Учреждение образования
«БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ
ТЕХНОЛОГИЧЕСКИЙ УНИВЕРСИТЕТ»

А. А. Дятко, Л. С. Мороз

ОСНОВЫ КОМПЬЮТЕРНОЙ ГЕОМЕТРИИ И ГРАФИКИ

*Рекомендовано
учебно-методическим объединением по образованию в области
информатики и радиоэлектроники в качестве учебно-методического пособия
для студентов учреждений высшего образования по направлению
специальности 1-40 01 02-03 «Информационные системы и технологии
(издательско-полиграфический комплекс)»*

Минск 2013

УДК 004.92:514(075.8)
ББК 32.97я73
Д99

Рецензенты:
кафедра информационных технологий
автоматизированных систем УО «Белорусский государственный
университет информатики и радиоэлектроники»
(кандидат технических наук, доцент кафедры *О. В. Герман*);
кандидат физико-математических наук, доцент,
заведующий кафедрой экономико-математических методов
управления Академии управления при Президенте
Республики Беларусь *Б. В. Новыш*

Дятко, А. А.

Д99 Основы компьютерной геометрии и графики : учеб.-метод. пособие для студентов направления специальности 1-40 01 02-03 «Информационные системы и технологии (издательско-полиграфический комплекс» / А. А. Дятко, Л. С. Мороз. – Минск : БГТУ, 2013. – 221 с.

ISBN 978-985-530-257-6.

Учебно-методическое пособие посвящено изучению основ компьютерной геометрии и графики. Рассматриваются основные растровые алгоритмы построения графических примитивов. Важное место в учебно-методическом пособии отводится изучению геометрических преобразований, в частности аффинным преобразованиям систем координат и объектов. Рассмотрены базовые возможности графической библиотеки OpenGL и основные приемы работы с этой библиотекой.

Пособие будет полезным для всех студентов высших учебных заведений, изучающих основы компьютерной геометрии и графики.

УДК 004.92:514(075.8)
ББК 32.97я73

ISBN 978-985-530-257-6

ВВЕДЕНИЕ

В основании бесконечно развивающегося здания компьютерной графики краеугольными камнями лежат фундаментальные дисциплины – аналитическая геометрия и оптика, скрепленные раствором – искусством программирования [1]. Возникнув из потребностей рынка, развития информатики и вычислительной техники, компьютерная графика изучает методы построения изображений различных геометрических объектов и сцен. Задачи, связанные с применением компьютерной графики, возникают в самых различных сферах информационных технологий. Сюда можно отнести системы автоматизированного проектирования, в которых осуществляется интерактивное взаимодействие конструктора и синтезированного с помощью компьютера изображения создаваемого изделия или сооружения. Широко используется компьютерная графика в автоматизированных системах научных исследований, в которых осуществляется визуализация результатов экспериментов в виде трехмерных статических или динамических изображений, интерпретирующих огромные массивы первичных данных. Важную роль играют методы компьютерной графики для распознавания и обработки изображений в системах искусственного зрения, авиационной и космической картографии и других областях человеческой деятельности. Знание основ компьютерной графики в наше время необходимо любому ученому или инженеру. Для формирования навыков решения вышеупомянутых задач и служит учебно-методическое пособие «Основы компьютерной геометрии и графики».

Глава 1

ЦЕЛИ И ЗАДАЧИ КОМПЬЮТЕРНОЙ ГРАФИКИ

Понятие «компьютерная графика» объединяет довольно широкий круг операций по обработке графической информации с помощью компьютера. Причем наблюдается явная тенденция «компьютеризации» изображений, циркулирующих в обществе. Стали обыденностью термины «цифровое фото» и «видео». В виртуальных буднях грядущего компьютерной графике отводится огромная роль. Это связано с тем, что зрительная система, по мнению ученых, исследующих проблемы мозга, в иерархии мозговых структур человека занимает особое место. С восприятием и обработкой визуальной информации непосредственно связано примерно 20% мозга человека. Благодаря зрению мы получаем по разным оценкам от 70 до 90% сведений об окружающем мире. Следовательно, образный мир компьютерной графики является одним из глубинных проявлений человеческой природы.

В компьютерной графике можно выделить несколько основных направлений [2, 4].

1.1. Визуализация научных данных

Большинство современных математических программных пакетов (например, Maple, MatLab, MathCAD) имеют средства для отображения графиков, поверхностей и трехмерных тел, построенных на основе каких-либо расчетов. Кроме того, графическая информация может активно использоваться в самом процессе вычислений. Визуализация позволяет представить большой объем данных в удобной для анализа форме и широко используется при обработке результатов различных измерений и вычислений.

1.2. Геометрическое проектирование и моделирование

Это направление компьютерной графики связано с решением задач начертательной геометрии – построением чертежей, эскизов, объемных изображений с помощью программных систем, полу-

чивших название CAD-системы (от англ. *computer-aided design*), например AutoCAD.

Существует большое количество специализированных CAD-систем в машиностроении, архитектуре и т. д.

1.3. Распознавание образов

Способность распознавать абстрактные образы считают одним из важнейших факторов, определивших развитие мыслительных способностей человека, выделив его из животного мира. Задача распознавания и классификации графической информации является одной из ключевых и при создании искусственного интеллекта. Уже в наши дни компьютеры распознают образы повсеместно (системы идентификации футбольных хулиганов у входа на стадион; анализ аэро- и космических фотоснимков; системы сортировки, наведения и т. д.). Возможно, самый известный пример распознавания образов – сканирование и перевод «фотографии» текста в набор отдельных символов, формирующих слова. Такую операцию позволяет выполнить программное обеспечение многих современных сканеров. Кроме того, существуют специализированные программы распознавания текста, например FineReader.

1.4. Изобразительное искусство

К этому направлению можно отнести разнообразную графическую рекламу: от текстовых транспарантов и фирменных знаков до компьютерных видеофильмов; обработку фотографий, создание рисунков, мультипликацию и т. д. В качестве примера популярных программ из этой области компьютерной графики можно назвать Adobe Photoshop (обработка растровых изображений), CorelDRAW (создание векторной графики), 3DS Max (трехмерное моделирование).

1.5. Виртуальная реальность

Виртуальная реальность – созданный техническими средствами мир (объекты и субъекты), передаваемый человеку через его ощущения: зрение, слух, обоняние, осязание и др.

Реальность, даже виртуальная, подразумевает воздействия на всю совокупность органов чувств человека, в первую очередь на

его зрение. К компьютерной графике можно отнести задачи моделирования внешнего мира в различных приложениях: от компьютерных игр до тренажеров.

1.6. Цифровое видео

Все более широкое распространение получают анимированные изображения, записанные в цифровом формате. Это, прежде всего, фильмы, передаваемые через компьютерные сети, а также видеодиски, цифровое, кабельное и спутниковое телевидение.

Приведенная классификация сфер применения компьютерной графики является во многом условной.

Возможно, найдутся задачи, которые нельзя отнести ни к одному из обозначенных направлений.

Глава 2

ОСНОВНЫЕ ПОНЯТИЯ КОМПЬЮТЕРНОЙ ГРАФИКИ

2.1. Визуализация изображений

Наиболее известны два способа визуализации: **растровый** и **векторный**. Первый способ ассоциируется с такими графическими устройствами, как дисплей, телевизор, принтер. Второй используется в векторных дисплеях, плоттерах.

Наиболее удобно, когда способ описания графического изображения соответствует способу визуализации. Иначе нужна конвертация. Например, изображение может храниться в растровом виде, а его необходимо вывести (визуализировать) на векторном устройстве. Для этого нужна предварительная **векторизация** – преобразование из растрового в векторное описание. Или, наоборот, описание изображения может быть в векторном виде, а нужно визуализировать на растровом устройстве – необходима **растеризация.**

Растровая визуализация основывается на представлении изображения на экране или бумаге в виде совокупности (массива) отдельных точек (пикселов). Вместе пикселы образуют **растр.**

Для обозначения массива пикселов часто используется термин **bitmap** (битовая карта). В **bitmap** каждому пикселу отводится определенное число битов (одинаковое для всех пикселов изображения). Это число называется **битовой глубиной** пиксела или **цветовой глубиной** изображения, так как от количества битов, отводимых на один пиксел, зависит количество цветов изображения.

Векторная визуализация основывается на формировании изображения на экране или бумаге рисованием линий (векторов) – прямых или кривых. Совокупность типов линий (графических примитивов), которые используются как базовые для векторной визуализации, зависит от определенного устройства. Типичная последовательность действий при векторной визуализации для плоттера или векторного дисплея такова: переместить перо в начальную точку (для дисплея – отклонить пучок электронов); опустить перо (увеличить яркость луча); переместить перо в конечную точку; поднять перо (уменьшить яркость луча).

Качество векторной визуализации для векторных устройств обуславливается точностью вывода и номенклатурой базовых графических примитивов – линий, дуг, кругов, эллипсов и других.

Доминирующим сейчас является растровый способ визуализации. Это обусловлено большей распространенностью растровых дисплеев и принтеров. Недостаток растровых устройств – дискретность изображения. Недостатки векторных устройств – проблемы при сплошном заполнении фигур, меньшее количество цветов, меньшая скорость (в сравнении с растровыми устройствами).

2.2. Растровые изображения и их основные характеристики

Растр – это матрица (массив) ячеек (пикселов). Каждый пиксел может иметь свой цвет. Совокупность пикселов различного цвета образует изображение. В зависимости от расположения пикселов в пространстве различают квадратный, прямоугольный, гексагональный или иные типы растра. Для описания расположения пикселов используют разнообразные системы координат. Общим для всех таких систем является то, что координаты пикселов образуют дискретный ряд значений (необязательно целые числа). Часто используется система целых координат – номеров пикселов с (0,0) в левом верхнем углу. Такую систему мы будем использовать и в дальнейшем, ибо она удобна для рассмотрения алгоритмов графического вывода.

2.3. Геометрические характеристики растра

Разрешающая способность характеризует расстояние между соседними пикселами (рис. 2.1).

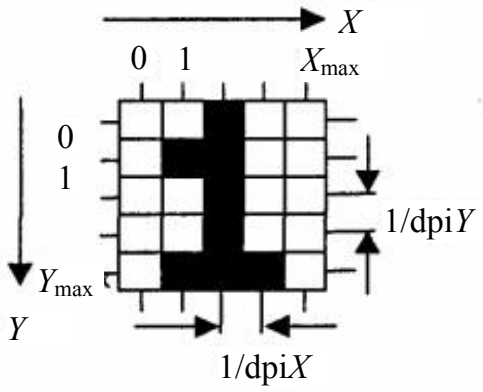


Рис. 2.1

Разрешающую способность измеряют количеством пикселов на единицу длины. Наиболее популярной единицей измерения является *dpi* (dots per inch) – количество пикселов в одном дюйме длины (2,54 см). Не следует отождествлять шаг с размерами пикселов – размер пикселов может быть равен шагу, а может быть как меньше, так и больше, чем шаг.

Размер растра обычно измеряется количеством пикселов по горизонтали и вертикали. Можно сказать, что для компьютерной графики зачастую наиболее удобен растр с одинаковым шагом для обеих осей, т. е. dpi$X$ = dpi$Y$. Это удобно для многих алгоритмов вывода графических объектов.

Форма пикселов растра определяется особенностями устройства графического вывода (рис. 2.2).

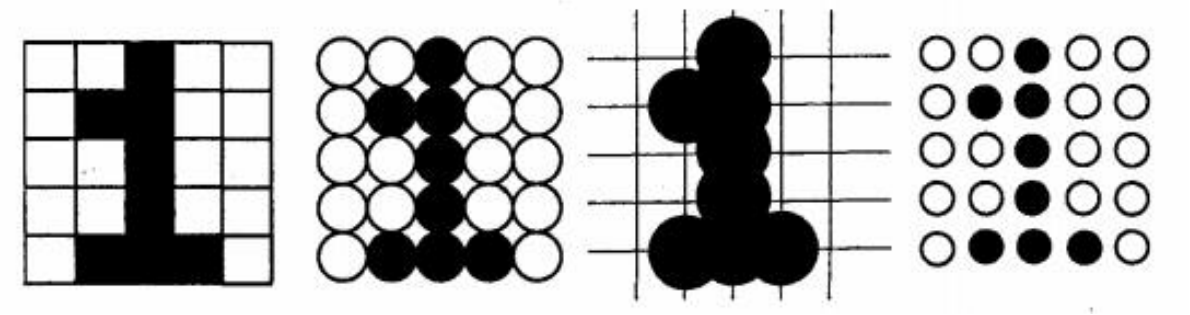


Рис. 2.2

Например, пикселы могут иметь форму прямоугольника или квадрата, которые по размерам равны шагу растра (дисплей на жидких кристаллах); пикселы круглой формы, которые по размерам могут и не равняться шагу растра (принтеры).

2.4. Количество цветов

Количество цветов (глубина цвета) – также одна из важнейших характеристик растра. Количество цветов является важной характеристикой для любого изображения, а не только растрового. Согласно психофизиологическим исследованиям, глаз человека способен различать 350 000 цветов.

Изображения классифицируются следующим образом:

– двухцветные (бинарные) – 1 бит на пиксел. Среди двухцветных чаще всего встречаются черно-белые изображения;

– полутоновые – градации серого или иного цвета. Например, 256 градаций (1 байт на пиксел);

– цветные изображения – 2 бита на пиксел и выше. Глубина цвета 16 бит на пиксел (65 536 цветов) получила название HighColor, 24 бит на пиксел (16,7 млн. цветов) – TrueColor. В компьютерных графических системах используют и большую глубину цвета – 32, 48 и более бит на пиксел.

2.5. Кодирование цвета. Палитра

Для того чтобы компьютер имел возможность работать с цветными изображениями, необходимо представлять цвета в виде чисел – кодировать цвет. Способ кодирования зависит от цветовой модели и формата числовых данных в компьютере.

Чтобы оцифровать цвет, его необходимо измерить. Наука, которая изучает цвет и его измерения, называется **колориметрией**. Она описывает общие закономерности цветового восприятия света человеком. Одними из основных законов колориметрии являются законы смешивания цветов. Эти законы в наиболее полном виде были сформулированы в 1855 году немецким математиком Германом Грассманом:

– закон трехмерности – любой цвет может быть представлен комбинацией трех основных цветов;

– закон непрерывности – к любому цвету можно подобрать бесконечно близкий;

– закон аддитивности – цвет смеси зависит только от цвета составляющих.

Первый закон означает, что для любого заданного цвета (*Color*) можно записать следующее цветовое уравнение, выражающее линейную зависимость цветов:

$$Color = k_1 Color1 + k_2 Color2 + k_3 Color3,$$

где *Color*1, *Color*2, *Color*3 – некоторые базисные, линейно независимые цвета; коэффициенты k_1, k_2, k_3 указывают количество соответствующего смешиваемого цвета. Линейная независимость цветов означает, что ни один из них не может быть выражен взвешенной суммой (линейной комбинацией) двух других.

Цветовая модель RGB. За основные три цвета приняты красный (Red), зеленый (Green), синий (Blue). В модели RGB любой цвет (Color) получается в результате сложения основных цветов.

Для модели RGB каждая из компонент может представляться числами, ограниченными некоторым диапазоном – например,

дробными числами от 0 до 1 либо целыми числами от 0 до некоторого максимального значения. В настоящее время достаточно распространенным является формат TrueColor, в котором каждая компонента представлена в виде байта, что дает 256 градаций для каждой компоненты: R = 0–255, G = 0–255, B = 0–255. Количество цветов составляет 256·256·256 = 16,7 млн (2^{24}).

Такой способ кодирования цветов можно назвать **компонентным**. В компьютере коды изображений TrueColor представляются в виде троек байтов либо упаковываются в длинное целое (четырехбайтное) – 32 бита (так, например, сделано в APIWindows):

C = 00000000 bbbbbbbb gggggggg rrrrrrrr.

Цветовая модель RGB применяется для создания графических образов в устройствах, излучающих свет – мониторах, телевизорах.

Цветовая модель CMYK. В полиграфических системах напечатанный на бумаге графический объект сам не излучает световых волн. Изображение формируется на основе отраженной волны от окрашенных поверхностей. Окрашенные поверхности, на которые падает белый свет (т. е. сумма всех цветов), должны поглотить (т. е. вычесть) все составляющие цвета кроме той, цвет которой мы видим. Цвет поверхности можно получить красителями, которые поглощают, а не излучают. Например, если мы видим зеленое дерево, то это означает, что из падающего белого цвета, т. е. суммы красного, зеленого, синего, поглощены красный и синий, а зеленый отражен. Цвета красителей должны быть дополняющими:

– голубой (Cyan = B + G), дополняющий красного;

– пурпурный (Magenta = R + B), дополняющий зеленого;

– желтый (Yellow = R + G), дополняющий синего.

Но так как цветные красители по отражающим свойствам не одинаковы, то для повышения контрастности применяется еще черный (Black). Модель CMYK названа по первым буквам слов Cyan, Magenta, Yellow и последней букве слова Black (по одной из версий). Так как цвета вычитаются, модель называется **субстрактивной**.

При работе с изображениями в системах компьютерной графики часто приходится искать компромисс между качеством изображения (требуется как можно больше цветов) и ресурсами, необходимыми для хранения и воспроизведения изображения, исчисляемыми, например, объемом памяти (надо уменьшать количество бит на пиксел).

Кроме того, некоторое изображение само по себе может использовать ограниченное количество цветов. Например, для черче-

ния может быть достаточно двух цветов, для человеческого лица важны оттенки розового, желтого, пурпурного, красного, зеленого; а для неба – оттенки голубого и серого. В этих случаях использование полноцветного кодирования цвета является избыточным.

При ограничении количества цветов используют **палитру**, представляющую набор цветов, важных для данного изображения. Палитру можно воспринимать как таблицу цветов. Палитра устанавливает взаимосвязь между кодом цвета и его компонентами в выбранной цветовой модели.

2.6. Графический формат

Графическим форматом называют порядок (структуру), согласно которому данные, описывающие изображение, записаны в файле.

Типы форматов графических файлов определяются способом хранения и типом графических данных. Наиболее широко используются **векторный**, **растровый (битовый)** и **метафайловый** форматы.

Векторный формат наиболее удобен для хранения изображений, которые можно разложить на простые геометрические фигуры (например, чертежи или текст). Векторные файлы содержат математические описания элементов изображения. Наиболее распространенные векторные форматы: AutoCADDXF и MicrosoftSYLK.

Растровый формат используется для хранения растровых данных. Файлы такого типа особенно хорошо подходят для хранения изображений реального мира, например оцифрованных фотографий. Растровые файлы содержат битовую карту изображения и ее спецификацию. Наиболее распространенные растровые форматы: BMP, TIFF, GIF, PCX, JPEG.

Каждый из этих форматов имеет свои преимущества и недостатки.

Битовые изображения, как правило, выводятся на экран быстрее, так как их внутренняя структура аналогична (до некоторой степени) структуре видеопамяти. Изображения, получаемые при помощи сканеров и цифровых видеокамер, получаются именно как битовые изображения.

К недостаткам битовых изображений можно отнести большой объем памяти, требующийся для их хранения (около 1 Мбайт в режиме True Color), невозможность масштабирования без потери

качества изображения, а также сложность выделения и изменения отдельных объектов изображения.

Векторные изображения состоят из описаний отдельных элементов, поэтому они легко масштабируются. Однако вывод векторных изображений выполняется, как правило, медленнее, чем битовых.

Следует отметить, что некоторые устройства вывода, такие как плоттер (графопостроитель), способны работать только с векторными изображениями, так как с помощью пера можно рисовать только линии.

Метафайловый формат позволяет хранить в одном файле и векторные, и растровые данные. Примером такого формата являются файлы CorelDRAW – CDR.

Кроме того, существуют файловые форматы для хранения мультипликации (видеоинформации), мультимедиа-форматы (одновременно хранят звуковую, видео- и графическую информацию), гипертекстовые (позволяют хранить не только текст, но и связи-переходы внутри него) и гипермедиа-форматы (гипертекст плюс графическая и видеоинформация), форматы трехмерных сцен, форматы шрифтов и т. д.

Глава 3

РАБОТА С РАСТРОВЫМИ ИЗОБРАЖЕНИЯМИ

3.1. Виды растровых форматов

К настоящему времени известно много форматов файлов для растровых изображений. Здесь мы рассмотрим работу с одним из самых популярных форматов, который обязан своей распространенностью операционной системе Windows, – форматом **bmp** (Bitmap picture).

В операционной системе Windows используются два формата растровых (битовых) изображений – **аппаратно-зависимый DDB** (device dependent bitmap) и **аппаратно-независимый DIB** (device independent bitmap).

Согласно определению, данному в документации к SDK (software development kit – комплект средств разработки), **битовое изображение DDB** есть набор бит в оперативной памяти, который может быть отображен на устройстве вывода (например, выведен на экран видеомонитора или распечатан на принтере). Внутренняя структура изображения DDB жестко привязана к аппаратным особенностям устройства вывода. Поэтому представление изображения DDB в оперативной памяти полностью зависит от устройства вывода.

Для описания растровых изображений в ОС Windows используется структура **BITMAP**:

```
typedef struct tagBITMAP {
LONG bmType;  //Тип битового изображения (= 0)
LONG bmWidth; //Ширина битового изображения в
              //пикселах(> 0)
LONG bmHeight;//Высота битового изображения в
              //пикселах (>0)
LONG bmWidthBytes; //Размер памяти, занимаемый одной
              //строкой растра битового изображения
WORD bmPlanes;//Количество плоскостей в битовом
              //изображении
WORD bmBitsPixel;//Количество бит, используемых
                 //для представления цвета пиксела
LPVOID bmBits;   //Указатель на массив, содержащий
                 //биты изображения
} BITMAP, *PBITMAP;
```

Если бы в Windows можно было работать только с изображениями DDB, было бы необходимо иметь отдельные наборы изо-

бражений для каждого типа видеоконтроллера и каждого видеорежима, что, очевидно, крайне неудобно.

Аппаратно-независимое битовое изображение DIB содержит описание цвета пикселов изображения, которое не зависит от особенностей устройства отображения. Операционная система Windows после соответствующего преобразования может отобразить такое изображение на любом устройстве вывода. Несмотря на некоторое замедление процесса вывода по сравнению с выводом изображений DDB, универсальность изображений DIB делает их весьма привлекательными для хранения изображений.

3.2. Битовые изображения в формате DDB

Как было сказано выше, битовые изображения в формате DDB являются **аппаратно-зависимыми**. Поэтому структура изображения в оперативной памяти зависит от особенностей аппаратуры.

Как правило, изображения DDB либо загружаются из ресурсов приложения, либо создаются непосредственно в оперативной памяти. Для вывода изображений DDB на экран используются такие функции, как **BitBlt** и **StretchBlt**.

Ниже приведено описание функций **BitBlt** и **StretchBlt**.

```
BOOL WINAPI BitBlt(
HDC hdcDest, //Контекст для рисования
int nXDest,  //X-координата верхнего левого угла
             //области рисования
int nYDest,  //Y-координата верхнего левого угла
             //области рисования
int nWidth,  //Ширина изображения
int nHeight, //Высота изображения
HDC hdcSrc,  //Идентификатор исходного контекста
int nXSrc,   //X-координата верхнего левого угла
             //исходной области
int nYSrc,   //Y-координата верхнего левого угла
             //исходной области
DWORD dwRop) //Код растровой операции
```

Функция копирует битовое изображение из исходного контекста **hdcSrc** в контекст отображения **hdcDest**. Возвращаемое значение равно TRUE при успешном завершении или FALSE при ошибке.

В качестве кода растровой операции чаще всего используется константа SRCCOPY. При этом цвет пикселов копируемого изо-

бражения полностью замещает цвет соответствующих пикселов контекста отображения. В этом случае цвет кисти, выбранной в контекст отображения, не имеет значения, так как ни цвет кисти, ни цвет фона не влияют на цвет нарисованного изображения.

Для рисования битовых изображений можно использовать вместо функции **BitBlt** функцию **StretchBlt**, с помощью которой можно выполнить масштабирование (сжатие или растяжение) битовых изображений:

```
BOOL WINAPI StretchBlt(
HDC hdcDest,      //Контекст для рисования
int nXDest,       //X-координата верхнего левого
                  //угла области рисования
int nYDest,       //Y-координата верхнего левого угла
                  //Области рисования
int nWidthDest,   //Новая ширина изображения
int nHeightDest, //Новая высота изображения
HDC hdcSrc,       //Идентификатор исходного контекста
int nXSrc,        //X-координата верхнего левого
                  //угла исходной области
int nYSrc,        //Y-координата верхнего левого
                  //угла исходной области
int nWidthSrc,    //Ширина исходного изображения
Int nheightsrc,   //Высота исходного изображения
DWORD dwRop)      //Код растровой операции
```

Параметры этой функции аналогичны параметрам функции **BitBlt** за исключением того, что ширина и высота исходного и полученного изображения должны определяться отдельно. Размеры исходного изображения (логические) задаются параметрами **nWidthSrc** и **nHeightSrc**, размеры нарисованного изображения задаются параметрами **nWidthDest** и **nHeightDest**.

Возвращаемое значение равно TRUE при успешном завершении или FALSE при ошибке.

3.3. Битовые изображения в формате DIB

Изображения DIB, в отличие от изображений DDB, являются **аппаратно-независимыми**, поэтому без дополнительного преобразования их нельзя отображать на экране с помощью функций **BitBlt** и **StretchBlt**. В операционной системе Windows битовые изображения хранятся в файлах с расширением имени **bmp**, при этом используется аппаратно-независимый формат **DIB**.

Формат **bmp**-файлов представлен на рис. 3.1.

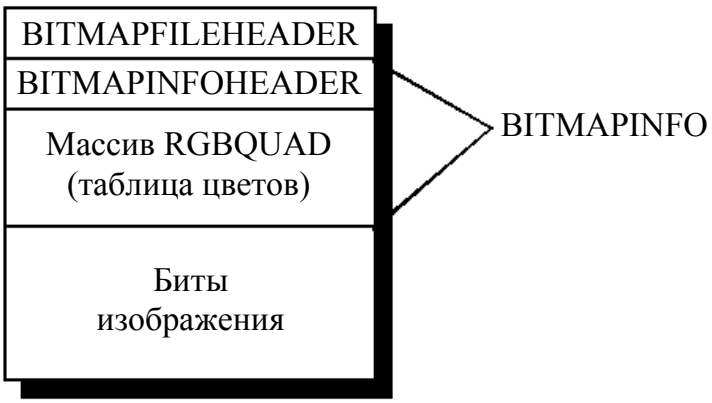


Рис. 3.1

Файл, содержащий битовое изображение, начинается со структуры BITMAPFILEHEADER. Эта структура описывает тип файла и его размер, а также смещение области битов изображения.

```
Typedef struct tagBITMAPFILEHEADER {
WORD bfType; //признак bmp-файла, 42 4D (коды букв BM)
DWORD bfSize;//размер файла
WORD bfReserved1; //первое резервное поле –
                  //всегда ноль
WORD bfReserved2; //второе резервное поле – тоже ноль
DWORD bfOffBits;  //смещение от начала файла до
                  //первого байта графических данных
} BITMAPFILEHEADER, *PBITMAPFILEHEADER;
```

Сразу после структуры BITMAPFILEHEADER в файле следует структура BITMAPINFO, которая содержит описание изображения и таблицу цветов.

```
typedef struct tagBITMAPINFO {
BITMAPINFOHEADER bmiHeader;
RGBQUAD bmiColors[1];
} BITMAPINFO, *PBITMAPINFO;
```

Описание изображения (размеры изображения, метод компрессии, размер таблицы цветов и т. д.) находится в структуре BITMAPINFOHEADER.

```
typedef struct tagBITMAPINFOHEADER{
DWORD biSize; //размер BITMAPINFOHEADER в байтах.
LONGbiWidth;  //ширина картинки в пикселах
LONGbiHeight; //высота картинки в пикселах
WORDbiPlanes; //количество битовых плоскостей (=1)
WORDbiBitCount; //количество бит на пиксел
                //(определяет максимальное число
                //цветов в bitmap)
```

```
    DWORD biCompression; //способ сжатия (0 – данные
                         //не сжимаются)
    DWORD biSizeImage;   //размер изображения в байтах
                         //(без заголовков)
    LONG biXPelsPerMeter; //число пикселов на метр
                          //по горизонтали
    LONG biYPelsPerMeter; //число пикселов на метр
                          //по вертикали
    DWORD biClrUsed; //количество элементов палитры,
                     //хранящихся после заголовка
    DWORDbiClrImportant; //количество разных цветов,
                         //действительно используемых в
рисунке
    } BITMAPINFOHEADER, *PBITMAPINFOHEADER;
```

В некоторых случаях (не всегда) в файле может присутствовать **палитра** – таблица цветов (как массив структур RGBQUAD), имеющихся в изображении.

```
    typedef struct tagRGBQUAD
    {
    BYTE rgbBlue;
    BYTE rgbGreen;
    BYTE rgbRed;
    BYTE rgbReserved;
    } RGBQUAD;
```

Палитра отсутствует, если число бит на пиксел равно 24. Также палитра не нужна и для некоторых цветовых форматов 16 и 32 бит на пиксел.

После палитры (если она есть) в файле bmp записывается растр в виде битового (а точнее, байтового) массива. В битовом массиве последовательно записываются байты строк растра. Количество байт в строке должно быть кратно четырем, поэтому если количество пикселов по горизонтали не соответствует такому условию, то справа в каждую строку дописывается некоторое число битов (выравнивание строк на границу двойного слова).

Точное значение смещения битов изображения находится в структуре BITMAPFILEHEADER.

3.4. Загрузка данных из bmp-файла

Чтобы изображение загрузить с диска в оперативную память и получить дескриптор изображения (типа HBITMAP), используются функции **LoadBitmap()** и **LoadImage()**. Так, например, из файла

с расширением .bmp изображение можно прочитать при помощи функции **LoadImage()**:

```
HBITMAP Bit = (HBITMAP)LoadImage(NULL, char* Filename,
IMAGE_BITMAP, 0, 0, LR_LOADFROMFILE |
LR_CREATEDIBSECTION)
```

Значение NULL **первого параметра** указывает, что картинка вводится из файла. При получении изображения из ресурса в первом параметре указывается дескриптор приложения. Во **втором параметре** задается имя файла или идентификатор ресурса. **Третий параметр** задает тип изображения и может принимать значения IMAGE_BITMAP, IMAGE_CURSOR, IMAGEICON. **Четвертый** и **пятый параметры**, которые указывают размеры изображения, при вводе битовой карты задаются нулями, так как эта информация имеется в заголовке bmp-файла. **Шестой параметр** содержит сочетание флагов. В частности, флаг LR_LOADFROMFILE указывает, что изображение вводится из файла. При задании флага LR_CREATEDIBSECTION создается **аппаратно-независимое**, а при его отсутствии – **аппаратно-зависимое** растровое изображение. Ниже приводится код функции для загрузки битовой карты, которая получает путь к файлу FileName и тип растра Type. Нулевое значение будет означать, что используется тип **DIB**, а единичное значение будет применяться для типа **DDB**.

```
HBITMAP LoadPict(char *FileName, int Type)
{
if (Type)
   return LoadImage(NULL, FileName, IMAGE_BITMAP, 0, 0,
   LR_LOADFROMFILE);
else
return Loadlmage(NULL, FileName, IMAGE_BITMAP, 0, 0,
 LR_LOADFROMFILE|LR_CREATEDIBSECTION);
}
```

Функция **LoadImage()** позволяет загружать из ресурса битовую карту, курсор или пиктограмму, но для этого более удобны простые специализированные функции **LoadBitmap()**, **LoadCursor()**, **LoadIcon()**.

3.5. Вывод растровых изображений на экран

Рассмотрим пример функции для отображения рисунка формата **bmp** в окне Windows.

```
    Int ShowBitMap(HWND hWnd, HBITMAP hBit, int x, int y)
    //функция отображает рисунок в заданной позиции окна
    //hWnd - дескриптор окна, куда выводится изображение
    //hBit - дескриптор рисунка
    //(x,y) - координаты левого верхнего угла
    //изображения в окне вывода
    {
    BITMAP BitMap;
    //в структуру BitMap считываются параметры картинки
    GetObject(hBit, sizeof(BITMAP), &BitMap);
    //из ее полей bmHeight и bmWidth получаем размеры
    //для копирования
    int Height = BitMap.bmHeight;
    int Width = BitMap.bmWidth;
    //получаем контекст изображения
    HDChdc = GetDC(hWnd);
    //создаем контекст памяти
    HDChdcMem = CreateCompatibleDC(hdc);
    //в созданный контекст памяти заносим дескриптор
    //битовой карты
    HBITMAP  OldBitmap  =  (HBITMAP)SelectObject(hdcMem,
hBit);
    //если в исходном тексте программы есть строка
    //#include<windowsx.h>,то дескриптор hBit можно
    //включить в контекст при помощи инструкции:
    //HBITMAP OldBitmap=SelectBitmap(hdcMem,hBit);
    //контекст hdcMem можно использовать для рисования
    //средствами GDI - создаваемое изображение с учетом
    //заданной растровой операции наносится на включенную
    //в контекст картинку.
    //В окно, с которым связан контекст изображения hdc,
    //картинка переносится при помощи функции
    //копирования:
    BitBlt(hdc,x,y,Width,Height,hdcMem,0,0,
    SRCCOPY);
    //PATCOPY
    //после копирования уничтожаются контексты памяти
    //и изображения
    SelectObject(hdcMem, OldBitmap);
    //SelectBitmap(hdcMem, OldBitmap);
    ReleaseDC(hWnd, hdc);
    DeleteDC(hdcMem);
    return 0;
    }
```

Рассмотрим пример функции для сохранения в файле на диске рабочей области окна в формате **bmp**.

```
int ClientToBmp(HWND hWnd,char *Name)
//сохранение рабочей области окна в файле Name.bmp
//hWnd - дескриптор окна, рабочая область которого
//сохраняется
//Name - имя файла для сохранения
{
BITMAPFILEHEADER bmfHdr;
BITMAPINFOHEADER bi;
RECT r;
intBitToPixel=16;//устанавливаем цветовую
                   //глубину 16 бит
GetClientRect(hWnd,&r);//узнаем размер рабочей
                        //области
HDChdc = GetDC(hWnd);
HDChdcMem=CreateCompatibleDC(hdc);
//создаем битовую карту BitMap по размеру рабочей
//области окна.
//Битовая карта создается на основе контекста
//устройства hdc, поэтому она хранит изображение
//DDB-формата (а не DIB).
HBITMAPBitMap = CreateCompatibleBitmap(hdc, r.right,
r.bottom);
HBITMAP OldBitmap = (HBITMAP)SelectObject(hdcMem,
BitMap);
BitBlt(hdcMem, 0, 0, r.right, r.bottom, hdc,
 0, 0, SRCCOPY);
BitMap = (HBITMAP)SelectObject(hdcMem,OldBitmap);
ZeroMemory(&bi,sizeof(BITMAPINFOHEADER));//Это
// аналог функции memset(), который заполняет
//заголовок нулями.
bi.biSize = sizeof(BITMAPINFOHEADER);
bi.biWidth = r.right;
bi.biHeight = r.bottom;
bi.biPlanes = 1;
bi.biBitCount = BitToPixel;//в примерах режим 16
//бит тоже сохраняется,
//как 24 - это DIB,но 8 бит на пиксел работать не
//будут – не записаны после заголовка элементы
//палитры. Для вычисления размера изображения
//в байтах мы увеличиваем значение
//r.right * BitToPixel/8 байт
//на строку до значения, кратного четырем. Это
//вычисление может выполнить и функция GetDIBits()
bi.biSizeImage = (r.right * BitToPixel+31)/32*4*
r.bottom;
DWORDdwWritten; //количество записанных файлов
//Открываем файл
```

```
HANDLE fh = CreateFile(Name, GENERIC_WRITE, 0, NULL,
CREATE_ALWAYS, FILE_ATTRIBUTE_NORMAL |
FILE_FLAG_SEQUENTIAL_SCAN, NULL);
if (fh == INVALID_HANDLE_VALUE) return 2;
bmfHdr.bfType = ('M'<<8)|'B';  //заполняем дисковый
                               //заголовок
bmfHdr.bfSize = bi.biSizeImage + sizeof(bmfHdr)+
bi.biSize; //размер файла
bmfHdr.bfReserved1 = bmfHdr.bfReserved2 = 0;
bmfHdr.bfOffBits = sizeof(bmfHdr) + bi.biSize; //
//смещение до начала данных.
//Запись заголовка в файл
WriteFile(fh, (LPSTR)&bmfHdr, sizeof(bmfHdr),
&dwWritten, NULL);
//Запись в файл загружаемого заголовка
WriteFile(fh, (LPSTR)&bi, sizeof(BITMAPINFOHEADER),
&dwWritten, NULL);
//Выделяем место в памяти для того, чтобы функция
//GetDIBits()перенесла туда коды цвета в DIB-формате
char *lp=(char *) GlobalAlloc(GMEM_FIXED,
 bi.biSizeImage);
//Из карты BitMap строки с нулевой по bi.biHeight
//функция пересылает в массив lp по формату bi
//(беспалитровый формат)
Interr = GetDIBits(hdc, BitMap,0,(UINT)r.bottom, lp,
(LPBITMAPINFO)&bi,DIB_RGB_COLORS);
//Запись изображения на диск
WriteFile(fh, lp, bi.biSizeImage, &dwWritten, NULL);
//Освобождение памяти и закрытие файла
GlobalFree(GlobalHandle(lp));
CloseHandle(fh);
ReleaseDC(hWnd,hdc);
DeleteDC(hdcMem);
if (dwWritten == 0) return 2;
return 0;
}
```

Глава 4

НЕКОТОРЫЕ АЛГОРИТМЫ КОМПЬЮТЕРНОЙ ГРАФИКИ

4.1. Классификация алгоритмов компьютерной графики

Алгоритмы компьютерной графики можно разделить на два уровня: **нижний** и **верхний**. Группа алгоритмов нижнего уровня предназначена для реализации графических примитивов (линий, окружностей, заполнений и т. п.). Эти алгоритмы или подобные им воспроизведены в графических библиотеках языков высокого уровня или реализованы аппаратно в графических процессорах рабочих станций.

Среди алгоритмов **нижнего** уровня можно выделить следующие группы:

I группа – используют простые математические методы и отличаются простотой реализации. Как правило, такие алгоритмы не являются наилучшими по объему выполняемых вычислений или требуемым ресурсам памяти.

II группа – используют более сложные математические методы по сравнению с алгоритмами I группы (но часто и эвристические) и отличаются большей эффективностью.

III группа – алгоритмы, которые могут быть без больших затруднений реализованы аппаратно (допускающие распараллеливание, рекурсивные, реализуемые в простейших командах). В эту группу могут попасть и алгоритмы, представленные в первых двух группах.

К алгоритмам **верхнего** уровня относятся в первую очередь алгоритмы **удаления невидимых линий и поверхностей**. Задача удаления невидимых линий и поверхностей продолжает оставаться центральной в машинной графике. От эффективности алгоритмов, позволяющих решить эту задачу, зависят качество и скорость построения трехмерного изображения.

К задаче удаления невидимых линий и поверхностей примыкает **задача построения (закрашивания) полутоновых (реалистических) изображений**, т. е. учета явлений, связанных с количеством и характером источников света, учета свойств поверхности тела (прозрачность, преломление, отражение света).

Однако при этом не следует забывать, что вывод объектов в алгоритмах верхнего уровня обеспечивается примитивами, реализующими алгоритмы нижнего уровня, поэтому нельзя игнорировать проблему выбора и разработки эффективных алгоритмов нижнего уровня.

Для различных областей применения компьютерной графики на первый план могут выдвигаться разные свойства алгоритмов. Для научной графики большое значение имеет универсальность алгоритма, быстродействие может отходить на второй план. Для систем моделирования, воспроизводящих движущиеся объекты, быстродействие становится главным критерием, поскольку требуется генерировать изображение практически в реальном масштабе времени.

Особенности растровой графики связаны с тем, что обычные изображения, с которыми сталкивается человек в своей деятельности (чертежи, графики, карты, художественные картины и т. п.), реализованы на плоскости, состоящей из бесконечного набора точек. Экран же растрового дисплея представляется матрицей дискретных элементов, имеющих конкретные физические размеры. При этом число их существенно ограничено. Поэтому нельзя провести точную линию из одной точки в другую, а можно выполнить только аппроксимацию этой линии с отображением ее на дискретной матрице (плоскости). Такую плоскость также называют **целочисленной решеткой**, растровой плоскостью или растром. Эта решетка представляется квадратной сеткой с шагом 1. Отображение любого объекта на целочисленную решетку называется **разложением его в растр** или просто **растровым представлением**.

4.2. Базовые растровые алгоритмы

Сформировать растровое изображение можно по-разному. Для того чтобы создать изображение на растровом дисплее, можно просто скопировать готовый растр в видеопамять. Этот растр может быть получен, например, с помощью сканера или цифрового фотоаппарата. А можно создавать изображение объекта путем последовательного рисования отдельных простых элементов.

Простые элементы, из которых складываются сложные объекты, будем называть **графическими примитивами**. Их можно встретить повсюду. Например, для построения изображения используется некоторый набор примитивов, которые поддерживаются определенными графическими устройствами вывода. Графиче-

ские примитивы также можно применять для описания пространственных объектов в базе данных компьютерной системы – модели объектов. Могут использоваться различные множества примитивов для модели и для алгоритмов отображения. Удобно, когда эти множества совпадают, тогда упрощается процесс отображения.

Простейшим и, вместе с тем, наиболее универсальным растровым графическим примитивом является **пиксел**. Любое растровое изображение можно нарисовать по пикселам, но это трудно и долго. Необходимы более сложные элементы, для которых рисуются сразу несколько пикселов.

4.2.1. Инкрементные алгоритмы растеризации

Джек Е. Брезенхэм предложил подход, позволяющий разрабатывать так называемые **инкрементные** алгоритмы растеризации [2]. Основной целью для разработки таких алгоритмов было построение циклов вычисления координат на основе только целочисленных операций сложения/вычитания без использования умножения и деления. Инкрементные алгоритмы выполняются как последовательное вычисление координат соседних пикселов путем добавления приращений координат.

Разработаны инкрементные алгоритмы для вывода отрезка, окружности, эллипса.

4.2.2. Алгоритм вывода прямой линии

Пусть на растре заданы две точки $A_0(x_0, y_0)$ и $A_1(x_1, y_1)$ с целочисленными координатами (рис. 4.1).

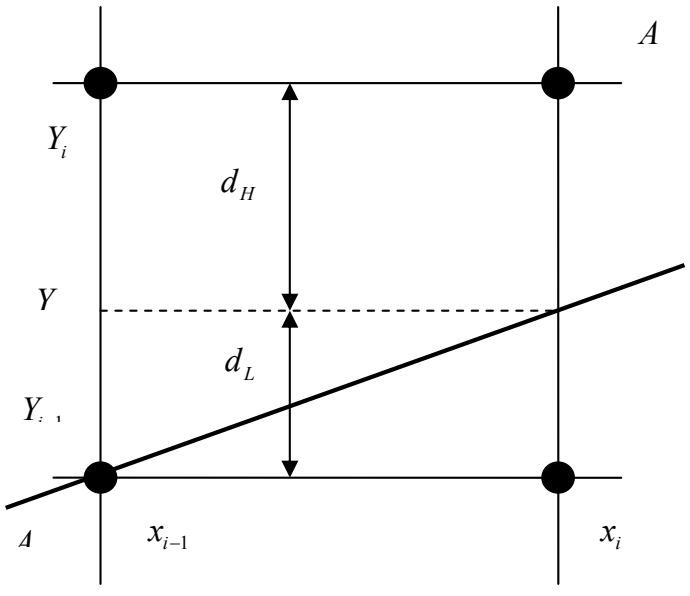


Рис. 4.1

Запишем уравнение прямой, проходящей через эти точки:

$$\frac{x - x_0}{x_1 - x_0} = \frac{y - y_0}{y_1 - y_0}$$

или

$$y = y_0 + \frac{y_1 - y_0}{x_1 - x_0}(x - x_0) = y_0 + \frac{\Delta y}{\Delta x}(x - x_0) = kx + b, \qquad (4.1)$$

где

$$k = \frac{y_1 - y_0}{x_1 - x_0};\ b = y_0 - kx_0;\ \Delta x = x_1 - x_0;\ \Delta y = y_1 - y_0.$$

Будем рассматривать случай для $k < 1$ и $x_1 > x_0$.

Пусть точка $A_{i-1}(x_{i-1}, y_{i-1})$ уже поставлена. Необходимо принять решение о том, какие координаты должна иметь точка $A_i(x_i, y_i)$: $(x_{i-1} + 1, y_{i-1})$ или $(x_{i-1} + 1, y_{i-1} + 1)$.

Запишем выражение для координаты y математической прямой для $x = x_i$:

$$y = kx_i + b = k(x_{i-1} + 1) + b.$$

Вычислим разности:

$$d_L = y - y_{i-1} = k(x_{i-1} + 1) + b - y_{i-1};$$
$$d_H = y_i - y = (y_{i-1} + 1) - y = y_{i-1} + 1 - k(x_{i-1} + 1) - b;$$
$$p_{i-1} = d_L - d_H = [k(x_{i-1} + 1) + b - y_{i-1}] - [y_{i-1} + 1 - k(x_{i-1} + 1) - b] =$$
$$= 2k(x_{i-1} + 1) - 2y_{i-1} + 2b - 1.$$

Тогда

$$y_i = \begin{cases} y_{i-1}, \text{ при } p_{i-1} < 0, \\ y_{i-1} + 1, \text{ при } p_{i-1} \geq 0. \end{cases}$$

Выполним замену в p_{i-1}:

$$k = \frac{\Delta y}{\Delta x};$$

$$p_{i-1} = 2\frac{\Delta y}{\Delta x}(x_{i-1} + 1) - 2y_{i-1} + 2b - 1$$

или

$$\Delta x\, p_{i-1} = 2\Delta y(x_{i-1} + 1) - 2\Delta x y_{i-1} + (2b - 1)\Delta x.$$

Полагая, что $\Delta x\, p_{i-1} = d_{i-1}$, получим

$$d_{i-1} = 2\Delta y x_{i-1} - 2\Delta x y_{i-1} + c, \tag{4.2}$$

где

$$c = 2\Delta y + (2b - 1)\Delta x; \tag{4.3}$$

$$d_i = 2\Delta y x_i - 2\Delta x y_i + c. \tag{4.4}$$

Так как $\Delta x > 0$, то

$$y_i = \begin{cases} y_{i-1}, \text{ при } d_{i-1} < 0, \\ y_{i-1} + 1, \text{ при } d_{i-1} \geq 0. \end{cases}$$

Вычитая (4.2) из (4.4), получим

$$d_i - d_{i-1} = 2\Delta y(x_i - x_{i-1}) - 2\Delta x(y_i - y_{i-1}).$$

Но $x_i - x_{i-1} = 1$, тогда $d_i = d_{i-1} + 2\Delta y - 2\Delta x(y_i - y_{i-1})$.

При этом

$$y_i = \begin{cases} y_{i-1}, \text{ при } d_{i-1} < 0, \\ y_{i-1} + 1, \text{ при } d_{i-1} \geq 0. \end{cases}$$

Поэтому

$$d_i = \begin{cases} d_{i-1} + 2\Delta y, \text{ при } d_{i-1} < 0, \\ d_{i-1} + 2\Delta y - 2\Delta x, \text{ при } d_{i-1} \geq 0. \end{cases}$$

Вычислим начальное значение d_0.

Из (4.2) получаем

$$d_0 = 2\Delta y x_0 - 2\Delta x y_0 + c. \tag{4.5}$$

Значение d_0 соответствует точке $A_0(x_0, y_0)$.

Точка $A_0(x_0, y_0)$ по условию удовлетворяет уравнению прямой (4.1) $y_0 = kx_0 + b$.

Отсюда

$$b = y_0 - kx_0 = y_0 - \frac{\Delta y}{\Delta x} x_0.$$

Подставляя полученное значение b в выражение (4.3) для c, получим

$$c = 2\Delta y + (2b-1)\Delta x = 2\Delta y + \left[2\left(y_0 - \frac{\Delta y}{\Delta x}x_0\right) - 1\right]\Delta x =$$
$$= 2\Delta y + (2\Delta x y_0 - 2\Delta y x_0 - \Delta x).$$

Подставляя полученное выражение для c в выражение (4.5) для d_0, получим

$$d_0 = 2\Delta y x_0 - 2\Delta x y_0 + 2\Delta y + (2\Delta x y_0 - 2\Delta y x_0 - \Delta x) = 2\Delta y - \Delta x.$$

Ниже приведен возможный вариант функции для рисования линии между точками (x_1, y_1) и (x_2, y_2) для $k < 1$ цветом color для работы с использованием библиотеки классов MFC.

```
    void Line(CDC& dc, int x1, int y1, int x2, int y2,
COLORREF color)
    //Алгоритм Брезенхема для прямой y=kx+b, при k<1!!!
    //dc – ссылка на контекст устройства
    //(x1,y1) – координаты начальной точки
    //(x2,y2) – координаты конечной точки
    //color – цвет линии
    {
       int dx = x2 - x1;
       int dy = y2 - y1;
       int d = 2 * dy - dx;
       int d1 = 2 * dy;
       int d2 = 2 * (dy - dx);
       int y = y1;
       dc.SetPixel(x1, y1, color);
       for(int x = x1 + 1; x <= x2; x++)
    {
       if(d < 0)d+= d1;
       else
          {
          y++;
          d+ = d2;
          }
       dc.SetPixel(x, y, color);
    }
    }
```

Приведенная ниже функция реализует алгоритм Брезенхема для любых k.

```
    void MyLine(CDC& dc, int x1, int y1, int x2, int y2,
COLORREF color)
    //Алгоритм Брезенхема для прямой y=kx+b при любых k
```

```
{
 int dx = abs(x2 - x1);
 int dy = abs(y2 - y1);
 int incX = x2>=x1?1:-1;
 int incY = y2>=y1?1:-1;
 if(dy<= dx)
 {
      int d = 2 * dy - dx;
      int d1 = 2 * dy;
      int d2 = 2 * (dy - dx);
      dc.SetPixel(x1, y1, color);
      int x = x1;
      int y = y1;
      for(int i = 1; i <=dx; i++)
      {
           x+= incX;
           if(d<0)d+ = d1;
           else
           {
                y+ = incY;
                d+ = d2;
           }
           dc.SetPixel(x, y, color);
      }
 }
 else
 {
      int d = 2 * dx - dy;
      int d1 = 2 * dx;
      int d2 = 2 * (dx - dy);
      dc.SetPixel(x1, y1, color);
      int x = x1;
      int y = y1;
      for(int i = 1; i <= dy;i++)
      {
           y+ = incY;
           if(d<0)d+ = d1;
           else
           {
                x+ = incX;
                d+ = d2;
           }
           dc.SetPixel(x, y, color);
      }
 }
}
```

4.2.3. Алгоритм построения окружности методом средней точки

Рассмотрим уравнение окружности радиуса r с центром в начале координат

$$x^2 + y^2 = r^2. \tag{4.6}$$

Введем в рассмотрение функцию окружности

$$f(x, y) = x^2 + y^2 - r^2. \tag{4.7}$$

Любая точка $A(x, y)$, которая лежит на окружности, удовлетворяет уравнению $f(x, y) = 0$.

Если точка находится внутри круга, то функция окружности будет иметь отрицательное значение. Если точка лежит за пределами круга, значение функции окружности будет положительным. Подытоживая, можно сказать, что относительное положение любой точки с координатами $A(x, y)$ определяется проверкой знака функции окружности:

$$f \to \begin{cases} < 0, \text{ если точка } A(x, y) \text{ лежит внутри окружности,} \\ = 0, \text{ если точка } A(x, y) \text{ лежит на окружности,} \\ > 0, \text{ если точка } A(x, y) \text{ лежит за пределами окружности.} \end{cases} \tag{4.8}$$

Проверка выполняется на каждом этапе выборки для средних положений между пикселами вблизи заданной окружности. Таким образом, функция окружности – это параметр принятия решения в алгоритме средней точки, и для этой функции можно установить операции приращения, как это было сделано для алгоритма построения прямой линии.

На рис. 4.2 показана средняя точка между двумя возможными пикселами в точке выборки. Предположим, что мы только что поставили точку в пикселе с координатами $A_i(x_i, y_i)$. Теперь нужно определить, какой из двух пикселов ближе к заданной окружности – пиксел с координатами $(x_i + 1, y_i)$ или пиксел с координатами $(x_i + 1, y_i - 1)$. Параметром принятия решения будет функция окружности (4.7), которая рассчитывается для средней точки между этими двумя пикселами:

$$d_i = f\left(x_i + 1, y_i - \frac{1}{2}\right) = (x_i + 1)^2 + \left(y_i - \frac{1}{2}\right)^2 - r^2.$$

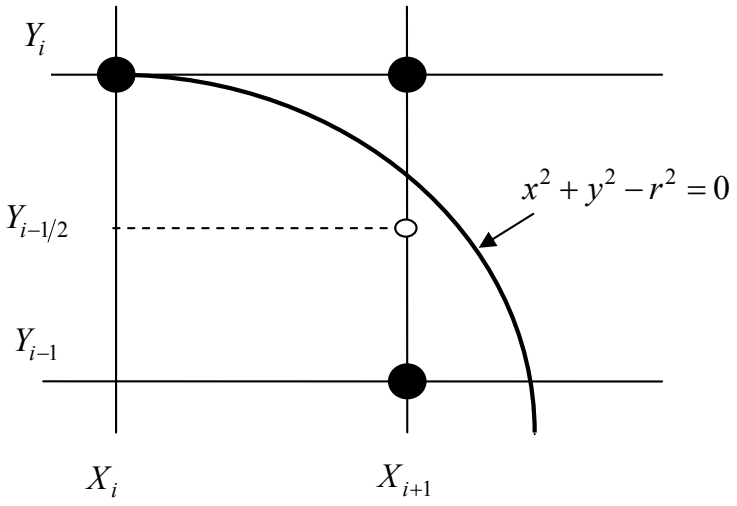


Рис. 4.2

В случае, когда $d_i < 0$, полагаем $y_{i+1} = y_i$ (если средняя точка **внутри** окружности, то она (окружность) ближе к верхнему пикселу). Тогда

$$d_{i+1} = f\left(x_i + 2, y_i - \frac{1}{2}\right) = (x_i + 2)^2 + \left(y_i - \frac{1}{2}\right)^2 - r^2;$$

$$\Delta d_i = d_{i+1} - d_i = 2x_i + 3;$$

$$d_{i+1} = d_i + \Delta d_i = d_i + 2x_i + 3 = d_i + p_i,$$

где

$$p_i = 2x_i + 3 = 2(x_{i-1} + 1) + 3 = 2x_{i-1} + 3 + 2 = p_{i-1} + 2, \quad i = 1, 2, 3, \ldots.$$

При этом

$$p_0 = 2x_0 + 3.$$

Тогда

$$d_{i+1} = d_i + p_i, \ \ i = 0,1,2,\ldots;$$

$$p_i = p_{i-1} + 2, \quad i = 1,2,3,\ldots \text{, для } \ d_i < 0.$$

В случае, когда $d_i \geq 0$, делаем шаг **вниз** (если средняя точка **вне** окружности, то она (окружность) ближе к нижнему пикселу), полагая $y_{i+1} = y_i - 1$. Тогда

$$d_{i+1} = f\left(x_i + 2, y_i - \frac{3}{2}\right) = (x_i + 2)^2 + \left(y_i - \frac{3}{2}\right)^2 - r^2;$$

$$\Delta d_i = d_{i+1} - d_i = 2x_i + 3 - 2y_i + 2 = 2(x_i - y_i) + 5;$$
$$d_{i+1} = d_i + 2(x_i - y_i) + 5 = d_i + 2x_i + 3 - 2y_i + 2 =$$
$$= d_i + p_i - 2y_i + 2 = d_i + p_i + s_i = d_i + q_i,$$

где

$$q_i = p_i + s_i;$$
$$s_i = -2y_i + 2.$$

Для рассматриваемого случая, когда $d_i \geq 0$, $y_i = y_{i-1} - 1$, получаем

$$s_i = -2y_i + 2 = -2(y_{i-1} - 1) + 2 = -2y_{i-1} + 2 + 2 = s_{i-1} + 2, \quad i = 1, 2, 3, \ldots.$$

При этом $s_0 = -2y_0 + 2$. Тогда

$$q_i = p_i + s_i = p_{i-1} + 2 + s_{i-1} + 2 = q_{i-1} + 4, \ i = 1, 2, 3, \ldots.$$

$$q_0 = p_0 + s_0.$$

Таким образом

$$d_{i+1} = d_i + q_i, \ i = 0, 1, 2, ...;$$
$$q_i = q_{i-1} + 4, \ i = 1, 2, 3, ...,$$

для $d_i \geq 0$.

