки в координатной системе размерности меньше n. В задачах машинной графики наиболее часто используются преобразования из трёхмерного пространства – 3D (сокращение от «3 dimensional» – «трёхмерное», пер. с англ.) в двухмерное – 2D. В 3D-пространстве моделируется сцена, которая должна быть отображена на плоской поверхности устройства вывода (монитора, принтера, плоттера и т.п.). Вполне вероятно, что в недалёком будущем при условии широкого распространения трёхмерных устройств вывода графической информации насущность в подобных преобразованиях заметно снизится, однако в настоящее время проективные преобразования играют одну из ведущих ролей при достижении реалистичности изображения. Следует отметить, что в машинной графике также встречается и преобразование из системы размерностью $n = 3$ в систему с размерностью $n = 1$. Оно соответствует операции перевода цветного изображения в полутоновое.

Проекция строится с помощью воображаемых линий, называемых проекторами либо лучами проекции. Проекторы исходят из центра проекции, прохо-

дят через каждую точку объекта и пересекают плоскость проекции (в случае построения 2D-проекции). В машинной графике оперируют только плоскими проекциями – т.е. поверхность проекции есть плоскость. Большинство фотокамер и видеокамер в качестве плоскости проекции используют плёнку либо матрицу. Однако в зрительной системе человека сетчатка глаза не плоская – проецирование производится на внутреннюю поверхность сферы. Также в машинной графике используют только геометрические проекции – т. е. те, в которых проекторы являются прямыми лучами.

В зависимости от количества центров проекции различают два основных класса проекций – центральные (или перспективные) и параллельные. Рис. 3.5 иллюстрирует разницу между двумя классами проекций.

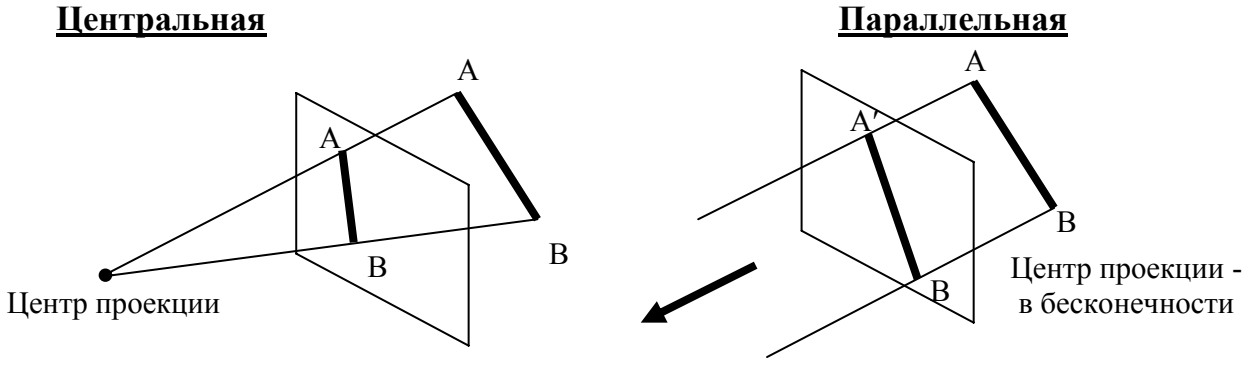


Рис. 3.5. Формирование центральных и параллельных проекций

В свою очередь, параллельные проекции в зависимости от взаимного расположения системы координат объекта, лучей проекции и плоскости проекции, делятся на несколько видов. Их классификация приведена на рис. 3.6.

Из всего множества возможных способов проецирования в OpenGL на системном уровне поддерживаются два типа – ортогональные и центральные одноточечные, которые также называют перспективными. Разновидности аксонометрических проекций можно получить самостоятельно, производя соответствующие преобразования над взаимным расположением объектов сцены и наблюдателя с выполнением на заключительном этапе ортогонального преобразования.

