Е. В. Шикин, А. В. Боресков

КОМПЬЮТЕРНАЯ ГРАФИКА

Полигональные моде



МОСКВА ■ "ДИАЛОГ-МИФИ" ■ 2001

УДК 681.3
Ш57

Шикин А. В., Боресков А. В.
Ш57 Компьютерная графика. Полигональные модели. - М.: ДИАЛОГ-МИФИ, 2001. - 464 с.

ISBN 5-86404-139-4

Книга знакомит с такими основными понятиями и методами компьютерной графики, как трехмерная математика, растровые алгоритмы, непосредственная работа с графическими устройствами, вычислительная геометрия, удаление невидимых линий и поверхностей, текстурирование, построение графического интерфейса, OpenGL. Она дает представление об основных направлениях компьютерной графики и позволяет освоить базовые приемы реализации ее алгоритмов на персональных компьютерах. Приведенные в книге программы могут быть использованы для широкого класса задач. Книгу можно рассматривать как практическое руководство, так как она содержит ряд упражнений, которые способен выполнить прочитавший книгу.

Учебно-справочное издание
Шикин Евгений Викторович
Боресков Алексей Викторович
Компьютерная графика. Полигональные модели

Редактор О. А. Голубев
Корректор В. С. Кустов
Макет Н. В. Дмитриевой

Лицензия ЛР N 071568 от 25.12.97. Подписано в печать 11.03.2001.
Формат 60x84/16. Бум. офс. Печать офс. Гарнитура Таймс.
Усл. печ. л. 26.97. Уч.-изд. л. 16.9. Тираж 4 000 экз. Заказ 440

ЗАО "ДИАЛОГ-МИФИ"
115409, Москва, ул. Москворечье, 31, корп. 2

Подольская типография
142100, г. Подольск, Московская обл., ул. Кирова, 25

ISBN 5-86404-139-4

© Шикин А. В., Боресков А. В., 2001

© Оригинал-макет, оформление обложки.
АО "ДИАЛОГ-МИФИ", 2001

Предисловие

Это третья книжка по компьютерной графике, которую выпускают авторы в издательстве "ДИАЛОГ-МИФИ". Две книги, выпущенные ранее (в 1993-м и в 1995 г.), давно разошлись. Предлагаемая книжка имеет с ними много общего, но отнюдь не поглощает их.

Развитие компьютерной графики идет бурно и неравномерно - что-то удивительно быстро устаревает, что-то обретает более отчетливые формы, появляется и очень много нового. Постоянно расширяющиеся возможности доступных вычислительных средств корректируют набор используемых методов и эффективно применяемых алгоритмов.

Работая над рукописью, авторы старались отбирать материал, полезный заинтересованному читателю, привлекая для этого публикации и в научных журналах, и в трудах конференций.

В основу книжки положен базовый вводный курс по компьютерной графике и сопровождающие его специальные курсы, читаемые авторами последние несколько лет на факультете вычислительной математики и кибернетики Московского университета им. М. В. Ломоносова. За это время, общаясь со студентами, авторы накопили весьма разнообразные впечатления, главными из которых следует признать их неспадающий интерес к предмету и постоянно повышающийся уровень представляемых студентами графических работ, неизменно сопровождающих и обязательный курс по компьютерной графике, и развивающие его спецкурсы.

Это обстоятельство, да еще благожелательное отношение со стороны руководства издательства, и подталкивали авторов к написанию - занятию скорее альтруистическому (здесь стоит отметить, однако, что работа над рукописью шла при частичной поддержке Российского фонда фундаментальных исследований, гранты 98-01-00550 и 98-01-00550).

Было бы странно выпускать сейчас книжку по компьютерной графике без визуальных материалов. Поэтому к бумажному носителю авторы прилагают постоянно расширяющийся site

http://graphics.cs.msu.su/courses/cg2000s

где специально выделено место для возникающих вопросов и последующих ответов на них.

Если у вас нет доступа в Интернет, то в издательстве "Диалог-МИФИ" вы можете купить компакт-диск с этим материалом.

А. В. Боресков,
Е. В. Шикин

Октябрь 1999 г.

Глава 1
СВЕТ. ЦВЕТОВОСПРИЯТИЕ. ЦВЕТОВЫЕ МОДЕЛИ

Понятия света и цвета в компьютерной графике являются основополагающими.

Свет можно рассматривать двояко - либо как поток частиц различной энергии (тогда его цвет определяет энергия частиц), либо как поток электромагнитных волн (в этом случае цвет определяется длиной волны). Далее мы будем рассматривать свет как поток электромагнитных волн.

.Электромагнитная волна (рис. 1.1) характеризуется своей амплитудой A, длиной волны λ, фазой и поляризацией. Видимый свет - это волны с длиной λ от 400 до 700 нм. Амплитуда определяет энергию волны, которая пропорциональна квадрату амплитуды. Фазу и поляризацию электромагнитных волн мы будем в дальнейшем игнорировать.

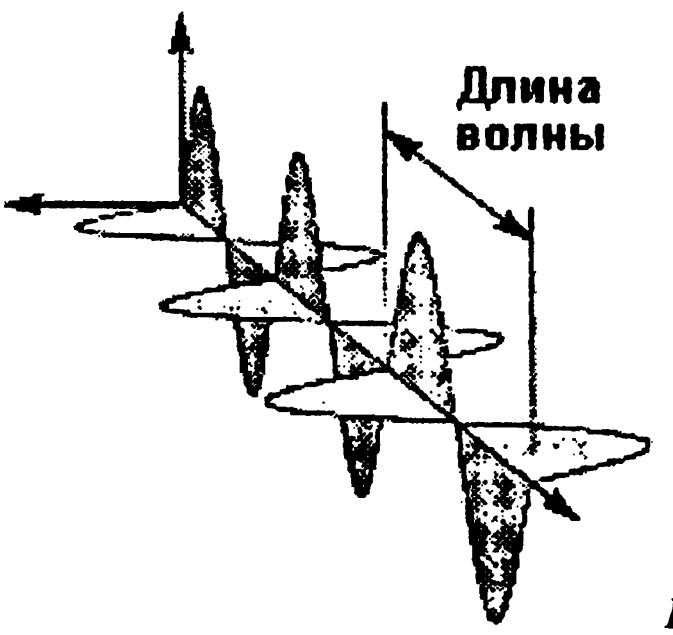


Рис. 1.1

На практике мы редко сталкиваемся со светом какой-то одной определенной длины волны (исключение составляет лишь излучение лазера).

Обычно свет представляет собой непрерывный поток волн с различными длинами волн и различными амплитудами. Такой свет можно характеризовать так называемой энергетической (мощностной) спектральной кривой $I(\lambda)$, где само значение функции $I(\lambda)$ представляет собой мощностной вклад волн с длиной волны λ в общий волновой поток.

При этом общая мощность света равняется интегралу от спектральной функции по всему видимому диапазону длин волн. Типичная спектральная кривая приведена на рис. 1.2.

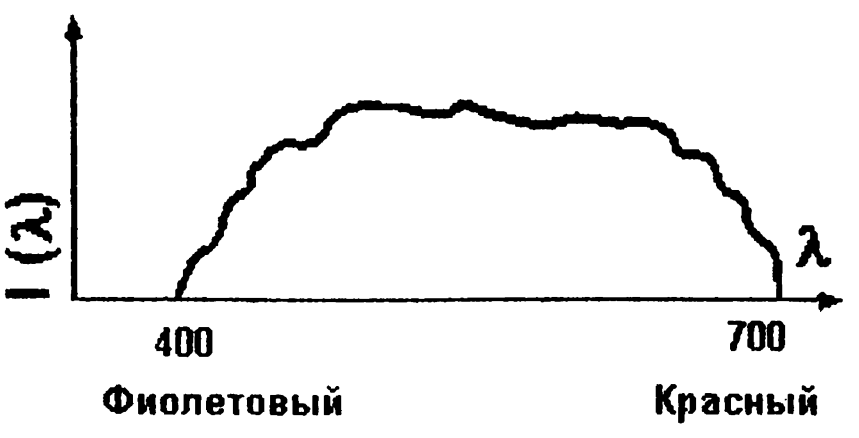


Рис. 1.2

Само понятие цвета тесно связано с тем, как человек (человеческий глаз) воспринимает свет; можно сказать, что цвет зарождается в глазу.

Рассмотрим, каким именно образом происходит восприятие света человеческим глазом. Сетчатка глаза содержит два принципиально различных типа фоторецепторов - палочки, обладающие широкой спектральной кривой чувствительности, вследствие чего они не различают длин волн и, следовательно, цвета, и колбочки, харак-

теризующиеся узкими спектральными кривыми и поэтому обладающие цветовой чувствительностью.

Колбочки бывают трех типов, отвечающих за чувствительность к длинным, средним и коротким волнам. Выдаваемое колбочкой значение является результатом интегрирования спектральной функции $I(\lambda)$ с весовой функцией чувствительности. На рис. 1.3 представлены графики функций чувствительности для всех трех типов колбочек. Видно, что у одной из них пик чувствительности приходится на волны с короткой длиной волны (синий цвет), у другой - на волны средней длины волны (желто-зеленый цвет), а у третьей - на волны с большой длиной волны (красный цвет).

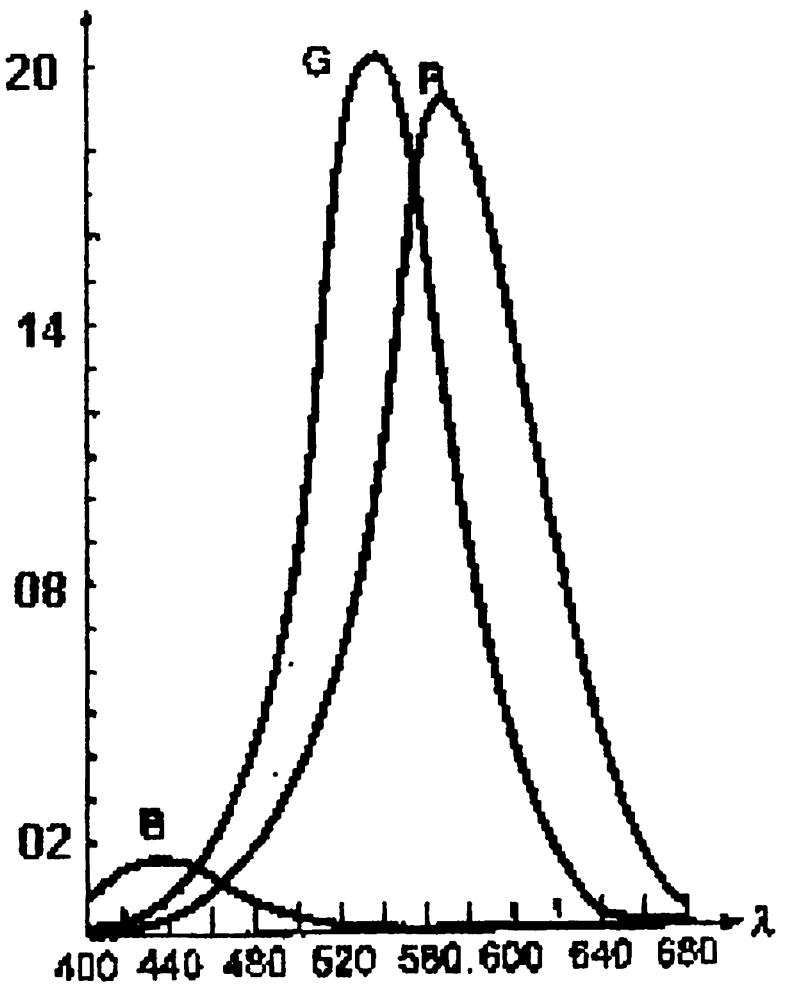


Рис. 1.3

Таким образом, глаз человека ставит в соответствие спектральной функции $I(\lambda)$ тройку чисел (R, G, B), получаемую по формулам

$$\left.\begin{aligned} R &= \int I(\lambda) P_R(\lambda) d\lambda, \\ G &= \int I(\lambda) P_G(\lambda) d\lambda, \\ B &= \int I(\lambda) P_B(\lambda) d\lambda, \end{aligned}\right\} \quad (1.1)$$

где $P_R(\lambda), P_G(\lambda)$ и $P_B(\lambda)$ - весовые функции чувствительности колбочек различных типов. Видно, что наименьшая чувствительность приходится на синий цвет, а наибольшая - на желто-зеленый.

График кривой, отвечающей за общую чувствительность глаза к свету (рис. 1.4), получается в результате суммирования всех трех кривых с рис. 1.3.

Соотношения (1.1) ставят в соответствие каждой спектральной кривой $I(\lambda)$ тройку чисел (R, G, B). Это соответствие не является взаимно однозначным - одному и тому же набору чисел (R, G, B) соответствует бесконечное множество различных спектральных кривых, называемых метамерами.

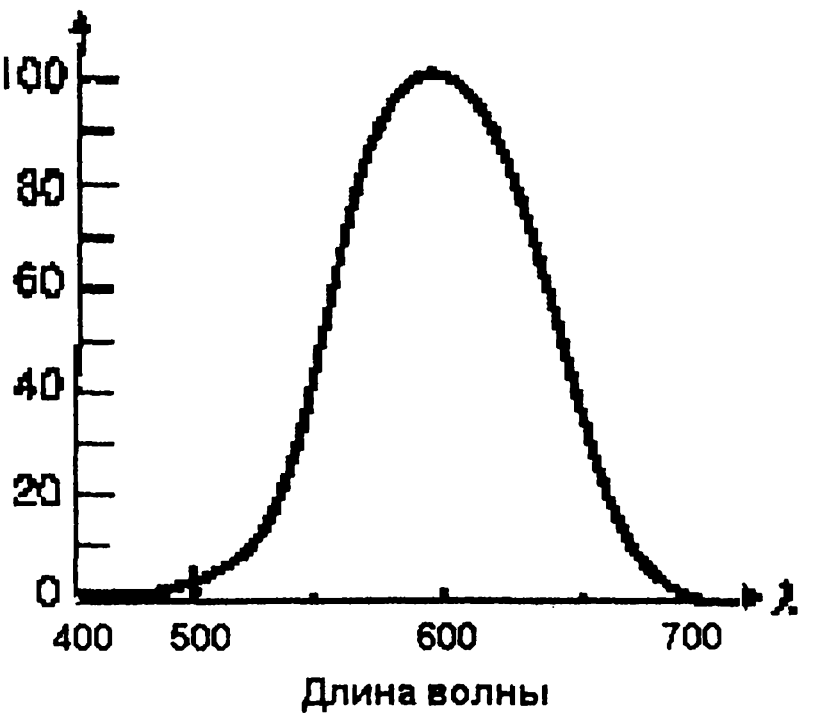


Рис. 1.4

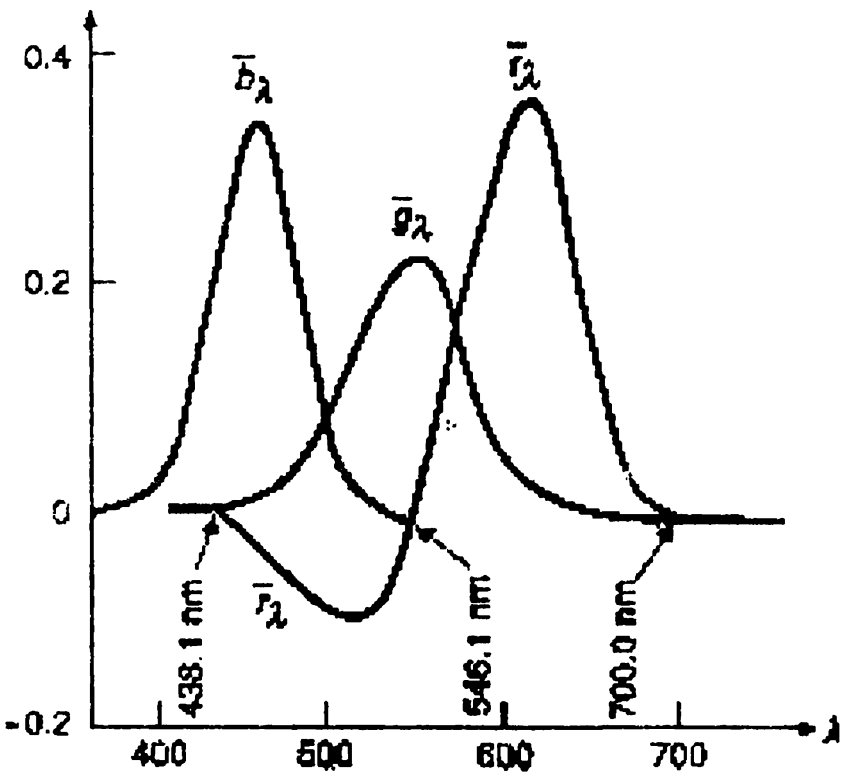


Рис. 1.5

На рис. 1.5 представлены значения коэффициентов (*R, G, B*) для волн различной длины из видимой части спектра. Как видно из приведенных графиков для отдельных длин волн, некоторые из коэффициентов *R*, *G* и *B* могут быть меньше нуля. Это означает, что не все цвета представимы при помощи RGB-модели.

Тем самым цветные мониторы, построенные на основе RGB-модели (изображение строится при помощи трех типов люминофора - красного, зеленого и синего цветов), не могут воспроизвести всех возможных цветов.

Это приводит к необходимости введения другой цветовой модели, которая описывала бы все видимые цвета при помощи неотрицательных коэффициентов.

В 1931 г. Commission Internationale de L'Eclairage (CIE) приняла стандартные кривые для гипотетического идеального наблюдателя. Они приведены на рис. 1.6. С их помощью строится цветовая модель CIE XYZ, где величины *X*, *Y* и *Z* задаются соотношениями

$$X = \int I(\lambda)\bar{x}(\lambda)d\lambda,$$

$$Y = \int I(\lambda)\bar{y}(\lambda)d\lambda,$$

$$Z = \int I(\lambda)\bar{z}(\lambda)d\lambda.$$

Этими тремя числами X, Y и Z любой цвет, воспринимаемый глазом, можно охарактеризовать однозначно

Несложно заметить, что кривая, отвечающая за вторую координатуY, совпадает с кривой чувствительности глаза к свету.

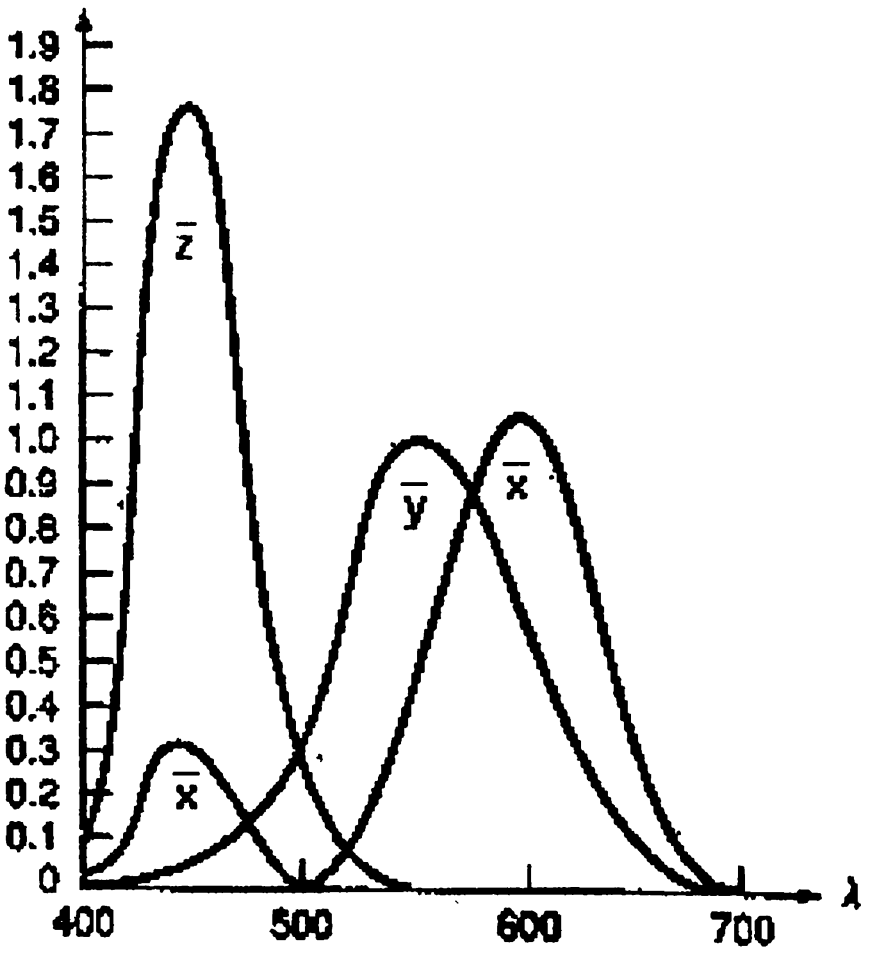


Рис. 1.6

Интенсивность - это мера потока мощности, который излучается или падает на поверхность. Она линейно зависит от спектральной кривой и выражается в ваттах на квадратный метр.

Величина Y, выражающая интенсивность с учетом спектральной чувствительности глаза, называется люминантностью (CIE luminance).

В ряде случаев возникает необходимость отделить информацию о люминантности от информации о самом цвете. С этой целью вводятся так называемые хроматические координаты *x* и *y*

$$x = \frac{X}{X + Y + Z},$$

$$y = \frac{Y}{X + Y + Z}$$

Любой цвет, воспринимаемый глазом, можно охарактеризовать тройкой чисел (x, y, Y); по ней тройку X, Y и Z можно восстановить однозначно.

При изменении длины волны λ вдоль видимого диапазона точка (x, y) описывает кривую на плоскости переменных x и y. Если концы этой кривой соединить отрезком (рис. 1.7), то внутри получившейся области будут находиться все видимые цвета. При этом сам построенный отрезок будет соответствовать сиреневым цветам, которые спектральными не являются (им не соответствует никакая длина волны: сиреневые цвета являются взвешенной смесью красного и синего цветов).



Рис. 1.7

Еще одним неспектральным цветом является белый цвет, который представляет собой смесь всех цветов. CIE определяет белый цвет, при помощи спектральной кривой D_{65}, которая вводит его как приближение обычного дневного света. Координаты белого цвета в системе CIE XYZ обозначают через (X_n, Y_n, Z_n).

Рассмотрим на хроматической диаграмме две точки, которым соответствуют цвета C_1 и C_2. Цветам, получаемым в результате их смешивания, на хроматической диаграмме соответствует отрезок, соединяющий эти точки. Если взять на хроматической диаграмме три точки, то в результате их смешения можно получить все цвета из треугольника, вершинами которого являются эти точки. Вместе с тем при взгляде на рис. 1.7 нетрудно заметить, что какие бы три цвета мы ни взяли, порождаемый ими треугольник не покроет всей области. Тем самым никакие три цвета всех види-

мых цветов не могут дать. Наибольшее же множество представимых цветов порождают синий, зеленый и красный цвета.

Восприятие глазом люминантности Y носит нелинейный характер. Источник света, имеющий интенсивность всего 18 % от исходного, кажется лишь наполовину менее ярким. Более того, система CIE XYZ не является линейно воспринимаемой, т. е. разность двух цветов $\Delta C = C_2 - C_1$ для разных значений цветов C_1 и C_2 воспринимается глазом по-разному.

С целью получения равномерно воспринимаемого цветового пространства были введены системы $CIE\ L^*u^*v^*$ и $CIE\ L^*a^*b^*$

$$L^* = 116\left(\frac{Y}{Y_n}\right)^{\frac{1}{3}} - 16, \frac{Y}{Y_n} > 0.008856,$$

$$L^* = 903.3\frac{Y}{Y_n}, \frac{Y}{Y_n} \le 0.008856,$$

$$u'_n = \frac{4X_n}{X_n + 15Y_n + 3Z_n},$$

$$v'_n = \frac{9Y_n}{X_n + 15Y_n + 3Z_n},$$

$$u' = \frac{4X}{X + 15Y + 3Z},$$

$$v' = \frac{9Y}{X + 15Y + 3Z},$$

$$u^* = 13L^*(u' - u_n),$$

$$v^* = 13L^*(v' - v_n)$$

$$a^* = 500\left[\left(\frac{X}{X_n}\right)^{1/3} - \left(\frac{Y}{Y_n}\right)^{1/3}\right],$$

$$b^* = 200\left[\left(\frac{Y}{Y_n}\right)^{1/3} - \left(\frac{Z}{Z_n}\right)^{1/3}\right]$$

Величина L^* изменяется в пределах от 0 до 100, при этом изменение интенсивности $\Delta L^* = 1$ считается пределом чувствительности глаза.

Введем еще несколько понятий, определяемых CIE.

Тон (hue) - атрибут визуального восприятия, согласно которому область кажется обладающей одним из воспринимаемых цветов (красного, желтого, зеленого и синего) или комбинацией любых двух из них.

Насыщенность (saturation) - это пропорция чистого (красного, синего, зеленого и т. д.) и белого цветов, необходимая для того, чтобы определить цвет. Насыщенность показывает, насколько чистым является цвет (насколько в нем мало белого цвета). Красный цвет имеет насыщенность, равную 100 %, а серые цвета - насыщенность, равную нулю.

Интенсивность света, генерируемого физическим устройством (например, монитором), обычно зависит от приложенного сигнала. Так, для обычного монитора зависимость интенсивности от входного сигнала (напряжения) нелинейна:

$$Intensity \sim Voltage^{2.5}$$

Показатель степени обычно обозначают буквой γ. В связи с этим изображение для вывода на экран должно быть подвергнуто так называемой γ-коррекции. В соответствии с рекомендацией 709, которой соответствует большинство мониторов и видеоустройств, видеокамера проводит преобразование линейного RGB-сигнала следующим образом:

$$R' = \begin{cases} 4.5R, R \le 0.018, \\ -0.099 + 1.099R^{0.45} \end{cases} \qquad (1.2)$$

Для компонент G и B аналогично. Идеальный монитор инвертирует отображение.

$$R = \begin{cases} \dfrac{R'}{4.5}, R' \le 0.018, \\ \left(\dfrac{R' + 0.99}{1.099}\right)^{\frac{1}{0.45}} \end{cases} \qquad (1.3)$$

На основе R', G' и B' часто вводится величина

$$Y' = 0.299R' + 0.5876G' + 0.114B',$$

называемая люмой (luma).

Существуют и другие цветовые системы; некоторые из них мы рассмотрим ниже.

Наиболее простой является система RGB, применяемая в целом ряде видеоустройств. Это аддитивная цветовая модель: для получения искомого цвета базовые цвета в ней складываются. Цветовым пространством является единичный куб. Главная диагональ куба, характеризуемая равным вкладом трех базовых цветов, представляет серые цвета: от черного (0, 0, 0) до белого - (1, 1, 1) (рис. 1.8).

Рекомендация 709 определяет хроматические координаты люминофора монитора и белого цвета D_{65}:

	Red	*Green*	*Blue*	*White*
x	0.640	0.300	0.150	0.3127
y	0.330	0.600	0.060	0.3290

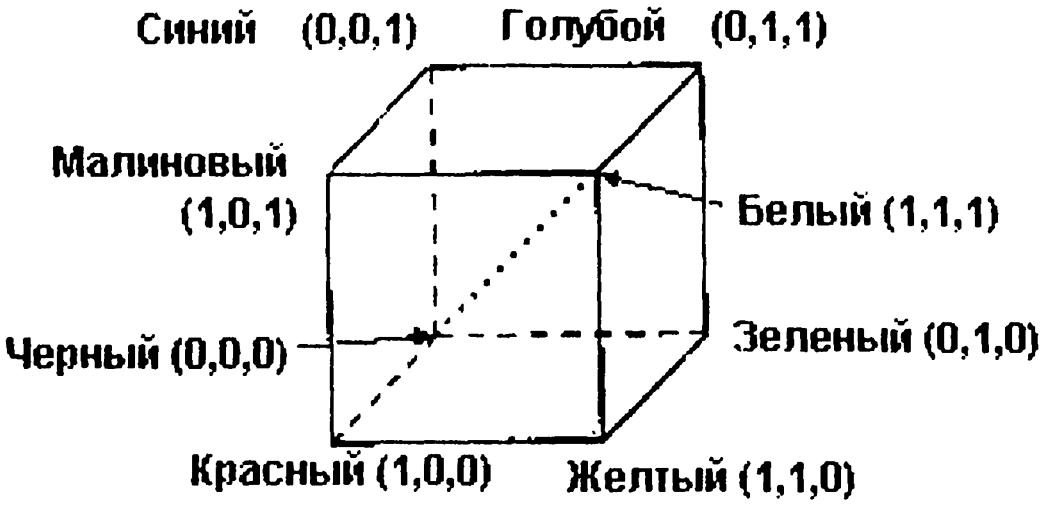


Рис 1.8

Исходя из этого, можно записать формулы для перехода от системы CIE XYZ к системе RGB:

$$\begin{pmatrix} R \\ G \\ B \end{pmatrix} = \begin{pmatrix} 3.240479 & -1.537156 & -0.498535 \\ -0.969256 & 1.875992 & 0.041556 \\ 0.055648 & -0.204043 & 1.057311 \end{pmatrix} \begin{pmatrix} X \\ Y \\ Z \end{pmatrix}$$

Если какой-либо цвет не может быть представлен в RGB-модели, то у него хотя бы одна из компонент будет либо отрицательной, либо большей единицы.

Приведем обратное преобразование из RGB в CIE XYZ:

$$\begin{pmatrix} X \\ Y \\ Z \end{pmatrix} = \begin{pmatrix} 0.412453 & 0.357580 & 0.180423 \\ 0.212671 & 0.715160 & 0.072169 \\ 0.019334 & 0.119193 & 0.950227 \end{pmatrix} \begin{pmatrix} R \\ G \\ B \end{pmatrix}.$$

В цветной печати чаще используются модели CMY (Cyan, Magenta, Yellow) и CMYK (Cyan, Magenta, Yellow, blacK). Эти модели в отличие от RGB являются субтрактивными (точнее сказать, мультипликативными) - для того чтобы получить требуемый цвет, базовые цвета вычитаются из белого цвета.

Рассмотрим, как это происходит. Когда на поверхность бумаги наносится голубой (cyan) цвет, то красный цвет, падающий на бумагу, полностью поглощается. Таким образом, голубой краситель как бы вычитает красный цвет из падающего белого (являющегося суммой красного, зеленого и синего цветов). Аналогично малиновый краситель (magenta) поглощает зеленый, а желтый краситель - синий цвет. Поверхность, покрытая голубым и желтым красителями, поглощает красный и синий, оставляя только зеленую компоненту. Голубой, желтый и малиновый красители поглощают красный, зеленый и синий цвета, оставляя в результате черный. Эти соотношения можно представить в виде следующей формулы:

$$\begin{pmatrix} C \\ M \\ Y \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix} - \begin{pmatrix} R \\ G \\ B \end{pmatrix}. \qquad (1.4)$$

Обратное преобразование осуществляется по формуле

$$\begin{pmatrix} R \\ G \\ B \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix} - \begin{pmatrix} C \\ M \\ Y \end{pmatrix}.$$

По целому ряду причин (большой расход дорогостоящих цветных чернил, высокая влажность бумаги, получаемая при печати на струйных принтерах, нежелательные визуальные эффекты, возникающие за счет того, что при выводе точки трех базовых цветов ложатся с небольшими отклонениями) использование трех красителей для получения черного цвета оказывается неудобным. Поэтому его просто добавляют к трем базовым цветам. Так получается модель CMYK (Cyan, Magenta, Yellow, blacK).

Для перехода от модели CMY к модели CMYK используют следующие соотношения:

$$\begin{aligned} K &= \min(C, M, Y), \\ C &= C - K, \\ M &= M - K, \\ Y &= Y - K. \end{aligned} \qquad (1.5)$$

Замечание. Соотношения (1.4) и (1.5) верны лишь в том случае, когда спектральные кривые отражения для базовых цветов не пересекаются. Однако на самом деле между соответствующими спектральными кривыми пересечение существует, поэтому для точной передачи цветов и оттенков изображения эти соотношения мало применимы.

В телевидении часто используется модель YIQ. Перевод из системы RGB в YIQ осуществляется по следующим формулам:

$$\begin{pmatrix} Y \\ I \\ Q \end{pmatrix} = \begin{pmatrix} 0.30 & 0.59 & 0.11 \\ 0.60 & -0.28 & -0.32 \\ 0.21 & -0.52 & 0.31 \end{pmatrix} \begin{pmatrix} R \\ G \\ B \end{pmatrix}$$

Модели RGB, CMY и CMYK ориентированы на работу с цветопередающей аппаратурой и для задания цвета человеком неудобны. С другой стороны, модель HSV (Hue, Saturation, Value), иногда называемая HSB (Hue, Saturation, Brightness), больше ориентирована на работу с человеком и позволяет задавать цвета, опираясь на интуитивные понятия тона, насыщенности и яркости. В этой модели используется цилиндрическая система координат, а множество всех допустимых цветов представляет собой шестигранный конус, поставленный на вершину (рис. 1.9).

Основание конуса представляет яркие цвета и соответствует $V = 1$ Однако цвета основания $V = 1$ не имеют одинаковой воспринимаемой интенсивности (люминантности). Тон (H) измеряется углом, отсчитываемым вокруг вертикальной оси OV При этом красному цвету соответствует угол 0°, зеленому - угол 120° и т. д. Цвета, взаимно дополняющие друг друга до белого, находятся напротив один другого, т. е. их тона отличаются на 180° Величина S изменяется от 0 на оси OV до 1 на гранях конуса.

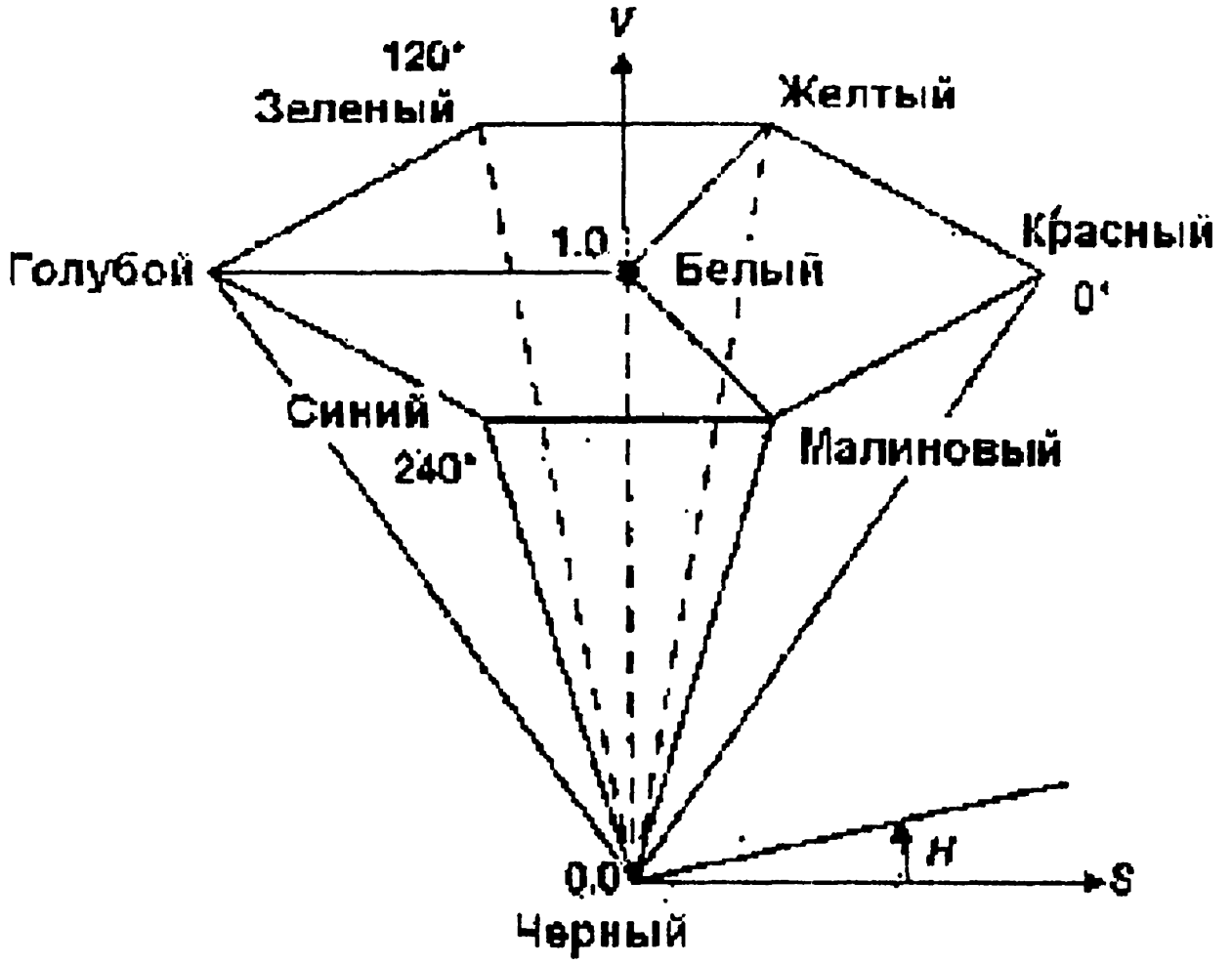


Рис. 1.9

Конус имеет единичную высоту ($V = 1$) и основание, расположенное в начале координат. В основании конуса величины H и S смысла не имеют. Белому цвету соответствует пара $S = 1$, $V = 1$. Ось OV ($S = 0$) - серым тонам. При $S = 0$ значение H не имеет смысла (по соглашению принимает значение HUE_UNDEFINED).