Рассмотрим алгоритмы вычисления s_i и q_i для случая, когда $d_i < 0$, $y_i = y_{i-1}$.

В этом случае

$$s_i = -2y_i + 2 = -2y_{i-1} + 2 = s_{i-1}.$$

И выражение для q_i принимает вид

$$q_i = p_i + s_i = p_{i-1} + 2 + s_{i-1} = q_{i-1} + 2, \quad i = 1, 2, 3, \ldots.$$

Таким образом, если $d_i < 0$, то

$$\begin{cases} x_{i+1} = x_i + 1, \\ y_{i+1} = y_i, \\ d_{i+1} = d_i + p_i, \quad i = 0, 1, 2, ...; \end{cases}$$

$$\begin{cases} p_i = p_{i-1} + 2, \\ q_i = q_{i-1} + 2, \quad i = 1, 2, 3, ..., \end{cases}$$

если $d_i \geq 0$, то

$$\begin{cases} x_{i+1} = x_i + 1, \\ y_{i+1} = y_i - 1, \\ d_{i+1} = d_i + q_i, \quad i = 0, 1, 2, \ldots; \end{cases}$$

$$\begin{cases} p_i = p_{i-1} + 2, \\ q_i = q_{i-1} + 4, \quad i = 1, 2, 3, \ldots. \end{cases}$$

Для удобства заменим в выражениях для p_i и q_i i на $i+1$, тогда для $d_i < 0$

$$\begin{cases} x_{i+1} = x_i + 1, \\ y_{i+1} = y_i, \\ d_{i+1} = d_i + p_i, \\ p_{i+1} = p_i + 2, \\ q_{i+1} = q_i + 2, \quad i = 0, 1, 2, \ldots. \end{cases}$$

При $d_i \geq 0$ получаем

$$\begin{cases} x_{i+1} = x_i + 1, \\ y_{i+1} = y_i - 1, \\ d_{i+1} = d_i + q_i, \\ p_{i+1} = p_i + 2, \\ q_{i+1} = q_i + 4, \quad i = 0, 1, 2, \ldots. \end{cases}$$

Начальные значения параметров d, p и q определим из условия $x_0 = 0$ и $y_0 = r$.

Тогда

$$d_0 = f(x_0, y_0) = f\left(0, r - \frac{1}{2}\right) = \frac{5}{4} - r.$$

Поскольку радиус r и приращение Δd_i целые числа, то значение d_0 можно округлить и принять:

$$d_0 = 1 - r;$$

$$p_0 = 2x_0 + 3 = 2 \cdot 0 + 3 = 3;$$

$$s_0 = -2y_0 + 2 = -2r + 2;$$

$$q_0 = p_0 + s_0 = 3 - 2r + 2 = -2r + 5.$$

Подводя итог изложенному, запишем в собранном виде алгоритм построения окружности методом средней точки.

Установить начальные значение для x_0, y_0, d_0, p_0, q_0.

Поставить начальные точки.

Выполнять в цикле пока $y > x$:

если $d_i < 0$, то

$$\begin{cases} x_{i+1} = x_i + 1, \\ y_{i+1} = y_i, \\ d_{i+1} = d_i + p_i, \\ p_{i+1} = p_i + 2, \\ q_{i+1} = q_i + 2, \quad i = 0, 1, 2, \ldots, \end{cases}$$

если $d_i \geq 0$, то

$$\begin{cases} x_{i+1} = x_i + 1, \\ y_{i+1} = y_i - 1, \\ d_{i+1} = d_i + q_i, \\ p_{i+1} = p_i + 2, \\ q_{i+1} = q_i + 4, \quad i = 0, 1, 2, \ldots. \end{cases}$$

Поставить точки, соответствующие текущему шагу (рис. 4.3)

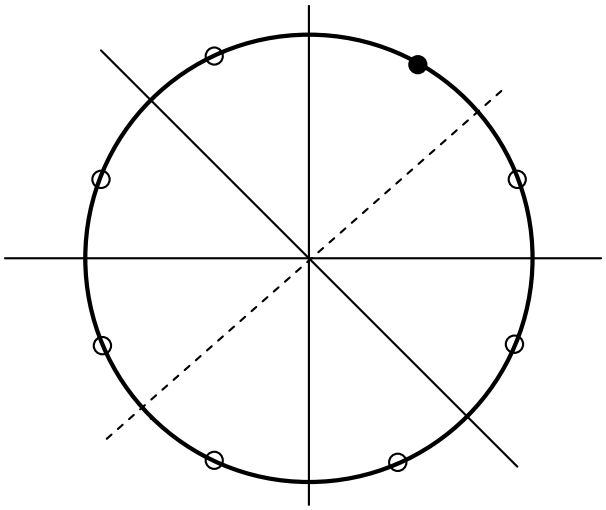


Рис. 4.3

Приводимая ниже функция расставляет точки окружности, соответствующие текущему шагу.

```
    void CirclePoints(CDC& dc, int x, int y, CPoint &P,
COLORREF color)
    {
      dc.SetPixel(x+P.x, y+P.y, color);  //1
      dc.SetPixel(y+P.x, x+P.y, color);  //2
      dc.SetPixel(y+P.x, -x+P.y, color); //3
      dc.SetPixel(x+P.x, -y+P.y, color); //4
      dc.SetPixel(-x+P.x, -y+P.y, color);//5
      dc.SetPixel(-y+P.x, -x+P.y, color);//6
      dc.SetPixel(-y+P.x, x+P.y, color); //7
      dc.SetPixel(-x+P.x, y+P.y, color); //8
    }
```

Приводимая ниже функция реализует алгоритм построения окружности методом средней точки.

```
    void MyCircle(CDC& dc, CPoint &P, int r, COLORREF
color)
    {
      int x = 0;
      int y = r;
      int d = 1 - r;
      int p = 3;
      int q = -2 * r + 5;
      CirclePoints(dc, x, y, P, color);
      while(y > x)
      {
            if(d < 0)
            {
                  d+ = p;
                  p+ = 2;
                  q+ =2;
                  x++;
            }
            else
            {
                  d+ = q;
                  p+ = 2;
                  q+ = 4;
                  x++;
                  y--;
            }
            CirclePoints(dc, x, y, P, color);
      }
    }
```

4.3. Стиль линии. Перо

Для описания различных по виду изображений на основе линий используют термины «**стиль линий**» или «**перо**» [2]. Термин перо иногда делает более понятной суть алгоритма вывода линий для некоторых стилей – в особенности для толстых линий. Например, если для тонкой непрерывной линии перо соответствует одному пикселу, то для толстых линий перо можно представить как фигуру или отрезок линии, который скользит вдоль оси линии, оставляя за собой след (рис. 4.4).

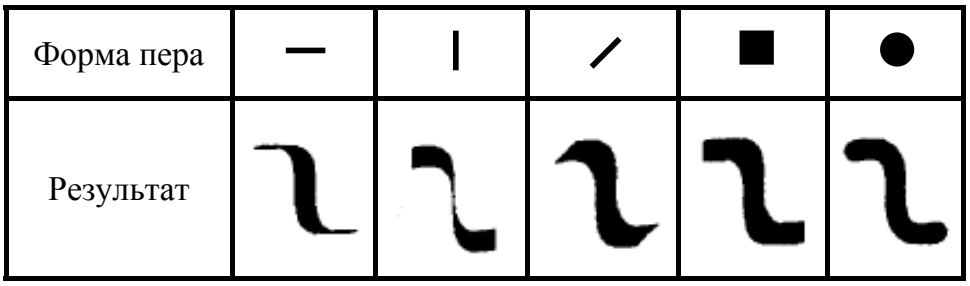


Рис. 4.4

4.3.1. Алгоритмы вывода толстой линии

Взяв за основу любой алгоритм вывода обычных тонких линий (например, алгоритм Брезенхэма), запишем его в следующем обобщенном виде:

Вывод пиксела (*x*, *y*)

Можно представить себе такой алгоритм как цикл, в котором определяются координаты (x, y) каждого пиксела. Этот алгоритм можно модифицировать для вывода толстой линии следующим образом:

Вывод фигуры (или линии) пера с центром (*x*, *y*)

Вместо вывода отдельного пиксела стоит вывод фигуры или линии, соответствующей перу – прямоугольник, круг, отрезок прямой.

Такой подход к разработке алгоритмов толстых линий имеет преимущества и недостатки. Преимущество – можно прямо использовать эффективные алгоритмы для вычисления координат точек линии оси, например алгоритмы Брезенхэма. Недостаток – неэффективность для некоторых форм пера. Для перьев, которые соответствуют фигурам с заполнением, количество тактов работы алгоритма пропорционально квадрату толщины линии. При этом

большинство пикселов многократно закрашивается в одних и тех же точках (рис. 4.5).

Рис. 4.5

Такие алгоритмы более эффективны для перьев в виде отрезков линий. В этом случае каждый пиксел рисуется только один раз. Но здесь важным является наклон изображаемой линии. Ширина пера зависит от наклона (рис. 4.6).

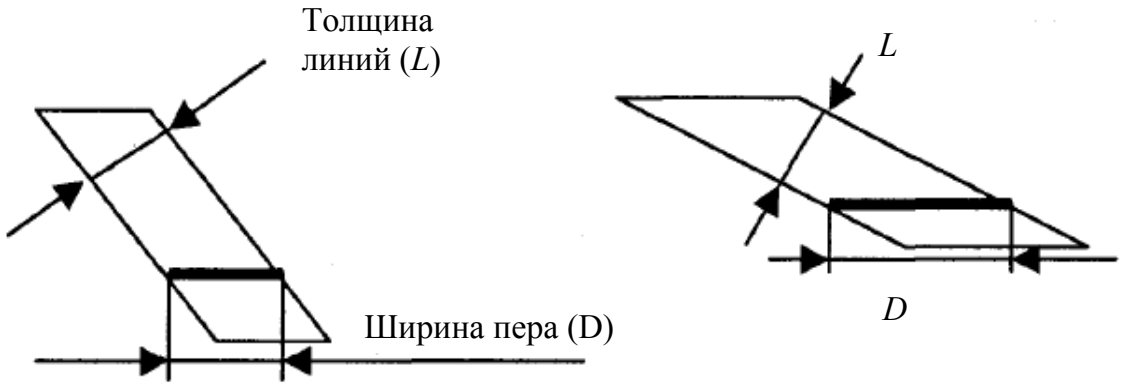


Рис. 4.6

Очевидно, что горизонтальное перо не может рисовать толстую горизонтальную линию.

Для вывода толстых линий с помощью пера в качестве отрезка линии чаще всего используются отрезки горизонтальной или вертикальной линий, реже – диагональные отрезки под углом 45°. Целесообразность такого способа определяется большой скоростью вывода горизонтальных и вертикальных отрезков прямой. Для того чтобы достигнуть минимального количества тактов вывода, толстые линии, которые по наклону ближе к вертикальным, рисуют горизонтальным пером, а пологие линии – вертикальным пером.

4.3.2. Алгоритмы вывода пунктирной линии

Алгоритм для рисования тонкой пунктирной линии можно получить из алгоритма вывода тонкой непрерывной линии заменой процедуры вывода пиксела более сложной конструкцией:

Вывод пиксела (x, y)
Проверка значения счетчика C:
Если C удовлетворяет некоторым условиям,
то вывод пиксела (x, y)
Значение C увеличивается на единицу

При выводе полилиний, которые состоят из отрезков прямых, необходимо предотвратить обнуление значения счетчика в начале каждого отрезка и обеспечить продолжение непрерывного приращения вдоль всей сложной линии. Иначе будут нестыковки пунктира. Использование переменной-счетчика затруднено при генерации пунктирных линий в алгоритмах, которые используют симметрию, например при выводе круга или эллипса. В этом случае будут нестыковки пунктира на границах октантов или квадрантов.

4.4. Стиль заполнения. Кисть. Текстура

При выводе фигур могут использоваться различные стили заполнения. Для обозначения стилей заполнения, отличных от сплошного стиля, используют такие понятия, как **кисть** и **текстура**. Их можно считать синонимами, однако понятие текстуры обычно используется применительно к трехмерным объектам, а кисть – для изображения двумерных объектов. Текстура – это стиль заполнения, закрашивание, которое имитирует сложную рельефную объемную поверхность, выполненную из какого-то материала.

Для описания алгоритмов заполнения фигур с определенным стилем можно использовать тот же способ, что и для описания алгоритмов рисования линий. Описание всех разновидностей подобных алгоритмов можно дать с помощью обобщенной схемы:

Вывод пиксела заполнения цвета C с координатами (x, y)

Например, в алгоритме вывода полигонов пикселы заполнения рисуются в теле цикла горизонталей, а все другие операции предназначены для подсчета координат этих пикселов. Сплошное заполнение означает, что цвет (C) всех пикселов одинаков, т. е. $C = \text{const}$. Для получения определенного узора необходимо изменять цвет пикселов заполнения. Преобразуем алгоритм заполнения следующим образом:

$C = f(x, y)$

Вывод пиксела заполнения (x, y) цветом C

Функция $f(x,y)$ будет определять стиль заполнения. Аргументами функции цвета являются координаты текущего пиксела заполнения. Однако в отдельных случаях эти аргументы не нужны. Например, если цвет C вычислять как случайное значение в определенных границах: $C = random()$, то можно создать иллюзию шершавой матовой поверхности (рис. 4.7, *а*).

а *б*

Рис. 4.7

Другой стиль заполнения – штриховой (рис. 4.7, *б*). Для него функцию цвета также можно записать в аналитической форме:

$$f(x,y) = \begin{cases} C_{\text{ш}}, \text{ если } (x+y) \bmod S < T, \\ C_{\text{ф}}, \text{ в других случаях,} \end{cases}$$

где $C_{\text{ш}}$ – цвет штрихов; S – период; T – толщина штрихов; $C_{\text{ф}}$ – цвет фона.

Подобную функцию можно записать и для других типов штриховки. Аналитическая форма описания стиля заполнение позволяет достаточно просто изменять размеры штрихов при изменениях масштаба показа, например для обеспечения режима WYSIWYG.

Глава 5

МИРОВЫЕ И ЭКРАННЫЕ КООРДИНАТЫ

При отображении пространственных объектов на экране или на листе бумаги с помощью принтера необходимо знать координаты объектов. Мы рассмотрим две системы координат. Первая – **мировая система координат (МСК)**, которые описывают истинное положение объектов в пространстве с заданной точностью. Другая – **оконные координаты** или **система координат устройства изображения**, в котором осуществляется вывод изображения объектов в заданной проекции.

Мировые координаты объектов являются трехмерными. Положение объекта может быть описано, например, в прямоугольной или сферической системе координат. Где располагается центр системы координат и каковы единицы измерения вдоль каждой оси, не очень важно. Важно то, что для отображения должны быть известны какие-то числовые значения координат отображаемых объектов.

Первоначально рассмотрим работу с плоскими изображениями, для описания которых используются двухмерные мировые координаты. Положим, что изображение объекта сформировано в некоторой плоскости пространства. Рассмотрим задачу получения формул для пересчета мировых координат объекта в оконные.

Пусть в мировом пространстве задана плоскость P. Свяжем с этой плоскостью систему координат XOY и выделим на ней прямоугольную область D (рис. 5.1).

Рассмотрим оконную систему координат X^wOY^w с началом в левом верхнем углу окна и выделим в ней некоторую прямоугольную область D^w (рис. 5.2).

Пусть $M = M(x, y)$ – некоторая точка в мировой системе координат, а $M^w = M^w(x^w, y^w)$ ее образ в оконной системе координат.

Из приведенных рисунков можно получить пропорции:

$$\begin{cases} \dfrac{x - x_L}{x_H - x_L} = \dfrac{x^w - x_L^w}{x_H^w - x_L^w}, \\ \dfrac{y - y_L}{y_H - y_L} = -\dfrac{y^w - y_L^w}{y_H^w - y_L^w}; \end{cases} \tag{5.1}$$

$$\begin{cases} x^w = x_L^w + \dfrac{x_H^w - x_L^w}{x_H - x_L}\left(x - x_L\right), \\ y^w = y_H^w - \dfrac{y_H^w - y_L^w}{y_H - y_L}\left(y - y_L\right). \end{cases} \tag{5.2}$$

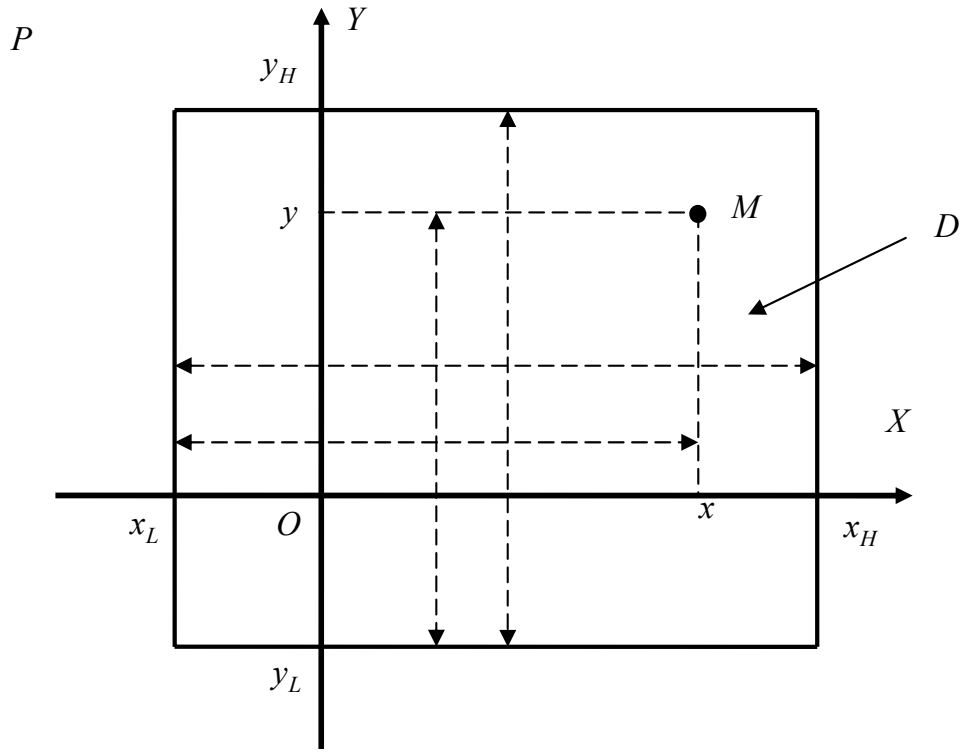


Рис. 5.1

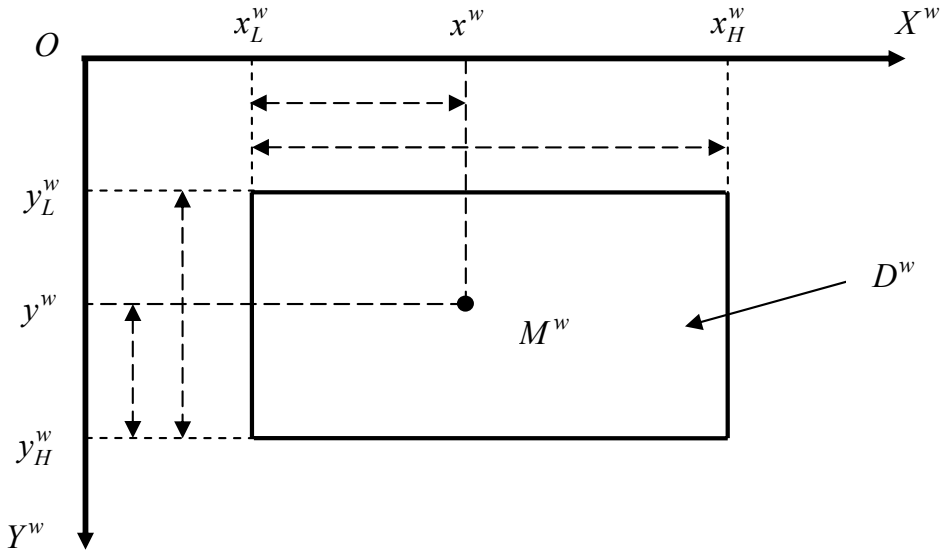


Рис. 5.2

Введем обозначения:

$x_H^w - x_L^w = \Delta x^w$ – ширина области отображения в оконных координатах;

$x_H - x_L = \Delta x$ – ширина области отображения в мировых координатах;

$y_H^w - y_L^w = \Delta y^w$ – высота области отображения в оконных координатах;

$y_H - y_L = \Delta y$ – высота области отображения в мировых координатах.

Тогда выражение (5.2) можно переписать в виде

$$\begin{cases} x^w = x_L^w + \dfrac{\Delta x^w}{\Delta x}(x - x_L), \\ y^w = y_H^w - \dfrac{\Delta y^w}{\Delta y}(y - y_L) \end{cases} \tag{5.3}$$

или

$$\begin{cases} x^w = x_L^w + k_x(x - x_L), \\ y^w = y_H^w - k_y(y - y_L), \end{cases} \tag{5.4}$$

где $k_x = \dfrac{\Delta x^w}{\Delta x}$, $k_y = \dfrac{\Delta y^w}{\Delta y}$.

Представим соотношения (5.4) в матричном виде:

$$\begin{pmatrix} x^w \\ y^w \\ 1 \end{pmatrix} = \begin{pmatrix} k_x & 0 & x_L^w \\ 0 & -k_y & y_H^w \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} x - x_L \\ y - y_L \\ 1 \end{pmatrix} \tag{5.5}$$

или

$$V^w = T_1 V_1, \tag{5.6}$$

где

$$V_1^w = \begin{pmatrix} x^w \\ y^w \\ 1 \end{pmatrix}, \quad T_1 = \begin{pmatrix} k_x & 0 & x_L^w \\ 0 & -k_y & y_H^w \\ 0 & 0 & 1 \end{pmatrix}, \quad V_1 = \begin{pmatrix} x - x_L \\ y - y_L \\ 1 \end{pmatrix}.$$

Перепишем пару рассматриваемых равенств в виде, удобном для дальнейшего использования:

$$\begin{cases} x^w = k_x x + (x_L^w - k_x x_L), \\ y^w = -k_y y + (y_h^w + k_y y_L). \end{cases} \tag{5.7}$$

Представим (5.7) также и в матричном виде:

$$\begin{pmatrix} x^w \\ y^w \\ 1 \end{pmatrix} = \begin{pmatrix} k_x & 0 & x_L^w - k_x x_L \\ 0 & -k_y & y_H^w + k_y y_L \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}, \tag{5.8}$$

или

$$V^w = TV, \tag{5.9}$$

где

$$V^w = \begin{pmatrix} x^w \\ y^w \\ 1 \end{pmatrix}, \quad T = \begin{pmatrix} k_x & 0 & x_L^w - k_x x_L \\ 0 & -k_y & y_H^w + k_y y_L \\ 0 & 0 & 1 \end{pmatrix}, \quad V = \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}.$$

Глава 6

ФИЗИЧЕСКИЕ И ЛОГИЧЕСКИЕ КООРДИНАТЫ

6.1. Основные определения

Физические координаты, как это следует из названия, имеют непосредственное отношение к физическому устройству вывода. В качестве единицы измерения длины в системе физических координат всегда используется пиксел. Если устройством вывода является экран монитора, физические координаты обычно называют **экранными координатами**.

Логические координаты передаются функциям GDI, выполняющим рисование фигур или вывод текста. Используемые единицы измерения зависят от режима отображения.

При отображении GDI преобразует логические координаты в физические. Способ преобразования зависит от режима отображения и других атрибутов контекста отображения, таких как расположение начала системы координат для окна, расположение начала системы физических координат, масштаб осей для окна и масштаб осей физических координат.

6.2. Физическая система координат

На рис. 6.1 показана физическая система координат для экрана видеомонитора. Начало этой системы координат располагается в левом верхнем углу экрана. Ось *X* направлена слева направо, ось *Y* – сверху вниз. В качестве единицы длины в данной системе координат используется пиксел.

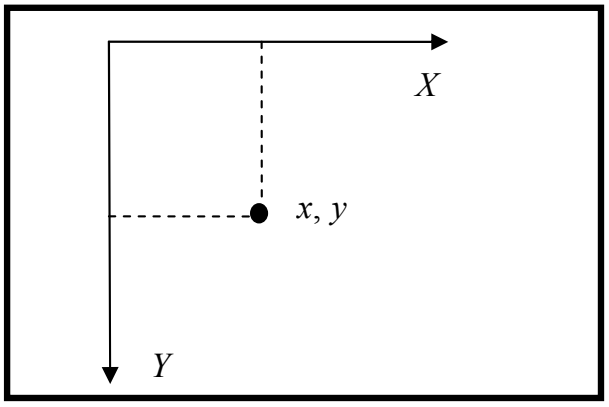


Рис. 6.1

6.3. Логическая система координат

Приложения Windows могут использовать одну из нескольких логических координат, устанавливая соответствующий режим отображения в контексте отображения. При этом можно использовать любое направление координатных осей и любое расположение начала координат. Например, возможна система координат, в которой задаются положительные и отрицательные координаты по любой оси (рис. 6.2).

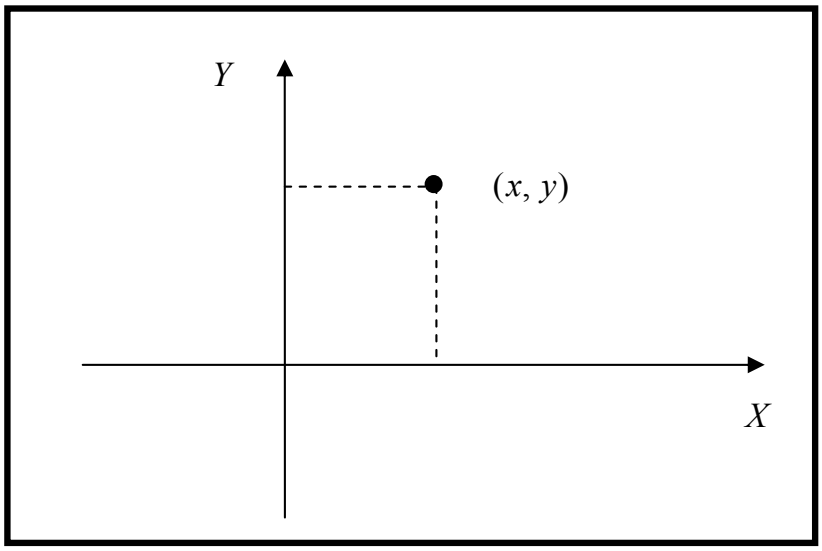


Рис. 6.2

Для установки режима отображения, непосредственно определяющего направление осей и размер логической единицы системы координат, используется функция **SetMapMode**:

```
int dc.SetMapMode(int nMapMode)  // MFC
```

Параметр **nMapMode** может принимать одно из следующих значений, представленных в таблице.

Значения параметра nMapMode

Режим отображения	Направление оси *X*	Направление оси *Y*	Размер одной логической единицы
MM_TEXT	Вправо	Вниз	1 пиксел
MM_LOMETRIC	Вправо	Вверх	0,1 мм
MM_HIMETRIC	Вправо	Вверх	0,01 мм
MM_LOENGLISH	Вправо	Вверх	0,01 дюйма
MM_HIENGLISH	Вправо	Вверх	0,001 дюйма
MM_TWIPS	Вправо	Вверх	1/1440 дюйма

Окончание таблицы

Режим отображения	Направление оси X	Направление оси Y	Размер одной логической единицы
MM_ISOTROPIC	Можно выбирать	Можно выбирать	Произвольный, одинаковый для осей *X* и *Y*
MM_ANISOTROPIC	Можно выбирать	Можно выбирать	Произвольный, может быть разный для осей *X* и *Y*

В любой момент времени приложение может определить номер режима отображения, выбранный в контекст отображения, используя функцию **GetMapMode**:

```
Int dc.GetMapMode() // MFC
```

6.4. Преобразование координат

Приложение, вызывая для рисования функции GDI, указывает **логические** координаты. Перед выводом GDI преобразует их в **физические** с использованием следующих формул:

$$\begin{cases} xViewport = (xWindow - xWinOrg)\dfrac{xViewExt}{xWinExt} + xViewOrg, \\ yViewport = (yWindow - yWinOrg)\dfrac{yViewExt}{yWinExt} + yViewOrg, \end{cases} \quad (6.1)$$

где *xWindow*, *yWindow* – логические координаты по оси *X* и *Y* соответственно;

xViewport, *yViewport* – физические (экранные) координаты по оси *X* и *Y* соответственно;

xWinOrg, *yWinOrg* – определяют расположение начала **логической** системы координат, по умолчанию *xWinOrg* = 0, *yWinOrg* = 0;

xViewOrg, *yViewOrg* – определяют расположение начала **физической** системы координат, по умолчанию *xViewOrg* = 0, *yViewOrg* = 0;

xViewExt, *xWinExt* – задают масштаб, который используется в процессе преобразования координат по оси *X*;

yViewExt, *yWinExt* – задают масштаб, который используется в процессе преобразования координат по оси *Y*.

Эти масштабы зависят от установленного режима отображения. Приложения могут изменить его только в режимах MM_ISOTROPIC

и MM_ANISOTROPIC, для остальных режимов отображения используются фиксированные значения.

Таким образом, логические координаты (*xWindow*, *yWindow*) преобразуются в физические координаты (*xViewport*, *yViewport*).

6.5. Режимы отображения

После рассмотрения физических и логических координат, а также их преобразований займемся подробным описанием каждого режима отображения.

6.5.1. Режим MM_TEXT

Режим отображения MM_TEXT устанавливается в контексте отображения по умолчанию. Для этого режима формулы преобразования координат упрощаются:

$$\begin{cases} xViewport = xWindow - xWinOrg + xViewOrg, \\ yViewport = yWindow - yWinOrg + yViewOrg. \end{cases} \quad (6.2)$$

Так как по умолчанию $xViewOrg = 0$, $yViewOrg = 0$, $xWinOrg = 0$, $yWinOrg = 0$, то

$$\begin{cases} xViewport = xWindow, \\ yViewport = yWindow. \end{cases} \quad (6.3)$$

Соответствующая система координат представлена на рис. 6.3 (начало системы координат расположено точно в левом верхнем углу внутренней области окна, рисунок иллюстрирует только направление координатных осей).

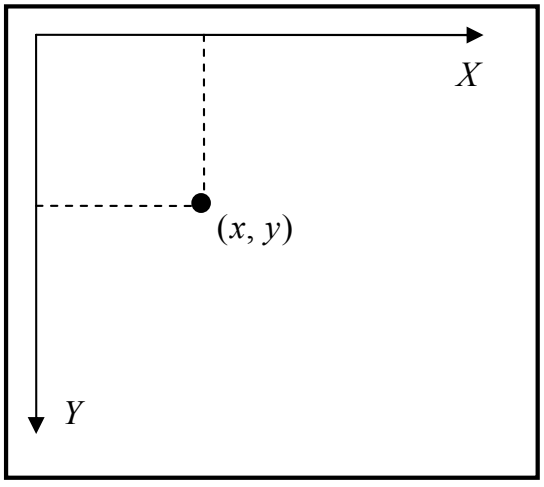


Рис. 6.3

Так как в формуле преобразования не присутствуют переменные *xViewExt*, *xWinExt* и *yViewExt*, *yWinExt*, в данном режиме преобразования невозможно изменить масштаб осей координат. Поэтому логическая единица длины в режиме отображения MM_TEXT равна физической, т. е. одному пикселу.

Тем не менее приложение может изменить смещение физической или логической системы координат, изменив, соответственно, значение пар переменных (*xViewOrg*, *yViewOrg*) и (*xWinOrg*, *yWinOrg*). Для установки смещения можно использовать функции **SetViewportOrg** и **SetWindowOrg**, которые являются методами класса CDC из библиотеки MFC:

```
virtual CPoint SetViewportOrg(intx, int y) или
CPoint SetViewportOrg(CPoint point),
virtual CPoint SetWindowOrg(int x, int y) или
CPointSetWindowOrg(CPoint point),
```

где параметры *x* и *y* представляют собой новые значения соответствующих переменных.

6.5.2. Метрические режимы отображения

К метрическим режимам отображения относятся режимы MM_LOMETRIC, MM_HIMETRIC, MM_LOENGLISH, MM_HIENGLISH и MM_TWIPS. Эти режимы позволяют использовать привычные единицы измерения, такие как миллиметры и дюймы.

В метрических режимах отображения используются полные формулы преобразования координат, приведенные выше. В этих формулах приложение **может изменять** переменные, определяющие смещение начала физической или логической системы координат (*xViewOrg*, *yViewOrg*) и (*xWinOrg*, *yWinOrg*).

Приложение **не может изменить** значения переменных *xViewExt*, *xWinExt* и *yViewExt*, *yWinExt*, от которых зависит масштаб по осям координат. Отношения $xViewExt/xWinExt$ и $yViewExt/yWinExt$ имеют фиксированное значение для каждого из метрических режимов отображения.

Заметим, что для этих режимов отношение $yViewExt/yWinExt$ имеет **отрицательный знак**, в результате чего ось *Y* оказывается направленной **снизу вверх**.

Остановимся подробнее на описании режимов MM_ISOTROPIC и MM_ANISOTROPIC.

Режимы отображения MM_ISOTROPIC (изотропный) и MM_ANISOTROPIC (анизотропный) допускают изменение направ-

ления осей *X* и *Y*, а также изменение масштаба осей координат. В изотропном режиме отображения MM_ISOTROPIC масштаб вдоль осей *X* и *Y* всегда **одинаковый** (т. е. для обеих осей используются одинаковые логические единицы длины). Анизотропный режим MM_ANISOTROPIC предполагает использование разных масштабов для разных осей (хотя можно использовать и одинаковые масштабы).

Для изменения ориентации и масштаба осей можно воспользоваться функциями **SetViewportExt** и **SetWindowExt**, которые являются методами класса CDC из библиотеки MFC:

```
virtual CSize SetWindowExt(int cx, int cy),
virtual CSize SetViewportExt(int cx, int cy).
```

Функция **SetWindowExt** устанавливает для формулы преобразования координат значения переменных *xWinExt* и *yWinExt*.

Функция **SetViewportExt** устанавливает для формулы преобразования координат значения переменных *xViewExt* и *yViewExt*. Функция **SetViewportExt** должна использоваться **после функции SetWindowExt.**

Изотропный режим отображения **удобно использовать** в тех случаях, когда надо сохранить установленное отношение масштабов осей *X* и *Y* при любом изменении размеров окна, в которое выводится изображение.

Анизотропный режим удобен в тех случаях, когда изображение должно занимать всю внутреннюю поверхность окна при любом изменении размеров окна. Соотношение масштабов при этом не сохраняется.

Сравним теперь выражения для пересчета координат из мировых в оконные (5.3), полученные нами ранее:

$$\begin{cases} x^w = x_L^w + \dfrac{\Delta x^w}{\Delta x}(x - x_L), \\ y^w = y_H^w - \dfrac{\Delta y^w}{\Delta y}(y - y_L), \end{cases}$$

и формулы (6.1), по которым пересчитывают координаты функции GDIWindows:

$$\begin{cases} xViewport = (xWindow - xWinOrg)\dfrac{xViewExt}{xWinExt} + xViewOrg, \\ yViewport = (yWindow - yWinOrg)\dfrac{yViewExt}{yWinExt} + yViewOrg. \end{cases}$$

Из сравнения следует, что если положить

$$xWinExt = \Delta x, \qquad xViewExt = \Delta x^w,$$
$$yWinExt = \Delta y, \qquad yViewExt = -\Delta y^w,$$
$$xWinOrg = x_L, \qquad xViewOrg = x_L^w,$$
$$yWinOrg = y_L, \qquad yViewOrg = y_H^w,$$

то пересчет координат из мировых в оконные можно возложить на функции GDI Windows, установив предварительно режим отображения MM_ANISOTROPIC.

Замечание. При использовании режима MM_ANISOTROPIC следует иметь в виду особенности реализации в Win32 API объекта логического пера [6]. Дело в том что при вызове функции создания пера значение параметра, задающего его толщину, определяется отношениями $xViewExt/xWinExt$ и $yViewExt/yWinExt$. При таком определении толщина пера может быть различной для рисования по вертикали и горизонтали. Исключением является нулевое значение толщины, определяющее перо толщиной в 1 пиксел.

Ниже приводится возможный вариант функции (при использовании библиотеки классов MFC) для установки режима отображения MM_ANISOTROPIC, когда заданы области отображения в мировых и оконных координатах.

```
void Set MyMode(CDC&dc, CRect&RS, CRect&RW)  //MFC
//dc - ссылка на класс CDCMFC
//RS - область в мировых координатах
//RW - область в оконных координатах
{
  int dsx = RS.right - RS.left;
  int dsy = RS.top - RS.bottom;
  int xsL = RS.left;
  int ysL = RS.bottom;

  int dwx = RW.right - RW.left;
  int dwy = RW.bottom - RW.top;
  int xwL = RW.left;
  int ywH = RW.bottom;

  dc.SetMapMode(MM_ANISOTROPIC);
  dc.SetWindowExt(dsx, dsy);
  dc.SetViewportExt(dwx, -dwy);
  dc.SetWindowOrg(xsL, ysL);
  dc.SetViewportOrg(xwL, ywH);
}
```

Глава 7

ГЕОМЕТРИЧЕСКИЕ ОСНОВЫ КОМПЬЮТЕРНОЙ ГРАФИКИ

В настоящее время существует большое число прекрасных курсов высшей математики, являющейся тем фундаментом, на котором построены все методы и алгоритмы компьютерной графики. Необходимые разделы математики включаются также и в соответствующие учебные пособия по компьютерной графике [3].

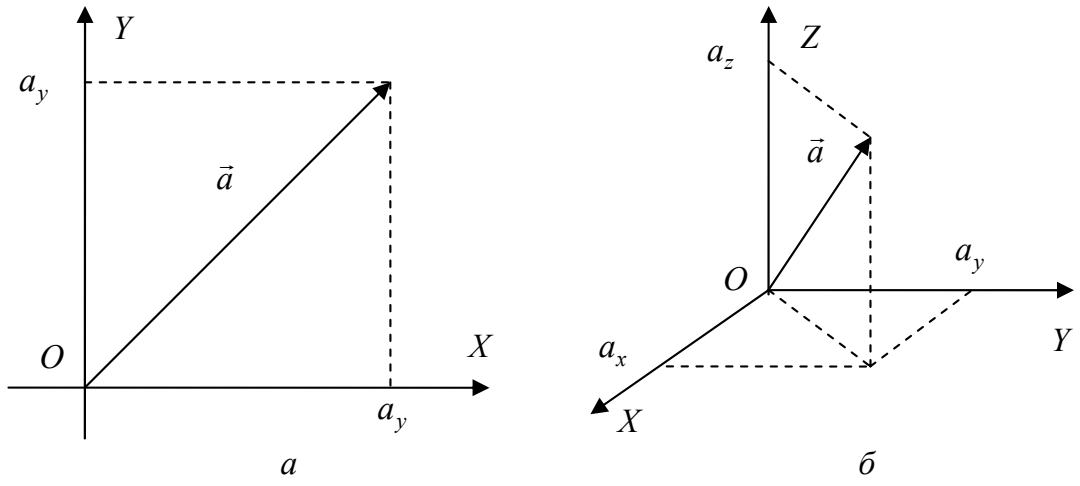
Тем не менее авторы считают необходимым привести в качестве справочного материала некоторые сведения из курса аналитической геометрии и математического анализа, которые имеют отношение к тематике учебного пособия.

7.1. Системы координат и векторы

Рассмотрим различные способы представления векторов на плоскости и в пространстве. Пусть $\vec{a}$ – некоторый вектор, заданный в декартовой системе координат XY на плоскости (рис. 7.1, *а*) и декартовой системе координат XYZ в пространстве (рис. 7.1, *б*).

В прямоугольной системе координат XYZ направление осей задается тройкой перпендикулярных единичных векторов $\vec{i}$, $\vec{j}$, $\vec{k}$.

Система координат называется **правой,** если при повороте от вектора $\vec{i}$ к вектору $\vec{j}$ на $\pi/2$ направление вектора $\vec{k}$ совпадает с поступательным движением винта с правой резьбой (рис. 7.2). Начальная точка векторов обозначается буквой *O*.



а *б*

Рис. 7.1

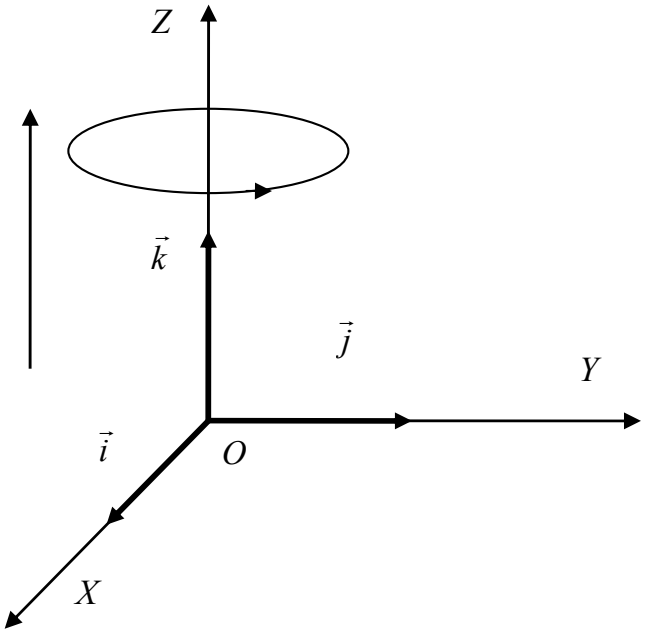


Рис. 7.2

Ниже представлены различные формы записи векторов на плоскости и в пространстве:

$$\vec{a} = a_x\vec{i} + a_y\vec{j} = \vec{a}(a_x, a_y); \tag{7.1}$$

$$\vec{a} = a_x\vec{i} + a_y\vec{j} + a_z\vec{k} = \vec{a}(a_x, a_y, a_z). \tag{7.2}$$

Другая форма записи векторов:

$$a = \begin{pmatrix} a_x \\ a_y \end{pmatrix} = \left(a_x, a_y\right)^T; \tag{7.3}$$

$$a = \begin{pmatrix} a_x \\ a_y \\ a_z \end{pmatrix} = \left(a_x, a_y, a_z\right)^T. \tag{7.4}$$

Модуль (длина) вектора:

$$|a| = \sqrt{a_x^2 + a_y^2} = \sqrt{a^T a}; \tag{7.5}$$

$$|a| = \sqrt{a_x^2 + a_y^2 + a_z^2} = \sqrt{a^T a}. \tag{7.6}$$

Пусть заданы два вектора $\vec{a}$ и $\vec{b}$ в форме (7.4):

$$a = \left(a_x, a_y, a_z\right)^T \text{ и } b = \left(b_x, b_y, b_z\right)^T.$$

Тогда

$$c = a + b = \left(a_x + b_x, a_y + b_y, a_z + b_z\right)^T; \tag{7.7}$$

$$p = \lambda a = \left(\lambda a_x, \lambda a_y, \lambda a_z\right)^T. \tag{7.8}$$

При этом справедливы равенства

$$\left.\begin{aligned}
&\vec{a} + \vec{b} = \vec{b} + \vec{a};\\
&(\vec{a} + \vec{b}) + \vec{c} = \vec{a} + (\vec{b} + \vec{c});\\
&\vec{a} + 0 = \vec{a};\\
&\vec{a} + (-\vec{a}) = 0;\\
&\lambda(\vec{a} + \vec{b}) = \lambda\vec{a} + \lambda\vec{b};\\
&1\vec{a} = \vec{a};\\
&0\vec{a} = 0.
\end{aligned}\right\} \tag{7.9}$$

7.2. Скалярное произведение векторов

Скалярное произведение векторов $\vec{a}$ и $\vec{b}$ (обозначается символом «·») определяется выражением

$$\vec{a} \cdot \vec{b} = (\vec{a}, \vec{b}) = \left|\vec{a}\right|\left|\vec{b}\right|\cos\varphi, \tag{7.10}$$

где φ – угол между векторами a и b.

В координатной форме

$$\vec{a} \cdot \vec{b} = \left(a_x\vec{i} + a_y\vec{j} + a_z\vec{k}\right)\left(b_x\vec{i} + b_y\vec{j} + b_z\vec{k}\right) = a_x b_x + a_y b_y + a_z b_z \tag{7.11}$$

или

$$a \cdot b = a^T b, \tag{7.12}$$

если векторы $\vec{a}$ и $\vec{b}$ представлены в форме (7.4).

Из (7.10) получаем

$$\cos\varphi = \frac{\vec{a} \cdot \vec{b}}{\left|\vec{a}\right| \cdot \left|\vec{b}\right|}. \tag{7.13}$$

Если $\cos\varphi > 0$, то угол между векторами $\vec{a}$ и $\vec{b}$ является острым $\left(\left|\varphi\right| < \frac{\pi}{2}\right)$, при $\cos\varphi = 0$ векторы $\vec{a}$ и $\vec{b}$ ортогональны

$\left(|\varphi| = \frac{\pi}{2}\right)$, а если $\cos\varphi < 0$, то угол между векторами является тупым $\left(|\varphi| > \frac{\pi}{2}\right)$.

Отметим некоторые свойства скалярного произведения.

Пусть p, q, c – некоторые числа. Тогда справедливы равенства:

$$\left.\begin{aligned} &p(q\vec{a}\cdot\vec{b}) = pq(\vec{a}\cdot\vec{b});\\ &(p\vec{a}+q\vec{b})\cdot c = p\vec{a}\cdot c + q\vec{b}\cdot c;\\ &\vec{a}\cdot\vec{b} = \vec{b}\cdot\vec{a};\\ &\vec{a}\cdot\vec{a} = 0 \text{ при } \vec{a} = 0. \end{aligned}\right\} \quad (7.14)$$

7.3. Векторное произведение векторов

Векторное произведение векторов $\vec{a}$ и $\vec{b}$ определяется выражением

$$\vec{c} = \vec{a}\times\vec{b} = [\vec{a},\vec{b}], \quad (7.15)$$

где φ – угол между векторами $\vec{a}$ и $\vec{b}$. Направление вектора $\vec{c}$ перпендикулярно векторам $\vec{a}$ и $\vec{b}$ и таково, что вектора $\vec{a}$, $\vec{b}$, $\vec{c}$ в приведенной последовательности образуют правостороннюю тройку (рис. 7.2).

Отметим некоторые свойства векторного произведения.

Длина вектора $\vec{c}$ равна

$$|\vec{c}| = |\vec{a}||\vec{b}|\sin\varphi. \quad (7.16)$$

Если $\vec{b} = \lambda\vec{a}$, где λ – скаляр, то

$$\vec{c} = \vec{a}\times\vec{b} = \lambda(\vec{a}\times\vec{a}) = 0, \quad (7.17)$$

так как $\vec{a}\times\vec{a} = 0$ (см. ниже).

Пусть p – некоторая константа, тогда справедливы соотношения

$$\left.\begin{aligned} &(p\vec{a})\times\vec{b} = p(\vec{a}\times\vec{b});\\ &\vec{a}\times(\vec{b}+\vec{c}) = \vec{a}\times\vec{b} + \vec{a}\times\vec{c};\\ &\vec{a}\times\vec{b} = -\vec{b}\times\vec{a}. \end{aligned}\right\} \quad (7.18)$$

Рассмотрим еще одну форму записи векторного произведения:

$$\vec{c} = \vec{a} \times \vec{b} = (a_x\vec{i} + a_y\vec{j} + a_z\vec{k}) \times \left(b_x\vec{i} + b_y\vec{j} + b_z\vec{k}\right) =$$

$$= (a_y b_z - a_z b_y)\vec{i} - (a_x b_z - a_z b_x)\vec{j} + (a_x b_y - a_y b_x)\vec{k} = \begin{vmatrix} \vec{i} & \vec{j} & \vec{k} \\ a_x & a_y & a_z \\ b_x & b_y & b_z \end{vmatrix}. \quad (7.19)$$

7.4. Уравнение прямой на плоскости

Пусть в декартовой системе координат на плоскости XY заданы две точки:

$$P_1 = (x_1, y_1)^T \text{ и } P_2 = (x_2, y_2)^T.$$

Уравнение прямой, проходящей через эти точки, имеет вид:

$$\frac{x - x_1}{x_2 - x_1} = \frac{y - y_1}{y_2 - y_1}. \quad (7.20)$$

Полагая в (7.20)

$$\frac{x - x_1}{x_2 - x_1} = \frac{y - y_1}{y_2 - y_1} = t,$$

получаем параметрическое уравнение прямой на плоскости, проходящей через точки P_1 и P_2:

$$\begin{cases} x = x_1 + (x_2 - x_1)t, \\ y = y_1 + (y_2 - y_1)t. \end{cases} \quad (7.21)$$

Пусть $P = (x, y)^T$.

Тогда (7.21) можно записать в виде

$$\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} x_1 \\ y_1 \end{pmatrix} + \begin{pmatrix} x_2 - x_1 \\ y_2 - y_1 \end{pmatrix} t, \quad (7.22)$$

или в более компактной форме

$$P = P_1 + (P_2 - P_1)t. \quad (7.23)$$

Из (7.23) следует, что при $t = 0$ $P = P_1$, а при $t = 1$ получаем, что $P = P_2$.

Таким образом, при $0 \le t \le 1$ точка P принадлежит отрезку P_1P_2.

Следовательно, при $t \in [0, 1]$ уравнение (7.23) можно считать уравнением отрезка P_1P_2.

Рассмотрим задачу о взаимном расположении двух отрезков на плоскости.

Условие задачи. На плоскости заданы два отрезка прямой – A_1A_2 и B_1B_2:

$$A_1 = \left(x_1^a, y_1^a\right)^T; \quad A_2 = \left(x_2^a, y_2^a\right)^T;$$
$$B_1 = \left(x_1^b, y_1^b\right)^T; \quad B_2 = \left(x_2^b, y_2^b\right)^T.$$

Определить: пересекаются или нет отрезки A_1A_2 и B_1B_2.

Запишем уравнения прямых A и B, определяемых отрезками A_1A_2 и B_1B_2 соответственно, в параметрической форме:

$$A = A_1 + \left(A_2 - A_1\right)t; \tag{7.24}$$
$$B = B_1 + \left(B_2 - B_1\right)s. \tag{7.25}$$

Предположим, что прямые пересекаются и P – точка их пересечения.

В этой точке справедливо равенство

$$A = B$$

или

$$A_1 + \left(A_2 - A_1\right)t = B_1 + \left(B_2 - B_1\right)s.$$

Отсюда

$$\left(A_2 - A_1\right)t - \left(B_2 - B_1\right)s = B_1 - A_1;$$
$$\begin{pmatrix} x_2^a - x_1^a \\ y_2^a - y_1^a \end{pmatrix} t - \begin{pmatrix} x_2^b - x_1^b \\ y_2^b - y_1^b \end{pmatrix} s = \begin{pmatrix} x_1^b - x_1^a \\ y_1^b - y_1^a \end{pmatrix}$$

или

$$\begin{cases} \left(x_2^a - x_1^a\right)t - \left(x_2^b - x_1^b\right)s = x_1^b - x_1^a, \\ \left(y_2^a - y_1^a\right)t - \left(y_2^b - y_1^b\right)s = y_1^b - y_1^a. \end{cases} \tag{7.26}$$

Определитель системы (7.26) равен

$$\Delta = \begin{vmatrix} x_2^a - x_1^a - (x_2^b - x_1^b) \\ y_2^a - y_1^a - (y_2^b - y_1^b) \end{vmatrix}. \tag{7.27}$$

Если $\Delta = 0$, то система уравнений (7.26) решения не имеет: отрезки параллельны (или накладываются).

Если $\Delta \neq 0$, то система уравнений (7.26) имеет единственное решение: $t = t_0$ и $s = s_0$.

При этом если $t_0 \in [0,1]$ и $s_0 \in [0,1]$, то отрезки пересекаются, иначе пересекаются их продолжения, или один отрезок пересекается с продолжением другого.

Подставляя найденные значения t_0 и s_0 в (7.24) или в (7.25), можно определить координаты точки пересечения отрезков P_0:

$$P_0 = A_0 = A_1 + (A_2 - A_1)t_0 \tag{7.28}$$

или

$$P_0 = B_0 = B_1 + (B_2 - B_1)s_0. \tag{7.29}$$

7.5. Уравнение прямой в пространстве

Пусть теперь две точки заданы в пространстве:

$$P_1 = (x_1, y_1, z_1)^T \text{ и } P_2 = (x_2, y_2, z_2)^T. \tag{7.30}$$

Тогда уравнение прямой, проходящей через эти точки, будет иметь вид:

$$\frac{x - x_1}{x_2 - x_1} = \frac{y - y_1}{y_2 - y_1} = \frac{z - z_1}{z_2 - z_1}. \tag{7.31}$$

И в параметрической форме

$$\begin{cases} x = x_1 + (x_2 - x_1)t, \\ y = y_1 + (y_2 - y_1)t, \\ z = z_1 + (z_2 - z_1)t. \end{cases} \tag{7.32}$$

Уравнения (7.32), как и ранее, можно представить в компактной форме

$$P = P_1 + (P_2 - P_1)t. \tag{7.33}$$

Рассмотрим задачу о пересечении плоскости отрезком прямой.

Условие задачи. В декартовой системе координат задана плоскость S, определяемая уравнением

$$Ax + By + Cz + D = 0, \tag{7.34}$$

и отрезок прямой P_1P_2, где точки P_1 и P_2 определяются по (7.30).

Определить: пересекает отрезок P_1P_2 плоскость S (рис. 7.3) или нет (рис. 7.4).

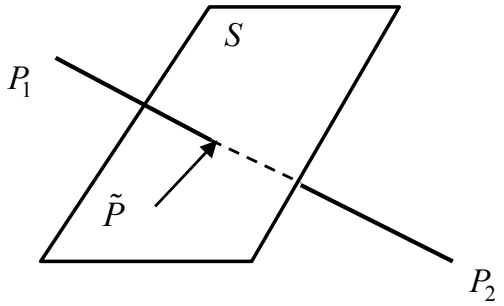


Рис. 7.3

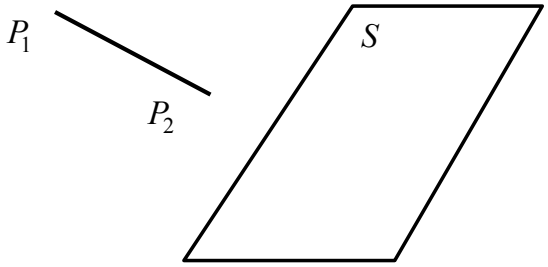


Рис. 7.4

Для решения поставленной задачи определим значение параметра $t = \tilde{t}$, при котором прямая (7.32) и плоскость (7.34) имеют общую точку. Подставляя выражения для координат (7.32) в уравнение плоскости (7.34) и вводя обозначения $\Delta x = x_2 - x_1$, $\Delta y = y_2 - y_1$, $\Delta z = z_2 - z_1$, получим

$$A(x_1 + \Delta x t) + B(y_1 + \Delta y t) + C(z_1 + \Delta z t) + D = 0. \qquad (7.35)$$

Из (7.35) получаем

$$t = \tilde{t} = -\frac{Ax_1 + By_1 + Cz_1 + D}{A\Delta x + B\Delta y + C\Delta z} \qquad (7.36)$$

для $A\Delta x + B\Delta y + C\Delta z \neq 0$.

Тогда если $\tilde{t} \in [0, 1]$, то отрезок P_1P_2 пересекает плоскость S. В противном случае плоскость S пересекает продолжение рассматриваемого отрезка.

Координаты точки пересечения определяются путем подстановки значения $\tilde{t}$ в (7.32):

$$\begin{aligned} \tilde{x} &= x_1 + (x_2 - x_1)\tilde{t}; \\ \tilde{y} &= y_1 + (y_2 - y_1)\tilde{t}; \\ \tilde{z} &= z_1 + (z_2 - z_1)\tilde{t}, \end{aligned} \qquad (7.37)$$

или в компактной форме

$$\tilde{P} = P_1 + (P_2 - P_1)\tilde{t}\,. \qquad (7.38)$$

Рассмотрим случай, когда

$$A\Delta x + B\Delta y + C\Delta z = 0\,. \qquad (7.39)$$

Известно, что в уравнении плоскости коэффициенты A, B и C являются координатами вектора $\vec{N}$, нормального к этой плос-

кости, т. е. $\vec{N} = \vec{N}(A,B,C)$. Рассмотрим вектор $\overrightarrow{P_1P_2}(\Delta x, \Delta y, \Delta z)$, построенный на отрезке P_1P_2. Тогда условие (7.39) означает, что

$$\vec{N} \cdot \overrightarrow{P_1P_2} = 0. \tag{7.40}$$

Равенство нулю скалярного произведения (7.40) означает, что вектор $\overrightarrow{P_1P_2}$ перпендикулярен вектору нормали $\vec{N}$ и, следовательно, параллелен плоскости S. Если при этом также справедливо равенство $Ax_1 + By_1 + Cz_1 + D = 0$, то вектор $\overrightarrow{P_1P_2}$ (отрезок P_1P_2) лежит в плоскости S.