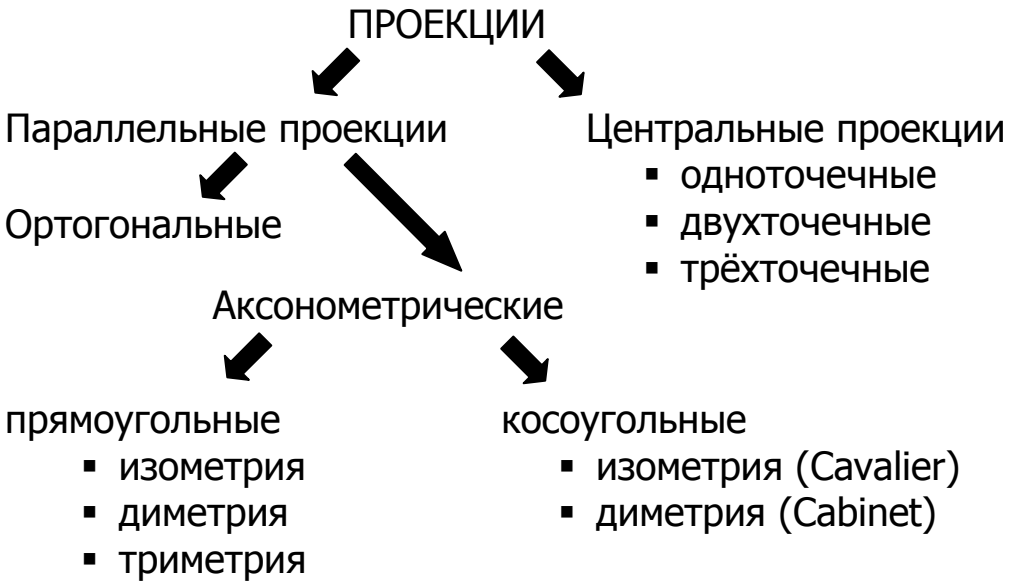


Рис. 3.6. Виды геометрических проекций

Задание способа проецирования итогового изображения на экран производится с помощью функций *gluOrtho2D()* и *gluPerspective()*. Для получения разновидностей проекций необходимо подключить матрицу проецирования командой *glMatrixMode()* с параметром GL_PROJECTION и провести над ней необходимые преобразования. Разницу между ортогональными и перспективными проекциями демонстрирует рис. 3. 7.

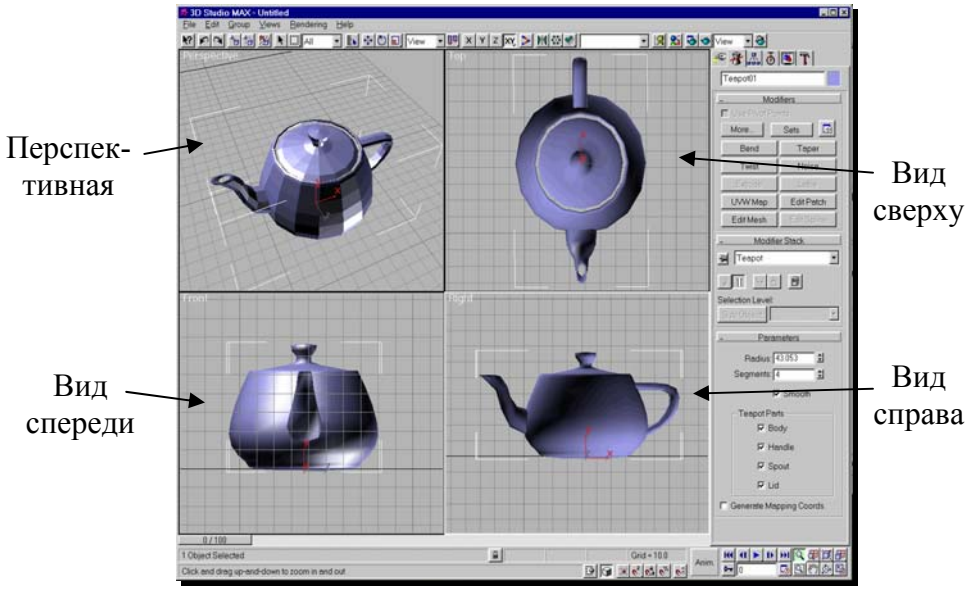


Рис. 3.7. Пример объекта в перспективной (центральной одноточечной) и в ортогональных (виды спереди, сверху, справа) проекциях