Процесс добавления белого цвета к заданному можно представить как уменьшение насыщенности S, а процесс добавления черного цвета - как уменьшение яркости V.

Основанию шестигранного куба соответствует проекция RGB куба вдоль его главной диагонали.

Ниже приводится программа для преобразования RGB в HSV и наоборот.

```
// File RGBHSV.cpp
void  RGB2HSV (float r, float g, float b,
          float& h, float& s, float& v)
{
        float       cMin = min3( r, g, b );
        float       cMax = max3( r, g, b );
        float       delta = cMax - cMin;

        if ( ( v = cMax ) != 0 )
                    s = delta / cMax;
        else
                    s = 0;

        if ( s == 0 )
                    h = HUE_UNDEFINED;
        else
        {
                    if ( r == v )
                                h = ( g - b ) / delta;
                    else
                    if ( g == v )
```

```
                        h = 2 + ( b - r ) / delta;
                else
                        h = 4 + ( r - g ) / delta;

                if ( ( h *= 60 ) < 0 )
                        h += 360;
        }
}
void  HSV2RGB ( float h, float s, float v,
                float& r, float& g, float& b )
{
        if ( s == 0 )
                if ( h == HUE_UNDEFINED )
                        r = g = b = v;
                else
                        error ();
        else
        {
                if ( h == 360 )
                        h = 0;

                h /= 60;

                int         i = floor ( h );
                float       f = h - i;
                float       p = v * ( 1 - s );
                float       q = v * ( 1 - s * f );
                float       t = v * ( 1 - s * ( 1 - f ) );

                switch ( i )
                {
                        case 0:
                                r = v; g = t; b = p;
                                break;

                        case 1:
                                r = q; g = v; b = p;
                                break;

                        case 2:
                                r = p; g = v; b = t;
                                break;

                        case 3:
                                r = p; g = q; b = v;
                                break;

                        case 4:
                                r = t; g = p; b = v;
                                break;

                        case 5:
                                r = v; g = p; b = q;
                                break;
                }
        }
}
```

Еще одним примером системы, построенной на интуитивных понятиях тона, насыщенности и яркости, является система HLS (Hue, Lightness, Saturation). Здесь также используется цилиндрическая система координат, однако множество всех цветов представляет собой два шестигранных конуса, поставленных друг на друга (основание к основанию, рис. 1.10), причем вершина нижнего конуса совпадает с началом координат. Тон по-прежнему задается углом, отсчитываемым от вертикальной оси с красным цветом (угол 0°).

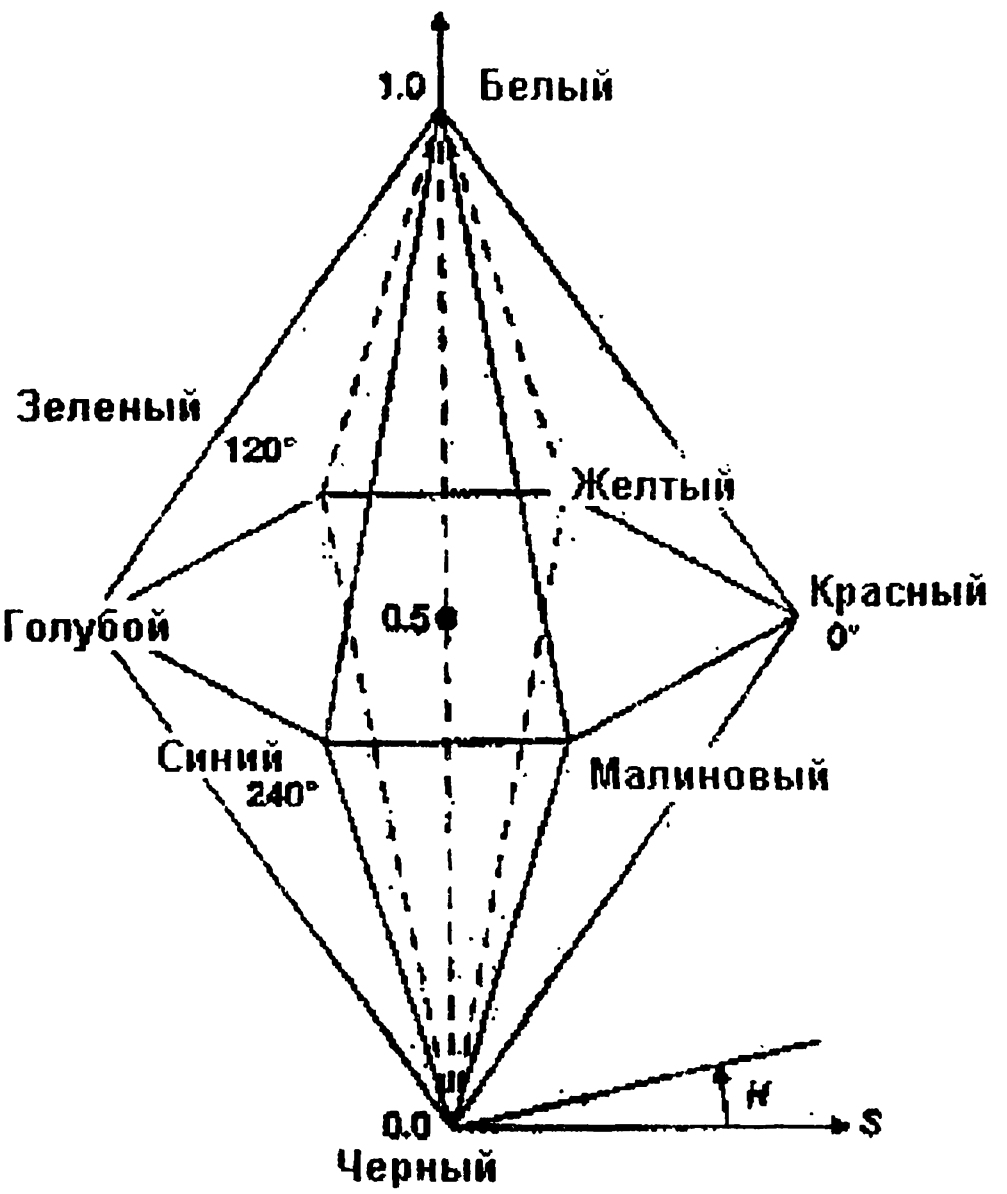


Рис. 1.10

Порядок цветов на периметре общего основания конусов такой же, как и в моде ли HSV Модель HLS можно рассматривать как модификацию модели HSV, где бе лый цвет сдвинут вверх, чтобы сформировать верхний конус из плоскости $V = 1$.

Процедура для перевода цвета из модели HLS в модель RGB приводится ниже.

```
// File HLS2RGB.cpp
void   HLS2RGB (float h, float l, float s,
          float& r, float& g, float& b )
{
float  m1, m2;

       if ( l <= 0.5 )
                  m2 = l * ( 1 + s );
       else
                  m2 = l + s - l * s;

       m1 = 2 * l - m2;

       if ( s == 0 )
                  if ( h == HUE_UNDEFINED )
                             r = g = b = l;
                  else
                             error ();
       else
       {
                  r = HLSValue ( m1, m2, h + 120 );
                  g = HLSValue ( m1, m2, h );
```

```
                b = HLSValue ( m1, m2, h - 120 );
        }
}
float   HLSValue ( float m1, float m2, float hue )
{
        if ( hue >= 360 )
                hue -= 360;
        if ( hue < 0 )
                hue += 360;
        if ( hue < 60 )
                return m1 + ( m2 - m1 ) * hue / 60;
        else
        if ( hue < 180 )
                return m2;
        else
        if ( hue < 240 )
                return m1 + ( m2 - m1 ) * ( 240 - hue ) / 60;
        else
                return m1;
}
```

Дополнительную информацию по вопросам, затронутым в этой главе, можно ти в Internet по следующим адресам:
ftp://ftp.inforamp.net/pub/users/poynton/doc/colour/
http://www.inforamp.net/~poynton/
ftp://ftp.westminster.ac.uk/pub/itrg/

Упражнения

Докажите, что для двух произвольных точек на хроматической диаграмме взвешенная сумма соответствующих цветов лежит на отрезке, соединяющем эти точки.

Напишите процедуру преобразования цвета из модели RGB в модель HLS.

Напишите программу на нахождения значения γ для своего монитора. Стандартный подход заключается в построении квадрата, заполняемого шаблоном из черных и белых точек, дающим среднюю интенсивность в 0.5 от белого. В центре этого квадрата рисуется квадрат меньшего размера, заполненный серым цветом. Для этого серого цвета путем подбора RGB-значений в палитре добиваются совпадения средних интенсивностей и параметр γ находится из соотношения $R^{\gamma} = 0.5$, где R - нормированное (т. е. лежащее в промежутке [0,1]) значение красной компоненты (вместо красной можно взять любую другую, так как для оттенков серого все три RGB-компоненты совпадают между собой).

Глава 2
РАСПРОСТРАНЕНИЕ СВЕТА. ОСВЕЩЕННОСТЬ

Во взаимодействии света с веществом можно выделить два основных аспекта: распространение света в однородной среде и взаимодействие света с границей раздела двух сред.

Распространение света в однородной среде происходит вдоль прямолинейной траектории с постоянной скоростью. Отношение скорости распространения света в вакууме к скорости распространения света в среде называется коэффициентом преломления (индексом рефракции) среды η. Обычно этот коэффициент зависит от длины волны λ луча (эту зависимость мы в дальнейшем будем игнорировать).

Среда может также поглощать свет, проходящий через нее. При этом имеет место экспоненциальное затухание света с коэффициентом $e^{-\beta l}$, где l - расстояние, пройденное лучом в среде, а β - коэффициент затухания, зависящий от среды.

При взаимодействии с границей двух сред происходит отражение и преломление света. Рассмотрим несколько идеальных моделей, в каждой из которых границей раздела сред является плоскость.

2.1. Зеркальное отражение

Отраженный луч падает в точку P в направлении i и отражается в направлении, задаваемом вектором r, который определяется следующим законом: вектор r лежит в той же плоскости, что вектор i и вектор внешней нормали к поверхности n, и направлен так, что угол падения θ_i равен углу отражения θ_r (рис. 2.1).

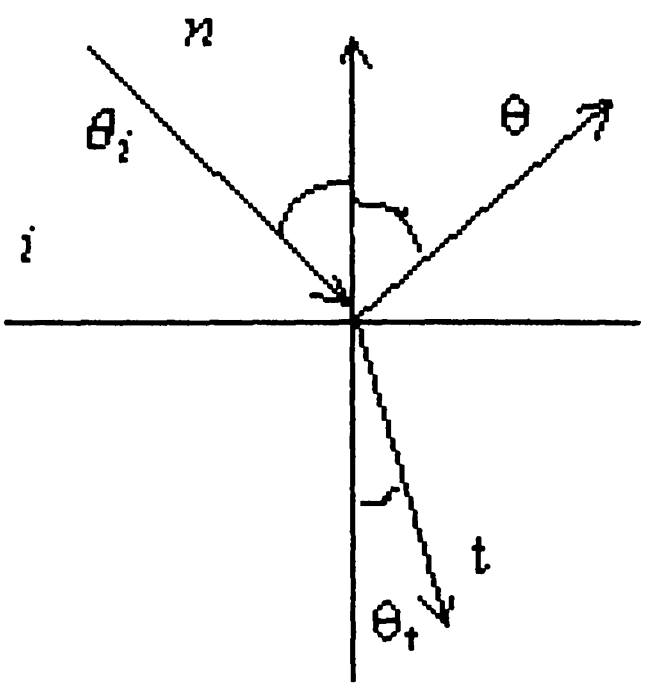


Рис. 2.1

Будем считать все векторы единичными. Тогда из первого условия следует, что вектор r равен линейной комбинации векторов i и n, то есть

$$r = \alpha i + \beta n.$$

Так как $\theta_i = \theta_r$, то $(-i, n) = \cos\theta_i = \theta_r = (r, n)$.
Отсюда легко получается формула

$$r = i - 2\,(i, n)n. \qquad (2.1)$$

Вектор, задаваемый соотношением (2.1), является единичным.

2.2. Диффузное отражение

Идеальное диффузное отражение описывается законом Ламберта, согласно которому падающий свет рассеивается во все стороны с одинаковой интенсивностью. Таким образом, однозначно определенного направления, в котором бы отражался падающий луч не существует; все направления равноправны, и освещенность точки пропорциональна только доле площади, видимой от источника, т. е. (i, n).

2.3. Идеальное преломление

Луч, падающий в точку P в направлении вектора i, преломляется внутрь второй среды в направлении вектора t (рис. 2.1). Преломление подчиняется закону Снеллиуса, согласно которому векторы i, n и t лежат в одной плоскости, а для углов справедливо соотношение

$$\eta_i \sin\theta_i = \eta_t \sin\theta_t , \qquad (2.2)$$

где η_i - коэффициент преломления для среды, откуда идет луч, а η_t - для среды, в которую он входит.

Найдем явное выражение для вектора i, представив его в следующем виде:

$$t = \alpha i + \beta n .$$

Соотношение (2.2) можно переписать так:

$$\sin\theta_t = \eta \sin\theta_i ,$$

где $\eta = \dfrac{\eta_i}{\eta_t}$

Тогда

$$\eta^2 \sin^2\theta_i = \sin^2\theta_t$$

или

$$\eta^2(1-\cos^2\theta_i) = 1-\cos^2\theta_t .$$

Так как

$$\cos\theta_i = (-i, n) , \ \cos\theta_t = (-t, n) ,$$

то $$\alpha^2 (i,n)^2 + 2\alpha\beta(i,n) + \beta^2 = 1 + \eta^2((i,n)^2 - 1) \qquad (2.3)$$

Из условия нормировки вектора t имеем:

$$\|t\|^2 = (t,t) = \alpha^2 + 2\alpha\beta(i,n) + \beta^2 = 1$$

Вычитая это соотношение из равенства (2.3), получим

$$\alpha^2((i,n)^2 - 1) = \eta^2((i,n)^2 - 1) ,$$

откуда $\alpha = \pm\eta$.

Из физических соображений ясно, что $\alpha = \eta$.

Второй параметр определяется из уравнения

$$\beta^2 + 2\beta\eta(i,n) + \eta^2 - 1 = 0,$$

дискриминант которого равен

$$D = 4\left\{1 + \eta^2((i,n)^2 - 1)\right\}.$$

Решение этого уравнения задается формулой

$$\beta = \frac{-2\eta \pm 2\sqrt{1 + \eta^2((i,n)^2 - 1)}}{2},$$

и, значит,

$$t = \eta i + \left\{\eta C_i - \sqrt{1 + \eta^2(C_i^2 - 1)}\right\}n,$$

где $C_i = \cos\theta_i = -(i,n)$

Случай, когда выражение над корнем отрицательно

$$1 + \eta^2\left(C_i^2 - 1\right) < 0,$$

соответствует так называемому полному внутреннему отражению (вся световая энергия отражается от границы раздела сред, и преломления фактически не происходит).

2.4. Диффузное преломление

Диффузное преломление полностью аналогично диффузному отражению; преломленный свет распространяется по всем направлениям t, (t, n) < 0, с одинаковой интенсивностью.

2.5. Распределение энергии

Рассмотрим теперь распределение энергии при отражении и преломлении. Из курса физики известно, что доля отраженной энергии задается коэффициентами Френеля

$$F_r(\lambda,\theta) = \frac{1}{2}\left\{\left(\frac{\cos\theta_i - \eta\cos\theta_t}{\cos\theta_i + \eta\cos\theta_t}\right)^2 + \left(\frac{\eta\cos\theta_i - \cos\theta_t}{\eta\cos\theta_i + \cos\theta_t}\right)^2\right\} \qquad (2.4)$$

Формула (2.4) верна для диэлектрических материалов.

Существует и несколько иная форма записи этих соотношений:

$$F_r(\lambda,\theta)=\frac{1}{2}\left(\frac{c-g}{c+g}\right)\left\{1+\left(\frac{c(c+g)-1}{c(c-g)-1}\right)^2\right\},$$

где $c=\cos\theta_i$; $g=\sqrt{\eta^2+c^2-1}=\eta\cos\theta_t$

Для проводников обычно используется формула

$$F_r=\frac{1}{2}\left\{\left(\frac{(\eta_t^2+k_t^2)\cos^2\theta_i-2\eta_t\cos\theta_i+1}{(\eta_t^2+k_t^2)\cos^2\theta_i+2\eta_t\cos\theta_i+1}\right)^2+\right.$$

$$\left.+\left(\frac{(\eta_t^2+k_t^2)-2\eta_t\cos\theta_i+\cos^2\theta_i}{(\eta_t^2+k_t^2)+2\eta_t\cos\theta_i+\cos^2\theta_i}\right)^2\right\},$$

где k_t- индекс поглощения.

2.6. Микрофасетная модель поверхности

Все рассмотренные случаи являются идеализациями. В действительности нет ни идеальных зеркал, ни идеально гладких поверхностей. Поэтому на практике обычно считают, что поверхность состоит из множества случайно ориентированных плоских идеальных микрозеркал (микрограней) с заданным законом распределения (рис. 2.2).

Для описания поверхности, состоящей из случайно ориентированных микрограней, необходимо задать вероятностный закон, описывающий распределение нормалей этих микрограней. Каждой отдельной микрограни ставится в соответствие угол α между нормалью к микрограни h и нормалью к поверхности n (рис. 2.3), который является случайной величиной с некоторым законом распределения.

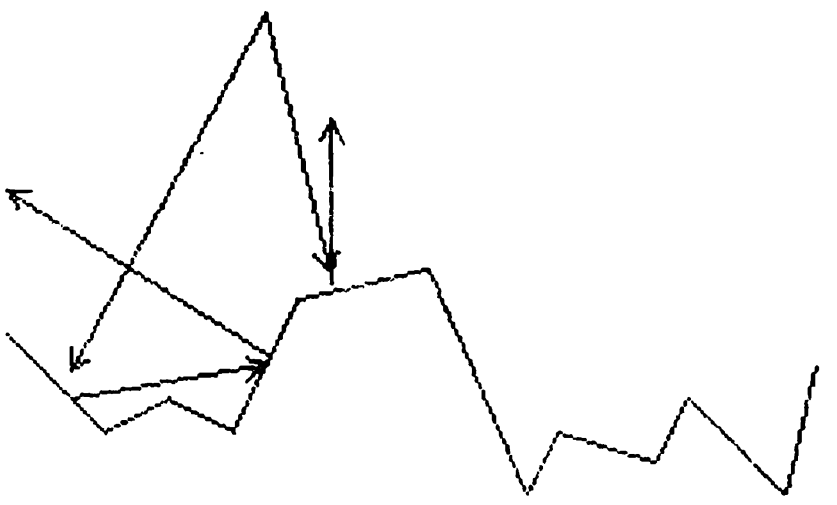


Рис. 2.2

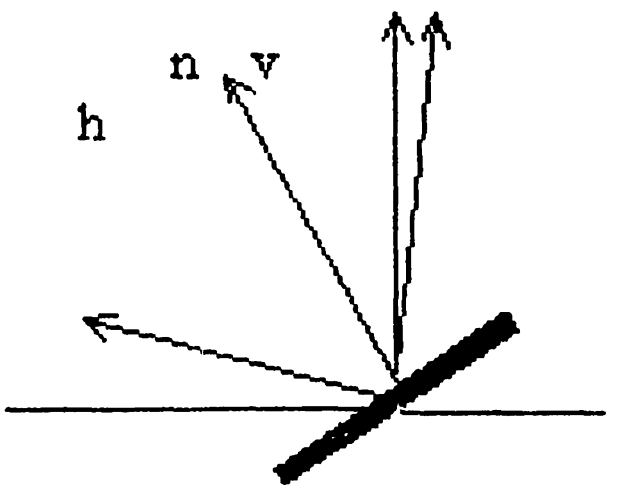


Рис. 2.3

Мы будем описывать поверхность с помощью функции $D(\alpha)$, задающей плотность распределения случайной величины α (для идеально гладкой поверхности функция $D(\alpha)$ совпадает с δ-функцией Дирака).

Существует несколько распространенных моделей. Укажем две из них:

гауссово распределение:

$$D(\alpha) = Ce^{-\left(\frac{\alpha}{m}\right)^2},$$

распределение Бекмена:

$$D(\alpha) = \frac{1}{4\pi m^2 \cos^4 \alpha} e^{-\left(\frac{tg\alpha}{m}\right)^2}$$

В этих моделях величина m характеризует степень неровности поверхности - чем меньше m, тем более гладкой является поверхность.

Рассмотрим отражение луча света, падающего в точку P вдоль направления, задаваемого вектором l. Поскольку микрограни распределены случайным образом, то отраженный луч может уйти практически в любую сторону. Определим долю энергии, уходящей в направлении v. Для того чтобы луч отразился в этом направлении, необходимо, чтобы он попал на микрогрань, нормаль h к которой удовлетворяет соотношению

$$h = \frac{l + v}{\|l + v\|}$$

Доля энергии, которая отразится от микрограни, определяется коэффициентом Френеля $F_r(\lambda, \theta)$, где

$$\theta = \arccos(h, v) = \arccos(h, l),$$

векторы h, v и l единичные.

Если поверхность состоит из множества микрограней, начинает сказываться затеняющее влияние соседних граней, которое обычно описывается с помощью функции

$$G = \min\left(1, \frac{2(n,h)(n,v)}{(v,h)}, \frac{2(n,h)(n,l)}{(v,h)}\right),$$

где n - вектор внешней нормали к поверхности.

В этом случае интересующая нас доля энергии задается формулой

$$BRDF(l, v, \lambda) = \frac{F_r(\lambda, \theta) D(\alpha) G(n, v, l)}{(n, l)(n, v)} \qquad (2.5)$$

Преломление света поверхностью, состоящей из микрозеркал, рассматривается совершенно аналогично.

С использованием соотношения (2.5) можно построить формулу, полностью описывающую энергию (и отраженную, и преломленную) в заданном направлении. Функция, показывающая, какая именно доля энергии, пришедшей в направлении, задаваемом вектором l, уходит в направлении, задаваемом вектором v, называется двунаправленной функций отражения (Bidirectional Reflection Distribution Function - BRDF). Для поверхностей, состоящих из множества микрограней, BRDF задается выражением (2.5).

В случае идеальной диффузной поверхности функция BRDF постоянна, а в случае идеальной зеркальной поверхности задается при помощи δ-функции Дирака.

В связи с тем что вычисление BRDF по формуле (2.5) оказывается слишком сложным, на практике обычно используются более простые формулы, например такая:

$$\mathrm{BRDF}(l, v, \lambda) = (n, h)^k F_r(\lambda, \theta). \qquad (2.6)$$

В общем случае BRDF удовлетворяет условию симметричности

$$\mathrm{BRDF}(l, v) = \mathrm{BRDF}(v, l).$$

Обозначим через $R_n(a)$ оператор поворота вокруг вектора нормали n на угол α. Если для всех α выполняется равенство

$$\mathrm{BRDF}(R_n(\alpha)l, R_n(\alpha)v) = \mathrm{BRDF}(l, v),$$

то такой материал называется изотропным, и анизотропным в противном случае. В дальнейшем мы будем рассматривать только изотропные материалы.

Несмотря на то что коэффициенты Френеля заметно влияют на степень реалистичности изображения, на практике их применяют очень редко. Дело в том, что их использование наталкивается на ряд серьезных препятствий, одним из которых является сложность вычисления, а другим - отсутствие точной информации о зависимости величин, входящих в состав формулы, от длины волны λ. Поэтому часто вместо формулы (2.6) используется более простая формула

$$\mathrm{BRDF}(l, v, \lambda) = (n, h)^k$$

В общем случае освещенность разбивается на непосредственную освещенность (отраженный и преломленный свет, идущие непосредственно от источников) и вторичную освещенность (отраженный и преломленный свет, идущие от других поверхностей).

Здесь мы рассмотрим модели, учитывающие только первичную освещенность (более сложные модели будут рассмотрены в главах, посвященных методам трассировки лучей и излучательности).

Для компенсации неучтенной вторичной освещенности вводится так называемая фоновая освещенность - равномерная освещенность, идущая со всех сторон и ни от чего не зависящая. Кроме того, считается, что каждый материал проявляет как диффузные, так и зеркальные свойства (с заданными весами). Поэтому в итоговую формулу входят члены трех видов - отвечающие за фоновую освещенность, за диффузную освещенность и за зеркальную (микрофасетную) освещенность.

Простейшую модель освещенности можно описать при помощи соотношения

$$I(\lambda) = K_a I_a(\lambda) C(\lambda) + K_d C(\lambda) \sum_i I_{l_i}(\lambda)(n, l_i) + K_s \sum_i I_{l_i}(\lambda)(n, h_i)^p$$

где $I_a(\lambda)$ - интенсивность фонового освещения,

$I_{l_i}(\lambda)$ - интенсивность *i*-го источника света,

$C(\lambda)$ - цвет в точке P,

K_a - коэффициент фонового освещения,

K_d - коэффициент диффузного освещениz,

K_s - коэффициент зеркального освещения,

n - вектор внешней нормали в точке *P*,

l_i - единичный вектор направления из точки P на *i*-й источник света.

Иногда используется так называемая металлическая модель, учитывающая тот факт, что для металла (в отличие от пластика) цвет блика совпадает с цветом металла. Металлическая модель задается следующим соотношением:

$$I(\lambda) = K_a I_a(\lambda) C(\lambda) + K_d C(\lambda) \sum_i I_{l_i}(\lambda)(n, l_i) + K_s C(\lambda) \sum_i I_{l_i}(\lambda)(n, h_i)^p$$

Упражнения

1. Покажите, что вектор, задаваемый соотношением (2.1), действительно является единичным.
2. Напишите процедуру преобразования цвета из модели RGB в модель HLS.
3. Напишите программу на нахождения значения γ для своего монитора. Стандартный подход заключается в построении квадрата, заполняемого шаблоном из черных и белых точек, дающим среднюю интенсивность в 0.5 от белого. В центре этого квадрата рисуется квадрат меньшего размера, заполненный серым цветом. Для этого серого цвета путем подбора RGB-значений в палитре добиваются совпадения средних интенсивностей и параметр γ находится из соотношения $R^r = 0{,}5$, где R - нормированное (т. е. лежавщее в промежутке [0,1]) значение красной компоненты (вместо красной можно взять любую другую, так как для оттенков серого все три RGB-компоненты совпадают между собой).

Глава 3
ГРАФИЧЕСКИЕ ПРИМИТИВЫ В ЯЗЫКАХ ПРОГРАММИРОВАНИЯ

Графические устройства делятся на векторные и растровые. Векторные устройства (например, графопостроители) представляют изображение в виде линейных объектов. На большинстве ЭВМ (включая и IBM PC/AT) принят растровый способ изображения графической информации - изображение представлено прямоугольной матрицей точек (пикселов), и каждый пиксел имеет свой цвет, выбираемый из заданного набора цветов - палитры. Для реализации этого подхода компьютер содержит в своем составе видеоадаптер, который, с одной стороны, хранит в своей памяти (ее принято называть видеопамятью) изображение (при этом на каждый пиксел изображения отводится фиксированное количество бит памяти), а с другой - обеспечивает регулярное (50-70 раз в секунду) отображение видеопамяти на экране монитора. Размер палитры определяется объемом видеопамяти, отводимой под 1 пиксел, и зависит от типа видеоадаптера.

Для ПЭВМ типа IBM PC/AT и PS/2 существует несколько различных типов видеоадаптеров, различающихся как своими возможностями, так и аппаратным устройством и принципами работы с ними. Основными видеоадаптерами для этих машин являются CGA, EGA, VGA и Hercules. Существует также большое количество адаптеров, совместимых с EGA/VGA, но предоставляющих по сравнению с ними ряд дополнительных возможностей.

Практически каждый видеоадаптер поддерживает несколько режимов работы, отличающихся друг от друга размерами матрицы пикселов (разрешением) и размером палитры (количеством цветов, которые можно одновременно отобразить на экране). Разные режимы даже одного адаптера зачастую имеют разную организацию видеопамяти и способы работы с ней. Более подробную информацию о работе с видеоадаптерами можно получить из следующей главы.

Большинство адаптеров строится по принципу совместимости с предыдущими. Так, адаптер EGA поддерживает все режимы адаптера CGA. Поэтому любая программа, рассчитанная на работу с адаптером CGA, будет также работать и с адаптером EGA, даже не замечая этого. При этом адаптер EGA поддерживает еще ряд своих собственных режимов. Аналогично адаптер VGA поддерживает все режимы адаптера EGA.

Фактически любая графическая операция сводится к работе с отдельными пикселами - поставить точку заданного цвета и узнать цвет заданной точки. Однако большинство графических библиотек поддерживают работу и с более сложными объектами, поскольку работа только на уровне отдельно взятых пикселов была бы очень затруднительной для программиста и к тому же неэффективной.

Среди подобных объектов (представляющих собой объединения пикселов) можно выделить следующие основные группы:

- линейные изображения (растровые образы линий);
- сплошные объекты (растровые образы двумерных областей);

- шрифты;
- изображения (прямоугольные матрицы пикселов).

Как правило, каждый компилятор имеет свою графическую библиотеку, обеспечивающую работу с основными группами графических объектов. При этом требуется, чтобы подобная библиотека поддерживала работу с основными типами видеоадаптеров.

Существует несколько путей обеспечения этого.

Один из них заключается в написании версий библиотеки для всех основных типов адаптеров. Однако программист должен изначально знать, для какого конкретно видеоадаптера он пишет свою программу, и использовать соответствующую библиотеку. Полученная программа уже не будет работать на других адаптерах, несовместимых с тем, для которого писалась программа. Поэтому вместо одной программы получается целый набор программ для разных видеоадаптеров. Принцип совместимости адаптеров выручает здесь мало: хотя программа, рассчитанная на адаптер CGA, и будет работать на VGA, но она не сможет полностью использовать все его возможности и будет работать с ним только как с CGA.

Можно включить в библиотеку версии процедур для всех основных типов адаптеров. Это обеспечит некоторую степень машинной независимости. Однако нельзя исключать случай наличия у пользователя программы какого-либо типа адаптера, не поддерживаемого библиотекой (например, SVGA). Но самым существенным недостатком такого подхода является слишком большой размер полученнего выполняемого файла, что уменьшает объем оперативной памяти, доступной пользователю.

Наиболее распространенным является использование драйверов устройств. Выделяется некоторый основной набор графических операций, так, что все остальные операции можно реализовать, используя только операции основного набора. Привязка к видеоадаптеру заключается именно в реализации этих основных (базисных) операций. Для каждого адаптера пишется так называемый драйвер - небольшая программа со стандартным интерфейсом, реализующая все эти операции для данного адаптера и помещаемая в отдельный файл. Библиотека в начале своей работы определяет тип имеющегося видеоадаптера и загружает соответствующий драйвер в память. Таким образом достигается почти полная машинная независимость написанных программ.

Рассмотрим работу одной из наиболее популярных графических библиотек - библиотеки компилятора Borland C++.

Для использования этой библиотеки необходимо сначала подключить ее при помощи команды меню Options/Linker/Libraries.

Опишем основные группы операций.

3.1. Инициализация и завершение работы с библиотекой

Для инициализации библиотеки служит функция

```
void far initgraph (int far *driver, int far *mode, char far *path);
```

Первый параметр задает библиотеке тип адаптера, с которым будет вестись работа. В соответствии с этим параметром загружается драйвер указанного видеоадап-

тера и производится инициализация всей библиотеки. Определен ряд констант, задающих набор стандартных драйверов: CGA, EGA, VGA, DETECT.

Значение DETECT сообщает библиотеке о том, что тип имеющегося видеоадаптера надо определить ей самой и выбрать для него режим наибольшего разрешения.

Второй параметр - mode - определяет режим.

Параметр	*Режим*
CGAC0, CGAC1, CGAC2, CGAC3	320 на 200 точек на 4 цвета
CGAHI	640 на 200 точек на 2 цвета
EGALO	640 на 200 точек на 16 цветов
EGAHI	640 на 350 точек на 16 цветов
VGALO	640 на 200 точек на 16 цветов
VGAMED	640 на 350 точек на 16 цветов
VGAHI	640 на 480 точек на 16 цветов

Если в качестве первого параметра было взято значение DETECT, то параметр mode не используется.

В качестве третьего параметра выступает имя каталога, где находится драйвер адаптера - файл типа BGI (Borland's Graphics Interface):

- CGA.BGI - драйвер адаптера CGA;
- EGAVGA.BGI - драйвер адаптеров EGA и VGA;
- HERC.BGI - драйвер адаптера Hercules.

Функция graphresult возвращает код завершения предыдущей графической операции
int far graphresult (void);

Успешному выполнению соответствует значение функции grOk.

Для окончания работы с библиотекой необходимо вызвать функцию closegraph:
void far closegraph (void);

Ниже приводится простейший пример, инициализирующий графическую библиотеку, рисующий прямоугольную рамку по границам экрана и завершающий работу с графической библиотекой.

```
// File example1.cpp
#include  <conio.h>
#include  <graphics.h>
#include  <process.h>
#include  <stdio.h>
main ()
{
  int  mode;
  int  res;
  int  driver = DETECT;
  initgraph ( &driver, &mode, "" );
  if ( ( res = graphresult () ) != grOk )
  {
    printf("\nGraphics error: %s\n", grapherrormsg ( res) );
    exit ( 1 );
  }
```

```
  line ( 0, 0, 0, getmaxy () );
  line ( 0, getmaxy (), getmaxx (), getmaxy () );
  line ( getmaxx (), getmaxy (), getmaxx (), 0 );
  line ( getmaxx (), 0, 0, 0 );
  getch ();
  closegraph ();
}
```

Программа переходит в графический режим и рисует по краям экрана прямо угольник. В случае ошибки выдается стандартное диагностическое сообщение.

После инициализации библиотеки адаптер переходит в соответствующий режим, экран очищается и на нем устанавливается следующая координатная система (рис. 3.1): начальная точка с координатами (0, 0) располагается в левом верхнем углу экрана.