Рассмотрим один частный случай данной задачи, когда плоскость (7.34) параллельна плоскости XY и определяется уравнением $z = z_0$ или

$$z - z_0 = 0. \tag{7.41}$$

В этом случае $A = 0$, $B = 0$, $C = 1$ и $D = -z_0$. Подставляя эти значения в (7.36), получаем

$$\tilde{t} = -\frac{z_1 - z_0}{\Delta z} = -\frac{z_1 - z_0}{z_2 - z_1}. \tag{7.42}$$

Подставив найденное значение $\tilde{t}$ в (37), находим

$$\left.\begin{aligned} \tilde{x} &= x_1 - (x_2 - x_1)\frac{z_1 - z_0}{z_2 - z_1} = x_1 - \frac{\Delta x}{\Delta z}(z_1 - z_0); \\ \tilde{y} &= y_1 - (y_2 - y_1)\frac{z_1 - z_0}{z_2 - z_1} = y_1 - \frac{\Delta y}{\Delta z}(z_1 - z_0); \\ \tilde{z} &= z_0. \end{aligned}\right\} \tag{7.43}$$

Рассмотренная задача возникает при построении линий уровня (см. ниже).

7.6. Вычисление нормали к поверхности

В данном разделе рассматривается задача вычисления вектора нормали к поверхности, которая возникает во многих задачах компьютерной графики.

Сначала вычислим нормаль к плоскости, которая определяется расположенными на ней тремя точками (рис. 7.5).

Пусть точки $A_1 = A_1(x_1, y_1, z_1)$, $A_2 = A_2(x_2, y_2, z_2)$ и $A_3 = A_3(x_3, y_3, z_3)$ лежат в одной плоскости.

Образуем векторы (рис. 7.6):

$$\vec{P}_1 = \vec{P}_1(x_2 - x_1, y_2 - y_1, z_2 - z_1) \text{ и } \vec{P}_2 = \vec{P}_2(x_3 - x_1, y_3 - y_1, z_3 - z_1).$$

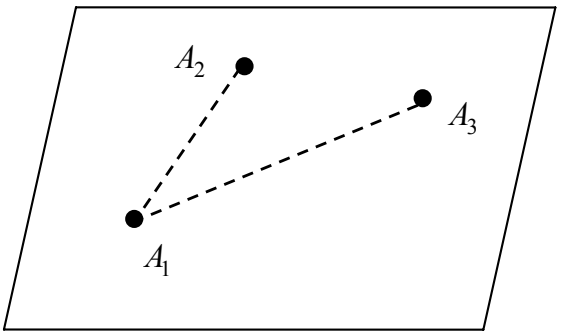


Рис. 7.5

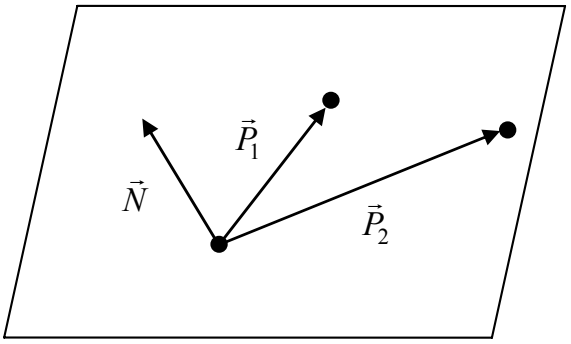


Рис. 7.6

Тогда вектор нормали $\vec{N}$ к плоскости можно вычислить как векторное произведение векторов $\vec{P}_1$ и $\vec{P}_2$:

$$\vec{N} = \vec{P}_1 \times \vec{P}_2 = \begin{vmatrix} \vec{i} & \vec{j} & \vec{k} \\ x_2 - x_1 & y_2 - y_1 & z_2 - z_1 \\ x_3 - x_1 & y_3 - y_1 & z_3 - z_1 \end{vmatrix}. \qquad (7.44)$$

Выражение (7.44) используется для вычисления координат вектора нормали в случае описания поверхности векторно-полигональной моделью (см. ниже).

Рассмотрим теперь произвольную поверхность, заданную в явном виде функцией

$$z = f(x, y). \qquad (7.45)$$

При таком задании поверхности вектор нормали имеет вид

$$\vec{N} = \vec{N}(N_x, N_y, N_z) = \vec{N}\left(-\frac{\partial f}{\partial x}, -\frac{\partial f}{\partial y}, 1\right), \qquad (7.46)$$

где $N_x = -\frac{\partial f}{\partial x}$, $N_y = -\frac{\partial f}{\partial y}$, $N_z = 1$ – координаты вектора нормали.

Единичный вектор нормали будет определяться выражением

$$\vec{e}_N = \vec{e}(e_N^x, e_N^y, e_N^z) = \vec{e}_N\left(\frac{N_x}{|\vec{N}|}, \frac{N_y}{|\vec{N}|}, \frac{1}{|\vec{N}|}\right), \qquad (7.47)$$

где $\left|\vec{N}\right| = \sqrt{N_x^2 + N_y^2 + N_z^2} = \sqrt{f_x'^2 + f_y'^2 + 1}$ – модуль вектора нормали; $e_N^x = \frac{N_x}{\left|\vec{N}\right|}$, $e_N^y = \frac{N_y}{\left|\vec{N}\right|}$, $e_N^z = \frac{1}{\left|\vec{N}\right|}$ – координаты вектора единичной нормали.

Пусть теперь поверхность задана неявно уравнением

$$F(x, y, z) = 0. \tag{7.48}$$

При неявном определении поверхности вектор нормали примет вид:

$$\vec{N} = \vec{N}(N_x, N_y, N_z) = \vec{N}\left(\frac{F_x'}{F_z'}, \frac{F_y'}{F_z'}, 1\right) = \vec{N}\left[\frac{1}{F_z'}\left(F_x', F_y', F_z'\right)\right] = \frac{grad\, F}{F_z'}, \tag{7.49}$$

где $N_x = \frac{F_x'}{F_z'}$, $N_y = \frac{F_y'}{F_z'}$, $N_z = 1$ – координаты вектора нормали,

$$grad\, F = \frac{\partial F}{\partial x}\vec{i} + \frac{\partial F}{\partial y}\vec{j} + \frac{\partial F}{\partial z}\vec{k} - \tag{7.50}$$

градиент функции $F(x, y, z)$.

Единичный вектор нормали в этом случае:

$$\vec{e}_N = \vec{e}_N(e_N^x, e_N^y, e_N^z) = \frac{grad\, F}{\left|grad\, F\right|} = \frac{1}{\left|grad\, F\right|}\left(\frac{\partial F}{\partial x}\vec{i} + \frac{\partial F}{\partial y}\vec{j} + \frac{\partial F}{\partial z}\vec{k}\right), \tag{7.51}$$

где

$$\left|grad\, F\right| = \sqrt{F_x'^2 + F_y'^2 + F_z'^2} \; - \tag{7.52}$$

модуль градиента функции F; $e_N^x = \frac{1}{\left|grad\, F\right|}\frac{\partial F}{\partial x}$, $e_N^y = \frac{1}{\left|grad\, F\right|}\frac{\partial F}{\partial y}$, $e_N^z = \frac{1}{\left|grad\, F\right|}\frac{\partial F}{\partial z}$ – координаты вектора единичной нормали.

При выборе вектора нормали необходимо следить за тем, чтобы получить **нормаль нужного направления**, т. е. правильно выбрать **нужную сторону поверхности**. Если для поверхности $z = f(x, y)$ указано, что нормаль к ней составляет с осью Z угол больший, чем $\pi/2$, то вместо (7.46) в качестве вектора нормали необходимо взять вектор

$$\vec{N}' = -\vec{N}(N_x, N_y, N_z) = \vec{N}'\left(\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, -1\right). \tag{7.53}$$

Аналогично для поверхности $F(x,y,z)=0$. В этом случае вместо (7.49) для вычисления вектора нормали следует воспользоваться выражением

$$\vec{N}' = -\vec{N}(N_x, N_y, N_z) =$$
$$= \vec{N}'\left[-\frac{1}{F_z'}\left(F_x', F_y', F_z'\right)\right] = -\frac{grad\, F}{F_z'}. \qquad (7.54)$$

Соответственно выражение для вектора единичной нормали будет иметь вид

$$\vec{e}_N' = -\vec{e}_N(e_N^x, e_N^y, e_N^z) = -\frac{grad\, F}{|grad\, F|}. \qquad (7.55)$$

Пусть теперь поверхность задана параметрически:

$$\begin{cases} x = x(u,v); \\ y = y(u,v); \\ z = z(u,v). \end{cases} \qquad (7.56)$$

Тогда вектор нормали $\vec{N}$ к поверхности имеет вид:

$$\vec{N} = \vec{N}\left(\left|\frac{D(y,z)}{D(u,v)}\right|, \left|\frac{D(z,x)}{D(u,v)}\right|, \left|\frac{D(x,y)}{D(u,v)}\right|\right), \qquad (7.57)$$

где

$$\left|\frac{D(y,z)}{D(u,v)}\right| = \begin{vmatrix} \frac{\partial y}{\partial u} & \frac{\partial y}{\partial v} \\ \frac{\partial z}{\partial u} & \frac{\partial z}{\partial v} \end{vmatrix}; \left|\frac{D(z,x)}{D(u,v)}\right| = \begin{vmatrix} \frac{\partial z}{\partial u} & \frac{\partial z}{\partial v} \\ \frac{\partial x}{\partial u} & \frac{\partial x}{\partial v} \end{vmatrix}; \left|\frac{D(x,y)}{D(u,v)}\right| = \begin{vmatrix} \frac{\partial x}{\partial u} & \frac{\partial x}{\partial v} \\ \frac{\partial y}{\partial u} & \frac{\partial y}{\partial v} \end{vmatrix}. \qquad (7.58)$$

7.7. Полярная система координат

Приведем в качестве справки соотношения, связывающие координаты точки P, в декартовой и полярной системах координат (рис. 7.7):

$$P = P(r, \varphi);$$
$$\begin{cases} x = r\cos\varphi, \\ y = r\sin\varphi. \end{cases} \qquad (7.59)$$

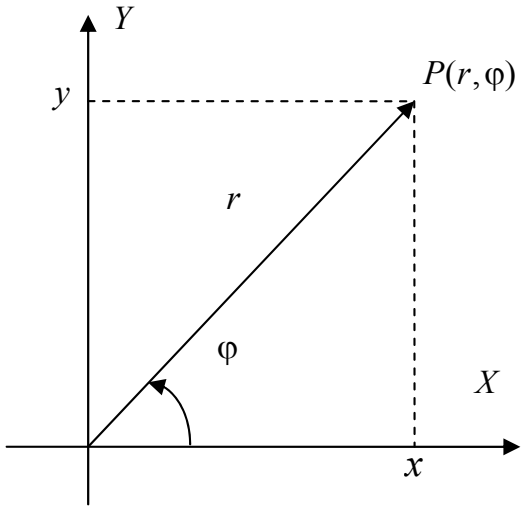


Рис. 7.7

7.8. Сферическая система координат

Ниже приведены соотношения, связывающие координаты точки P, в декартовой и сферической системе координат (рис. 7.8):

$$P = P\left(r, \varphi, \theta\right);$$
$$\begin{cases} x = r\sin\theta\cos\varphi, \\ y = r\sin\theta\sin\varphi, \\ z = r\cos\theta. \end{cases} \qquad (7.60)$$

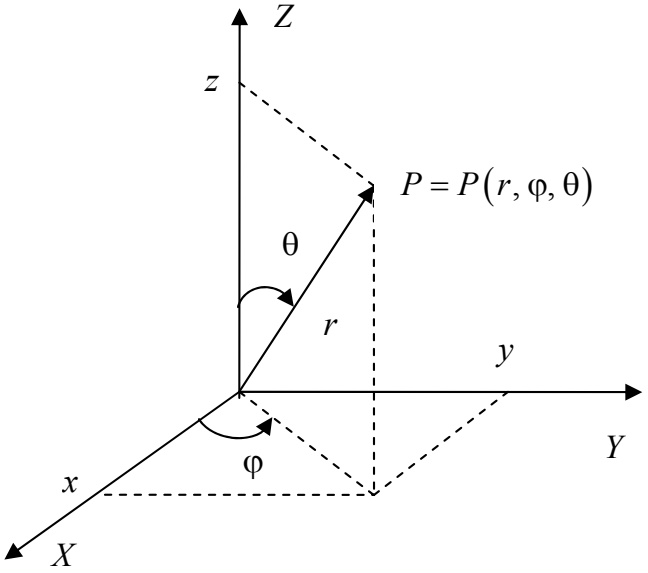


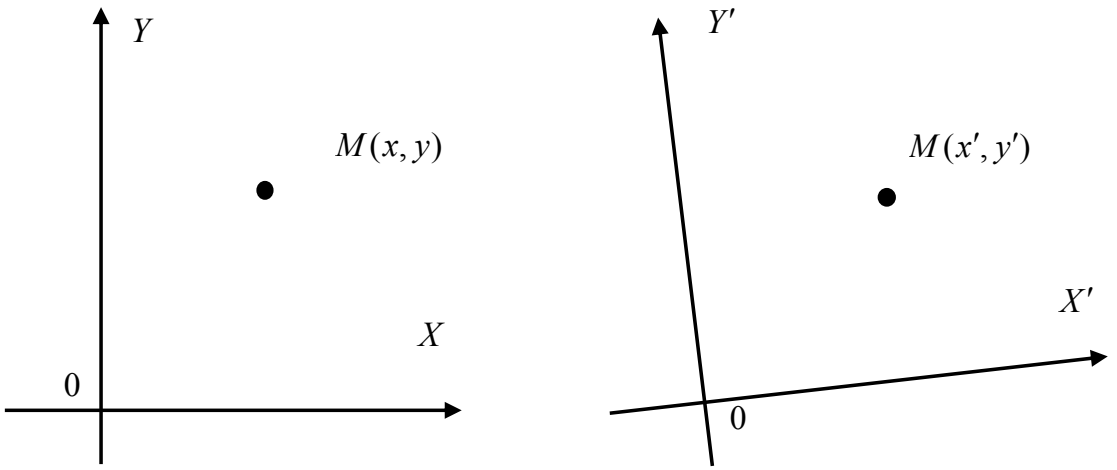
Рис. 7.8

Глава 8

АФФИННЫЕ ПРЕОБРАЗОВАНИЯ НА ПЛОСКОСТИ

8.1. Аффинные преобразования системы координат

Пусть на плоскости задана система координат (СК) $X0Y$ и точка $M(x, y)$, принадлежащая некоторому объекту. Система координат $X0Y$ трансформируется в СК $X'0Y'$ путем ряда последовательных смещений и поворотов относительно своего исходного состояния. При этом точка M (объект) остается **неподвижной**. Необходимо определить координаты точки $M(x', y')$ (объекта) в системе координат $X'0Y'$ (рис. 8.1, *а*, *б*).



а *б*

Рис. 8.1

В общем случае преобразование координат мочки M при переходе от системы координат $X0Y$ к системе координат $X'0Y'$ определяется системой линейных уравнений:

$$\begin{cases} x' = a_{11}x + a_{12}y + a_{13}, \\ y' = a_{21}x + a_{22}y + a_{23}, \end{cases} \tag{8.1}$$

где $\begin{vmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{vmatrix} \neq 0$.

Выражение (8.1) представляет собой аффинное преобразование координат при переходе от системы координат $X0Y$ к системе координат $X'0Y'$.

Обратный переход от СК $X'0Y'$ к СК $X0Y$ определяется как

$$\begin{cases} x = a'_{11}x' + a'_{12}y' + a'_{13}, \\ y = a'_{21}x' + a'_{22}y' + a'_{23}. \end{cases} \tag{8.2}$$

Аффинное преобразование (8.1) удобно представить в матричном виде:

$$\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \begin{pmatrix} x' \\ y' \end{pmatrix} + \begin{pmatrix} a_{13} \\ a_{23} \end{pmatrix}. \tag{8.3}$$

В компьютерной графике принято использовать ***однородные*** координаты, которые вводятся следующим образом. Точке $M(x, y)$ ставится в соответствие точка $M(x, y, 1)$, а точке $M(x', y')$ – точка $M(x', y', 1)$.

Тогда переход от системы координат $X0Y$ к системе координат $X'0Y'$ в матричном виде можно записать как

$$\begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} = \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} \tag{8.4}$$

или

$$X' = A\,X, \tag{8.5}$$

где $X' = (x', \quad y', \quad 1)^T$, $\quad X = (x, \quad y, \quad 1)^T$,

$$A = \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ 0 & 0 & 1 \end{pmatrix}. \tag{8.6}$$

Обратное преобразование

$$X = A^{-1}\,X' = A'\,X', \tag{8.7}$$

где

$$A' = A^{-1} = \begin{pmatrix} a'_{11} & a'_{12} & a'_{13} \\ a'_{21} & a'_{22} & a'_{23} \\ 0 & 0 & 1 \end{pmatrix}. \tag{8.8}$$

Рассмотрим теперь различные виды аффинных преобразований на плоскости и соответствующие им матрицы A и A'.

Параллельный сдвиг системы координат (рис. 8.2):

$$\begin{cases} x' = x - \Delta x, \\ y' = y - \Delta y; \end{cases} \tag{8.9}$$

$$\begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & -\Delta x \\ 0 & 1 & -\Delta y \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} \tag{8.10}$$

или

$$X' = A\,X = T^s X, \tag{8.11}$$

где

$$A = T^s = T^s(\Delta x, \Delta y) = \begin{pmatrix} 1 & 0 & -\Delta x \\ 0 & 1 & -\Delta y \\ 0 & 0 & 1 \end{pmatrix}. \tag{8.12}$$

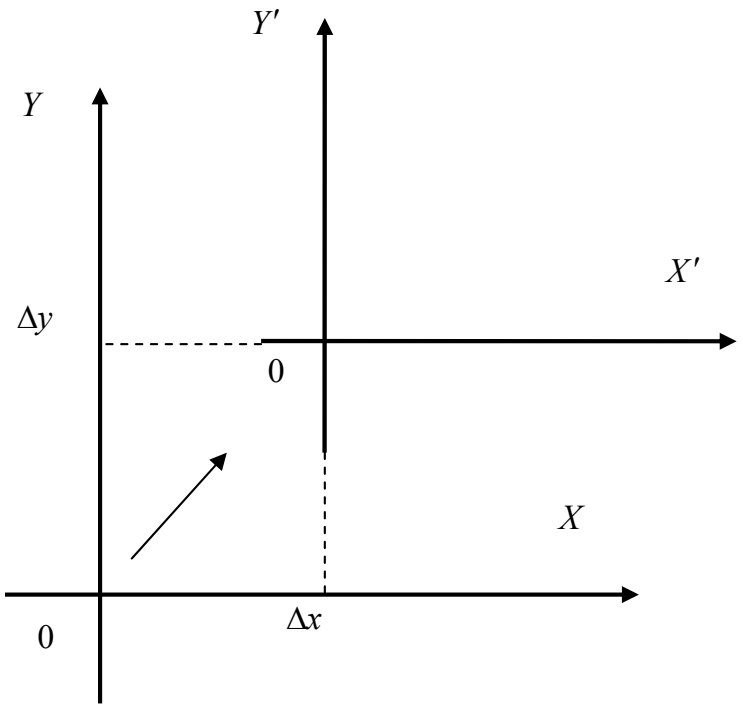


Рис. 8.2

Здесь и далее индекс «s» означает, что преобразованию подвергается система координат.

Обратное преобразование. Обратное преобразование соответствует перемещению системы координат в противоположном направлении. Соответствующие выражения для преобразования координат могут быть получены из (8.9)–(8.12) путем соответствующей замены в них $(x', y') \to (x, y)$ и $(\Delta x, \Delta y) \to (-\Delta x, -\Delta y,)$.

Матрица преобразования будет иметь вид:

$$A' = \left(T^s\right)^{-1} = \left[T^s\left(\Delta x, \Delta y\right)\right]^{-1} = \begin{pmatrix} 1 & 0 & \Delta x \\ 0 & 1 & \Delta y \\ 0 & 0 & 1 \end{pmatrix} = T^s\left(-\Delta x, -\Delta y\right). \quad (8.13)$$

Растяжение – сжатие системы координат (рис. 8.3):

$$\begin{cases} x' = x / k_x, \\ y' = 1 / k_y; \end{cases} \quad (8.14)$$

$$\begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} = \begin{pmatrix} 1/k_x & 0 & 0 \\ 0 & 1/k_y & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} \quad (8.15)$$

или

$$X' = AX = K^s X\,, \quad (8.16)$$

где

$$A = K^s = K^s\left(k_x, k_y\right) = \begin{pmatrix} 1/k_x & 0 & 0 \\ 0 & 1/k_y & 0 \\ 0 & 0 & 1 \end{pmatrix}. \quad (8.17)$$

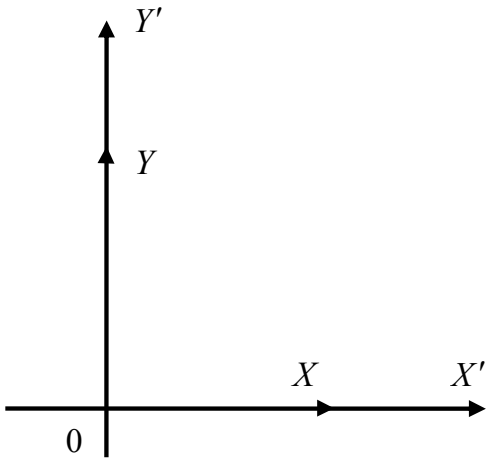


Рис. 8.3

Обратное преобразование. Соответствующие выражения для преобразования координат могут быть получены из (8.14)–(8.17) путем соответствующей замены в них $(x', y') \to (x, y)$ и $(k_x, k_y) \to (1/k_x, 1/k_y)$.

Приведем выражение для матрицы преобразования:

$$A' = \left(K^s\right)^{-1} = \left[K^s\left(k_x, k_y\right)\right]^{-1} = \begin{pmatrix} k_x & 0 & 0 \\ 0 & k_y & 0 \\ 0 & 0 & 1 \end{pmatrix} = K^s\left(1/k_x, 1/k_y\right). \quad (8.18)$$

Поворот системы координат на угол φ показан на рис. 8.4 (*а*, *б*).

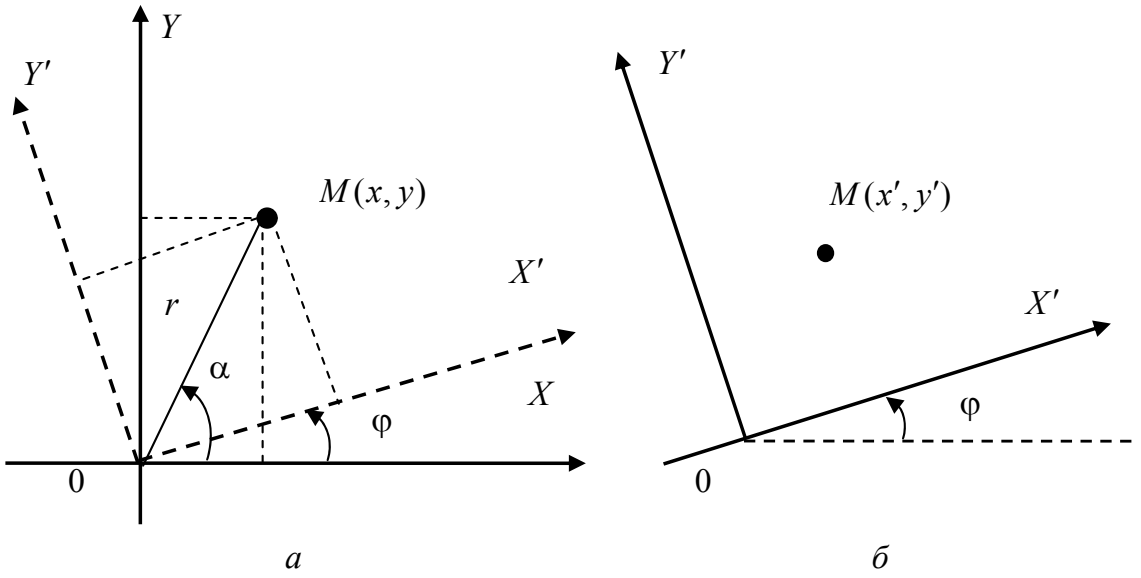


Рис. 8.4

Координаты точки M в системе координат $X0Y$:

$$\begin{cases} x = r\cos\alpha, \\ y = r\sin\alpha. \end{cases} \quad (8.19)$$

В системе координат $X'0Y'$:

$$\begin{cases} x' = r\cos(\alpha - \varphi) = r\cos\alpha\cos\varphi + r\sin\alpha\sin\varphi, \\ y' = r\sin(\alpha - \varphi) = r\sin\alpha\cos\varphi - r\cos\alpha\sin\varphi. \end{cases}$$

С учетом (8.19) получаем:

$$\begin{cases} x' = x\cos\varphi + y\sin\varphi, \\ y' = -x\sin\varphi + y\cos\varphi \end{cases} \quad (8.20)$$

или

$$\begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} = \begin{pmatrix} \cos\varphi & \sin\varphi & 0 \\ -\sin\varphi & \cos\varphi & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}, \tag{8.21}$$

$$X' = AX = R^s X, \tag{8.22}$$

где

$$A = R^s = R^s(\varphi) = \begin{pmatrix} \cos\varphi & \sin\varphi & 0 \\ -\sin\varphi & \cos\varphi & 0 \\ 0 & 0 & 1 \end{pmatrix}. \tag{8.23}$$

Обратное преобразование. Обратное преобразование соответствует повороту системы координат $X'0Y'$ на угол $-\varphi$. Соответствующие выражения для преобразования координат могут быть получены из (8.19)–(8.23) путем замены в них $(x', y') \to (x, y)$ и $\varphi \to -\varphi$.

Матрица преобразования будет иметь вид:

$$A' = \left(R^s\right)^{-1} = \left[R^s(\varphi)\right]^{-1} = \begin{pmatrix} \cos\varphi & -\sin\varphi & 0 \\ \sin\varphi & \cos\varphi & 0 \\ 0 & 0 & 1 \end{pmatrix} = R^s(-\varphi). \tag{8.24}$$

8.2. Аффинные преобразования объектов на плоскости

Под преобразованием объектов будем понимать изменение координат точек, принадлежащих этому объекту при изменении его положения в некоторой системе координат.

Пусть в системе координат $X0Y$ некоторая точка $M(x, y)$ перемещается из положения $M(x, y)$ в положение $M(x', y')$ (рис. 8.5, *а*, *б*) или обратно из положения $M(x', y')$ в положение $M(x, y)$.

Тогда старые координаты точки (x, y) и новые (x', y') связаны соотношениями (8.1)–(8.8). Подчеркнем лишь, что в данном случае выражения (8.1)–(8.8) описывают взаимосвязь между старыми и новыми координатами точки M при изменении ее положения **в одной и той же** системе координат.

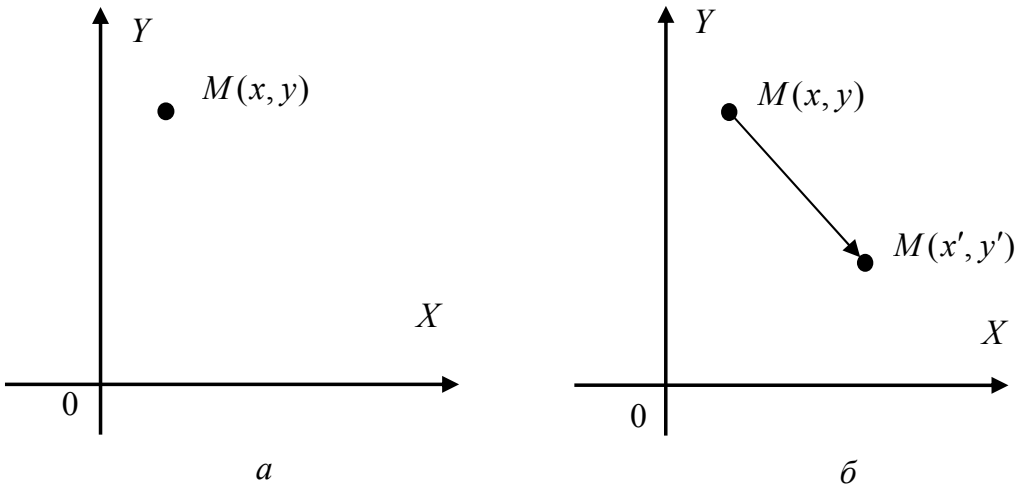


Рис. 8.5

Рассмотрим частные случаи аффинных преобразований объектов.

Сдвиг объекта показан на рис. 8.6.

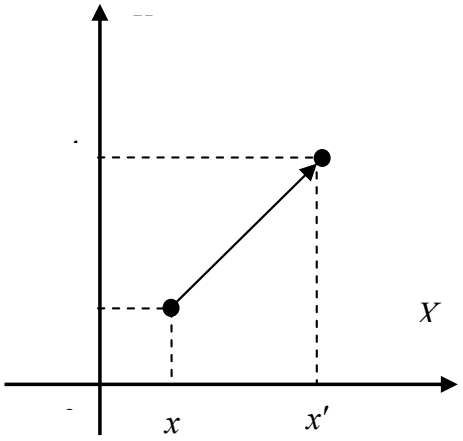


Рис. 8.6

Из рис. 8.6 следует:

$$\begin{cases} x' = x + \Delta x, \\ y' = y + \Delta y; \end{cases} \tag{8.25}$$

$$\begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & \Delta x \\ 0 & 1 & \Delta y \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} \tag{8.26}$$

или

$$X' = A\,X = T^o X\,, \tag{8.27}$$

где

$$A = T^o = T^o(\Delta x, \Delta y) = \begin{pmatrix} 1 & 0 & \Delta x \\ 0 & 1 & \Delta y \\ 0 & 0 & 1 \end{pmatrix}. \qquad (8.28)$$

Здесь и далее индекс «o» означает, что преобразованию подвергаются координаты объекта.

Обратное преобразование. Обратное преобразование соответствует перемещению объекта в противоположном направлении. Соответствующие выражения для преобразования координат могут быть получены из (8.25)–(8.28) путем замены в них $(x', y') \to (x, y)$ и $(\Delta x, \Delta y) \to (-\Delta x, -\Delta y)$.

Матрица преобразования будет иметь вид:

$$A' = \left(T^o\right)^{-1} = \left[T^o(\Delta x, \Delta y)\right]^{-1} = \begin{pmatrix} 1 & 0 & -\Delta x \\ 0 & 1 & -\Delta y \\ 0 & 0 & 1 \end{pmatrix} = T^o(-\Delta x, -\Delta y). \qquad (8.29)$$

Растяжение – сжатие объекта:

$$\begin{cases} x' = k_x x, \\ y' = k_x y; \end{cases} \qquad (8.30)$$

$$\begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} = \begin{pmatrix} k_x & 0 & 0 \\ 0 & k_y & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} \qquad (8.31)$$

или

$$X' = AX = K^o X, \qquad (8.32)$$

где

$$A = K^o = K^o(k_x, k_y) = \begin{pmatrix} k_x & 0 & 0 \\ 0 & k_y & 0 \\ 0 & 0 & 1 \end{pmatrix}. \qquad (8.33)$$

Обратное преобразование. Соответствующие выражения для преобразования координат могут быть получены из (8.30)–(8.33) путем замены в них $(x', y') \to (x, y)$ и $(k_x, k_y) \to (1/k_x, 1/k_y)$.

Приведем выражение для матрицы преобразования:

$$A' = \left(K^o\right)^{-1} = \left[K^o\left(k_x, k_y\right)\right]^{-1} = \begin{pmatrix} 1/k_x & 0 & 0 \\ 0 & 1/k_y & 0 \\ 0 & 0 & 1 \end{pmatrix} = K^o\left(1/k_x, 1/k_y\right). \quad (8.34)$$

Поворот объекта вокруг центра координат показан на рис. 8.7.

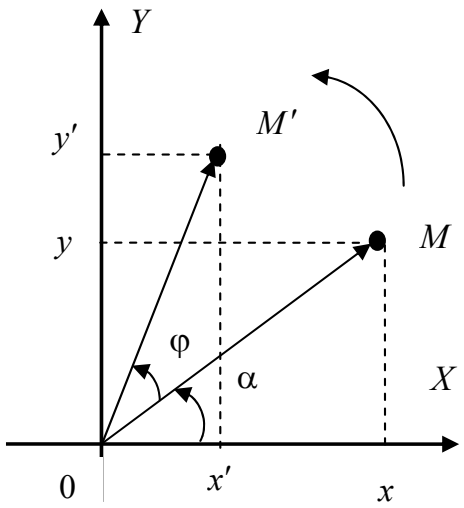


Рис. 8.7

Координаты точки M в системе координат $X0Y$ (рис. 8.7):

$$\begin{cases} x = r\cos\alpha, \\ y = r\sin\alpha. \end{cases} \quad (8.35)$$

Координаты точки M' в системе координат $X0Y$:

$$\begin{cases} x' = r\cos(\alpha+\varphi) = r\cos\alpha\cos\varphi - r\sin\alpha\sin\varphi, \\ y' = r\sin(\alpha+\varphi) = r\sin\alpha\cos\varphi + r\cos\alpha\sin\varphi. \end{cases}$$

С учетом (8.35) получаем:

$$\begin{cases} x' = x\cos\varphi - y\sin\varphi, \\ y' = x\sin\varphi + y\cos\varphi \end{cases} \quad (8.36)$$

или

$$\begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} = \begin{pmatrix} \cos\varphi & -\sin\varphi & 0 \\ \sin\varphi & \cos\varphi & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}; \quad (8.37)$$

$$X' = AX = R^o X, \tag{8.38}$$

где

$$A = R^o = R^o(\varphi) = \begin{pmatrix} \cos\varphi & -\sin\varphi & 0 \\ \sin\varphi & \cos\varphi & 0 \\ 0 & 0 & 1 \end{pmatrix}. \tag{8.39}$$

Обратное преобразование. Обратное преобразование соответствует повороту объекта $X'0Y'$ на угол $-\varphi$. Соответствующие выражения для преобразования координат могут быть получены из (8.35)–(8.39) путем замены в них $(x', y') \to (x, y)$ и $\varphi \to -\varphi$.

Приведем выражение для матрицы преобразования:

$$A' = \left(R^o\right)^{-1} = \left[R^o(\varphi)\right]^{-1} = \begin{pmatrix} \cos\varphi & \sin\varphi & 0 \\ -\sin\varphi & \cos\varphi & 0 \\ 0 & 0 & 1 \end{pmatrix} = R^o(-\varphi). \tag{8.40}$$

Связь преобразований объектов с преобразованиями координат. Преобразование объектов и преобразование систем координат тесно связаны между собой. Движение объектов можно рассматривать как движение в обратном направлении соответствующей системы координат. Такая относительность движения дает дополнительные возможности для моделирования и визуализации различных объектов.

Запишем соответствующие соотношения, основываясь на сравнении выражений (8.12) и (8.28) для сдвига:

$$T^o(\Delta x, \Delta y) = T^s(-\Delta x, -\Delta y), \tag{8.41}$$

выражений (8.18) и (8.33) для растяжения – сжатия:

$$K^o(k_x, k_y) = K^s(1/k_x, 1/k_y), \tag{8.42}$$

выражений (8.23) и (8.39) для поворота:

$$R^o(\varphi) = R^s(-\varphi). \tag{8.43}$$

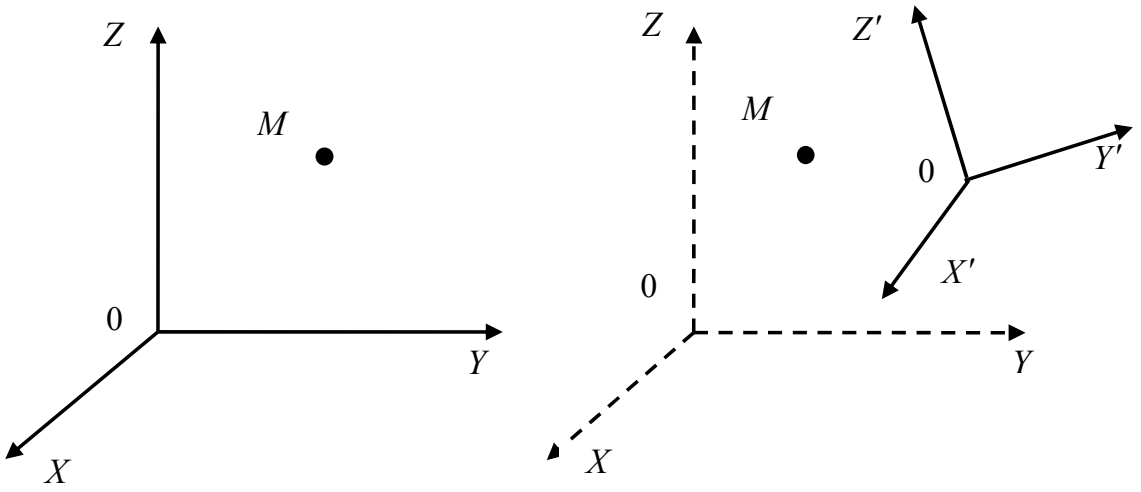
В приложении 1 приведен пример приложения Windows, где используются аффинные преобразования на плоскости.

Глава 9

АФФИННЫЕ ПРЕОБРАЗОВАНИЯ В ПРОСТРАНСТВЕ

9.1. Аффинные преобразования системы координат

Пусть в пространстве задана система координат (СК) XYZ и точка $M(x, y, z)$, принадлежащая некоторому объекту. Система координат XYZ трансформируется в СК $X'Y'Z'$ путем ряда последовательных смещений и поворотов относительно своего исходного состояния. При этом точка M (объект) остается **неподвижной**. Необходимо определить координаты точки $M(x', y', z')$ (объекта) в системе координат $X'Y'Z'$ (рис. 9.1 *а*, *б*).



а *б*

Рис. 9.1

В общем случае преобразование координат точки M при переходе от системы координат XYZ к системе координат $X'Y'Z'$ определяется системой линейных уравнений:

$$\begin{cases} x' = a_{11}x + a_{12}y + a_{13}z + a_{14}, \\ y' = a_{21}x + a_{22}y + a_{23}z + a_{24}, \\ z' = a_{31}x + a_{32}y + a_{33}z + a_{34}, \end{cases} \qquad (9.1)$$

где $\begin{vmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{vmatrix} \neq 0$.

Выражение (9.1) представляет собой аффинное преобразование координат при переходе от системы координат XYZ к системе координат $X'Y'Z'$.

Обратный переход от СК $X'Y'Z'$ к СК XYZ определяется как

$$\begin{cases} x = a'_{11}x' + a'_{12}y' + a'_{13} + a'_{14}, \\ y = a'_{21}x' + a'_{22}y' + a'_{23} + a'_{24}, \\ z' = a'_{31}x' + a'_{32}y' + a'_{33} + a'_{34}. \end{cases} \tag{9.2}$$

Аффинные преобразования (9.1) удобно представить в матричном виде:

$$\begin{pmatrix} x' \\ y' \\ z' \end{pmatrix} = \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix} + \begin{pmatrix} a_{14} \\ a_{24} \\ a_{34} \end{pmatrix}. \tag{9.3}$$

Как и ранее, введем в рассмотрение однородные координаты, когда точке $M(x, y, z)$ ставится в соответствие точка $M(x, y, z, 1)$, а точке $M(x', y', z')$ – точка $M(x', y', z', 1)$.

Тогда переход от системы координат XYZ к системе координат $X'Y'Z'$ в матричном виде можно записать как

$$\begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix} = \begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} \tag{9.4}$$

или

$$X' = A\,X, \tag{9.5}$$

где

$$X' = (x', \quad y', \quad z', \quad 1)^T, \qquad X = (x, \quad y, \quad z, \quad 1)^T,$$

$$A = \begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ 0 & 0 & 0 & 1 \end{pmatrix}. \tag{9.6}$$

Обратное преобразование имеет вид

$$X = A^{-1} X' = A' X', \tag{9.7}$$

где

$$A' = A^{-1} = \begin{pmatrix} a'_{11} & a'_{12} & a'_{13} & a'_{14} \\ a'_{21} & a'_{22} & a'_{23} & a'_{24} \\ a'_{31} & a'_{32} & a'_{33} & a'_{34} \\ 0 & 0 & 0 & 1 \end{pmatrix}. \tag{9.8}$$

Рассмотрим теперь различные виды аффинных преобразований в пространстве и соответствующие им матрицы A и A'.

Параллельный сдвиг системы координат (рис. 9.2):

$$\begin{cases} x' = x - \Delta x, \\ y' = y - \Delta y, \\ z' = z - \Delta z. \end{cases} \tag{9.9}$$

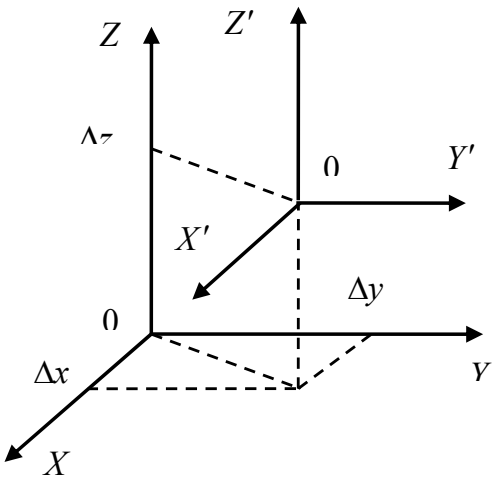


Рис. 9.2

$$\begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & -\Delta x \\ 0 & 1 & 0 & -\Delta y \\ 0 & 0 & 1 & -\Delta z \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} \tag{9.10}$$

или

$$X' = A X = T^s X, \tag{9.11}$$

где

$$A = T^s = T^s(\Delta x, \Delta y, \Delta z) = \begin{pmatrix} 1 & 0 & 0 & -\Delta x \\ 0 & 1 & 0 & -\Delta y \\ 0 & 0 & 1 & -\Delta z \\ 0 & 0 & 0 & 1 \end{pmatrix}. \qquad (9.12)$$

Здесь и далее индекс «s» означает, что преобразованию подвергается система координат.

Обратное преобразование. Обратное преобразование соответствует перемещению системы в противоположном направлении. Соответствующие выражения для преобразования координат могут быть получены из (9.9)–(9.12) путем замены в них $(x', y', z') \to (x, y, z)$ и $(\Delta x, \Delta y, \Delta z) \to (-\Delta x, -\Delta y, -\Delta z)$.

Матрица преобразования будет иметь вид

$$A' = \left(T^s\right)^{-1} = \left[T^s(\Delta x, \Delta y, \Delta z)\right]^{-1} = \begin{pmatrix} 1 & 0 & 0 & \Delta x \\ 0 & 1 & 0 & \Delta y \\ 0 & 0 & 1 & \Delta z \\ 0 & 0 & 0 & 1 \end{pmatrix} = T^s(-\Delta x, -\Delta y, -\Delta z). \qquad (9.13)$$

Растяжение – сжатие системы координат показано на рис. 9.3.

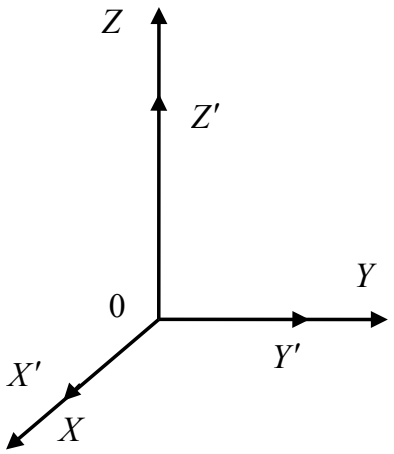


Рис. 9.3

$$\begin{cases} x' = x/k_x, \\ y' = y/k_y, \\ z' = z/k_z; \end{cases} \qquad (9.14)$$

$$\begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix} = \begin{pmatrix} 1/k_x & 0 & 0 & 0 \\ 0 & 1/k_y & 0 & 0 \\ 0 & 0 & 1/k_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} \tag{9.15}$$

или

$$X' = AX = K^s X\,, \tag{9.16}$$

где

$$A = K^s = K^s\left(k_x, k_y, k_z\right) = \begin{pmatrix} 1/k_x & 0 & 0 & 0 \\ 0 & 1/k_y & 0 & 0 \\ 0 & 0 & 1/k_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}. \tag{9.17}$$

Обратное преобразование. Соответствующие выражения для преобразования координат могут быть получены из (9.14)–(9.17) путем замены в них $(x', y', z') \to (x, y, z)$ и $(k_x, k_y, k_z) \to (1/k_x, 1/k_y, 1/k_z)$.

Приведем выражение для матрицы преобразования:

$$A' = \left(K^s\right)^{-1} = \left[K^s\left(k_x, k_y, k_z\right)\right]^{-1} = \begin{pmatrix} k_x & 0 & 0 & 0 \\ 0 & k_y & 0 & 0 \\ 0 & 0 & k_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} = K^s\left(1/k_x, 1/k_y, 1/k_z\right). \tag{9.18}$$

Поворот системы координат вокруг оси *X* на угол φ показан на рис. 9.4.

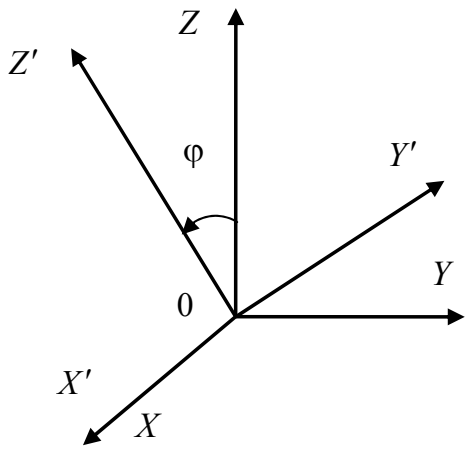


Рис. 9.4

$$\begin{cases} x' = x, \\ y' = y\cos\varphi + z\sin\varphi, \\ z' = -y\sin\varphi + z\cos\varphi; \end{cases} \tag{9.19}$$

$$\begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\varphi & \sin\varphi & 0 \\ 0 & -\sin\varphi & \cos\varphi & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} \tag{9.20}$$

или

$$X' = AX = R_X^s X\,, \tag{9.21}$$

где

$$A = R_X^s = R_X^s(\varphi) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\varphi & \sin\varphi & 0 \\ 0 & -\sin\varphi & \cos\varphi & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}. \tag{9.22}$$

Обратное преобразование. Обратное преобразование соответствует повороту системы координат вокруг оси X в противоположном направлении.

Соответствующие выражения для преобразования координат могут быть получены из (9.19)–(9.22) путем замены в них $(x', y', z') \to (x, y, z)$ и $\varphi \to -\varphi$.

Приведем выражение для матрицы преобразования:

$$A' = \left(R_X^s\right)^{-1} = \left[R_X^s(\varphi)\right]^{-1} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\varphi & -\sin\varphi & 0 \\ 0 & \sin\varphi & \cos\varphi & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} = R_X^s(-\varphi). \tag{9.23}$$

Поворот системы координат вокруг оси *Y* на угол φ показан на рис. 9.5:

$$\begin{cases} x' = x\cos\varphi - z\sin\varphi, \\ y' = y, \\ z' = -x\sin\varphi + z\cos\varphi. \end{cases} \tag{9.24}$$

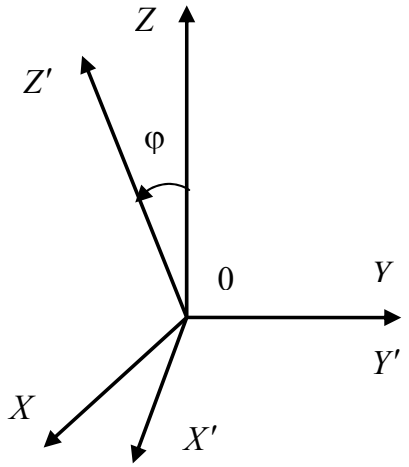


Рис. 9.5

$$\begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix} = \begin{pmatrix} \cos\varphi & 0 & -\sin\varphi & 0 \\ 0 & 1 & 0 & 0 \\ \sin\varphi & 0 & \cos\varphi & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} \tag{9.25}$$

или

$$X' = AX = R_Y^s X\,, \tag{9.26}$$

где

$$A = R_Y^s = R_Y^s(\varphi) = \begin{pmatrix} \cos\varphi & 0 & -\sin\varphi & 0 \\ 0 & 1 & 0 & 0 \\ \sin\varphi & 0 & \cos\varphi & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}. \tag{9.27}$$

Обратное преобразование. Обратное преобразование соответствует повороту системы координат вокруг оси Y в противоположном направлении. Соответствующие выражения для преобразования координат могут быть получены из (9.24)–(9.27) путем замены в них $(x', y', z') \to (x, y, z)$ и $\varphi \to -\varphi$.

Приведем выражение для матрицы преобразования:

$$A' = \left(R_Y^s\right)^{-1} = \left[R_Y^s(\varphi)\right]^{-1} = \begin{pmatrix} \cos\varphi & 0 & \sin\varphi & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\varphi & 0 & \cos\varphi & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} = R_Y^s(-\varphi)\,. \tag{9.28}$$

Поворот системы координат вокруг оси Z на угол показан на рис. 9.6.

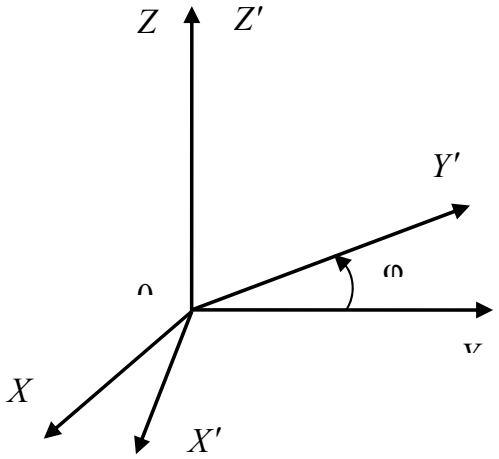


Рис. 9.6

$$\begin{cases} x' = x\cos\varphi + y\sin\varphi, \\ y' = -x\sin\varphi + y\cos\varphi, \\ z' = z; \end{cases} \tag{9.29}$$

$$\begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix} = \begin{pmatrix} \cos\varphi & \sin\varphi & 0 & 0 \\ -\sin\varphi & \cos\varphi & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} \tag{9.30}$$

или

$$X' = AX = R_Z^s X\,, \tag{9.31}$$

где

$$A = R_Z^s = R_Z^s(\varphi) = \begin{pmatrix} \cos\varphi & \sin\varphi & 0 & 0 \\ -\sin\varphi & \cos\varphi & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}. \tag{9.32}$$

Обратное преобразование. Обратное преобразование соответствует повороту системы координат вокруг оси Z в противоположном направлении. Соответствующие выражения для преобразования координат могут быть получены из (9.29)–(9.32) путем замены в них $(x', y', z') \to (x, y, z)$ и $\varphi \to -\varphi$.

Матрица преобразования для этого случая будет иметь вид:

$$A' = \left(R_Z^s\right)^{-1} = \left[R_Z(\varphi)\right]^{-1} = \begin{pmatrix} \cos\varphi & -\sin\varphi & 0 & 0 \\ \sin\varphi & \cos\varphi & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} = R_Z^s(-\varphi). \qquad (9.33)$$

9.2. Аффинные преобразования координат объектов

Как и ранее, под преобразованием объекта будем понимать изменение координат точек, принадлежащих этому объекту при изменении его положения в некоторой системе координат.

Пусть в пространстве задана система координат XYZ и точка $M(x, y, z)$, принадлежащая некоторому объекту. Объект перемещается из одной точки пространства в другую путем ряда последовательных смещений и поворотов относительно своего исходного положения. При этом система координат остается **неподвижной**. Необходимо определить новые координаты точки $M(x', y', z')$ (объекта) в СК XYZ. В общем случае старые координаты точки (x, y, z) и новые (x', y', z) связаны соотношениями (9.1)–(9.8). Подчеркнем лишь, что в данном случае выражения (9.1)–(9.8) описывают взаимосвязь между старыми и новыми координатами точки M при изменении ее положения **в одной и той же** системе координат.

Рассмотрим частные случаи аффинных преобразований координат объектов.

Смещение объекта вдоль координатных осей показано на рис. 9.7.

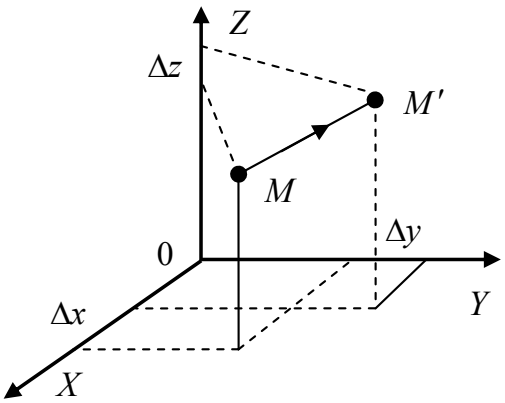


Рис. 9.7

$$\begin{cases} x' = x + \Delta x, \\ y' = y + \Delta y, \\ z' = z + \Delta z; \end{cases} \tag{9.34}$$

$$\begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & \Delta x \\ 0 & 1 & 0 & \Delta y \\ 0 & 0 & 1 & \Delta z \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} \tag{9.35}$$

или

$$X' = A\,X = T^o X\,, \tag{9.36}$$

где

$$A = T^o = T^o\left(\Delta x,\, \Delta y, \Delta z\right) = \begin{pmatrix} 1 & 0 & 0 & \Delta x \\ 0 & 1 & 0 & \Delta y \\ 0 & 0 & 1 & \Delta z \\ 0 & 0 & 0 & 1 \end{pmatrix}. \tag{9.37}$$

Здесь и далее индекс «o» означает, что преобразованию подвергаются координаты объекта.

Обратное преобразование. Обратное преобразование соответствует перемещению объекта в противоположном направлении.

Соответствующие выражения для преобразования координат могут быть получены из (9.34)–(9.37) путем замены в них $(x', y', z') \to (x, y, z)$ и $(\Delta x, \Delta y, \Delta z) \to (-\Delta x, -\Delta y, -\Delta z)$.

Приведем выражение для матрицы преобразования:

$$A' = \left(T^o\right)^{-1} = \left[T^o\left(\Delta x,\, \Delta y, \Delta z\right)\right]^{-1} = \begin{pmatrix} 1 & 0 & 0 & -\Delta x \\ 0 & 1 & 0 & -\Delta y \\ 0 & 0 & 1 & -\Delta z \\ 0 & 0 & 0 & 1 \end{pmatrix} = T^o\left(-\Delta x, -\Delta y, -\Delta z\right). \tag{9.38}$$

Растяжение – сжатие объекта:

$$\begin{cases} x' = k_x x, \\ y' = k_y y, \\ z' = k_z z; \end{cases} \tag{9.39}$$

$$\begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix} = \begin{pmatrix} k_x & 0 & 0 & 0 \\ 0 & k_y & 0 & 0 \\ 0 & 0 & k_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}, \tag{9.40}$$

или

$$X' = A\,X = K^o X\,, \tag{9.41}$$

где

$$A = K^o = K^o\left(k_x,\, k_y, k_z\right) = \begin{pmatrix} k_x & 0 & 0 & 0 \\ 0 & k_y & 0 & 0 \\ 0 & 0 & k_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}. \tag{9.42}$$

Обратное преобразование. Соответствующие выражения для преобразования координат могут быть получены из (9.39)–(9.42) путем замены в них $(x', y', z') \to (x, y, z)$ и $(k_x, k_y, k_z) \to (1/k_x, 1/k_y, 1/k_z)$.

Приведем выражение для матрицы преобразования:

$$A' = \left(K^o\right)^{-1} = \left[K^o\left(k_x, k_y, k_z\right)\right]^{-1} = \begin{pmatrix} 1/k_x & 0 & 0 & -\Delta x \\ 0 & 1/k_y & 0 & -\Delta y \\ 0 & 0 & 1/k_z & -\Delta z \\ 0 & 0 & 0 & 1 \end{pmatrix} = K^o\left(1/k_x, 1/k_y, 1/k_z\right). \tag{9.43}$$

Поворот объекта вокруг оси *X* на угол φ показан на рис. 9.8.