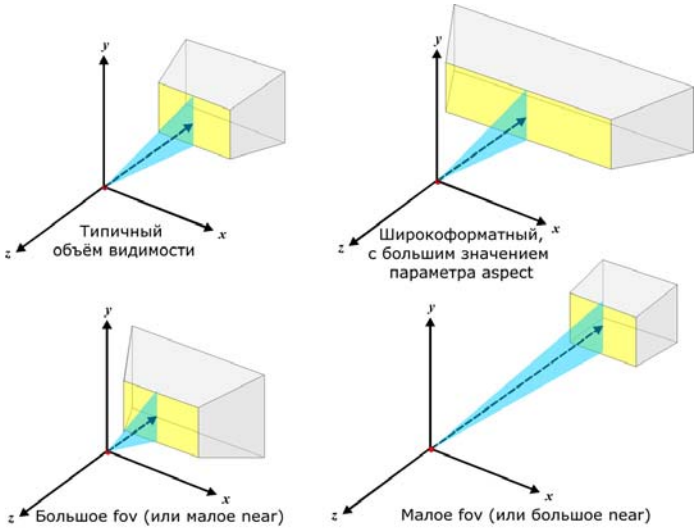


Рис. 3.8. Примеры вариантов объёма видимости в зависимости от задаваемых параметров функции *glFrustum()*

При проведении перспективных преобразований в OpenGL важную роль играет понятие «объём видимости» (рис. 3.8), которое сильно влияет на итоговый вид проецируемой сцены (рис. 3.9).

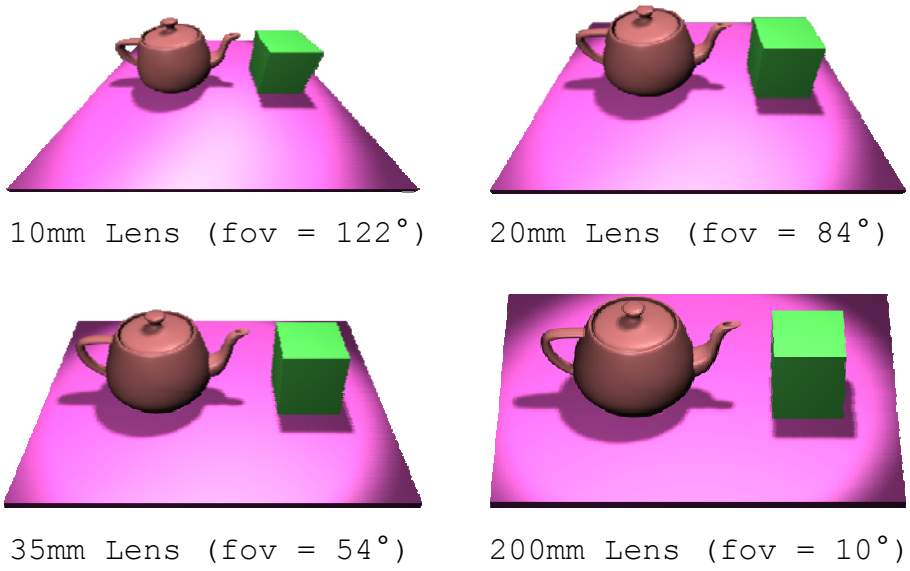
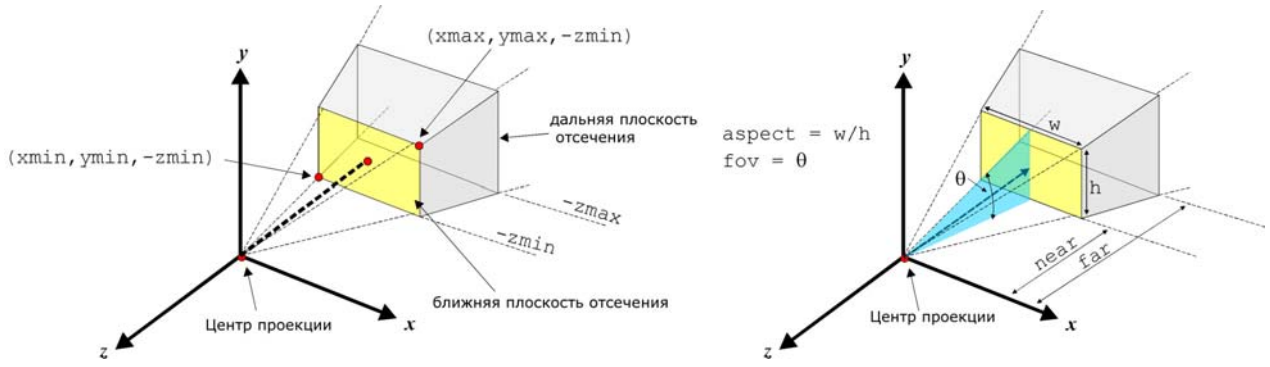


Рис. 3.9. Примеры вариантов проецирования одной сцены

Для задания объёма видимости необходимо воспользоваться функцией *glFrustum()*; для определения параметров перспективы – функцией *gluPerspective()*, смысл параметров которой иллюстрирует рис. 3.10.