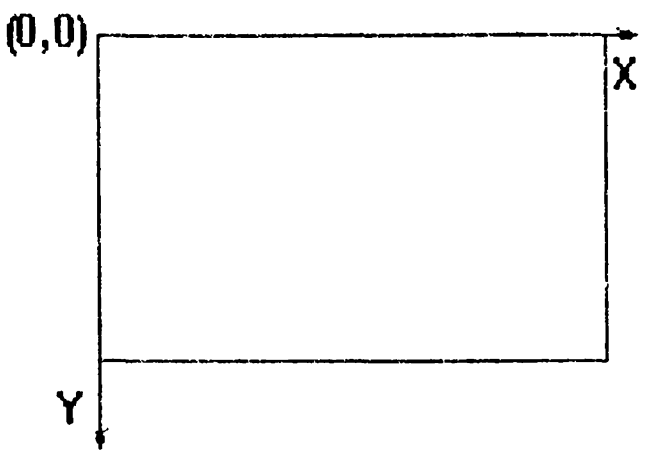


Рис. 3.1

Узнать максимальные значения *x* и *y* координат пиксела можно, используя функции getmaxx и getmaxy:

```
int far getmaxx ( void );
int far getmaxy ( void );
```

Узнать, какой именно режим в действительности установлен, можно при помощи функции getgraphmode:

```
int far getgraphmode ( void );
```

Для очистки экрана удобно использовать функцию clearviewport:

```
void far clearviewport ( void );
```

3.2. Работа с отдельными точками

Функция putpixel ставит пиксел заданного цвета color в точке с координатами (*x*, *y*):

```
void far putpixel ( int x, int y, int color );
```

Функция getpixel возвращает цвет пиксела с координатами (*x*, *y*):

```
unsigned far getpixel ( int x, int y );
```

3.3. Рисование линейных объектов

При рисовании линейных объектов основным инструментом является перо, которым эти объекты рисуются. Перо имеет следующие характеристики:

- цвет (по умолчанию белый);
- толщина (по умолчанию 1);
- шаблон (по умолчанию сплошной).

Шаблон служит для рисования пунктирных и штрихпунктирных линий. Для установки параметров пера используются следующие функции выбора.

Процедура setcolor устанавливает цвет пера:

```
void far setcolor ( int color );
```

Функция *setlinestyle* определяет остальные параметры пера:

void far setlinestyle (int style, unsigned pattern, int thickness);

Первый параметр задает шаблон линии. Обычно в качестве этого параметра выступает один из предопределенных шаблонов: SOLID_LINE, DOTTED_LINE, CENTER_LINE, DASHED_LINE, USERBIT_LINE. Значение USERBIT_LINE указывает на то, что шаблон задается (пользователем) вторым параметром. Шаблон определяется 8 битами, где значение бита 1 означает, что в соответствующем месте будет поставлена точка, а значение 0 - что точка ставиться не будет.

Третий параметр задает толщину линии в пикселах. Возможные значения параметра - NORM_WIDTH и THICK_WIDTH (1 и 3).

При помощи пера можно рисовать ряд линейных объектов - прямолинейные отрезки, дуги окружностей и эллипсов, ломаные.

3.3.1. Рисование прямолинейных отрезков

Функция *line* рисует отрезок, соединяющий точки (x_1, y_1) и: (x_2, y_2)
void far line (int x1, int y1, int x2, int y2);

3.3.2. Рисование окружностей

Функция *circle* рисует окружность радиуса *r* с центром в точке (x, y):
void far circle (int x, int y, int r);

3.3.4. Рисование дуг эллипса

Функции arc и ellipse рисуют дуги окружности (с центром в точке (*x*, *y*) и радиусом *r*) и эллипса (с центром (*x*, *y*), полуосями *rx* и *ry*, параллельными координатным осям) начиная с угла *startAngle* и заканчивая углом *endAngle*.

Углы задаются в градусах в направлении против часовой стрелки (рис. 3.2):

void far arc (int x, int y, int startAngle, int endAngle, int r);
void far ellipse (int x, int y, int startAngle, int endAngle, int rx, int ry);

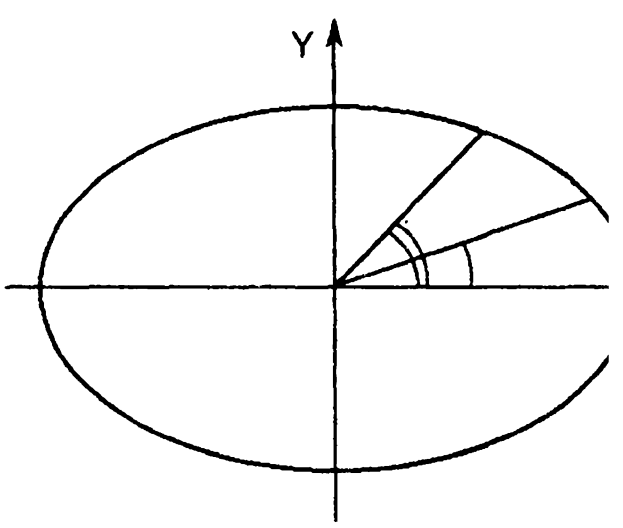


Рис. 3.2

3.4. Рисование сплошных объектов

3.4.1. Закрашивание объектов

С понятием закрашивания тесно связано понятие кисти. Кисть определяется цветом и шаблоном - матрицей 8 на 8 точек (бит), где бит, равный единице, означает, что нужно ставить точку цвета кисти, а 0 - что нужно ставить черную точку (цвета 0).

Для задания кисти используются следующие функции:
void far setfillstyle (int pattern, int color);
void far setfillpattern (char far * pattern, int color);

Функция setfillstyle задает кисть. Параметр style определяет шаблон кисти либо как один из стандартных (EMPTY_FILL, SOLID_FILL, LINE_FILL, LTSLASH_FILL), либо как шаблон, задаваемый пользователем (USER_FILL). Поль-

зовательский шаблон устанавливает процедура setfillpattern, первый параметр в которой и задает шаблон - матрицу 8 на 8 бит, собранных по горизонтали в байты. По умолчанию используется сплошная кисть (SOLID_FILL) белого цвета.

Процедура bar закрашивает выбранной кистью прямоугольник с левым верхним углом (x_1, y_1) и правым нижним углом (x_2, y_2):

```
void far bar ( int x1, int y1, int x2, int y2 );
```

Функция fillellipse закрашивает сектор эллипса:

```
void far fillellipse (int x, int y, int startAngle,
                      int endAngle, int rx, int ry);
```

Функция floodfill служит для закраски связной области, ограниченной линией цвета borderColor и содержащей точку (x, y) внутри себя:

```
void far floodfill ( int x, int y, int borderColor );
```

Функция fillpoly осуществляет закраску многоугольника, заданного массивом значений x и y координат:

```
void far fillpoly ( int numpoints, int far * points );
```

3.4.2. Работа с изображениями

Библиотека поддерживает также возможность запоминания прямоугольного фрагмента изображения в обычной (оперативной) памяти и вывода его на экран. Это может использоваться для сохранения изображения в файл, создания мультипликации и т. п.

Объем памяти в байтах, требуемый для запоминания фрагмента изображения, можно получить при помощи функции imagesize:

```
unsigned far imagesize (int x1, int y1, int x2, int y2);
```

Для запоминания изображения служит процедура getimage:

```
void far getimage (int x1, int y1, int x2, int y2, void far * image);
```

При этом прямоугольный фрагмент, определяемый точками (x_1, y_1) и (x_2, y_2), записывается в область памяти, задаваемую последним параметром - image.

Для вывода изображения служит процедура putimage:

```
void far putimage (int x, int y, void far * image, int op);
```

Хранящееся в памяти изображение, которое задается параметром image, выводится на экран так, чтобы точка (x, y) была верхним левым углом изображения. Последний параметр определяет способ наложения выводимого изображения на уже имеющееся на экране (см. функцию setwritemode). Поскольку значение (цвет) каждого пиксела представлено фиксированным количеством бит, то в качестве возможных вариантов наложения выступают побитовые логические операции. Значения для параметра op приведены ниже:

- COPY_PUT - происходит простой вывод (замещение);
- NOT_PUT - происходит вывод инверсного изображения;
- OR_PUT - используется побитовая операция ИЛИ;
- XOR_PUT - используется побитовая операция ИСКЛЮЧАЮЩЕЕ ИЛИ;
- AND_PUT - используется побитовая операция И.

Рассмотрим, каким образом действует параметр op. На рис. 3.3 приведены возможные варианты наложения первого изображения (source) на второе (destination).

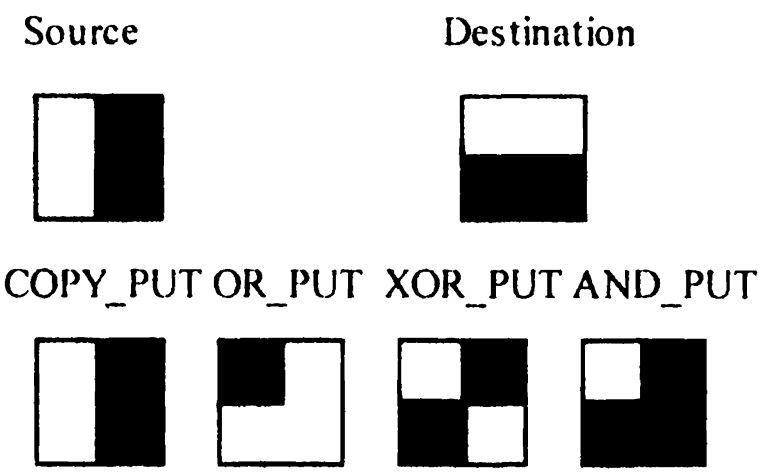


Рис. 3.3

```
// get/putimage example
unsigned  imageSize = imagesize ( x1, y1, x2, y2 );
void    * image     = malloc ( imageSize );
. . .
if ( image != NULL )
  getimage ( x1, y1, x2, y2, image );
. . .
if ( image != NULL )
{
  putimage ( x, y, image, COPY_PUT );
  free ( image );
}
```

В этой программе происходит динамическое выделение под заданный фрагмент изображения на экране требуемого объема памяти. Этот фрагмент сохраняется в отведенной памяти. Далее сохраненное изображение выводится на новое место (в вершину левого верхнего угла - (x, y)), и отведенная под изображение память освобождается.

3.5. Работа со шрифтами

Под шрифтом обычно понимается набор изображений символов. Шрифты могут различаться по организации (растровые и векторные), по размеру, по направлению вывода и по ряду других параметров. Шрифт может быть фиксированным (размеры всех символов совпадают) или пропорциональным (высоты символов совпадают, но они могут иметь разную ширину).

Для выбора шрифта и его параметров служит функция settextstyle:
`void far settextstyle (int font, int direction, int size );`

Здесь параметр font задает идентификатор одного из шрифтов:

- DEFAULT_FONT - стандартный растровый шрифт размером 8 на 8 точек, находящийся в ПЗУ видеоадаптера;
- TRIPLEX_FONT, GOTHIC_FONT, SANS_SERIF_FONT, SMALL_ FONT – стандартные пропорциональные векторные шрифты, входящие в комплект Borland C++ (шрифты хранятся в файлах типа CHR и по этой команде подгружаются в оперативную память; файлы должны находиться в том же каталоге, что и драйверы устройств).

Параметр direction задает направление вывода:

- HORIZ_DIR - вывод по горизонтали;

- VERT_DIR - вывод по вертикали.

Параметр size задает, во сколько раз нужно увеличить шрифт перед выводом на экран. Допустимые значения 1, 2, ..., 10.

При желании можно использовать любые шрифты в формате CHR. Для этого надо сначала загрузить шрифт при помощи функции

```
int far installuserfont ( char far * fontFileName );
```

а затем возвращенное функцией значение передать settextstyle в качестве идентификатора шрифта:

```
int myFont = installuserfont ( "MYFONT.CHR" );
settextstyle ( myFont, HORIZ_DIR, 5 );
```

Для вывода текста служит функция outtextxy:

```
void far outtextxy ( int x, int y, char far * text );
```

При этом строка text выводится так, что точка (x, y) оказывается вершиной левого верхнего угла первого символа.

Для определения размера, который займет на экране строка текста при выводе текущим шрифтом, используются функции, возвращающие ширину и высоту в пикселах строки текста:

```
int far textwidth (char far * text );
int far textheight (char far * text );
```

3.6. Понятие режима (способа) вывода

При выводе изображения на экран обычно происходит замещение пиксела, ранее находившегося на этом месте, на новый. Можно, однако, установить такой режим, что в видеопамять будет записываться результат наложения ранее имевшегося значения на выводимое. Поскольку каждый пиксел представлен фиксированным количеством бит, то совершенно естественно, что в качестве такого наложения выступают побитовые операции. Для установки используемой операции служит процедура setwritemode:

```
void far setwritemode (int mode);
```

Параметр mode задает способ наложения и может принимать одно из значений:

- COPY_PUT- происходит простой вывод (замещение);
- XOR_PUT - используется побитовая операция ИСКЛЮЧАЮЩЕЕ ИЛИ.

Режим XOR_PUT удобен тем, что повторный вывод одного и того же изображения на то же место уничтожает результат первого вывода, восстанавливая изображение, которое до этого было на экране.

Замечание. Не все функции графической библиотеки поддерживают использование режимов вывода, например, функции закраски игнорируют установленный режим наложения (вывода). Кроме того, некоторые функции могут не совсем корректно работать в режиме XOR_PUT

3.7. Понятие окна (порта вывода)

При желании пользователь может создать на экране окно - своего рода маленький экран со своей локальной системой координат. Для этого служит функция setviewport:

```
void far setviewport (int x1, int y1, int x2, int y2, int clip);
```

Эта функция устанавливает окно с глобальными координатами $(x_1, y_1) - (x_2, y_2)$. При этом локальная система координат вводится так, чтобы точке с координатами (0, 0) соответствовала точка с глобальными координатами (x_1, y_1). Это означает, что локальные координаты отличаются от глобальных координат лишь сдвигом на (x_1, y_1). Все процедуры рисования (кроме setviewport) всегда работают с локальными координатами. Параметр clip определяет, нужно ли проводить отсечение изображения, не помещающегося внутрь окна, или нет.

Замечание. Отсечение ряда объектов проводится не совсем корректно; так, функция outtextxy производит отсечение не на уровне пикселов, а по символам.

3.8. Понятие палитры

Адаптер EGA и все совместимые с ним адаптеры предоставляют дополнительные возможности по управлению цветом. Наиболее распространенной схемой представления цветов для видеоустройств является так называемое RGB-представление, в котором любой цвет задается как сумма трех основных цветов - красного (Red), зеленого (Green) и синего (Blue) с заданными интенсивностями. Все пространство цветов представляется в виде единичного куба, и каждый цвет определяется тройкой чисел (r, g, b). Например, желтый цвет задается как (1, 1, 0), а малиновый - как (1, 0, 1). Белому цвету соответствует набор (1, 1, 1), а черному - (0, 0, 0).

Обычно под хранение каждой из компонент цвета отводится фиксированное количество *n* бит памяти. Поэтому допустимый диапазон значений для компонент цвета $[0,2^n-1]$, а не [0, 1].

Практически любой видеоадаптер способен отобразить значительно большее количество цветов, чем определяется количеством бит, отводимых в видеопамяти под 1 пиксел. Для использования этой возможности вводится понятие палитры.

Палитра - это массив, в котором каждому возможному значению пиксела ставится в соответствии значение цвета (r, g, b), выводимое на экран. Размер палитры и ее организация зависят от типа используемого видеоадаптера.

Наиболее простой является организация палитры на EGA-адаптере. Под каждый из 16 возможных логических цветов (значений пиксела) отводится 6 бит, по 2 бита на каждую цветовую компоненту. При этом цвет в палитре задается байтом вида

```
00rgbRGB,
```

где r, g, b, R, G, B могут принимать значения 0 или 1.

Используя функцию setpalette

```
void far setpalette ( int color, int colorValue );
```

можно для любого из 16 логических цветов задать любой из 64 возможных физических цветов.

Функция getpalette

```
void far getpalette ( struct palettetype far * palette );
```

служит для получения текущей палитры, которая возвращается в виде следующей структуры:

```
struct palettetype
{
  unsigned char size;
  signed char  colors [MAXCOLORS+1];
};
```

Приведенная ниже программа демонстрирует использование палитры для получения четырех оттенков красного цвета.

```
// File  example2.cpp
#include  <conio.h>
#include  <graphics.h>
#include  <process.h>
#include  <stdio.h>
// show 4 shades of red
main ()
{
  int  driver = DETECT;
  int  mode;
  int  res;
  int  i;
  initgraph ( &driver, &mode, "" );
  if ( ( res = graphresult () ) != grOk )
  {
    printf("\nGraphics error: %s\n", grapherrormsg (res) );
    exit ( 1 );
  }
  setpalette ( 0, 0  );
  setpalette ( 1, 32 );
  setpalette ( 2, 4  );
  setpalette ( 3, 36 );
  bar ( 0, 0, getmaxx (), getmaxy () );
  for ( i = 0; i < 4; i++ )
  {
    setfillstyle ( SOLID_FILL, i );
    bar ( 120 + i*100, 75, 219 + i*100, 274 );
  }
  getch ();
  closegraph ();
}
```

Реализация палитры для 16-цветных режимов адаптера VGA намного сложнее. Помимо поддержки палитры адаптера EGA видеоадаптер дополнительно содержит 256 специальных DAC-регистров, где для каждого цвета хранится его 18-битовое представление (по 6 бит на каждую компоненту). При этом исходному логическому номеру цвета с использованием 6-битовых регистров палитры EGA ставится в соответствие, как и раньше, значение от 0 до 63, но оно уже является не RGB-разложением цвета, а номером DAC-регистра, содержащего физический цвет.

Для установки значений DAC-регистров служит функция setrgbpalette:

```
void far setrgbpalette ( int color, int red, int green, int blue );
```

Следующий пример переопределяет все 16 цветов адаптера VGA в 16 оттенков серого цвета.

```
// File  example3.cpp
#include  <conio.h>
#include  <graphics.h>
#include  <process.h>
#include  <stdio.h>
main ()
{
 int    driver = VGA;
 int    mode = VGAHI;
 int    res;
 palettetype    pal;
 initgraph ( &driver, &mode, "" );
 if ( ( res = graphresult () ) != grOk )
 {
  printf("\nGraphics error: %s\n", grapherrormsg (res) );
  exit ( 1 );
 }
 getpalette ( &pal );
 for ( int i = 0; i < pal.size; i++ )
 {
  setrgbpalette ( pal.colors [i], (63*i)/15, (63*i)/15,
                  (63*i)/15 );
  setfillstyle  ( SOLID_FILL, i );
  bar ( i*40, 100, 39 + i*40, 379 );
 }
 getch ();
 closegraph ();
}
```

Для 256-цветных режимов адаптера VGA значение пиксела используется непосредственно для индексации массива DAC-регистров.

3.9. Понятие видеостраниц и работа с ними

Для большинства режимов (например, для EGAHI) объем видеопамяти, необходимый для хранения всего изображения (экрана), составляет менее половины имеющейся видеопамяти (256 Кбайт для EGA и VGA). В этом случае вся видеопамять делится на равные части (их количество обычно является степенью двойки), называемые страницами, так, что для хранения всего изображения достаточно одной страницы. Для режима EGAHI видеопамять делится на две страницы - 0-ю (адрес 0xA000:0) и 1-ю (адрес 0xA000: 0x8000).

Видеоадаптер отображает на экран только одну из имеющихся у него страниц. Эта страница называется видимой и устанавливается следующей процедурой:

```
void far setvisualpage ( int page );
```

где page - номер той страницы, которая станет видимой на экране после вызова этой процедуры.

Графическая библиотека может осуществлять работу с любой из имеющихся страниц. Страница, с которой работает библиотека, называется активной. Активная страница устанавливается процедурой setactivepage:

```
void far setactivepage ( int page );
```

где page - номер страницы, с которой работает библиотека и на которую происходит весь вывод.

Использование видеостраниц играет очень большую роль при мультипликации.

Реализация мультипликации на ПЭВМ заключается в последовательном рисовании на экране очередного кадра. При традиционном способе работы (кадр рисуется, экран очищается, рисуется следующий кадр) постоянные очистки экрана и построение нового изображения на чистом экране создают нежелательный эффект мерцания.

Для устранения этого эффекта очень удобно использовать страницы видеопамяти: пока на видимой странице пользователь видит один кадр, активная, но невидимая страница очищается и на ней рисуется новый кадр. Как только кадр готов, активная и видимая страницы меняются местами и пользователь вместо старого кадра сразу видит новый.

```
// File example4.cpp
#include <conio.h>
#include <graphics.h>
#include <process.h>
#include <stdio.h>

int xc = 450;    // center of circle
int yc = 100;
int vx = 7;      // velocity
int vy = 5;
int r = 20;      // radius

void drawFrame ( int n )
{
 if ( ( xc += vx ) >= getmaxx () - r || xc < r )
 {
  xc -= vx; vx = -vx;
 }
 if ( ( yc += vy ) >= getmaxy () - r || yc < r )
 {
  yc -= vy; vy = -vy;
 }
 circle ( xc, yc, r );
}

main ()
{
 int driver = EGA;
 int mode   = EGAHI;
 int res;
```

```
  initgraph ( &driver, &mode, "" );
  if ( ( res = graphresult () ) != grOk )
  {
    printf("\nGraphics error: %s\n", grapherrormsg (res) );
    exit ( 1 );
  }

  drawFrame ( 0 );
  setactivepage ( 1 );
  for ( int frame = 1;; frame++ )
  {
    clearviewport ();
    drawFrame ( frame );
    setactivepage ( frame & 2 );
    setvisualpage ( 1 - ( frame & 2 ) );
    if ( kbhit () )
    break;
  }
  getch ();
  closegraph ();
}
```

Замечание. Не все режимы поддерживают работу с несколькими страницами, например VGAHI поддерживает работу только с одной страницей.

3.10. Подключение нестандартных драйверов устройств

Иногда возникает необходимость использовать нестандартные драйверы устройств, например в случае, если вы хотите работать с режимом адаптера VGA разрешением 320 на 200 точек при количестве цветов 256 или режимами адаптера SVGA. Эти режимы стандартными драйверами, входящими в комплект Borland C++, не поддерживаются. Однако существует ряд специальных драйверов, предназначенных для работы с ними. Приведем пример программы, подключающей драйвер для работы с 256-цветным режимом высокого разрешения для VESA-совместимого адаптера SVGA и устанавливающей палитру из 64 оттенков желтого цвета.

```
// File example5.cpp
#include <conio.h>
#include <graphics.h>
#include <process.h>
#include <stdio.h>

int huge myDetect ( void )
{
  return 2; // return suggested mode #
}

main ()
{
  int driver = DETECT;
```

```
  int  mode;
  int  res;
  installuserdriver ( "VESA", MyDetect );
  initgraph ( &driver, &mode, "" );
  if ( ( res = graphresult () ) != grOk )
  {
    printf("\nGraphics error: %s\n", grapherrormsg (res) );
    exit ( 1 );
  }
  for ( int i = 0; i < 64; i++ )
  {
    setrgbpalette ( i, i, i, 0 );
    setfillstyle ( SOLID_FILL, i );
    bar ( i*10, 0, 9 + i*10, getmaxy () );
  }
  getch ();
  closegraph ();
}
```

Последним параметром для функции installuserdriver является функция, определяющая наличие соответствующей карты и возвращающей рекомендуемый режим.

Существует еще один способ подключения нестандартного драйвера устройства, когда вместо адреса проверяющей функции передается NULL, а возвращенное функцией installuserdriver значение используется в качестве первого параметра для функции initgraph; вторым параметром является рекомендуемый номер режима (он зависит от используемого драйвера).

```
// File example6.cpp
int  driver;
int  mode;
int  res;
if ( ( driver = installuserdriver ( "VESA", NULL ) ) == grError )
{
  printf ( "\nCannot load extended driver" );
  exit ( 1 );
}
initgraph ( &driver, &mode, "" );
```

3.11. Построение графика функции

Ниже приводится программа, осуществляющая построение графика функции одной переменной при заданных границах изменения аргумента. Программа автоматически подбирает шаги разбиения осей (целые степени 10) и расставляет необходимые метки. Построение графика осуществляется на всем экране, но программу несложно переделать для построения графика и в заданной прямоугольной области.

```
// File example6.cpp
void   plotGraphic ( float a, float b, float (*f)( float ) )
{
        float       xStep = pow ( 10, floor (log (b - a) / log (10.0)));
        float       xMin  = xStep * floor ( a / xStep );
        float       xMax  = xStep * ceil  ( b / xStep );
```

```
float * fVal  = new float [ 100 ];

for ( int i = 0; i < 100; i++ )
        fVal [i] = f ( a + i * ( b - a ) / 100.0 );

float   yMin = fVal [0],
        yMax = fVal [0];

for ( i = 1; i < 100; i++ )
        if ( fVal [i] < yMin )
                yMin = fVal [i];
        else
        if ( fVal [i] > yMax )
                yMax = fVal [i];

float   yStep = pow (10,floor(log(yMax-yMin) / log(10.0)));

yMin  = yStep * floor ( yMin / yStep );
yMax  = yStep * ceil  ( yMax / yStep );

int     x0 = 60;
int     x1 = getmaxx () - 20;
int     y0 = 10;
int     y1 = getmaxy () - 40;

line ( x0, y0, x1, y0 );
line ( x1, y0, x1, y1 );
line ( x1, y1, x0, y1 );
line ( x0, y1, x0, y0 );

float   kx = ( x1 - x0 ) / ( xMax - xMin );
float   ky = ( y1 - y0 ) / ( yMax - yMin );
float   x  = a;
float   h  = ( b - a ) / 100.0;

moveto ( x0 + (x - xMin) * kx, y0 + (yMax - fVal [0]) * ky );

for ( i = 1; i < 100; i++, x += h )
        lineto ( x0 + (x-xMin)*kx, y0 + (yMax-fVal [i])*ky );

char    str [128];

settextstyle ( SMALL_FONT, HORIZ_DIR, 1 );

for ( x = xMin; x <= xMax; x += xStep )
{
        int         ix = x0 + ( x - xMin ) * kx;

        line    ( ix, y1, ix, y1 + 10 );

        if ( x + xStep <= xMax )
                for ( i = 1; i < 10; i++ )
                        line ( ix + i*xStep*kx*0.1, y1, ix +
                i*xStep*kx*0.1, y1 + 5 );

        sprintf ( str, "%g", x );
        outtextxy ( ix - textwidth (str) / 2, y1 + 15, str );
}

for ( float y = yMin; y <= yMax; y += yStep )
{
```

```
                int         iy = y0 + ( yMax - y ) * ky;
                line    ( x0 - 10, iy, x0, iy );
                if ( y + yStep <= yMax )
                        for ( i = 1; i < 10; i++ )
                                line ( x0 - 5, iy - i*yStep*ky*0.1,
                                        x0, iy - i*yStep*ky*0.1 );
                sprintf ( str, "%g", y );
                outtextxy ( x0 - 10 - textwidth ( str ), iy, str );
        }
        delete fVal;
}
```

Построение графика функции двух переменных будет подробно рассмотрено в гл. 10.

Здесь мы приведем лишь процедуру построения изолиний функции двух переменных, т. е. семейства линий, на которых эта функция сохраняет постоянные значения.

Для этого вся область изменения аргументов разбивается на набор прямоугольников, каждый из которых затем делится диагональю на два треугольника (рис. 3.4).

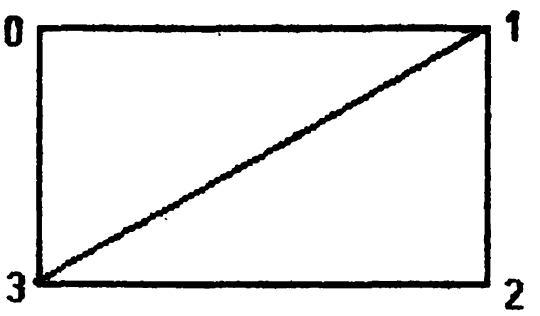


Рис. 3.4

В каждом треугольнике заданная функция приближается линейной функцией, график которой проходит через соответствующие вершины, при этом участок изолинии, лежащий в данном треугольнике, является отрезком прямой, соединяющим точки пересечения сторон треугольника с плоскостью $z = const$. Приведенная ниже процедура для такого треугольника строит все содержащиеся в нем изолинии.

```
// File example7.cpp
struct Point
{
        int         x;
        int         y;
};
void   plotIsolines ( float x0, float y0, float x1,
                        float y1, float (*f)( float, float ),
                        int n1, int n2, int nLines )
{
        float * fVal = new float [n1*n2];
        float       hx   = ( x1 - x0 ) / n1;
        float       hy   = ( y1 - y0 ) / n2;
        int         xStep = getmaxx () / n1;
        int         yStep = getmaxy () / n2;
        moveto ( 0, 0 );
        lineto ( ( n1 - 1 ) * xStep, 0 );
        lineto ( ( n1 - 1 ) * xStep, ( n2 - 1 ) * yStep );
        lineto ( 0, ( n2 - 1 ) * yStep );
        lineto ( 0, 0 );
        for ( int i = 0; i < n1; i++ )
```

```
   for ( int j = 0; j < n2; j++ )
            fVal [i + j*n1] = f ( x0 + i*hx, y0 + j*hy );

float       zMin = fVal [0];
float       zMax = fVal [0];

for ( i = 0; i < n1 * n2; i++ )
   if ( fVal [i] < zMin )
            zMin = fVal [i];
            else
            if ( fVal [i] > zMax )
               zMax = fVal [i];

float       dz = ( zMax - zMin ) / nLines;
Point       p [3];

for ( i = 0; i < n1 - 1; i++ )
   for ( int j = 0; j < n2 - 1; j++ )
   {
            int         k = i + j*n1;
            int         x = i * xStep;
            int         y = j * yStep;
            float       t;

            for ( float z = zMin; z <= zMax; z += dz )
            {
               int      count = 0;

               // edge 0-1
if ( fVal [k] != fVal [k+1] )
{
            t = (z - fVal [k])/(fVal [k+1] - fVal [k]);
            if ( t >= 0 && t <= 1 )
            {
               p [count  ].x = (int)( x + t * xStep );
               p [count++].y = y;
            }
}
// edge 1-3
if ( fVal [k+1] != fVal [k+n1] )
{
            t = (z-fVal[k+n1])/(fVal[k+1]-fVal[k+n1]);
            if ( t >= 0 && t <= 1 )
            {
               p [count  ].x = (int)( x + t * xStep );
               p [count++].y = (int)(y+(1-t)*yStep);
            }
}
// edge 3-0
if ( fVal [k] != fVal [k+n1] )
{
            t = (z - fVal[k+n1])/(fVal[k] - fVal[k+n1]);
            if ( t >= 0 && t <= 1 )
```

```
                {
                    p [count  ].x = x;
                    p [count++].y = y + ( 1 - t ) * yStep;
                }
        }
        if ( count > 0 )
                line ( p [0].x, p [0].y, p [1].x, p [1].y );
        if ( count > 2 ) // line through vertex
                line ( p [1].x, p [1].y, p [2].x, p [2].y );
        count = 0;
        // edge 1-2
        if ( fVal [k+1] != fVal [k+n1+1] )
        {
                t = (z-fVal[k+1])/(fVal[k+n1+1]-fVal[k+1]);
                if ( t >= 0 && t <= 1 )
                {
                    p [count  ].x = x + xStep;
                    p [count++].y = y + t * yStep;
                }
        }
        // edge 2-3
        if ( fVal [k+n1] != fVal [k+n1+1] )
        {
                t = (z-fVal[k+n1])/(fVal[k+n1+1]-fVal[k+n1]);
                if ( t >= 0 && t <= 1 )
                {
                    p [count  ].x = x + t * xStep;
                    p [count++].y = y + yStep;
                }
        }
        // edge 3-1
        if ( fVal [k+1] != fVal [k+n1] )
        {
                t = (z-fVal[k+n1])/(fVal[k+1]-fVal[k+n1]);
                if ( t >= 0 && t <= 1 )
                {
                    p [count  ].x = (int)( x + t*xStep );
                    p [count++].y = (int)( y + (1-t)*yStep );
                }
        }
        if ( count > 0 )
                line ( p [0].x, p [0].y, p [1].x, p [1].y );
        if ( count > 2 )      // line through vertex
                line ( p [1].x, p [1].y, p [2].x, p [2].y );
        }
    }
    delete fVal;
}
```

Упражнения

Напишите процедуру построения графика функции, заданной в полярных координатах

$$\rho = \rho(\varphi), \alpha \le \varphi \le \beta$$

Напишите процедуру построения графика функции, заданной параметрически

$$x = x(t), y = y(t), a \le t \le b$$

Постройте палитру (256 цветов), соответствующую цветам радуги (красному, оранжевому, желтому, зеленому, голубому, синему, фиолетовому) с плавными переходами между основными цветами.

Напишите процедуру построения по заданному набору чисел диаграммы соотношения в виде прямоугольников различной высоты.

Напишите процедуру построения по заданному набору чисел круговой диаграммы.

Реализуйте игру "Жизнь" Конвея: Имеется прямоугольная решетка из ячеек, в каждой из которых может находиться живая клетка. Каждая ячейка имеет 8 соседних с ней ячеек. Закон перехода на следующий временной шаг выглядят таким образом:
каждая клетка, у которой две или три соседних, выживает и переходит в следующее поколение;
клетка, у которой больше трех соседей, погибает от "перенаселенности" Клетка, у которой меньше двух соседей, умирает от одиночества;
если у пустой ячейки ровно три соседних, то в ней зарождается жизнь, т. е. появляется живая клетка.

Необходимо реализовать переход от одного поколения к следующему, при этом для определения цвета, которым выводится клетка, желательно использовать ее возраст. В качестве палитры можно брать либо оттенки серого цвета, либо палитру из цветов радуги. Ниже приводятся наиболее интересные комбинации (рис. 3.5).



Рис. 3.5

Глава 4
РАБОТА С ОСНОВНЫМИ ГРАФИЧЕСКИМИ УСТРОЙСТВАМИ

Несмотря на наличие различных графических библиотек (например, в составе компилятора Borland C++), часто возникает необходимость прямой работы с тем или иным графическим устройством. Это может быть связано как с тем, что библиотека не поддерживает соответствующее устройство (например, мышь или принтер), так и с тем, что работа с данным устройством организована недостаточно эффективно и всех его возможностей не использует.

Рассмотрим основные приемы работы с некоторыми устройствами.

4.1. Клавиатура

Для начала мы рассмотрим работу с клавиатурой. Хотя она и не является графическим устройством, правильная работа с ней необходима при написании большинства игровых программ.

При нажатии или отпускании клавиши генерируется прерывание 9 и при этом в младших 7 битах значения, прочитанного из порта 60h, содержится номер нажатой клавиши (ее scan-код), а в старшем бите - 0, если клавиша была нажата, и 1, если клавиша была отпущена.

Стандартный обработчик прерывания 9 читает номер нажатой клавиши, переводит его в ASCII-код и помещает номер и ASCII-код в стандартный буфер клавиатуры.

В ряде игровых программ требуется знать, какие клавиши нажаты в данный момент, при этом стандартный обработчик прерывания 9 не годится, так как он не в состоянии определить все возможные комбинации нажатых клавиш (например, Ctrl и "стрелка вниз").