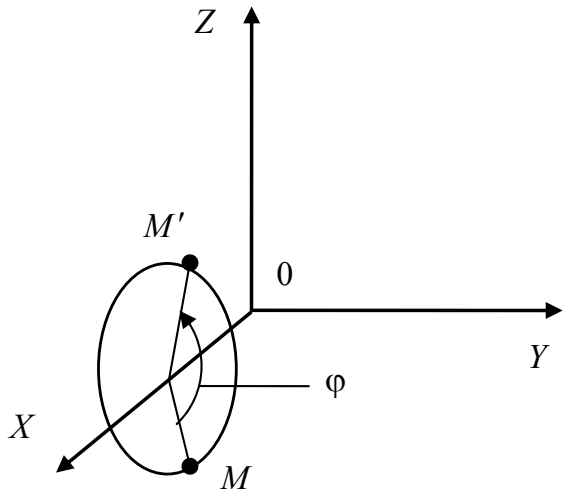


Рис. 9.8

$$\begin{cases} x' = x, \\ y' = y\cos\varphi - z\sin\varphi, \\ z' = y\sin\varphi + z\cos\varphi; \end{cases} \tag{9.44}$$

$$\begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\varphi & -\sin\varphi & 0 \\ 0 & \sin\varphi & \cos\varphi & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}, \tag{9.45}$$

или

$$X' = A\,X = R_X^o X\,, \tag{9.46}$$

где

$$A = R_X^o = R_X^o(\varphi) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\varphi & -\sin\varphi & 0 \\ 0 & \sin\varphi & \cos\varphi & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}. \tag{9.47}$$

Обратное преобразование. Обратное преобразование соответствует повороту объекта вокруг оси X в противоположном направлении.

Соответствующие выражения для преобразования координат могут быть получены из (9.44)–(9.47) путем замены в них $(x', y', z') \to (x, y, z)$ и $\varphi \to -\varphi$.

Приведем выражение для матрицы преобразования:

$$A' = \left(R_X^o\right)^{-1} = \left(R_X^o(\varphi)\right)^{-1} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\varphi & \sin\varphi & 0 \\ 0 & -\sin\varphi & \cos\varphi & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} = R_X^o(-\varphi). \tag{9.48}$$

Поворот объекта вокруг оси *Y* на угол φ показан на рис. 9.9.

$$\begin{cases} x' = x\cos\varphi + z\sin\varphi, \\ y' = y, \\ z' = -x\sin\varphi + z\cos\varphi. \end{cases} \tag{9.49}$$

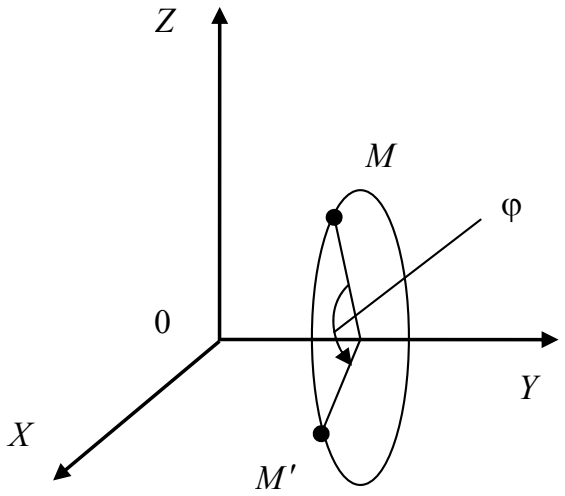


Рис. 9.9

$$\begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix} = \begin{pmatrix} \cos\varphi & 0 & \sin\varphi & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\varphi & 0 & \cos\varphi & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} \tag{9.50}$$

или

$$X' = A\,X = R_Y^o X\,, \tag{9.51}$$

где

$$A = R_Y^o = R_Y^o(\varphi) = \begin{pmatrix} \cos\varphi & 0 & \sin\varphi & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\varphi & 0 & \cos\varphi & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}. \tag{9.52}$$

Обратное преобразование. Соответствующие выражения для преобразования координат могут быть получены из (9.49)–(9.52) путем замены в них $(x', y', z') \to (x, y, z)$ и $\varphi \to -\varphi$.

Приведем выражение для матрицы преобразования:

$$A' = \left(R_Y^o\right)^{-1} = \left(R_Y^o(\varphi)\right)^{-1} = \begin{pmatrix} \cos\varphi & 0 & \sin\varphi & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\varphi & 0 & \cos\varphi & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} = R_Y^o(-\varphi)\,. \tag{9.53}$$

Поворот объекта вокруг оси *Z* на угол φ показан на рис. 9.10.

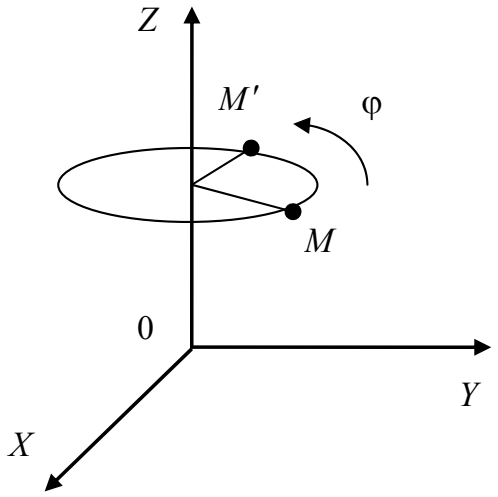


Рис. 9.10

$$\begin{cases} x' = x\cos\varphi - y\sin\varphi, \\ y' = x\sin\varphi + y\cos\varphi, \\ z' = z; \end{cases} \tag{9.54}$$

$$\begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix} = \begin{pmatrix} \cos\varphi & -\sin\varphi & 0 & 0 \\ \sin\varphi & \cos\varphi & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} \tag{9.55}$$

или

$$X' = A\,X = R_Z^o X\,, \tag{9.56}$$

где

$$A = R_Z^o = R_Z^o(\varphi) = \begin{pmatrix} \cos\varphi & -\sin\varphi & 0 & 0 \\ \sin\varphi & \cos\varphi & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}. \tag{9.57}$$

Обратное преобразование. Соответствующие выражения для преобразования координат могут быть получены из (9.54)–(9.57) путем замены в них $(x', y', z') \to (x, y, z)$ и $\varphi \to -\varphi$.

Приведем выражение для матрицы преобразования:

$$A' = \left(R_Z^o\right)^{-1} = \left(R_Z^o(\varphi)\right)^{-1} = \begin{pmatrix} \cos\varphi & \sin\varphi & 0 & 0 \\ -\sin\varphi & \cos\varphi & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} = R_Z^o(-\varphi). \quad (9.58)$$

9.3. Связь преобразований объектов с преобразованиями координат

Как было отмечено при рассмотрении аффинных преобразований на плоскости, движение объектов можно рассматривать как движение в обратном направлении соответствующей системы координат.

Запишем соответствующие соотношения, связывающие матрицы аффинных преобразований для перемещений объектов и системы координат, основываясь на сравнении выражений (9.12) и (9.37) для сдвига:

$$T^o\left(\Delta x, \Delta y, \Delta z\right) = T^s\left(-\Delta x, -\Delta y, \Delta z\right), \quad (9.59)$$

выражений (9.18) и (9.43) для растяжения – сжатия:

$$K^o\left(k_x, k_y, k_z\right) = K^s\left(1/k_x, 1/k_y, 1/k_z\right), \quad (9.60)$$

выражений (9.22), (9.27), (9.32) и (9.47), (9.52), (9.57) для поворотов:

$$R_X^o(\varphi) = R_X^s(-\varphi); \quad (9.61)$$

$$R_Y^o(\varphi) = R_Y^s(-\varphi); \quad (9.62)$$

$$R_Z^o(\varphi) = R_Z^s(-\varphi). \quad (9.63)$$

Глава 10

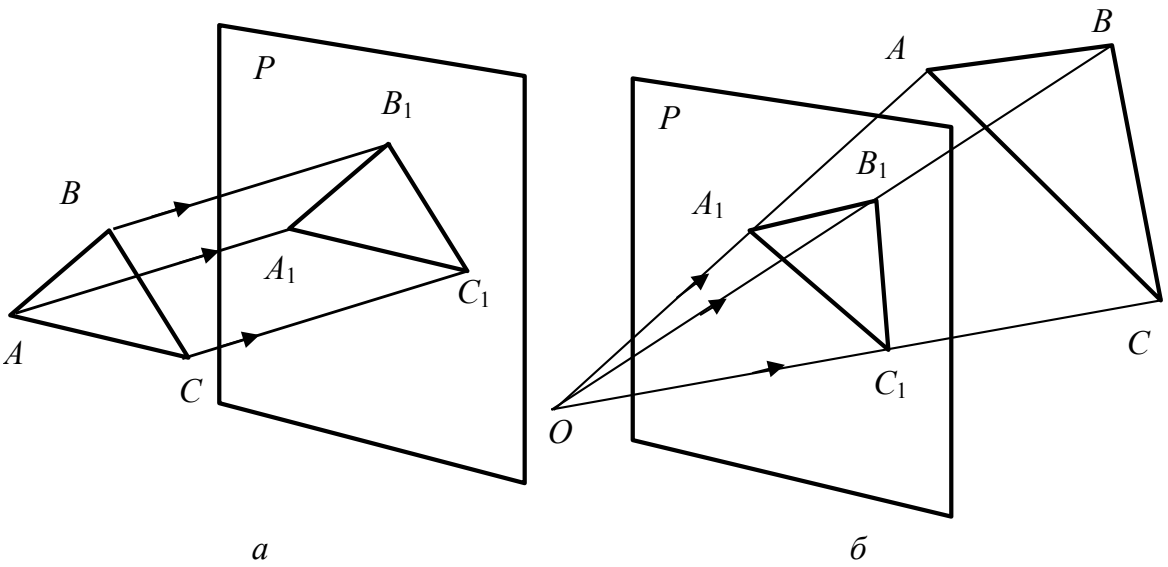
ПРОЕКЦИИ

10.1. Основные типы проекций

В настоящее время наиболее распространены устройства отображения, которые синтезируют изображение на плоскости – экране дисплея или бумаге.

При использовании любых графических устройств обычно применяются проекции. Проекция задает способ отображения объектов на графическом устройстве. Мы будем рассматривать только проекции на плоскость. Плоскость, на которую выполняется проекция объекта, называется **картинной плоскостью**.

В компьютерной графике наиболее распространены **параллельная** (рис. 10.1, *а*) и **центральная** (рис. 10.1, *б*) проекции.



а *б*

Рис. 10.1

Для центральной проекции (также называемой перспективной) лучи проецирования исходят из одной точки, расположенной на конечном расстоянии от объектов и плоскости проецирования. Для параллельной проекции лучи проецирования параллельны. Частным случаем параллельной проекции является **аксонометрическая** проекция. Для аксонометрической проекции все лучи проецирования располагаются под прямым углом к плоскости проецирования.

10.2. Видовая система координат

Для изображения объекта на экране его мировые координаты необходимо преобразовать (пересчитать) в другую систему координат, которая связана с точкой наблюдения. Эта система координат называется видовой системой координат и является левосторонней.

Рассмотрим правую систему мировых координат XYZ (рис. 10.2) и зададим в ней точку наблюдения $E(x, y, z)$, где

$$\begin{cases} x = r\sin\theta\cos\varphi, \\ y = r\sin\theta\sin\varphi, \\ z = r\cos\theta. \end{cases} \qquad (10.1)$$

Система видовых координат $X^E Y^E Z^E$ показана на рис. 10.3.

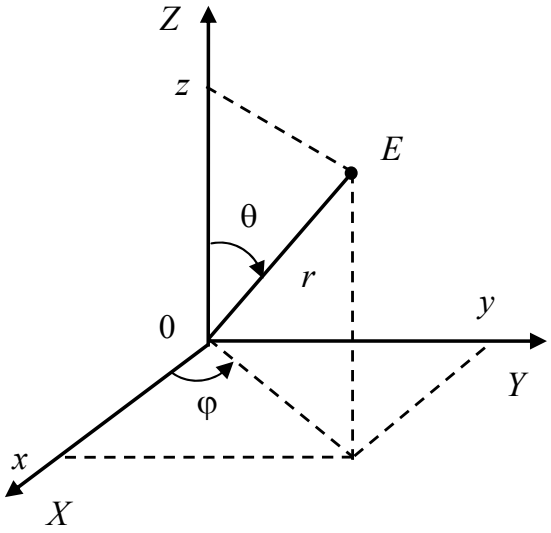


Рис. 10.2

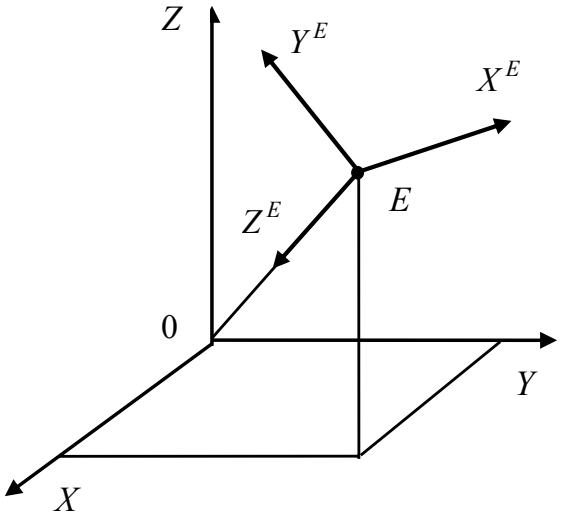


Рис. 10.3

Преобразование мировых координат в видовые можно представить в виде

$$V^E = AV, \qquad (10.2)$$

где $V^E = (x^E, y^E, z^E, 1)^T$ – координаты некоторой точки в видовой системе координат; A – матрица преобразования; $V = (x, y, z, 1)^T$ – координаты этой же точки в мировой системе координат.

Получим выражение для матрицы A, выполняя переход от мировой системы координат к видовой путем последовательности элементарных преобразований [4].

Сместим начало системы координат XYZ (рис. 10.2) в точку E (рис. 10.4), переход $XYZ \to X^1Y^1Z^1$.

Преобразование координат имеет вид

$$V^1 = T^s V\,, \tag{10.3}$$

где

$$T^s = \begin{pmatrix} 1 & 0 & 0 & -x \\ 0 & 1 & 0 & -y \\ 0 & 0 & 1 & -z \\ 0 & 0 & 0 & 1 \end{pmatrix}, \tag{10.4}$$

где $V^1 = \left(x^1, y^1, z^1, 1\right)^T$.

Повернем систему координат $X^1Y^1Z^1$ (рис. 10.4) вокруг оси Z^1 на угол $\frac{\pi}{2} - \varphi$, переход $X^1Y^1Z^1 \to X^2Y^2Z^2$ (рис. 10.5).

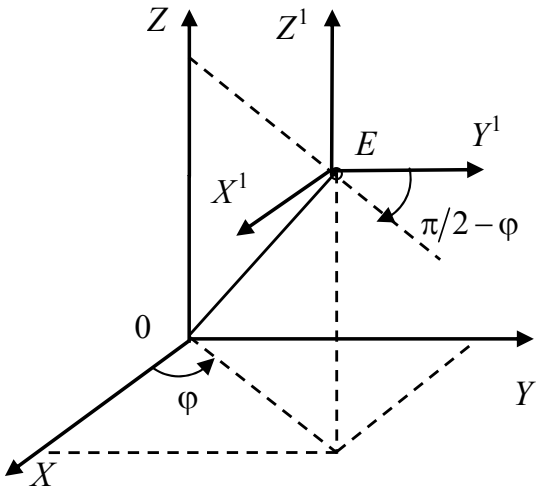


Рис. 10.4

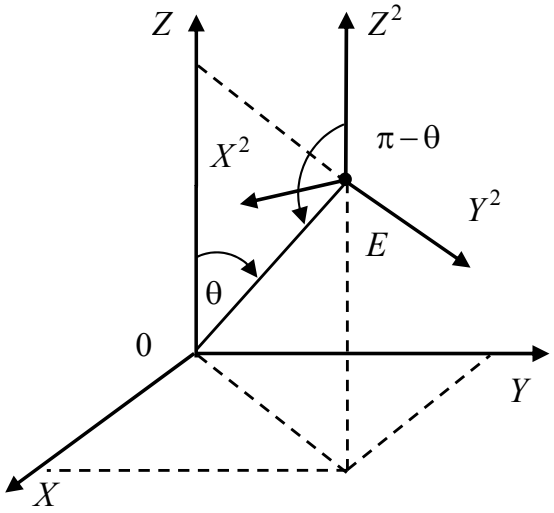


Рис. 10.5

Преобразование координат имеет вид:

$$V^2 = R^s_{Z^1} V^1, \tag{10.5}$$

где $V^2 = \left(x^2, y^2, z^2, 1\right)^T$.

Поскольку поворот выполняется по часовой стрелке (в отрицательном направлении), то в матрице преобразования $R^s_{Z^1}$ (9.33) следует использовать аргумент $-\left(\frac{\pi}{2} - \varphi\right)$. В результате матрица преобразования примет вид:

$$R^s_{Z_1}=\begin{pmatrix}\cos\left(\frac{\pi}{2}-\varphi\right) & -\sin\left(\frac{\pi}{2}-\varphi\right) & 0 & 0\\ \sin\left(\frac{\pi}{2}-\varphi\right) & \cos\left(\frac{\pi}{2}-\varphi\right) & 0 & 0\\ 0 & 0 & 1 & 0\\ 0 & 0 & 0 & 1\end{pmatrix}=\begin{pmatrix}\sin\varphi & -\cos\varphi & 0 & 0\\ \cos\varphi & \sin\varphi & 0 & 0\\ 0 & 0 & 1 & 0\\ 0 & 0 & 0 & 1\end{pmatrix}. \quad (10.6)$$

Повернем систему координат $X^2Y^2Z^2$ (рис. 10.5) вокруг координатной оси X^2 на угол $(\pi-\theta)$, переход $X^2Y^2Z^2\to X^3Y^3Z^3$(рис. 10.6)

Преобразование координат имеет вид:

$$V^3=R^s_{X^2}V^2, \quad (10.7)$$

где $V^3=\left(x^3,y^3,z^3,1\right)^T$.

В результате матрица преобразования примет вид:

$$R^s_{X^2}=\begin{pmatrix}1 & 0 & 0 & 0\\ 0 & \cos(\pi-\theta) & \sin(\pi-\theta) & 0\\ 0 & -\sin(\pi-\theta) & \cos(\pi-\theta) & 0\\ 0 & 0 & 0 & 1\end{pmatrix}=\begin{pmatrix}1 & 0 & 0 & 0\\ 0 & -\cos\theta & \sin\theta & 0\\ 0 & -\sin\theta & -\cos\theta & 0\\ 0 & 0 & 0 & 1\end{pmatrix}. \quad (10.8)$$

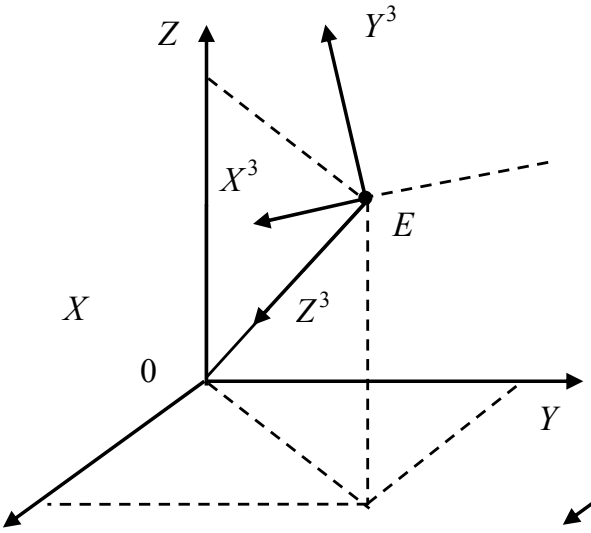


Рис. 10.6

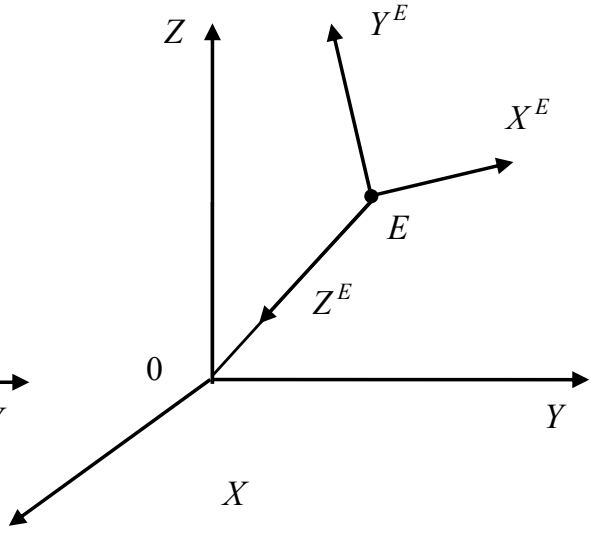


Рис. 10.7

Изменим направление оси X^3 (рис. 10.6) на противоположное $\left(X^3\to -X^3\right)$, переход $X^3Y^3Z^3\to X^EY^EZ^E$ (рис. 10.7).

Преобразование координат имеет вид:

$$V^E = MV^3, \tag{10.9}$$

где

$$M = \begin{pmatrix} -1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} - \tag{10.10}$$

матрица преобразования для перехода $(X \to -X)$.

Выражение (10.9) с учетом (10.3), (10.5), (10.7) и (10.10) можно представить в виде

$$V^E = MR^s_{X^2} R^s_{Z^1} T^s V = AV, \tag{10.11}$$

где

$$A = MR^s_{X^2} R^s_{Z^1} T^s. \tag{10.12}$$

Подставляя в (10.12) выражения для матриц (10.4), (10.6), (10.8) и (10.10), получим

$$A = \begin{pmatrix} -1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & -\cos\theta & \sin\theta & 0 \\ 0 & -\sin\theta & -\cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} \sin\varphi & -\cos\varphi & 0 & 0 \\ \cos\varphi & \sin\varphi & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \times$$

$$\times \begin{pmatrix} 1 & 0 & 0 & -x \\ 0 & 1 & 0 & -y \\ 0 & 0 & 1 & -z \\ 0 & 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} -\sin\varphi & \cos\varphi & 0 & 0 \\ -\cos\theta\cos\varphi & -\cos\theta\sin\varphi & \sin\theta & 0 \\ -\sin\theta\cos\varphi & -\sin\theta\sin\varphi & -\cos\theta & r \\ 0 & 0 & 0 & 1 \end{pmatrix}. \tag{10.13}$$

При получении (10.13) учтено, что элементы матрицы T^s x, y и z определяются выражениями (10.1).

Таким образом, преобразование (10.2), связывающее координаты мировой и видовой систем координат, можно представить в виде

$$\begin{pmatrix} x^E \\ y^E \\ z^E \\ 1 \end{pmatrix} = \begin{pmatrix} -\sin\varphi & \cos\varphi & 0 & 0 \\ -\cos\theta\cos\varphi & -\cos\theta\sin\varphi & \sin\theta & 0 \\ -\sin\theta\cos\varphi & -\sin\theta\sin\varphi & -\cos\theta & r \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} \tag{10.14}$$

или

$$\begin{cases} x^E = -x\sin\varphi + y\cos\varphi, \\ y^E = -x\cos\theta\cos\varphi - y\cos\theta\sin\varphi + z\sin\theta, \\ z^E = -x\sin\theta\cos\varphi - y\sin\theta\sin\varphi - z\cos\theta + r. \end{cases} \qquad (10.15)$$

10.3. Перспективные преобразования

Рассмотрим, как вычисляются координаты объекта в картинной плоскости, когда лучи проецирования исходят из одной точки, расположенной на конечном расстоянии от объекта (и плоскости проецирования) (рис. 10.8).

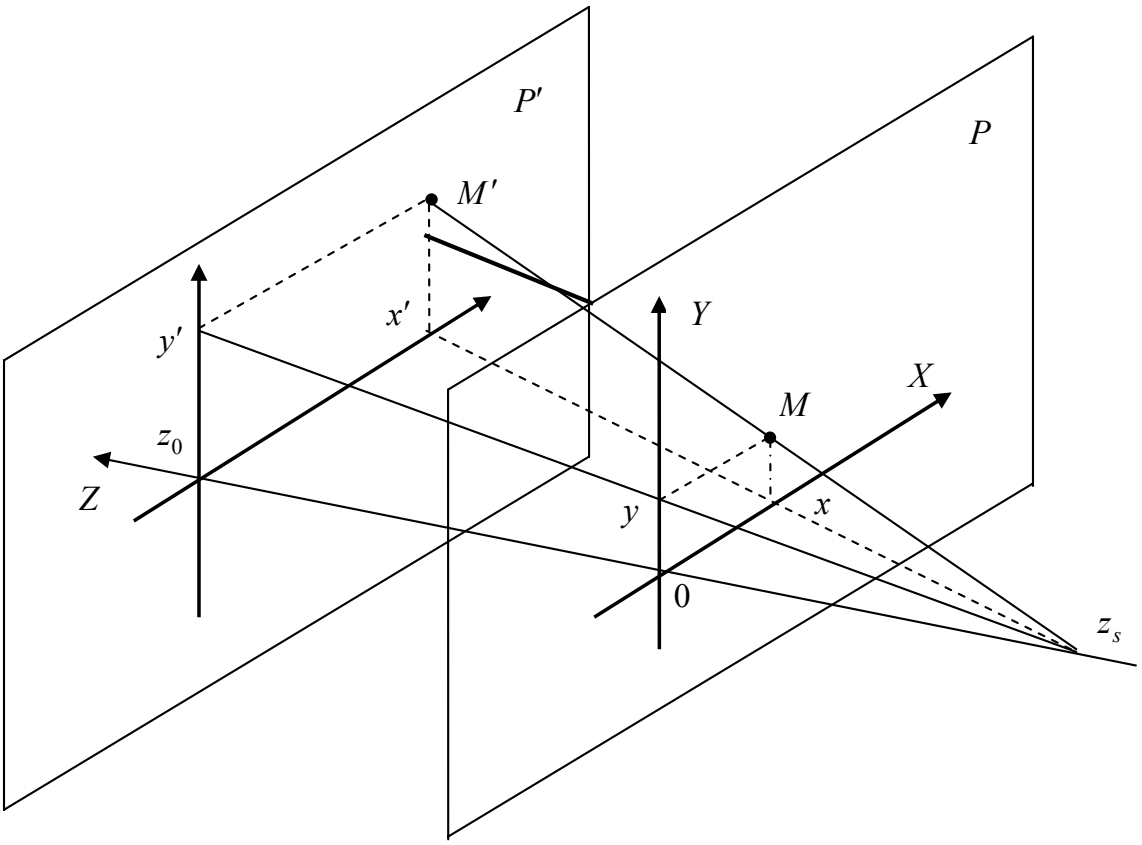


Рис. 10.8

Пусть P – картинная плоскость в видовой системе координат XYZ (плоскость проецирования – $X0Y$), $M'(x', y', z_0)$ – координаты некоторой точки, полученные пересчетом ее мировых координат в видовую систему координат XYZ без учета перспективы (аксонометрическая проекция) в соответствии с выражениями

(10.15). Далее, пусть $M(x,y)$ – проекция точки M' на картинную плоскость P, полученная с учетом перспективы, z_s – z-координата точки схода лучей. Необходимо вычислить координаты точки M, полагая, что координаты точки M' являются известными.

Из подобия треугольников $z_s 0y$ и $z_s z_0 y'$ получаем пропорцию

$$\frac{|z_s|}{y} = \frac{z_0 - z_s}{y'}, \tag{10.16}$$

откуда находим

$$y = \frac{|z_s|}{z_0 - z_s} y'. \tag{10.17}$$

Аналогично из подобия треугольников $z_s 0x$ и $z_s z_0 x'$ получим

$$\frac{|z_s|}{x} = \frac{z_0 - z_s}{x'}, \tag{10.18}$$

и

$$x = \frac{|z_s|}{z_0 - z_s} x'. \tag{10.19}$$

Вводя обозначения

$$\frac{|z_s|}{z_0 - z_s} = \beta, \tag{10.20}$$

можно записать

$$\begin{cases} x = \beta x', \\ y = \beta y'. \end{cases} \tag{10.21}$$

Последние выражения можно записать в матричном виде:

$$\begin{pmatrix} x \\ y \\ 1 \end{pmatrix} = \begin{pmatrix} \beta & 0 & 0 \\ 0 & \beta & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} \tag{10.22}$$

или

$$V = A_\beta V', \tag{10.23}$$

где

$$V = (x, y, 1)^T, \quad V' = (x', y', 1)^T,$$

$$A_\beta = \begin{pmatrix} \beta & 0 & 0 \\ 0 & \beta & 0 \\ 0 & 0 & 1 \end{pmatrix}. \qquad (10.24)$$

Таким образом, этапы преобразования координат, при переходе от мировых координат к экранным, можно отобразить в виде блок-схемы, которая показана на рис. 10.9.

Сперва мировые координаты преобразуются в видовые с началом в точке E (рис. 10.3). Затем при необходимости выполняется перспективное преобразование, добавляющее эффект перспективы. При построении аксонометрической проекции перспективное преобразование не выполняется и видовые координаты (x', y', z') (рис. 10.9) сразу используются для получения соответствующих оконных координат.

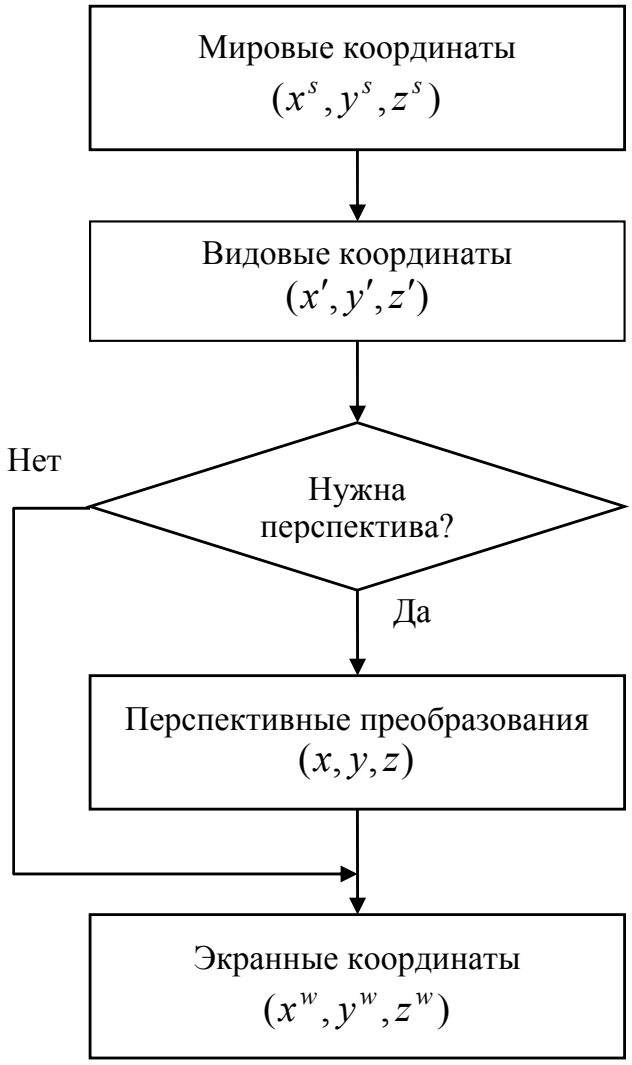


Рис. 10.9

Глава 11

МОДЕЛИ ОПИСАНИЯ ПОВЕРХНОСТЕЙ

Для описания формы поверхностей в системах компьютерной графики могут использоваться разнообразные методы. Рассмотрим некоторые из них.

11.1. Аналитическая модель

Под аналитической моделью будем понимать описание поверхности в виде математических формул. В компьютерной графике можно использовать много разновидностей такого описания. Например, в виде функции двух аргументов $z = f(x, y)$ или в неявной форме $F(x, y, z)$. Приведем примеры: $z(x, y) = x^2 + y^2$ – параболоид вращения, $x^2 + y^2 + z^2 - R^2 = 0$ – сферическая поверхность радиуса R.

Часто используется параметрическая форма описания поверхности, для которой выражения, описывающие эту поверхность в трехмерной декартовой системе координат, имеют вид:

$$\begin{cases} x = F_x(s,t), \\ y = F_y(s,t), \\ z = F_z(s,t), \end{cases} \qquad (11.1)$$

где F_x, F_y и F_z – функции, определяющие форму поверхности; s, t – параметры, которые изменяются в определенных пределах.

Например, параметрическое описание поверхности сферы радиуса R будет иметь вид

$$\begin{cases} x = R\sin\theta\cos\varphi, \\ y = R\sin\theta\sin\varphi, \quad \theta \in [0, \pi], \ \varphi \in [0, 2\pi], \\ z = R\cos\theta, \end{cases} \qquad (11.2)$$

где в качестве параметров традиционно используются переменные θ и φ.

Преимущества параметрического описания – легко описывать поверхности, которые отвечают неоднозначным функциям, замкнутые поверхности.

Для описания сложных поверхностей часто используют их приближения, составленные из элементарных фрагментов. В случае, когда эти элементарные фрагменты строятся по единой сравнительно простой схеме, такие составные поверхности принято называть **сплайновыми поверхностями**. Упомянутые элементарные фрагменты поверхности называются сплайнами. Таким образом, сплайн – это специальная функция, более всего пригодная для аппроксимации отдельных фрагментов поверхности. Несколько сплайнов образуют модель сложной поверхности. Другими словами, сплайн – эта тоже поверхность, но такая, для которой можно достаточно просто вычислять координаты ее точек. Обычно используют кубические сплайны, так как третья степень – наименьшая из степеней, позволяющих описывать любую форму, и при стыковке сплайнов можно обеспечить непрерывную первую производную – такая поверхность будет без изломов в местах стыка.

Рассмотрим одну из разновидностей сплайнов – сплайн Безье. Приведем его сначала в обобщенной форме – степени $n \times m$ [2]:

$$P(s,t) = \sum_{i=0}^{m} \sum_{j=0}^{m} C_m^i s^i (1-s)^{m-i} C_n^j t^j (1-t)^{n-j} P_{ij}, \qquad (11.3)$$

где P_{ij} – опорные точки-ориентиры; $0 \le s \le 1$; $0 \le t \le 1$; C_m^i и C_n^j – коэффициенты бинома Ньютона, которые вычисляются по формуле

$$C_n^k = \frac{n!}{k!(n-k)!}. \qquad (11.4)$$

Кубический сплайн Безье соответствует значениям $m = 3$, $n = 3$. Для его определения нужны 16 точек-ориентиров P_{ij} (рис. 11.1). Коэффициенты C_m^i и C_n^j равны соответственно 1, 3, 3, 1 при i, j = = 0, 1, 2, 3.

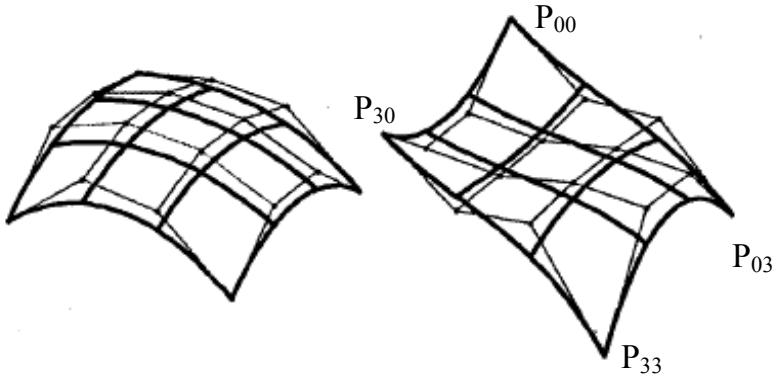


Рис. 11.1

Характеризуя аналитическую модель в целом, можно сказать, что эта модель наиболее пригодна для многих операций анализа поверхностей. С позиций компьютерной графики можно указать такие положительные черты модели: легкая процедура расчета координат каждой точки поверхности, нормали; небольшой объем информации для описания достаточно сложных форм.

К недостаткам относятся следующие: сложные формулы описания с использованием функций, которые медленно вычисляются на компьютере, снижают скорость выполнения операций отображения; невозможность в большинстве случаев применения данной формы описания непосредственно для построения изображения поверхности. В таких случаях поверхность отображают как многогранник, используя формулы аналитического описания для расчета координат вершин граней в процессе отображения, что уменьшает скорость сравнительно с полигональной моделью описания.

11.2. Векторная полигональная модель

Для описания пространственных объектов здесь используются следующие элементы: вершины, отрезки прямых (векторы), полилинии, полигоны, полигональные поверхности (рис. 11.2).

Элемент «вершина» (vertex) – главный элемент описания, все другие являются производными. При использовании трехмерной декартовой системы координаты вершин определяются как (x_i, y_i, z_i). Каждый объект однозначно определяется координатами собственных вершин.

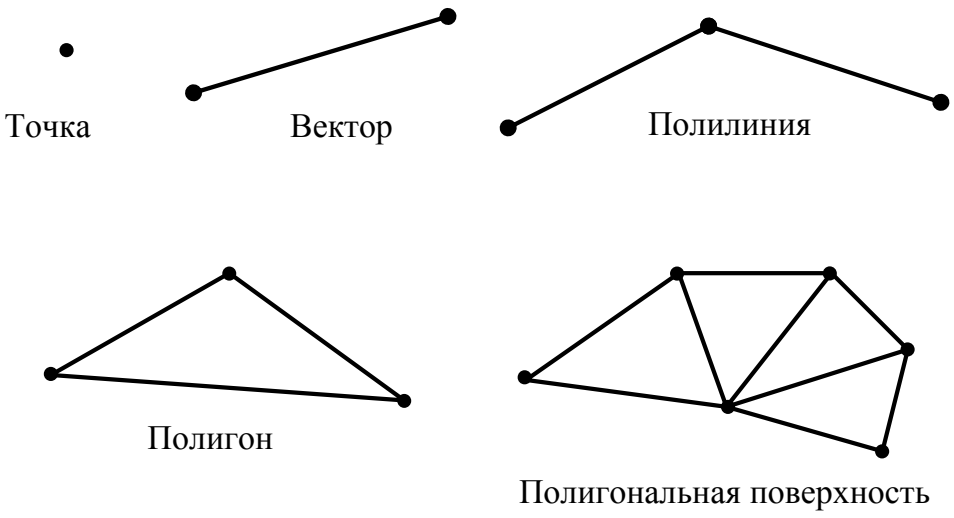


Рис. 11.2

Вершина может моделировать отдельный точечный объект, размер которого не имеет значения, а также может использоваться в качестве конечных точек для линейных объектов и полигонов. Двумя вершинами задается вектор. Несколько векторов составляют полилинию. Полилиния может моделировать отдельный линейный объект, толщина которого не учитывается, а также может представлять контур полигона. Полигон моделирует площадный объект. Один полигон может описывать плоскую грань объемного объекта. Несколько граней составляют объемный объект в виде полигональной поверхности – многогранник или незамкнутую поверхность (в литературе часто употребляется название «полигональная сетка»).

Векторную полигональную модель можно считать наиболее распространенной в современных системах трехмерной компьютерной графики. Ее используют в системах автоматизированного проектирования, в компьютерных играх и тренажерах, в САПР, геоинформационных системах и т. п.

Векторная полигональная модель, как и всякая другая, имеет свои достоинства и недостатки.

Положительные черты векторной полигональной модели:

– удобство масштабирования объектов. При увеличении или уменьшении объекты выглядят более качественно, чем при растровых моделях описания. Диапазон масштабирования определяется точностью аппроксимации и разрядностью чисел для представления координат вершин;

– небольшой объем данных для описания простых поверхностей, которые адекватно аппроксимируются плоскими гранями;

– необходимость вычислять только координаты вершин при преобразованиях систем координат или перемещении объектов;

– аппаратная поддержка многих операций в современных графических видеосистемах, которая обуславливает достаточную скорость для анимации.

– Недостатки полигональной модели:

– сложные алгоритмы визуализации для создания реалистичных изображений; сложные алгоритмы выполнения топологических операций, например разрезов;

– аппроксимация плоскими гранями приводит к погрешности моделирования. При моделировании поверхностей, которые имеют сложную форму, обычно невозможно увеличить количество граней из-за ограничений по быстродействию и объему памяти компьютера.

11.3. Равномерная сетка

Эта модель описывает координаты отдельных точек поверхности следующим способом (рис. 11.3). Каждому узлу сетки с индексами (i, j) приписывается значение высоты z_{ij}. Индексам (i, j) отвечают определенные значения координат (x_i, y_j). Расстояние между узлами одинаковое и равное $\Delta x = x_i - x_{i-1}$ по оси X и $\Delta y = y_j - y_{j-1}$ по оси Y.

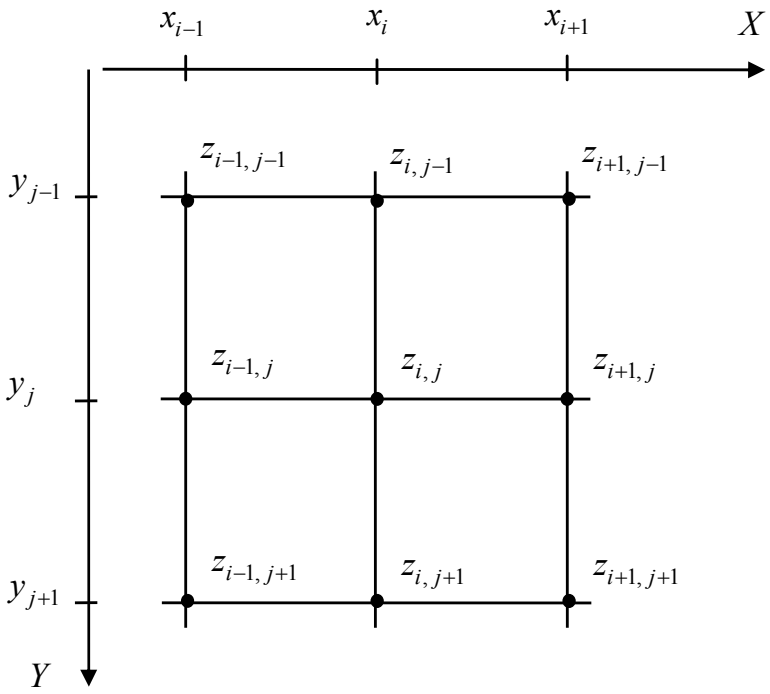


Рис. 11.3

Фактически такая модель – двумерный массив, растр, матрица, каждый элемент которой сохраняет значение высоты.

Не всякая поверхность может быть представлена этой моделью. Если в каждом узле записывается только одно значение высоты, то это означает, что поверхность описывается однозначной функцией $z = f(x, y)$. Иначе говоря, это такая поверхность, которую любая вертикаль пересекает только один раз. Не могут моделироваться также вертикальные грани. Необходимо заметить, что для сетки необязательно использовать только декартовые координаты. Например, для того чтобы описать поверхность шара однозначной функцией, можно использовать полярные координаты. Равномерная сетка часто используется для описания рельефа земной поверхности.

Рассмотрим, как можно вычислить значения высоты для любой точки внутри границ сетки. Пусть ее координаты равны $(\tilde{x}, \tilde{y})$. Надо найти соответствующее значение $\tilde{z}$. Решением такой задачи является интерполяция значений координат z ближайших узлов (рис. 11.4).

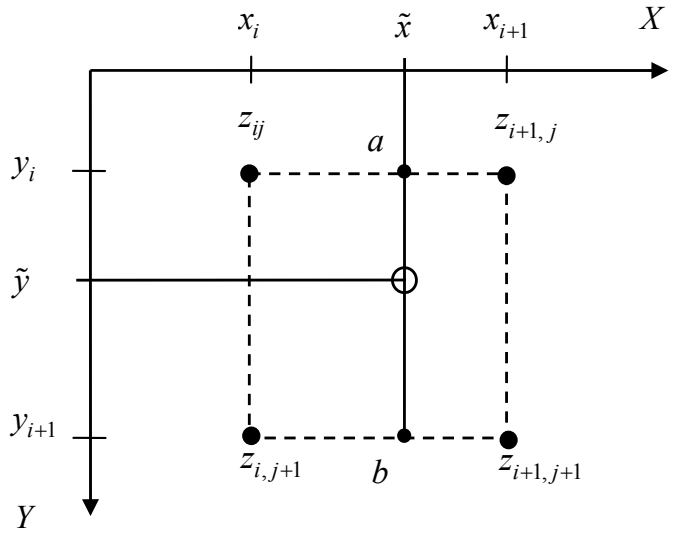


Рис. 11.4

Вычисляем индексы ближайшего узла

$$i = \left[\frac{\tilde{x} - x_0}{\Delta x} \right]; \tag{11.5}$$

$$j = \left[\frac{\tilde{y} - y_0}{\Delta y} \right], \tag{11.6}$$

где $[\bullet]$ – целая часть числа.

Используя линейную интерполяцию по оси X вдоль прямой $y = y_i$, вычислим значение z в точке a (значение z_a). Для этой цели запишем уравнение прямой, проходящей через точки (x_i, z_{ij}) и $(x_{i+1}, z_{i+1,j})$:

$$\frac{x - x_i}{x_{i+1} - x_i} = \frac{z - z_{ij}}{z_{i+1,j} - z_{ij}} \tag{11.7}$$

или

$$z = z_{ij} + \frac{z_{i+1,j} - z_{ij}}{\Delta x}(x - x_i). \tag{11.8}$$

Подставляя в (11.8) значение $x = \tilde{x}$, получаем

$$z_a = z_{ij} + \frac{z_{i+1,j} - z_{ij}}{\Delta x}(\tilde{x} - x_i). \qquad (11.9)$$

Совершенно аналогично, используя линейную интерполяцию по оси X вдоль прямой $y = y_{i+1}$, вычислим значение z в точке b (значение z_b):

$$z_b = z_{i,j+1} + \frac{z_{i+1,j+1} - z_{i,j+1}}{\Delta x}(\tilde{x} - x_i). \qquad (11.10)$$

Используя теперь линейную интерполяцию по оси Y вдоль прямой $x = \tilde{x}$, вычислим значение z в точке $(\tilde{x}, \tilde{y})$. Для этой цели запишем уравнение прямой, проходящей через точки (y_j, z_a) и (y_{j+1}, z_b):

$$\frac{y - y_j}{y_{j+1} - y_j} = \frac{z - z_a}{z_b - z_a} \qquad (11.11)$$

или

$$z = z_a + \frac{z_b - z_a}{\Delta y}(y - y_j). \qquad (11.12)$$

Подставляя в (11.12) значение $y = \tilde{y}$, получаем

$$\tilde{z} = z_a + \frac{z_b - z_a}{\Delta y}(\tilde{y} - y_j). \qquad (11.13)$$

Отметим некоторые положительные и отрицательные черты равномерной сетки.

Положительные черты равномерной сетки:

– простота описания поверхностей;

– возможность быстро узнать высоту любой точки поверхности простой интерполяцией.

Недостатки равномерной сетки:

– поверхности, которые соответствуют неоднозначной функции высоты в узлах сетки, не могут моделироваться;

– для описания сложных поверхностей необходимо большое количество узлов, которое может быть ограничено объемом памяти компьютера;

– описание отдельных типов поверхностей может быть сложнее, чем в других моделях. Например, многогранная поверхность требует избыточный объем данных для описания по сравнению с полигональной моделью.

11.4. Линии уровня

При визуализации данных, полученных экспериментальным или расчетным путем, часто встречается задача построения линий уровня на поверхностях.

Пусть функция $z = f(x, y)$ определяет некоторую поверхность в декартовой системе координат XYZ. Тогда геометрическое место точек (x, y), удовлетворяющее условию $z_0 = f(x, y)$, где $z_0 = const$, называется линией уровня. Линия уровня представляет собой результат сечения поверхности $z = f(x, y)$ плоскостью $z_0 = const$.

Рассмотрим пример. Пусть поверхность определяется уравнением $z(x, y) = x^2 + y^2$ (рис. 11.5).

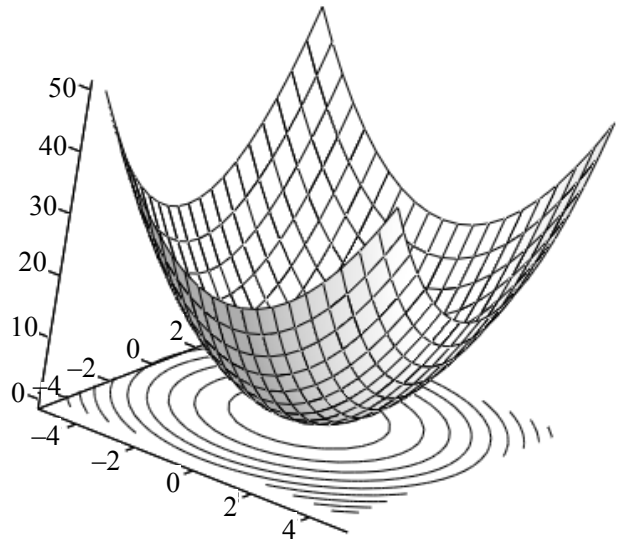


Рис. 11.5

Зададимся множеством плоскостей $z_i = const$, $i = 1, 2, \ldots$, $z_i \geq 0$. Теперь можно записать уравнение для линий уровня:

$$x^2 + y^2 = z_i. \qquad (11.14)$$

Как видно из (11.14), линии уровня для рассматриваемой поверхности представляют собой окружности радиуса $\sqrt{z_i}$ с центром в начале координат (рис. 11.5).

В частности, линии уровня знакомы нам по географическим картам, на которых они служат для обозначения высоты местности над уровнем моря.

Рассмотрим общий подход к построению линий уровня.

Пусть поверхность $z = f(x, y)$ задана массивом своих значений $z_{ij} = f(x_i, y_j)$, рассчитанных на сетке $x = \{x_1, x_2, \ldots, x_n\}$, $y = \{y_1, y_2, \ldots, y_m\}$ (рис. 11.6).

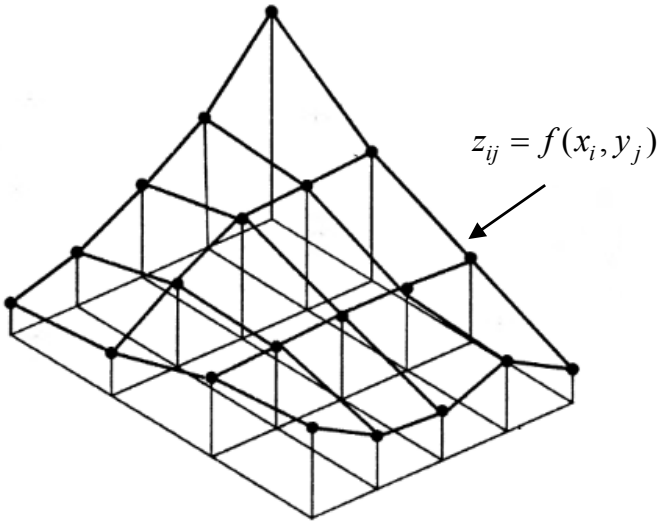


Рис. 11.6

Тогда процедура построения линий уровня заключается в следующем.

1. Выполняется триангуляция поверхности, каждая ячейка сетки разбивается на два треугольника (рис. 11.7).

2. Для каждого треугольника находится пересечение с плоскостью $z_0 = \text{const}$ (рис. 11.8).

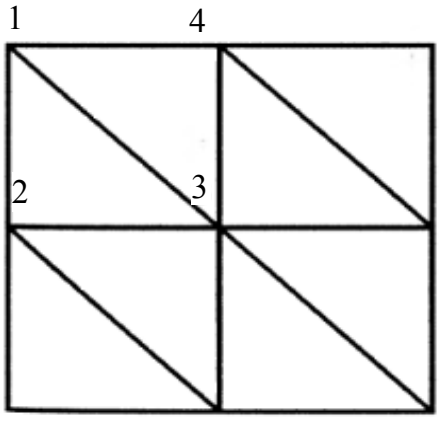


Рис. 11.7

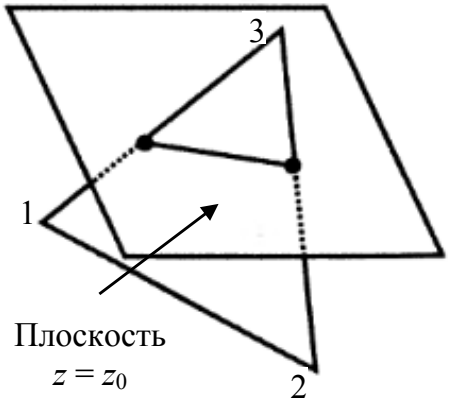


Рис. 11.8

Этот метод достаточно прост и позволяет получить хорошее изображение линии уровня. Надо отметить, что линия уровня может быть с разрывами.

Глава 12

ВИЗУАЛИЗАЦИЯ ОБЪЕМНЫХ ИЗОБРАЖЕНИЙ

Любой объект, в том числе и объемный, может быть изображен различными способами. В одном случае необходимо показать внутреннюю структуру объектов, в другом – внешнюю форму объекта, в третьем – имитировать реальную действительность. Условно способы визуализации объектов можно разделить по характеру изображений и по степени сложности соответствующих алгоритмов.

Рассмотрим следующие уровни визуализации:

1) каркасная («проволочная») модель;

2) показ поверхностей в виде многогранников с плоскими гранями или сплайнов с удалением невидимых точек;

3) то же, что и для второго уровня, плюс сложное закрашивание объектов для имитации отражения света, затенения, прозрачности, использование текстур.

12.1. Каркасная визуализация

Каркас обычно состоит из отрезков прямых линий (соответствует многограннику), хотя можно строить каркас и на основе кривых, в частности сплайновых кривых Безье. Все ребра, показанные в окне вывода, видны – как ближние, так и дальние (рис. 12.1–12.2).

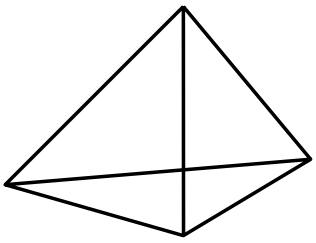


Рис. 12.1

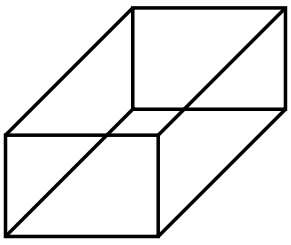


Рис. 12.2

Для построения каркасного изображения надо знать координаты всех вершин в мировой системе координат. Потом преобразовать координаты каждой вершины в экранные координаты в соответствии с выбранной проекцией. Затем выполнить цикл вывода в плоскости экрана всех ребер как отрезков прямых (или кривых), соединяющих вершины.

12.2. Показ с удалением невидимых граней

Для построения правильного изображения трехмерных объектов необходимо уметь определять, какие части объектов (ребра, грани) будут видны при заданном проектировании, а какие будут закрыты другими гранями объектов. В качестве возможных видов проектирования традиционно рассматриваются параллельное и центральное (перспективное) проектирования.

Далее будем считать, что все объекты представлены набором выпуклых плоских граней, которые пересекаются только вдоль своих ребер. В качестве примера на рис. 12.3 и 12.4 показаны ранее изображенные фигуры, у которых удалены невидимые грани.

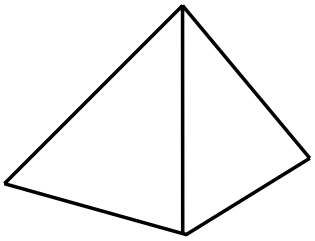


Рис. 12.3

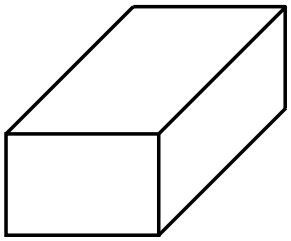


Рис. 12.4

12.2.1. Отсечение нелицевых граней

Рассмотрим многогранник, для каждой грани которого задан вектор внешней нормали (рис. 12.5). Несложно заметить, что если вектор нормали грани $\vec{n}_i$ $(i = 1, 2, 3)$ составляет с вектором $\vec{r}$, задающим направление проектирования, тупой угол, то эта грань заведомо не может быть видна. Такие грани называются **нелицевыми**. В случае, когда соответствующий угол является острым, грань называется **лицевой**.

Пусть φ – угол между вектором нормали $\vec{n}_i$ к некоторой грани и направлением проектирования $\vec{r}$, которые заданы в мировой системе координат.

Так как для случая **параллельного** проектирования направление проектирования не зависит от грани (значение вектора $\vec{r}$ не зависит от номера грани i), то условие отбора лицевой грани с номером i можно записать в виде

$$\cos\varphi_i = \frac{\vec{n}_i \cdot \vec{r}}{|\vec{n}||\vec{r}|} \geq 0, \qquad (12.1)$$

что говорит о том, что угол φ является острым и его значение принадлежит отрезку $[0, \pi/2]$.