а б

Рис. 3.10. Смысл параметров функций:

а – *glFrustum()*, б – *gluPerspective()*

3.5. Примеры типовых заданий к лабораторной работе

1. Используя функции библиотек OpenGL, разработать программу отображения произвольной осмысленной трёхмерной сцены, содержащей не менее 10 – 15 объектов, которые должны изменять своё положение относительно точки наблюдения с течением времени.

2. Разработать программу отображения движущейся на экране объёмной фигуры робота, человека или животного.

3.6. Контрольные вопросы

1. В чем состоит идея однородных координат?

2. Каковы достоинства матричного представления преобразований? Как можно ускорить их выполнение?

3. В каких случаях и при каких трансформациях произведение матриц аддитивно? Коммутативно?

4. Как влияют параметры проецирования на итоговый вид проекции?

3.7. Литература по теме работы

1. Херн, Д. Микрокомпьютерная графика и стандарт OpenGL/ Д. Херн, М.Бейкер. – М.: “Вильямс”, 2005, §§§5.1 – 5.11, 6.1 – 6.4, 7.4 – 7.10.

2. Райт-мл., Р.С. OpenGL суперкнига/ Р.С. Райт-мл., Б. Липчак. – М.: “Вильямс”, 2005, §§4.

ЛАБОРАТОРНАЯ РАБОТА №4.
МОДЕЛИРОВАНИЕ ОСВЕЩЕНИЯ В OPENGL

Цель работы: освоить теоретические принципы и инструментальные средства расчета освещения трехмерной сцены в OpenGL.

4.1. Постановка задачи

Создать трехмерную сцену с несколькими источниками света разных типов, с заданной моделью освещенности OpenGL и оптическими свойствами материалов объектов сцены.

4.2. Теоретические основы

После построения трехмерной сцены следующий шаг в создании реалистичного изображения – расчет освещения. При этом яркость и цвет каждой видимой точки объектов вычисляются исходя из так называемой *модели освещения*.

Освещенность отдельной точки определяется двумя основными параметрами: свойствами поверхности и характером освещения. Тип поверхности влияет на формирование отраженного света, а свойства освещения задаются законами распространения света.

Основное понятие, используемое в построении освещенных сцен, – *источник света*. Под источником света понимается любой объект, излучающий энергию и влияющий на освещенность других объектов. Чаще всего при построении сцены используется *точечные* источники. Такие объекты характеризуются радиальным распространением лучей света и пренебрежимо малыми собственными размерами. Если источник света расположен относительно далеко от сцены, так что вектора освещенности от этого источника можно считать параллельными, то такой источник называют *бесконечно удаленным*. Точечный источник, излучающий свет не радиально, а в определенном направлении, называют *направленным*. Нередки ситуации, когда размерами излучающего объ-

екта пренебречь нельзя. В таких случаях говорят о *светоизлучающей поверхности*. Как правило, она моделируется сеткой точечных источников излучения.

Для создания реалистичных изображений необходимо учитывать законы распространения и отражения света. Основной принцип, используемый при расчете распространения света – закон затухания излучения. В случае с точечным источником можно говорить об обратно-квадратичной зависимости, т.е. коэффициенте затухания, равном $1/d_l^2$. Однако при такой зависимости часто возникает неадекватный разброс интенсивностей, поэтому обычно применяется аппроксимированная формула с добавлением линейного члена:

$$f_{рад.затух.}(d_l) = \frac{1}{a_0 + a_1 d_l + a_2 d_l^2} \quad . \tag{4.1}$$

При расчете направленного источника света дополнительно рассчитывается еще и угловое затухание интенсивности, предполагающее снижение яркости света под углами падения, отличными от 90^o. Широко используется следующее выражение:

$$f_{угл.затух.}(\phi) = \cos^a \phi, 0^o < \phi < \theta \quad . \tag{4.2}$$

В формулах затухания параметры a_i подгоняются таким образом, чтобы обеспечить наиболее точное распространение излучения.