Ниже приводится пример класса Keyboard, который позволяет для любой клавиши определить, нажата ли она или отпущена. Для этого заводится массив из 128 байт, где каждый байт соответствует определенной клавише.

```
// File Keyboard.h
#ifndef          __KEYBOARD__
#define          __KEYBOARD__

class Keyboard
{
        static void interrupt (*oldKbdHandler)(...);
        static int          keys [128];
        static int              scanCode [256];
        static int              charCode [128];
        static void interrupt   newKbdHandler (...);

public:
        Keyboard ();
        ~Keyboard ();
```

```
        int         isPressed ( int key )
        {
                    return keys [getScanCode ( key )];
        }
        int         getChar ( int scanCode )
        {
                    return charCode [scanCode & 0x7F];
        }
        int         getScanCode ( int key );
};
#endif
```

```
// File keyboard.cpp
#include        <dos.h>
#include        "keyboard.h"
void interrupt (*Keyboard::oldKbdHandler)(...);
int Keyboard::keys [128];
int Keyboard :: scanCode [256] =
{
        0,  0,  0,  0,  0,  0,  0, 15,
        14, 0,  0,  0,  0, 28,  0,  0,
        0,  0,  0,  0,  0,  0,  0,  0,
        0,  0,  0,  1,  0,  0,  0,  0,
        57, 0,  0,  0,  0,  0,  0, 40,
        0,  0, 55, 78, 51, 12, 52, 53,
        11, 2,  3,  4,  5,  6,  7,  8,
        9,  10, 0,  39, 0, 13,  0,  0,
        0,  30, 48, 46, 32, 18, 33, 34,
        35, 23, 36, 37, 38, 50, 49, 24,
        25, 16, 19, 31, 20, 22, 47, 17,
        45, 21, 44, 26, 43, 27, 0,  0,
        41, 30, 48, 46, 32, 18, 33, 34,
        35, 23, 36, 37, 38, 50, 49, 24,
        25, 16, 19, 31, 20, 22, 47, 17,
        45, 21, 44,  0,  0,  0,  0,  0,
        0,  0,  0,  0,  0,  0,  0,  0,
        0,  0,  0,  0,  0,  0,  0,  0,
        0,  0,  0,  0,  0,  0,  0,  0,
        0,  0,  0,  0,  0,  0,  0,  0,
        0,  0,  0,  0,  0,  0,  0,  0,
        0,  0,  0,  0,  0,  0,  0,  0,
        0,  0,  0,  0,  0,  0,  0,  0,
        56,  0,  0, 59, 60, 61, 62, 63,
        64, 65, 66, 67, 68, 87, 88, 71,
        72, 73, 74, 75,  0, 77, 78, 79,
        80, 81, 82, 83
};
int     Keyboard    charCode [128] =
```

```
{
        0,  27,  49,  50,  51,  52, 53,  54,
       55,  56,  57,  48,  45,  61,  8,   9,
      113, 119, 101, 114, 116, 121,117, 105,
      111, 112,  91,  93,  13,   0, 97, 115,
      100, 102, 103, 104, 106, 107,108,  59,
       39,  96,   0,  92, 122, 120, 99, 118,
       98, 110, 109,  44,  46,  47, 0,  42,
        0,  32,   0,   0,   0,   0,  0,   0,
        0,   0,   0,   0,   0,   0,  0,   0,
        0,   0,  45,   0,   0,   0, 43,   0,
        0,   0,   0,   0,   0,   0,  0,   0,
        0,   0,   0,   0,   0,   0,  0,   0,
        0,   0,   0,   0,   0,   0,  0,   0,
        0,   0,   0,   0,   0,   0,  0,   0,
        0,   0,   0,   0,   0,   0,  0,   0,
        0,   0,   0,   0,   0,   0,  0,   0
};
Keyboard    Keyboard ()
{
        oldKbdHandler = getvect ( 9 ); // save old keyboard ISR
        setvect ( 9, newKbdHandler );
}

Keyboard :: ~Keyboard ()
{
        setvect ( 9, oldKbdHandler );
}

void   interrupt Keyboard    newKbdHandler (...)
{
        char        scanCode  = inportb ( 0x60 );
        char        otherCode = inportb ( 0x61 );

        outportb ( 0x61, otherCode | 0x80 );
        outportb ( 0x61, otherCode );
        outportb ( 0x20, 0x20 );

        keys [scanCode & 0x7F] = ( scanCode & 0x80 == 0 );
}

int     Keyboard :: getScanCode ( int key )
{
        if ( key < 0x0100 )
                return scanCode [key];
        return key >> 8;
}
```

4.2. Мышь

Наиболее распространенным устройством ввода графической информ в ПЭВМ является мышь. При перемещении мыши и/или нажатии/отпускании пок мышь передает информацию в компьютер о своих параметрах (величине мещения и статусе кнопок).

Первый манипулятор типа мыши был разработан Дугом Энгельбартом в 1963 г. Это было аналоговое устройство, где металлические колесики были связаны с переменными резисторами. В начале 70-х гг. в исследовательском центре Palo Alto Research Center корпорации Xerox был разработан манипулятор мышь, напоминающий современные аналоги, и он был использован в компьютерной системе Alto. В 1982 г. появился манипулятор Microsoft Mouse. В начале 1983 г. Apple Computer Corporation выпустила Apple Lisa - первый компьютер с графическим пользовательским интерфейсом, в состав которого входила мышь.

Механическая и оптомеханическая мыши содержат резиновый шарик, вращение которого при перемещении мыши передается на специальные валики. Для считывания вращения этих валиков в механической мыши используется диэлектрический диск с нанесенными на него радиально расположенными контактами. При передвижении мыши происходит замыкание и размыкание контактов, генерирующие импульсы, количество которых пропорционально величине смещения мыши (рис. 4.1).

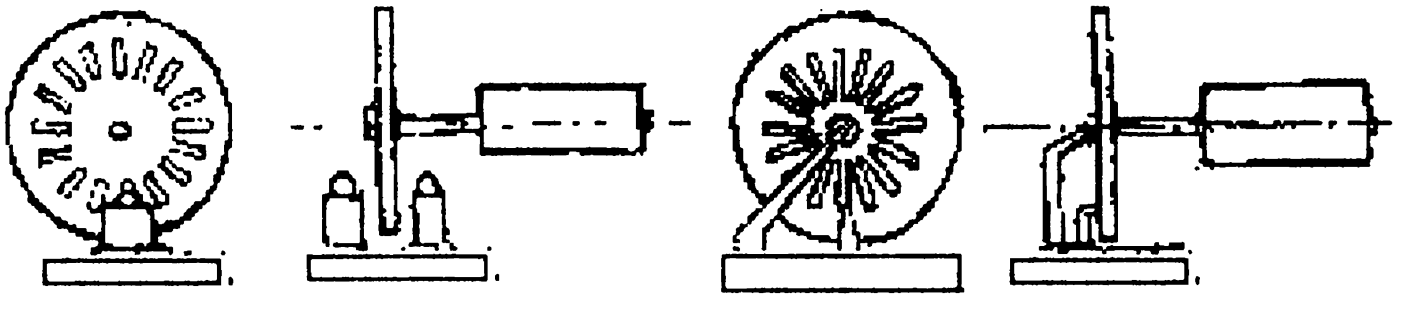


Рис. 4.1

В оптомеханической мыши используется диск с отверстиями и пара светодиод-фотодиод для определения перемещения мыши (рис. 4.2).

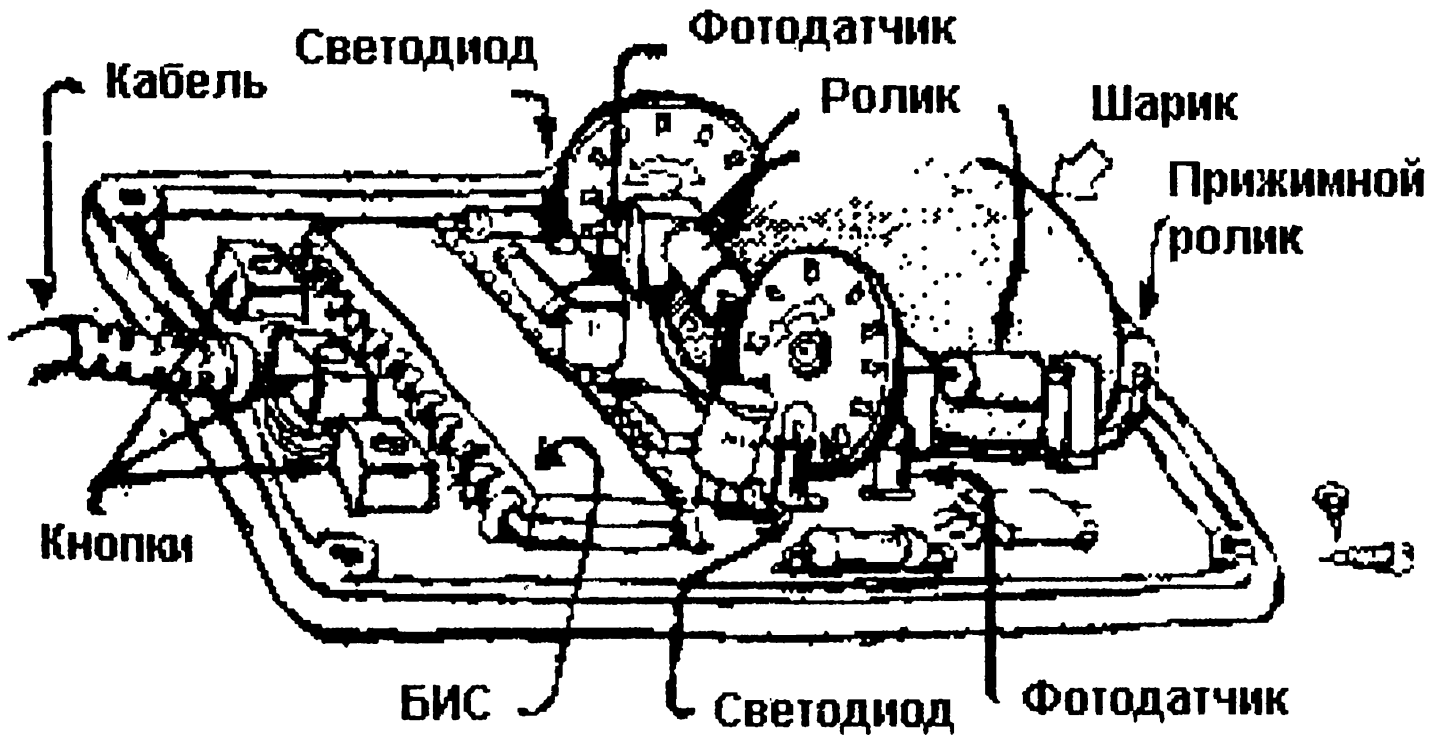


Рис. 4.2

Оптическая мышь никаких движущихся деталей не использует. Она передвигается по специальной подложке, покрытой тонкой сеткой отражающих свет линий.

При этом линии одного направления имеют один цвет (обычно синий), а линии другого направления - другой (обычно черный). Когда мышь перемещают по подложке, свет, излучаемый двумя светодиодами заданных цветов, отражается от соответствующих линий и попадает в фотодиоды, передавая тем самым информацию о перемещении мыши (рис. 4.3).

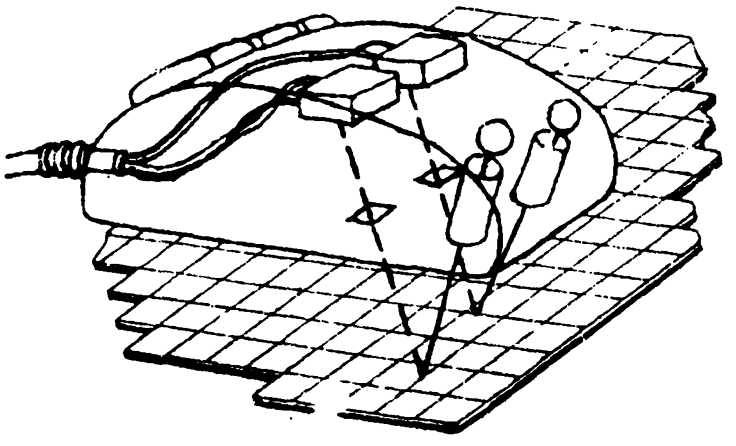


Рис. 4.3

Для достижения некоторой унификации каждая мышь поставляется обычно вместе со своим драйвером - специальной программой, понимающей данный конкретный тип мыши и предоставляющей некоторый (почти универсальный) интерфейс прикладным программам. При этом вся работа с мышью происходит через драйвер, который отслеживает перемещения мыши, нажатие и отпускание кнопок мыши и обеспечивает работу с курсором мыши - специальным маркером на экране (обычно в виде стрелки), дублирующим все передвижения мыши и дающим возможность пользователю указывать мышью на те или иные объекты на экране.

Работа с мышью реализуется через механизм прерываний. Прикладная программа осуществляет прерывание 33h, передавая в регистрах необходимые параметры, и в регистрах же получает значения, возвращенные драйвером мыши.

Приводим набор функций для работы с мышью в соответствии со стандартом фирмы Microsoft (ниже приведены используемые файлы Mouse.h и Mouse.cpp).

```
// File Mouse.h
#ifndef         __MOUSE__
#define         __MOUSE__

#include        "point.h"
#include        "rect.h"
                                        // mouse event flags
#define         MOUSE_MOVE_MASK                 0x01
#define         MOUSE_LBUTTON_PRESS  0x02
#define         MOUSE_LBUTTON_RELEASE           0x04
#define         MOUSE_RBUTTON_PRESS 0x08
#define         MOUSE_RBUTTON_RELEASE           0x10
#define         MOUSE_MBUTTON_PRESS 0x20
#define         MOUSE_MBUTTON_RELEASE           0x40
#define         MOUSE_ALL_EVENTS        0x7F
                                        // button flags
#define         MK_LEFT                         0x0001
#define         MK_RIGHT                        0x0002
#define         MK_MIDDLE                       0x0004

struct MouseState
{
        Point,,,    loc;
        int         buttons;
};

class CursorShape
```

```
{
public:
        unsigned short andMask [16];
        unsigned short xorMask [16];
        Point           hotSpot;

        CursorShape ( unsigned short *, unsigned short *,
                      const Point& );
};

typedef         void        (cdecl far * MouseHandler)( int, int, int, int );

int     resetMouse              ();
void    showMouseCursor         ();
void    hideMouseCursor         ();
void    hideMouseCursor         ( const Rect& );
void    readMouseState          ( MouseState& );
void    moveMouseCursor         ( const Point );
void    setMouseHorzRange       ( int, int );
void    setmouseVertRange       ( int, int );
void    setMouseShape           ( const CursorShape& );
void    setMouseHandler         ( MouseHandler, int = MOUSE_ALL_EVENTS );
void    removeMouseHandler      ();

#endif
```

```
// File Mouse.cpp
#include        <alloc.h>
#include        "Mouse.h"
#pragma inline
static MouseHandler     curHandler = NULL;
CursorShape   CursorShape ( unsigned short * aMask,
                            unsigned short * xMask,
                            const Point& p )  hotSpot ( p )
{
        for ( register int i = 0; i < 16; i++ )
        {
                andMask [i] = aMask [i];
                xorMask [i] = xMask [i];
        }
}
////////////////////////////////////////////////////////////////
int     resetMouse ()
{
        asm {
                xor         ax, ax
                int         33h
        }
        return _AX == 0xFFFF;
}
void    showMouseCursor ()
```

```
{
        asm {
                mov     ax, 1
                int     33h
        }
}
void   hideMouseCursor ()
{
        asm {
                mov     ax, 2
                int     33h
        }
}
void   readMouseState ( MouseState& s )
{
        asm {
                mov     ax, 3
                int     33h
        }
#if defined(__COMPACT__) || defined(__LARGE__) || defined(__HUGE__)
        asm {
                push    es
                push    di
                les     di, dword ptr s
                mov     es:[di ], cx
                mov     es:[di+2], dx
                mov     es:[di+4], bx
                pop     di
                pop     es
        }
#else
        asm {
                push    di
                mov     di, word ptr s
                mov     [di ], cx
                mov     [di+2], dx
                mov     [di+4], bx
                pop     di
        }
#endif
}
void   moveMouseCursor ( const Point p )
{
        asm {
                mov     ax, 4
                mov     cx, p.x
                mov     dx, p.y
                int     33h
        }
}
```

```
void   setHorzMouseRange ( int xmin, int xmax )
{
        asm {
                        mov         ax, 7
                        mov         cx, xmin
                        mov         dx, xmax
                        int         33h
        }
}
void   setVertMouseRange ( int ymin, int ymax )
{
        asm {
                        mov         ax, 8
                        mov         cx, ymin
                        mov         dx, ymax
                        int         33h
        }
}
void   setMouseShape ( const CursorShape& c )
{
#if defined(__COMPACT__) || defined(__LARGE__) || defined(__HUGE__)
        asm {
                        push        es
                        push        di
                        les         di, dword ptr c
                        mov         bx, es:[di+64]
                        mov         cx, es:[di+66]
                        mov         dx, di
                        mov         ax, 9
                        int         33h
                        pop         di
                        pop         es
        }
#else
        asm {
                        push        di
                        mov         di, word ptr c
                        mov         bx, [di+64]
                        mov         cx, [di+66]
                        mov         dx, di
                        mov         ax, 9
                        int         33h
                        pop         di
        }
#endif
}
void    hideMouseCursor ( const Rect& r )
{
#if defined(__COMPACT__) || defined(__LARGE__) || defined(__HUGE__)
        asm {
                        push        es
```

```
                push    si
                push    di
                les     di, dword ptr r
                mov     ax, 10h
                mov     cx, es:[di]
                mov     dx, es:[di+2]
                mov     si, es:[di+4]
                mov     di, es:[di+6]
                int     33h
                pop     di
                pop     si
                pop     es
        }
#else
        asm {
                push    si
                push    di
                mov     di, word ptr r
                mov     ax, 10h
                mov     cx, [di]
                mov     dx, [di+2]
                mov     si, [di+4]
                mov     di, [di+6]
                int     33h
                pop     di
                pop     si
        }
#endif
}
static void far mouseStub ()
{
        asm {
                push    ds          // preserve ds
                push    ax          // preserve ax
                mov     ax, seg curHandler
                mov     ds, ax
                pop     ax          // restore ax
                push    dx          // y
                push    cx          // x
                push    bx          // button state
                push    ax          // event mask
                call    curHandler
                add     sp, 8       // clear stack
                pop     ds
        }
}
void   setMouseHandler ( MouseHandler h, int mask )
{
        void        far * p = mouseStub;
        curHandler = h;
```

```
        asm {
                push    es
                mov     ax, 0Ch
                mov     cx, mask
                les     dx, p
                int     33h
                pop     es
        }
}
void  removeMouseHandler ()
{
        curHandler = NULL;
        asm {
                mov     ax, 0Ch
                mov     cx, 0
                int     33h
        }
}
```

4.2.1. Инициализация и проверка наличия мыши

Функция resetMouse производит инициализацию мыши и возвращает ненулевое значение, если мышь обнаружена.

4.2.2. Высветить на экране курсор мыши

Функция showMouseCursor выводит на экран курсор мыши. При этом курсор перемещается синхронно с перемещениями самой мыши.

4.2.3. Убрать (сделать невидимым) курсор мыши

Функция hideMouseCursor убирает курсор мыши с экрана. Однако при этом драйвер мыши продолжает отслеживать ее перемещения, причем к этой функции возможны вложенные вызовы. Каждый вызов функции hideMouseCursor уменьшает значение внутреннего счетчика драйвера на единицу, каждый вызов функции showMouseCursor увеличивает значение счетчика на единицу. Курсор мыши виден только тогда, когда значение счетчика равно нулю (изначальное значение счетчика равно -1).

При работе с мышью следует иметь в виду, что выводить изображение поверх курсора мыши нельзя. Поэтому, если нужно произвести вывод на экран в том месте, где может находиться курсор мыши, следует убрать его с экрана, выполнить требуемый вывод и затем снова вывести курсор мыши на экран.

4.2.4. Прочесть состояние мыши (ее координаты и состояние кнопок)

Функция readMouseState возвращает состояние мыши в полях структуры MouseState. Поля *x* и *y* содержат текущие координаты курсора в пикселах, поле buttons определяет, какие кнопки нажаты. Установленный бит 0 соответствует нажа-

той левой кнопке, бит 1 - правой, и бит 2 - средней (этим значениям соответствуют константы MK_LEFT, MK_RIGHT и MK_MIDDLE).

4.2.5. Передвинуть курсор мыши в точку с заданными координатами

Функция moveMouseCursor служит для установки курсора мыши в точку с заданными координатами.

4.2.6. Установка области перемещения курсора

При необходимости можно ограничить область перемещения мыши по экрану. Для задания области возможного перемещения курсора по горизонтали служит функция setHorzMouseRange, для задания области перемещения по вертикали - функция setVertMouseRange.

4.2.7. Задание формы курсора

В графических режимах высокого разрешения (640 на 350 пикселов и выше) курсор задается двумя масками 16 на 16 бит и смещением координат курсора от верхнего левого угла масок. Каждую из масок можно трактовать как изображение, составленное из пикселов белого (соответствующий бит равен единице) и черного (соответствующий бит равен нулю) цветов. При выводе курсора на экран сначала на содержимое экрана накладывается (с использованием операции AND_PUT) первая маска, называемая иногда AND-маской, а затем на то же самое место накладывается вторая маска (с использованием операции XOR_PUT). Все необходимые параметры для задания курсора мыши содержатся в полях структуры CursorShape. Устанавливается форма курсора при помощи функции setMouseShape.

4.2.8. Установка области гашения

Иногда желательно задать на экране область, при попадании в которую курсор мыши автоматически гасится. Для этого используется функция setHideRange. Но при выходе курсора из области гашения он не восстанавливается. Поэтому для восстановления нормальной работы курсора необходимо вызвать функцию showMouseCursor, независимо от того, попал ли курсор в область гашения или нет.

4.2.9. Установка обработчика событий

Вместо того чтобы все время опрашивать драйвер мыши, можно передать драйверу адрес функции, которую нужно вызывать при наступлении заданных событий. Для установки этой функции следует воспользоваться функцией setMouseHandler, где в качестве первого параметра выступает указатель на функцию, а второй параметр задает события, при наступлении которых переданную функцию следует вызвать. События задаются посредством битовой маски. Возможные события определяются при помощи символических констант MOUSE_MOVE_MASK, MOUSE_LBUTTON_PRESS и др. Требуемые условия соединяются побитовой операцией ИЛИ. Передаваемая функция получает 4 параметра - маску события, повлекшего за собой вызов функции, маску состояния кнопок мыши и текущие коор-

динаты курсора. По окончании работы программы необходимо обязательно убрать обработчик событий (при помощи функции removeMouseHandler).

Ниже приводится пример простейшей программы, устанавливающей обработчик событий мыши на нажатие правой кнопки мыши и задающей свою форму курсора.

```
// File moustest.cpp
#include      <bios.h>
#include      <conio.h>
#include      "mouse.h"

unsigned short andMask [] =
{
        0x0FFF, 0x07FF, 0x01FF, 0x007F,
        0x801F, 0xC007, 0xC001, 0xE000,
        0xE0FF, 0xF0FF, 0xF0FF, 0xF8FF,
        0xF8FF, 0xFCFF, 0xFCFF, 0xFEFF
};

unsigned short xorMask [] =
{
        0x0000, 0x6000, 0x7800, 0x3E00,
        0x3F80, 0x1FE0, 0x1FF8, 0x0FFE,
        0x0F00, 0x0700, 0x0700, 0x0300,
        0x0300, 0x0100, 0x0100, 0x0000,
};

CursorShape    cursor ( andMask, xorMask, Point ( 1, 1 ) );
int    doneFlag = 0;

void    setVideoMode ( int mode )
{
        __asm {
                mov     ax, word ptr mode
                int     10h
        }
}

#ifdef __WATCOMC__
#pragma off (check_stack)
#pragma off (unreferenced)
#endif
void    cdecl far waitPress ( int mask, int button, int x, int y )
{
        if ( mask & MOUSE_RBUTTON_PRESS )
                doneFlag = 1;
}
#ifdef __WATCOMC__
#pragma on (check_stack)
#pragma on (unreferenced)
#endif

main ()
{
        Point   p ( 0, 0 );
```

```
        setVideoMode    ( 0x12 );
        resetMouse      ();
        showMouseCursor ();
        setMouseShape   ( cursor );
        setMouseHandler ( waitPress );
        moveMouseCursor ( p );
        while ( !doneFlag )
                ;
        hideMouseCursor ();
        removeMouseHandler ();
        setVideoMode ( 3 );
}
```

Если вы хотите работать с мышью в защищенном режиме DPMI32, то это приводит к некоторым осложнениям. Обычно DPMI-сервер предоставляет интерфейс драйверу мыши, но использование 32-битового защищенного режима вносит свои сложности. Ниже приводится файл, реализующий описанные выше функции для защищенного режима компилятора Watcom.

```
// File mouse32.cpp
//
// This file provides mouse interface for DPMI32 for Watcom compiler
//
#include      <stdio.h>
#include      <dos.h>
#include      <i86.h>
#include      "mouse.h"
static  MouseHandler    curHandler = NULL;
CursorShape :: CursorShape ( unsigned short * aMask,
    unsigned short * xMask,
                                        const Point& p ) : hotSpot ( p )
{
        for ( register int i = 0; i < 16; i++ )
        {
                andMask [i] = aMask [i];
                xorMask [i] = xMask [i];
        }
}
////////////////////////////////////////////////////////////////
int     resetMouse ()
{
        union REGS inRegs, outRegs;
        inRegs.w.ax = 0;
        int386 ( 0x33, &inRegs, &outRegs );
        return outRegs.w.ax == 0xFFFF;
}
void    showMouseCursor ()
{
        union REGS inRegs, outRegs;
```

```
    inRegs.w.ax = 1;
    int386 ( 0x33, &inRegs, &outRegs );
}
void   hideMouseCursor ()
{
    union REGS inRegs, outRegs;
    inRegs.w.ax = 2;
    int386 ( 0x33, &inRegs, &outRegs );
}
void   readMouseState ( MouseState& s )
{
    union REGS inRegs, outRegs;
     inRegs.w.ax = 3;
     int386 ( 0x33, &inRegs, &outRegs );
     s.loc.x   = outRegs.w.cx;
     s.loc.y   = outRegs.w.dx;
     s.buttons = outRegs.w.bx;
}
void   moveMouseCursor ( const Point p )
{
     union REGS inRegs, outRegs;
     inRegs.w.ax = 4;
     inRegs.w.cx = p.x;
     inRegs.w.dx = p.y;
     int386 ( 0x33, &inRegs, &outRegs );
}
void   setHorzMouseRange ( int xmin, int xmax )
{
     union REGS inRegs, outRegs;
     inRegs.w.ax = 7;
     inRegs.w.cx = xmin;
     inRegs.w.dx = xmax;

     int386 ( 0x33, &inRegs, &outRegs );
}
void   setVertMouseRange ( int ymin, int ymax )
{
     union REGS inRegs, outRegs;
     inRegs.w.ax = 8;
     inRegs.w.cx = ymin;
     inRegs.w.dx = ymax;
     int386 ( 0x33, &inRegs, &outRegs );
}
void   setMouseShape ( const CursorShape& c )
```

```
{
        union  REGS  inRegs, outRegs;
        struct SREGS segRegs;

        segread ( &segRegs );

        inRegs.w.ax  = 9;
        inRegs.w.bx  = c.hotSpot.x;
        inRegs.w.cx  = c.hotSpot.y;
        inRegs.x.edx = FP_OFF ( c.andMask );
        segRegs.es   = FP_SEG ( c.andMask );

        int386x ( 0x33, &inRegs, &outRegs, &segRegs );
}
void    hideMouseCursor ( const Rect& r )
{
        union REGS  inRegs, outRegs;

        inRegs.w.ax = 0x10;
        inRegs.w.bx = r.x1;
        inRegs.w.cx = r.y1;
        inRegs.w.si = r.x2;
        inRegs.w.di = r.y2;

        int386 ( 0x33, &inRegs, &outRegs );
}
#pragma off (check_stack)
static void _loadds far mouseStub ( int mask, int btn, int x, int y )
{
#pragma aux mouseStub parm [EAX] [EBX] [ECX] [EDX]
        (*curHandler)(mask, btn, x, y );
}
#pragma on (check_stack)
void    setMouseHandler ( MouseHandler h, int mask )
{
        curHandler = h;

        union  REGS  inRegs, outRegs;
        struct SREGS segRegs;

        segread ( &segRegs );

        inRegs.w.ax  = 0x0C;
        inRegs.w.cx  = (short) mask;
        inRegs.x.edx = FP_OFF ( mouseStub );
        segRegs.es   = FP_SEG ( mouseStub );

        int386x ( 0x33, &inRegs, &outRegs, &segRegs );
}
void    removeMouseHandler ()
{
        union REGS  inRegs, outRegs;

        inRegs.w.ax  = 0x0C;
        inRegs.w.cx  = 0;
```

```
    int386 ( 0x33, &inRegs, &outRegs );
    curHandler  = NULL;
}
```

4.3. Джойстик

Еще одним устройством, часто используемым для ввода графической информации (обычно в игровых программах), является джойстик.

Устроен джойстик крайне просто - его ручка связана с двумя потенциометрами (при отклонении ручки сопротивление потенциометров изменяется). Каждый потенциометр соединен с конденсатором, при этом время зарядки конденсатора обратно пропорционально сопротивлению потенциометра. Как только заряд конденсатора достигает заданной пороговой величины, в порту джойстика выставляется соответствующий бит.

Обычно с джойстиком связан порт 201h, который в состоянии поддерживать два джойстика одновременно. Рассмотрим роль каждого бита.

Бит	*Его значение*	*Бит*	*Его значение*
0	Джойстик А ось X	4	Джойстик А кнопка 1
1	Джойстик А ось Y	5	Джойстик А кнопка 2
2	Джойстик В ось X	6	Джойстик В кнопка 1
3	Джойстик В ось Y	7	Джойстик В кнопка 2

Биты 0-3 служат для определения координат джойстика, биты 4-7 - для чтения состояния кнопок.

При нажатой кнопке соответствующий бит принимает значение 0, при отпущенной - значение 1. Для чтения позиции джойстика в его порт записывают 0 и находят время до выставления 1 в нужном разряде порта. Так как время обычно измеряют числом итераций цикла, то соответствующее значение будет различным для компьютеров с разным быстродействием. Поэтому программа перед первым использованием джойстика должна произвести его калибровку, т. е. определить, какие значения соответствуют крайним положениям рукоятки джойстика, и затем нормировать все снимаемые с джойстика значения.

Для проверки наличия джойстика в цикле опрашивается в течение определенного времени порт джойстика. Если при этом значения всех битов равны нулю, то можно с уверенностью считать, что джойстик к компьютеру не подключен.

Помимо непосредственного опрашивания порта и определения времени для установления координат джойстика можно использовать BIOS.

Ниже приводятся файлы для работы с джойстиком и пример его использования.

```
// File joystick.h
#ifndef     __JOYSTICK__
#define     __JOYSTICK__

#define     JOYPORT         0x201           // joystick port

#define     BUTTON_1_A      0x10    // joystick A, button 1
#define     BUTTON_2_A      0x20    // joystick A, button 2
```

```
#define         BUTTON_1_B      0x40    // joystick B, button 1
#define         BUTTON_2_B      0x80    // joystick B, button 2
#define         JOYSTICK_1_X    0x01    // joystick A, X axis
#define         JOYSTICK_1_Y    0x02    // joystick A, Y axis
#define         JOYSTICK_2_X    0x04    // joystick B, X axis
#define         JOYSTICK_2_Y    0x08    // joystick B, Y axis

                                        // global values
extern          unsigned joy1MinX, joy1MaxX, joy1MinY, joy1MaxY;
extern          unsigned joy2MinX, joy2MaxX, joy2MinY, joy2MaxY;

unsigned joystickButtons   ( char button );
unsigned joystickValue     ( char stick  );
unsigned joystickValueBIOS ( char stick  );
int      joystickPresent   ( char stick );
#endif
```



```
// File joystick.cpp
#include        <dos.h>
#include        "joystick.h"

unsigned joy1MinX, joy1MaxX, joy1MinY, joy1MaxY;
unsigned joy2MinX, joy2MaxX, joy2MinY, joy2MaxY;

unsigned joystickButtons ( char button )
{
        outportb ( JOYPORT, 0 );
        return ~inportb ( JOYPORT ) & button;
}
unsigned joystickValue   ( char stick  )
{
        asm {
                cli
                mov     ah, byte ptr stick
                xor     al, al
                xor     cx, cx
                mov     dx, JOYPORT
                out     dx, al
        }
discharge:
        asm {
                in      al, dx
                test    al, ah
                loopne  discharge

                sti
                xor     ax, ax
                sub     ax, cx
        }
}
unsigned joystickValueBIOS   ( char stick  )
{
        REGS    inregs, outregs;
```

```
        inregs.h.ah = 0x84;
        inregs.x.dx = 0x01;
        int86 ( 0x15, &inregs, &outregs );
        switch ( stick )
        {
                case JOYSTICK_1_X:
                        return outregs.x.ax;
                case JOYSTICK_1_Y:
                        return outregs.x.bx;
                case JOYSTICK_2_X:
                        return outregs.x.cx;
                case JOYSTICK_2_Y:
                        return outregs.x.dx;
        }
        return 0;
}
int     joystickPresent ( char stick )
{
        asm {
                mov     ah, 84h
                mov     dx, 1
                int     15h             // call BIOS to read
                                        // joystick values
        }
        if ( _AX == 0 && _BX == 0 && stick == 1 )
                return 0;
        if ( _CX == 0 && _DX == 0 && stick == 2 )
                return 0;
        return 1;
}

// File joytest.cpp
#include        <bios.h>
#include        <stdio.h>
#include        "joystick.h"
main ()
{
        if ( !joystickPresent ( 1 ) )
        {
                printf ( "\nJoystick 1 not found." );
                return 0;
        }
        else
                printf ( "\nJoystick 1 found." );
                                        // now calibrate joystick
        joy1MinX = 0xFFFF;
```

```
        joy1MaxX = 0;
        joy1MinY = 0xFFFF;
        joy1MaxY = 0;

        while ( ! joystickButtons ( BUTTON_1_A | BUTTON_1_B ) )
        {
                unsigned x = joystickValueBIOS ( JOYSTICK_1_X );
                unsigned y = joystickValueBIOS ( JOYSTICK_1_Y );

                if ( x < joy1MinX )
                        joy1MinX = x;

                if ( x > joy1MaxX )
                        joy1MaxX = x;

                if ( y < joy1MinY )
                        joy1MinY = y;

                if ( y > joy1MaxY )
                        joy1MaxY = y;
        }

        printf ( "\nCalibration complete.\nxMin = %u. xMax = %u. yMin = %u. yMax = %u",
                joy1MinX, joy1MaxX, joy1MinY, joy1MaxY );
}
```

4.4. Сканер

Сканер - это устройство для чтения картинки с листа бумаги, слайда и т.п. По устройству они обычно разделяются на ручные и планшетные. По типу снимаемого изображения - на цветные (читается цветное изображение) и черно-белые (читается изображение в градациях серого цвета). Многие сканеры поддерживают сканирование изображения с различными разрешениями. Обычно каждый сканер поставляется вместе со своим драйвером и программой для сканирования изображений.

4.5. Принтер

В качестве устройства для получения "твердой" копии изображения на экране обычно выступает принтер. Практически любой принтер позволяет осуществить построение изображения, так как сам выводит символы, построенные из точек (каждый символ представляется матрицей точек; для большинства матричных принтеров - матрицей размера 8 на 11).

Принтеры бывают матричными, струйными, термосублимационными и лазерными.

Принцип работы матричных принтеров напоминает работу обыкновенной пишущей машинки. Принтер состоит из механизма прогонки бумаги, красящей ленты и движущейся печатающей головки.

Головка содержит набор вертикально расположенных игл, удары которых переносят краситель с ленты на бумагу.

Струйный принтер также содержит печатающую головку, состоящую из набора микросопел, через которые на бумагу "выстреливаются" капельки жидких чернил. В современных струйных принтерах применяются две основные конструкции мик-

росопел. В одном случае "выстреливание" капельки чернил достигается путем использования пьезокристалла, преобразующего электрический сигнал в механическое перемещение, приводящее к "выстрелу" капли. Этот принцип применяется в принтерах фирмы Epson. В струйных принтерах фирмы Hewlett Packard используется другой принцип - в микросопле находится тонкопленочный резистор. При подаче на него электрического импульса он нагревает чернила, они начинают испаряться и за счет возникающего давления выбрасывается капелька чернил. Обе конструкции приведены на рис. 4.4.