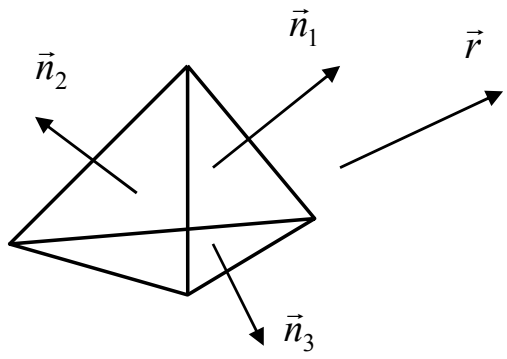


Рис. 12.5

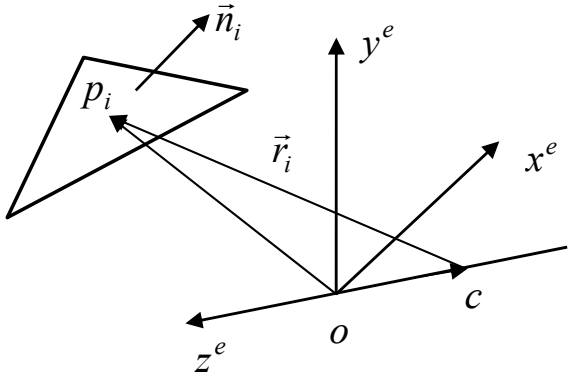


Рис. 12.6

Поскольку знаменатель в (12.1) всегда положителен, то условие (12.1) можно заменить на более простое:

$$\vec{n}_i \cdot \vec{r} \geq 0 . \qquad (12.2)$$

Рассмотрим случай **центрального** проектирования (рис. 12.6). Пусть для этого случая в видовой системе координат заданы: p_i – произвольная точка на некоторой грани с номером i; $\vec{n}_i$ – вектор внешней нормали к этой грани; c – точка, определяющая центр проектирования; $\vec{r}_i$ – направление проектирования для точки p_i; φ_i – угол между вектором нормали $\vec{n}_i$ и направлением проектирования $\vec{r}_i$. В отличие от случая параллельного проектирования значение вектора $\vec{r}_i$ зависит от положения точки p_i на выбранной грани:

$$\vec{r}_i = \overrightarrow{op_i} - \overrightarrow{oc} . \qquad (12.3)$$

Теперь можно записать условие отбора лицевой грани:

$$\cos\varphi_i = \frac{\vec{n}_i \cdot (-\vec{r}_i)}{|\vec{n}_i||\vec{r}_i|} \geq 0, \qquad (12.4)$$

или более просто

$$\vec{n}_i \cdot (-\vec{r}_i) \geq 0 \ (\vec{n}_i \cdot \vec{r}_i \leq 0). \qquad (12.5)$$

Знак скалярного произведения в (12.5) не зависит от выбора точки на грани, а определяется тем, в каком полупространстве относительно плоскости, содержащей данную грань, лежит центр проектирования. Поэтому для определения того, является заданная грань лицевой или нет, достаточно взять произвольную точку на этой грани и проверить выполнение условия (12.5).

В случае, когда фигура представляет собой один выпуклый многогранник, удаление нелицевых граней полностью решает задачу удаления невидимых граней.

В общем случае предложенный подход хотя и не решает задачу полностью, но позволяет примерно вдвое сократить количество рассматриваемых граней.

12.2.2. Сортировка граней по глубине

Это означает рисование полигонов граней в порядке от самых дальних к самым близким. Данный метод не является универсальным, ибо иногда нельзя четко различить, какая грань ближе (рис. 12.7).

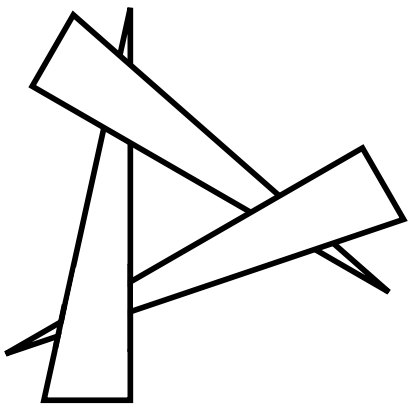


Рис. 12.7

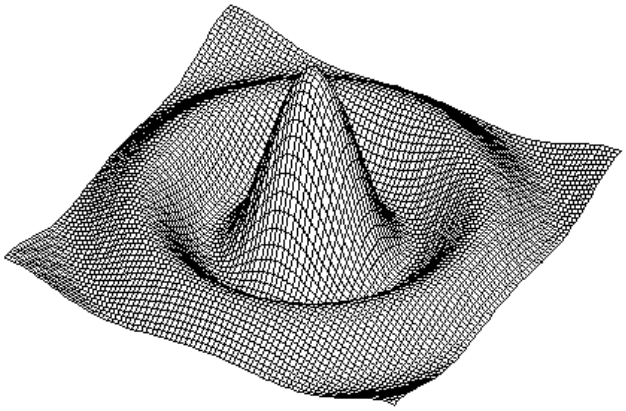


Рис. 12.8

Известны модификации этого метода, которые позволяют корректно рисовать такие грани. Метод сортировки по глубине эффективен для показа поверхностей (рис. 12.8), заданных функциями $z = f(x, y)$.

В качестве примера рассмотрим возможный вариант определения дальней грани в зависимости от положения наблюдателя для случая построения графика поверхности, определяемой функцией $z = f(x, y)$.

Пусть рассчитана матрица значений функции $f(x, y)$:

$$M_{ij} = z_{ij} = f(x_i, y_j), \tag{12.6}$$

где

$$x_i = x_L + i\Delta x, \;\; i = 0, 1, 2, \ldots, N_x, \quad N_x = \frac{x_H - x_L}{\Delta x}; \tag{12.7}$$

$$y_j = y_L + j\Delta y, \;\; j = 0, 1, 2, \ldots, N_y, \; N_y = \frac{y_H - y_L}{\Delta y}, \tag{12.8}$$

x_L, x_H – соответственно нижняя и верхняя границы для изменения аргумента x; y_L, y_H – нижняя и верхняя границы для изменения аргумента y; Δx и Δy – интервалы дискретизации для аргументов x и y.

Далее, пусть $P(x_p, y_p, z_p)$ – точка в системе координат XYZ, где расположен наблюдатель, а $P_{xy}(x_p, y_p)$ – проекция точки P на плоскость XY (рис. 12.9–12.10). На приведенных рисунках матрица M представлена прямоугольником $ACBD$.

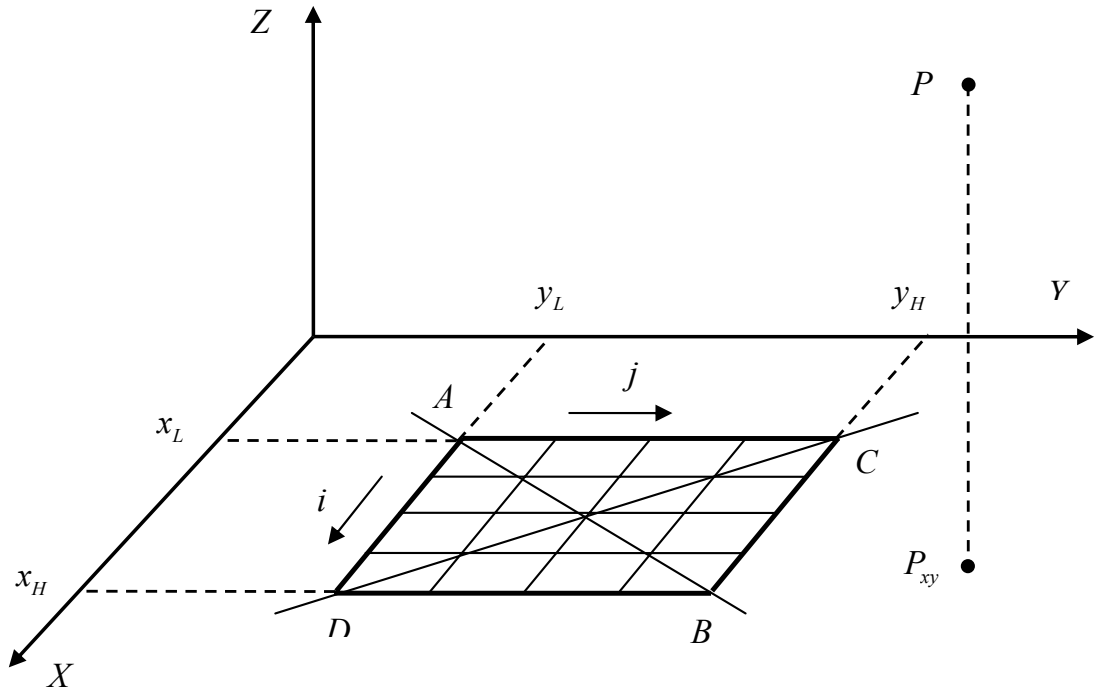


Рис. 12.9

Через точки A и B проведем прямую AB, уравнение которой будет иметь вид

$$\frac{x - x_L}{x_H - x_L} = \frac{y - y_L}{y_H - y_L}$$

или

$$y = y_L + \frac{y_H - y_L}{x_H - x_L}(x - x_L) = y_1(x). \qquad (12.9)$$

Аналогично через точки C и D проведем прямую CD, уравнение которой будет иметь вид

$$\frac{x - x_L}{x_H - x_L} = \frac{y - y_L}{y_L - y_H}$$

или

$$y = y_L - \frac{y_H - y_L}{x_H - x_L}(x - x_L) = y_2(x). \qquad (12.10)$$

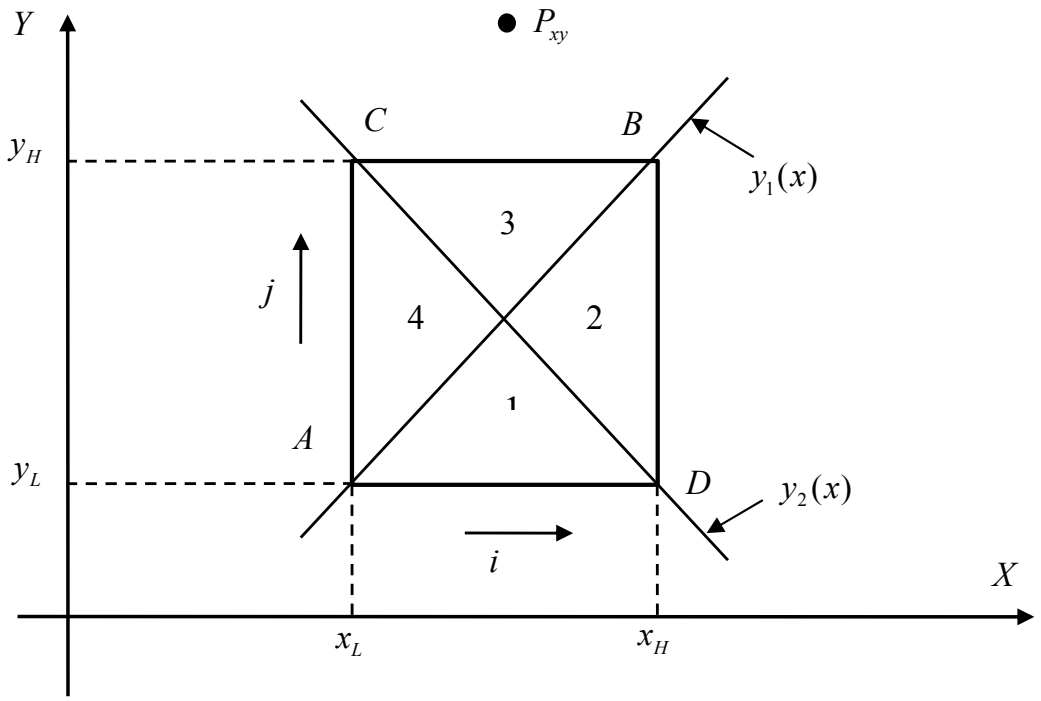


Рис. 12.10

Прямые AB и CD делят плоскость XY на четыре области: 1, 2, 3, 4 (рис. 12.10). Из рис. 12.10 видно, что дальняя грань определяется по отношению к расположению точки $P_{xy}(x_p, y_p)$ на плоскости XY.

Если $y_p \geq y_1(x_p)$ и $y_p \geq y_2(x_p)$, то точка P_{xy} находится в области 3 и дальней является грань AD.

Если $y_2(x_p) < y_p < y_1(x_p)$, то точка P_{xy} находится в области 2 и дальней является грань AC.

Если $y_p \leq y_1(x_p)$ и $y_p \leq y_2(x_p)$, то точка P_{xy} находится в области 1 и дальней является грань CB.

Если $y_1(x_p) < y_p < y_2(x_p)$, то точка P_{xy} находится в области 4 и дальней является грань BD.

12.2.3. Метод плавающего горизонта

В отличие от предыдущего метода при методе плавающего горизонта грани выводятся в последовательности от ближайших к самым дальним. На каждом шаге границы граней образовывают две ломаные линии – верхний горизонт и нижний горизонт. Во время вывода каждой новой грани рисуется только то, что выше верхнего горизонта, и то, что ниже нижнего горизонта. Соответственно, каждая новая грань поднимает верхний и опускает нижний горизонты. Этот метод также используют для показа поверхностей, которые описываются функциями $z = f(x, y)$.

12.2.4. Метод *z*-буфера

Метод основывается на использовании дополнительного массива, буфера в памяти, в котором сохраняются координаты z для каждого пиксела растра. Координата z отвечает расстоянию точек пространственных объектов до плоскости проецирования.

Рассмотрим алгоритм рисования объектов согласно этому методу. Пусть чем ближе точка в пространстве к плоскости проецирования, тем больше значение z. Тогда сначала z-буфер заполняется минимальными значениями. Потом начинается вывод всех объектов. Причем не имеет значение порядок вывода объектов. Для каждого объекта выводятся все его пикселы в любом порядке. Во время вывода каждого пиксела по его координатам (x, y) находится текущее значение z в z-буфере. Если рисуемый пиксел имеет большее значение z, чем значение в z-буфере, то это означает, что данная точка ближе к объекту. В этом случае пиксел действительно рисуется, а его z-координата записывается в z-буфер. Таким образом, после рисования всех пикселов всех объектов растровое изображение будет состоять из пикселов, которые соответствуют точкам объектов с самыми большими значениями координат z, т. е. видимые точки – ближе всех к нам.

Этот метод прост и эффективен благодаря тому, что не нужно сортировать ни объекты, ни их точки. При рисовании объектов, которые описываются многогранниками или полигональными сетками, манипуляции со значениями z-буфера легко совместить с выводом пикселов заполнения полигонов плоских граней.

В настоящее время метод z-буфера используется во многих графических 3d-акселераторах, которые аппаратно реализуют этот метод. Наиболее целесообразно, когда акселератор имеет собственную память для z-буфера, доступ к которой осуществляется быстрее, чем к оперативной памяти компьютера. Возможности аппаратной реализации используются разработчиками и пользователями компьютерной анимации, позволяя достичь большой скорости прорисовки кадров.

Глава 13

ЗАКРАШИВАНИЕ ПОВЕРХНОСТЕЙ

13.1. Модели отражения света

Рассмотрим, как можно определить цвет пикселов изображения поверхности согласно интенсивности отраженного света при учете взаимного расположения поверхности, источника света и наблюдателя.

13.1.1. Зеркальное отражение света

Угол между нормалью и падающим лучом φ равен углу между нормалью и отраженным лучом. Падающий луч, отраженный и нормаль располагаются в одной плоскости (рис. 13.1).

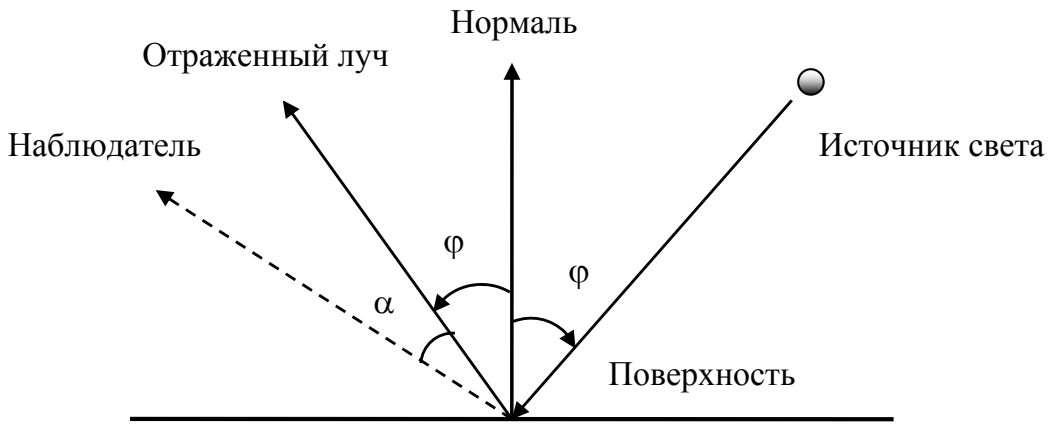


Рис. 13.1

Поверхность считается **идеально зеркальной**, если на ней отсутствуют какие-либо неровности, шероховатости. Собственный цвет у такой поверхности не наблюдается. Световая энергия падающего луча отражается только по линии отраженного луча. Какое-либо рассеяние в стороны от этой линии отсутствует. В природе, вероятно, нет идеально гладких поверхностей, поэтому полагают, что если глубина шероховатостей существенно меньше длины волны излучения, то рассеивания не наблюдается. Для видимого спектра можно принять, что глубина шероховатостей поверхности зеркала должна быть существенно меньше 0,5 мкм [2].

Если поверхность зеркала отполирована неидеально, то наблюдается зависимость интенсивности отраженного света от дли-

ны волны – чем больше длина волны, тем лучше отражение. Например, красные лучи отражаются сильнее, чем синие.

Падающий луч, попадая на слегка шероховатую поверхность реального зеркала, порождает не один отраженный луч, а несколько лучей, рассеиваемых по различным направлениям. Зона рассеивания зависит от качества полировки и может быть описана некоторым законом распределения. Как правило, форма зоны рассеивания симметрична относительно линии идеального зеркально отраженного луча.

К числу простейших, но достаточно часто используемых, относится эмпирическая модель распределения Фонга, согласно которой интенсивность зеркально отраженного излучения определяется выражением

$$I_s = I_0 K_s \cos^p \alpha, \qquad (13.1)$$

где I_0 – интенсивность излучения источника; K_s – коэффициент пропорциональности; α – угол отклонения от линии идеально отраженного луча (рис. 13.1); показатель p находится в диапазоне от 1 до 200 и зависит от качества полировки.

13.1.2. Диффузное отражение света

Этот вид отражения присущ **матовым** поверхностям. Матовой можно считать такую поверхность, размер шероховатостей которой уже настолько велик, что падающий луч рассеивается равномерно во все стороны. Такой тип отражения характерен, например, для гипса, песка, бумаги.

Для матовой поверхности законы отражения установлены Ламбертом. Их суть иллюстрируется рис. 13.2 и 13.3.

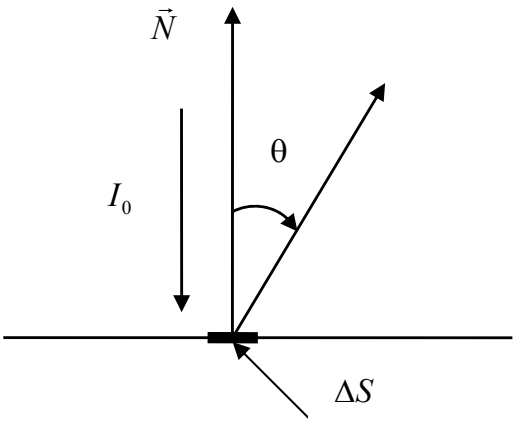


Рис. 13.2

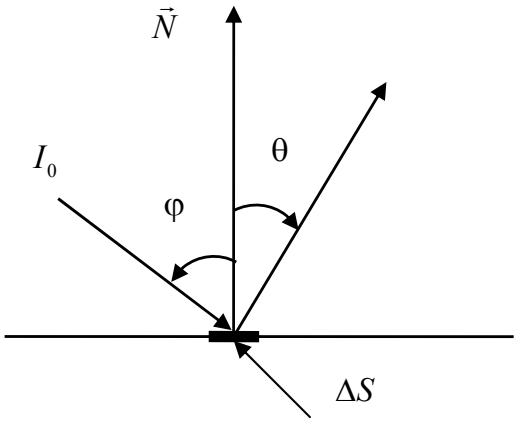


Рис. 13.3

Если световой поток интенсивностью (мощностью) I_0 падает нормально к матовой поверхности (рис. 13.2), то интенсивность вторичного излучения I_d под углом θ к нормали, проведенной к элементарной площадке ΔS, пропорционально $\cos\theta$ [2]:

$$I_d = I_0 K_d \cos\theta, \qquad (13.2)$$

где K_d – коэффициент, который учитывает свойства материала поверхности. Значение K_d находится в диапазоне от 0 до 1.

Физический смысл формулы (13.2) заключается в том, что в направлении, определяемом углом θ, площадь ΔS проектируется как $\Delta S\cos\theta$, при этом пропорционально $\cos\theta$ уменьшается площадь излучающей площадки и интенсивность излучения.

Если поток излучения падает под углом φ по отношению к нормали, проведенной к поверхности в месте падения (рис. 13.3), то $I_d = I_1 K_d \cos\theta$, где $I_1 = I_0 \cos\varphi$.

Следовательно

$$I_d = I_0 K_d \cos\varphi \cos\theta. \qquad (13.3)$$

Здесь I_0 – интенсивность светового потока, которую фиксировал бы приемник, находящийся в «зените», если бы площадка облучалась тоже с зенита.

Физический смысл формулы (13.3) состоит в том, что с увеличением φ уменьшается перехватываемый поверхностью падающий световой поток, отчего уменьшается ее освещенность и, как следствие, яркость.

Матовая поверхность имеет свой цвет. Наблюдаемый цвет матовой поверхности определяется комбинацией собственного цвета поверхности и цвета излучения источника света.

При создании реалистичных изображений следует учитывать то, что в природе, вероятно, не существует идеально зеркальных или полностью матовых поверхностей. При изображении объектов средствами компьютерной графики обычно моделируют сочетание зеркальности и диффузного рассеивания в пропорции, характерной для конкретного материала. В этом случае модель отражения записывают в виде суммы диффузной и зеркальной компонент:

$$I_{\text{отр}} = I_0[K_d \cos\varphi\cos(\varphi+\alpha) + K_s \cos^p \alpha], \qquad (13.4)$$

где константы K_d и K_s определяют отражательные свойства материала. Согласно этой формуле интенсивность отраженного света

равна нулю для некоторых углов φ и α. Однако в реальных сценах обычно нет полностью затемненных объектов, следует учитывать фоновую подсветку, освещение рассеянным светом, отраженным от других объектов. В таком случае интенсивность может быть эмпирически выражена следующей формулой:

$$I_{\text{отр}} = I_a K_a + I_0[K_d \cos\varphi \cos(\varphi + \alpha) + K_s \cos^p \alpha], \qquad (13.5)$$

где I_a – интенсивность рассеянного света; K_a – константа.

Можно еще усовершенствовать модель отражения, если учесть то, что энергия от точечного источника света уменьшается пропорционально квадрату расстояния. Использование такого правила вызывает сложности, поэтому на практике часто реализуют модель, выражаемую эмпирической формулой [2]:

$$I_{\text{отр}} = I_a K_a + \frac{I_0}{R + k}[K_d \cos\varphi \cos(\varphi + \alpha) + K_s \cos^p \alpha], \qquad (13.6)$$

где R – расстояние от центра проекции до поверхности; k – константа.

Распределение световой энергии по возможным направлениям световых лучей можно отобразить с помощью векторных диаграмм, в которых длина векторов соответствует интенсивности (рис. 13.4). На приведенном рисунке показаны векторные диаграммы для различных видов отражений: *а* – идеальное зеркальное; *б* – неидеальное зеркальное; *в* – диффузионное; *г* – сумма диффузионного и зеркального.

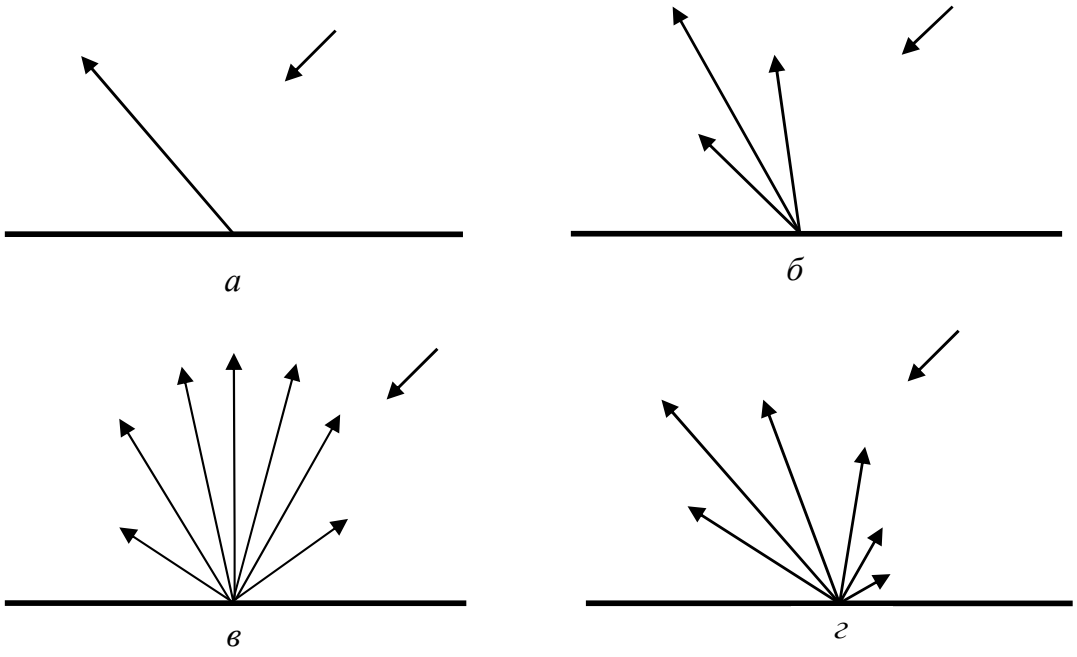


Рис. 13.4

Рассмотрим, как определить цвет закрашивания точек объектов в соответствии с выбранной моделью. Наиболее просто выполняется расчет в градациях серого цвета (например, для белого источника света и серых объектов). В данном случае интенсивность отраженного света соответствует яркости. Сложнее обстоит дело с цветными источниками света, освещающими цветные поверхности. Например, для модели RGB составляются три формулы расчета интенсивности отраженного света для различных цветовых компонент. Коэффициенты K_a и K_d различны для разных компонент – они выражают собственный цвет поверхности. Поскольку цвет отраженного зеркального луча равен цвету источника, то коэффициент K_s будет одинаковым для всех компонент цветовой модели. Цвет источника света выражается значениями интенсивности I для соответствующих цветовых компонент.

13.2. Вычисление углов

В рассматриваемых нами моделях отражения света присутствуют такие параметры, как угол падения и отражения φ, угол α – угол отклонения от линии идеально отраженного луча (рис. 13.1), угол θ – угол, под которым наблюдается результат диффузионного отражения света (рис. 13.2, 13.3). Рассмотрим алгоритмы вычисления этих углов в выбранной системе координат.

Диффузионное отражение. Определим косинус угла ϑ между вектором нормали $\vec{N}$ к поверхности и некоторым направлением, определяемым вектором $\vec{S}$. Вектор $\vec{S}$ может указывать как на положение источника света, так и на положение точки наблюдения. Таким образом, угол ϑ – это угол, соответствующий углам φ или θ на рис. 13.1–13.3.

Сначала рассмотрим случай, когда источник света или наблюдатель находятся на бесконечности по отношению к некоторому элементу поверхности (рис. 13.5).

Пусть заданы:

$\vec{N} = \vec{N}(N_x, N_y, N_z)$ – вектор нормали к элементу поверхности;

$\vec{S} = \vec{S}(S_x, S_y, S_z)$ – вектор, определяющий некоторое направление в пространстве.

Тогда

$$\cos\vartheta = \frac{\vec{S}\cdot\vec{N}}{\left|\vec{S}\right|\cdot\left|\vec{N}\right|} = \frac{S_x N_x + S_y N_y + S_z N_z}{\sqrt{S_x^2 + S_y^2 + S_z^2}\sqrt{N_x^2 + N_y^2 + N_z^2}}. \qquad (13.7)$$

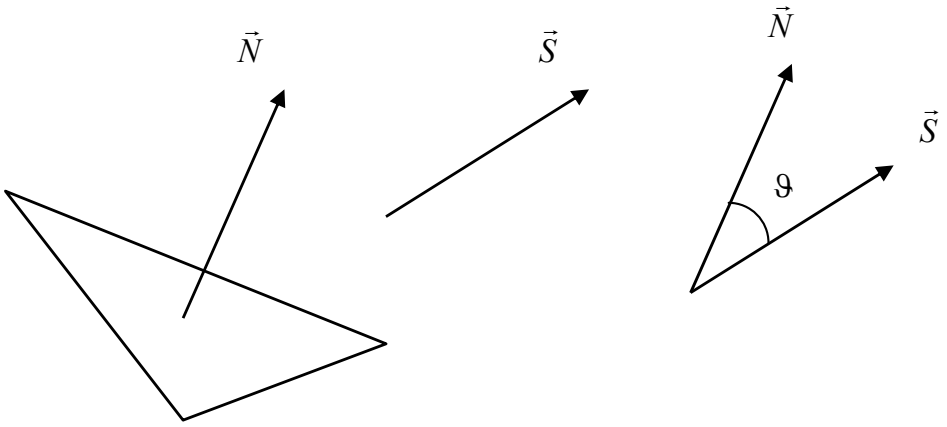


Рис. 13.5

Перейдем к рассмотрению случая, когда источник света или наблюдатель находятся на конечном расстоянии от поверхности (рис. 13.6).

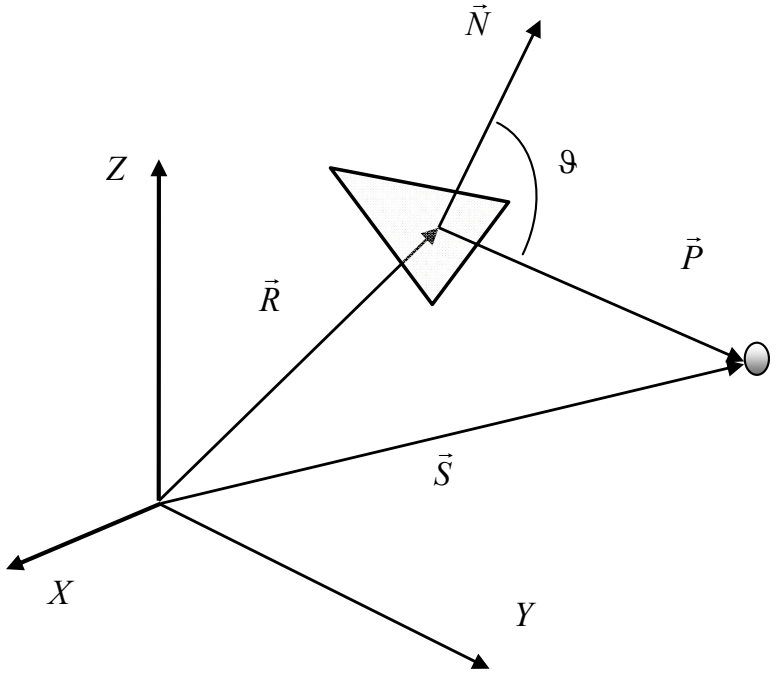


Рис. 13.6

Пусть заданы в системе координат XYZ (рис. 13.6):

$\vec{N} = \vec{N}(N_x, N_y, N_z)$ – вектор нормали к элементу поверхности;

$\vec{S} = \vec{S}(S_x, S_y, S_z)$ – радиус-вектор, определяющий положение источника света или точки наблюдения;

$\vec{R} = \vec{R}(R_x, R_y, R_z)$ – радиус-вектор, определяющий положение элемента поверхности.

Тогда

$$\cos\vartheta = \frac{\vec{P}\cdot\vec{N}}{\left|\vec{P}\right|\cdot\left|\vec{N}\right|} = \frac{P_x N_x + P_y N_y + P_z N_z}{\sqrt{P_x^2 + P_y^2 + P_z^2}\sqrt{N_x^2 + N_y^2 + N_z^2}}. \qquad (13.8)$$

Координаты вектора $\vec{P} = \vec{P}(P_x, P_y, P_z)$ определяются из рис. 13.6:

$$\vec{P} = \vec{S} - \vec{R}. \qquad (13.9)$$

Таким образом

$$\vec{P} = \vec{P}(P_x, P_y, P_z) = \vec{P}(S_x - R_x, S_y - R_y, S_z - R_z). \qquad (13.10)$$

С учетом (13.9) и (13.10) выражение (13.8) можно представить в виде

$$\cos\vartheta = \frac{(\vec{S}-\vec{R})\cdot\vec{N}}{\left|\vec{S}-\vec{R}\right|\cdot\left|\vec{N}\right|} = \frac{(S_x-R_x)N_x + (S_y-R_y)N_y + (S_z-R_z)N_z}{\sqrt{(S_x-R_x)^2 + (S_y-R_y)^2 + (S_z-R_z)^2}\sqrt{N_x^2+N_y^2+N_z^2}}. \qquad (13.11)$$

Зеркальное отражение. Пусть заданы (рис. 13.7):

$\vec{N} = \vec{N}(N_x, N_y, N_z)$ – вектор нормали к элементу поверхности;

$\vec{S} = \vec{S}(S_x, S_y, S_z)$ – вектор, определяющий направление на источник света;

$\vec{R} = \vec{R}(R_x, R_y, R_z)$ – вектор, определяющий направление отраженного луча;

$\vec{K} = \vec{K}(K_x, K_y, K_z)$ – вектор, определяющий направление на камеру.

Будем полагать, что

$$\left|\vec{R}\right| = \left|\vec{S}\right|. \qquad (13.12)$$

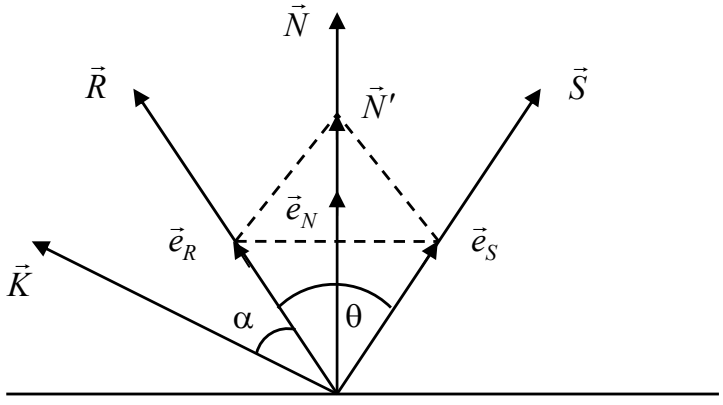


Рис. 13.7

Требуется определить косинус угла α.

На векторах $\vec{R}$, $\vec{N}$ и $\vec{S}$ построим соответствующие им единичные векторы $\vec{e}_R$, $\vec{e}_N$, $\vec{e}_S$ и выполним построения, показанные на рис. 13.7:

$$\vec{e}_R = \frac{\vec{R}}{|\vec{R}|}, \qquad \vec{e}_N = \frac{\vec{N}}{|\vec{N}|}, \qquad \vec{e}_S = \frac{\vec{S}}{|\vec{S}|}. \tag{13.13}$$

Из рис. 13.7 получаем

$$\vec{e}_R + \vec{e}_S = \vec{N}' = 2\vec{e}_N |\vec{e}_S| \cos\theta = 2\vec{e}_N \cos\theta; \tag{13.14}$$

$$\cos\theta = \frac{\vec{e}_N \cdot \vec{e}_S}{|\vec{e}_N||\vec{e}_S|} = \vec{e}_N \cdot \vec{e}_S. \tag{13.15}$$

Подставляя (13.15) в (13.14), получим

$$\vec{e}_R + \vec{e}_S = 2\vec{e}_N \cos\theta = 2\vec{e}_N (\vec{e}_N \cdot \vec{e}_S). \tag{13.16}$$

Отсюда

$$\vec{e}_R = 2\vec{e}_N (\vec{e}_N \cdot \vec{e}_S) - \vec{e}_S. \tag{13.17}$$

Подставляя в (13.17) выражения для единичных векторов из (13.13), получим

$$\frac{\vec{R}}{|\vec{R}|} = 2\frac{\vec{N}}{|\vec{N}|}\left(\frac{\vec{N}}{|\vec{N}|} \cdot \frac{\vec{S}}{|\vec{S}|}\right) - \frac{\vec{S}}{|\vec{S}|} = 2\frac{\vec{N}(\vec{N} \cdot \vec{S})}{|\vec{N}|^2 |\vec{S}|} - \frac{\vec{S}}{|\vec{S}|},$$

откуда с учетом (13.12)

$$\vec{R} = 2\frac{\vec{N}(\vec{N} \cdot \vec{S})}{|\vec{N}|^2} - \vec{S}. \tag{13.18}$$

И окончательно

$$\cos\alpha = \frac{\vec{K} \cdot \vec{R}}{|\vec{K}||\vec{R}|}. \tag{13.19}$$

13.3. Метод Гуро

Этот метод предназначен для создания иллюзии гладкой криволинейной поверхности, описанной в виде многогранников или полигональной сетки с плоскими гранями. Если каждая плоская грань имеет один постоянный цвет, определенный с учетом отражения, то

различные цвета соседних граней очень заметны, и поверхность выглядит именно как многогранник. Казалось бы, этот дефект можно замаскировать за счет увеличения количества граней при аппроксимации поверхности. Но зрение человека имеет способность подчеркивать перепады яркости на границах смежных граней – такой эффект называется **эффектом полос Маха**. Поэтому для создания иллюзии гладкости нужно намного увеличить количество граней, что приводит к существенному замедлению визуализации – чем больше граней, тем меньше скорость рисования объектов.

Метод Гуро основывается на идее закрашивания каждой плоской грани не одним цветом, а плавно изменяющимися оттенками, вычисляемыми путем интерполяции цветов примыкающих граней. Закрашивание граней по методу Гуро осуществляется в четыре этапа:

– вычисляются нормали к каждой грани;

– определяются нормали в вершинах, нормаль в вершине находится усреднением нормалей примыкающих граней (рис. 13.8);

– на основе нормалей в вершинах вычисляются значения интенсивностей в вершинах согласно выбранной модели отражения света;

– закрашиваются полигоны граней цветом, соответствующим линейной интерполяции значений интенсивности в вершинах.

Вектор нормали в вершине *a* (рис. 13.8) равен

$$\vec{N}_a = \frac{\vec{N}_1 + \vec{N}_2 + \vec{N}_3}{3}. \qquad (13.20)$$

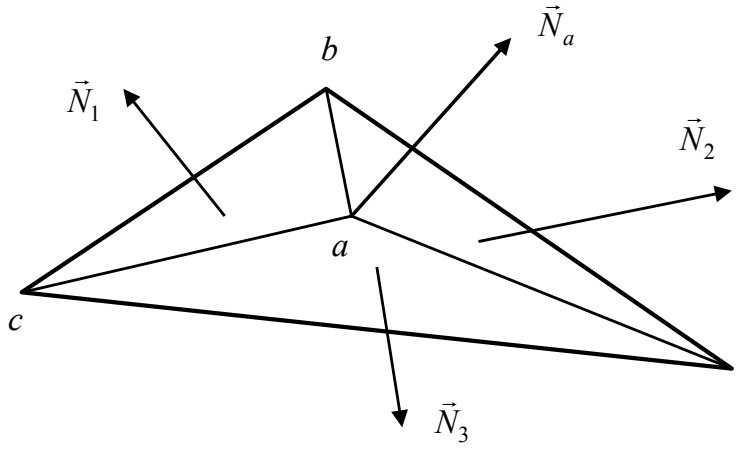


Рис. 13.8

Определение интерполированных значений интенсивности отраженного света в каждой точке грани (и, следовательно, цвет каждого пиксела) удобно выполнять во время цикла заполнения

полигона. Рассмотрим заполнение контура грани горизонталями в экранных координатах (рис. 13.9).

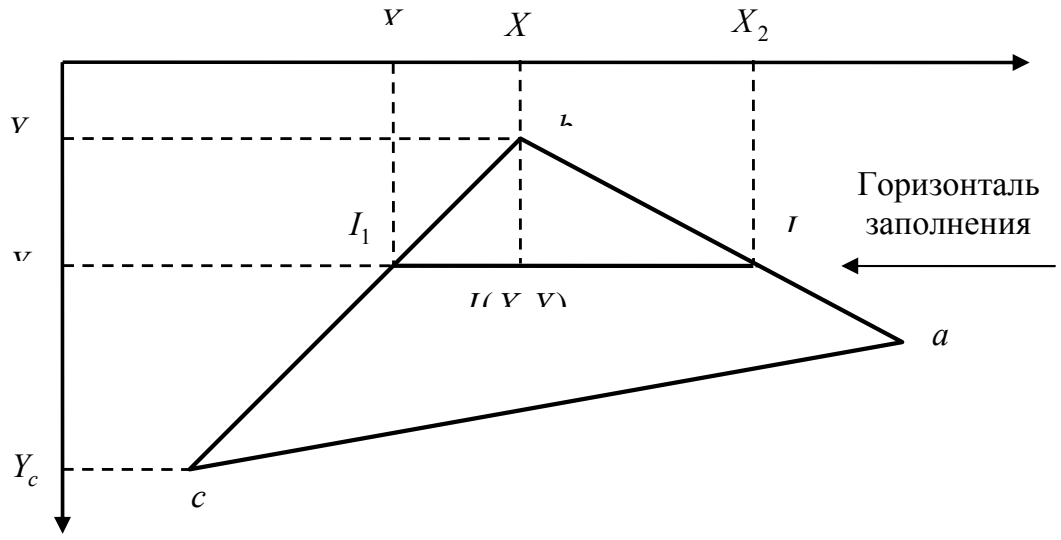


Рис. 13.9

Интерполированная интенсивность I в точке (X,Y) определяется исходя из пропорции

$$\frac{I - I_1}{X - X_1} = \frac{I_2 - I_1}{X_2 - X_1}. \qquad (13.21)$$

Отсюда находим

$$I = I_1 + (I_2 - I_1)\frac{X - X_1}{X_2 - X_1}. \qquad (13.22)$$

Значения интенсивностей I_1 и I_2 на концах горизонтального отрезка представляют собой интерполяцию интенсивности в вершинах и могут быть получены из пропорций:

$$\frac{I_1 - I_b}{Y - Y_b} = \frac{I_c - I_b}{Y_c - Y_b};$$

$$\frac{I_2 - I_b}{Y - Y_b} = \frac{I_a - I_b}{Y_a - Y_b}.$$

Откуда получаем

$$I_1 = I_b + (I_c - I_b)\frac{Y - Y_b}{Y_c - Y_b}; \qquad (13.23)$$

$$I_2 = I_b + (I_a - I_b)\frac{Y - Y_b}{Y_a - Y_b}. \qquad (13.24)$$

Глава 14

ПОСТРОЕНИЕ КРИВЫХ

Одним из важнейших применений компьютерной графики является визуализация научных данных [5]. Визуализацию научных данных принято относить к той области компьютерной графики, которая получила название научная графика. Научная графика – это представление результатов научных расчетов или экспериментов в графической форме. Такая форма представления значительно упрощает анализ полученных результатов и позволяет взглянуть на них по-новому. Первые графики на машине получали в режиме символьной печати. Затем появились специальные устройства – графопостроители (плоттеры) для вычерчивания чертежей и графиков чернильным пером на бумаге. Современная научная компьютерная графика дает возможность проводить вычислительные эксперименты с наглядным представлением их результатов в виде кривых на плоскости и поверхностей в пространстве.

Данный раздел учебно-методического пособия посвящен некоторым методам построения плоских кривых по дискретному набору данных – точек на плоскости, заданных своими координатами.

14.1. Интерполяция функций

Интерполяция – в вычислительной математике способ нахождения промежуточных значений величины по имеющемуся дискретному набору известных значений. Многим из тех, кто сталкивается с научными и инженерными расчетами, часто приходится оперировать наборами значений, полученных экспериментальным путем или методом случайной выборки. Как правило, на основании этих наборов требуется построить функцию, на которую могли бы с высокой точностью попадать другие получаемые значения. Такая задача называется **аппроксимацией**. **Интерполяцией** называют такую разновидность аппроксимации, при которой кривая построенной функции проходит точно через имеющиеся точки данных.

14.2. Постановка задачи интерполяции

Пусть задано множество несовпадающих точек (x_i, y_i), $i = 0, 1, 2, ..., n$ на плоскости.

Требуется найти такую функцию $f(x)$, принадлежащую заданному классу функций, что для всех точек x_i, $i = 0, 1, 2, \ldots, n$ выполняется условие $f(x_i) = y_i$.

Геометрически это означает, что нужно найти кривую $y = f(x)$, которая на плоскости XY проходит через заданную систему точек (x_i, y_i) (рис. 14.1).

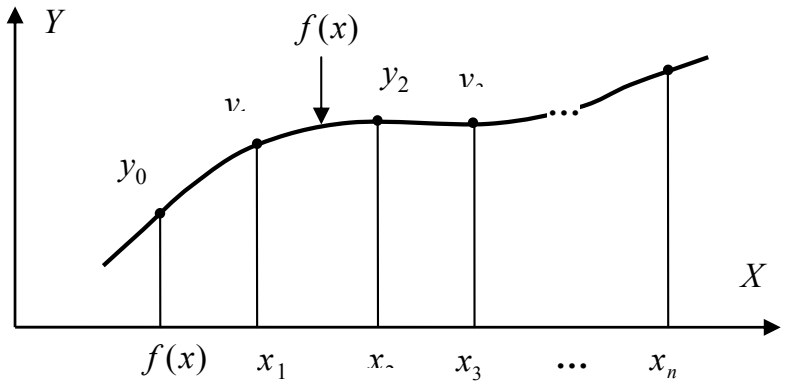


Рис. 14.1

Сформулированная задача называется **задачей интерполяции**. Функция $f(x)$ называется **интерполирующей** функцией. Точки $x_0, x_1, x_2, \ldots, x_n$ называются **узлами интерполяции**.

Разумеется, задача интерполяции может иметь решение (и при том не единственное) или не иметь его вовсе. Все зависит от заданного класса функций.

Множество точек (x_i, y_i) можно рассматривать как набор дискретных значений некоторой неизвестной функции $y = \varphi(x)$. В этом случае функцию $\varphi(x)$ называют **интерполируемой** функцией.

Полученную функцию $f(x)$ обычно используют в случае приближенного вычисления значений функции $\varphi(x)$ для значений аргумента x, отличных от узлов интерполяции. Процесс вычисления значений функции $\varphi(x)$ в точках, отличных от узлов интерполяции, называется **интерполированием** функции $\varphi(x)$.

При этом различают **интерполирование в узком смысле**, если $x \in [x_0, x_n]$, и **экстраполирование**, когда $x \notin [x_0, x_n]$.

Замена функции $\varphi(x)$ ее интерполирующей функцией $f(x)$ может потребоваться не только тогда, когда известна лишь таблица ее значений (например, полученная в результате эксперимента), но и в том случае, когда аналитическое выражение для функции

$\varphi(x)$ известно, однако является слишком сложным и неудобным для дальнейших математических преобразований (интегрирования, дифференцирования и т. д.).

14.3. Интерполяционный полином Лагранжа

Будем искать функцию $f(x)$ в виде полинома заданной степени n – **интерполяционного полинома**, т. е. положим, что

$$f(x) = P_n(x),$$

где

$$P_n(x) = a_0 + a_1 x + a_2 x^2 + \ldots + a_n x^n, \qquad (14.1)$$

и в соответствии с определением интерполирующей функции

$$P_n(x_i) = y_i, \quad i = 0, 1, 2, \ldots, n. \qquad (14.2)$$

Поскольку степень полинома известна, то поиск полинома сводится к нахождению набора его коэффициентов $a_0, a_1, ..., a_n$.

Воспользовавшись выражением (14.1) для искомого полинома $P_n(x)$ и условием (14.2), получим систему линейных уравнений относительно неизвестных коэффициентов $a_0, a_1, ..., a_n$.

$$\begin{cases} a_0 + a_1 x_0 + a_2 x_0^2 + ... + a_n x_0^n = y_0, \\ a_0 + a_1 x_1 + a_2 x_1^2 + ... + a_n x_1^n = y_1, \\ \ldots \\ a_0 + a_1 x_n + a_2 x_n^2 + ... + a_n x_n^n = y_n. \end{cases} \qquad (14.3)$$

Систему уравнений (14.3) представим в матричном виде, введя соответствующие обозначения:

$$S = \begin{pmatrix} 1 & x_0 & x_0^2 & .... & x_0^n \\ 1 & x_1 & x_1^2 & .... & x_1^n \\ \ldots & & & & \\ 1 & x_n & x_n^2 & .... & x_n^n \end{pmatrix} \quad Y = \begin{pmatrix} y_0 \\ y_1 \\ \ldots \\ y_n \end{pmatrix} \quad A = \begin{pmatrix} a_0 \\ a_1 \\ \ldots \\ a_n \end{pmatrix}.$$

В результате система уравнений (14.3) приобретает вид

$$SA = Y. \qquad (14.4)$$

Откуда

$$A = S^{-1}Y. \qquad (14.5)$$

Отметим, что задача интерполяции в описанной выше постановке имеет единственное решение. Это означает, что не существует двух различных наборов коэффициентов a_0, $a_1, ..., a_n$, удовлетворяющих условию (14.2). Это следует из того, что искомые коэффициенты полинома определяются как решение системы линейных уравнений, имеющей единственное решение.

Французский математик **Ж. Л. Лагранж** предложил способ построения интерполяционного полинома без предварительного вычисления коэффициентов a_0, $a_1, ..., a_n$, т. е. без решения системы уравнений (14.3).

Будем искать интерполяционный полином, который в данном случае обозначим через $L_n(x)$, в виде

$$L_n(x) = A_0(x-x_1)(x-x_2)...(x-x_n) + A_1(x-x_0)(x-x_2)...(x-x_n) + ...$$
$$+A_i(x-x_0)(x-x_1)...(x-x_{i-1})(x-x_{i+1})...(x-x_n) + ...$$
$$+A_n(x-x_0)(x-x_1)...(x-x_{n-1}) = \sum_{i=0}^{n} A_i \prod_{\substack{j=0 \\ j\neq i}}^{n} (x-x_j). \qquad (14.6)$$

Неизвестные коэффициенты A_i $(i = 0, 1, 2, \ldots, n)$ определим из условия $L_n(x_i) = f(x_i) = y_i$.

Последовательно полагая $x = x_i$, $i = 0, 1, 2, \ldots, n$, получим

$$A_i(x_i - x_0)(x_i - x_1)...(x_i - x_{i-1})(x_i - x_{i+1})...(x_i - x_n) = y_i,$$

$$A_i = \frac{y_i}{(x_i - x_0)(x_i - x_1)...(x_i - x_{i-1})(x_i - x_{i+1})...(x_i - x_n)}.$$

Подставляя найденные значения коэффициентов A_i $(i = 0, 1, 2, \ldots, n)$ в выражение (14.6) для многочлена $L_n(x)$, получим

$$L_n(x) = \sum_{i=0}^{n} y_i \frac{(x-x_0)(x-x_1)...(x-x_{i-1})(x-x_{i+1})...(x-x_n)}{(x_i-x_0)(x_i-x_1)...(x_i-x_{i-1})(x_i-x_{i+1})...(x_i-x_n)}. \qquad (14.7)$$

Полученный таким образом полином называется интерполяционным полиномом Лагранжа.

Следует отметить, что полином Лагранжа $L_n(x)$ просто представляет собой другую форму записи рассмотренного ранее полинома $P_n(x)$, что следует из единственности решения задачи интерполяции.

Выражение для полинома Лагранжа $L_n(x)$ может быть легко преобразовано к виду $P_n(x)$ путем группировки коэффициентов при соответствующих степенях аргумента x.

Примеры.

Пусть имеем две точки (x_0, y_0) и (x_1, y_1), что соответствует значению $n = 1$. Тогда

$$L_1(x) = y_0 \frac{(x - x_1)}{(x_0 - x_1)} + y_1 \frac{(x - x_0)}{(x_1 - x_0)} = \frac{y_0 x_1 + y_1 x_0}{x_0 - x_1} + \frac{y_0 - y_1}{x_0 - x_1} x .$$

Обозначая $a_0 = \frac{y_0 x_1 + y_1 x_0}{x_0 - x_1}$ и $a_1 = \frac{y_0 - y_1}{x_0 - x_1} x$, получаем

$$L_1(x) = a_0 + a_1 x = P_1(x) .$$

Пусть имеем три точки $(x_0, y_0) = (1,2)$, $(x_1, y_1) = (2,8)$ и $(x_2, y_2) = (3,6)$, что соответствует значению $n = 2$. Тогда

$$L_2(x) = y_0 \frac{(x - x_1)(x - x_2)}{(x_0 - x_1)(x_0 - x_2)} + y_1 \frac{(x - x_0)(x - x_2)}{(x_1 - x_0)(x_1 - x_2)} + y_2 \frac{(x - x_0)(x - x_1)}{(x_2 - x_0)(x_2 - x_1)} =$$

$$= 2\frac{(x-2)(x-3)}{(1-2)(1-3)} + 8\frac{(x-1)(x-3)}{(2-1)(2-3)} + 6\frac{(x-1)(x-2)}{(3-1)(3-2)} =$$

$$= (x-2)(x-3) - 8(x-1)(x-3) + 3(x-1)(x-2) = -4x^2 + 18x - 12 .$$

Достоинства полинома Лагранжа:

– график интерполяционного многочлена Лагранжа проходит через каждую точку массива;

– конструируемая функция легко описывается (число подлежащих определению коэффициентов интерполяционного многочлена Лагранжа на сетке n равно $n + 1$);

– построенная функция имеет непрерывные производные любого порядка;

– заданным массивом интерполяционный многочлен определен однозначно.

Недостатки полинома Лагранжа:

– степень интерполяционного многочлена Лагранжа зависит от числа узлов сетки, и чем больше это число, тем выше степень интерполяционного многочлена и, значит, тем больше требуется вычислений;

– изменение хотя бы одной точки в массиве требует полного пересчета коэффициентов интерполяционного многочлена Лагранжа.

14.4. Интерполяция сплайнами

Сплайн (англ. *spline* – планка, рейка) – функция, область определения которой разбита на конечное число отрезков, на каждом из которых сплайн совпадает с некоторым алгебраическим полиномом. Сплайны имеют многочисленные применения как в математической теории, так и в разнообразных вычислительных приложениях. В частности, сплайны двух переменных интенсивно используются для задания поверхностей в различных системах компьютерного моделирования.

14.4.1. Определение сплайна

Пусть на отрезке $[a, b]$ задана упорядоченная система (массив) несовпадающих точек x_i ($i = 0, 1, \ldots, n$).

Определение. **Сплайном** $S_m(x)$ называется определенная на $[a, b]$ функция, принадлежащая классу $C^l[a, b]$ l раз непрерывно дифференцируемых функций, такая, что на каждом промежутке $[x_{i-1}, x_i]$, $i = 0, 1, \ldots, n$, она – многочлен m-й степени. Разность $d = m - l$ между степенью сплайна m и показателем его гладкости l называется **дефектом** сплайна.

Если сплайн $S_m(x)$ строится по некоторой функции $f(x)$ так, чтобы выполнялись условия $S_m(x_i) = f(x_i)$, то такой сплайн называется **интерполяционным сплайном** для функции $f(x)$; при этом узлы сплайна, вообще говоря, могут не совпадать с узлами интерполяции x_k.

14.4.2. Интерполяционный кубический сплайн

Рассмотрим наиболее известный и широко применяемый интерполяционный сплайн степени 3 дефекта 1. При этом будем исходить из предположения, что узлы сплайна

$$a = x_0,\ x_1,\ \ldots,\ x_n = b \tag{14.8}$$

одновременно служат узлами интерполяции, т. е. в них известны значения функции $f_i = f(x_i)$, $i = 0, 1, \ldots, n$.

Определение. Кубическим сплайном дефекта 1, интерполирующим на отрезке $[a, b]$ данную функцию $f(x)$, называется функция

$$g(x) = \{g_i(x),\ \text{при}\ x \in [x_{i-1}, x_i]\}_{i=1}^{n}, \tag{14.9}$$

где

$$g_i(x) = a_i + b_i(x - x_i) + c_i(x - x_i)^2 + d_i(x - x_i)^3 \tag{14.10}$$

удовлетворяет совокупности условий:

$$g(x_i) = f(x_i) - \qquad (14.11)$$

условие интерполяции в узлах сплайна;

$$g(x) \in C^2_{[a,\,b]} - \qquad (14.12)$$

двойная непрерывная дифференцируемость;

$$g''(a) = g''(b) = 0 - \qquad (14.13)$$

краевые (граничные) условия.

Заметим, что граничные условия вида (14.13) называются естественными граничными условиями.

Определенный таким образом сплайн называют еще **естественным** или **чертежным сплайном** и связано это со следующим обстоятельством. Желая провести плавную линию через заданные точки плоскости, чертежники фиксировали в этих точках гибкую упругую рейку, тогда под влиянием упругих сил она принимала нужную форму, обеспечивающую минимум потенциальной энергии.