В большинстве сцен также принято учитывать фоновое рассеянное освещение, т.к. им освещена каждая поверхность на сцене.

Как было сказано выше, при расчете освещенности объекта также учитываются оптические свойства поверхности объекта. В упрощенной модели можно говорить о двух основных явлениях, вытекающих из этих свойств: диффузном и зеркальном отражении света.

При диффузном отражении падающий свет отражается равномерно во всех направлениях. Такой тип отражения мы можем наблюдать на матовых поверхностях. Для расчета освещенности в точке с учетом диффузного отражения не-

обходимо ввести два единичных вектора: вектор нормали **N** и вектор направления на точечный источник света **L**.

Общая формула при этом выглядит следующим образом:

$$I_{\text{дифф.отр.}} = \begin{cases} k_a I_a + k_d I_l (\mathbf{N} * \mathbf{L}), \mathbf{N} * \mathbf{L} > 0, \\ k_a I_a, \mathbf{N} * \mathbf{L} \le 0. \end{cases} \quad (4.3)$$

Здесь k_d – коэффициент диффузного отражения, изменяющийся от 0 до 1. Помимо этого, в формуле присутствует расчет рассеянного освещения, о котором говорилось выше. Для учета фонового рассеянного света используются два параметра: k_a – коэффициент рассеянного отражения и I_a – интенсивность рассеянного излучения.

Зеркальное отражение подразумевает наличие яркого блика на поверхности объекта, если вектор отраженного света направлен в точку наблюдения.

Для расчета зеркального отражения вводится дополнительный единичный вектор **V**, направленный в точку наблюдения. Формула расчета зеркального отражения выглядит так:

$$I_{\text{зерк.отр.}} = \begin{cases} k_s I_l (\mathbf{V} * \mathbf{R}), \textit{ если } \mathbf{V} * \mathbf{R} > 0 \textit{ и } \mathbf{N} * \mathbf{L} > 0, \\ 0, \textit{ если } \mathbf{V} * \mathbf{R} < 0 \textit{ и ли } \mathbf{N} * \mathbf{L} \le 0. \end{cases} \quad (4.4)$$

При этом вектор зеркального отражения **R** рассчитывается следующим образом:

$$\mathbf{R} = (2\mathbf{N} * \mathbf{L})\mathbf{N} - \mathbf{L}. \quad (4.5)$$

Приведенные выше формулы были выведены из условия монохроматического освещения. Для расчета цветной освещенности необходимо учитывать разложение белого цвета на RGB-составляющие (R – красный, G – зеленый, B – синий). При этом все коэффициенты и параметры интенсивности раскладываются на три составляющие и рассчитываются независимо.

4.3. Функции освещения и визуализации OpenGL

При описании трехмерной сцены в OpenGL можно добавить один или несколько источников света, а также указать его оптические свойства. Для этого используется следующая функция:

glLight[*] *(имяИсточника, свойстваИсточника, значениеСвойства).*

Каждый источник света снабжается идентификатором, и параметру *имяИсточника* присваивается значение одной из символьных констант OpenGL (GL_LIGHT0, GL_LIGHT1 и т.д.). Как правило, количество источников света ограничено восемью, хотя это зависит от реализации стандарта OpenGL.

После описания источника света его необходимо активировать функцией *glEnable(имяИсточника).* Кроме этого, необходимо активировать расчет освещения в OpenGL функцией *glEnable(GL_LIGHTING).*

Свойства источника света задаются последовательным вызовом функции *glLight*[*]*()* с указанием конкретных свойств. При этом могут быть использованы следующие обозначения:

GL_POSITION – положение источника света;

GL_AMBIENT – коэффициент рассеянного отражения;

GL_DIFFUSE – коэффициент диффузного отражения;

GL_SPECULAR – коэффициент зеркального отражения;

GL_CONSTANT_ATTENUATION, GL_LINEAR_ATTENUATION и GL_QUADRATIC_ATTENUATION – коэффициенты радиального затухания интенсивности, соответствующие коэффициентам a_0, a_1 и a_2 в формуле 4.1.