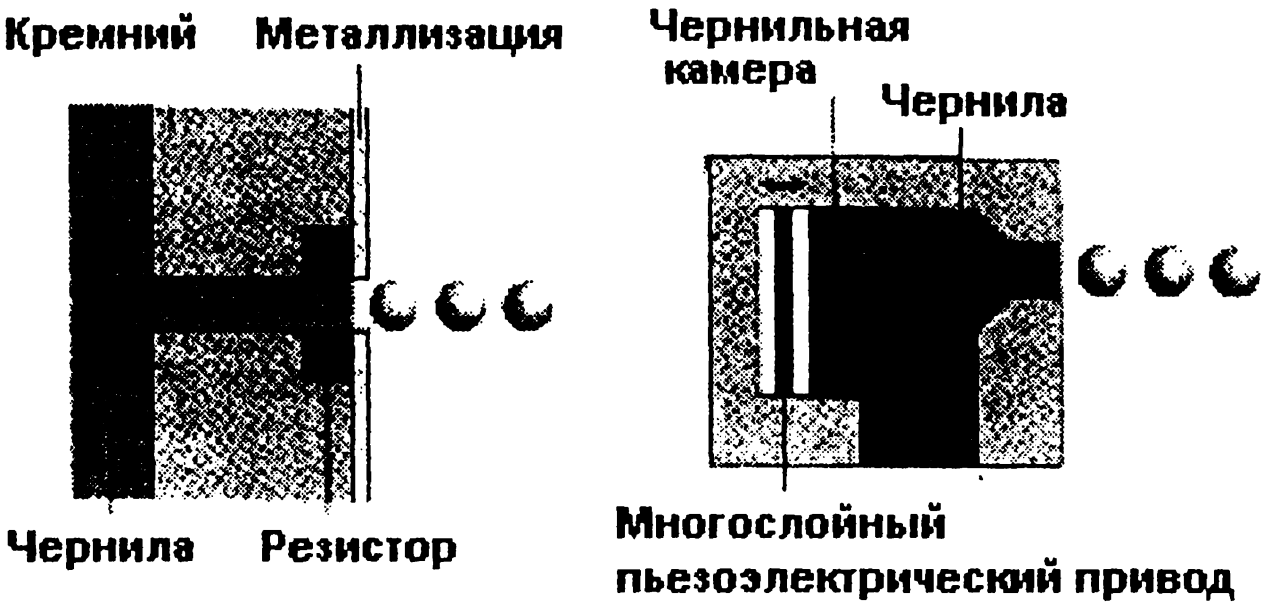


Рис. 4.4

Еще одной разновидностью принтеров являются термосублимационные. В них краситель находится на специальной пленке и перенос его на бумагу осуществляется специальной термической головкой (рис. 4.5).

Еще одним типом принтера, получившим широкое распространение, является лазерный.

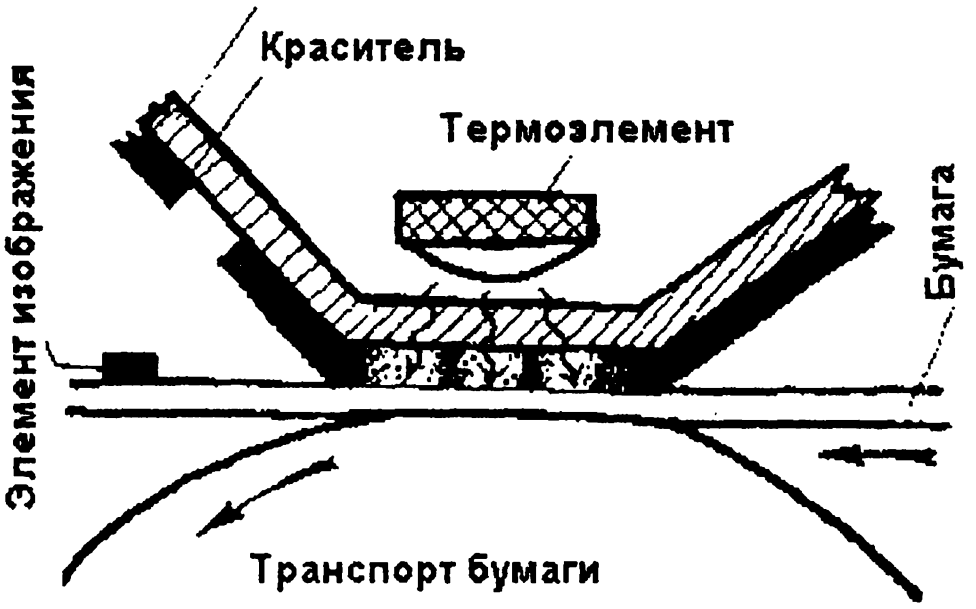


Рис. 4.5

Принцип работы лазерного принтера напоминает работу обычного ксерокопировального аппарата, только в принтере изображение строится лазерным лучом на специальном селеновом барабане.Попадание лазерного луча на этот барабан приводит к снятию статического электрического разряда в той точке, куда попал луч. Порошкообразный краситель (тонер) прилипает к заряженным местам и переносится на бумагу. Чтобы впоследствии краситель с бумаги не ссыпался, она нагревается, краситель плавится и прочно фиксируется на бумаге. Устройство типичного лазерного принтера приведено на рис. 4.6.

Для осуществления управления принтером существует специальный набор команд (обычно называемых Esc-последовательностями), позволяющий управлять режимом работы принтера, прогонкой бумаги на заданное расстояние и печатью гра-

фической информации. Каждая команда представляет собой некоторый набор символов (кодов), просто посылаемых для печати на принтер. Чтобы принтер мог отличить эти команды от обычного печатаемого текста, они, как правило, начинаются с символа с кодом меньше 32, т. е. с кода, которому не соответствует ни один ASCII-символ. Для большинства команд в качестве такового выступает символ Escape (код 27). Совокупность подобных команд образует язык управления принтером.

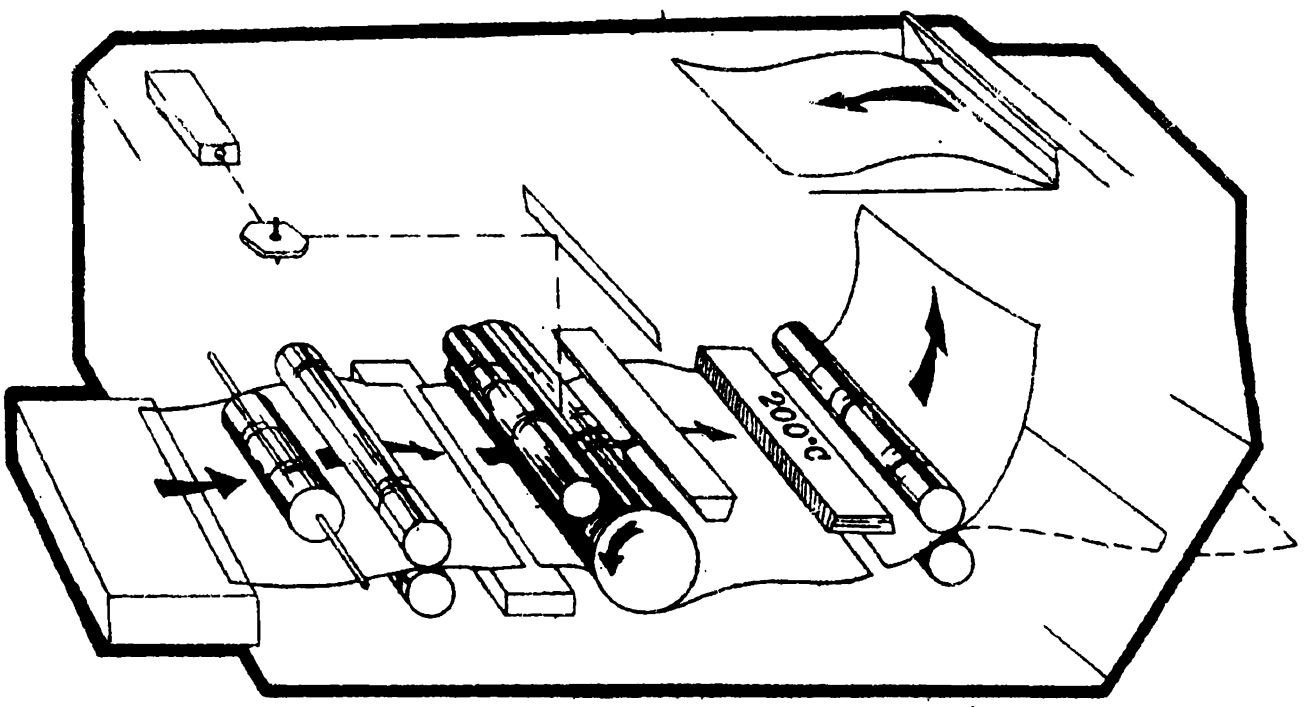


Рис. 4.6

Каждый принтер имеет свои особенности, которые находят естественное отражение в наборе команд. Однако можно выделить некоторый набор команд, реализованный на достаточно широком классе принтеров.

4.5.1. Девятиигольчатые принтеры

Рассмотрим класс 9-игольчатых принтеров типа EPSON, STAR и совместимых с ними. Ниже приводится краткая сводка основных команд для этого класса принтеров.

Мнемоника	*Десятичные коды*	*Комментарий*
LF	10	Переход на следующую строку, каретка не возвращается к началу строки
CR	13	Возврат каретки к началу строки
FF	12	Прогон бумаги до начала следующей страницы
Esc A n	27, 65, n	Установка расстояния между строками/ (величину прогона бумаги по команде LF) в n/72 дюйма
Esc J n	27, 74, n	Сдвиг бумаги на n/216 дюйма
Esc K n1 n2 data	27, 75, n1, n2, data	Печать блока графики высотой 8 пикселов и шириной n2*256+n1 пикселов с нормальной плотностью (60 точек на дюйм)
Esc L n1 n2 data	27, 76, n1, n2, data	Печать блока графики высотой 8 пикселов и шириной n2*256+n1 пикселов с двойной плотностью (120 точек на дюйм)

Esc * m n1 n2	27, 42, m, n1, n2, data	Печать блока графики высотой 8 пикселов и шириной n2*256+n1 пикселов с заданной плотностью (см. следующую таблицу)
Esc 3 n	27, 51, n	Установка расстояния между строками для последующих команд перевода строки. Расстояние устанавливается равным n/216 дюйма

Возможные режимы вывода графики задаются следующей таблицей.

Значение m	*Режим*	*Плотность (точек на дюйм)*
0	Обычная плотность	60
1	Двойная плотность	120
2	Двойная плотность, двойная скорость	120
3	Четверная плотность	240
4	CRT I	80
5	Plotter Graphics	72
6	CRT II	90
7	Plotter Graphics, двойная плотность	144

Например, для возврата каретки в начальное положение и сдвига бумаги на 5/216 дюйма нужно послать на принтер следующие байты: 13, 27, 74, 5.

Первый байт обеспечивает возврат каретки, а три следующих - сдвиг бумаги.

При печати графического изображения головка принтера за один проход рисует блок (изображение) шириной n1+256*n2 точек и высотой 8 точек. После n2 идут байты, задающие изображение, - по 1 байту на каждые 8 вертикально стоящих пикселов. Если точку нужно ставить в i-м снизу пикселе, то i-й бит в байте равен единице.

Пример:

128			•		•		•			
64		•						•		
32	•								•	
16		•						•		
8			•		•		•			
4					•					
2	•		•		•		•		•	
1					•					
Всего	34	80	138	0	143	0	138	80	34	0

Рассмотрим, как формируются байты для этой команды. Так как ширина изображения равна 10, то отсюда n1 = 10 % 256, n2 = 10/256. Для формирования первого байта, описывающего изображение, возьмем первый столбец из 8 пикселов и закодируем его битами: точке поставим в соответствие 1, а пустому месту - 0. Получившиеся биты запишем сверху вниз. При этом получается двоичное число 00100010, десятичное значение которого равно 34. Второй столбец кодируется набором бит 01010000 с десятичным значением 80. Проведя аналогичные расчеты, получим, что для печати этого изображения на принтер необходимо послать следующие коды: 27, 75, 10, 0, 34, 80, 138, 0, 143, 0, 138, 80, 34, 0.

Для вывода на принтер изображения высотой больше 8 пикселов оно предварительно разбивается на полосы высотой по 8 пикселов.

Ниже приводится пример программы, копирующей изображение экрана на 9-игольчатый матричный принтер.

```
// File Example1.cpp
#include        <bios.h>
#include        <conio.h>
#include        <graphics.h>
#include        <process.h>
#include        <stdio.h>

int     port = 0;               // use LPT1:

inline int      print ( char byte )
{
        return biosprint ( 0, byte, port );
}

void    printScreenFX ( int x1, int y1, int x2, int y2 )
{
        int         numPasses = ( y2 >> 3 ) - ( y1 >> 3 ) + 1;
        int         numCols   = x2 - x1 + 1;
        int         byte;

        print ( '\r' );
        for (int pass = 0, y = y1; pass < numPasses; pass++, y += 8)
        {
                print ( '\x1B ');
                print ( 'L' );
                print ( numCols & 0xFF );
                print ( numCols >> 8 );

                for ( int x = x1; x <= x2; x++ )
                {
                        byte = 0;
                        for ( int i = 0; i < 8 && y + i <= y2; i++ )
                                if ( getpixel ( x, y + i ) > 0 )
                                        byte |= 0x80 >> i;

                        print ( byte );
                }

                print ( '\x1B' );
                print ( 'J' );
```

```
                print ( 24 );
                print ( '\r' );
        }
}
main ()
{
        int         driver = DETECT;
        int         mode;
        int         res;

        initgraph ( &driver, &mode, "" );
        if ( ( res = graphresult () ) != grOk )
        {
           printf("\nGraphics error: %s\n", grapherrormsg ( res ) );
           exit ( 1 );
        }

        line ( 0, 0, 0, getmaxy () );
        line ( 0, getmaxy (), getmaxx (), getmaxy () );
        line ( getmaxx (), getmaxy (), getmaxx (), 0 );
        line ( getmaxx (), 0, 0, 0 );

        for ( int i = TRIPLEX_FONT; i <= GOTHIC_FONT; i++ )
        {
                settextstyle ( i, HORIZ_DIR, 5 );
                outtextxy ( 100, 50*i, "Some string" );
        }
        getch ();
        printScreenFX ( 0, 0, getmaxx (), getmaxy () );
        closegraph ();
}
```

4.5.2. Двадцатичетырехигольчатые (LQ) принтеры

Язык управления для большинства 24-игольчатых принтеров является надмножеством над языком для 9-игольчатых принтеров, поэтому все приведенные ранее команды будут работать и с LQ-принтерами (используя только 8 игл, а не 24). Для использования всех 24 игл предусмотрены дополнительные режимы в команде Esc '*'.

Значение m	*Режим*	*Плотность (точек на дюйм)*
32	Обычная плотность	60
33	Двойная плотность	120
38	CRT III	90
39	Тройная плотность	180

Количество столбцов пикселов, как и раньше, равно n1 + 256*n2, но для каждого столбца задается уже 3 байта.

Большинство струйных принтеров на уровне языка управления совместимы с LQ-принтерами.

4.5.3. Лазерные принтеры

Одним из наиболее распространенных классов лазерных принтеров являются лазерные принтеры серии HP LaserJet фирмы Hewlett Packard. Все они управляются языком PCL. Отметим, что большое количество лазерных принтеров других фирм также поддерживают язык PCL. Ниже приводится краткая сводка основных команд этого языка, используемых при выводе графики.

Мнемоника	*Десятичные коды*	*Комментарий*
Esc * t 75 R	27, 42, 116, 55, 53, 82	Установка плотности печати 75 точек на дюйм
Esc * t 100 R	27, 42, 116, 49, 48, 48, 82	Установка плотности печати 100 точек на дюйм
Esc * t 150 R	27, 42, 116, 49, 53, 48, 82	Установка плотности печати 150 точек на дюйм
Esc * t 300 R	27, 42, 116, 51, 48, 48, 82	Установка плотности печати 300 точек на дюйм
Esc & a # R	27, 38, 97, #...#, 82	Вертикальное позиционирование
Esc & a # C	27, 38, 97, #...#, 67	Горизонтальное позиционирование
Esc * r 1 A	27, 42, 114, 49, 65	Начать вывод графики
Esc * b # W data	27, 42, 98, #...#, 87, data	Передать графические данные
Esc * r B	27, 42, 114, 66	Закончить вывод графики

Здесь символ # означает, что в этом месте выводятся цифры, задающие десятичное значение числа. Пикселы собираются в байты по горизонтали, т. е. за одну команду Esc * b передается сразу целая строка пикселов.

Ниже представлена программа, копирующая содержимое экрана на лазерный принтер, поддерживающий язык PCL.

```
// File Example2.cpp
#include        <bios.h>
#include        <conio.h>
#include        <graphics.h>
#include        <process.h>
#include        <stdio.h>

int     port = 0;               // use LPT1:

inline int      print ( char byte )
{
        return biosprint ( 0, byte, port );
}

int     printStr ( char * str )
{
        int             st;

        while ( *str != '\0' )
                if ( ( st = print ( *str++ ) ) & 1 )
                        return st;
```

```
        return 0;
}
void    printScreenLJ ( int x1, int y1, int x2, int y2 )
{
        int             numCols   = x2 - x1 + 1;
        int             byte;
        char            str [20];

        printStr ( "\x1B*t150R" );          // set density 150 dpi
        printStr ( "\x1B&a5C" );            // move cursor to col 5
        printStr ( "\x1B*r1A" );            // begin raster graphics
                                                    // prepare line header
        sprintf ( str, "\x1B*b%dW", (numCols+7)>>3);

        for ( int y = y1; y <= y2; y++ )
        {
                printStr ( str );

                for ( int x = x1; x <= x2; )
                {
                        byte = 0;
                        for ( int i = 0; i < 8 && x <= x2; i++, x++ )
                                if ( getpixel ( x, y ) > 0 )
                                        byte |= 0x80 >> i;

                        Print ( byte );
                }
        }
        printStr ( "\x1B*rB" );
}
main ()
{
        int             driver = DETECT;
        int             mode;
        int             res;

        initgraph ( &driver, &mode, "" );
        if ( ( res = graphresult () ) != grOk )
        {
            printf("\nGraphics error: %s\n", grapherrormsg (res) );
            exit ( 1 );
        }

        line ( 0, 0, 0, getmaxy () );
        line ( 0, getmaxy (), getmaxx (), getmaxy () );
        line ( getmaxx (), getmaxy (), getmaxx (), 0 );
        line ( getmaxx (), 0, 0, 0 );

        for ( int i = TRIPLEX_FONT; i <= GOTHIC_FONT; i++ )
        {
                settextstyle ( i, HORIZ_DIR, 5 );
                outtextxy ( 100, 50*i, "Some string" );
        }

        getch ();
```

```
        printScreenLJ ( 0, 0, getmaxx (), getmaxy () );
        closegraph ();
}
```

4.5.4. PostScript-устройства

Наиболее высококачественные и дорогие устройства вывода (принтеры, фотонаборные автоматы) обычно управляются не набором Esc-последовательностей, а при помощи языка PostScript.

PostScript - это специальный язык для описания страницы. Он является полноценным языком программирования - в нем можно вводить переменные, есть условный оператор и оператор цикла, можно вводить свои функции.

При этом этот язык является аппаратно-независимым. Одна и та же программа, написанная на PostScript, будет успешно работать на любом PostScript-устройстве, максимально используя возможности этого устройства, будь это лазерный принтер с разрешением в 300 dpi или фотонаборный автомат с разрешением 2540 dpi.

Полное описание этого языка выходит за рамки данной книги (краткое описание можно найти в [20]), поэтому мы ограничимся примером программы, осуществляющей копирование прямоугольной области экрана на PostScript-принтер.

```
// File Example3.cpp
#include        <bios.h>
#include        <conio.h>
#include        <graphics.h>
#include        <process.h>
#include        <stdio.h>
#include        <stdlib.h>

int     port = 0;               // use LPT1:

inline int          print ( char byte )
{
        return biosprint ( 0, byte, port );
}

int     printStr ( char * str )
{
        int         st;

        while ( *str != '\0' )
                if ( ( st = print ( *str++ ) ) & 1 )
                        return st;

        return 0;
}

void    printScreenPS ( int x1, int y1, int x2, int y2, int mode )
{
        int         xSize = x2 - x1 + 1;
        int         ySize = y2 - y1 + 1;
        int         numCols = ( x2 - x1 + 8 ) >> 3;
        int         byte, bit;
        char        str [20];
```

```
        printStr ( "/bmap_wid " );
        itoa ( xSize, str, 10 );
        printStr ( str );

        printStr ( "/bmap_hgt " );
        itoa ( ySize, str, 10 );
        printStr ( str );

        printStr ( "/bpp 1 def\n" );
        printStr ( "/res " );
        itoa ( mode, str, 10 );
        printStr ( str );
        printStr ( " def\n\n" );
        printStr ( "/x 5 def\n" );
        printStr ( "/y 5 def\n" );
        printStr ( "/scx 100 100 div def\n" );
        printStr ( "/scy 100 100 div def\n" );
        printStr ( "/scg 100 100 div def\n" );
        printStr ( "scaleit\n" );
        printStr ( "imagedata\n\n" );

        for ( int y = y1; y <= y2; y++ )
        {
                for ( int i = 0, x = x1; i < numCols; i++ )
                {
                        for ( int j = 0, bit = 0x80, byte = 0; j < 8
                        && x + j <= x2; j++, bit >>= 1 )
                                if ( getpixel ( x + j, y ) > 0 )
                                        byte |= bit;
                        itoa ( byte, str, 16 );
                        printStr ( str );
                }
                printStr ( "\n" );
        }
        printStr ( "\nshowit\n" );
}
main ()
{
        int         driver = DETECT;
        int         mode;
        int         res;

        initgraph ( &driver, &mode, "" );
        if ( ( res = graphresult () ) != grOk )
        {
            printf("\nGraphics error: %s\n", grapherrormsg (res) );
            exit ( 1 );
        }

        line ( 0, 0, 0, getmaxy () );
        line ( 0, getmaxy (), getmaxx (), getmaxy () );
        line ( getmaxx (), getmaxy (), getmaxx (), 0 );
        line ( getmaxx (), 0, 0, 0 );
```

```
        for ( int i = TRIPLEX_FONT; i <= GOTHIC_FONT; i++ )
        {
                settextstyle ( i, HORIZ_DIR, 5 );
                outtextxy ( 100, 50*i, "Some string" );
        }
        getch ();
        printScreenPS ( 0, 0, getmaxx (), getmaxy (), 100 );
        closegraph ();
}
```

4.6. Видеокарты EGA и VGA

Основным графическим устройством, с которым чаще всего приходится работать, является видеосистема компьютера. Обычно она состоит из видеокарты (адаптера) и подключенного к ней монитора. Изображение хранится в растровом виде в памяти видеокарты: аппаратура карты обеспечивает регулярное (50-70 раз в секунду) чтение этой памяти и отображение ее на экране монитора. Поэтому вся работа с изображением сводится к тем или иным операциям с видеопамятью.

Наиболее распространенными видеокартами сейчас являются клоны карт EGA (Enhanced Graphics Adaptor) и VGA (Video Graphics Array). Кроме того, существует большое количество различных SVGA-карт, которые будут рассмотрены в конце главы.

Приведем список основных режимов для этих карт. Режим определяется номером, разрешением экрана и количеством цветов.

Номер режима	*Разрешение экрана*	*Количество цветов*
0Dh	320×200	16
0Eh	640×200	16
0Fh	640×350	2
10h	640×350	16
11h (VGA)	640×480	2
12h (VGA)	640×480	16
13h (VGA)	320×200	256

Каждая видеоплата содержит в своем составе собственный BIOS для работы с ней и поддержки основных функций платы.

Ниже приводится файл, содержащий базовые функции по работе с графикой, доступные через BIOS.

```
// File ega.Cpp
#include        <dos.h>
#include        "ega.h"

int     findEGA ()
{
        asm {
                mov     ax, 1200h
```

```
                mov     bx, 10h
                int     10h
        }
        return _BL != 0x10;
}
int     findVGA ()
{
        asm {
                mov     ax, 1A00h
                int     10h
        }
        return _AL == 0x1A;
}
void    setVideoMode ( int mode )
{
        asm {
                mov     ax, mode
                int     10h
        }
}
void    setVisiblePage ( int page )
{
        asm {
                mov     ah, 5
                mov     al, byte ptr page
                int     10h
        }
}
char far * findROMFont ( int size )
{
        int     b = ( size == 16 ? 6 : 2 );
        asm {
                push    es
                push    bp
                mov     ax, 1130h
                mov     bh, byte ptr b
                mov     bl, 0
                int     10h
                mov     ax, es
                mov     bx, bp
                pop     bp
                pop     es
        }
        return (char far *) MK_FP ( _AX, _BX );
}
void    setPalette ( RGB far * palette, int size )
{
        asm {
                push    es
```

```
                mov     ax, 1012h
                mov     bx, 0                   // first color to set
                mov     cx, size                // # of colors
                les     dx, palette             // ES:DX == table of
                int     10h                     // color values
                pop     es
        }
}
void    getPalette ( RGB far * palette )
{
        asm {
                push    es
                mov     ax, 1017h
                mov     bx, 0                   // from index
                mov     cx, 256                 // # of pal entries
                les     dx, palette
                int     10h
                pop     es
        }
}
```

Функции findEGA и findVGA позволяют определить наличие EGA- или VGA-совместимой видеокарты.

Для установки нужного режима можно воспользоваться процедурой setVideoMode.

Функция findROMFont возвращает адрес системного шрифта заданного размера (8, 14 или 16 пикселов высоты).

Функция setPalette служит для установки палитры и является аналогом функции setrgbpalette.

Функция getPalette возвращает текущую палитру (256 цветов).

4.7. Шестнадцатицветные режимы адаптеров EGA и VGA

Для 16-цветных режимов под каждый пиксел изображения необходимо выделить 4 бита видеопамяти ($2^4 = 16$). Однако эти 4 бита выделяются не последовательно в одном байте, а разнесены в 4 разных блока (цветовые плоскости) видеопамяти.

Вся видеопамять карты (обычно 256 Кбайт) делится на 4 равные части, называемые цветовыми плоскостями. Каждому пикселу ставится в соответствие по одному биту в каждой плоскости, причем все эти биты одинаково расположены относительно ее начала.

Обычно цветовые плоскости представляют параллельно расположенными одна над другой, так что каждому пикселу соответствует 4 бита, расположенных друг под другом (рис. 4.7).

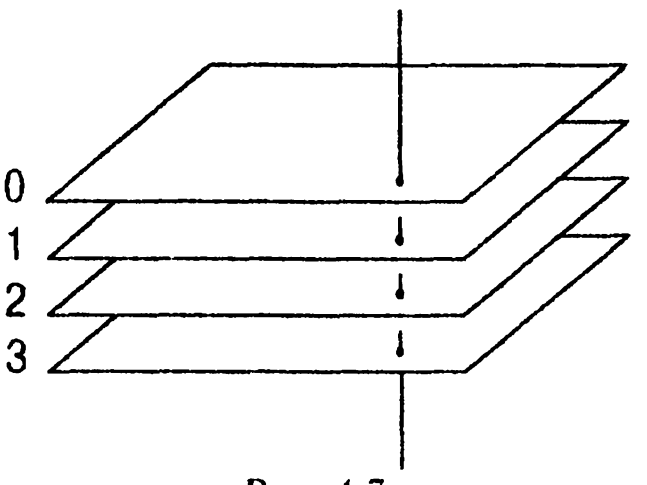


Рис. 4.7

Все эти плоскости проектируются на один и тот же участок адресного пространства процессора начиная с адреса 0xA000:0.

При этом все операции чтения и записи видеопамяти опосредуются видеокартой. Поэтому если вы записали байт по адресу 0xA000:0, то это вовсе не означает, что посланный байт в действительности запишется хотя бы в одну из этих плоскостей, точно так же как при операции чтения прочитанный байт не обязательно будет совпадать с одним из 4 байтов в соответствующих плоскостях. Механизм этого опосредования определяется логикой карты, но для программиста существует возможность известного управления этой логикой (при работе одновременно с 8 пикселами).

Для работы с пикселом необходимо определить адрес байта в ви-деопамяти, содержащего данный пиксел, и позицию пиксела внутри байта (поскольку 1 пиксел отображается на 1 бит в каждой плоскости, то байт соответствует сразу 8 пикселам).

Поскольку видеопамять под пикселы отводится последовательно слева направо и сверху вниз, то одна строка соответствует 80 байтам адреса и каждым 8 последовательным пикселам, начинающимся с позиции, кратной 8, соответствует 1 байт. Тем самым адрес байта задается выражением 80*y + (x>>3), а его номер внутри байта - выражением x&7, где (x, y) - координаты пиксела.

Для идентификации позиции пиксела внутри байта часто используется не номер бита, а битовая маска - байт, в котором отличен от нуля только бит, стоящий на позиции пиксела.

Битовая маска задается следующим выражением: 0x80>>(x&7).

На видеокарте находится набор специальных 8-битовых регистров. Часть из них доступна только для чтения, часть - только для записи, а некоторые вообще недоступны программисту. Доступ к регистрам осуществляется через порты ввода/вывода процессора.

Регистры видеокарты делятся на несколько групп. Каждой группе соответствует пара последовательных портов (порт адреса и порт значения). Для записи значения в регистр видеокарты необходимо сначала записать номер регистра в первый порт (порт адреса), а затем записать значение в следующий порт (порт значения). Для чтения регистра в порт адреса записывается номер регистра, а затем его значение читается из порта значения.

Ниже приводится файл, определяющий необходимые константы и inline-функции для работы с портами видеокарты. Функции writeReg и readReg служат для доступа к регистрам.

```
// File ega.h
#ifndef     __EGA__
#define     __EGA__

#include    <dos.h>

#define     EGA_GRAPHICS        0x3CE  // Graphics Controller addr
#define     EGA_SEQUENCER       0x3C4  // Sequencer base addr
#define     EGA_CRTC            0x3D4

#define     EGA_SET_RESET             0
#define     EGA_ENABLE_SET_RESET      1
#define     EGA_COLOR_COMPARE         2
#define     EGA_DATA_ROTATE           3
```

```
#define         EGA_READ_MAP_SELECT     4
#define         EGA_MODE                5
#define         EGA_MISC                6
#define         EGA_COLOR_DONT_CARE     7
#define         EGA_BIT_MASK            8

#define         EGA_MAP_MASK            2

struct RGB
{
        char        red;
        char        dreen;
        char        blue;
};

inline void         writeReg ( int base, int reg, int value )
{
        outportb ( base,     reg );
        outportb ( base + 1, value );
}

inline char         readReg ( int base, int reg )
{
        outportb ( base, reg );

        return inportb ( base + 1 );
}

inline char         pixelMask ( int x )
{
        return 0x80 >> ( x & 7 );
}

inline char         leftMask ( int x )
{
        return 0xFF >> ( x & 7 );
}

inline char         rightMask ( int x )
{
        return 0xFF << ( 7 ^ ( x & 7 ) );
}

inline void         setRWMode ( int readMode, int writeMode )
{
        writeReg ( EGA_GRAPHICS, EGA_MODE, ( writeMode & 3 ) |
                        ( ( readMode & 1 ) << 3 ) );
}

inline void         setWriteMode ( int mode )
{
        writeReg ( EGA_GRAPHICS, EGA_DATA_ROTATE, ( mode & 3 ) <<
}

int             findEGA ();
int             findVGA ();
void            setVideoMode ( int );
void            setVisiblePage ( int );
```

```
char far *    findROMFont ( int );
void          setPalette ( RGB far * palette, int );

#endif
```

Рассмотрим две основные группы регистров, принадлежащих двум частям видеокарты, - Graphics Controller и Sequencer.

Каждой группе соответствует своя пара портов.

4.7.1. Graphics Controller (порты 3CE-3CF)

Номер	*Регистр*	*Стандартное значение*
0	Set/Reset	00
1	Enable Set/Reset	00
2	Color Compare	00
3	Data rotate	00
4	Read Map Select	00
5	Mode	10
6	Miscellaneous	05
7	Color Don't Care	0F
8	Bit Mask	FF

Для записи в регистр необходимо сначала послать номер регистра в порт 3CE, а затем записать соответствующее значение в порт 3CF.

Для EGA-карты все эти регистры доступны только для чтения; VGA-адаптер поддерживает и запись, и чтение.

Проиллюстрируем это на процессе установки регистра битовой маски (Bit Mask) (установка остальных регистров проводится аналогично).

```
void  setBitMask (char mask)
{
  writeReg (EGA_GRAPHICS, EGA_BIT_MASK, mask);
}
```

4.7.2. Sequencer (порты 3C4-3C5)

Из регистров этой группы мы рассмотрим только регистр маски плоскости (Map Mask) и номер 2.

Процедура setMapMask устанавливает значение регистра маски плоскости.

Рассмотрим, как проходит работа с видеопамятью.

При операции чтения байта из видеопамяти читаются сразу 4 байта - по одному из каждой плоскости. При этом прочитанные значения записываются в специальные регистры - "защелки" (latch-регистры), для прямого доступа недоступные. Байт, прочитанный процессором, является комбинацией значений latch-регистров.

При операции записи посланный процессором байт накладывается на значения latch-регистров по правилам, определяемым значениями других регистров, а результирующие 4 байта записываются в соответствующие плоскости.

Так как при записи используются значения latch-регистров, то часто необходимо, чтобы перед записью в них находились исходные значения тех байтов, кото-

рые затем изменяются. Это часто приводит к необходимости осуществлять чтение байта по адресу перед записью по этому адресу нового значения.

Правила, определяющие наложение при записи посланных процессором данных на значения latch-регистров, определяются установленным режимом записи, и, соответственно, режим чтения задает способ, которым определяется значение, прочитанное процессором.

Видеокарта EGA поддерживает два режима чтения и три режима записи; у карты VGA есть еще один дополнительный режим записи.

Установка режимов чтения и записи осуществляется записью соответствующих значений в регистр Mode. Бит 3 отвечает за режим чтения, биты 0 и 1 - за режим записи.

Функция setRWMode служит для установки режимов чтения и записи.

4.8. Режимы чтения

4.8.1. Режим чтения 0

В этом режиме возвращается байт из latch-регистра (плоскости) с номером из регистра Read Map Select.

В приведенном ниже примере возвращается значение (цвет) пиксела с координатами (*x, y*). Для этого с каждой из плоскостей по очереди читаются биты и из них собирается цветовое значение пиксела.

```
// File ReadPxl.cpp
int     readPixel ( int x, int y )
{
   int       color = 0;
   char far * vptr  = (char far *) MK_FP (0xA000, y*80+(x>>3));
   char      mask  = pixelMask ( x );

   for ( int plane = 3; plane >= 0; plane-- )
   {
       writeReg ( EGA_GRAPHICS, EGA_READ_MAP_SELECT, plane );
       color <<= 1;
       if ( *vptr & mask )
          color |= 1;
   }

   return color;
}
```

4.8.2. Режим чтения 1

В возвращаемом значении *i*-й бит равен единице, если

GetPixel & ColorDon'tCare == ColorCompare & ColorDon'tCare

В случае, если ColorDon'tCare == 0F, в прочитанном байте в тех позициях, где цвет пиксела совпадает со значением в регистре ColorCompare, будет стоять единица.

Этот режим очень удобен для поиска точек заданного цвета.