Для построения по данной функции $f(x)$ интерполирующего ее сплайна (14.9) нужно найти $4n$ его коэффициентов a_i, b_i, c_i, d_i $(i = 1, 2, 3, \ldots, n)$.

Имеем:

из условий интерполяции (14.11) для функции

$$g_1(x_0) = f_0, \;\; g_i(x_i) = f_i = a_i \text{ при } i = 1, 2, \ldots, n, \qquad (14.14)$$

из условий гладкой стыковки звеньев сплайна (14.12)

$$\begin{cases} g_{i-1}(x_{i-1}) = g_i(x_{i-1}), \\ g'_{i-1}(x_{i-1}) = g'_i(x_{i-1}), \;\; \text{при} \;\; i = 2, 3, \ldots, n \\ g''_{i-1}(x_{i-1}) = g''_i(x_{i-1}), \end{cases} \qquad (14.15)$$

из краевых условий (14.13)

$$g''(x_0) = g''(x_n) = 0. \qquad (14.16)$$

Подставляя сюда выражения (14.9) для функций $g_i(x)$

$$g_i(x) = a_i + b_i(x - x_i) + c_i(x - x_i)^2 + d_i(x - x_i)^3,$$

их производных

$$g'_i(x) = b_i + 2c_i(x - x_i) + 3d_i(x - x_i)^2 \qquad (14.17)$$

и

$$g_i''(x) = 2c_i + 6d_i(x - x_i) \qquad (14.18)$$

через коэффициенты a_i, b_i, c_i, d_i при указанных значениях i и полагая для краткости

$$h_i = x_i - x_{i-1}, \qquad (14.19)$$

можно получить систему уравнений для определения неизвестных коэффициентов [7].

Полученная система решается методом прогонки [7].

Рассмотрим случай равноотстоящих узлов. Тогда $h_k = h$ и расчетные формулы для коэффициентов полинома [7] принимают вид:

$$\delta_1 = -\frac{1}{4}, \quad \lambda_1 = \frac{3}{4h^2}(f_2 - 2f_1 + f_0).$$

Для $i = 3, 4, \ldots, n$ вычисляем

$$\delta_{i-1} = -\frac{1}{4 + \delta_{i-2}}, \qquad \lambda_{i-1} = \frac{\frac{3}{h^2}(f_0 - 2f_{i-1} + f_{i-2}) - \lambda_{i-2}}{4 + \delta_{i-2}}.$$

Полагаем $c_n = 0$ и для $i = n, n-1, \ldots, 2$ вычисляем

$$c_{i-1} = \delta_{i-1}c_i + \lambda_{i-1}.$$

Полагаем $i = 1, 2, \ldots, n$ и с учетом $c_n = 0$ получаем

$$b_i = \frac{f_i - f_{i-1}}{h} + \frac{h}{3}(2c_i + c_{i-1}), \qquad d_i = \frac{c_i - c_{i-1}}{3h}.$$

В результате при $x \in [x_{i-1}, x_i]$ значение $f(x)$ можно заменить значением

$$g_i(x) = f_i + b_i(x - x_i) + c_i(x - x_i)^2 + d_i(x - x_i)^3$$

с найденными значениями коэффициентов b_i, c_i, d_i.

Достоинства кубической сплайн-интерполяции:

– график построенной функции проходит через каждую точку массива;

– конструируемая функция сравнительно легко описывается (число подлежащих определению коэффициентов равно $4n$);

– заданным массивом построенная функция определена однозначно;

– степень многочленов не зависит от числа узлов сетки и, следовательно, не изменяется при его увеличении;

– построенная функция имеет непрерывные первые и вторые производные;

– построенная функция обладает хорошими аппроксимационными свойствами.

К недостаткам кубических сплайнов относится то, что они склонны осциллировать в окрестностях точки, существенно отличающейся от своих соседей.

14.5. Геометрические сплайны

Во многих задачах требование того, чтобы конструируемая кривая однозначно проектировалась соответственно на прямую, является слишком жестким. Расширяя допустимые классы кривых, естественно обратиться и к более общему способу описания их частичных фрагментов [5]. В качестве нового способа задания кривых удобно использовать параметрический способ.

Формулировка задачи: по заданному множеству вершин $P = \{P_0, P_1, \ldots, P_{m-1}, P_m\}$ с учетом их нумерации построить гладкую кривую, которая, плавно изменяясь, последовательно проходила бы вблизи этих вершин и удовлетворяла некоторым дополнительным условиям. Эти условия могут иметь различный характер. Например, можно потребовать, чтобы искомая кривая проходила через все заданные вершины или, проходя через заданные вершины, касалась заданных направлений, являлась замкнутой или имела заданную регулярность и т. п.

При отыскании подходящего решения задачи приближения важную роль играет ломаная, звенья которой соединяют соседние вершины заданного набора. Эту ломаную называют **контрольной** или **опорной**, а ее вершины – контрольными или опорными.

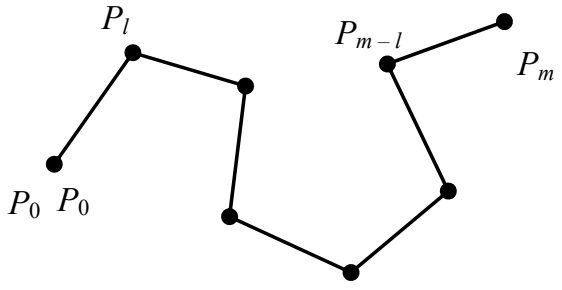


Рис. 14.2

Во многих случаях она довольно точно показывает, как будет проходить искомая кривая, что особенно полезно при решении задачи сглаживания. Каждая вершина заданного массива является либо **внутренней**, либо **граничной** (концевой). В массиве P вершины $P_1, \ldots, P_{m-1}$ внутренние, а вершины P_0 и P_m – граничные (концевые).

Никаких ограничений на множество вершин не накладывается – они могут быть заданы как на плоскости, так и в пространстве, их взаимное расположение может быть совершенно произвольным, некоторые из вершин могут совпадать и т. д. Поэтому описание нужной кривой ищут в следующем виде:

$$R(t) = \sum a_i(t) P_i , \qquad (14.20)$$

где $a_i(t)$ – некоторые функциональные коэффициенты, подлежащие определению.

Если количество вершин в заданном множестве P достаточно велико, то найти универсальные функциональные коэффициенты, как правило, довольно затруднительно. Если универсальные коэффициенты $a_i(t)$ все же найдены, то часто оказывается, что они наряду с нужными свойствами обладают и такими, которые не всегда удовлетворительно согласуются с ожидаемым поведением соответствующей кривой (например, кривая, описываемая уравнением (14.20) с этими коэффициентами, может осциллировать или заметно отклоняться от заданного множества).

Для успешного решения поставленной задачи приближения весьма удобно привлечь кривые, составленные из элементарных фрагментов. В случае, когда эти элементарные фрагменты строятся по единой сравнительно простой схеме, такие составные кривые принято называть сплайновыми кривыми.

Параметрические уравнения каждого элементарного фрагмента ищутся в виде (14.20) с той лишь разницей, что всякий раз привлекается только часть заданных вершин множества P. Для описания элементарных кривых и вычисления их геометрических характеристик (информация о которых необходима при состыковке) в качестве функциональных коэффициентов обычно используются многочлены невысоких степеней (2-й или 3-й), в первую очередь потому, что они сравнительно просто вычисляются. Наибольшее распространение получили методы конструирования составных кривых, в которых используются кубические многочлены.

14.5.1. Кривая Безье

Разработана математиком **Пьером Безье**. Кривые и поверхности Безье были использованы в 1960-х годах компанией «Рено» для компьютерного проектирования формы кузовов автомобилей. В настоящее время они широко используются в компьютерной графике.

Кривые Безье описываются в параметрической форме [2, 5]:

$$\begin{cases} x = P_x(t), \\ y = P_y(t). \end{cases} \tag{14.21}$$

Значение t выступает как параметр, которому отвечают координаты отдельной точки линии. Параметрическая форма описания может быть более удобной для некоторых кривых, чем задание в виде функции $y = f(x)$. Это потому, что функция $f(x)$ может быть намного сложнее, чем $P_x(t)$ и $P_x(t)$, кроме того, $f(x)$ может быть неоднозначной.

Многочлены Безье для $P_x(t)$ и $P_x(t)$ имеют такой вид:

$$\begin{cases} P_x(t) = \sum_{i=0}^{m} C_m^i t^i (1-t)^{m-i} x_i, \\ P_y(t) = \sum_{i=0}^{m} C_m^i t^i (1-t)^{m-i} y_i, \end{cases} \tag{14.22}$$

где $C_m^i = \dfrac{m!}{i!(m-i)!}$ – сочетание m по i; x_i и y_i – координаты точек-ориентиров P_i. Значение m можно рассматривать и как степень полинома, и как значение, которое на единицу меньше количества точек-ориентиров.

Рассмотрим кривые Безье, классифицируя их по значениям m.

1. $\boldsymbol{m = 1}$ (по двум точкам).

Кривая вырождается в отрезок прямой линии, определяемой концевыми точками (рис. 14.3):

$$P(t) = (1-t)P_0 + tP_1, \tag{14.23}$$

где $P = (P_x, \quad P_y)^T$.

2. $\boldsymbol{m = 2}$ (по трем точкам, рис. 14.4):

$$P(t) = (1-t)^2 P_0 + 2t(1-t)P_1 + t^2 P_2. \tag{14.24}$$

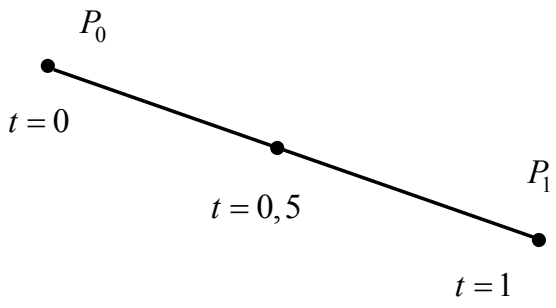


Рис. 14.3

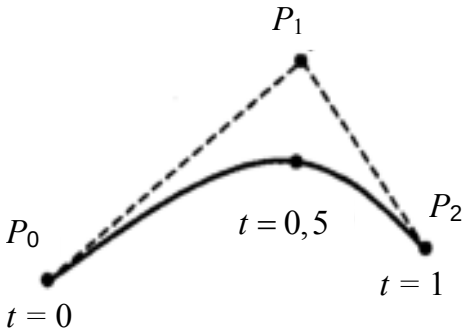
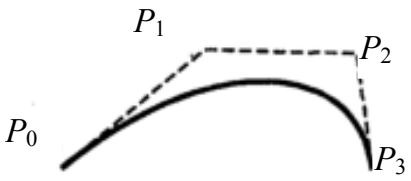


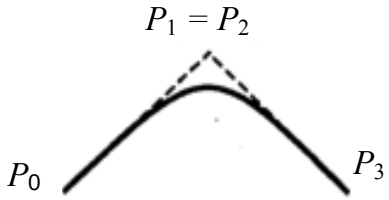
Рис. 14.4

3. ***m* = 3** (по четырем точкам, рис. 14.5 *а*, *б*):

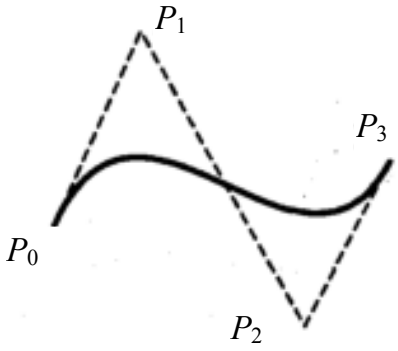
$$P(t) = (1-t)^3 P_0 + 3t(1-t)^2 P_1 + 3t^2(1-t)P_2 + t^3 P_3 . \quad (14.25)$$



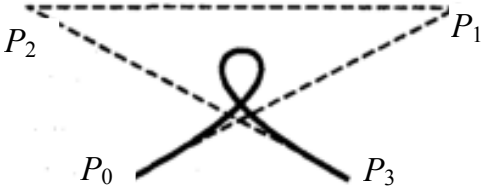
а



б



в



г

Рис. 14.5

14.5.2. Геометрический алгоритм для кривой Безье

Этот алгоритм позволяет вычислить координаты (x, y) точки кривой Безье по значению параметра t.

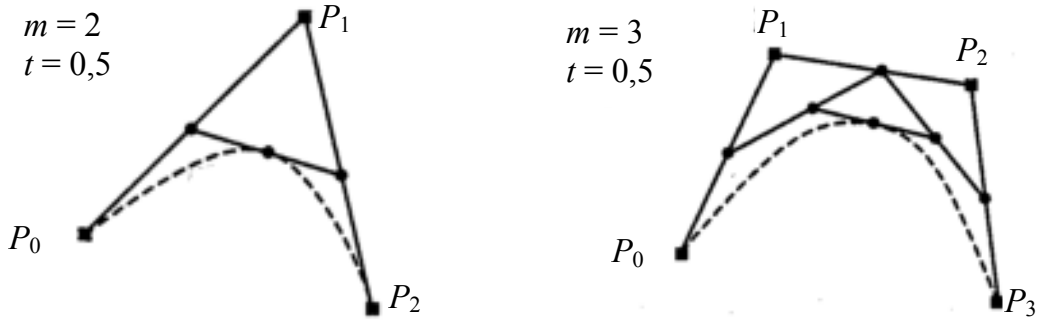
Суть алгоритма заключается в следующем:

– каждая сторона контура многоугольника, проходящего по точкам-ориентирам, делится пропорционально значению t;

– точки деления соединяются отрезками прямых и образуют новый многоугольник. Количество узлов нового контура на единицу меньше, чем количество узлов предыдущего контура;

– стороны нового контура снова делятся пропорционально значению t. И так далее.

Это продолжается до тех пор, пока не будет получена единственная точка деления. Эта точка и будет точкой кривой Безье (рис. 14.6, *а*, *б*).



а *б*

Рис. 14.6

Приведем запись геометрического алгоритма на языке C++:

```
for(i = 0; i <= m; i++)
R[i] = P[i];  // Формируем вспомогательный массив R
for(j = m; j > 0; j--)
for(i = 0; i < j; i++)
  R[i] = R[i] + t * (R[i+1] - R[i])
```

Результат работы алгоритма – координаты одной точки кривой Безье – записываются в R[0].

Глава 15

ГРАФИЧЕСКАЯ БИБЛИОТЕКА OPENGL

15.1. Общие сведения

OpenGL является одним из самых популярных прикладных программных интерфейсов (API – Application Programming Interface) для разработки приложений в области двумерной и трехмерной графики [9–11]. Стандарт OpenGL (Open Graphics Library – открытая графическая библиотека) был разработан и утвержден в 1992 году ведущими фирмами в области разработки программного обеспечения как эффективный аппаратно-независимый интерфейс, пригодный для реализации на различных платформах. Основой стандарта стала библиотека IRISGL, разработанная фирмой Silicon Graphics Inc. Библиотека насчитывает около 120 различных команд, которые программист использует для задания объектов и операций, необходимых для написания интерактивных графических приложений. На сегодняшний день графическая система OpenGL поддерживается большинством производителей аппаратных и программных платформ. Эта система доступна тем, кто работает в среде Windows, а также пользователям компьютеров Apple. Характерными особенностями OpenGL, которые обеспечили распространение и развитие этого графического стандарта, являются:

– **стабильность**: дополнения и изменения в стандарте реализуются таким образом, чтобы сохранить совместимость с разработанным ранее программным обеспечением;

– **надежность и переносимость**: приложения, использующие OpenGL, гарантируют одинаковый визуальный результат вне зависимости от типа используемой операционной системы и организации отображения информации. Кроме того, эти приложения могут выполняться как на персональных компьютерах, так и на рабочих станциях и суперкомпьютерах;

– **легкость применения**: стандарт OpenGL имеет продуманную структуру и интуитивно понятный интерфейс, что позволяет с меньшими затратами создавать эффективные приложения, содержащие меньше строк кода, чем с использованием других графических библиотек. Необходимые функции для обеспечения совместимости с различным оборудованием реализованы на уровне библиотеки и значительно упрощают разработку приложений.

15.2. Основные возможности

Описать возможности OpenGL можно через функции его библиотеки. Все функции можно разделить на пять категорий:

1. **Функции описания примитивов** определяют объекты нижнего уровня иерархии (примитивы), которые способна отображать графическая подсистема. В OpenGL в качестве примитивов выступают точки, линии, многоугольники и т. д.

2. **Функции описания источников света** служат для описания положения и параметров источников света, расположенных в трехмерной сцене.

3. **Функции задания атрибутов** дают возможность программисту определять, как будут выглядеть на экране отображаемые объекты. Другими словами, если с помощью примитивов определяется, **что** появится на экране, то атрибуты определяют **способ** вывода на экран. В качестве атрибутов OpenGL позволяет задавать цвет, характеристики материала, текстуры, параметры освещения.

4. **Функции визуализации** позволяют задать положение наблюдателя в виртуальном пространстве, параметры объектива камеры. Зная эти параметры, система сможет не только правильно построить изображение, но и отсечь объекты, оказавшиеся вне поля зрения.

5. **Функции геометрических преобразований** позволяют программисту выполнять различные преобразования объектов – поворот, перенос, масштабирование. При этом OpenGL может выполнять дополнительные операции, такие как использование сплайнов для построения линий и поверхностей, удаление невидимых фрагментов изображений на уровне пикселов и т. д.

15.3. Интерфейс OpenGL

OpenGL состоит из набора библиотек. Все базовые функции хранятся в основной библиотеке, для обозначения которой используется аббревиатура **GL**. Помимо основной, OpenGL включает в себя несколько дополнительных библиотек.

Первая из них – библиотека утилит GL (GLU – GL Utility). Все функции этой библиотеки определены через базовые функции GL. В состав GLU вошла реализация более сложных функций, таких как набор популярных геометрических примитивов (куб, шар, цилиндр, диск), функции построения сплайнов, реализация дополнительных операций над матрицами и т. п.

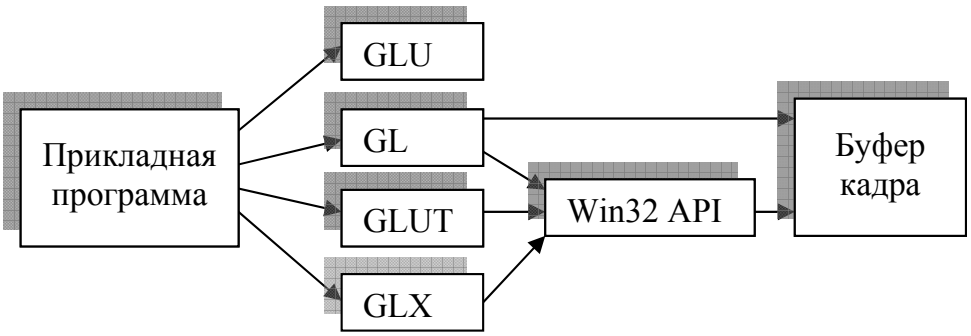


Рис. 15.1

OpenGL не включает в себя никаких специальных команд для работы с окнами или ввода информации от пользователя. Поэтому были созданы специальные переносимые библиотеки для обеспечения часто используемых функций взаимодействия с пользователем и для отображения информации с помощью оконной подсистемы. Наиболее популярной является библиотека **GLUT** (GL Utility Toolkit). Формально GLUT не входит в OpenGL, но фактически включается почти во все его дистрибутивы и имеет реализации для различных платформ. GLUT предоставляет только минимально необходимый набор функций для создания OpenGL-приложения. Существует также функционально аналогичная GLUT библиотека GLX. Однако она менее популярна.

Кроме того, функции, специфичные для конкретной оконной подсистемы, обычно входят в ее прикладной программный интерфейс. Так, функции, поддерживающие выполнение OpenGL, есть в составе Win32 APIи Window. На рис. 15.1 схематически представлена организация системы библиотек в версии, работающей под управлением ОС Windows. Аналогичная организация используется и в других версиях OpenGL.

15.4. Архитектура OpenGL

Функции OpenGL реализованы в модели клиент – сервер. Приложение выступает в роли клиента – оно вырабатывает команды, а сервер OpenGL интерпретирует и выполняет их. Сам сервер может находиться как на том же компьютере, на котором находится клиент (например, в виде динамически загружаемой библиотеки – DLL), так и на другом (при этом может быть использован специальный протокол передачи данных между машинами).

GL обрабатывает и рисует в буфере кадра графические примитивы с учетом некоторого числа выбранных режимов. Каждый

примитив – это точка, отрезок, многоугольник и т. д. Каждый режим может быть изменен независимо от других. Определение примитивов, выбор режимов и другие операции описываются с помощью команд в форме вызовов функций прикладной библиотеки.

Примитивы определяются набором из одной или более вершин (vertex). Вершина определяет точку, конец отрезка или угол многоугольника. С каждой вершиной ассоциируются некоторые данные (координаты, цвет, нормаль, текстурные координаты и т. д.), называемые атрибутами. В подавляющем большинстве случаев каждая вершина обрабатывается независимо от других.

С точки зрения архитектуры графическая система OpenGL является конвейером (рис. 15.2), состоящим из нескольких последовательных этапов обработки графических данных.

Рис. 15.2

Команды OpenGL всегда обрабатываются в том порядке, в котором они поступают, хотя могут происходить задержки перед тем, как проявится эффект от их выполнения. В большинстве случаев OpenGL предоставляет непосредственный интер-

фейс, т. е. определение объекта вызывает его визуализацию в буфере кадра.

С точки зрения разработчиков, OpenGL – это набор команд, которые управляют использованием графической аппаратуры. Если аппаратура состоит только из адресуемого буфера кадра, тогда OpenGL должен быть реализован полностью с использованием ресурсов центрального процессора. Обычно графическая аппаратура предоставляет различные уровни ускорения: от аппаратной реализации вывода линий и многоугольников до изощренных графических процессоров с поддержкой различных операций над геометрическими данными.

OpenGL является прослойкой между аппаратурой и пользовательским уровнем, что позволяет предоставлять единый интерфейс на разных платформах, используя возможности аппаратной поддержки.

Кроме того, OpenGL можно рассматривать как конечный автомат, состояние которого определяется множеством значений специальных переменных и значениями текущей нормали, цвета, координат текстуры и других атрибутов и признаков. Вся эта информация будет использована при поступлении в графическую систему координат вершины для построения фигуры, в которую она входит. Смена состояний происходит с помощью команд, которые оформляются как вызовы функций.

15.5. Синтаксис команд

Определения команд GL находятся в файле gl.h, для включения которого нужно написать (для Windows 7)

#include<gl.h>

Для работы с библиотекой GLU нужно аналогично включить файл glu.h. Версии этих библиотек, как правило, включаются в дистрибутивы систем программирования, например Microsoft Visual Studio, Borland Delphi.

В отличие от стандартных библиотек, пакет GLUT нужно инсталлировать и подключать отдельно.

Все команды (процедуры и функции) библиотеки GL начинаются с префикса **gl**, все константы – с префикса **GL_**. Соответствующие команды и константы библиотек GLUи GLUT аналогично имеют префиксы **glu** (GLU_) и **glut** (GLUT_). Кроме того, в имена команд входят суффиксы, несущие информацию о

числе и типе передаваемых параметров. В OpenGL полное имя команды имеет вид:

```
type LibNameCommandName[1 2 3 4][b s i f d ub us ui][v]
(type1 arg1,…,typeNargN),
```

где **type** – тип возвращаемого функцией значения;

LibNameCommandName – имя команды, которое состоит из двух частей: **LibName** и **CommandName**;

LibName – имя библиотеки, в которой описана эта функция. Для базовых функций OpenGL, функций из библиотек GL, GLU, GLUT, GLAUX это **gl**, **glu**, **glut**, **aux** соответственно;

CommandName – имя команды (процедуры или функции),

[1 2 3 4] – число аргументов команды;

[b s i f d ub us ui] – тип аргумента, символ.

Соответствие типов OpenGL и языка C(C++) приведено в табл. 15.1.

Таблица 15.1

Типы OpenGL и языка C(C++)

Символ	Описание	Тип OpenGL	Тип C(C++)
b	8-битовое целое	GLbyte	signed char
s	16-битовое целое	GLshort	short
i	32-битовое целое	GLint Gsizei	long
f	32-битовое число с плавающей точкой	GLfloat GLclampf	float
d	64-битовое число с плавающей точкой	GLdouble GLclampf	double
ub	8-битовое беззнаковое целое	GLubyte GLboolean	unsigned char
us	16-битовое беззнаковое целое	GLushort	unsigned short
ui	32-битовое беззнаковое целое	GLuint GLenum GLbitfield	unsigned long

Наличие символа **h [v]** показывает, что в качестве параметров функции используется указатель на массив значений. Символы в квадратных скобках в некоторых названиях не используются. Например, команда **glVertex2i()**, описанная в библиотеке GL, использует в качестве параметров два целых числа, а команда **glColor3fv()** использует в качестве параметра указатель на массив из трех вещественных чисел. Использования нескольких вариан-

тов каждой команды можно частично избежать, применяя перегрузку функций языка C++. Но интерфейс OpenGL не рассчитан на конкретный язык программирования, и, следовательно, должен быть максимально универсален.

15.6. Структура GLUT-приложения

Далее будем рассматривать построение консольного приложения при помощи библиотеки GLUT. Эта библиотека обеспечивает единый интерфейс для работы с окнами вне зависимости от платформы, поэтому описываемая ниже структура приложения остается неизменной для операционных систем Windows, Linux и других.

Функции GLUT могут быть классифицированы на несколько групп по своему назначению:

- инициализация;
- начало обработки событий;
- управление окнами;
- управление меню;
- регистрация функций с обратным вызовом;
- управление индексированной палитрой цветов;
- отображение шрифтов;
- отображение дополнительных геометрических фигур (тор, конус и др.).

Инициализация проводится с помощью функции:

void glutInit (int *argcp, char **argv),

где переменная **argcp** – указатель на стандартную переменную **argc**, описываемую в функции main(), а **argv** – указатель на параметры, передаваемые программе при запуске, который описывается там же. Эта функция проводит необходимые начальные действия для построения окна приложения, и только несколько функций GLUT могут быть вызваны до нее. К ним относятся:

void glutInitWindowPosition (int x, int y) – служит для установки положения окна OpenGL на экране, здесь **x** – это X-координата **левого верхнего** угла окна, **y** – это Y-координата **левого верхнего** угла окна. Координаты **x** и **y** определяются в пикселах;

void glutInitWindowSize (int width, int height) – служит для установки размеров окна на экране, здесь **width –** ширина окна, **height** – высота окна. Значения **width** и **height** определяются в пикселах.

Если вызов этих функций опустить, то система присвоит параметрам окна значения по умолчанию;

void glutInitDisplayMode (unsigned int mode) – задает режим отображения, где **mode** определяет различные режимы отображения информации, которые могут совместно задаваться с использованием операции побитового «ИЛИ» («|»). Значения параметра представлены в табл. 15.2.

Таблица 15.2

Значения параметра mode

Значение	Комментарий
GLUT_RGBA	Выбор режима RGBA
GLUT_RGB	То же, что и GLUT_RGBA
GLUT_INDEX	Палитровый видеорежим
GLUT_SINGLE	Использование одинарного видеобуфера
GLUT_DOUBLE	Использование двойной буферизации, применяется для создания анимации
GLUT_ACCUM	Создание аккумулирующего буфера
GLUT_ALPHA	Создание окна с α-каналом
GLUT_DEPTH	Создание окна с буфером глубины
GLUT_SPENCIL	Создание окна с буфером трафарета

После инициализации можно создать окно, в которое будет происходить вывод при помощи функции

int glutCreateWindow (char * name),

где параметр **name** задает заголовок окна. Функция возвращает целочисленный идентификатор окна.

Функции библиотеки GLUT реализуют так называемый событийно-управляемый механизм. Это означает, что есть некоторый внутренний цикл, который запускается после соответствующей инициализации и обрабатывает одно за другим все события, объявленные во время инициализации. К событиям относятся: **щелчок мыши**, **закрытие окна**, **изменение свойств окна**, **передвижение курсора**, **нажатие клавиши** и «пустое» (**idle**) событие, когда ничего не происходит. Для проведения периодической проверки совершения того или иного события надо зарегистрировать функцию, которая будет его обрабатывать. Для этого используются следующие функции:

void glutDisplayFunc(void (*func) (void)) – служит для установки обработчика события, связанного с необходимостью перерисовки содержимого окна. Параметр **func** задает функцию рисо-

вания для окна приложения, которая вызывается при необходимости создания или восстановления изображения;

void glutPostRedisplay (void) – служит для явного указания, что окно надо обновить;

void glutReshapeFunc(void (*func) (int width, int height)) – служит для установки обработчика события, связанного с изменением размера окна. Параметр **func** задает функцию, которая будет вызываться при изменении размеров окна перед вызовом функции отрисовки содержимого окна. Параметры **width** и **height** определяют *новые* значения ширины и высоты окна в пикселах;

void glutKeyboardFunc(void (*func)(unsigned char key, int x, int y)) – служит для установки обработчика сообщений от клавиатуры. Параметр **func** задает функцию, которая будет вызываться в ответ на сообщения клавиатуры, параметр **key** содержит ASCII-код символа, а параметры **x** и **y** – координаты курсора мыши в пикселах по отношению к верхнему левому углу окна.

Обработчик клавиш, не генерирующих ASCII-кода (такие как <Shift>), обрабатываются с помощью функции

void glutSpecialFunc(void(*func)(int key, int x, int y)),

где параметр key несет в себе информацию о нажатой клавише и может принимать одно из значений, указанных в табл. 15.3;

Таблица 15.3

Значения параметра key

Значение key	Клавиша	Значение key	Клавиша
GLUT_KEY_F1	<F1>	GLUT_KEY_F12	<F12>
GLUT_KEY_F2	<F2>	GLUT_KEY_LEFT	<←>
GLUT_KEY_F3	<F3>	GLUT_KEY_RIGHT	<→>
GLUT_KEY_F4	<F4>	GLUT_KEY_UP	<↑>
GLUT_KEY_F5	<F5>	GLUT_KEY_DOWN	<↓>
GLUT_KEY_F6	<F6>	GLUT_KEY_PAGE_UP	<Page Up>
GLUT_KEY_F7	<F7>	GLUT_KEY_PAGE_DOWN	<Page Down>
GLUT_KEY_F8	<F8>	GLUT_KEY_HOME	<Home>
GLUT_KEY_F9	<F9>	GLUT_KEY_END	<End>
GLUT_KEY_F10	<F10>	GLUT_KEY_INSERT	<Insert>
GLUT_KEY_F11	<F11>	–	–

void glutMouseFunc(void (*func) (int button, int state, int x, int y)) – служит для установки обработчика сообщений от мыши. Параметр **func** задает функцию, которая будет вызываться в ответ на сообщения от мыши. Параметр **button** принимает одно из значений

GLUT_LEFT_BUTTON, GLUT_MIDDLE_BUTTON, GLUT_RIGHT_BUTTON и несет в себе информацию о том, какая клавиша мыши была нажата (или отпущена). Параметр **state** принимает одно из значений GLUT_DOWN (клавиша нажата) или GLUT_UP (клавиша отпущена). Параметры **x** и **y** содержат координаты курсора мыши.

Вызов обработчика, установленного **glutMouseFunc**, происходит лишь при нажатии или отпускании клавиши мыши. Если необходимо установить обработчик на перемещение мыши, то для этого следует воспользоваться функциями:

void glutMotionFunc (void (*func)(int x, int y)),

void glutPassiveMotionFunc (void (*func)(int x, int y)).

Первая из них устанавливает обработчик, который будет вызываться при перемещении мыши при **нажатой** клавише (любой из имеющихся) мыши, а вторая – при передвижении мыши в том случае, когда ни одна из клавиш мыши **не нажата**;

int glutCreateMenu(void (*func)(int ItemNamber) – служит для создания меню. В качестве аргумента используется **имя** функции, которая будет вызываться при выборе пунктов меню. Параметр **ItemNamber** этой функции определяет пункт меню, который будет выбран при ее вызове;

glutAddMenuEntry(char* ItemName, int Namber) – служит для добавления пунктов в уже созданное меню. Параметр **ItemName** определяет название пункта меню, а параметр **Namber** – идентификатор пункта, который через параметр **ItemNamber** передается функции, вызываемой при выборе этого пункта меню;

glutAttachMenu(int button) – задает кнопку мыши, которая будет использоваться для активизации меню. Параметр **button** принимает одно из значений GLUT_LEFT_BUTTON, GLUT_MIDDLE_BUTTON, GLUT_RIGHT_BUTTON.

Ниже приводится последовательность команд для создания меню, добавления в него пунктов и активизации с помощью правой клавиши мыши:

```
glutCreateMenu(NaneFunction);
  glutAddMenuEntry("Первый пункт меню",1);
  glutAddMenuEntry("Второй пункт меню",2);
glutAttachMenu(GLUT_RIGHT_BUTTON)
```

В приведенном примере параметр NaneFunction – имя функции, которая будет вызвана при выборе пункта меню;

void glutIdleFunc (void (*func) (void)) – служит для установки обработчика, который будет постоянно вызываться в «фоновом»

режиме. Наиболее часто этот обработчик используется для проведения некоторых вспомогательных действий, которые можно осуществить в моменты времени, когда приложение «ничего не делает»;

void glutMainLoop (void) – служит для запуска бесконечного цикла обработки сообщений. Функция вызывается в конце любой программы, использующей GLUT;

void glutSwapBuffers() – при использовании двойной буферизации служит для смены буферов местами (буфера, содержимое которого видно сейчас на экране, и буфера, в который производится вывод);

int glutGet(GLEnumstate) – служит для получения информации о внутренних переменных glut, параметр **state** может принимать значения, указанные в табл. 15.4.

Таблица 15.4

Значения параметра state

Значение **state**	Комментарий
GLUT_WINDOW_X	Для возврата x-координаты окна
GLUT_WINDOW_Y	Для возврата y-координаты окна
GLUT_WINDOW_WIDTH	Для возврата ширины окна
GLUT_WINDOW_HEIGHT	Для возврата высоты окна
GLUT_ELAPSED_TIME	Для возврата времени в миллисекундах с момента инициализации GLUT

Библиотека OpenGL реализована по модели клиент – сервер, приложение выступает в роли клиента – вырабатывает команды, а OpenGL (сервер) интерпретирует и обрабатывает их. При этом сервер может располагаться как на том же компьютере, что и клиент, так и на другом. Сервер может поддерживать несколько контекстов OpenGL, каждый из которых инкапсулирует состояние OpenGL. Клиент может подключиться к любому из них. По этой причине желательно гарантировать, что выполнение всех вызванных команд завершится либо за определенное, либо за конечное время. Для обеспечения этого в OpenGL реализованы две команды.

Первая из них – **void glFinish() –** блокирует дальнейшее выполнение программы, пока не будут завершены все вызванные перед ней команды OpenGL, включая изменения состояния и содержимого буфера кадра.

Вторая команда – **void gIFlush() –** служит для освобождения буферов OpenGL.

Различные реализации команд OpenGL для работы с буферами размешаются в разных местах, включая сетевые буферы и графические акселераторы. Команда **gIFIush** освобождает все эти буферы, заставляя все вызванные команды выполниться так быстро, как это требуется фактической реализацией воспроизведения. И хотя выполнение не может быть закончено за некоторый заданный отрезок времени, оно завершается за конечный период.

Поскольку любая программа OpenGL может выполняться в сети или на графическом акселераторе, который накапливает команды, необходимо убедится, что функция **gIFIush** вызвана во всех программах, которые требуют, чтобы все их предварительно вызванные команды были закончены. Например, необходимо вызвать эту команду перед ожиданием пользовательского ввода, который зависит от сформированного образа.

Команда **gIFIush**, в отличие от **glFinish**, не ожидает завершения выполнения всех предварительно вызванных команд.

Мы рассмотрели только малую часть команд, составляющих библиотеку GLUT. Полное описание GLUT, а также сами файлы для работы с ней можно найти на официальном сайте библиотеки OpenGL **www.opengl.org.**

15.7. Рисование геометрических объектов

15.7.1. Процесс обновления изображения

Как правило, задачей программы, использующей OpenGL, является обработка трехмерной сцены и интерактивное отображение в буфере кадра. Сцена состоит из набора трехмерных объектов, источников света и виртуальной камеры, определяющей текущее положение наблюдателя. Обычно приложение OpenGL в бесконечном цикле вызывает функцию обновления изображения в окне. В этой функции и сосредоточены вызовы основных команд OpenGL. Если используется библиотека GLUT, то это будет функция с обратным вызовом, зарегистрированная с помощью вызова **glutDisplayFunc()**. GLUT вызывает эту функцию, когда операционная система информирует приложение о том, что содержимое окна необходим перерисовать (например, если окно было перекрыто другим). Создаваемое изображение может быть как статичным, так и анимированным, т. е. зависеть от каких-либо параметров, изменяющихся со временем. В этом случае лучше вызывать функцию обновления самостоятельно. Например, с помощью

команды **glutPostRedisplay()**. Типичная функция обновления изображения, как правило, состоит из трех шагов:

1) очистка буферов OpenGL;

2) установка положения наблюдателя;

3) преобразование и рисование геометрических объектов.

OpenGL содержит внутри себя несколько различных буферов. Среди них **фрейм-буфер** (куда производится построение изображения), **z-буфер** (буфер глубины), служащий для удаления невидимых поверхностей, **буфер трафарета** и **аккумулирующий буфер** (буфер накопления) (рис. 15.3)

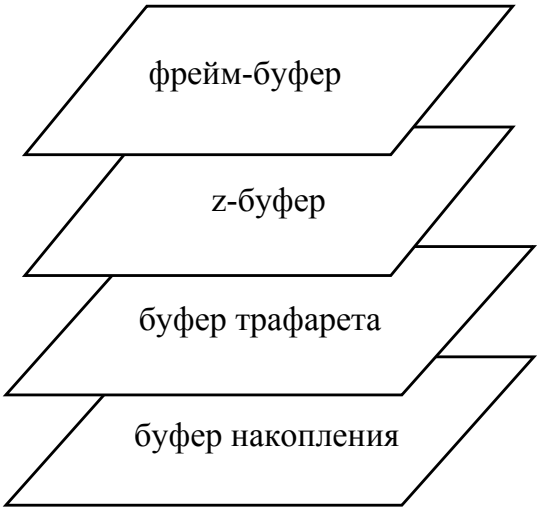


Рис. 15.3

Для очистки окна (экрана, внутренних буферов) служит команда

void glClear(GLbitfieldbuf),

которая очищает буферы, заданные переменной **buf**. Параметр **buf** определяет комбинацию констант, соответствующую буферам, которые нужно очистить. Его значения заданы в табл. 15.5.

Таблица 15.5

Значения параметра buf

Значение переменной buf	Комментарий
GL_COLOR_BUFFER_BIT	Очистить буфер изображения (фрейм-буфер)
GL_DEPTH_BUFFER_BIT	Очистить z-буфер
GL_ACCUM_BUFFER_BIT	Очистить аккумулирующий буфер
GL_STENSIL_BUFFER_BIT	Очистить буфер трафарета

Типичная программа вызывает команду для очистки буферов цвета и глубины:

glClear(GL_COLOR_BUFFER_BIT GL_DEPTH_BUFFER_BIT).

При этом цвет, которым очищается буфер изображения, задается командой

void glClearColor (GLclampf red, GLclampf green, GLclampf blue, GLclampf alpha).

Команда glClearColor устанавливает цвет, которым будет заполнен буфер кадра. Первые три параметра команды задают R, G и B компоненты цвета и должны принадлежать отрезку [0, 1]. Четвертый параметр задает так называемую альфа-компоненту. Как правило, он равен 1. По умолчанию цвет – черный (0, 0, 0, 1).

Установка положения наблюдателя и преобразования трехмерных объектов (поворот, сдвиг и т. д.) контролируются с помощью задания **матриц преобразования**. Преобразования объектов и настройка положения виртуальной камеры описаны ниже.

Рассмотрим вопрос о том, как передать в OpenGL описания объектов, находящихся в сцене. Каждый объект является набором примитивов OpenGL.

15.7.2. Вершины и примитивы

Вершина является атомарным графическим примитивом OpenGL и определяет точку, конец отрезка, угол многоугольника и т. д. Все остальные примитивы формируются с помощью задания вершин, входящих в данный примитив. Например, отрезок определяется двумя вершинами, являющимися концами отрезка. С каждой вершиной ассоциируются ее атрибуты. В число основных атрибутов входят **положение** вершины в пространстве, **цвет** вершины и **вектор** нормали.

Положение вершины в пространстве. Положение вершины определяются заданием ее координат в двух-, трех- или четырехмерном пространстве (однородные координаты). Это реализуется с помощью нескольких вариантов команды **glVertex***:

void glVertex[2 3 4][sifd] (typex, …),

void glVertex[2 3 4][s i f d]v (type *v).

Каждая команда задает четыре координаты вершины: **x**, **y**, **z**, **w**. Команда **glVertex2*** получает значения **x** и **y**. Координата **z** в таком случае устанавливается по умолчанию равной 0, координата **w** – равной 1. **Vertex3*** получает координаты **x**, **y**, **z** и заносит в координату **w** значение 1. **Vertex4*** позволяет задать все четыре координаты. Например:

```
glVertex2s(1, 2)
glvertex3f(2.3, 1.5, 0,2)
GLdoublevect[] = {1.0, 2.0, 3.0, 4.0}
glVertex4dv (vect)
```

Для ассоциации с вершинами цветов, нормалей и текстурных координат используются текущие значения соответствующих данных, что отвечает организации OpenGL как конечного автомата. Эти значения могут быть изменены в любой момент с помощью вызова соответствующих команд.

Цвет вершины. Для задания текущего цвета вершины используются команды:

void glColor[3 4][bsif d ub us ui] (type r, type g, type b, type a),

void glColor[3 4][bs i f d ub us ui]v (const type*v).

Первые три параметра задают R, G, B компоненты цвета, а последний параметр определяет коэффициент непрозрачности (так называемая альфа-компонента). Если в названии команды указан тип **«f»** (float), то значения всех параметров должны принадлежать отрезку [0, 1], при этом по умолчанию значение альфа-компоненты устанавливается равным 1, 0, что соответствует полной непрозрачности. Тип «**ub**» (unsigned byte) подразумевает, что значения должны лежать в отрезке [0, 255].

Вершинам можно назначать различные цвета, и если включен соответствующий режим, то будет проводиться линейная интерполяция цветов по поверхности примитива.

Для управления режимом интерполяции используется команда

void glShadeModel (GLenum mode),

вызов которой с параметром GL_SMOOTH включает интерполяцию (установка по умолчанию), а с GL_FLAT – отключает.

Нормаль. Вектор нормали применяется для расчетов освещения и затенения. Нормаль в OpenGL является атрибутом вершины. В общем случае, когда речь идет о грани трехмерного объекта, вершины одной грани имеют разные нормали, так как каждая вершина принадлежит нескольким граням, лежащим в разных плоскостях. Тогда нормаль к вершине можно определить как сумму нормалей к граням, содержащим рассматриваемую вершину: $\vec{N} = \sum_{i=1}^{K} \vec{N}_i$, или как среднюю нормаль: $\vec{N} = \frac{1}{K} \sum_{i=1}^{K} \vec{N}_i$, где N_i – вектор нормали к i-й грани; K – число граней, образующих вершину. Для стандартных объектов OpenGL рассчитывает нормали автомати-

чески. При отображении объектов, создаваемых пользователем, нормали должны быть рассчитаны самостоятельно.

После определения координат нормали она должна быть задана командами

void glNormal3[bsifd] (GLtype nx, GL type ny, GLtype nz),

void glNormal3[b s i f d]v (GLtype *v),

где **nx**, **ny**, **nz** – координаты новой текущей нормали; **v** – массив координат вектора нормали.

Нормали задаются к каждой выводимой вершине.

Для правильного расчета освещения необходимо, чтобы вектор нормали имел единичную длину. Командой **glEnable (**GL_NORMALIZE**)** можно включить специальный режим, при котором задаваемые нормали будут нормироваться автоматически. По умолчанию нормализация не выполняется.

Режим автоматической нормализации должен быть включен, если приложение использует модельные преобразования растяжения/сжатия, так как в этом случае длина нормалей изменяется при умножении на модельно-видовую матрицу.

Однако применение этого режима уменьшает скорость работы механизма визуализации OpenGL, так как нормализация векторов имеет заметную вычислительную сложность (взятие квадратного корня и т. п.). Поэтому лучше сразу задавать единичные нормали.

Отметим, что команды

void glEnable (GLenum mode),

void glDisable (GLenum mode),

где параметр **mode** определяет вид режима, производят включение и отключение того или иного режима работы конвейера OpenGL. Эти команды применяются достаточно часто, и их возможные параметры будут рассматриваться в каждом конкретном случае.

15.7.3. Операторные скобки glBegin / glEnd

Выше было рассмотрено задание атрибутов одной вершины. Однако чтобы задать атрибуты графического примитива, одних координат вершин недостаточно. Эти вершины надо объединить в одно целое, определив необходимые свойства. Для этого в OpenGL используются так называемые операторные скобки, являющиеся вызовами специальных команд OpenGL. Определение примитива или последовательности примитивов происходит между вызовами команд

void glBegin (GLenum mode),

void glEnd ()

Параметр **mode** определяет тип примитива, который задается внутри и может принимать значения, представленные на рис. 15.4 и в табл. 15.6.

Таблица 15.6

Значения параметра mode

Значение mode	Комментарий
GL_POINTS	Каждая вершина задает координаты некоторой точки
GL_LINES	Каждая отдельная пара вершин определяет отрезок; если задано нечетное число вершин, то последняя вершина игнорируется: v_0v_1, v_2v_3…
GL_LINE_STRIP	Каждая следующая вершина задает отрезок вместе с предыдущей – незамкнутая ломаная $v_0v_1v_2v_3…v_n$
GL_LINE_LOOP	Замкнутая ломаная $v_0v_1v_2v_3…v_nv_0$
GL_TRIANGLES	Каждые отдельные три вершины определяют треугольник. Если задано не кратное трем число вершин, то последние вершины игнорируются $v_0v_1v_2$, $v_3v_4v_5$...
GL_TRIANGLE_STRIP	Каждая следующая вершина задает треугольник вместе с двумя предыдущими – связанная полоса треугольников $v_0v_1v_2$, $v_2v_1v_3$, $v_2v_3v_4$…
GL_TRIANGLE_FAN	Треугольники задаются первой вершиной и каждой следующей парой вершин (пары не пересекаются) – веер треугольников $v_0v_1v_2$, $v_0v_2v_3$, $v_0v_3v_4$…
GL_QUADS	Каждая отдельная четверка вершин определяет четырехугольник; если задано не кратное четырем число вершин, то последние вершины игнорируются $v_0v_1v_2v_3$, $v_4v_5v_6v_7$, …
GL_QUAD_STRIP	Полоса четырехугольников $v_0v_1v_3v_2$, $v_2v_3v_5v_4$ … .
GL_POLYGON	Последовательно задаются вершины выпуклого многоугольника

Например, чтобы нарисовать треугольник с разными цветами в вершинах, достаточно написать:

```
GLfloat BlueCol[3] = {0,0,1};
glBegin(GL_TRIANGLES);
glColor3f(1.0, 0.0, 0.0);    //Красный
glVertex3f(0.0, 0.0, 0.0);
glColor3ub(0,255,0);       //Зеленый
glVertex3f(1.0, 0.0, 0.0);
glColor3fv(BlueCol);         //Синий
glVertex3f(1.0, 1.0, 0.0);
glEnd();
```

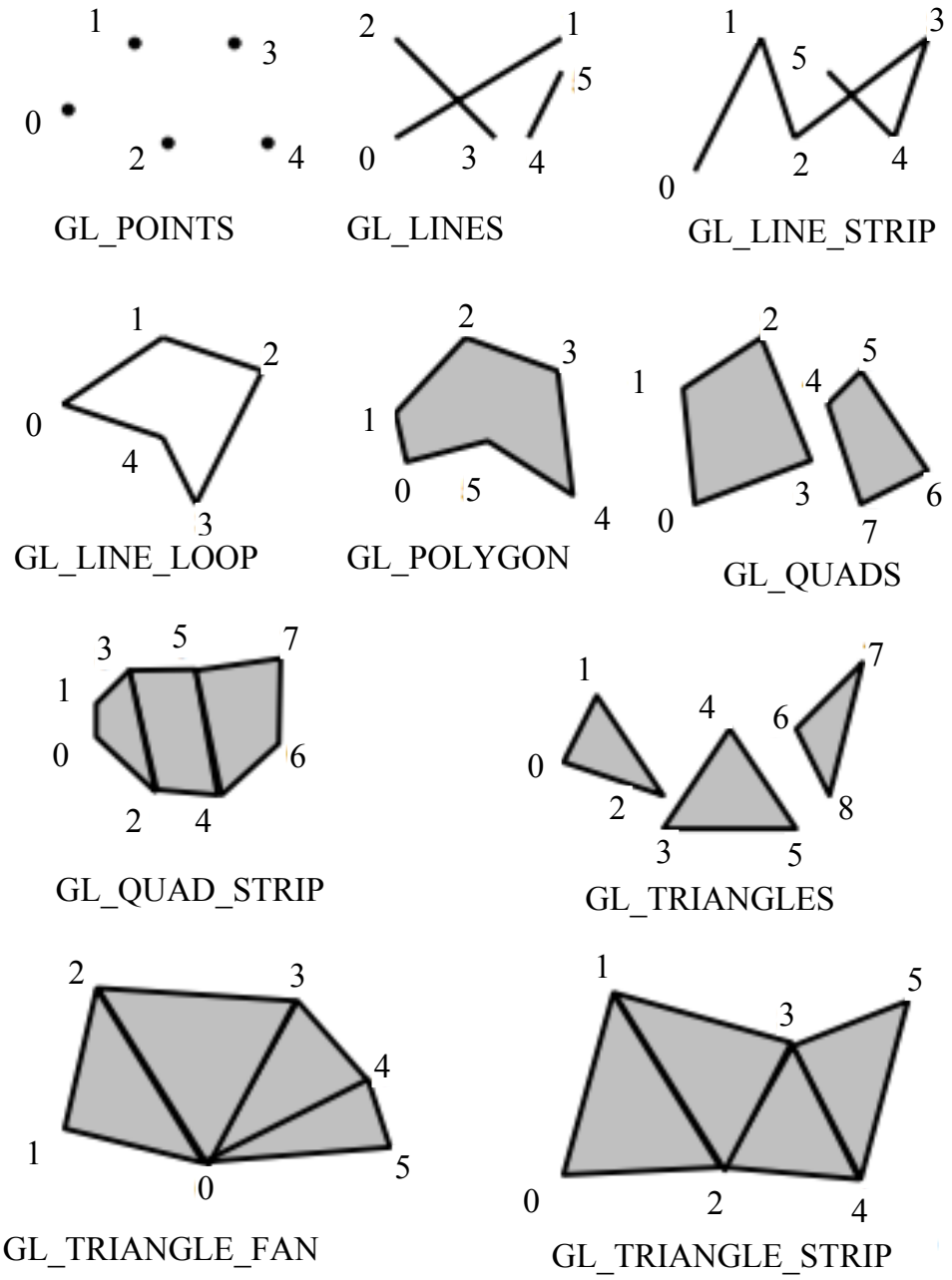


Рис. 15.4

Как правило, разные типы примитивов имеют различную скорость визуализации на разных платформах. Для увеличения производительности предпочтительнее использовать примитивы, требующие меньшее количество информации для передачи на сервер, такие как GL_TRIANGLE_STRIP, GL_QUAD_STRIP, GL_TRIANGLE_FAN. Кроме задания самих многоугольников, можно определить метод их отображения на экране. Для этого необходимо определить понятие **лицевых** и **обратных граней**. Под **гранью** (многоугольником) в OpenGL подразумевается замкнутый выпуклый многоугольник с несамопересекающейся границей.

По умолчанию лицевой считается та сторона, вершины которой обходятся **против** часовой стрелки. Направление обхода вершин лицевых граней можно изменить вызовом команды

void glFrontFace (GLenum mode)

со значением параметра **mode**, равным GL_CW(clockwise), а вернуть значение по умолчанию можно, указав GL_CCW (counter-clockwise). Чтобы изменить метод отображения многоугольника, используется команда

void glPolygonMode (GLenum face, Glenum mode).

Параметр **mode** определяет, **как** будут отображаться многоугольники, а параметр **face** устанавливает **тип** многоугольников, к которым будет применяться эта команда, и может принимать следующие значения:

GL_FRONT – для лицевых граней;

GL_BACK – для обратных граней;

GL_FRONT_AND_BACK – для всех граней.

Параметр **mode** может быть равен:

GL_POINT – отображение только вершин многоугольников;

GL_LINE – многоугольники будут представляться набором отрезков;

GL_FILL – многоугольники будут закрашиваться текущим цветом с учетом освещения, и этот режим установлен по умолчанию.

Также можно указывать, какой тип граней отображать на экране. Для этого сначала надо установить соответствующий режим вызовом команды **glEnable (GL_CULL_FACE)**, а затем выбрать тип отображаемых граней с помощью команды

void glCullFace (GLenum mode).

Вызов с параметром GL_FRONT приводит к удалению из изображения всех лицевых граней, а с параметром GL_BACK – обратных (установка по умолчанию).

Кроме рассмотренных стандартных примитивов в библиотеках GLU и GLUT описаны более сложные фигуры, такие как сфера, цилиндр, диск (в GLU) и сфера, куб, конус, тор, тетраэдр, додекаэдр, икосаэдр, октаэдр и чайник (в GLUT). Автоматическое наложение текстуры предусмотрено только для фигур из библиотеки GLU.

При изображении объектов в OpenGL можно изменять размер точек и толщину линий. Для этой цели служат следующие команды;

void glPointSize(GLfloat size) – устанавливает диаметр (size) в пикселах точек, изображаемых с помощью примитива GL_POINTS.

По умолчанию значение параметра size равно 1,0. В любой реализации OpenGL гарантирована поддержка размера точки, равного 1,0. В типичных реализациях Microsoft Windows поддерживается размер точки от 0,5 до 10 с шагом 0,125;

void glLineWidth(GLfloat size) – устанавливает ширину (size) в пикселах линий, изображаемых с помощью примитивов GL_LINES, GL_LINE_STRIP или GL_LINE_LOOP. По умолчанию значение параметра size равно 1,0. В любой реализации OpenGL гарантирована поддержка размера точки, равного 1,0. В типичных реализациях Microsoft Windows поддерживается размер точки от 0,5 до 10 с шагом 0,125.

15.7.4. Дисплейные списки

В традиционных языках программирования существуют функции и процедуры – т. е. можно выделить определенный набор команд, запомнить его в некотором одном определенном месте и вызывать каждый раз, когда возникает потребность в соответствующей последовательности команд. Подобная возможность существует и в OpenGL – набор команд OpcnGL можно запомнить в так называемый дисплейный список (***display list***), при этом все команды и данные переводятся в некоторое внутреннее представление, наиболее удобное для данной реализации OpenGL, и затем вызвать при помощи всего одной команды.

Каждому дисплейному списку соответствует некоторое целое число, идентифицирующее этот список.

Для создания нового дисплейного списка необходимо поместить все команды, которые должны в него войти, между следующими операторными скобками:

void glNewList (GLuint list, GLenum mode),

void glEndList (),

где параметр **list** задает номер списка, а параметр **mode** определяет режим обработки команд, входящих в список, и может принимать следующие значения:

GL_COMPILE – команды записываются в список без выполнения,

GL_COMPILE_AND_EXECUTE – команды сначала выполняются, а затем записываются в список.

Следует иметь в виду, что если в качестве параметра **list** списка используется уже существующий номер, то новый список не создается, а существующий список заменяется на новый.

Узнать, занят ли данный номер **list** каким-либо дисплейным списком, можно при помощи функции

GLboolean gllsList (GLuint list).

Зарезервировать диапазон (range) свободных подряд идущих номеров для идентификации дисплейных списков можно при помощи функции

GLuint glGenLists (GLsizei range).

Эта функция возвращает первый свободный номер из блока зарезервированных номеров данной командой.

После того как список создан, его можно вызвать командой **void glCallList (GLuint list)**, указав в параметре **list** идентификатор нужного списка. Для вызова сразу нескольких списков можно воспользоваться командой **void glCallLists (GLsizei n, GLenumtype, constGLvoid *lists)**, вызывающей **n** списков с идентификаторами из массива **lists**, тип элементов которого указывается в параметре **type**. Это могут быть типы GL_BYTE, GL_UNSIGNED_ BYTE, GL_SHORT,GL_INT, GL_UNSIGNED_INT и некоторые другие. Для удаления списков используется команда

void glDeleteLists (Glint list, GLsizei range),

которая удаляет списки с идентификаторами **ID** из диапазона **list ≤ ≤ ID ≤ list + range – 1**.