Пример создания точечного источника света:

```
GLfloat light1PosType [ ] = { 2.0, 0.0, 3.0, 1.0 };
GLfloat blackColor [ ] = { 0.0, 0.0, 0.0, 0.0 };
GLfloat whiteColor [ ] = { 1.0, 1.0, 1.0, 1.0 };
glLightfv(GL_LIGHT1, GL_POSITION, light1PosType);
glLightfv(GL_LIGHT1, GL_AMBIENT, blackColor);
glLightfv(GL_LIGHT1, GL_DIFFUSE, whiteColor);
glLightfv(GL_LIGHT1, GL_SPECULAR, whiteColor);
glEnable(GL_LIGHT1);
```

[*] В зависимости от типа аргумента у функции различается суффикс – ***i, f*** или ***v.***

Для создания направленного источника света в OpenGL есть три константы свойств:

GL_SPOT_DIRECTION – вектор направления излучения в глобальных координатах;

GL_SPOT_CUTOFF – угол конуса распространения света;

GL_SPOT_EXPONENT – параметр затухания.

Кроме этого, OpenGL позволяет задавать глобальные параметры освещения, влияющие на расчет всей сцены. Для этого используется функция *glLightModel*(имяПараметра, значениеПараметра)*. В этой функции применяются такие же суффиксы, как и в *glLight*()*.

Коэффициенты отражения света от объектов являются свойствами материала и задаются функцией *glMaterial*(типПоверхности, свойствоПоверхности, значениеСвойства)*. Параметр *типПоверхности* принимает одно из трех значений: GL_FRONT, GL_BACK и GL_FRONT_AND_BACK. Свойства поверхности могут быть следующими:

GL_EMISSION – цвет излучения с поверхности;

GL_AMBIENT – коэффициент рассеянного отражения;

GL_DIFFUSE – коэффициент диффузного отражения;

GL_AMBIENT_AND_DIFFUSE – объединенный коэффициент рассеянного и диффузного отражения, которые зачастую равны по значению, так что этот параметр используется для сокращенной записи;

GL_SPECULAR – коэффициент зеркального отражения;

GL_SHININESS – показатель зеркального отражения.

Ход действий при построении реалистичной сцены, как правило, следующий: описываются свойства материала, затем формируется геометрия объекта. При этом объекту будут присвоены оптические свойства материала, который был описан до него. На заключительном этапе задаются условия освещения.

4.4. Примеры типовых заданий к лабораторной работе

1. Используя функции библиотек OpenGL, доработать программу, реализующую задание по лабораторной работе №2 для отображения трёхмерной сцены с учётом нескольких источников освещения (точечных, бесконечно удалённых, светящихся поверхностей).

2. Разработать программу отображения движущихся прожекторов на фоне объёмной панорамы города (со светящимися окнами, рекламой и т.п.).

4.5. Контрольные вопросы

1. Из каких составляющих складывается уровень освещенности точки?

2. Покажите, что при расчете радиального затухания необходимо добавление линейного члена для построения адекватной модели.

3. Сохранится ли уровень освещенности при диффузном отражении при смене точки наблюдения? А при зеркальном отражении?

4. От чего зависит размытость краев блика на объекте при зеркальном отражении?

5. Выведите интегральную формулу для расчета освещенности точки, учитывающую все типы отражения и затухания.

6. За что отвечает параметр GL_DIFFUSE при задании свойств источника света и материала соответственно?

7. Какими способами можно устранить зеркальное отражение от описываемого материала?

8. Какими способами можно визуализировать полупрозрачные объекты?

4.6. Литература по теме работы

1. Херн, Д. Микрокомпьютерная графика и стандарт OpenGL/ Д. Херн, М.Бейкер. – М.: “Вильямс”, 2005, §§§10.1 – 11.16.

2. Райт-мл., Р.С. OpenGL суперкнига/ Р.С. Райт-мл., Б. Липчак. – М.: “Вильямс”, 2005, §§5,6.

Св. план 2009, поз. 66

Учебное издание

Самаль Дмитрий Иванович
Супонев Виктор Алексеевич
Прытков Валерий Александрович

“Машинная графика”
Лабораторный практикум
для студентов специальности 1- 40 02 01

Редактор Г.С. Корбут
Корректор

Подписано в печать .10.2008. Формат 60x84 1/16.
Бумага офсетная. Печать ризографическая. Усл.печ.л.
Уч.-изд.л. 3. Тираж экз. Заказ

Издатель и полиграфическое исполнение:
Учреждение образования “Белорусский государственный университет информатики и радиоэлектроники”
Лицензия ЛП № 156 от 05.02.2001
Лицензия ЛВ № 509 от 03.08.2001
220013, Минск, П. Бровки, 6.