Приведенная процедура осуществляет поиск пиксела цвета Color в строке *y* начиная с позиции *x*. При этом используется режим чтения 1. Все байты, соот-

ветствующие данной строке, читаются по очереди, и, как только будет получено ненулевое значение (найден по крайней мере 1 пиксел данного цвета в байте), оно возвращается.

```
// File FindPxl.cpp
int     findPixel ( int x1, int x2, int y, int color )
{
 char far * vptr  = (char far *) MK_FP (0xA000, y*80+(x1>>3));
 int       cols  = ( x2 >> 3 ) - ( x1 >> 3 ) - 1;
 char      lmask = leftMask ( x1 );
 char  rmask = rightMask ( x2 );
 char  mask;

    setRWMode ( 1, 0 );
    writeReg ( EGA_GRAPHICS, EGA_COLOR_COMPARE, color );
    if ( cols < 0 )
        return *vptr & lmask & rmask;

    if ( mask = *vptr++ & lmask )
        return mask;

    while ( cols-- > 0 )
        if ( mask = *vptr++ )
            return mask;

   return *vptr & rmask;
}
```

4.9. Режимы записи

4.9.1. Режим записи 0

Это, пожалуй, самый сложный из всех рассматриваемых режимов, дающий, однако, и самые большие возможности.

В рассматриваемом режиме регистр BitMask позволяет защищать от изменения определенные пикселы. В тех позициях, где соответствующий бит из регистра BitMask равен нулю, пиксел не изменяет своего значения. Регистр MapMask позволяет защищать от изменения определенные плоскости.

Биты 3 и 4 регистра DataRotate определяют способ наложения выводимого изображения на существующее (аналогично функции setwritemode).

Значение битов	*Операция*	*Эквивалент в BGI*
0 0	Замена	COPY_PUT
0 1	Or	OR_PUT
1 0	And	AND_PUT
1 1	Xor	XOR_PUT

Процедура setWriteMode устанавливает соответствующий режим наложения.

Посланный процессором байт циклически сдвигается вправо на указанное в битах 0-2 регистра Data Rotate количество раз.

Результирующее значение определяется следующим образом. На плоскость, соответствующий бит которой в регистре Enable Set/Reset равен нулю, накладывается посланный процессором байт, "прокрученный" заданное количество раз, с учетом регистров BitMask и MapMask. Если соответствующий бит равен единице, то во все позиции, разрешенные регистром BitMask, записывается бит из регистра Set/Reset, соответствующий плоскости.

На практике наиболее часто встречаются следующие два случая:

- Enable Set/Reset = 0 (байт, посланный процессором, циклически сдвигается в соответствии со значением битов 0-2 регистра Data Rotate; после этого получившийся байт заданным способом (см. биты 3-4 регистра Data Rotate) накладывается на те плоскости, которые разрешены регистром Map Mask, причем изменяются лишь разрешенные регистром BitMask биты);
- Enable Set/Reset = 0F (в позиции, разрешенные регистром BitMask, ставятся точки цвета, заданного в регистре Set/Reset; байт, посланный процессором, никакой роли не играет).

Для того чтобы нарисовать только нужный пиксел, необходимо поставить регистр BitMask так, чтобы защитить от изменения остальные 7 пикселов, соответствующих этому байту.

```
// File WritePxl.cpp
void  writePixel ( int x, int y, int color )
{
 char far * vptr = (char far *) MK_FP (0xA000, y*80+(x>>3));
                    // enable all planes
  writeReg ( EGA_GRAPHICS, EGA_ENABLE_SET_RESET, 0x0F );
  writeReg ( EGA_GRAPHICS, EGA_SET_RESET, color );
  writeReg ( EGA_GRAPHICS, EGA_BIT_MASK, PixelMask ( x ) );
  *vptr += 1;       // perform read/write at memory loc.
                 // disable all planes
 writeReg ( EGA_GRAPHICS, EGA_ENABLE_SET_RESET, 0 );
                 // restore reg
 writeReg ( EGA_GRAPHICS, EGA_BIT_MASK, 0xFF );
}
```

4.9.2. Режим записи 1

В этом режиме значения latch-регистров непосредственно копируются в соответствующие плоскости. Регистры масок и режима не действуют. Посланное процессором значение не играет никакой роли. Этот режим позволяет осуществлять быстрое копирование фрагментов видеопамяти. При чтении байта по исходному адресу прочитанные 4 байта с плоскостей загружаются в latch-регистры, а при записи значения latch-регистров записываются в плоскости по адресу, по которому шла запись. Таким образом, за одну операцию перезаписи копируется сразу 4 байта (8 пикселов).

Приведенная ниже функция осуществляет копирование прямоугольной области экрана в соответствующую область с верхним левым углом в точке (x, y).

В силу ограничений режима записи 1 эта процедура может копировать только области, где $x1$ кратно 8 и ширина кратна 8, так как копирование осуществляется блоками по 8 пикселов сразу. Кроме того, этот пример не учитывает возможности

того, что область, куда производится копирование, имеет непустое пересечение с исходной областью. В подобном случае возможна некорректная работа процедуры и, чтобы подобного не возникало, необходимо заранее проверять области на пересечение: при непустом пересечении копирование осуществляется в обратном порядке.

```
// File copyrect.cpp
void copyRect(int x1, int y1, int x2, int y2, int x, int y)
{
   char far *src = (char far *) MK_FP (0xA000, y1*80+(x1 >> 3));
   char far *dst = (char far *) MK_FP (0xA000, y*80+(x >> 3));
   int    cols  = ( x2 >> 3 ) - ( x1 >> 3 );

      setRWMode ( 0, 1 );
      for ( int i = y1; i <= y2; i++ )
      {
         for ( int j = 0; j < cols; j++ )
            *dst++ = *src++;

         src += 80 - cols;
         dst += 80 - cols;
      }

      setRWMode ( 0, 0 );
   }
```

4.9.3. Режим записи 2

В этом режиме младшие 4 бита байта, посланного процессором, определяют цвет, которым будут построены незащищенные битовой маской пикселы. Регистр битовой маски защищает от изменения определенные пикселы. Регистр маски плоскости защищает от изменения определенные плоскости. Регистр DataRotate устанавливает способ наложения построенных пикселов на существующее изображение.

Приведенный ниже пример рисует прямоугольник заданного цвета, используя режим записи 2.

```
// File bar.cpp
void  bar ( int x1, int y1, int x2, int y2, int color )
{
   char far * vptr  = (char far *) MK_FP (0xA000, y1*80+(x1>>3));
   int     cols  = ( x2 >> 3 ) - ( x1 >> 3 ) - 1;
   char    lmask = leftMask ( x1 );
   char    rmask = rightMask ( x2 );
   char    latch;

   setRWMode ( 0, 2 );
   if ( cols < 0 )   // both x1 & x2 are located in the same byte
   {
      writeReg ( EGA_GRAPHICS, EGA_BIT_MASK, lmask & rmask );
      for ( int y = y1; y <= y2; y++, vptr += 80 )
      {
         latch = *vptr;
         *vptr = color;
      }
```

```
        writeReg ( EGA_GRAPHICS, EGA_BIT_MASK, 0xFF );
    }
    else
    {
        for ( int y = y1; y <= y2; y++ )
        {
            writeReg ( EGA_GRAPHICS, EGA_BIT_MASK, lmask );
            latch   = *vptr;
            *vptr++ = color;
            writeReg ( EGA_GRAPHICS, EGA_BIT_MASK, 0xFF );

            for ( int x = 0; x < cols; x++ )
            {
                latch   = *vptr;
                *vptr++ = color;
            }
            writeReg ( EGA_GRAPHICS, EGA_BIT_MASK, rmask );

            latch   = *vptr;
            *vptr++ = color;
            vptr   += 78 - cols;
        }
    }

    setRWMode ( 0, 0 );
    writeReg  ( EGA_GRAPHICS, EGA_BIT_MASK, 0xFF );
}
```

Следующие две функции служат для запоминания и восстановления запис го изображения.

```
// File store.cpp
void  storeRect ( int x1, int y1, int x2, int y2, char huge * buf )
{
    char far * vptr  = (char far *) MK_FP (0xA000, y1*80+(x1>>3));
    int        cols  = ( x2 >> 3 ) - ( x1 >> 3 ) - 1;
    if ( cols < 0 )
        cols = 0;

    for ( int y = y1; y <= y2; y++, vptr += 80 )
        for ( int plane = 0; plane < 4; plane++ )
        {
            writeReg ( EGA_GRAPHICS, EGA_READ_MAP_SELECT, plane );
            for ( int x = 0; x < cols + 2; x++ )
                *buf++ = *vptr++;

            vptr -= cols + 2;
        }
}
void restoreRect (int x1, int y1, int x2, int y2, char huge * buf)
{
    char far * vptr  = (char far *) MK_FP (0xA000, y1*80+(x1>>3));
    int        cols  = ( x2 >> 3 ) - ( x1 >> 3 ) - 1;
    char       lmask = leftMask ( x1 );
    char       rmask = rightMask ( x2 );
    char       latch;
```

```
    if ( cols < 0 )
    {
        lmask &= rmask;
        rmask  = 0;
        cols   = 0;
    }
    for ( int y = y1; y <= y2; y++, vptr += 80 )
        for ( int plane = 0; plane < 4; plane++ )
        {
            writeReg ( EGA_GRAPHICS,  EGA_BIT_MASK, lmask );
            writeReg ( EGA_SEQUENCER, EGA_MAP_MASK, 1 << plane );
            latch  = *vptr;
           *vptr++ = *buf++;

            writeReg ( EGA_GRAPHICS, EGA_BIT_MASK, 0xFF );
            for ( int x = 0; x < cols; x++ )
                *vptr++ = *buf++;

            writeReg ( EGA_GRAPHICS,  EGA_BIT_MASK, rmask );
            latch   = *vptr;
            *vptr++ = *buf++;
            vptr   -= cols + 2;
        }
    writeReg ( EGA_GRAPHICS,  EGA_BIT_MASK, 0xFF );
    writeReg ( EGA_SEQUENCER, EGA_MAP_MASK, 0x0F );
}
```

4.9.4. Режим адаптера VGA с 256-цветами

Из всех видеорежимов этот режим является самым простым. При разрешении экрана 320x200 точек он позволяет одновременно использовать все 256 цветов. Для одновременного отображения 256 цветов необходимо под каждую точку на экране отвести по 8 бит. В рассматриваемом режиме эти 8 бит идут последовательно один за другим, образуя 1 байт. Тем самым в этом режиме плоскости не используются. Видеопамять начинается с адреса 0xA000:0. При этом точке с координатами (x, y) соответствует байт памяти по адресу $320y + x$.

```
// File vga256.cpp
void writePixel ( int x, int y, int color )
{
pokeb ( 0xA000, 320*y + x, color );
}
int readPixel ( int x, int y )
{
  return peekb ( 0xA000, 320*y + x );
}
```

Для установки регистров палитры можно воспользоваться ранее введенной функцией setPalette, осуществляющей установку регистров через BIOS, а сделать это можно непосредственным программированием регистров DAC. Функция setPaletteDirect устанавливает палитру из 256 цветов путем программирования DAC-регистров.

```
// File dacpal.cpp
void setPaletteDirect ( RGB palette [] )
{
        // wait for vertical retrace
    while ( (inportb (0x3DA) & 0x08) != 0 )
        ;
    while ( (inportb (0x3DA) & 0x08) == 0 )
        ;

    outportb ( 0x3C8, 0 );
    for ( register int i = 0; i < 256; i++ )
    {
        outportb ( 0x3C9, palette [i].red     >> 2 );
        outportb ( 0x3C9, palette [i].green   >> 2 );
        outportb ( 0x3C9, palette [i].blue    >> 2 );
    }
}
```

Для того чтобы избежать искажения изображения на экране, DAC-регистры нужно программировать только во время обратного вертикального хода луча.

Сдвиг цветовых компонент палитры необходим, поскольку только младшие 6 бит DAC-регистра являются значащими.

4.9.5. Спрайты и работа с ними

Понятие спрайта появилось в связи с разработкой компьютерных игр. Под спрайтами обычно понимают небольшие графические объекты (изображения), которые находятся на игровом поле, могут двигаться и изменять свою форму (например, идущий человечек - при этом циклически выводится набор изображений, соответствующих разным фазам движения). На некоторых компьютерах (Atari, Amiga, Yamaha) спрайты реализованы аппаратно.

Поскольку на IBM PC аппаратной поддержки спрайтов нет, то поддержка должна осуществляться на программном уровне.

Фактически спрайт представляет собой набор изображений одинакового размера, соответствующих различным состояниям объекта, при этом среди всех используемых цветов в изображениях выделяется один - так называемый прозрачный цвет, т. е. пикселы, соответствующие этому цвету, не выводятся и в этих местах изображение под спрайтом сохраняется.

Для того чтобы корректно работать со спрайтами, необходимо уметь восстанавливать изображение под спрайтом (при смене состояния спрайта, его перемещении или убирании). Существует два пути, которыми этого можно достигнуть:

- полностью перерисовать весь экран (фон, на который будут потом выводиться спрайты);
- перед выводом спрайта запомнить участок изображения, которое он собой закрывает, и потом восстановить этот участок.

В зависимости от типа игры предпочтительнее оказывается тот или другой подход.

Рассмотрим класс, отвечающий за программную реализацию спрайта (при этом предполагается, что установлен 256-цветный режим с линейной адресацией памяти).

```
// File sprite.h
```

```
#ifndef         __SPRITE__
#define         __SPRITE__

#define         MAX_STAGES      20
#define         TRANSP_COLOR    0xFF

class Sprite
{
public:
        int         x, y;                   // location of upper-left corner
        int         width, height;          // size of single image
        int         stageCount;             // number of stages
        int         curStage;               // current stage
        char  * underImage;                 // place to store image
                                            // under the sprite
        char  * image [MAX_STAGES];

        Sprite ( int, int, char *, ... );
        ~Sprite ()
        {
                free ( underImage );
        }

        void        set ( int ax, int ay )
        {
                x = ax;
                y = ay;
        }

        void        draw        ();
        void        storeUnder  ();
        void        restoreUnder ();
};

inline int          min ( int x, int y )
{
        return x < y ? x : y;
}

inline int          max ( int x, int y )
{
        return x > y ? x : y;
}

extern              char far * videoAddr;
extern              int         screenWidth;
extern              int         screenHeight;
extern              int         orgX;
extern              int         orgY;

#endif

// File sprite.cpp
#include          <ailoc.h>
#include          <dos.h>
#include          "sprite.h"

Sprite :: Sprite ( int w, int h, char * im1,     )
```

```
{
        char ** imPtr = &im1;

        x       = 0;
        y       = 0;
        width     = w;
        height    = h;
        curStage  = 0;
        underImage = (char *) malloc ( width * height );

        for ( stageCount=0; *imPtr != NULL; imPtr ++, stageCount ++ )
                image [stageCount] = * imPtr;
}

void  Sprite :: draw () ·
{
        int             x1 = max ( 0, x - orgX );
        int             x2 = min ( screenWidth, x - orgX + width );

        if ( x2 < x1 )
                return;

        int             y1 = max ( 0, y - orgY );
        int             y2 = min ( screenHeight, y - orgY + height );

        if ( y2 < y1 )
                return;

        char far * videoPtr = videoAddr + y1 * screenWidth + x1;
        char       * dataPtr  = image [curStage]+(y1-y)*width + (x1-x);
        int             step    = screenWidth - (x2 - x1);

        for ( register int y = y1; y < y2; y++, videoPtr += step )
        {
                for (int x = x1; x < x2; x++, dataPtr++, videoPtr++)
                        if ( * dataPtr != TRANSP_COLOR )
                                * videoPtr = * dataPtr;

                dataPtr += width - (x2 - x1);
        }
}

void  Sprite   storeUnder ()
{
        int             x1 = max ( 0, x - orgX );
        int             x2 = min ( screenWidth, x - orgX + width );
        int             y1 = max ( 0, y - orgY );
        int             y2 = min ( screenHeight, y - orgY + height );
        char far * videoPtr = videoAddr + y1 * screenWidth + x1;
        char       * ptr     = underImage;
        int             step    = screenWidth - (x2 - x1);

        for ( register int y = y1; y < y2; y++, videoPtr += step )
                for ( register int x = x1; x < x2; x++ )
                        * ptr ++ = * videoPtr ++;
}

void  Sprite   restoreUnder ()
{
```

```
    int             x1 = max ( 0, x - orgX );
    int             x2 = min ( screenWidth, x - orgX + width );
    int             y1 = max ( 0, y - orgY );
    int             y2 = min ( screenHeight, y - orgY + height );
    char far * videoPtr = videoAddr + y1 * screenWidth + x1;
    char        * ptr = underImage;
    int             step    = screenWidth - (x2 - x1);

    for ( register int y = y1; y < y2; y++, videoPtr += step )
            for ( register int x = x1; x < x2; x++ ),
                    * videoPtr ++ = * ptr ++;
}
```

Для ускорения вывода спрайта можно воспользоваться механизмом RLE-кодирования (RLE-Run Length Encoding) строки, разбивающим изображение на прозрачные и непрозрачные участки. Каждый такой участок начинается с байта, задающего длину участка, при этом старший бит этого байта определяет, является ли этот участок набором прозрачных пикселов (бит установлен) или набором выводимых пикселов (бит сброшен), а в младших 7 битах хранится длина участка (количество пикселов). Для участков, состоящих из непрозрачных пикселов, за байтом длины следует соответствующее количество байт данных. Подобная организация данных позволяет избежать проверки на прозрачность при выводе каждого пиксела и выводить пикселы сразу целыми группами. Так, последовательность из 12 пикселов

0, 0, 0, 0, 0, 1, 1, 5, 1, 0, 0, 7

будет закодирована следующим набором байтов:

0x85, 0x04, 0x01, 0x01, 0x05, 0x01, 0x82, 0x01, 0x07.

Однако поскольку теперь количество байт, задающих строку, не является больше постоянной величиной, то для каждой фазы спрайта (соответствующего изображения) необходимо задать начало каждой строки относительно начала изображения.

Соответствующая программная реализация приводится ниже.

```
// File sprite2.h
#ifndef         __SPRITE__
#define         __SPRITE__

#define         MAX_STAGES      20
#define         MAX_HEIGHT      100
#define         TRANSP_COLOR    0xFF

class Sprite
{
public:
        int         x, y;                   // location of upper-left corner
        int         width, height;          // size of single image
        int         stageCount;             // number of stages
        int         curStage;               // current stage
        char        * underImage;           // place to store image
                                            // under the sprite
        char        * lineStart [MAX_STAGES*MAX_HEIGHT];

        Sprite ( int, int, char *,    );
        ~Sprite ()
```

```
        {
                free ( underImage );
        }
        void    set ( int ax, int ay )
        {
                x = ax;
                y = ay;
        }
        void    draw            ();
        void    storeUnder      ();
        void    restoreUnder    ();
};
inline int      min ( int x, int y )
{
        return x < y ? x : y;
}
inline int      max ( int x, int y )
{
        return x > y ? x : y;
}
extern          char far *      videoAddr;
extern          int             screenWidth;
extern          int             screenHeight;
extern          int             orgX;
extern          int             orgY;
#endif
```

```
// File sprite2.cpp
#include        <alloc.h>
#include        <dos.h>
#include        "sprite2.h"
Sprite :: Sprite ( int w, int h, char * im1,    )
{
        char ** imPtr = &im1;
        x       = 0;
        y       = 0;
        width   = w;
        height  = h;
        curStage  = 0;
        underImage = (char *) malloc ( width * height );
        for ( int lineCount = 0; * imPtr != NULL; imPtr ++ )
        {
                char  * ptr = * imPtr;
                for ( int i = 0; i < height; i++ )
                {
                        lineStart [lineCount] = ptr;
                        for ( int j = 0; j < width; j++ ).
```

```
                        {
                                int         count = * ptr++;
                                                    // not transparent
                                if ( ( count & 0x80 ) == 0 )
                                        ptr += count & 0x7F;
                                j += count & 0x7F;
                        }
                }
        }
}
void  Sprite :: draw ()
{
        int             x1 = max ( 0, x - orgX );
        int             x2 = min ( screenWidth, x - orgX + width );
        if ( x2 < x1 )
                return;
        int             y1 = max ( 0, y - orgY );
        int             y2 = min ( screenHeight, y - orgY + height );
        if ( y2 < y1 )
                return;
        char far * videoPtr = videoAddr + y1 * screenWidth + x1;
        int             step    = screenWidth - (x2 - x1);
        for ( register int y = y1; y < y2; y++, videoPtr += step )
    {
        char    * dataPtr =lineStart [curStage*height+y1-y];
        char far * videoPtr=videoAddr + y * screenWidth + x1;
        for ( int i = x; x < x1; )          // skip invisible
        {                                               // pixels at start
                int         count = * dataPtr++;
                if ( count & 0x80 )  // transparent block
                        i += count & 0x7F;
                else
                {
                        count &= 0x7F;
                        if ( i + count < x1 )
                        {
                                i       += count;
                                dataPtr += count;
                        }
                        else
                        {
                                dataPtr += x1 - i;
                                count   -= x1 - i;
                                if ( count > x2 - i )
                                        count = x2 - i;
                                for ( ; count > 0; count-- )
                                        * videoPtr++ = * dataPtr++;
```

```
                        }
                    }
                }
                for ( ; i < x2; i++ )
                {
                    int       count = * dataPtr++;
                    if ( count & 0x80 )
                    {
                                        count    &= 0x7F;
                                        i        += count;
                                        videoPtr += count;
                     }
                     else
                     {
                                        count &= 0x7F;
                                        if ( count > x2 - i )
                                                count = x2 - i;
                                        i += count;
                                        for ( ; count > 0; count-- )
                                                * videoPtr++ = * dataPtr++;
                        }
                }
        }
}
void  Sprite :: storeUnder ()
{
        int             x1 = max ( 0, x - orgX );
        int             x2 = min ( screenWidth, x - orgX + width );
        int             y1 = max ( 0, y - orgY );
        int             y2 = min ( screenHeight, y - orgY + height );
        char far * videoPtr = videoAddr + y1 * screenWidth + x1;
        char        * ptr = underImage;
        int             step    = screenWidth - (x2 - x1);
        for ( register int y = y1; y < y2; y++, videoPtr += step )
                for ( register int x = x1; x < x2; x++ )
                        * ptr ++ = * videoPtr ++;
}
void  Sprite   restoreUnder ()
{
        int             x1 = max ( 0, x - orgX );
        int             x2 = min ( screenWidth, x - orgX + width );
        int             y1 = max ( 0, y - orgY );
        int             y2 = min ( screenHeight, y - orgY + height );
        char far * videoPtr = videoAddr + y1 * screenWidth + x1;
        char        * ptr = underImage;
        int             step    = screenWidth - (x2 - x1);
        for ( register int y = y1; y < y2; y++, videoPtr += step )
                for ( register int x = x1; x < x2; x++ )
```

```
* videoPtr ++ = * ptr ++;
}
```

Для успешной работы с подобными спрайтами нужен менеджер спрайтов, осуществляющий управление их выводом и, при необходимости, скроллирование экрана.

Используя класс Sprite, несложно написать простейший вариант игры типа Command&Conquer, Red Alert, StarCraft или другой стратегии real-time.

Каждый уровень игры строится из набора стандартных картинок (tile) и набора спрайтов.

Карта уровня представляет собой прямоугольный массив номеров картинок. Обычно полное изображение карты заметно превосходит разрешение экрана и возникает необходимость вывода только тех картинок и спрайтов, которые видны на экране (рис. 4.8).

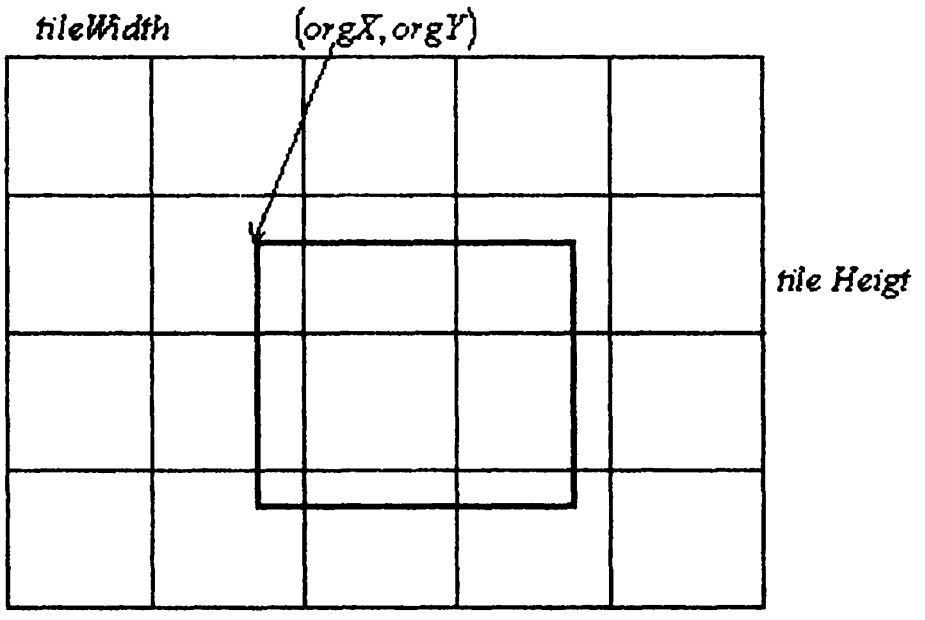


Рис. 4.8

При этом каждая клетка имеет размер *tile Width* на *tile Heigt* пикселов. Считаем, что верхний левый угол экрана (окна) имеет глобальные координаты (*orgX*, *orgY*). Простейший вариант реализации игры представлен ниже.

```
// File stategy.cpp
#include        "mouse.h"
#include        "array.h"
#include        "video.h"
#include        "image.h"
#include        "sprite.h"

char        screenMap [MAX_TILE_X][MAX_TILE_Y];
Image *     tiles [MAX_TILES];
Array *     sprites;
Sprite      mouse;
int         orgX = 0;
int         orgY = 0;
int         done = 0;

void    drawScreen ()
{
        int             i0 = orgX / tileWidth;
        int             i1 = (orgX + screenWidth - 1) / tileWidth;
        int             x0 = i0 * tileWidth - orgX;
        int             j0 = orgY / tileHeight;
        int             j1 = (orgY + screenHeight - 1) / tileHeight;
        int             y0 = j0 * tileHeight - orgY;

        for (int i = i0, x = x0; i <= i1; i++, x += tileWidth)
            for (int j = j0, y = y0; j <= j1; j++, y += tileHeight)
                    tiles [screenMap [i][j]] -> draw ( x, y );

        for ( i = 0; i < sprites -> getCount (); i++ )
```

```
        ( (Sprite *) sprites -> objectAt ( i ) ) -> draw ();
    mouse.draw ();
    swapBuffers ();
}

main ()
{
    MouseState mouseState;
    loadMap     ();
    loadTiles   ();
    loadSprites ();
    initVideo   ();
    resetMouse  ();
    for ( ; !done; )
    {
        performActions ();
        drawScreen     ();
        readMouseState ( mouseState );
        if ( mouseState.loc.x >= screenWidth - 2 &&
        orgX < maxOrgX )
            orgX++;
        else
        if ( mouseState.loc.x <= 2 && orgX > 0 )
            orgX--;
        if ( mouseState.loc.y >= screenHeight - 2 &&
            orgY < maxOrgY )
            orgY++;
        else
        if ( mouseState.loc.y <= 2 && orgY > 0 )
            orgY--;
        mouse.set ( mouseState.loc.x + orgX,
            mouseState.loc.y + orgY );
        if ( mouseState.buttons )
            handleMouse ( mouseState );
        handleKeyboard ();
    }
    doneVideo   ();
    freeTiles   ();
    freeSprites ();
}
```

Предполагается, что функция drawScreen выводит изображение в буфер и невидимую страницу и вызов функции swapBuffers делает построенное изобра видимым.

4.9.6. Нестандартные режимы адаптера VGA (X-режимы)

Для 256-цветных режимов существует еще один способ организации видеопамяти: 8 бит, отводимых под каждый пиксел, хранятся вместе, образуя 1 байт, но эти байты находятся на разных плоскостях видеопамяти.

Пиксел	*Адрес*	*Плоскость*
(0, 0)	0	0
(1, 0)	0	1
(2, 0)	0	2
(3, 0)	0	3
(4, 0)	1	0
(5, 0)	1	1
...	...	..
(x, y)	y * 80 + (x >> 2)	x & 3

В этом режиме сохраняются все свойства основных регистров и механизм их действия за исключением того, что изменяется интерпретация находящихся в видеопамяти значений. Режим позволяет за одну операцию менять до 4 пикселов сразу. Еще одним преимуществом этого режима является возможность работы с несколькими страницами видеопамяти, недоступная в стандартном 256-цветном режиме.

Ниже приводится программа, устанавливающая режим с разрешением 320 на 200 пикселов с использованием 256 цветов посредством изменения стандартного режима 13h, и иллюстрируется возможность работы сразу с четырьмя страницами.

```
// File example2.cpp
#include        <alloc.h>
#include        <conio.h>
#include        <mem.h>
#include        <stdio.h>
#include        "ega.h"

unsigned  pageBase = 0;
char      leftPlaneMask  [] = { 0x0F, 0x0E, 0x0C, 0x08 };
char      rightPlaneMask [] = { 0x01, 0x03, 0x07, 0x0F };
char far * font;

void  setX ()
{
        setVideoMode ( 0x13 );

        pageBase = 0xA000;

        writeReg ( EGA_SEQUENCER, 4, 6 );
        writeReg ( EGA_CRTC, 0x17, 0xE3 );
        writeReg ( EGA_CRTC, 0x14, 0 );

                                        // clear screen
        writeReg ( EGA_SEQUENCER, EGA_MAP_MASK, 0x0F );
        _fmemset ( MK_FP ( pageBase, 0 ), '\0', 0xFFFF );
}
```

```
void   setVisualPage ( int page )
{
       unsigned addr = page * 0x4000;
                                              // wait for vertical retrace
       while ( ( inportb ( 0x3DA ) & 0x08 ) == 0 )
                     ;
       writeReg ( EGA_CRTC, 0x0C, addr >> 8 );
       writeReg ( EGA_CRTC, 0xDC, addr & 0x0F );
}
void   setActivePage ( int page )
{
       pageBase = 0xA000 + page * 0x400;
}
void   writePixel ( int x, int y, int color )
{
       writeReg ( EGA_SEQUENCER, EGA_MAP_MASK, 1 << ( x & 3 ) );
       pokeb ( pageBase, y*80 + ( x >> 2 ), color );
       writeReg ( EGA_SEQUENCER, EGA_MAP_MASK, 0x0F );
}
int    readPixel ( int x, int y )
{
       writeReg ( EGA_GRAPHICS, EGA_READ_MAP_SELECT, x & 3 );
       return peekb ( pageBase, y*80 + ( x >> 2 ) );
}
void   bar ( int x1, int y1, int x2, int y2, int color )
{
       char far * videoPtr = (char far *) MK_FP ( pageBase,
                                                  y1*80 + (x1 >> 2) );
       char far * ptr      = videoPtr;
       int            cols    = ( x2 >> 2 ) - ( x1 >> 2 ) - 1;
       char           lmask   = leftPlaneMask [ x1 & 3 ];
       char           rmask   = rightPlaneMask [ x2 & 3 ];
       if ( cols < 0 )                        // both x1 & x2 are located
       {                                      // in the same byte
          writeReg ( EGA_SEQUENCER, EGA_MAP_MASK, lmask & rmask );
          for ( int y = y1; y <= y2; y++, videoPtr += 80 )
                                 *videoPtr = color;
          writeReg ( EGA_SEQUENCER, EGA_MAP_MASK, 0x0F );
       }
       else
       {
          writeReg ( EGA_SEQUENCER, EGA_MAP_MASK, lmask );
          for ( int y = y1; y <= y2; y++, videoPtr += 80 )
                                 * videoPtr = color;
          writeReg ( EGA_SEQUENCER, EGA_MAP_MASK, 0x0F );
```

```
                videoPtr = ++ptr;
                for ( y = y1; y <= y2; y++, videoPtr += 80 - cols )
                        for ( int x = 0; x < cols; x++ )
                                * videoPtr++ = color;
                writeReg ( EGA_SEQUENCER, EGA_MAP_MASK, rmask );
                videoPtr = ptr + cols;
                for ( y = y1; y <= y2; y++, videoPtr += 80 )
                        * videoPtr = color;
        }
        writeReg ( EGA_SEQUENCER, EGA_MAP_MASK, 0x0F );
}
void    getImage ( int x, int y, int width, int height, void * buf )
{
        char far * videoPtr;
        char        * dataPtr = (char *) buf;
        for ( int i = x; i < x + width; i++ )
        {
           videoPtr = (char far *) MK_FP (pageBase, (i >> 2)+y*80);
           writeReg ( EGA_GRAPHICS, EGA_READ_MAP_SELECT, i & 3 );
           for ( int j = 0; j < height; j++, videoPtr += 80 )
                        * dataPtr ++ = * videoPtr;
        }
}
void    putImage ( int x, int y, int width, int height, void * buf )
{
        char far * videoPtr;
        char        * dataPtr = (char *) buf;
        for ( int i = x; i < x + width; i++ )
        {
           videoPtr = (char far *) MK_FP (pageBase, (i >> 2)+y*80);
           writeReg ( EGA_SEQUENCER, EGA_MAP_MASK, 1 << ( i & 3 ) );
           for ( int j = 0; j < height; j++, videoPtr += 80 )
                        * videoPtr = * dataPtr++;
        }
}
void    drawString ( int x, int y, char * str, int color )
{
        for ( ; *str != '\0'; str++, x+= 8 )
                for ( int j = 0; j < 16; j++ )
                {
                        char        byte = font [16 * (*str) + j];
                        for ( int i = 0; i < 8; i++, byte <<= 1 )
                                if ( byte & 0x80 )
                                        writePixel (x+i, y+j, color);
                }
}
main ()
```

```
if ( !findVGA () )
{
        printf ( "\nVGA compatible card not found." );
        return -1,
}
void    * buf = malloc ( 100*50 );
if ( buf == NULL )
{
        printf ( "\nmalloc failure." );
        return -1;
}
setX ();                // set 320x200 256 colors X-mode
font = findROMFont ( 16 );
for ( int i = 0; i < 256; i++ )
        writePixel ( i, 0, i );
for ( i = 5; i < 140; i++ )
        bar ( 2*i, i, 2*i+30, i+30, i );
getImage   ( 1, 1, 100, 50, buf );
drawString ( 110, 100, "Page 0", 70 );
getch ();
setActivePage ( 1 );
setVisualPage ( 1 );
bar ( 10, 20, 300, 200, 33 );
drawString ( 110, 100, "Page 1", 75 );
getch ();
setActivePage ( 2 );
setVisualPage ( 2 );
bar ( 10, 20, 300, 200, 39 );
drawString ( 110, 100, "Page 2", 80 );
getch ();
setActivePage ( 3 );
setVisualPage ( 3 );
bar ( 10, 20, 300, 200, 44 );
drawString ( 110, 100, "Page 3", 85 );
getch ();
setVisualPage ( 0 );
setActivePage ( 0 );
getch ();
putImage ( 151, 3, 100, 50, buf );
getch ();
setVisualPage ( 1 );
getch ();
setVisualPage ( 2 );
getch ();
setVideoMode ( 3 );
```