Пример:

```
glNewList(1, GL_COMPILE);
  glBegin(GL_TRIANGLES);
    glVertex3f(1.0f, 1.0f, 1.0f);
    glVertex3f(10.0f, 1.0f, 1.0f);
    glVertex3f(10.0f, 10.0f, 1.0f);
  glEnd();
glEndList()
…
glCallList(1);
```

Замечание. Дисплейные списки нельзя изменять. Список можно уничтожить или создать заново. В дисплейном списке команды запоминаются вместе со своими аргументами на момент передачи, так что в следующем примере последний оператор присваивания дисплейный список не изменяет.

```
GLfloat color [] ={0.0, 0.0, 0.0};
glNewList (1, GL.COMPILE);
glColor3fv (color);
glEndList ();
color [2] = 1.0; //Не изменяет дисплейный список.
```

Не все команды OpenGL могут быть записаны в дисплейный список. К ним относятся такие команды, как **glDeleteLists()**, **glFinish()**, **glGenLists()** и некоторые другие.

Дисплейные списки в оптимальном, скомпилированном виде хранятся в памяти сервера, что позволяет рисовать примитивы в такой форме максимально быстро. В то же время большие объемы данных занимают много памяти, что влечет, в свою очередь, падение производительности. Такие большие объемы (больше нескольких десятков тысяч примитивов) лучше рисовать с помощью массивов вершин.

15.7.5. Массивы вершин

Если вершин много, то, чтобы не вызывать для каждой команду glVertex*(), удобно объединять вершины в массивы, используя команду

void glVertexPointer (Glint size, GLenum type,
GLsizei stride, void* ptr),

которая определяет способ хранения и координаты вершин. При этом **size** определяет число координат вершины (может быть равен 2, 3, 4), **type** определяет тип данных (может быть равен GL_SHORT, GL_INT, GL_FLOAT, GL_DOUBLE). Иногда удобно хранить в одном массиве другие атрибуты вершины, тогда параметр **stride** задает смещение от координат одной вершины до координат следующей; если **stride = 0**, это значит, что координаты расположены последовательно. В параметре **ptr** указывается адрес, где находятся данные.

Аналогично можно определить массив нормалей, цветов и некоторых других атрибутов вершины, используя команды:

void glNormalPointer (GLenum type, GLsizei stride,
void *pointer),
void glColorPointer (GLint size, GLenum type, GLsizei stride,
void *pointer).

Для того чтобы эти массивы можно было использовать в дальнейшем, надо вызвать команду

void glEnableClientState (GLenum array),

где параметр **array** может принимать значения GL_VERTEX_ ARRAY, GL_NORMAL_ARRAY и GL_COLOR_ ARRAY соответственно. После окончания работы с массивом желательно вызвать команду

void glDisableClientState (GLenum array)

с соответствующим значением параметра **array**.

Для отображения содержимого массивов используется команда

void glArrayElement (GLint index),

которая передает OpenGL атрибуты вершины, используя элементы массива с номером **index**. Это аналогично последовательному применению команд вида **glColor*(…)**, **glNormal*(…)**, **glVertex*(…)** с соответствующими параметрами. Однако вместо нее обычно вызывается команда

void glDrawArrays (GLenum mode, GLint first, GLsizei count),

рисующая **count** примитивов, определяемых параметром **mode**, используя элементы из массивов с индексами от **first** до **first + + count – 1**. Это эквивалентно вызову последовательности команд **glArrayElement()** с соответствующими индексами. Параметр **mode** определяет тип примитива и может принимать значения: GL_POINTS, GL_LINE, GL_LINE_LOOP, GL_LINE_STRIP, GL_TRIANGLES, GL_TRIANGLE_STRIP, GL_TRIANGLE_FAN, GL_QUADS, GL_QUAD_STRIP и GL_POLYGON.

В случае, если одна вершина входит в несколько примитивов, то вместо дублирования ее координат в массиве удобно использовать ее индекс. Для этого надо вызвать команду

void glDrawElements (GLenum mode, GLsizei count,
GLenum type, void *indices),

где **indices** – это массив номеров вершин, которые надо использовать для построения примитивов, **type** определяет тип элементов этого массива: GL_UNSIGNED_BYTE, GL_UNSIGNED_SHORT, GL_UNSIGNED_INT, а **count** задает их количество.

Важно отметить, что использование массивов вершин позволяет оптимизировать передачу данных на сервер OpenGL, и, как следствие, повысить скорость рисования трехмерной сцены. Такой метод определения примитивов является одним из самых быстрых и хорошо подходит для визуализации больших объемов данных.

Помимо рассмотренных примитивов, определяемых внутри командных скобок, в OpenGL предусмотрены команды

void glRect[d f i s](GLtype x1, GLtype y1, GLtype x2, GLtype y2)

и

void glRect[d f i s] v (GLtype* v1, GLtype* v2),

которые позволяют достаточно эффективно задавать прямоугольник в плоскости **z = 0**, определяя два его противоположных угла.

Параметры **xl** и **yl** задают координаты одной из вершин прямоугольника, а **x2** и **y2** – координаты его противоположной вер-

шины. Параметры **vl** и **v2** определяют указатели на соответствующие двухточечные массивы вершин.

Если прямоугольник изображается с помощью функции **glRect**, то стороны многоугольника прокладываются между вершинами в следующем порядке: **(x1, yl)**, **(x2, y1)**, **(x2, y2)**, **(xl, y2)**, а затем обратно к первой вершине.

Их действие аналогично, например, следующей последовательности команд:

```
glBegin(GL_POLYGON) ;
glVertex2*(xl, yl); // * заменяется соответствующим
glVertex2*(x2, yl); // символом, зависящим от типа
glVertex2*(x2, y2); // используемых данных
glVertex2*(xl, y2);
glEnd();
```

15.7.6. Вывод текста

Существуют два типа шрифтов – **штриховой (эскизный)** и **растровый**. Штриховой шрифт формируется по тому же принципу, что и прочие графические примитивы. Начертание символа определяется вершинами соответствующих прямолинейных и криволинейных отрезков. Если символ формируется из замкнутых контуров, то его внутреннюю область можно залить. Преимущество использования штриховых шрифтов заключается в том, что с ними можно обращаться в графической системе точно так же, как с любыми другими графическими объектами. Несомненным достоинством штриховых шрифтов является и сохранение начертания во всех деталях при выполнении геометрических преобразований масштабирования или поворота. Поэтому определение символов набора выполняется только для одного, базового, размера, а все прочие получаются после выполнения стандартных геометрических преобразований – масштабирования и поворота. Однако визуализируются штриховые шрифты медленнее, чем растровые.

Растровый шрифт определяется значительно проще, а отображается быстрее (рис. 15.5). Символы шрифта определяются на прямоугольной области и представляют собой блоки битов. Каждый блок задает определенный символ в виде образа из нулей и единиц, которые соответствуют засвеченным и незасвеченным точкам растра. При отображении определенный таким способом символ помещается в буфер кадра с помощью операции побитового переноса, которая выполняется очень быстро.

Увеличить размер символов растрового шрифта можно только дублированием пикселов, что при большом увеличении приводит к формированию символов довольно «грубой» формы. Выполнять какие-либо геометрические преобразования растрового шрифта бессмысленно. Кроме того, поскольку растровые шрифты, как правило, хранятся в постоянной памяти, они не переносятся с компьютера на компьютер.

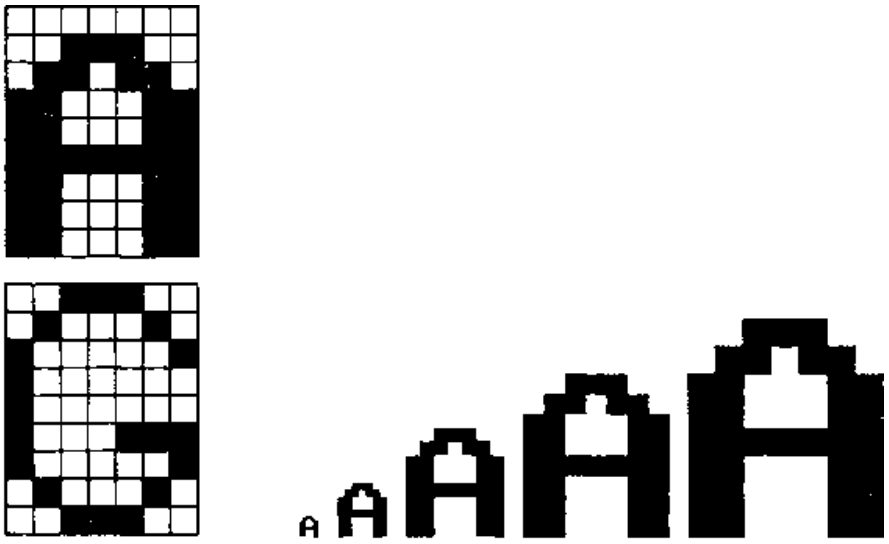


Рис. 15.5

Символы как штриховых, так и растровых шрифтов формируются из других примитивов, а потому в составе основной библиотеки OpenGL нет специального примитива для формирования текста. Но в составе дополнительной библиотеки GLUT имеется несколько наборов символов (как штриховых, так и растровых), которые определены программно, а следовательно, являются переносимыми.

Растровый символ GLUT можно получить с помощью функции **glutBitmapCharacter (font, character)**.

Параметр **font** определяет начертание шрифта и может принимать значение одной из символьных констант GLUT:

GLUT_BITMAP_8_BY_13;
GLUT_BITMAP_9_BY_15;
GLUT_BITMAP_TIMES_ROMAN_10;
GLUT_BITMAP_HELVETICA_12;
GLUT_BITMAP_HELVETICA_18.

Параметру **character** присваивается либо код ASCII, либо отдельный символ, который мы хотим изобразить. Таким образом,

чтобы изобразить прописную букву «А», можно воспользоваться либо значением 65 кода ASCII, либо обозначением 'A'.

Каждый символ, созданный с помощью функции **glutBitmap Character**, изображается так, что начало координат (нижний левый угол) битового массива находится в текущем растровом положении. После того как битовый массив символа загружается в буфер регенерации, к координате *x* текущего растрового положения добавляется смещение, равное ширине символа.

Изменение значения текущей позиции растра выполняется с помощью разных модификаций функции **glRasterPos*** (например, **glRasterPos2i(GLint x, GLint y)**, **glRasterPos3i(GLint x, GLinty y, GLint z)**.

При установке растрового положения текущий цвет растровых изображений устанавливается равным текущему цвету. В качестве примера изобразим известную текстовую строку «Hello World!!!».

```
MyText[]=”Hello World!!!”;
glColor3ub(255, 0, 0);  //Цвет текста
glRasterPos2i(-10, 3);   //Координаты начала вывода
len = strnlen(MyText2, 100);
for (int i = 0; i<len; i++)
glutBitmapCharacter(GLUT_BITMAP_9_BY_15, MyText[i]);
```

Эскизный символ изображается путем вызова функции **glutStrokeCharacter(font, character)**.

Для этой функции параметру **font** можно присвоить значение GLUT_STROKE_ ROMAN, чтобы изобразить пропорциональный шрифт, или GLUT_STROKE_MONO_ ROMAN, чтобы представить моноширинный шрифт. Размер и положение этих символов можно контролировать с помощью операций аффинных преобразований, которые вызываются перед выполнением процедуры **glutStrokeCharacter**.

15.7.7. Работа с z-буфером

Как уже было сказано, OpenGL поддерживает использование аппаратного буфера для удаления невидимых поверхностей. Каждое поступающее значение глубины фрагмента сравнивается с имеющимся в буфере значением глубины и выводится на экран (или нет) в зависимости от результатов выполнения этого теста. Более того, результат этого теста влияет на содержимое буфера трафарета, что в совокупности представляет поразительно мощное

средство создания изображений практически любой степени сложности.

Исходно тест глубины заблокирован и может быть разрешен при помощи команды **glEnable (GL_DEPTH_TEST)**.

Выключить эту проверку можно командой

glDisable (GL_DEPTH_TEST).

Функция сравнения, используемая в тесте, задается командой

void glDepthFunc(GLenum func).

Вызов этой команды позволяет определить ту функцию, которая будет использоваться для сравнения каждого поступающего **z**-значения с тем, которое хранится в буфере глубины. Функцию, используемую для сравнения глубин, определяет параметр **func**, который может принимать одно из следующих значений, указанных в табл. 15.7.

Таблица 15.7

Значения параметра func

Константа	Тест завершается положительно...
GL_NEVER	Никогда
GL_LESS	Если поступающее значение меньше, чем хранящееся в буфере
GL_EQUAL	Если поступающее значение равно хранящемуся в буфере
GL_LEQUAL	Если поступающее значение меньше, чем хранящееся в буфере, или равно ему
GL_GREATER	Если поступающее значение больше, чем хранящееся в буфере
GL_NOTEQUAL	Если поступающее значение не равно хранящемуся в буфере
GL_GEQUAL	Если поступающее значение больше или равно хранящемуся в буфере
GL_ALWAYS	Всегда

По умолчанию используется константа GL_LESS. Если буфера глубины нет, то считается, что тест всегда завершается положительно.

15.7.8. Преобразования координат объектов

Системы координат. В OpenGL используются как основные три системы координат: **левосторонняя**, **правосторонняя** и **оконная** (рис. 15.6). Первые две системы являются трехмерными и отличаются друг от друга направлением оси *Z*: в правосторонней

она направлена на наблюдателя, в левосторонней – в глубину экрана. Ось *X* направлена вправо относительно наблюдателя, ось *Y* – вверх.

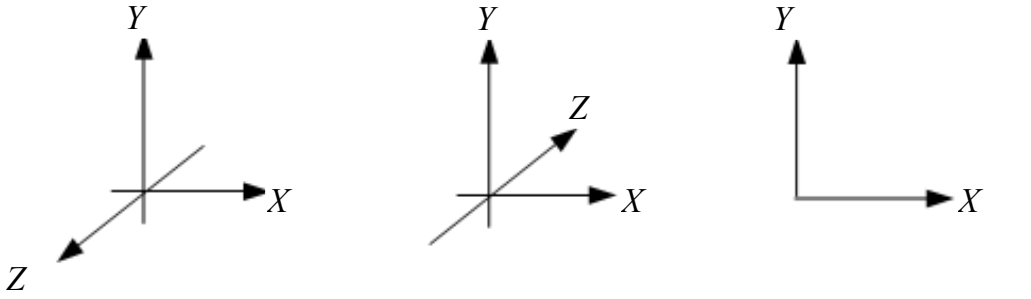


Рис. 15.6

Рассматривая какой-либо трехмерный объект, мы всегда определяем его положение и размеры относительно некоторой привычной и удобной в настоящий момент системы координат, связанной с реальным миром. Такая исходная система координат в компьютерной графике является **правосторонней** и называется **мировой системой координат (МСК).** В ней над объектами выполняются заданные действия, например аффинные преобразования. Для того чтобы можно было изобразить объект на экране, его необходимо предварительно перевести (или преобразовать) в другую систему координат, которая связана с точкой наблюдения и носит название **видовой системы координат** или **системы координат наблюдателя (СКН).** Эта система координат является левосторонней. В ней рассчитываются цвета объектов и выполняются заданные отсечения. Далее, координаты СКН переводятся в нормированные (нормализованные) координаты (НК), которые лежат в диапазоне [–1, 1]. И, наконец, любое трехмерное изображение мы всегда рисуем на двумерном экране, который имеет свою **экранную (оконную) систему координат** (ОСК). Взаимное расположение мировой (*XYZ*) и видовой ($X_eY_eZ_e$) систем координат, а также экранная система координат (*xy*) показаны на рис. 15.7 и 15.8 соответственно.

Таким образом, основная задача, которую необходимо решить, заключается в том, что объекты, описанные в мировых координатах, необходимо изобразить на плоской области вывода экрана, т. е. необходимо проделать некоторую последовательность преобразований и осуществить проекцию, переводящую трехмерные объекты на двумерную проекционную плоскость.

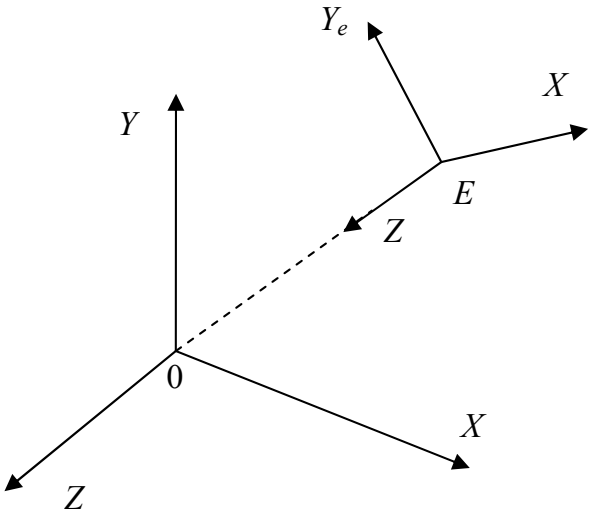


Рис. 15.7

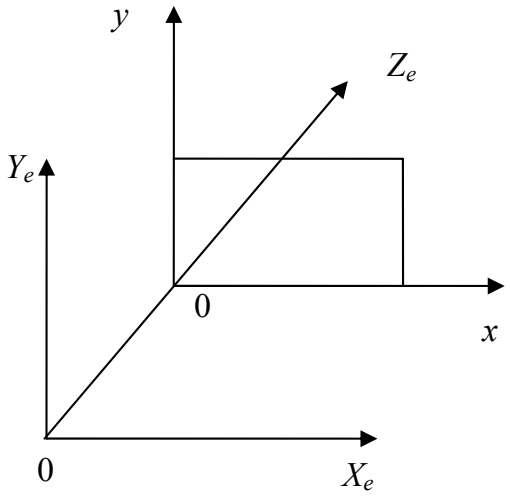


Рис. 15.8

Эту последовательность действий удобно выполнять в несколько этапов, как это и реализовано в OpenGL (рис. 15.9).

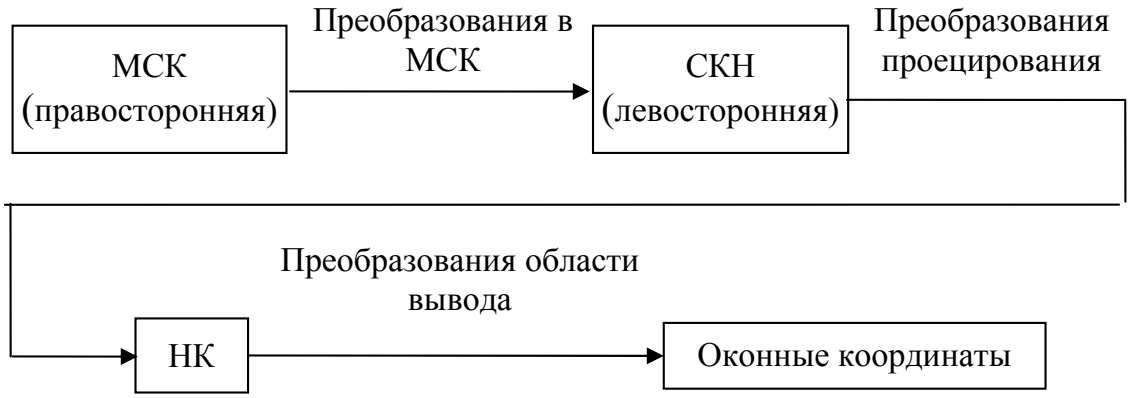


Рис. 15.9

Координаты отображаемых объектов (примитивов) задаются в МСК. В ней же под воздействием **видовой матрицы** выполняются заданные действия, например аффинные преобразования. В результате воздействия видовой матрицы координаты вершин объектов пересчитываются в СКН. В ней рассчитываются цвета объектов и выполняются заданные отсечения. Далее под воздействием **матрицы проецирования** координаты СКН пересчитываются в нормированные координаты. Диапазон их изменения – [–1.0, 1.0]. Непосредственно перед выводом в окно нормированные координаты преобразуются в оконные.

Матрицы. В OpenGL предусмотрены **три** типа матриц – **видовая**, **проекций** и **текстуры**, все они имеют размер 4×4.

Для того чтобы с какой-либо матрицей можно было работать, ее необходимо сделать текущей, для чего предусмотрена команда

void glMatrixMode(GLenum mode).

Параметр **mode** определяет, с каким набором матриц будут выполняться последующие операции, и может принимать одно из трех значений:

GL_MODELVIEW – последовательность операций над матрицами применяется к видовой матрице;

GL_PROJECTION – последовательность операций над матрицами применяется к матрице проекции;

GL_TEXTURE – последовательность операций над матрицами применяется к матрице текстуры.

После того как установлена текущая матрица, необходимо определить все элементы. Для этой цели можно использовать команду

void glLoadMatrix[fd](GLtype* m).

Параметр **m** определяет указатель на матрицу 4×4, хранящуюся по порядку расположения столбцов как 16 последовательных вещественных значений одинарной (f) или двойной (d) точности:

$$m = \begin{pmatrix} a_1 & a_5 & a_9 & a_{13} \\ a_2 & a_6 & a_{10} & a_{14} \\ a_3 & a_7 & a_{11} & a_{15} \\ a_4 & a_8 & a_{12} & a_{16} \end{pmatrix}$$

Эта команда заменяет текущую матрицу на ту, которая определена в **m**. Текущей является одна из матриц – видовая, проекций или текстуры, в зависимости от текущего режима.

Команда **void glLoadIdentity()** заменяет текущую матрицу на единичную (матрицу идентичности):

$$m = I = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Семантически эта команда эквивалентна **glLoadMatrix** с параметром, определяющим единичную матрицу, но выполняется более эффективно.

Часто бывает необходимо сохранить содержимое текущей матрицы для дальнейшего использования, для чего применяются команды

void glPushMatrix (void),
void glPopMatrix (void).

Они записывают и восстанавливают текущую матрицу из стека, причем для каждого типа матриц стек свой. Для модельно-видовых матриц его глубина равна как минимум 32, для остальных – как минимум 2.

При выполнении преобразований объектов, таких как перенос, вращение, масштабирование, могут понадобиться соответствующие операции над матрицами. OpenGL предоставляет набор команд для решения этой задачи. Фундаментальной является команда перемножения матриц

void glMultMatrix[fd](GLtype* m),

где параметр **m** определяет указатель на матрицу размером 4×4, хранящуюся по порядку расположения столбцов. Эта команда умножает матрицу, заданную параметром **m слева**, на текущую. Если обозначить текущую матрицу буквой M, передаваемую матрицу буквой T, то в результате выполнения команды **glMultMatrix** текущей становится матрица M·T (T←M·T). Однако обычно для изменения матрицы того или иного типа удобно использовать специальные команды, которые по значениям своих параметров создают нужную матрицу и умножают ее на текущую. К использованию этой команды следует прибегать только в самом крайнем случае, так как для перемножения требуются значительные вычислительные затраты.

В целом, для отображения трехмерных объектов сцены в окне приложения используется последовательность, показанная на рис. 15.10.

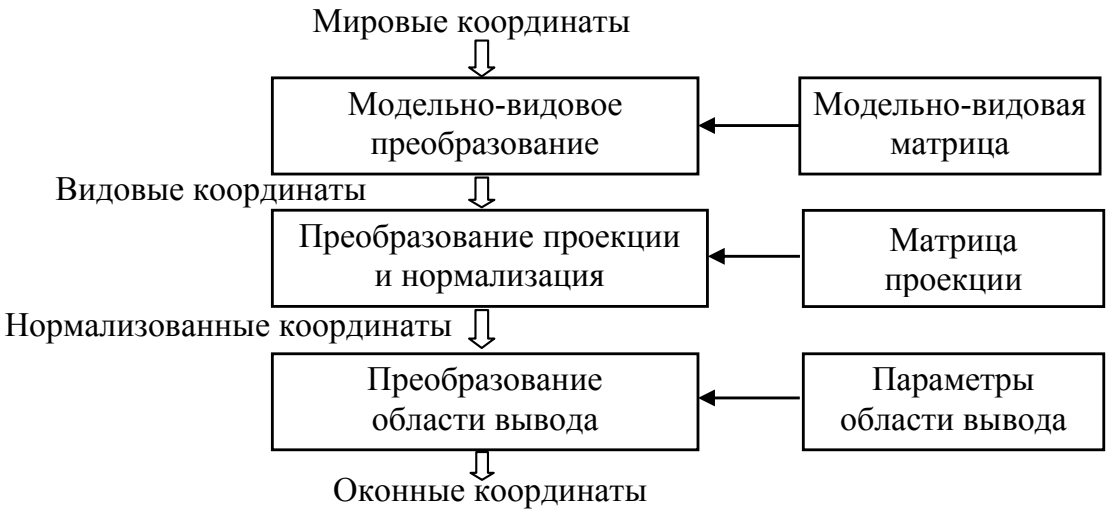


Рис. 15.10

Таким образом, все преобразования объектов и камеры в OpenGL производятся с помощью умножения векторов координат на матрицы. Причем умножение происходит на текущую матрицу в момент определения координаты командой **glVertex*** и некоторыми другими.

Модельно-видовые преобразования. К модельно-видовым преобразованиям будем относить перенос, поворот и изменение масштаба вдоль координатных осей. Для проведения этих операций достаточно умножить на соответствующую матрицу каждую вершину объекта и получить измененные координаты этой вершины:

$$\begin{pmatrix} x_e & y_e & z_e & 1 \end{pmatrix}^T = M \cdot \begin{pmatrix} z & y & z & 1 \end{pmatrix}^T,$$

где M – матрица модельно-видового преобразования. Перспективное преобразование и проектирование производится аналогично. Сама матрица может быть создана с помощью следующих команд:

void glTranslate[fd] (GLtype x, GLtype y, GLtype z),

void glRotate[f d] (GLtype angle, GLtype x, GLtype y, GLtype z),

void glScale[f d] (GLtype x, GLtype y, GLtype z),

где **glTranlsate*()** производит перенос объекта, прибавляя к координатам его вершин значения своих параметров;

glRotate*() производит поворот объекта против часовой стрелки на угол **angle** (измеряется в градусах) вокруг вектора (x, y, z);

glScale*() производит масштабирование объекта (сжатие или растяжение) вдоль вектора (x, y, z), умножая соответствующие координаты его вершин на значения своих параметров.

Все эти преобразования изменяют текущую матрицу, а поэтому применяются к примитивам, которые определяются позже. В случае, если надо, например, повернуть один объект сцены, а другой оставить неподвижным, удобно сначала сохранить текущую видовую матрицу в стеке командой **glPushMatrix()**, затем вызвать **glRotate()** с нужными параметрами, описать примитивы, из которых состоит этот объект, а затем восстановить текущую матрицу командой **glPopMatrix().**

Кроме изменения положения самого объекта часто бывает необходимо изменить положение наблюдателя, что также приводит к изменению модельно-видовой матрицы. Это можно сделать с помощью команды

void gluLookAt (GLdouble ex, GLdouble ey, GLdouble ez,

GLdouble center_x, GLdouble center_y, GLdouble center_z,

GLdouble up_x, GLdouble up_y, GLdouble up_z),

где точка **(ex, ey, ez)** определяет точку наблюдения в МСК, точка **(center_x, center_y, center_z)** задает центр сцены, который будет проектироваться в центр области вывода, а вектор **(up_x, up_y, up_z)** задает направление вектора, вдоль которого в видовом порте (см. ниже) ориентируется ось *Y*, определяя поворот камеры. Причем вектор **(up_x, up_y, up_z)** не должен быть параллелен линии, проведенной от точки наблюдения **(ex, ey, ez)** до наблюдаемой точки **(center_x, center_y, center_z)**.

Если, например, камеру не надо поворачивать, то задается значение **(0, 1, 0)**, а со значением **(0, –1, 0)** сцена будет перевернута.

Строго говоря, эта команда совершает перенос и поворот объектов сцены, но в таком виде задавать параметры бывает удобнее. Следует отметить, что вызывать команду **gluLookAt()** имеет смысл перед определением преобразований объектов, когда модельно-видовая матрица равна единичной.

Замечание. В общем случае матричные преобразования в OpenGL нужно записывать в обратном порядке. Например, если необходимо сначала повернуть объект, а затем передвинуть его, то сначала надо вызвать команду **glTranslate()**, а только потом – **glRotate()**. После этого можно определить сам объект.

15.7.9. Проекции

Несоответствие между пространственными объектами и плоскими изображениями устраняется путем введения **проекций**, которые отображают объекты на двухмерной проекционной картинной плоскости. В зависимости от расстояния между наблюдателем и картинной плоскостью различают два основных класса проекций: **параллельные** и **центральные (перспективные)**. Если это расстояние бесконечно, то проекция будет параллельной, а если конечно, то центральной. При описании центральной проекции необходимо явно задавать **центр проекции**, а для определения параллельной проекции необходимо определить направление проецирования.

OpenGL поддерживает два вида проекций – **ортографическую (параллельную)** и **перспективную**.

В OpenGL существуют стандартные команды для задания ортографической (параллельной) и перспективной проекций, выполнение которых устанавливает в системе координат наблюдателя область видимости графических данных. Первый тип проекции может быть задан командами

void glOrtho (GLdouble x_left, GLdouble x_right,
GLdouble y_bottom, GLdouble y_top, GLdouble z_near,
GLdouble z_far)

и

void gluOrtho2D (GLdouble x_left, GLdouble x_right,
GLdouble y_bottom, GLdouble y_top)

Первая команда создает матрицу проекции в усеченный объем видимости (**параллелепипед видимости**) в левосторонней системе координат. Параметры команды задают точки (**x_left, y_bottom, z_near**) и (**x_right, y_top, z_far**), которые отвечают левому нижнему и правому верхнему углам окна вывода. Параметры **z_near** и **z_far** задают расстояние до ближней и дальней плоскостей отсечения по удалению от точки **(0, 0, 0)** и могут быть отрицательными.

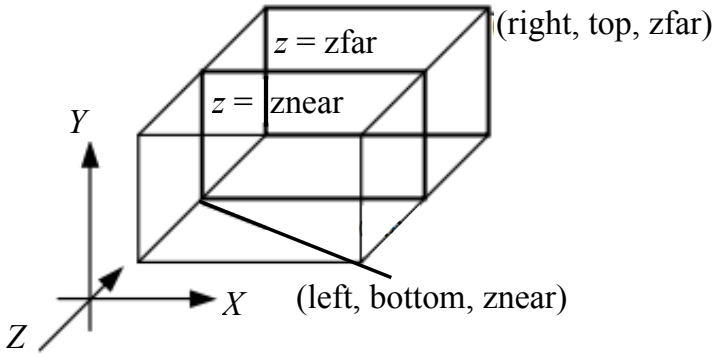


Рис. 15.11

Во второй команде, в отличие от первой, значения **znear** и **zfar** устанавливаются равными **–1** и **1** соответственно. Это удобно, если OpenGL используется для рисования двумерных объектов. В этом случае положение вершин можно задавать, используя команды **glVertex2*()**.

После преобразования ортографического проецирования координаты любых точек внутри преобразованной трехмерной области видимости находятся в диапазоне от **–1** до **1**. Таким образом, преобразованная область видимости представляет собой куб со стороной, равной двум. Напомним, что координаты, полученные после преобразования, называются **нормированными (нормализованными)**.

При использовании перспективной проекции в качестве области видимости необходимо определить не параллелепипед, а **усеченную пирамиду** (рис. 15.12). Для этого в OpenGL реализованы две команды.

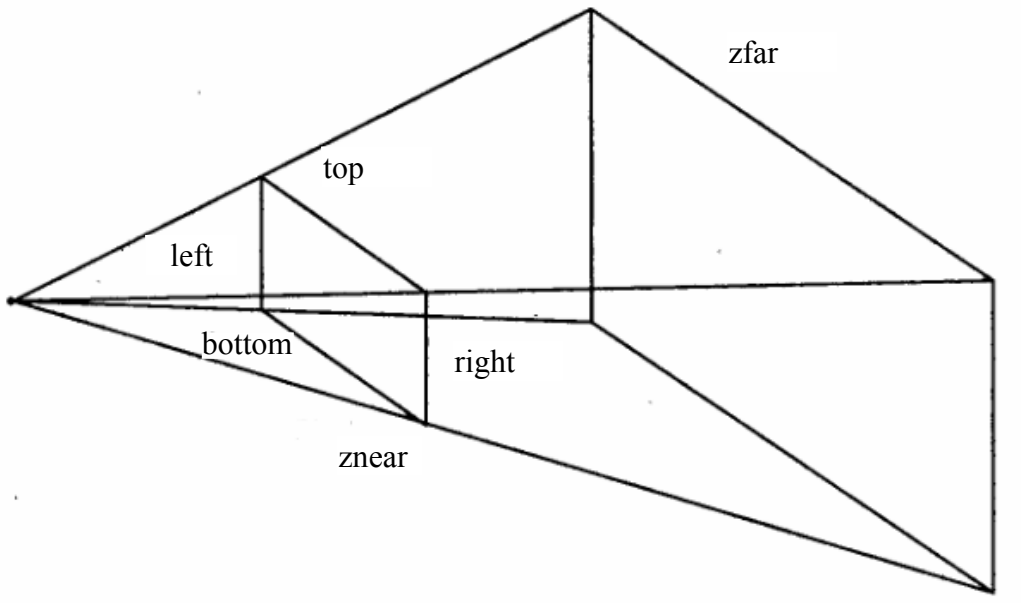


Рис. 15.12

Первая команда

void glFrustrum (GLdouble left, GLdouble right,
GLdouble top, GLdouble bottom,
GLdouble znear, GLdouble zfar)

создает матрицу перспективы. Параметры **left** и **right** определяют координаты левой и правой вертикальных, а **bottom** и **top** – нижней и верхней горизонтальных плоскостей отсечения. Параметры **znear** и **zfar** – расстояния до ближней и дальней плоскостей отсечения по глубине, причем оба значения должны быть положительными. Считается, что наблюдатель располагается в точке с координатами (0, 0, 0).

Вторая команда

void gluPerspective (GLdouble angley, GLdouble aspect,
GLdouble znear, GLdouble zfar)

задает усеченную пирамиду видимости в левосторонней (видовой) системе координат (рис. 15.13).

Параметр **angley** определяет угол видимости в градусах по оси *Y*, **tg(angley/2) = top/near** и должен находиться в диапазоне от 0 до 180. Угол видимости вдоль оси *X* задается параметром **aspect**, который обычно определяется как отношение ширины **(width)** и высоты **(height)** сторон области вывода (как правило, размеров окна). Величина отношения **aspect** должна соответствовать заданному видовому порту (см. ниже). Например, если **aspect=2,** то отношение **width/ height** (*x*/*y*) также следует задать равным двум.

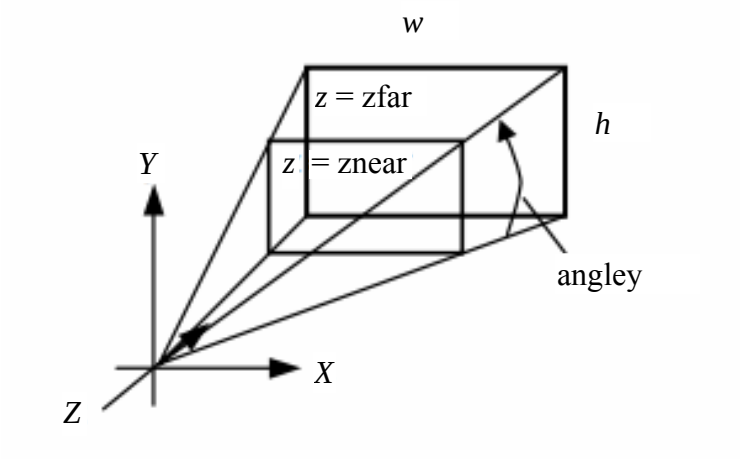


Рис. 15.13

Параметры **zfar** и **znear** задают расстояние от наблюдателя до плоскостей отсечения по глубине и должны быть положительными. Действие этой команды эквивалентно вызову **glFrustum** при **left = = –right, bottom = –top, tg(angley/2) = top/near, aspect = right/top**.

Чем больше отношение **zfar/znear**, тем хуже в буфере глубины будут различаться расположенные рядом поверхности, так как по умолчанию в него будет записываться «сжатая» глубина в диапазоне от 0 до 1 (см. далее).

Также как и для случая ортографической проекции, усеченный конус видимости отображается на куб со стороной, равной двум.

Для задания матрицы проекций необходимо включить режим работы с нужной матрицей командой

glMatrixMode(GL_PROJECTION)

и сбросить текущую, вызвав **glLoadIdentity()**.

Пример:

```
//ортографическая проекция
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
glOrtho(0, w, 0, h, -1.0, 1.0);
```

15.7.10. Область вывода

Область вывода представляет собой прямоугольник в оконной системе координат, размеры которого задаются командой:

void glViewPort (GLint x, GLint y, GLint width, GLint height).

Значения всех параметров задаются в пикселах. Параметры **x** и **y** определяют координаты левого нижнего угла прямоугольника вывода в оконной системе координат (видовой порт) и по умолча-

нию принимаются равными нулю. Размеры оконной системы координат определяются текущими размерами окна приложения, точка (0,0) находится в левом нижнем углу окна.

Команда **glViewPort** задает аффинные преобразования области вывода, переводящие нормированные координаты **(x_n, y_n)** в оконные координаты **(x_w, y_w)**.

Преобразование выполняется по формулам:

$$x_w = (x_n + 1{,}0)\left[\frac{\text{width}}{2}\right] + x;\quad y_w = (y_n + 1{,}0)\left[\frac{\text{height}}{2}\right] + x,$$

где $[\bullet]$ – целая часть числа.

Используя параметры команды **glViewPort()**, OpenGL вычисляет оконные координаты центра области вывода (o_x, o_y) по формулам

$$o_x = x + \frac{\text{width}}{2};\quad o_y = x + \frac{\text{height}}{2}.$$

При этом точка с координатами **(0, 0, 0)** располагается в *центре* области вывода.

Пусть $p_x = \text{width}$, $p_y = \text{height}$.

Тогда оконные координаты вершин можно представить в виде

$$x_w = \frac{p_x}{2}x_n + o_x;\quad y_w = \frac{p_y}{2}y_n + o_y.$$

Команда **glViewPort()** обычно используется в функции, зарегистрированной с помощью команды **glutReshapeFunc()**, которая вызывается, если пользователь изменяет размеры окна приложения.

15.8. Создание анимации в OpenGL

Для создания анимации в OpenGL необходимо выполнить следующие действия.

Установить режим двойной буферизации:

glutInitDisplayMode(GLUT_RGB|GLUT_DOUBLE).

Зарегистрировать и создать функцию обратного вызова (например, функцию с именем **IdleFunction**), которая вызывается системой, когда нет других событий.

Внутри функции **IdleFunction** описываются действия, которые влияют на вид изображения после перерисовки окна. Например, изменяется угол поворота объекта.

```
void IdleFunction( )
{
  //Необходимые действия
  glutPostRedisplay( ); //Обновить окно
}
```

В функции рисования **OnDraw( )** используется команда переключения буферов – **glutSwapBuffers**().

Для управления процессом запуска и остановки анимации можно использовать команды:

glutIdleFunc(IdleFunction) – разрешает выполнение функции IdleFunction;

glutIdleFunc(NULL) – блокирует выполнение функции IdleFunction.

15.9. Материалы и освещение

Для создания реалистичных изображений необходимо определить как свойства самого объекта, так и свойства среды, в которой он находится. Первая группа свойств включает в себя параметры материала, из которого сделан объект, способы нанесения текстуры на его поверхность, степень прозрачности объекта. Ко второй группе можно отнести количество и свойства источников света, уровень прозрачности среды, а также модель освещения. Все эти свойства можно задавать, вызывая соответствующие команды OpenGL.

15.9.1. Модель освещения

В OpenGL используется модель освещения, в соответствии с которой цвет точки определяется несколькими факторами: **свойствами материала** и **текстуры**, **величиной нормали в этой точке**, а также **положением источника света** и **наблюдателя**. Для корректного расчета освещенности в точке надо использовать единичные нормали, однако команды типа **glScale*()** могут изменять длину нормалей. Чтобы это учесть, необходимо использовать режим нормализации векторов нормалей, который включается вызовом команды **glEnable(GL_NORMALIZE)**.

Для задания глобальных параметров освещения используются команды

void glLightModel[i f] (GLenum pname, GLenum param),

void glLightModel[i f]v (GLenum pname, const GLtype *params).

Аргумент **pname** определяет, какой параметр модели **param** освещения будет настраиваться. Возможные значения аргументов **pname** и **param** приведены в табл. 15.8.

Таблица 15.8

Значения аргументов *pname* и *param*

Значение **pname**	Значение **param**
GL_LIGHT_MODEL_LOCAL_VIEWER	Параметр **param** должен быть ***булевым*** и задает положение наблюдателя. Если он равен GL_FALSE, то направление обзора считается параллельным оси *Z* вне зависимости от положения в видовых координатах. Если же он равен GL_TRUE, то наблюдатель находится в начале видовой системы координат. Это может улучшить качество освещения, но усложняет его расчет. Значение ***по умолчанию***: GL_FALSE
GL_LIGHT_MODEL_TWO_SIDE	Параметр **param** должен быть ***булевым*** и управляет режимом расчета освещенности как для лицевых, так и для обратных граней. Если он равен GL_FALSE, то освещенность рассчитывается только для лицевых граней. Если же он равен GL_TRUE, расчет проводится и для обратных граней. Значение ***по умолчанию***: GL_FALSE
GL_LIGHT_MODEL_AMBIENT	Параметр **param** должен содержать четыре целых или вещественных числа, которые определяют ***цвет фонового освещения*** даже в случае ***отсутствия*** определенных источников света. Значение ***по умолчанию:*** **(0.2, 0.2, 0.2, 1.0)**

15.9.2. Спецификация материалов

Для задания параметров текущего материала используются команды

void glMaterial[i f] (GLenum face, GLenum pname,
GLtypeparam),
void glMaterial[i f]v (GLenum face, GLenum pname,
GLtype *params).

С их помощью можно определить **рассеянный, диффузный** и **зеркальный** цвета материала, а также степень зеркального отражения и интенсивность излучения света, если объект должен светиться. Какой именно параметр будет определяться значением **param**, зависит от значения **pname**.

Возможные значения аргументов **pname** и **param** приведены в табл. 15.9.

Таблица 15.9

Значения аргументов *pname* и *param*

Значение **pname**	Значение **param**
GL_AMBIENT	Параметр **params** должен содержать четыре целых или вещественных значения цветов RGBA, которые определяют ***рассеянный цвет*** материала (цвет материала в тени). Значение ***по умолчанию***: **(0.2, 0.2, 0.2, 1.0)**
GL_DIFFUSE	Параметр **params** должен содержать четыре целых или вещественных значения цветов RGBA, которые определяют ***диффузный цвет*** материала. Значение ***по умолчанию***: **(0.8, 0.8, 0.8, 1.0)**
GL_SPECULAR	Параметр **params** должен содержать четыре целых или вещественных значения цветов RGBA, которые определяют ***зеркальный цвет*** материала. Значение ***по умолчанию***: **(0.0, 0.0, 0.0, 1.0)**
GL_SHININESS	Параметр **params** должен содержать одно целое или вещественное значение в диапазоне от **0** до **128**, которое определяет ***степень зеркального отражения*** материала. Значение ***по умолчанию***: **0**
GL_EMISSION	Параметр **params** должен содержать четыре целых или вещественных значения цветов RGBA, которые определяют ***интенсивность излучаемого света*** материала. Значение ***по умолчанию***: **(0.0, 0.0, 0.0, 1.0)**
GL_AMBIENT_AND_DIFFUSE	Это эквивалентно двум вызовам команды glMaterial*() со значением **pname** GL_AMBIENT и GL_DIFFUSE и одинаковыми значениями **params**

Из таблицы следует, что вызов команды **glMaterial[i f]()** возможен только для установки степени зеркального отражения ма-

териала (shininess). Команда **glMaterial[i f]v()** используется для задания остальных параметров.

Параметр **face** определяет тип граней, для которых задается этот материал и может принимать значения: GL_FRONT, GL_BACK или GL_FRONT_AND_BACK.

Если в сцене материалы объектов различаются лишь одним параметром, рекомендуется сначала установить нужный режим, вызвав **glEnable()** с параметром GL_COLOR_MATERIAL, а затем использовать команду

void glColorMaterial (GLenum face, GLenum pname),

где параметр **face** имеет аналогичный смысл, а параметр **pname** может принимать все перечисленные в таблице значения. После этого значения, выбранного с помощью **pname**, свойства материала для конкретного объекта (или вершины) устанавливаются вызовом команды **glColor*()**, что позволяет избежать вызовов более ресурсоемкой команды **glMaterial*()** и повышает эффективность программы.

Пример определения свойств материала:

```
float mat_dif[] = {0.8,0.8,0.8};
float mat_amb[] = {0.2, 0.2, 0.2};
float mat_spec[] = {0.6, 0.6, 0.6};
float shininess = 0.7 * 128;
…
glMaterialfv (GL_FRONT_AND_BACK,GL_AMBIENT, mat_amb);
glMaterialfv (GL_FRONT_AND_BACK,GL_DIFFUSE, mat_dif);
glMaterialfv (GL_FRONT_AND_BACK,GL_SPECULAR, mat_spec);
glMaterialf (GL_FRONT,GL_SHININESS, shininess);
```

15.9.3. Описание источников света

Определение свойств материала объекта имеет смысл, только если в сцене есть источники света. Иначе все объекты будут черными или, строго говоря, иметь цвет, равный рассеянному цвету материала, умноженному на интенсивность глобального фонового освещения (см. команду **glLightModel**). Добавить в сцену источник света можно с помощью команд

void glLight[if] (GLenum light, GLenum pname, GLfloat param),

void glLight[i f]v (GLenum light, GLenum pname, GLfloat *params).

Параметр **light** однозначно определяет источник света. Он выбирается из набора специальных символических имен вида

GL_LIGHTi, где i должно лежать в диапазоне от 0 до константы GL_MAX_LIGHT, которая обычно не превышает восьми.

Параметры **pname** и **params** имеют смысл, аналогичный команде **glMaterial**. Возможные взаимные значения аргументов **pname** и **param** приведены в табл. 15.10.

Таблица 15.10

Значения аргументов *pname* и *param*

Значение **pname**	Значение **param**
GL_SPOT_EXPONENT	Параметр **param** должен содержать целое или вещественное число от **0** до **128**, задающее распределение интенсивности света. Этот параметр описывает уровень сфокусированности источника света. Значение ***по умолчанию***: **0** (рассеянный свет)
GL_SPOT_CUTOFF	Параметр **param** должен содержать целое или вещественное число между **0** и **90** или равное **180**, которое определяет максимальный угол разброса света. Значение этого параметра есть половина угла в вершине конусовидного светового потока, создаваемого источником. Значение ***по умолчанию***: **180** (рассеянный свет)
GL_AMBIENT	Параметр **params** должен содержать четыре целых или вещественных значения цветов RGBA, которые определяют цвет фонового освещения. Значение ***по умолчанию***: **(0.0, 0.0, 0.0, 1.0)**
GL_DIFFUSE	Параметр **params** должен содержать четыре целых или вещественных значения цветов RGBA, которые определяют цвет диффузного освещения. Значение ***по умолчанию***: **(1.0, 1.0, 1.0, 1.0)** для GL_LIGHT0 и **(0.0, 0.0, 0.0, 1.0)** для остальных
GL_SPECULAR	Параметр **params** должен содержать четыре целых или вещественных значения цветов RGBA, которые определяют цвет зеркального отражения. Значение ***по умолчанию***: **(1.0, 1.0, 1.0, 1.0)** для GL_LIGHT0 и **(0.0, 0.0, 0.0, 1.0)** для остальных

Окончание табл. 15.10

Значение **pname**	Значение **param**
GL_POSITION	Параметр **params** должен содержать четыре целых или вещественных числа, которые определяют положение источника света. Если значение компоненты **w = 0.0**, то источник считается бесконечно удаленным и при расчете освещенности учитывается только ***направление на точку*** **(x, y, z)**, в противном случае считается, что источник расположен ***в точке*** **(x, y, z, w)**. В первом случае ослабления света при удалении от источника не происходит, т. е. источник считается бесконечно удаленным. Значение ***по умолчанию:*** **(0.0, 0.0, 1.0, 0.0)**
GL_SPOT_DIRECTION	Параметр **params** должен содержать четыре целых или вещественных числа, которые определяют направление света. Значение ***по умолчанию:*** (**0.0, 0.0, –1.0, 1.0)**. Эта характеристика источника имеет смысл, если значение GL_SPOT_CUTOFF ***отлично*** от 180 (которое задано по умолчанию)
GL_CONSTANT_ATTENUATION, GL_LINEAR_ATTENUATION, GL_QUADRATIC_ATTENUATION	Параметр **param** задает значение одного из трех приведенных коэффициентов, определяющих ослабление интенсивности света при удалении от источника. Допускаются только неотрицательные значения. Если источник не является направленным (см. GL_POSITION), то ослабление обратно пропорционально сумме: $att_{constant} + att_{linear}d + att_{quadratic}d^2$, где d – расстояние между источником света и освещаемой им вершиной; $att_{constant}$, att_{linear} и $att_{quadratic}$ равны параметрам, заданным с помощью констант GL_CONSTANT_ATTENUATION, GL_LINEAR_ATTENUATION и GL_QUADRATIC_ ATTENUATION соответственно. ***По умолчанию*** эти параметры задаются тройкой **(1, 0, 0)**, и фактически ослабления не происходит

При изменении положения источника света следует учитывать следующий факт: в OpenGL источники света являются объектами во многом такими же, как многоугольники и точки. На них распространяется основное правило обработки координат в OpenGL – параметры, описывающее положение в пространстве, преобразуются текущей модельно-видовой матрицей в момент формирования объекта, т. е. в момент вызова соответствующих команд OpenGL. Таким образом, формируя источник света одновременно с объектом сцены или камерой, его можно привязать к этому объекту. Или, наоборот, сформировать стационарный источник света, который будет оставаться на месте, пока другие объекты перемещаются.

Если **тип** и **положение** источника света не задаются, параметр GL_POSITION принимает значения по умолчанию (**0.0, 0.0, 1.0, 0.0**), что указывает на удаленный источник, лучи света от которого распространяются по отрицательному направлению оси **z**.

Положение источника света включается в описание сцены и вместе с положениями объектов преобразовывается в координаты наблюдения с помощью матриц геометрического преобразования и преобразования точки наблюдения. Следовательно, если требуется зафиксировать источник света относительно объектов на сцене, его положение задается после указания в программе геометрических преобразований и преобразований точки наблюдения. Однако если требуется, чтобы источник света двигался при изменении точки наблюдения, его положение задается перед спецификацией преобразования точки наблюдения. Кроме того, если необходимо, чтобы источник света двигался вокруг стационарной сцены, к его положению можно применить преобразование трансляции или вращения.

Для использования освещения сначала надо установить соответствующий режим вызовом команды **glEnable(GL_LIGHTNING)**, а затем включить нужный источник командой **glEnable (GL_LIGHTi)**. Еще раз обратим внимание на то, что при выключенном освещении цвет вершины равен текущему цвету, который задается командами **glColor*()**. При включенном освещении цвет вершины вычисляется исходя из информации о материале, нормалях и источниках света. При выключении освещения визуализация происходит быстрее, однако в таком случае приложение должно само рассчитывать цвета вершин.

В приложении 2 и 3 приведены примеры создания приложений с использованием библиотеки OpenGL.

ПРИЛОЖЕНИЕ 1

Ниже приведен пример однооконного приложения Windows (без поддержки архитектуры «документ – вид»), созданного с использованием библиотеки классов MFC.

После запуска приложения появляется пустое окно. После выбора команды меню «Фигуры / Фигура 1» на экране появляется изображение фигуры (рис. П.1). При нажатии левой клавиши мыши фигура начинает вращаться против часовой стрелки, а при нажатии правой клавиши вращение прекращается.

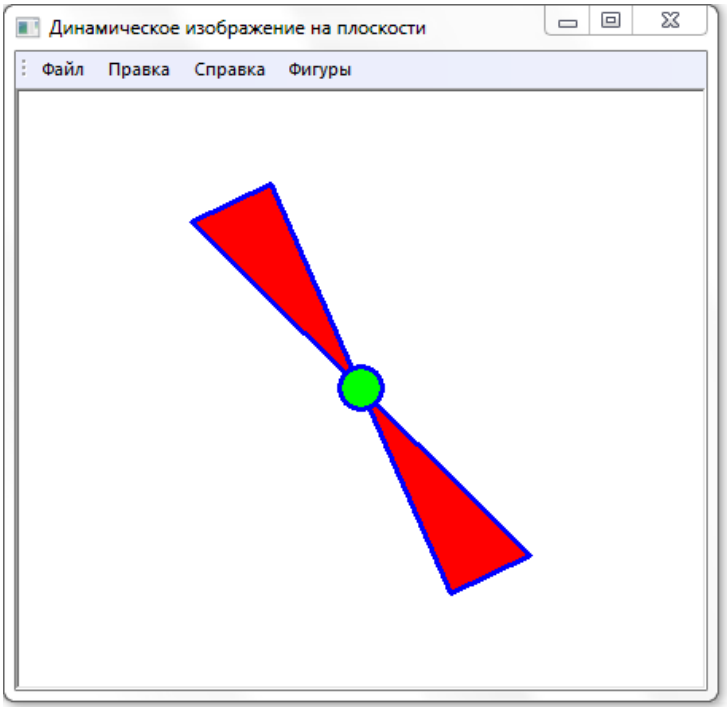


Рис. П.1

Ниже приводятся листинги файлов программы.

В файлах **CMatrix.h** и **CMatrix.cpp** приводится описание класса **CMatrix**, предназначенного для работы с матрицами.