Опишем процедуры, устанавливающие этот режим с нестандартными разрешеми 320 на 240 пикселов и 360 на 480 пикселов.

```
// File example3.cpp
#include        <alloc.h>
#include        <conio.h>
#include        <mem.h>
#include        <stdio.h>
#include        "Ega.h"

unsigned    pageBase = 0;
int         bytesPerLine;
char        leftPlaneMask  [] = { 0x0F, 0x0E, 0x0C, 0x08 };
char        rightPlaneMask [] = { 0x01, 0x03, 0x07, 0x0F };
char far *  font;

void  setX320x240 ()
{
      static int CRTCTable [] =
    {
              0x0D06,  // vertical total
              0x3E07,  // overflow (bit 8 of vertical counts)
              0x4109,  // cell height (2 to double-scan)
              0xEA10,  // vert sync start
              0xAC11,  // vert sync end and protect cr0-cr7
              0xDF12,  // vertical displayed
              0x0014,  // turn off dword mode
              0xE715,  // vert blank start
              0x0616,  // vert blank end
              0xE317   // turn on byte mode
    };
    setVideoMode ( 0x13 );

    pageBase    = 0xA000;
    bytesPerLine = 80;
    writeReg ( EGA_SEQUENCER, 4, 6 );
    writeReg ( EGA_CRTC, 0x17, 0xE3 );
    writeReg ( EGA_CRTC, 0x14, 0 );

    writeReg ( EGA_SEQUENCER, 0, 1 );   // synchronous reset
    outportb ( 0x3C2, 0xE3 );           // select 25 MHz dot clock
                                        // & 60 Hz scan rate
                                        // restart sequencer
    writeReg ( EGA_SEQUENCER, 0, 3 );
    writeReg ( EGA_CRTC, 0x11, ReadReg (EGA_CRTC, 0x11) & 0x7F );

    for ( int i = 0; i < sizeof (CRTCTable) / sizeof (int); i++ )
              outport ( EGA_CRTC, CRTCTable [i] );

                                        // clear screen
    writeReg ( EGA_SEQUENCER, EGA_MAP_MASK, 0x0F );
    _fmemset ( MK_FP ( pageBase, 0 ), '\0', 0xFFFF );
}

void  setX360x480 ()
{
```

```
 static int CRTCTable [] =
{
            0x6b00,
            0x5901,
            0x5A02,
            0x8E03,
            0x5E04,
            0x8A05,
            0x0D06,   // vertical total
            0x3E07,   // overflow (bit 8 of vertical counts)
            0x4009,   // cell height (2 to double-scan)
            0xEA10,   // vert sync start
            0xAC11,   // vert sync end and protect cr0-cr7
            0xDF12,   // vertical displayed
            0x2D13,
            0x0014,   // turn off dword mode
            0xE715,   // vert blank start
            0x0616,   // vert blank end
            0xE317    // turn on byte mode
 };
 setVideoMode ( 0x13 );
 pageBase     = 0xA000;
 bytesPerLine = 90;
      writeReg ( EGA_SEQUENCER, 4, 6 );
      writeReg ( EGA_CRTC, 0x17, 0xE3 );
      writeReg ( EGA_CRTC, 0x14, 0 );
      writeReg ( EGA_SEQUENCER, 0, 1 );   // synchronous reset
      outportb ( 0x3C2, 0xE7 );           // select 25 MHz dot clock
                                          // & 60 Hz scan rate
                                          // restart sequencer
      writeReg ( EGA_SEQUENCER, 0, 3 );
      writeReg ( EGA_CRTC, 0x11, ReadReg (EGA_CRTC, 0x11) & 0x7F );
      for ( int i = 0; i < sizeof (CRTCTable) / sizeof (int); i++ )
            outport ( EGA_CRTC, CRTCTable [i] );
                                          // clear screen
      writeReg ( EGA_SEQUENCER, EGA_MAP_MASK, 0x0F );
      _fmemset ( MK_FP ( pageBase, 0 ), '\0', 0xFFFF );
}
void  setVisualPage ( int page )
{
      unsigned addr = page * 0x4B00;
                                    // wait for vertical retrace
      while ( ( inportb ( 0x3DA ) & 0x08 ) == 0 )
            ;
      writeReg ( EGA_CRTC, 0x0C, addr >> 8 );
      writeReg ( EGA_CRTC, 0xDC, addr & 0x0F );
}
void  setActivePage ( int page )
```

```
{
        pageBase = 0xA000 + page * 0x4B0;
}
void    writePixel ( int x, int y, int color )
{
        writeReg ( EGA_SEQUENCER, EGA_MAP_MASK, 1 << ( x & 3 ) );
        pokeb ( pageBase, y * bytesPerLine + ( x >> 2 ), color );
        writeReg ( EGA_SEQUENCER, EGA_MAP_MASK, 0x0F );
}
int     readPixel ( int x, int y )
{
        writeReg ( EGA_GRAPHICS, EGA_READ_MAP_SELECT, x & 3 );
        return peekb ( pageBase, y * bytesPerLine + ( x >> 2 ) );
}
void    bar ( int x1, int y1, int x2, int y2, int color )
{
        char far * vptr  = (char far *) MK_FP ( pageBase,
                                        y1 * bytesPerLine + (x1 >> 2) );
        char far * ptr   = vptr;
        int             cols  = ( x2 >> 2 ) - ( x1 >> 2 ) - 1;
        char            lmask = leftPlaneMask [ x1 & 3 ];
        char            rmask = rightPlaneMask [ x2 & 3 ];
        if ( cols < 0 )         // both x1 & x2 are located in the same byte
        {
                writeReg (EGA_SEQUENCER,EGA_MAP_MASK,lmask & rmask);
                for (int y = y1; y <= y2; y++, vptr += bytesPerLine)
                        *vptr = color;
                writeReg ( EGA_SEQUENCER, EGA_MAP_MASK, 0x0F );
        }
        else
        {
                writeReg ( EGA_SEQUENCER, EGA_MAP_MASK, lmask );
                for (int y = y1; y <= y2; y++, vptr += bytesPerLine)
                        *vptr = color;
                writeReg ( EGA_SEQUENCER, EGA_MAP_MASK, 0x0F );
                vptr = ++ptr;
                for (y=y1; y<=y2; y++, vptr += bytesPerLine-cols )
                        for ( int x = 0; x < cols; x++ )
                                *vptr++ = color;
                writeReg ( EGA_SEQUENCER, EGA_MAP_MASK, rmask );
                vptr = ptr + cols;
                for ( y = y1; y <= y2; y++, vptr += bytesPerLine )
                        *vptr = color;
        }
```

```
        writeReg ( EGA_SEQUENCER, EGA_MAP_MASK, 0x0F );
}
void    drawString ( int x, int y, char * str, int color )
{
        for ( , *str != '\0'; str++, x+= 8 )
                for ( int j = 0; j < 16; j++ )
                {
                        char        byte = font [16 * (*str) + j];
                        for ( int i = 0; i < 8; i++, byte <<= 1 )
                            if ( byte & 0x80 )
                                dritePixel ( x+i, y+j, color );
                }
}
void    show360x480 ()
{
        RGB         pal [256];
        setX360x480 ();                 // set 320x480 256 colors X-mode
        for ( int i = 0; i < 256; i++ )
        {
            int    c = i & 0x40 ? i & 0x3F   0x3F - ( i & 0x3F );
            pal [i].red   = c;
            pal [i].green = c * c / 0x3F;
            pal [i].blue  = i & 0x80 ?
                    0x3F - ( i >> 1 ) & 0x3F   ( i >> 1 ) & 0x3F;
        }
        setPalette ( pal, 256 );
        for ( int x = 0; x < 180; x++ )
            for ( int y = 0; y < 240; y++ )
            {
                    unsigned long       x2    = ( x + 1 ) * (long)( 360 - x );
                    unsigned long       y2    = ( y + 1 ) * (long)( 480 - y );
                    int                 color = (int)((x2*x2)/y2/113);
                    writePixel ( x,      y,          color );
                    writePixel ( 359 -  x, y,        color );
                    writePixel ( x,      479 - y,    color );
                    writePixel ( 359 -  x, 479 - y, color );
            }
}
main ()
{
        if ( !findVGA () )
        {
                printf ( "\nVGA compatible card not found." );
                return -1;
        }
        setX320x240 ();                 // set 320x240 256 colors X-mode
        font = findROMFont ( 16 );
        for ( int j = 1; j < 220; j += 21 )
```

```
    for ( int i = 1; i < 300; i += 21 )
            bar ( i, j, i+20, j+20, ((j/21*15)+i/21) & 0xFF );
drawString ( 110, 100, "Page 0", 70 );
getch ();

setActivePage ( 1 );
setVisualPage ( 1 );
bar ( 10, 20, 300, 200, 33 );
drawString ( 110, 100, "Page 1", 75 );
getch ();

setActivePage ( 2 );
setVisualPage ( 2 );
bar ( 10, 20, 300, 200, 39 );
drawString ( 110, 100, "Page 2", 80 );
getch ();

setVisualPage ( 0 );
getch ();
setVisualPage ( 1 );
getch ();
setVisualPage ( 2 );
getch ();

show360x480 ();

getch ();
setVideoMode ( 3 );
}
```

4.10. Программирование SVGA-адаптеров

Существует большое количество видеокарт, хотя и совместимых с VGA, но предоставляющих достаточно большой набор дополнительных режимов. Обычно такие карты называют SuperVGA или SVGA. SVGA-карты различных производителей, весьма различаются по основным возможностям и, как правило, несовместимы друг с другом. Сам термин "SVGA" обозначает скорее не стандарт (как VGA), а некоторое его расширение.

Рассмотрим работу SVGA-адаптеров с 256-цветными режимами. Почти все они построены одинаково: под каждый пиксел отводится 1 байт и вся видеопамять разбивается на банки одинакового размера (обычно по 64 Кбайт), при этом область адресного пространства 0xA000:0-0xA000:0xFFFF соответствует выбранному банку. Ряд карт позволяет работать сразу с двумя банками.

При такой организации памяти процедура writePixel для карт с 64-килобайтовыми банками выглядит следующим образом:

```
void writePixel ( int x, int y, int color )
{
  long addr = bytesPerLine * (long)y + (long)x;
  setBank ( addr >> 16 );
  pokeb ( 0xA000, (unsigned)addr, color );
}
```

где функция setBank служит для установки банка с заданным номером.

Практически все различие между картами сводится к установке режима с заданным разрешеним и установке банка с заданным номером.

Ниже приводится пример программы, работающей с режимом 640 на 480 точек для SVGA Trident при 256 цветах. Функция findTrident служит для проверки того, что данный видеоадаптер действительно установлен.

```
// File Trident.Cpp
#include        <conio.h>
#include        <dos.h>

#define LOWORD(l)        ((int)(l))
#define HIWORD(l)        ((int)((l) >> 16))

static int      curBank = 0;

void  setTridentMode ( int mode )
{
        asm {
                mov     ax, mode
                int     10h

                mov     dx, 3CEh // set pagesize to 64k
                mov     al, 6
                out     dx, al
                inc     dx
                in      al, dx
                dec     dx
                or      al, 4
                mov     ah, al
                mov     al, 6
                out     dx, ax
                mov     dx, 3C4h  // set to BPS mode
                mov     al, 0Bh
                out     dx, al
                inc     dx
                in      al, dx
        }
}
void  setTridentBank ( int start )
{
        if ( start == curBank )
                return;

        curBank = start;
        asm {
                mov     dx, 3C4h
                mov     al, 0Bh
                out     dx, al
                inc     dx
                mov     al, 0
                out     dx, al
                in      al,dx
                dec     dx
```

```
                mov     al, 0Eh
                mov     ah, byte ptr start
                xor     ah, 2
                out     dx, ax
        }
}
void    writePixel ( int x, int y, int color )
{
        long    addr = 640l * (long)y + (long)x;
        setTridentBank ( HIWORD ( addr ) );
        pokeb ( 0xA000, LOWORD ( addr ), color );
}
main ()
{
        setTridentMode ( 0x5D );        // 640x480x256
        for ( int i = 0; i < 640; i++ )
                for ( int j = 0; j < 480; j++ )
                        writePixel ( i, j, ((i/20)+1)*(j/20+1) );
        getch ();
}
```

Аналогичный пример для SVGA Cirrus Logic выглядит следующим образом:

```
// File Cirrus.Cpp
#include        <conio.h>
#include        <dos.h>
#include        <process.h>
#include        <stdio.h>
#define LOWORD(l)       ((int)(l))
#define HIWORD(l)       ((int)((l) >> 16))
inline void     writeReg ( int base, int reg, int value )
{
        outportb ( base,    reg );
        outportb ( base + 1, value );
}
inline char     readReg ( int base, int reg )
{
        outportb ( base, reg );
        return inportb ( base + 1 );
}
static int      curBank = 0;
// check bits specified by mask in port for being
// readable/writable
int     testPort ( int port, char mask )
{
        char    save = inportb ( port );
        outportb ( port, save & ~mask );
```

```
        char        v1 = inportb ( port ) & mask;
        outportb ( port, save | mask );
        char        v2 = inportb ( port ) & mask;
        outportb ( port, save );
        return v1 == 0 && v2 == mask;
}
int     testReg ( int port, int reg, char mask )
{
        outportb ( port, reg );
        return testPort ( port + 1, mask );
}
int     findCirrus ()
{
        char        save = readReg ( 0x3C4, 6 );
        int         res  = 0;
        writeReg ( 0x3C4, 6, 0x12 );   // enable extended registers
        if ( readReg ( 0x3C4, 6 ) == 0x12 )
                if ( testReg ( 0x3C4, 0x1E, 0x3F ) &&
                     testReg ( 0x3D4, 0x1B, 0xFF ) )
                        res = 1;
        writeReg ( 0x3C4, 6, save );
        return res;
}
void    setCirrusMode ( int mode )
{
        asm {
                mov     ax, mode
                int     10h
                mov     dx, 3C4h  // enable extended registers
                mov     al, 6
                out     dx, al
                inc     dx
                mov     al, 12h
                out     dx, al
        }
}
void    setCirrusBank ( int start )
{
        if ( start == curBank )
                return;
        curBank = start;
        asm {
                mov     dx, 3CEh
                mov     al, 9
                mov     ah, byte ptr start
                mov     cl, 4
                shl     ah, cl
```

```
                out         dx, ax
        }
}
void   writePixel ( int x, int y, int color )
{
        long        addr = 640l * (long)y + (long)x;
        setCirrusBank ( HIWORD ( addr ) );
        pokeb ( 0xA000, LOWORD ( addr ), color );
}
main ()
{
        if ( !findCirrus () )
        {
                printf ( "\nCirrus card not found" );
                exit ( 1 );
        }
        setCirrusMode ( 0x5F );          // 640x480x256
        for ( int i = 0; i < 640; i++ )
                for ( int j = 0; j < 480; j++ )
                        writePixel ( i, j, ((i/20)+1)*(j/20+1) );
        getch ();
}
```

Тем самым можно построить библиотеку, обеспечивающую работу с основными SVGA-картами. Сильная привязанность подобной библиотеки к конкретному набору карт - ее главный недостаток.

Ассоциацией стандартов в области видеоэлектроники - VESA (Video Electronic Standarts Association) была сделана попытка стандартизации работы с различными SVGA-платами путем добавления в BIOS-платы некоторого стандартного набора функций, обеспечивающего получение необходимой информации о карте, установку заданного режима и банка памяти. При этом также вводится стандартный набор расширенных режимов. Номер режима является 16-битовым числом, где биты 9-15 зарезервированы и должны быть равны нулю, бит 8 для VESA-режимов равен единице, а для родных режимов карты - нулю.

Приведем таблицу основных VESA-режимов.

Номер	*Разрешение*	*Бит на пиксел*	*Количество цветов*
100h	640×400	8	256
101h	640×480	8	256
102h	800×600	4	16
103h	800×600	8	256
104h	1024×768	4	16
105h	1024×768	8	256
106h	1280×1024	4	16
107h	1280×1024	8	256
10Dh	320×200	15	32 K

10Eh	320×200	16	64 K
10Fh	320×200	24	16 M
110h	640×480	15	32 K
111h	640×480	16	64 K
112h	640×480	24	16 M
113h	800×600	15	32 K
114h	800×600	16	64 K
115h	800×600	24	16 M
116h	1024×768	15	32 K
117h	1024×768	16	64 K
118h	1024×768	24	16 M
119h	1280×1024	15	32 K
11Ah	1280×1024	16	64 K
11Bh	1280×1024	24	16 M

Ниже приводятся файлы, содержащие необходимые структуры и функци работы с VESA-совместимыми адаптерами.

```
// File Vesa.H
#ifndef         __VESA__
#define         __VESA__
                                        // 256-color modes
#define         VESA_640x400x256        0x100
#define         VESA_640x480x256        0x101
#define         VESA_800x600x256        0x103
#define         VESA_1024x768x256       0x105
#define         VESA_1280x1024x256      0x107
                                        // 32K color modes
#define         VESA_320x200x32K        0x10D
#define         VESA_640x480x32K        0x110
#define         VESA_800x600x32K        0x113
#define         VESA_1024x768x32K       0x116
#define         VESA_1280x1024x32K      0x119
                                        // 64K color modes
#define         VESA_320x200x64K        0x10E
#define         VESA_640x480x64K        0x111
#define         VESA_800x600x64K        0x114
#define         VESA_1024x768x64K       0x117
#define         VESA_1280x1024x64K      0x11A
                                        // 16M color mode
#define         VESA_320x200x16M        0x10F
#define         VESA_640x480x16M        0x112
#define         VESA_800x600x16M        0x115
#define         VESA_1024x768x16M       0x118
#define         VESA_1280x1024x16M      0x11B

struct VESAInfo
{
```

```
        char        sign [4]; // 'VESA' signature
        int         version; // VESA BIOS version
        char far * OEM;                 // Original Equipment Manufacturer
        long        capabilities;
        int far  * modeList;   // list of supported modes
        int         totalMemory;        // total memory on board
                                        // in 64Kb blocks
        char        reserved [236];
};
struct VESAModeInfo
{
        int         modeAttributes;
        char        winAAttributes;
        char        winBAttributes;
        int         winGranularity;
        int         winSize;
        unsigned    winASegment;
        unsigned    winBSegement;
        void far *  winFuncPtr;
        int         bytesPerScanLine;
                                        // optional data
        int         xResolution;
        int         yResolution;
        char        xCharSize;
        char        yCharSize;
        char        numberOfPlanes;
        char        bitsPerPixel;
        char        numberOfBanks;
        char        memoryModel;
        char        bankSize;
        char        numberOfPages;
        char        reserved;
                                        // direct color fields
        char        redMaskSize;
        char        redFieldPosition;
        char        greenMaskSize;
        char        greenFieldPosition;
        char        blueMaskSize;
        char        blueFieldPosition;
        char        rsvdMaskSize;
        char        rsvdFieldPosition;
        char        directColorModeInfo;
        char        resererved2 [216];
};
int     findVESA ( VESAInfo& );
int     findVESAMode ( int, VESAModeInfo& );
int     setVESAMode ( int );
int     getVESAMode ();
void    setVESABank ();
#endif
```

```
// File Vesa.cpp
// test for VESA
#include        <conio.h>
#include        <dos.h>
#include        <process.h>
#include        <stdio.h>
#include        <string.h>
#include        "Vesa.h"

#define LOWORD(l)        ((int)(l))
#define HIWORD(l)        ((int)((l) >> 16))

static int                      curBank     = 0;
static int                      granularity = 1;
static VESAModeInfo     curMode;

int     findVESA ( VESAInfo& vi )
{
#if defined(__COMPACT__) || defined(__LARGE__) || defined(__HUGE__)
        asm {
                push    es
                push    di
                les     di, dword ptr vi
                mov     ax, 4F00h
                int     10h
                pop     di
                pop     es
        }
#else
        asm {
                push    di
                mov     di, word ptr vi
                mov     ax, 4F00h
                int     10h
                pop     di
        }
#endif
        if ( _AX != 0x004F )
                return 0;

        return !strncmp ( vi.sign, "VESA", 4 );
}

int     findVESAMode ( int mode, VESAModeInfo& mi )
{
#if defined(__COMPACT__) || defined(__LARGE__) || defined(__HUGE__)
        asm {
                push    es
                push    di
                les     di, dword ptr mi
                mov     ax, 4F01h
                mov     cx, mode
                int     10h
                pop     di
```

```
                pop     es
        }
#else
        asm {
                push    di
                mov     di, word ptr mi
                mov     ax, 4F01h
                mov     cx, mode
                int     10h
                pop     di
        }
#endif
        return _AX == 0x004F;
}
int     setVESAMode ( int mode )
{
        if ( !findVESAMode ( mode, curMode ) )
                return 0;
        granularity = 64 / curMode.winGranularity;
        asm {
                mov     ax, 4F02h
                mov     bx, mode
                int     10h
        }
        return _AX == 0x004F;
}
int     getVESAMode ()
{
        asm {
                mov     ax, 4F03h
                int     10h
        }
        if ( _AX != 0x004F )
                return 0;
        else
                return _BX;
}
void    setVESABank ( int start )
{
        if ( start == curBank )
                return;
        curBank = start;
        start  *= granularity;
        asm {
                mov     ax, 4F05h
                mov     bx, 0
                mov     dx, start
                push    dx
```

```
                int     10h
                mov     bx, 1
                pop     dx
                int     10h
        }
}
void   writePixel ( int x, int y, int color )
{
        long    addr = (long)curMode.bytesPerScanLine * (long)y +
                        (long)x;
        setVESABank ( HIWORD ( addr ) );
        pokeb ( 0xA000, LOWORD ( addr ), color );
}
main ()
{
        VESAInfo  vi;
        if ( !findVESA ( vi ) )
        {
                printf ( "\nVESA VBE not found." );
                exit ( 1 );
        }
        if ( !setVESAMode ( VESA_640x480x256 ) )
                exit ( 1 );
        for ( int i = 0; i < 640; i++ )
                for ( int j = 0; j < 480; j++ )
                        writePixel ( i, j, ((i/20)+1)*(j/20+1) );
        getch ();
}
```

При помощи функции findVESA можно получить информацию о нали BIOS, а также узнать все режимы, доступные для данной карты.

Функция findVESAMode возвращает информацию о режиме в поля рыVESAModeInfo.

Укажем наиболее важные поля.

Поле	*Размер в байтах*	*Комментарий*
modeAttributes	2	Характеристики режима: бит 0 - режим доступен, бит 1 - режим зарезервирован, бит 2 - BIOS поддерживает вывод в этом бит 3 - режим цветной, бит 4 - режим графический
winBAttributes	1	Характеристики банка B
winGranularity	2	Шаг установки банка в килобайтах
winSize	2	Размер банка

winAAttributes	1	Характеристики банка A: бит 0 - банк поддерживается, бит 1 - из банка можно читать, бит 2 - в банк можно писать
winASegment	2	Сегментный адрес банка A
winBSegment	2	Сегментный адрес банка B
bytesPerScanLine	2	Количество байт под одну строку
bitsPerPixel	1	Количество бит, отводимых под 1 пиксел
numberOfBanks	1	Количество банков памяти

Приведем программу, выдающую информацию по всем доступным VE режимам.

```
// File VesaInfo.cpp
#include «vesa.h»
char * colorInfo ( int bits )
{
 switch ( bits )
 {
  case 4:   return "16 colors";
  case 8:   return "256 colors";
  case 15:  return "32K colors ( HiColor )";
  case 16:  return "64K colors ( HiColor )";
  case 24:  return "16M colors ( TrueColor )";
  default:  return "";
 }
}
void  dumpMode ( int mode )
{
 VESAModeInfo mi;
 if ( !findVESAMode ( mode, mi ) )
          return;
 if ((mi.modeAttributes & 1) == 0)
          return; // not available now
 printf ( "\n %4X %10s %4dx %4d %2d %s", mode,
    mi.modeAttributes & 0x10 ? "Graphics" : "Text",
    mi.xResolution, mi.yResolution, mi.bitsPerPixel,
    colorInfo ( mi.bitsPerPixel ) );
}
main ()
{
 VESAInfo Info;
 char  str [256];
 if ( !findVESA ( Info ) )
 {
  printf ( "VESA VBE not found" );
  exit ( 1 );
 }
 _fstrcpy ( str, Info.OEM );
 printf ("\nVESA VBE version %d.%d\nOEM: %s\nTotal memory: "
```

```
        "%dKb\n", Info.version >> 8, Info.version & 0xFF, str,
        Info.totalMemory * 64 );
  for ( int i = 0; Info.modeList [i] != -1, i++ )
    dumpMode ( Info.modeList [i] );
}
```

4.10.1. Непалитровые режимы адаптеров SVGA

Ряд SVGA-карт поддерживают использование так называемых непалитровых режимов - для каждого пиксела вместо индекса в палитре непосредственно задается его RGB-значение.

Обычно такими режимами являются режимы HiColor (15 или 16 бит на пиксел) и **TrueColor (24 или 32 бита на пиксел).**

Видеопамять для этих режимов устроена аналогично 256-цветным режимам SVGA - под каждый пиксел отводится целое количество байт памяти (2 байта для HiColor и 3 или 4 байта для TrueColor), все они расположены подряд и сгруппированы **в банки.**

Наиболее простой является организация режима TrueColor (16 млн цветов) - под каждую из трех компонент цвета отводится по 1 байту.

Для удобства работы ряд карт отводит по 4 байта на 1 пиксел, при этом старший байт игнорируется. Таким образом, память для 1 пиксела устроена так:

rrrrrrrrggggggggbbbbbbbb

или

00000000rrrrrrrrggggggggbbbbbbbb

Несколько сложнее организация режимов HiColor, где под каждый пиксел отводится по 2 байта и возможны два варианта:

- под каждую компоненту отводится по 5 бит, последний бит не используется (32 тыс. цветов);
- под красную и синюю компоненты отводится по 5 бит, под зеленую – 6 бит (64 тыс. цветов).

Соответствующая этим режимам раскладка памяти выглядит следующим образом:

0rrrrrgggggbbbbb

или

rrrrrgggggbbbbb

Замечание. В связи с некорректной работой Windows 95/98 с непалитровыми режимами некоторые из приведенных далее примеров следует запускать в DOS-режиме.

Ниже приводится простая программа, иллюстрирующая работу с режимом HiColor 32 тыс. цветов.

```
// File HiColor.cpp
// tes. for VESA
#include        <conio.h>
#include        <dos.h>
#include        <stdio.h>
#include        <string.h>
```

```
#include        "Vesa.h"

#define LOWORD(l)        ((int)(l))
#define HIWORD(l)        ((int)((l) >> 16))

inline int      RGBColor ( int red, int green, int blue )
{
        return ( ( red >> 3 ) << 10 ) | ( ( green >> 3 ) << 5 ) |
          ( blue >> 3 );
}

static int                      curBank    = 0;
static int                      granularity = 1;
static VESAModeInfo     curMode;

int     findVESA ( VESAInfo& vi )
{
#if defined(__COMPACT__) || defined(__LARGE__) || defined(__HUGE__)
        asm {
                push    es
                push    di
                les     di, dword ptr vi
                mov     ax, 4F00h
                int     10h
                pop     di
                pop     es
        }
#else
        asm {
                push    di
                mov     di, word ptr vi
                mov     ax, 4F00h
                int     10h
                pop     di
        }
#endif
        if ( _AX != 0x004F )
                return 0;

        return !strncmp ( vi.sign, "VESA", 4 );
}

int     findVESAMode ( int mode, VESAModeInfo& mi )
{
#if defined(__COMPACT__) || defined(__LARGE__) || defined(__HUGE__)
        asm {
                push    es
                push    di
                les     di, dword ptr mi
                mov     ax, 4F01h
                mov     cx, mode
                int     10h
                pop     di
                pop     es
        }
```

```
#else
        asm {
                push    di
                mov     di, word ptr mi
                mov     ax, 4F01h
                mov     cx, mode
                int     10h
                pop     di
        }
#endif
        return _AX == 0x004F;
}
int     setVESAMode ( int mode )
{
        if ( !findVESAMode ( mode, curMode ) )
                return 0;
        granularity = 64 / curMode.winGranularity;
        asm {
                mov     ax, 4F02h
                mov     bx, mode
                int     10h
        }
        return _AX == 0x004F;
}
int     getVESAMode ()
{
        asm {
                mov     ax, 4F03h
                int     10h
        }
        if ( _AX != 0x004F )
                return 0;
        else
                return _BX;
}
void    setVESABank ( int start )
{
        if ( start == curBank )
                return;
        curBank = start;
        start  *= granularity;
        asm {
                mov     ax, 4F05h
                mov     bx, 0
                mov     dx, start
                push    dx
                int     10h
                mov     bx, 1
```

```
                pop     dx
                int     10h
        }
}
void   writePixel ( int x, int y, int color )
{
        long    addr = (long)curMode.bytesPerScanLine * (long)y +
                        (long)(x<<1);
        SetVESABank ( HIWORD ( addr ) );
        poke ( 0xA000, LOWORD ( addr ), color );
}
main ()
{
        VESAInfo info;
        if ( !findVESA ( info ) )
        {
                printf ( "VESA VBE not found" );
                return 1;
        }
        if ( !setVESAMode ( VESA_640x480x32K ) )
        {
                printf ( "Mode not supported" );
                return 1;
        }
        for ( int i = 0; i < 256; i++ )
           for ( int j = 0; j < 256; j++ )
           {
                writePixel ( 320-i, 240-j, RGBColor ( 0,j,i) );
                writePixel ( 320+i, 240-j, RGBColor ( i,j,i) );
                writePixel ( 320+i, 240+j, RGBColor ( j,i,i) );
                writePixel ( 320-i, 240+j, RGBColor ( j,0,i) );
           }
        getch ();
}
```

4.10.2. Стандарт VBE 2.0 (VESA BIOS Extension 2.0)

Одним из существенных неудобств при работе с SVGA-адаптерами является не-ходимость все время отслеживать границы банков памяти и осуществлять их пе-ключение. Было бы гораздо удобнее иметь в своем распоряжении линейный блок мяти необходимой длины и работать с ним безо всякого переключения банков (но гда приложение для работы с линейным блоком памяти необходимой длины лжно использовать 32-разрядную адресацию).

Стандарт VBE 2.0 как раз и предоставляет такую возможность для 32-разрядных иложений, применяющих защищенный режим процессора (DPMI32-приложения). структуру VESAModeInfo включено несколько дополнительных полей, одним из

которых является физический адрес линейного блока видеопамяти. Используя функции DPMI (DOS Protected Mode Interface), можно преобразовать этот физический адрес в указатель на линейный блок видеопамяти и работать с ним, уже не думая о банках видеопамяти.

Поскольку приложение должно идти в защищенном режиме процессора, а прерывание 10h рассчитано на работу в реальной режиме, необходимо соответствующим образом модифицировать основные функции для работы с этим прерыванием там, где происходит передача адресов и заполнение структур информацией. Текст соответствующего модуля приводится ниже. Он осуществляет всю необходимую работу по получению линейного адреса видеопамяти, получению информационных структур, преобразованию указателей в этих структурах, используя стандарты VBE 2.0 и DPMI.

```
// File vesa.h
#ifndef     __VESA__
#define     __VESA__

                                    // 256-color modes
#define     VESA_640x400x256        0x100
#define     VESA_640x480x256        0x101
#define     VESA_800x600x256        0x103
#define     VESA_1024x768x256       0x105
#define     VESA_1280x1024x256      0x107
                                    // 32K color modes
#define     VESA_320x200x32K        0x10D
#define     VESA_640x480x32K        0x110
#define     VESA_800x600x32K        0x113
#define     VESA_1024x768x32K       0x116
#define     VESA_1280x1024x32K      0x119
                                    // 64K color modes
#define     VESA_320x200x64K        0x10E
#define     VESA_640x480x64K        0x111
#define     VESA_800x600x64K        0x114
#define     VESA_1024x768x64K       0x117
#define     VESA_1280x1024x64K      0x11A
                                    // 16M color mode
#define     VESA_320x200x16M        0x10F
#define     VESA_640x480x16M        0x112
#define     VESA_800x600x16M        0x115
#define     VESA_1024x768x16M       0x118
#define     VESA_1280x1024x16M      0x11B

struct VESAInfo
{
    char    vbeSign [4]; // 'VESA' signature
    short   version;     // VESA BIOS version
    char  * OEM;                    // Original Equipment Manufactureer
    long    capabilities;
    short * modeListPtr;            // list of supported modes
    short       totalMemory;        // total memory on board
                                    // in 64Kb blocks
    short   OEMSoftwareRev;// OEM software revision
```

```
        char   * OEMVendorName;
        char   * OEMProductName;
        char   * OEMProductRev;
        char         reserved [222];
        char         OEMData [256];
};
struct VESAModeInfo
{
        unsigned short    modeAttributes;
        char              winAAttributes;
        char              winBAttributes;
        unsigned short    winGranularity;
        unsigned short    winSize;
        unsigned short    winASegment;
        unsigned short    winBSegement;
        void         *    winFuncPtr;
        unsigned short    bytesPerScanLine;
        unsigned short    xResolution;
        unsigned short    yResolution;
        char              xCharSize;
        char              yCharSize;
        char              numberOfPlanes;
        char              bitsPerPixel;
        char              numberOfBanks;
        char              memoryModel;
        char              bankSize;
        char              numberOfPages;
        char              reserved;
                                  // direct color fields
        char              redMaskSize;
        char              redFieldPosition;
        char              greenMaskSize;
        char              greenFieldPosition;
        char              blueMaskSize;
        char              blueFieldPosition;
        char              rsvdMaskSize;
        char              rsvdFieldPosition;
        char              directColorModeInfo;
        void         *    physBasePtr;
        void         *    offScreenMemoOffset;
        unsigned short    offScreenMemSize;
        char              resererved2 [206];
};
extern      VESAInfo    *       vbeInfoPtr;
extern      VESAModeInfo *      vbeModeInfoPtr;
extern      void        *       lfbPtr;
extern      int                 bytesPerLine;
extern      int                 bitsPerPixel;

int         initVBE2 ();
void        doneVBE2 ();
int         findVESAMode ( int );
```

```
int             setVESAMode ( int );
int             getVESAMode ();
#endif

// File vesa.cpp
#include        <i86.h>
#include        <malloc.h>
#include        <string.h>
#include        "vesa.h"

                // DPMI strctures
struct DPMIDosPtr
{
        unsigned short segment;
        unsigned short selector;
};

struct
{
        unsigned long       edi;
        unsigned long       esi;
        unsigned long       ebp;
        unsigned long       esp;
        unsigned long       ebx;
        unsigned long       edx;
        unsigned long       ecx;
        unsigned long       eax;
        unsigned short      flags;
        unsigned short      es, ds, fs, gs, ip, cs, sp, ss;
} rmRegs;

static  DPMIDosPtr    vbeInfoPool = { 0, 0 },
static  DPMIDosPtr    vbeModePool = { 0, 0 };
static  REGS          regs;
static  SREGS         sregs;

VESAInfo     * vbeInfoPtr;
VESAModeInfo * vbeModeInfoPtr;
void         * lfbPtr = NULL;
int              bytesPerLine;
int              bitsPerPixel;

inline  void  * RM2PMPtr ( void * ptr )          // convert pointer
                                                 // from real mode to
{                                                // protected mode ptr
        return (void *) ( ( ( ( (unsigned long) ptr ) >> 16 ) << 4 )+
                ( (unsigned long) ptr ) & 0xFFFF );
}

void    DPMIAllocDosMem ( DPMIDosPtr& ptr, int paras )
{
        regs.w.ax = 0x0100;             // allocate DOS memory
        regs.w.bx = paras;              // # of memory in paragraphs

        int386 ( 0x31, &regs, &regs );
                                        // addr. of block:
        ptr.segment  = regs.w.ax;       // real-mode segment of block
```

```
        ptr.selector = regs.w.dx;          // selector of block
}
void    DPMIFreeDosMem ( DPMIDosPtr& ptr )
{
        regs.w.ax = 0x0101;                // free DOS memory
        regs.w.dx = ptr.selector;

        int386 ( 0x31, &regs, &regs );
}
void  * DPMIMapPhysical ( void * ptr, unsigned long size )
{
        regs.w.ax = 0x0800;                // map physical memory
        regs.w.bx = (unsigned short)(((unsigned long) ptr) >> 16);
        regs.w.cx = (unsigned short)(((unsigned long) ptr)&0xFFFF);
        regs.w.si = (unsigned short)( size >> 16 );
        regs.w.di = (unsigned short)( size & 0xFFFF );

        int386 ( 0x31, &regs, &regs );

        return (void *) ( ( regs.w.bx << 16 ) + regs.w.cx );
}
void    DPMIUnmapPhysical ( void * ptr )
{
        regs.w.ax = 0x0801;                // free physical address
                                           // mapping
        regs.w.bx = (unsigned short)(((unsigned long) ptr) >> 16 );
        regs.w.cx = (unsigned short)(((unsigned long) ptr)&0xFFFF);

        int386 ( 0x31, &regs, &regs );
}
void    RMVideoInt ()          // execute real-mode
{                              // video interrupt
        regs.w.ax  = 0x0300;
        regs.w.bx  = 0x0010;
        regs.w.cx  = 0;
        regs.x.edi = FP_OFF ( &rmRegs );
        sregs.es   = FP_SEG ( &rmRegs );

        int386x ( 0x31, &regs, &regs, &sregs );
}
initVBE2 ()
{
        memset ( &regs,   '\0', sizeof ( regs ) );
        memset ( &sregs,  '\0', sizeof ( sregs ) );
        memset ( &rmRegs, '\0', sizeof ( rmRegs ) );

        DPMIAllocDosMem ( vbeInfoPool, 512 / 16 );
        DPMIAllocDosMem ( vbeModePool, 256 / 16 );

        vbeInfoPtr     = (VESAInfo *)(vbeInfoPool.segment * 16);
        vbeModeInfoPtr = (VESAModeInfo *)(vbeModePool.segment * 16);

        memset  ( vbeInfoPtr, '\0', sizeof ( VESAInfo ) );
        strncpy ( vbeInfoPtr -> vbeSign, "VBE2", 4 );
```

```
        rmRegs.eax = 0x4F00;
        rmRegs.es  = vbeInfoPool.segment;
        rmRegs.edi = 0;

        RMVideoInt ();

        if ( rmRegs.eax != 0x4F )
                return 0;

        vbeInfoPtr -> OEM          = (char *)  RM2PMPtr (
                                                vbeInfoPtr->OEM );
        vbeInfoPtr -> modeListPtr  = (short *) RM2PMPtr (
                                                vbeInfoPtr->modeListPtr );
        vbeInfoPtr -> OEMVendorName  = (char *)  RM2PMPtr (
                                                vbeInfoPtr->OEMVendorNa
        vbeInfoPtr -> OEMProductName = (char *)  RM2PMPtr (
                                                vbeInfoPtr->OEMProductN
        vbeInfoPtr -> OEMProductRev  = (char *)  RM2PMPtr (
                                                vbeInfoPtr->OEMProductR

        if ( strncmp ( vbeInfoPtr -> vbeSign, "VESA", 4 ) )
                return 0;

        if ( vbeInfoPtr -> version >= 0x0200 &&
          vbeInfoPtr -> OEMVendorName == NULL )
                vbeInfoPtr -> version = 0x0102;

        return vbeInfoPtr -> version >= 0x200;
}

void   doneVBE2 ()
{
        if ( lfbPtr != NULL )                   // unmap mapped LFB
                DPMIUnmapPhysical ( lfbPtr );

        DPMIFreeDosMem ( vbeInfoPool );         // free allocated
                                                // Dos memory
        DPMIFreeDosMem ( vbeModePool );

        setVESAMode ( 3 );
}

findVESAMode ( int mode )
{
        rmRegs.eax = 0x4F01;                    // set up registers
        rmRegs.ecx = mode;                      // for RM interrupt
        rmRegs.es  = vbeModePool.segment;
        rmRegs.edi = 0;

        RMVideoInt ();                          // execute video
                                                // interrupt

        return ( rmRegs.eax & 0xFFFF ) == 0x4F;   // check for validity
}

setVESAMode ( int mode )
{
        if ( mode == 3 )
        {
```

```
            rmRegs.eax = 3;
            RMVideoInt ();
            return 1;
}
if ( !FindVESAMode ( mode ) )// retrieve
            return 0;                           // VESAModeInfo
rmRegs.eax = 0x4F02;
rmRegs.ebx = mode | 0x4000;// ask foir LFB
RMVideoInt ();                                  // execute video
                                                // interrupt
if ( ( rmRegs.eax & 0xFFFF ) != 0x4F )
            return 0;
if ( lfbPtr != NULL )                           // unmap previously
                                                        // mapped memory
            DPMIUnmapPhysical ( lfbPtr );
                                                        // map linear
                                                        // frame buffer
lfbPtr = DPMIMapPhysical ( vbeModeInfoPtr -> physBasePtr,
                  (long) vbeInfoPtr -> totalMemory*64*1024 );
bytesPerLine = vbeModeInfoPtr -> bytesPerScanLine;
bitsPerPixel = vbeModeInfoPtr -> bitsPerPixel;
return 1;
}
```

```
// File vbe2test.cpp
#include        "vesa.h"
#include        <conio.h>
#include        <stdio.h>
void   writePixel ( int x, int y, int color )
{
        * ( x + y * bytesPerLine + (char *)lfbPtr ) = (char) color;
}
main ()
{
        if ( !initVBE2 () )
        {
                printf ( "\nVBE2 not found." );
                return 1;
        }
        printf ("\nSign:%s",          vbeInfoPtr->vbeSign );
        printf ("\nVersion %04x",     vbeInfoPtr->version );
        printf ("\nOEM:%s",           vbeInfoPtr->OEM );
        printf ("\nOEMVendorName: %s", vbeInfoPtr->OEMVendorName );
        printf ("\nOEMProductName: %s",vbeInfoPtr->OEMProductName );
        printf ( "\nOEMProductRev: %s",vbeInfoPtr->OEMProductRev );
        getch ();
```

```
    if ( !setVESAMode ( VESA_640x480x256 ) )
    {
            printf ( "\nError SetVESAMode." );
            return 1;
    }
    for ( int i = 0; i < 640; i++ )
            for ( int j = 0; j < 480; j++ )
                    writePixel ( i, j, (i/20 + 1)*(j/20 + 1) );
    getch ();
    doneVBE2 ();
    return 0;
}
```