Файл **CMatrix.h**

```
#define CMATRIXH
#define CMATRIXH 1
Class CMatrix
```

```
{
  double **array;
  int n_rows;    // Число строк матрицы
  int n_cols;    // Число столбцов матрицы
public:
  CMatrix();     // Конструктор по умолчанию (1 на 1)

  CMatrix(int, int);  // Конструктор
  CMatrix(int);   // Конструктор - вектора (один столбец)
  CMatrix(const CMatrix&); // Конструктор копирования
  ~CMatrix();
  double &operator()(int, int); // Выбор элемента
                                //матрицы
                                // по индексу
  double &operator()(int);  // Выбор элемента вектора
                            //по индексу
  CMatrix operator-();     // Оператор "-"
  CMatrix operator = (const CMatrix&);   // Оператор
                                // "Присвоить": M1=M2
  CMatrix operator*(CMatrix&);  // Оператор
                             // "Произведение": M1*M2
  CMatrix operator+(CMatrix&);  // Оператор "+": M1+M2
  CMatrix operator-(CMatrix&);  // Оператор "-": M1-M2
  CMatrix operator+(double);    // Оператор "+": M+a
  CMatrix operator-(double);    // Оператор "-": M-a
  int rows()const{return n_rows;}; // Возвращает число
                                   //строк
  int cols()const{return n_cols;};// Возвращает число
                                  // столбцов
  CMatrix Transp();   // Возвращает матрицу,
                      //транспонированную к текущей
  CMatrix GetRow(int);     // Возвращает строку по
                           //номеру
  CMatrix GetRow(int, int, int);
  CMatrix GetCol(int);     // Возвращает столбец по
                           //номеру
  CMatrix GetCol(int, int, int);
  CMatrix RedimMatrix(int, int); // Изменяет размер
                      //матрицы с уничтожением данных
  CMatrix RedimData(int, int); // Изменяет размер
                               //матрицы
     //с сохранением данных, которые можно сохранить
  CMatrix RedimMatrix(int); // Изменяет размер
                            //матрицы
                            //с уничтожением данных
  CMatrix RedimData(int); // Изменяет размер матрицы
     //с сохранением данных, которые можно сохранить
```

```
 double MaxElement();// Максимальный элемент матрицы
 double MinElement();// Минимальный элемент матрицы
};
#endif
```

Файл **CMatrix.cpp**

```
#include"stdafx.h"
//#include "CMatrix.h"
CMatrix::CMatrix()
{
n_rows = 1;
n_cols = 1;
array = new double*[n_rows];
for(int i = 0; i<n_rows; i++) array[i] =
                      new double[n_cols];
for(int i = 0; i<n_ rows; i++)
for(int j = 0; j<n_cols; j++) array[i][j] = 0;
 }

//-------------------------------------------------
CMatrix::CMatrix(int Nrow, int Ncol)
// Nrow - число строк
// Ncol - число столбцов
{
n_rows = Nrow;
n_cols = Ncol;
array = new double*[n_rows];
for(int i = 0; i<n_rows; i++) array[i]=
                           new double[n_cols];
for(int i = 0; i<n_rows;i++)
for(int j = 0; j<n_cols; j++) array[i][j] = 0;
 }

//-------------------------------------------------
CMatrix::CMatrix(int Nrow)  //Вектор
// Nrow – число строк
{
n_rows = Nrow;
n_cols = 1;
array = new double*[n_rows];
for(int i = 0; i<n_rows; i++) array[i] =
                       new double[n_cols];
for(int i = 0; i<n_rows; i++)
for(int j = 0; j<n_cols; j++) array[i][j] = 0;
 }

//-------------------------------------------------
```

```
CMatrix::~CMatrix()
{
for(int i = 0; i<n_rows; i++) delete array[i];
delete array;
}

//---------------------------------------------------
double &CMatrix::operator()(int i, int j)
// i - номер строки
// j - номер столбца
{
if ((i>n_rows-1)||(j>n_cols - 1)) // проверка
                                  //выхода за диапазон
{
 TCHAR* error = _T("CMatrix::operator(int, int): выход
                    индекса за границу диапазона ");
 MessageBox(NULL, error, _T("Ошибка"), MB_ICONSTOP);
 exit(1);
}
Return array[i][j];
}

//---------------------------------------------------
double &CMatrix::operator()(int i)
// i - номер строки для вектора
{
if (n_cols>1)     // Число столбцов больше одного
{
 char* error = "CMatrix::operator(int): объект не
              вектор число столбцов больше 1 ";
 MessageBox(NULL, error, "Ошибка",MB_ICONSTOP);
 exit(1);
 }
if (i>n_rows-1)     //  проверка выхода за диапазон
{
 TCHAR* error=TEXT("CMatrix::operator(int): выход
                   индекса за границу диапазона ");
 MessageBox(NULL, error, TEXT("Ошибка"), MB_ICONSTOP);
 exit(1);
 }
return array[i][0];
}

//---------------------------------------------------
CMatrix CMatrix::operator-()
// Оператор -M
{
```

```
 CMatrix Temp(n_rows, n_cols);
 for(int i= 0; i<n_rows; i++)
      for(int j= 0;j<n_cols; j++) Temp(i,j) =
                                   -array[i][j];
 return Temp;
}

//---------------------------------------------------
CMatrix CMatrix::operator+(CMatrix& M)
// Оператор M1+M2
{
 int bb = (n_rows==M.rows())&&(n_cols==M.cols());
 if(!bb)
 {
      char* error = "CMatrix::operator(+):
               несоответствие размерностей матриц ";
      MessageBox(NULL, error, "Ошибка", MB_ICONSTOP);
      exit(1);
 }
 CMatrix Temp(*this);
 for(int i= 0; i<n_rows; i++)
      for(int j= 0;j<n_cols; j++) Temp(i, j)+=M(i, j);
 return Temp;
}

//---------------------------------------------------
CMatrix CMatrix::operator-(CMatrix& M)
// Оператор M1-M2
{
 int bb = (n_rows==M.rows())&&(n_cols==M.cols());
 if(!bb)
 {
      char* error = "CMatrix::operator(-):
               несоответствие размерностей матриц ";
      MessageBox(NULL, error, "Ошибка", MB_ICONSTOP);
      exit(1);
 }
 CMatrix Temp(*this);
 for(int i= 0; i<n_rows; i++)
      for(int j= 0; j<n_cols; j++) Temp(i,j)-=M(i,j);
 return Temp;
}

//---------------------------------------------------
CMatrix CMatrix::operator*(CMatrix& M)
// Умножение на матрицу M
{
```

```
double sum;
int nn = M.rows();
int mm = M.cols();
CMatrix Temp(n_rows, mm);
if (n_cols==nn)
  {
for (int i= 0; i<n_rows; i++)
for (int j= 0; j<mm; j++)
{
sum = 0;
for (int k= 0; k<n_cols; k++) sum+=(*this)(i,k)*M(k,j);
Temp(i, j) = sum;
          }
}
else
 {
TCHAR* error = TEXT("CMatrix::operator*: несоответствие
размерностей матриц ");
MessageBox(NULL, error, TEXT("Ошибка"), MB_ICONSTOP);
exit(1);
  }
return Temp;
}

//---------------------------------------------------
CMatrix CMatrix::operator = (const CMatrix& M)
// Оператор присваивания M1 = M
{
if (this==&M) return *this;
int nn = M.rows();
int mm = M.cols();
if ((n_rows==nn)&&(n_cols==mm))
 {
for (int i= 0; i<n_rows; i++)
for (int j= 0; j<n_cols;j++) array[i][j]=M.array[i][j];
}
else// для ошибки размерностей
  {
TCHAR* error=TEXT("CMatrix::operator=: несоответствие
                   размерностей матриц");
MessageBox(NULL, error, TEXT("Ошибка"), MB_ICONSTOP);
exit(1);
  }
return *this;
}

//---------------------------------------------------
```

```
CMatrix::CMatrix(const CMatrix &M)
// Конструктор копирования
{
 n_rows = M.n_rows;
 n_cols = M.n_cols;
 array = new double*[n_rows];
 for(int i = 0; i<n_rows; i++) array[i]=
                                  new double[n_cols];
 for(int i = 0; i<n_rows; i++)
for(int j = 0;j<n_cols; j++) array[i][j]=M.array[i][j];
}

//---------------------------------------------------
CMatrix CMatrix::operator+(double x)
// Оператор M+x, где M - матрица, x - число
{
CMatrix Temp(*this);
for(int i = 0; i<n_rows; i++)
 for(int j= 0; j<n_cols; j++) Temp(i, j)+ = x;
return Temp;
}

//---------------------------------------------------
CMatrix CMatrix::operator-(double x)
// Оператор M+x, где M - матрица, x - число
{
CMatrix Temp(*this);
for(int i = 0; i<n_rows; i++)
 for(int j= 0; j<n_cols; j++) Temp(i, j)-=x;
return Temp;
}

//---------------------------------------------------
CMatrix CMatrix::Transp()
// Возвращает матрицу, транспонированную к (*this)
{
CMatrix Temp(n_cols,n_rows);
for (int i = 0; i<n_cols; i++)
for (int j = 0; j<n_rows; j++) Temp(i, j)=array[j][i];
return Temp;
}

//---------------------------------------------------
CMatrix CMatrix::GetRow(int k)
// Возвращает строку матрицы по номеру k
{
 if(k>n_rows-1)
```

```
 {
      char* error = "CMatrix::GetRow(int k): параметр
      k превышает число строк ";
      MessageBox(NULL, error, "Ошибка", MB_ICONSTOP);
      exit(1);
 }
 CMatrix M(1, n_cols);
 for(int i= 0; i<n_cols; i++)M(0, i) = (*this)(k, i);
 return M;
}

//---------------------------------------------------
CMatrix CMatrix::GetRow(int k, int n, int m)
// Возвращает подстроку из строки матрицы с номером k
// n - номер первого элемента в строке
// m - номер последнего элемента в строке
{
 int b1 = (k>=0) && (k<n_rows);
 int b2 = (n>=0) && (n<=m);
 int b3 = (m>=0) && (m<n_cols);
 int b4 = b1 && b2 && b3;
 if(!b4)
 {
      char* error="CMatrix::GetRow(int k,int n, int m):
                   ошибка в параметрах ";
      MessageBox(NULL, error, "Ошибка", MB_ICONSTOP);
      exit(1);
 }
 int nCols = m - n + 1;
 CMatrix M(1, nCols);
 for(int i = n; i<=m; i++)M(0, i-n) = (*this)(k, i);
 return M;
}

//---------------------------------------------------
CMatrix CMatrix::GetCol(int k)
// Возвращает столбец матрицы по номеру k
{
 if(k>n_cols - 1)
 {
      char* error = "CMatrix::GetCol(int k): параметр k
                     превышает число столбцов ";
      MessageBox(NULL, error, "Ошибка", MB_ICONSTOP);
      exit(1);
 }
 CMatrix M(n_rows, 1);
 for(int i = 0; i<n_rows; i++)M(i, 0) = (*this)(i, k);
```

```
 return M;
}

//---------------------------------------------------
CMatrix CMatrix::GetCol(int k, int n, int m)
// Возвращает подстолбец из столбца матрицы с номером k
// n - номер первого элемента в столбце
// m - номер последнего элемента в столбце
{
 int b1 = (k>=0) && (k<n_cols);
 int b2 = (n>=0) && (n<=m);
 int b3 = (m>=0) && (m<n_rows);
 int b4 = b1 && b2 && b3;
 if(!b4)
 {
      char* error="CMatrix::GetCol(int k,int n, int m):
                   ошибка в параметрах ";
      MessageBox(NULL, error, "Ошибка", MB_ICONSTOP);
      exit(1);
 }
 int nRows = m - n + 1;
 CMatrix M(nRows, 1);
 for(int i = n; i<=m; i++)M(i-n, 0) = (*this)(i, k);
 return M;
}

//---------------------------------------------------
CMatrix CMatrix::RedimMatrix(int NewRow, int NewCol)
// Изменяет размер матрицы с уничтожением данных
// NewRow - новое число строк
// NewCol - новое число столбцов
{
 for(int i = 0; i<n_rows; i++) delete array[i];
 delete array;
 n_rows = NewRow;
 n_cols = NewCol;
 array = new double*[n_rows];
 for(int i=0; i<n_rows;i++) array[i]=
                                  new double[n_cols];
 for(int i = 0; i<n_rows; i++)
      for(int j = 0; j<n_cols; j++) array[i][j] = 0;
 return (*this);
}

//---------------------------------------------------
CMatrix CMatrix::RedimData(int NewRow, int NewCol)
// Изменяет размер матрицы с сохранением данных,
```

```
// которые можно сохранить
// NewRow - новое число строк
// NewCol - новое число столбцов
{
 CMatrix Temp = (*this);
 this->RedimMatrix(NewRow, NewCol);
 int min_rows =
Temp.rows()<(*this).rows()?Temp.rows():(*this).rows();
 intmin_cols =
Temp.cols()<(*this).cols()?Temp.cols():(*this).cols();
 for(int i = 0; i<min_rows; i++)
      for(int j = 0; j<min_cols; j++)
                          (*this)(i, j)=Temp(i, j);
 return (*this);
}

//---------------------------------------------------
CMatrix CMatrix::RedimMatrix(int NewRow)
// Изменяет размер матрицы с уничтожением данных
// NewRow - новое число строк
// NewCol = 1
{
 for(int i = 0; i<n_rows; i++) delete array[i];
 delete array;
 n_rows = NewRow;
 n_cols = 1;
 array = new double*[n_rows];
 for(int i = 0;i<n_rows;i++) array[i]=
                                  new double[n_cols];
 for(int i = 0; i<n_rows; i++)
      for(int j = 0; j<n_cols; j++) array[i][j] = 0;
 return (*this);
}

//---------------------------------------------------
CMatrix CMatrix::RedimData(int NewRow)
// Изменяет размер матрицы с сохранением данных,
// которые можно сохранить
// NewRow - новое число строк
// NewCol = 1
{
 CMatrix Temp = (*this);
 this->RedimMatrix(NewRow);
 intmin_rows =
Temp.rows()<(*this).rows()?Temp.rows():(*this).rows();
 for(int i = 0; i<min_rows; i++)(*this)(i)=Temp(i);
 return (*this);
}
```

```
//-------------------------------------------------
double CMatrix::MaxElement()
// Максимальное значение элементов матрицы
{
 double max = (*this)(0, 0);
 for(int i = 0; i<(this->rows()); i++)
 for(int j = 0; j<(this->cols()); j++)
 {
     if ((*this)(i, j)>max) max = (*this)(i, j);
 }
 return max;
}

//-------------------------------------------------
double CMatrix::MinElement()
// Минимальное значение элементов матрицы
{
 double min = (*this)(0, 0);
 for(int i = 0; i<(this->rows()); i++)
 for(int j = 0; j<(this->cols()); j++)
 {
   if ((*this)(i,j)<min) min = (*this)(i,j);
 }
 return min;
}
```

В файлах **LibGraph.h** и **LibGraph.cpp** описаны функции для аффинных преобразований на плоскости, для пересчета координат из мировой системы в оконную и некоторые структуры данных.

Файл **LibGraph.h**

```
#ifndefLIBGRAPH
#define LIBGRAPH 1
Const double pi = 3.14159;

Struct CSizeD
{
double cx;
double cy;
};

//-------------------------------------------------
Struct CRectD
 {
```

```
double left;
double top;
double right;
double bottom;
CRectD(){left = top = right = bottom = 0;};
CRectD(double l, double t,double r,double b);
void SetRectD(double l, double t, double r,double b);
CSizeD SizeD();
 };

//-------------------------------------------------
CMatrix SpaceToWindow(CRectD&rs, CRect&rw);
//Возвращает матрицу пересчета координат из
//мировых в оконные
//rs - область в мировых координатах - double
//rw - область в оконных координатах - int

//-------------------------------------------------
CMatrix CreateTranslate2D(double dx, double dy);
//Формирует матрицу для преобразования координат
//объекта при его смещении на dx по оси X и на dy
//по оси Y в фиксированной системе координат
// --- ИЛИ ---
//Формирует матрицу для преобразования координат
//объекта при смещении начала системы координат на
// -dx по оси X и на -dy по оси Y
//при фиксированном положении объекта

//-------------------------------------------------
CMatrix CreateRotate2D(double fi);
//Формирует матрицу для преобразования координат
//объекта при его повороте на угол fi (при fi>0 против
// часовой стрелки)в фиксированной системе координат
// --- ИЛИ ---
//Формирует матрицу для преобразования координат
// объекта при повороте начала системы координат на
// угол -fi при //фиксированном положении объекта
// fi - угол в градусах

#endif
```

Файл **LibGraph.cpp**

```
#include"stdafx.h"

CRectD::CRectD(double l, double t, double r, double b)
{
```

```
left = l;
top = t;
right = ;
bottom = b;
}
//-----------------------------------------------
void CRectD::SetRectD(double l, double t, double r,
                      double b)
{
left = l;
top = t;
right = r;
bottom = b;
}

//-----------------------------------------------
CSizeD CRectD::SizeD()
{
CSizeDcz;
cz.cx = fabs(right - left);// Ширина прямоугольной
                           //области
cz.cy = fabs(top - bottom);  // Высота прямоугольной
                             // области
return cz;
}

//-----------------------------------------------
CMatrix SpaceToWindow(CRectD& RS,  CRect& RW)
//Возвращает матрицу пересчета координат из
//мировых в оконные
//RS - область в мировых координатах - double
//RW - область в оконных координатах - int
{
CMatrix M(3, 3);
CSize sz = RW.Size();// Размер области в ОКНЕ
int dwx = sz.cx;// Ширина
int dwy = sz.cy;// Высота
CSizeD szd = RS.SizeD(); // Размер области в МИРОВЫХ
                         // координатах
double dsx = szd.cx; // Ширина в мировых координатах
double dsy = szd.cy; // Высота в мировых координатах
double kx = (double)dwx/dsx;   // Масштаб по x
double ky = (double)dwy/dsy;   // Масштаб по y

M(0, 0) = kx;  M(0, 1)=0;
M(0, 2) =(double)RW.left - kx*RS.left;
M(1, 0) = 0;  M(1, 1) = -ky;
```

```
M(1, 2) = (double)RW.bottom + ky*RS.bottom;
M(2, 0) = 0;   M(2, 1) = 0;    M(2, 2) = 1;
return M;
 }

//---------------------------------------------------
CMatrix CreateTranslate2D(double dx, double dy)
//Формирует матрицу для преобразования координат
//объекта при его смещении на dx по оси X и на dy по
// оси Y в фиксированной системе координат
// --- ИЛИ ---
//Формирует матрицу для преобразования координат
//объекта при смещении начала системы координат
//на -dx оси X и на –dy по оси Y при фиксированном
//положении объекта
{
CMatrixTM(3, 3);
TM(0, 0) = 1; TM(0, 2) = dx;
TM(1, 1) = 1;  TM(1, 2) = dy;
TM(2, 2) = 1;
return TM;
}

//---------------------------------------------------
CMatrix CreateRotate2D(double fi)
//Формирует матрицу для преобразования координат
//объекта при его повороте на угол fi (при fi>0 против
//часовой стрелки)в фиксированной системе координат
// --- ИЛИ ---
//Формирует матрицу для преобразования координат
//объекта при повороте начала системы координат
// на угол –fi при
//фиксированном положении объекта
//fi - угол в градусах
{
Double fg = fmod(fi, 360.0);
Double ff = (fg/180.0)*pi; // Перевод в радианы
CMatrixRM(3, 3);
RM(0, 0) = cos(ff); RM(0,1) = -sin(ff);
RM(1, 0) = sin(ff);  RM(1,1) = cos(ff);
RM(2, 2) = 1;
return RM;
}
```

В файлах **Figure.h** и **Figure.cpp** описан класс CFigure, экземпляром которого и является фигура, которая отображается на экране.

Файл **Figure.h**

```
#pragma once
class CFigure
{
 double fi,h;  //Параметры фигуры
 double Radius;
 CMatrix CoordsRS; //Координаты вершин фигуры в МСК
 CMatrix CoordsRW; //Координаты вершин фигуры в ОСК
                   //(оконная система координат)
 CMatrix KV;     //Матрица для пересчета координат
                 //из МСК в ОСК
 CMatrix V;
 CRectD RS;   //Область в мировых координатах
 CRect RW;    //Область в окне для рисования фигуры
 CRect r;     //Область в окне для рисования круга
 CRect rect;
 CPoint pts[4];
 void SetParams(void);
 public:
 void Set_fi(double fi);
 void Set_h(double h);
 void SetWindowRect(CRect &RS);
 void RotateFig(double fix);
 void Draw(CDC& dc);
 CFigure(void);
 ~CFigure(void);
};
```

Файл **Figure.cpp**

```
#include "StdAfx.h"

void CFigure::SetParams(void)
{
 double fix=(fi/180.0)*pi;
 double x=h*tan(fix);
//Заполняем матрицу данных - координаты вершин фигуры
 CoordsRS(0,0)=-x;
 CoordsRS(1,0)=h;
 CoordsRS(0,1)=x;
 CoordsRS(1,1)=h;
 CoordsRS(0,2)=-x;
 CoordsRS(1,2)=-h;
 CoordsRS(0,3)=x;
 CoordsRS(1,3)=-h;
 for(int i=0;i<4;i++) CoordsRS(2,i)=1;
```

```
 double d=sqrt(x*x+h*h);
 RS.SetRectD(-d,d,d,-d); //Область в МСК для фигуры
 KV=SpaceToWindow(RS,RW);
 CoordsRW=KV*CoordsRS;  //Координаты фигуры в ОСК

 pts[0].x =(int)CoordsRW(0,0);
 pts[0].y =(int)CoordsRW(1,0);

 pts[1].x =(int)CoordsRW(0,1);
 pts[1].y =(int)CoordsRW(1,1);

 pts[2].x =(int)CoordsRW(0,2);
 pts[2].y =(int)CoordsRW(1,2);

 pts[3].x =(int)CoordsRW(0,3);
 pts[3].y =(int)CoordsRW(1,3);

 V(0)=0; V(1)=0; // Точка (0,0) - координаты центра
                 //фигуры в МСК
 V=KV*V;          // Координаты центра фигуры в ОСК
 int x0=(int)V(0);   // x - кордината центра
                     //фигуры в ОСК
 int y0=(int)V(1);   // y - кордината центра
                     //фигуры в ОСК

 V(0)=0; V(1)=Radius; //Точка (x=0, y=Radius) в МСК
 V=KV*V;              //Точка (x=0, y=Radius) в ОСК
 int yR=(int)V(1);
 int m=abs(yR-y0);

 r.SetRect(x0-m,y0-m,x0+m,y0+m);  //Область в окне
                                  //для круга

}

//-------------------------------------------------
void CFigure::Set_fi(double fi)
{
 this->fi=fi;
 SetParams();
}
//-------------------------------------------------
void CFigure::Set_h(double h)
{
 this->h=h;
 SetParams();
}
```

```
//-------------------------------------------------
void CFigure::SetWindowRect(CRect &RW)
{
 this->RW.CopyRect(&RW);
 this->RW.DeflateRect(50,50,50,50); //Уменьшаем размер
 SetParams();
}

//-------------------------------------------------
void CFigure::RotateFig(double fix)
{
 CMatrix MR=CreateRotate2D(fix);
 CoordsRS=MR*CoordsRS;  //Координаты фигуры в МСК
                        //после поворота
 CoordsRW=KV*CoordsRS;  //Координаты фигуры в ОСК
 pts[0].x =(int)CoordsRW(0,0);
 pts[0].y =(int)CoordsRW(1,0);

 pts[1].x =(int)CoordsRW(0,1);
 pts[1].y =(int)CoordsRW(1,1);

 pts[2].x =(int)CoordsRW(0,2);
 pts[2].y =(int)CoordsRW(1,2);

 pts[3].x =(int)CoordsRW(0,3);
 pts[3].y =(int)CoordsRW(1,3);
}

//-------------------------------------------------
void CFigure::Draw(CDC& dc)
{
 CPen Pen(PS_SOLID, 3, RGB(0, 0, 255));
 CPen* pOldPen = dc.SelectObject(&Pen);

 CBrush Brush(RGB(255, 0, 0));
 CBrush Brush1(RGB(0, 255, 0));
 CBrush* pOldBrush = dc.SelectObject(&Brush);

 dc.Polygon(pts, 4);

 pOldBrush = dc.SelectObject(&Brush1);
 dc.Ellipse(r);

 dc.SelectObject(pOldPen);
 dc.SelectObject(pOldBrush);
}

//-------------------------------------------------
```

```
CFigure::CFigure(void)
{
 CoordsRS.RedimMatrix(3,4);
 CoordsRW.RedimMatrix(3,4);
 V.RedimMatrix(3,1);
 KV.RedimMatrix(3,3);
 V(2)=1;
 fi=10; //Градусы
 h=10;
 Radius=h/10;
 RW.SetRect(0,0,100,100); //Область в окне для фигуры
 SetParams();
}

//-------------------------------------------------
CFigure::~CFigure(void)
{
}
```

Файлы **ChildView.h** и **ChildView.cpp** созданы средой разработки VisualStudio 2010 и содержат обработчики команд меню и нажатий клавиш мыши.

Файл **ChildView.h**

```
// ChildView.h :интерфейс класса CChildView
#pragmaonce
// окно CChildView
class CChildView : publicCWnd
{
// Создание
public:
 CChildView();
// Атрибуты
public:
 CFigure Fig;
 int Index;
 int dT_Timer;
 double fix;
// Операции
public:

// Переопределение
 protected:
 virtual BOOL PreCreateWindow(CREATESTRUCT & cs);

// Реализация
```

```
public:
  virtual ~CChildView();

  // Созданные функции схемы сообщений
protected:
  afx_msg void OnPaint();
  DECLARE_MESSAGE_MAP()
public:
  afx_msg void OnFigure1();
  afx_msg void OnSize(UINT nType, int cx, int cy);
  afx_msg void OnTimer(UINT_PTR nIDEvent);
  afx_msg void OnLButtonDown(UINT nFlags, CPoint point);
  afx_msg void OnRButtonDown(UINT nFlags, CPoint point);
};
```

Файл **ChildView.cpp**

```
// ChildView.cpp :реализация класса CChildView

#include "stdafx.h"
#include "PrimZF1.h"
#include "ChildView.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#endif

// CChildView
CChildView::CChildView()
{
  Index = 0;
}
CChildView::~CChildView()
{
}
BEGIN_MESSAGE_MAP(CChildView, CWnd)
  ON_WM_PAINT()
  ON_COMMAND(ID_32771, & CChildView::OnFigure1)
  ON_WM_SIZE()
  ON_WM_TIMER()
  ON_WM_LBUTTONDOWN()
  ON_WM_RBUTTONDOWN()
END_MESSAGE_MAP()

// обработчики сообщений CChildView
BOOL CChildView::PreCreateWindow(CREATESTRUCT & cs)
{
```

```
     if (!CWnd::PreCreateWindow(cs))
          return FALSE;

     cs.dwExStyle |= WS_EX_CLIENTEDGE;
     cs.style& = ~WS_BORDER;
     cs.lpszClass = AfxRegis-
terWndClass(CS_HREDRAW|CS_VREDRAW|CS_DBLCLKS,
          ::LoadCursor(NULL, IDC_ARROW),
   reinterpret_cast<HBRUSH>(COLOR_WINDOW+1), NULL);
     return TRUE;
   }

   Void CChildView::OnPaint()
   {
     CPaintDCdc(this);
          if(Index==1)Fig.Draw(dc); //Отображение фигуры
   }

   Void CChildView::OnFigure1()// Выбор пункта меню
   {
     CRectRWX;
     CString strNew="Динамическое изображение
                    на плоскости";
     CFrameWnd* pWnd = GetParentFrame();
     pWnd->SetWindowTextA(strNew);
     this->GetClientRect(RWX);
     Fig.SetWindowRect(RWX);
     Fix = 5;
     dT_Timer = 100;
     Index = 1;
     this->Invalidate();
     }

   Void CChildView::OnSize(UINT nType, int cx, int cy)
   {
     CWnd::OnSize(nType, cx, cy);
   //Установка новой области для рисования в окне
   //после изменения его размеров
     CRect RWX;
     this->GetClientRect(RWX);
     Fig.SetWindowRect(RWX);
     }

   Void CChildView::OnTimer(UINT_PTR nIDEvent)
   {
     Fig.RotateFig(fix); //Поворот фигуры
     Invalidate();       //Перерисовка окна
```

```
  CWnd::OnTimer(nIDEvent);
}

Void CChildView::OnLButtonDown(UINT nFlags, CPoint point)
//Нажатие левой клавиши мыши – запуск таймера
{
  SetTimer(1, dT_Timer, NULL); //Запуск таймера
  CWnd::OnLButtonDown(nFlags, point);
}

Void CChildView::OnRButtonDown(UINT nFlags, CPoint point)
//Нажатие правой клавиши мыши – остановка таймера
{
  KillTimer(1); //Остановка таймера
  CWnd::OnRButtonDown(nFlags, point);
}
```

Файл **stdafx.h**, в котором подключаются созданные нами библиотечные файлы

```
// stdafx.h: включите файл для добавления стандартных
// системных файлов или конкретных файлов проектов,
// часто используемых, но редко изменяемых

#pragmaonce
#ifndef _SECURE_ATL
#define _SECURE_ATL 1
#endif

#ifndef VC_EXTRALEAN
#define VC_EXTRALEAN// Исключите редко используемые
// компоненты из заголовков Windows
#endif

#include "targetver.h"

#define _ATL_CSTRING_EXPLICIT_CONSTRUCTORS
// некоторые конструкторы CString будут явными

// отключает функцию скрытия некоторых общих и часто
//пропускаемых предупреждений MFC
#define _AFX_ALL_WARNINGS

#include<afxwin.h>// основные и стандартные компо-
ненты MFC
#include<afxext.h>// расширения MFC
#ifndef _AFX_NO_OLE_SUPPORT
```

```
#include<afxdtctl.h>//поддержка MFC для типовых
//элементов управления Internet Explorer 4
#endif
#ifndef _AFX_NO_AFXCMN_SUPPORT
#include<afxcmn.h>//поддержка MFC для типовых
                    //элементов управления Windows
#endif// _AFX_NO_AFXCMN_SUPPORT

#include<afxcontrolbars.h>//поддержка MFC для ленты
//панелей управления
// Подключение собственных файлов
#include<math.h>
#include "CMatrix.h"
#include "LibGraph.h"
#include "Figure.h"
```

Приложение содержит и другие файлы, которые созданы средой разработки Visual Studio 2010 при создании ***однооконного приложения***. В данном приложении эти файлы не изменялись, поэтому их отображение здесь нецелесообразно.

ПРИЛОЖЕНИЕ 2

Ниже приведен пример приложения Windows, созданного с использованием библиотеки OpenGL.

При запуске программы:

– при нажатии клавиши с латинской буквой «a» (или «A») в окне отображается пирамида без удаления нелицевых граней (рис. П.2.1);

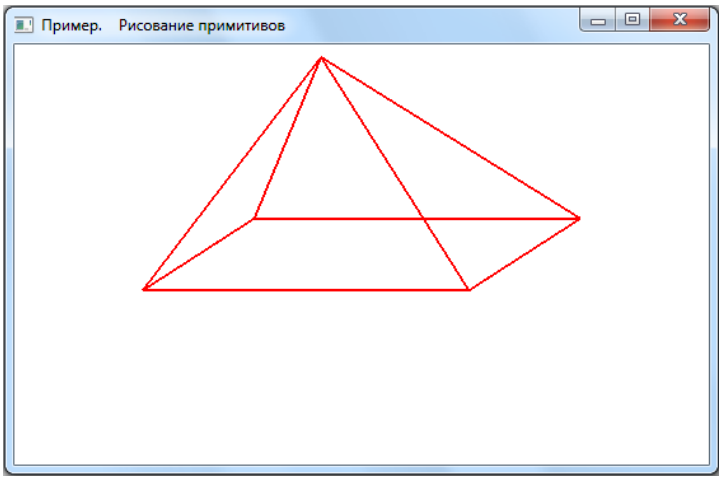


Рис. П.2.1

– при нажатии клавиши с латинской буквой «b» (или «B») в окне отображается пирамида с удалением нелицевых граней (рис. П.2.2);

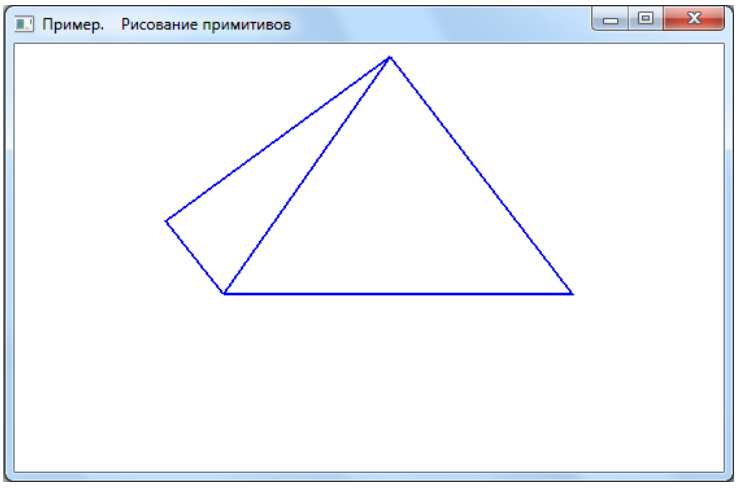


Рис. П.2.2

– при нажатии клавиши с латинской буквой «c» (или «C») в окне отображается параллелепипед с удалением нелицевых граней (рис. П.2.3);

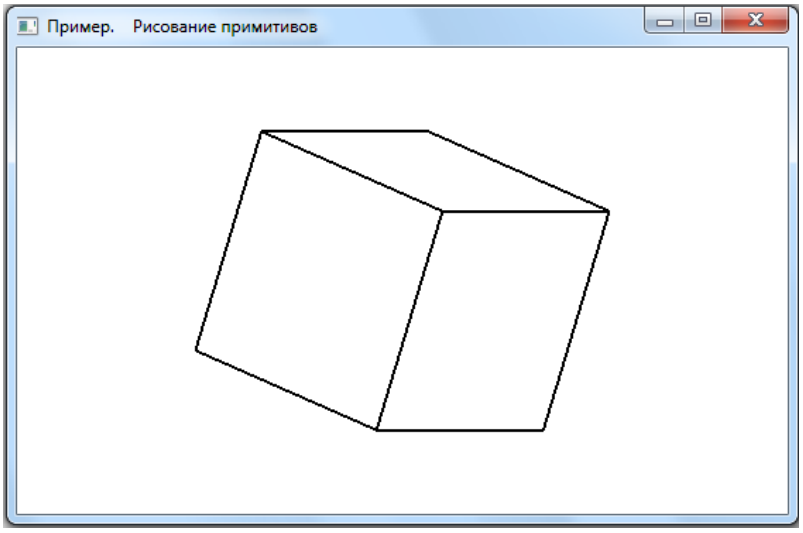


Рис. П.2.3

– при нажатии клавиш со стрелками «←» или «→» изображенная в окне фигура поворачивается вокруг оси *Y*;

– при нажатии клавиш со стрелками «↑» или «↓» изображенная в окне фигура поворачивается вокруг оси *X*.

Список файлов приложения

Файл **Primer.h**

```
#include<stdlib.h>
#include<math.h>
#include<string.h>

// Подключаем библиотеку GLUT

#include<glut.h>//Для Windows 7

//Функция управляет всем выводом на экран
void OnDraw(void);
//Функция вызывается при изменении размеров окна
void OnSise(GLint w, GLint h);
//Функция обрабатывает сообщения от клавиатуры
void Keyboard(unsignedchar key, int x, int y);
//Функция обрабатывает сообщения от клавиш, не имеющих
// ASCII-кодов
```

```
void SpecialKey(int key, int x, int y);
//Функция для рисования параллелепипеда
void Box(GLfloat x1, GLfloat x2, GLfloat y1,
         GLfloat y2, GLfloat z1, GLfloat z2);
//Функция для рисования пирамиды
void Pyramid(GLfloat x1, GLfloat x2, GLfloat y1,
             GLfloat y2, GLfloat z1,GLfloat z2);
```

Файл **Primer.cpp**

```
#include"Primer3.h"
const float pi=3.14159;

//---- Глобальные переменные-----------------------
// x - координата левого верхнего угла окна OpenGL на
// экране
GLint x_win = 100;
// y - координата левого верхнего угла окна OpenGL на
//экране
GLint y_win = 50;
// Начальная ширина окна
GLint Width = 800;
// Начальная высота окна
GLint Height = 600;
// Для определения области видимости графических
//данных после проектирования в СКН
Glint left, right, top, bottom, znear, zfar;
// Для выбора рисунка
GLint Index = 0;
// Угол для поворота вокруг оси X, град
float angleX = 0;
// Угол для поворота вокруг оси X, град
float angleY = 0;
// Шаг изменеия угла, град
float d_angle = 10;

//--- Эта функция управляет всем выводом на экран---
void OnDraw(void)
{
 switch(Index)
 {
      case 0:
      {
      glClearColor(1.0, 1.0, 1.0, 0.0);
      glClear(GL_COLOR_BUFFER_BIT);
      break;
      }
```

```
      case 1:
      {
      //Цвет фона в окне
      glClearColor(1.0, 1.0, 1.0, 0.0); //Белый
      //Очистить буфер цвета - залить цветом фона

glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);

      glColor3ub(255, 0, 0); //Цвет линии - красный
      glLineWidth(2.0) ; // Толщина линии

      glMatrixMode(GL_MODELVIEW); // Видовая матрица
      glLoadIdentity();   // Видовая матрица  M=I
      //Поворот вокруг оси Y, M=Ry(angleY)
      glRotatef(angleY, 0.0, 1.0, 0.0);
      //Поворот вокруг оси X, M=Ry(angleY)*Rx(angleX)
      glRotatef(angleX, 1.0, 0.0, 0.0);
      //Результат V2=Ry*Rx*V1
      glMatrixMode(GL_PROJECTION); // Ортографическая
                                   //проекция
      glLoadIdentity();
      left = -10; right = 10; bottom = -10; top = 10;
     znear = -10; zfar = 10;
      //Устанавливаем область видимости
      glOrtho(left, right, bottom, top, znear, zfar);
      glPolygonMode (GL_FRONT_AND_BACK, GL_LINE);
      GLfloat x1 = -5, x2 = 5,y1 = -5, y2 = 5,
              z1= 0.0,    z2 = 10;
      //Запрещаем удалять нелицевые грани
      glDisable(GL_CULL_FACE);
      Pyramid(x1, x2, y1, y2, z1, z2); //Рисуем
                                       //пирамиду
      break;
                }
      case 2:
      {
      //Цвет фона в окне
      glClearColor(1.0, 1.0, 1.0, 0.0); //Белый
      //Очистить буфер цвета - залить цветом фона
glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT );
      glColor3ub(0, 0, 255);   // Цвет линии - синий
      glLineWidth(2.0) ;       // Толщина линии

      glMatrixMode(GL_MODELVIEW); // Видовая матрица
      glLoadIdentity();   // Видовая матрица  M=I
      //Поворот вокруг оси Y, M=Ry(angleY)
      glRotatef(angleY, 0.0, 1.0, 0.0);
```

```
      //Поворот вокруг оси X, M=Ry(angleY)*Rx(angleX)
      glRotatef(angleX, 1.0, 0.0, 0.0);
      //Результат V2=Ry*Rx*V1
      glMatrixMode(GL_PROJECTION); // Ортографическая
                                    //проекция
      glLoadIdentity();
      left = -10; right = 10; bottom = -10; top = 10;
     znear = -10; zfar = 10;
      //Устанавливаем область видимости
      glOrtho(left, right, bottom, top, znear, zfar);
      //Разрешаем удалять нелицевые грани
      glEnable(GL_CULL_FACE);
      glCullFace(GL_BACK);
      glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);
      GLfloat x1 = -5, x2 = 5,y1 = -5, y2 = 5, z1= 0.0,
              z2 = 10;
      Pyramid(x1, x2, y1, y2, z1, z2); //Рисуем
                                       //пирамиду
      break;
      }
      case 3:
      {
      //Цвет фона в окне
      glClearColor(1.0, 1.0, 1.0, 0.0); //Белый
      //Очистить буфер цвета - залить цветом фона
glClear(GL_COLOR_BUFFER_BIT| GL_DEPTH_BUFFER_BIT );

      glColor3ub(0, 0, 0);// Цвет линии - черный
      glLineWidth(2.0) ;  // Толщина линии
      glMatrixMode(GL_MODELVIEW); // Видовая матрица
      glLoadIdentity();
      //Поворот вокруг оси Y, M=Ry(angleY)
      glRotatef(angleY, 0.0, 1.0, 0.0);
      //Поворот вокруг оси X, M=Ry(angleY)*Rx(angleX)
      glRotatef(angleX, 1.0, 0.0, 0.0);
      // Результат V2=Ry*Rx*V1
      glMatrixMode(GL_PROJECTION); // Ортографическая
                                   //проекция
      glLoadIdentity();
      left = -10; right = 10; bottom = -10; top = 10;
     znear = -10; zfar = 10;
      //Устанавливаем область видимости
      glOrtho(left, right, bottom, top, znear, zfar);
      glEnable(GL_CULL_FACE);
      glCullFace(GL_BACK );
      glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);
      GLfloat x1 = -2.5, x2 = 2.5, y1 = -5, y2 = 5,
```

```
                z1 = -5, z2 = 5;
        Box(x1, x2, y1, y2, z1, z2);
        break;
        }
  }
glFinish();
}
// - Функция вызывается при изменении размеров окна
void OnSize(GLint w, GLint h)
// w - новая ширина окна
// h - новая высота окна
{
// Устанавливаем новые размеры области отображения
glViewport(0, 0, w, h);
}
// Функция обрабатывает сообщения от клавиш,
//имеющих ASCII-код
void Keyboard(unsigned char key, int x, int y)
{
#define ESCAPE '\033'
if( key == ESCAPE ) exit(0);
switch(key)
{
  case'a':case'A': // Пирамида без удаления нелицевых
                   //граней
        {
        Index = 1;
        break;
        }
  case'b':case'B': // Пирамида с удалением нелицевых
                   // граней
        {
        Index = 2;
        break;
        }
  case'c':case'C':  // Параллелепипед с удалением
                    //нелицевых граней
        {
        Index = 3;
        break;
        }
}
glutPostRedisplay(); //Требуем перерисовки окна
}
//-----Функция обрабатывает сообщения от клавиш,
//  не имеющих ASCII-кода  ----------------------
void SpecialKey(int key, int x, int y)
```

```
{
 switch(key)
 {
      case GLUT_KEY_RIGHT: // Стрелка вправо (102)
      {
      angleY+ = d_angle;
      break;
      }
      case GLUT_KEY_LEFT:  // Стрелка влево (100)
      {
      angleY- = d_angle;
      break;
      }
      case GLUT_KEY_UP:   // Стрелка вверх (101)
      {
      angleX- = d_angle;
      break;
      }
      case GLUT_KEY_DOWN:  // Стрелка вниз (103)
      {
      angleX+ = d_angle;
      break;
      }
 }
glutPostRedisplay(); //Требуем перерисовки окна
}
//------- Пирамида ----------------------------------
void Pyramid(GLfloat x1, GLfloat x2, GLfloat y1,
             GLfloat y2, GLfloat z1,GLfloat z2)
{
//Основание ABCD: A(x1,y1,z1), B(x2,y1,z1),
//C(x2,y2,z1), D(x1,y2,z1), Вершина E(0,0,z2)

glBegin(GL_POLYGON);                //Основание
          glVertex3f(x2, y1, z1);  //D
          glVertex3f(x1, y1, z1);  //A
          glVertex3f(x1, y2, z1);  //B
          glVertex3f(x2, y2, z1);  //C
glEnd();

glBegin(GL_POLYGON);           //EDC
      glVertex3f(0, 0, z2);    //E
      glVertex3f(x2, y1, z1); //D
      glVertex3f(x2, y2, z1); //C
glEnd();

glBegin(GL_POLYGON);            //ECB
      glVertex3f(0, 0, z2);     //E
      glVertex3f(x2, y2, z1); //C
```

```
      glVertex3f(x1, y2, z1); //B
glEnd();

glBegin(GL_POLYGON);          //EBA
      glVertex3f(0, 0, z2);   //E
      glVertex3f(x1, y2, z1);  //B
      glVertex3f(x1, y1, z1);  //A
glEnd();

glBegin(GL_POLYGON);           //EAD
      glVertex3f( 0, 0, z2);    //E
      glVertex3f(x1, y1, z1);  //A
      glVertex3f(x2, y1, z1);  //D
glEnd ();
}
//-------------------------------------------------
void Box (GLfloat x1, GLfloat x2, GLfloat y1,
          GLfloat y2, GLfloat z1, GLfloat z2)
{
glBegin(GL_POLYGON);      // front face
      glVertex3f(x1, y1, z2);
      glVertex3f(x2, y1, z2);
      glVertex3f(x2, y2, z2);
      glVertex3f(x1, y2, z2);
glEnd ();

glBegin(GL_POLYGON);             // back face
      glVertex3f(x2, y1, z1);
      glVertex3f(x1, y1, z1);
      glVertex3f(x1, y2, z1);
      glVertex3f(x2, y2, z1);
glEnd();

glBegin(GL_POLYGON);              // left face
      glVertex3f(x1, y1, z1);
      glVertex3f(x1, y1, z2);
      glVertex3f(x1, y2, z2);
      glVertex3f(x1, y2, z1);
glEnd();

glBegin(GL_POLYGON);        // right face
      glVertex3f(x2, y1, z2);
      glVertex3f(x2, y1, z1);
      glVertex3f(x2, y2, z1);
      glVertex3f(x2, y2, z2);
glEnd();

glBegin(GL_POLYGON);           // top face
      glVertex3f(x1, y2, z2);
      glVertex3f(x2, y2, z2);
```

```
      glVertex3f(x2, y2, z1);
      glVertex3f(x1, y2, z1);
glEnd();

glBegin(GL_POLYGON);       // bottom face
      glVertex3f(x2, y1, z2);
      glVertex3f(x1, y1, z2);
      glVertex3f(x1, y1, z1);
      glVertex3f(x2, y1, z1);
glEnd();
}
//============== main ============================

int main(int argc, char *argv[])
{
//---- Функции библиотеки GLUT --------------------
glutInit(&argc, argv); // Инициализация библиотеки GLUT
//Используется режим RGB для выбора цвета, одинарный
//видеобуфер, допускается использование теста глубины
glutInitDisplayMode(GLUT_RGB|GLUT_SINGLE| GLUT_DEPTH);
// Координаты левого верхнего угла окна OpenGL
//на экране монитора
 glutInitWindowPosition(x_win, y_win);
//Ширина и высота окна OpenGL
glutInitWindowSize(Width, Height);
//Создание окна с заголовком
 glutCreateWindow("Пример.Рисование примитивов");
//Регистрация функции обратного вызова OnDraw -
// вызывается системой при необходимости перерисовать
//окно(аналог OnPaint в MFC)
glutDisplayFunc(OnDraw);
//Регистрация функции обратного вызова OnSize –
//вызывается системой при изменении размеров окна
//(аналог //OnSize в MFC)
glutReshapeFunc(OnSize);
//Регистрация функции обратного вызова Keyboard –
//вызывается системой при принажати клавиши, имеющей
 ASCII-код
glutKeyboardFunc(Keyboard);
//Регистрация функции обратного вызова SpesialKey –
//вызывается системой при принажати клавиши,
//не имеющей ASCII-кода
 glutSpecialFunc(SpecialKey);
//Запуск бесконечного цикла обработки сообщений Windows
glutMainLoop();
 return 0;
}
```

ПРИЛОЖЕНИЕ 3

Ниже приведен пример приложения Windows, созданного с использованием библиотеки OpenGL. При запуске программы на экране появляется пустое окно. По щелчку правой клавишей мыши в области окна появляется меню (рис. П.3.1–П.3.3), где можно выбрать объект для отображения (треугольник, прямоугольник) или очистить окно.

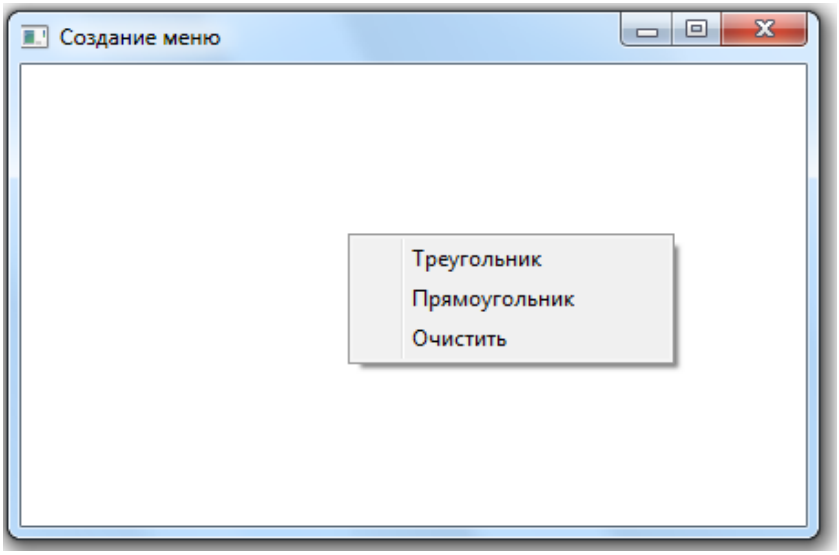


Рис. П.3.1

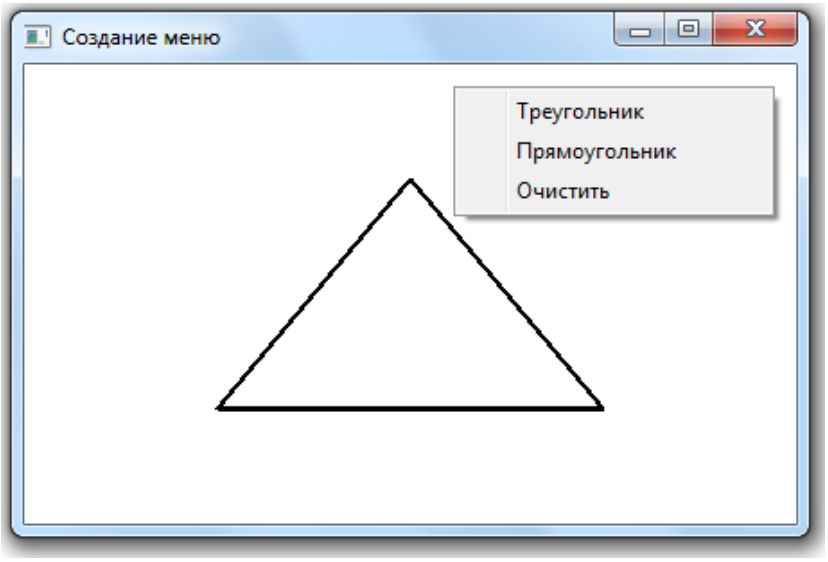


Рис. П.3.2

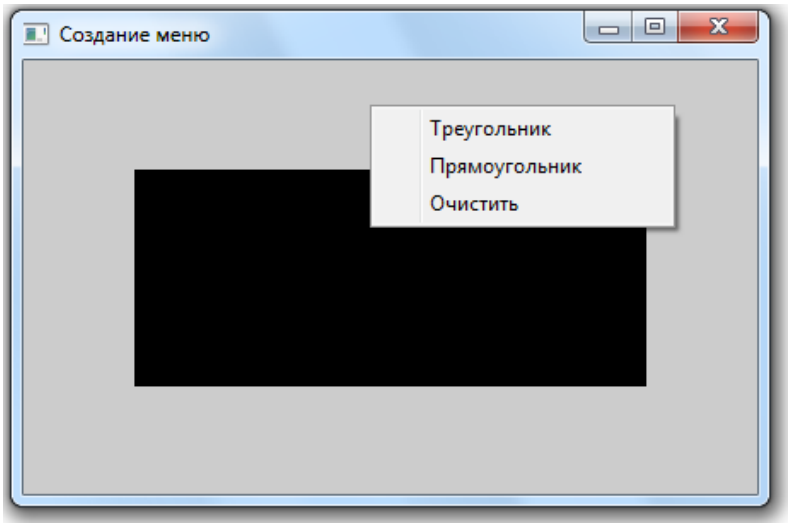


Рис. П.3.3

Список файлов приложения

Файл **Menu.h**

```
#include<stdlib.h>
#include<math.h>
#include<string.h>
// Подключаем библиотеку GLUT
#include<glut.h>// Для Windows 7
//Функция управляет всем выводом на экран
void OnDraw(void);
//Функция вызывается при изменении размеров окна
void OnSise(GLint w, GLint h);
void Figure(GLintTypeFigure);
void Triangle();    // Треугольник
void Rectangle();   // Прямоугольник
```

Файл **Menu.cpp**

```
#include"Menu.h"
//x - координата левого верхнего угла окна OpenGL на экране
GLint x_win = 100;
//y - координата левого верхнего угла окна OpenGL на экране
GLint y_win = 50;
// Начальная ширина окна OpenGL на экране
GLint Width = 800;
//Начальная высота окна OpenGL на экране
GLint Height = 600;
// Переменные для определения области видимости
// графических данных после проектирования в системе
// координат наблюдателя (СКН)
```

```
Glint left, right, top, bottom;
Glint Index = 0; // Для выбора рисунка
float angle=0;   // Угол для поворота треугольника, град

//--- Эта функция управляет всем выводом на экран------
void OnDraw(void)
{
     switch(Index)
     {
          case 0:
          {
          glClearColor(1.0, 1.0, 1.0, 0.0);
          glClear(GL_COLOR_BUFFER_BIT);
          break;
          }
          case 1:
          {
          //Цвет фона в окне
          glClearColor(1.0, 1.0, 1.0, 0.0); // - белый
// Очистить буфер цвета - залить цветом фона
          glClear(GL_COLOR_BUFFER_BIT);
          glColor3ub(0, 0, 0);     // Цвет линии
          glLineWidth(3.0) ;  // Толщина линии
          glMatrixMode(GL_MODELVIEW); // Видовая матрица
          glLoadIdentity();
// Ортографическая проекция
          glMatrixMode(GL_PROJECTION);
          glLoadIdentity();
          left = -10; right = 0; bottom = -10; top = 10;
          gluOrtho2D(left, right, bottom, top);
          Triangle(); //Рисуем треугольник
          break;
          }
          case 2:
          {
//Цвет фона в окне
          glClearColor(0.8, 0.8, 0.8, 0.0); // - серый
// Очистить буфер цвета - залить цветом фона
          glClear(GL_COLOR_BUFFER_BIT);
          glColor3ub(0, 0, 0);     // Цветфигуры
          glMatrixMode(GL_MODELVIEW); // Видовая матрица
          glLoadIdentity();
//Ортографическая проекция
          glMatrixMode(GL_PROJECTION);
          glLoadIdentity();
          left = -10; right = 10; bottom = -10; top = 10;
          gluOrtho2D(left, right, bottom, top);
```

```
          Rectangle();
          break;
          }
     }
glFinish();
}

// --Функция вызывается при изменении размеров окна--
void OnSize(GLint w, GLint h)
// w - новая ширина окна
// h - новая высота окна
{
// Устанавливаем новые размеры области отображения
glViewport(0,0, w, h);
}

//--------------- Figure -----------------------------
void Figure(GLintTypeFigure)
{
     index = TypeFigure;
     glutPostRedisplay();
}

//--------- Треугольник -----------------------------
void Triangle()
{
glBegin(GL_LINE_LOOP);
glVertex2f(-5.0, -5.0);
glVertex2f(0.0, 5.0);
glVertex2f(5.0, -5.0);
glEnd();
}

//--------- Прямоугольник ----------------------------
void Rectangle()
{
GLfloatVectCoords[4][2]={-7.0, -5.0, -7.0, 5.0, 7.0,
                         5.0, 7.0, -5.0};
glBegin(GL_POLYGON);
for(int i = 0; i<4; i++) glVertex2fv(VectCoords[i]);
glEnd();
}

//================ main ================================
int main(intargc, char *argv[])
{
//----- Функции библиотеки GLUT ---------------------
glutInit(&argc, argv); // Инициализация библиотеки GLUT
//Используется режим RGB для выбора цвета и одинарный
//видеобуфер
```

```
glutInitDisplayMode(GLUT_RGB|GLUT_SINGLE);
// Координаты левого верхнего угла окна OpenGL на экране
// монитора
glutInitWindowPosition (x_win, y_win);
//Ширина и высота окна OpenGL на экране монитора
glutInitWindowSize(Width, Height);
//Создание окна с заголовком
glutCreateWindow("Создание меню");
//Регистрация функции обратного вызова OnDraw - вызывается
//системой при необходимости перерисовать окно
glutDisplayFunc(OnDraw);
//Регистрация функции обратного вызова OnSize - вызывается
//системой при изменении размеров окна
glutReshapeFunc(OnSize);
//-------Создаем меню -------------------------
glutCreateMenu(Figure);
     glutAddMenuEntry("Треугольник", 1);
     glutAddMenuEntry("Прямоугольник", 2);
     glutAddMenuEntry("Очистить", 0);
     glutAttachMenu(GLUT_RIGHT_BUTTON);
//-----------------------------------------------
     glutMainLoop(); //Запуск бесконечного цикла обработки
 //сообщений Windows
     return 0;
}
```

ЛИТЕРАТУРА

1. Никулин, Е. А. Компьютерная геометрия и алгоритмы машинной графики / Е. А. Никулин. – СПб.: БХВ-Петербург, 2003.

2. Порев, В. Н. Компьютерная графика / В. Н. Порев. – СПб.: БХВ-Петербург, 2002.

3. Цисарж, В. В. Математические методы компьютерной графики / В. В. Цисарж. – Киев: Факт, 2004.

4. Поляков, А. Ю. Методы и алгоритмы компьютерной графики в примерах на VisualC++ / А. Ю. Поляков, В. А. Брусенцев. – СПб.: БХВ-Петербург, 2003.

5. Шикин, Е. В. Кривые и поверхности на экране компьютера. Руководство по сплайнам для пользователей / Е. В. Шикин, Л. И. Плис. – М.: ДИАЛОГ-МИФИ, 1996.

6. Щупак, Ю. А. Win32 API. Эффективная разработка приложений / Ю. А. Щупак. – СПб.: Питер, 2007.

7. Вержбицкий, В. М. Численные методы (математический анализ и обыкновенные дифференциальные уравнения): учеб. пособие для вузов / В. М. Вержбицкий. – М.: Высшая школа, 2001.

8. Графическая библиотека OpenGL: учеб.-метод. пособие / Ю. М. Баяковский [и др.]. – М.: Издательский отдел МГУ, 2003.

9. Тихомиров, Ю. В. Программирование трехмерной графики / Ю. В. Тихомиров. – СПб.: БХВ-Петербург, 2001.

10. Хилл, Ф. OpenGL. Программирование компьютерной графики. Для профессионалов / Ф. Хилл. – СПб.: Питер, 2002.

11. Райт, Р. OpenGL. Суперкнига / Р. Райт, Б. Липчак. – 3-е изд. – М.: Издательский дом «Вильямс», 2006.

ОГЛАВЛЕНИЕ

Учебное издание

Дятко Александр Аркадьевич
Мороз Леонарда Станиславовна

ОСНОВЫ КОМПЬЮТЕРНОЙ ГЕОМЕТРИИ И ГРАФИКИ

Учебно-методическое пособие

Редактор *О. П. Приходько*
Компьютерная верстка *О. П. Приходько*
Корректор *О. П. Приходько*

Подписано в печать 11.06.2013. Формат $60\times84^{1}/_{16}$.
Бумага офсетная. Гарнитура Таймс. Печать офсетная.
Усл. печ. л. 12,9. Уч.-изд. л. 13,3.
Тираж 100 экз. Заказ .

Издатель и полиграфическое исполнение:
УО «Белорусский государственный технологический университет».
ЛИ № 02330/0549423 от 08.04.2009.
ЛП № 02330/0150477 от 16.01.2009.
Ул. Свердлова, 13а, 220006, г. Минск.