Поскольку не все существующие карты поддерживают стандарт VBE 2.0, имеется специальная резидентная программа UNIVBE, прилагаемая на компакт-диске, которая осуществляет поддержку этого стандарта для большинства существующих видеокарт.

Ниже приводится файл surface.h, реализующий класс Surface (файлы surface.cpp, vesasurf.h и vesasurf.cpp прилагаются на компакт-диске). Этот класс отвечает за работу с растровым изображением (в том числе и хранящимся в видеопамяти) и реализует целый ряд дополнительных функций.

Для непосредственной работы с видеопамятью существуют класс VESASurface, позволяющий выбирать режим с заданным разрешением и числом бит на пиксел. Этот класс наследует от класса Surface все основные методы по реализации графических операций.

```
// File surface.h
//
// Basic class for images and all 2D graphics
// You can add your own functions
// currently supports only 8-bit and 15/16-bit modes
//
#ifndef __SURFACE__
#define __SURFACE__

#include      <malloc.h>
#include      "rect.h"
#include      "object.h"
#include      "font.h"

enum   RasterOperations
{
      RO_COPY,
      RO_OR,
      RO_AND,
      RO_XOR
};

class  RGB
{
```

```
public:
        char    red;
        char    green;
        char    blue;

        RGB () {}
        RGB ( int r, int g, int b )
        {
                red   = r;
                green = g;
                blue  = b;
        }
};

struct  PixelFormat
{
        unsigned redMask;       // bit mask for red color bits
        int     redShift;       // position of red bits in pixel
        int     redBits;        // # of bits for red field
        unsigned greenMask;     // bit mask for green color bits
        int     greenShift;     // position of green bits in pixel
        int     greenBits;      // # of bits for green field
        unsigned blueMask;      // bit mask for blue color bits
        int     blueShift;      // position of blue bits in pixel
        int     blueBits;
        int     bitsPerPixel;   // # of bits per pixel
        int     bytesPerPixel;  // # of bytes per single pixel
        void    completeFromMasks ();
};
inline int      operator == ( const PixelFormat& f1, const PixelFormat& f2 )
{
      return f1.redMask == f2.redMask &&
             f1.greenMask == f2.greenMask &&
             f1.blueMask == f2.blueMask;
}
inline  int     rgbTo24Bit ( int red, int green, int blue )
{
      return blue | (green << 8) | (red << 16);
}
inline  int     rgbTo16Bit ( int red, int green, int blue )
{
      return (blue>>3) | ((green>>2)<<5) | ((red>>3)<<10);
}
inline  int     rgbTo15Bit ( int red, int green, int blue )
{
      return (blue>>3) | ((green>>3)<<5) | ((red>>3)<<10);
}
inline  int     rgbToInt ( int red, int green, int blue, const PixelFormat& fmt )
{
```

```
        return ((blue>>(8-fmt.blueBits))<<fmt.blueShift) | ((green>>(8-
fmt.greenBits))<<fmt.greenShift) |
                ((red>>(8-fmt.redBits))<<fmt.redShift);
}
class   Surface   public Object
{
protected:                              // function pointers
        int   (Surface::*getPixelAddr)  (int x,  int y);
        void  (Surface::*drawPixelAddr) (int x,  int y, int color);
        void  (Surface::*drawLineAddr)  (int x1, int y1, int x2, int y2 );
        void  (Surface::*drawCharAddr)  (int x,  int y, int ch);
        void  (Surface::*drawStringAddr)(int x,  int y, const char * str);
        void  (Surface::*drawBarAddr)   (int x1, int y1, int x2, int y2 );
        void  (Surface::*copyAddr)      (Surface& dstSurface, const Rect& srcRect,
                                                        const Point& dst);
        void  (Surface::*copyTranspAddr)(Surface& dstSurface, const Rect& srcRect
                                                        const Point& dst, int transpColor);

public:
        int       bytesPerScanLine;   // # of bytes per scan line
        PixelFormat format;
        int       width;
        int       height;
        void      * data;               // image data
        RGB        * palette;           // currently used palette
        Rect       clipRect;           // area to clip
        Font      * curFont;
        int       color;                // current draw color
        int       backColor;           // current background color
        int       rasterOp;            // current raster operation
        Point      org;                 // drawing offset 8-bit (256 colors mode) functions
        int     getPixel8    ( int x,  int y );
        void    drawPixel8   ( int x,  int y, int color );
        void    drawLine8    ( int x1, int y1, int x2, int y2 );
        void    drawChar8    ( int x,  int y, int ch );
        void    drawString8  ( int x,  int y, const char * str );
        void    drawBar8     ( int x1, int y1, int x2, int y2 );
        void    copy8        ( Surface& dstSurface, const Rect& srcRect,
                               const Point& dst );
        void    copyTransp8  ( Surface& dstSurface, const Rect& srcRect,
                               const Point& dst, int transpColor );
                                                        // 15/16bit functions
        int     getPixel16   ( int x,  int y );
        void    drawPixel16  ( int x,  int y, int color );
        void    drawLine16   ( int x1, int y1, int x2, int y2 );
        void    drawChar16   ( int x,  int y, int ch );
        void    drawString16 ( int x,  int y, const char * str );
        void    drawBar16    ( int x1, int y1, int x2, int y2 );
        void    copy16       ( Surface& dstSurface, const Rect& srcRect,
                                 const Point& dst );
        void    copyTransp16 ( Surface& dstSurface, const Rect& srcRect,
                                 const Point& dst, int transpColor );
```

```
public:
        Surface ()   Object () {}
        Surface ( int w, int h, const PixelFormat& fmt );

        virtual ~Surface ()
        {
                if ( data != NULL )
                        free ( data );

                if ( palette != NULL )
                        delete palette;
        }

        virtual char  * getClassName () const
        {
                return "Surface";
        }

        virtual int     isOk () const
        {
                return data != NULL && width > 0 && height > 0;
        }

        virtual     int         put ( Store * ) const;
        virtual     int         get ( Store * );

        void        setFuncPointers ();

        int         closestColor ( const RGB& color ) const;

 virtual void    beginDraw () {}        // lock memory
 virtual void    endDraw () {}          // unlock memory

 void  * pixelAddr  ( const int x, const int y ) const
 {
            return x + y * bytesPerScanLine + (char *) data;
 }

 void    setClipRect ( const Rect& r )
 {
            clipRect = r;
 }

void    setOrg ( const Point& p )
{
     org = p;
}

void    setFont ( Font * fnt )
 {
     curFont = fnt;
}

 void    setColor ( int fg )
 {
            color = fg;
 }

 void        setBkColor ( int bkColor )
 {
```

```
                backColor = bkColor;
 }
 void   setRop ( int rop = RO_COPY )
{
      rasterOp = rop;
}
 int    getRop ()
{
      return rasterOp;
 }
Rect   getClipRect ()
{
      return clipRect;
}
Point  getOrg ()
{
      return org;
}
 void   drawFrame     (const Rect& r);
 void   scroll        (const Rect& area, const Point& dst);
int    getPixel ( int x, int y )
{
      return (this->*getPixelAddr)( x, y );
}
void   drawPixel ( int x,  int y, int color )
{
      (this->*drawPixelAddr)( x, y, color );
}
void   drawLine ( int x1, int y1, int x2, int y2 )
{
      (this->*drawLineAddr)( x1, y1, x2, y2 );
}
void   drawChar ( int x,  int y, int ch )
 {
      (this->*drawCharAddr)( x, y, ch );
}
void   drawString ( int x,  int y, const char * str )
{
      (this->*drawStringAddr)( x, y, str );
 }
void   drawBar ( int x1, int y1, int x2, int y2 )
{
      (this->*drawBarAddr)( x1, y1, x2, y2 );
}
void   copy ( Surface& dstSurface, const Rect& srcRect,
           const Point& dst )
{
```

```
            (this->*copyAddr)( dstSurface, srcRect, dst );
        }
        void    copyTransp ( Surface& dstSurface, const Rect& srcRect,
                                const Point& dst, int transpColor )
        {
                (this->*copyTranspAddr)( dstSurface, srcRect, dst, transpColor );
        }
        void    copy ( Surface& dstSurface, const Point& dstPoint )
        {
                (this->*copyAddr) ( dstSurface, Rect ( 0, 0, width-1, height-1 ),
                  dstPoint );
        }
        void    copyTransp  (Surface& dstSurface,
                  const Point& dstPoint, int transpColor)
        {
                (this->*copyTranspAddr) ( dstSurface, Rect(0,0,width-1,height-1),
                                dstPoint, transpColor );
        }
};
Object * loadSurface ();
#endif
```

Полностью реализацию классов Surface и VESASurface можно найти на ком-г-диске.

Упражнения

Реализуйте функции для работы со спрайтами для X-режима адаптера VGA.

Реализуйте функции для работы со спрайтами для SVGA-режимов (используя VBE 2.0).

Реализуйте работу со спрайтами для непалитровых режимов (используя VBE 2.0)

Обычно драйвер мыши не поддерживает вывод курсора мыши в SVGA-режимах. Перепишите файл mouse32.cpp для вывода курсора мыши (курсор удобнее всего сделать используя класс Sprite).

Реализуйте библиотеку основных графических функций для SVGA-адаптера в виде класса, используя стандарт VBE 2.0 в режимах 256 цветов и HiColor. Все функции должны поддерживать отсечение по заданному прямоугольнику и работу с видимой/невидимой страницами (если режим поддерживает).

Добавьте в класс Surface поддержку 24- и 32-битовых режимов.

Добавьте в класс Surface поддержку закраски прямоугольных областей по заданному шаблону, когда задается битовая маска, где пикселы, соответствующие бит 1, выводятся основным цветом, а пикселы, соответствующие биту 0, - цветом фона (или пропускаются, если этот цвет равен -1).

Реализуйте поддержку вывода спрайтов в объекты класса Surface.

Реализуйте простейшую real-time стратегическую игру (типа StarCraft).

Глава 5
Принципы построения пользовательского интерфейса

Интерфейс - некоторый способ (стандарт) взаимодействия (обмена информацией, данными) между программой и человеком, другой программой и т. п.

Под графическим пользовательским интерфейсом GUI (Graphical User Interface) понимается некоторая система (среда), служащая для организации взаимодействия прикладных программ с пользователем на основе графического многооконного представления данных.

Если посмотреть на любую хорошо сделанную прикладную программу, то придется признать, что не менее половины всего кода программы служит именно для организации интерфейса - ввод/вывод информации, работа с мышью, организация меню, реакция на ошибки и т. п.

В среде GUI организацию всего взаимодействия с пользователем берет на себя сама среда, оставляя прикладной программе делать только свою работу.

Считается, что основы GUI были заложены в 70-х гг. работами в исследовательском центре PARC фирмы Rank Xerox. В значительной степени под влиянием этих разработок в начале 80-х возник компьютер Lisa фирмы Apple (а вместе с ним и встроенный графический интерфейс), а в начале 1984 г. - компьютер Macintosh, который был фактически первым компьютером, специально спроектированным для работы с GUI и имевшим для этого все необходимое аппаратное и программное обеспечение (основная часть последнего находится в ПЗУ компьютера). Использование GUI в компьютере Macintosh сделало работу с ним чрезвычайно наглядной и понятной даже для начинающих пользователей.

Укажем некоторые преимущества этой среды. Все основные объекты (диски, каталоги, программы и пр.) представляются пиктограммами. Каждой программе отводится одно или несколько окон на экране, которые пользователь по своему усмотрению может передвигать, изменять размеры, уничтожать. Для манипулирования объектами активно используется мышь. Все программы имеют общие принципы построения, одинаковый дизайн, состоящий из одних и тех же элементов, причем все эти элементы просты и наглядны.

Вслед за этой средой появились другие графические среды - GEM фирмы Digital Research и Microsoft Windows (первая версия - ноябрь 1985 г.) - для персонального компьютера фирмы IBM.

К числу других графических сред можно также отнести:

- Presentation Manager (OS/2);
- OpenLook, Motif (Unix - станции);
- NextStep (Next).

Укажем несколько общих принципов, лежащих в основе перечисленных выше систем.

К числу таковых относятся:

- графический режим работы;
- представление ряда объектов пиктограммами;
- многооконность;
- использование указующего устройства - мыши;
- адекватность изображения на экране изображаемому объекту (принцип WYSIWYG - What You See Is What You Get);
- наглядность;
- стандартизация основных действий и элементов (все программы для данной графической среды выглядят и ведут себя совершенно одинаково, используют одинаковые принципы функционирования, так что если пользователь освоил работу с одной из программ, то он может легко освоить и остальные программы для данной среды);
- наличие большого числа стандартных элементов (кнопки, переключатели, поля редактирования), которые могут использоваться при конструировании прикладных программ, делая их похожими в обращении и облегчая процесс их написания;
- использование clipboard (pasteboard) - некоторого общего места (хранилища), с помощью которого программы могут обмениваться данными: в одной программе вы выделяете объект (фрагмент текста, изображение) и помещаете в clipboard, а в другой можете взять объект и вставить его в текущий документ (изображение);
- универсальность работы со всеми основными устройствами. Прикладная программа работает одинаково со всеми видеокартами, принтерами и т. п. через драйверы этих устройств. Таким образом, пользователь абстрагируется от специфики работы с конкретным устройством.

В основе любой системы GUI лежит достаточно мощный графический пакет: QuickDraw в Macintosh, GDI в Microsoft Windows, Display PostScript в NextStep. Этот пакет должен поддерживать работу с областями сложной формы и отсечения изображения по таким областям.

Рассмотрим теперь концепции, лежащие в основе многих GUI. Одной из основных, характерной для большинства существующих систем и программ для них, является понятие программы, управляемой данными. Как правило, эта концепция практически реализуется через механизм сообщений. Внешние устройства (клавиатура, мышь, таймер) посылают сообщения модулям программы о наступлении тех или иных событий (например, при нажатии клавиши или передвижении мыши). Поступающие сообщения попадают в очередь сообщений, откуда извлекаются прикладной программой.

Таким образом, программа не должна все время опрашивать мышь, клавиатуру и другие устройства в ожидании, не произошло ли чего-нибудь заслуживающего внимания. Когда событие произойдет, программа получит извещение об этом с тем, чтобы надлежащим образом его обработать. Поэтому программы для таких сред представляют собой цикл обработки сообщений: извлечь очередное сообщение, обработать его, если оно интересно, либо передать стандартному обработчику сообщений, обычно входящему в систему и представляющему собой стандартные действия системы в ответ на то или иное событие.

Приведем в качестве примера цикл обработки сообщений в программе для Microsoft Windows:

```
while ( GetMessage ( &msg, NULL, 0, 0 ) )
{
    TranslateMessage ( &msg );
    DispatchMessage  ( &msg );
}
```

В этом примере функция GetMessage извлекает очередное сообщение из системной очереди сообщений, а функция DispatchMessage отправляет его тому объекту, которому оно предназначено.

Сообщения могут посылаться не только устройствами, но и отдельными частями программы (в частности, возможна посылка сообщения самому себе). Причем один модуль может послать сообщение другому модулю, так, например, меню посылает сообщение о выборе определенного пункта. Существует и еще способ прямой посылки сообщения, минуя очередь, когда обработчик сообщений адресата вызывается непосредственно.

Второй основополагающей концепцией является понятие окна, окна как объекта. Окно - это не просто область на экране (как правило, прямоугольная), это еще и программа (функция), способная выполнять различные действия, присущие окну. В качестве таких действий выступают реагирование на поступающие сообщения и посылка сообщений другим объектам.

Все окна образуют иерархическую (древовидную) структуру - каждое окно может иметь подокна, непосредственно принадлежащие этому окну и содержащиеся в нем. Любое окно, кроме корневого, также имеет родителя - окно, которому оно само принадлежит. Родительское окно и его подокна могут обмениваться сообщениями.

Корневое окно обычно называется десктопом (desktop) и занимает собой весь экран.

Замечание. Оконная система компьютеров Apple Macintosh поддерживает всего три уровня иерархии - desktop, обычное окно и управляющий элемент (control) - особый тип подокна.

Пример иерархии окон представлен на рис. 5.1.

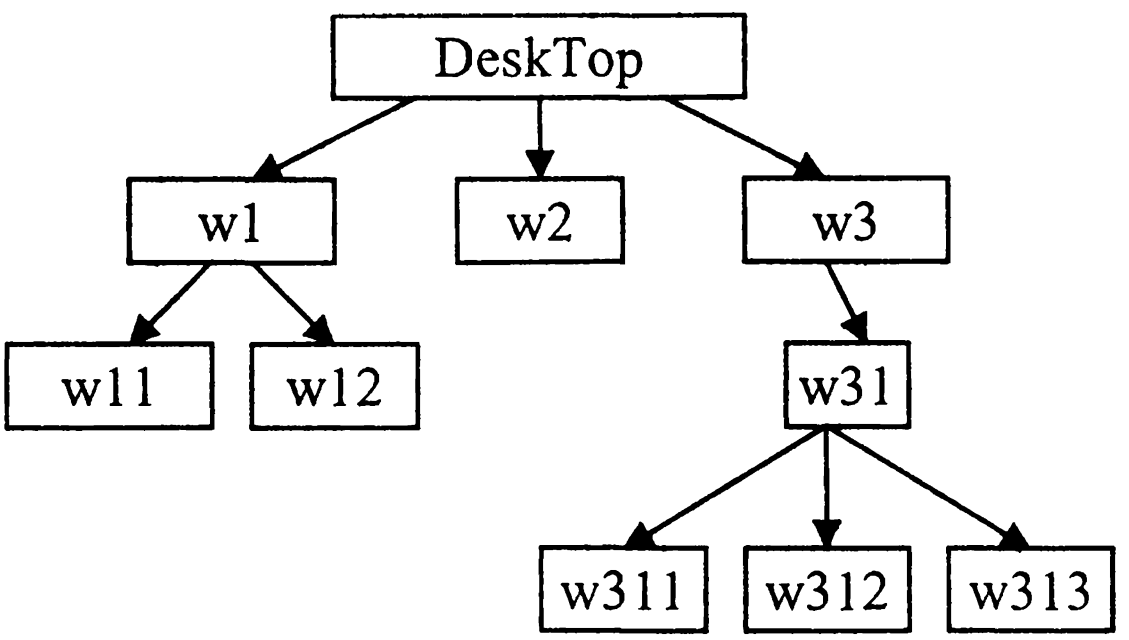


Рис. 5.1

С учетом этого простейшее окно может быть реализовано следующим классом:

```
class   Window
{
public:
    Rect        area;       // area of the window
    char    *   text;       // text or caption of the window
    Window  *   parent;     // owner of this window
    Window  *   child;      // topmost child
    Window  *   next;       // link all windows of the
    Window  *   prev;       // same parent
    long        style;      // window style
    long        status;     // current window status

    Window (const Rect& frameRect, char * text, Window * owner );
    virtual  ~Window ();

    virtual  int      handle ( const Message& );
    virtual  void     showWindow ();
    virtual  void     hideWindow ();
    virtual  void     moveWindow ( const Rect& );
};
```

Приведенный пример напоминает реализацию окна в среде Microsoft Windows, где окно фактически представлено как структура, содержащая указатель на функцию обработки сообщений; виртуальная функция handle реализует обработку всех поступающих сообщений.

Все сообщения обычно разделяются на информирующие сообщения - произошло определенное событие (была нажата клавиша, изменилось положение мыши, был изменен размер окна и т. д.) - и сообщения-запросы, требующие от окна выполнения определенных действий.

Одним из основных сообщений, на которое должно реагировать любое окно, является сообщение нарисовать себя (или свою часть) с учетом установленной области отсечения.

Рассмотрим теперь, каким образом реализуются основные операции над окнами: показать/убрать, изменить размер, передвинуть.

1. Показать окно (showWindow).

В переменной status устанавливается бит (WS_VISIBLE), показывающий, что окно отображено. Область отсечения устанавливается равной области окна минус области его непосредственных видимых подокон (у которых в status установлен бит WS_VISIBLE), и окну посылается сообщение нарисовать себя. Затем подобная же операция выполняется для всех непосредственных видимых подокон этого окна. При создании окна можно указать, следует ли его показывать сразу же, или это необходимо сделать явным вызовом showWindow

2. Убрать окно (hideWindow).

В переменной status сбрасывается бит WS_VISIBLE, соответствующий видимости, и определяется область экрана, которая будет открыта при убирании данного окна с экрана. Затем для каждого окна, которое полностью или частично откроется при убирании данного, определяется открываемая область. Когда эта область откроется, она становится новой областью отсечения и для окна выполняется запрос на

перерисовку себя. Таким образом, изображение, лежащее под убираемым окном, восстанавливается полностью. Рассмотрим ситуацию, изображенную на рис. 5.2. Пусть убирается окно w2. Тогда откроются следующие области (рис. 5.3). Таким образом, окно deskTop должно заполнить область d, а окно w1 - область w.

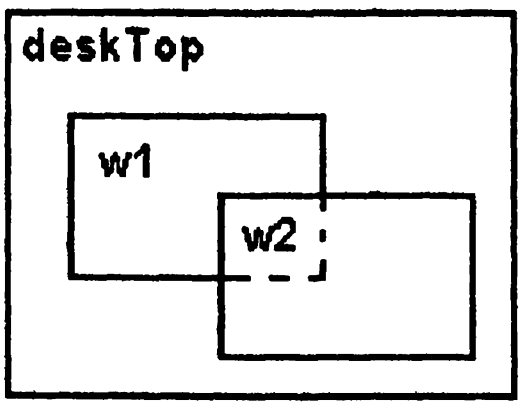


Рис. 5.2

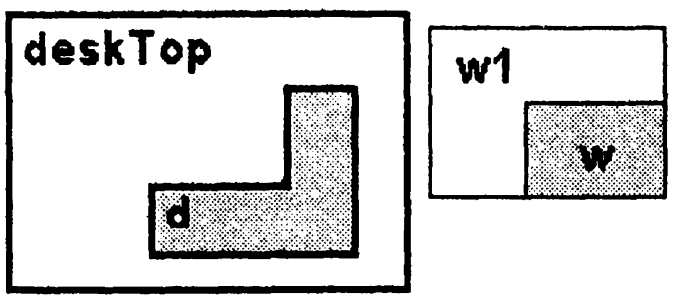


Рис. 5.3

3. Изменение размеров(resize).

Окно перерисовывается, как при его показе, а открывшиеся области заполняются аналогично тому, как это делается при убирании окна.

4. Передвижение окна (moveWindow).

Содержимое окна копируется на новое место (возможна и полная перерисовка всего окна), отрисовываются вновь открывшиеся части данного окна и других окон.

В ряде систем вместо немедленной перерисовки содержимого у окна устанавливается указатель на область, содержимое которой должно быть перерисовано. В подходящий момент для каждого окна, имеющего непустую область, требующую перерисовки, генерируется сообщение на перерисовку содержимого соответствующей области.

Рассмотрим механизм передачи сообщений в системе. Каждое сообщение обычно поступает сначала в системную очередь, откуда извлекается программой. Для некоторых сообщений при их создании явно указывается, какому окну они адресованы. Другие же сообщения, например сообщения от мыши и клавиатуры, изначально явных адресатов не имеют и потому распределяются специальным образом.

Обычно все сообщения от мыши посылаются тому окну, над которым находится курсор мыши (произошло событие). Однако существует путь обхода этого. Окно может "поймать" мышь, после чего все сообщения от мыши, откуда бы они ни приходили, будут поступать только этому окну до тех пор, пока окно, "поймавшее" мышь, не "отпустит" ее.

Рассмотрим следующую ситуацию: пользователь нажал кнопку мыши в тот момент, когда курсор мыши находился над нажимаемой кнопкой в окне. В этом случае кнопку нужно "нажать" (перерисовать ее изображение) и удерживать нажатой, пока нажата кнопка мыши. Однако если пользователь резко сдвинет мышь, удерживая кнопку мыши нажатой, то нажимаемая кнопка может не получить сообщения о том, что мышь покинула пределы нажимаемой кнопки (вследствие того, что, как только мышь покинет эти пределы, сообщения от нее будут поступать уже другому окну). При этом кнопка все время будет оставаться нажатой. Поэтому нажимаемая кнопка должна "захватить" мышь и удерживать ее, пока кнопка мыши нажата. Когда пользователь отпустит кнопку мыши, кнопка на экране "отжимается" и мышь "освобождается"

При работе с клавиатурой важную роль играет понятие фокуса ввода.

Фокус ввода - это то окно, которому поступают все сообщения от клавиатуры. Существует несколько способов перемещения фокуса ввода:

- при нажатии кнопки мыши фокус передается тому окну, над которым это произошло;
- окна диалога обычно переключают фокус между управляющими элементами диалога при нажатии определенных клавиш (стандартно это Tab и Shift-Tab);
- посредством явного вызова функции установки фокуса ввода.

Окну, теряющему фокус ввода, обычно посылается уведомление об этом, и оно может предотвратить переход фокуса от себя. Окну, получающему фокус, передается сообщение о том, что оно получило фокус ввода.

В некоторых системах (X Window) понятия фокуса вообще нет - сообщения от клавиатуры всегда получает то окно, над которым находится курсор мыши.

5.1. Основные типы окон

Любая система GUI имеет в своем составе достаточно большое количество стандартных типов окон, которые пользователь может применять непосредственно и на основе которых он может создавать свои собственные типы окон.

Нормальное ("классическое") окно состоит из заголовка (Caption), кнопки уничтожения (или системного меню) и рабочей области. Кроме этого могут присутствовать меню, кнопки минимизации (кнопка минимизации превращает окно в пиктограмму), максимизации (кнопка максимизации делает окно наибольшего возможного размера) и полосы прокрутки (Scroll Bar), служащие для управления отображением в окне объекта, слишком большого, чтобы целиком уместиться в нем (рис. 5.4).

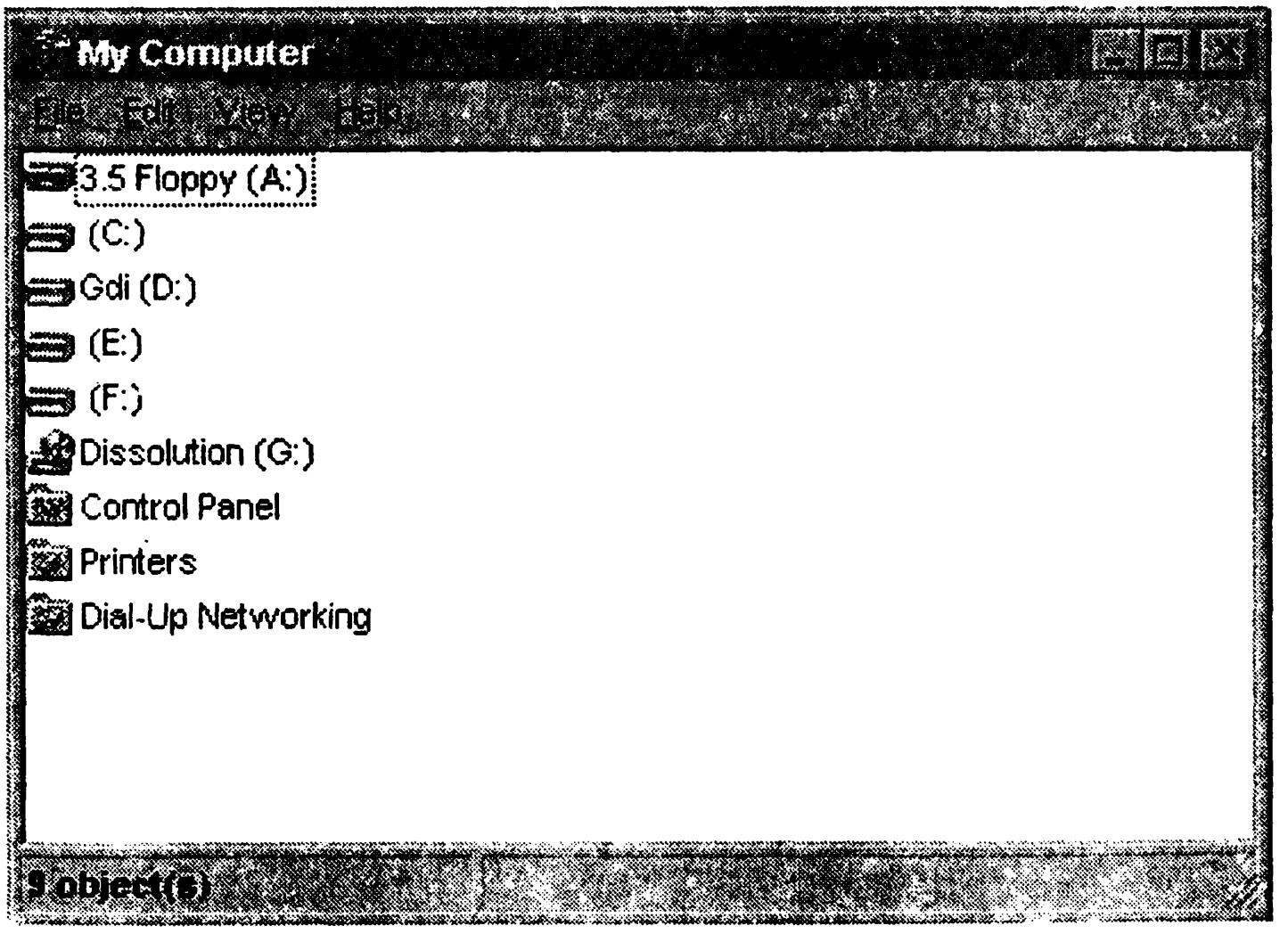


Рис. 5.4

Диалоговое окно представляет собой специальный тип окна, употребляемый для в дения диалога с пользователем. В качестве управляющих элементов применяется р специальных подокон - кнопки, переключатели, списки, поля редактирования и т. д.

Основной функцией диалоговых окон является организация взаимодействия его подокнами. Любой диалог, как правило, включает в себя несколько кнопок, од из которой является определенной по умолчанию; нажатие клавиши Enter эквив лентно нажатию этой кнопки. Обычно присутствует и кнопка отмены; нажатие кл виши Esc эквивалентно нажатию этой кнопки.

Основной функцией обработки сообщений диалогового окна является координ ция всех его управляющих элементов (подокон).

На рис. 5.5 приведен диалог открытия файла в Windows 95.

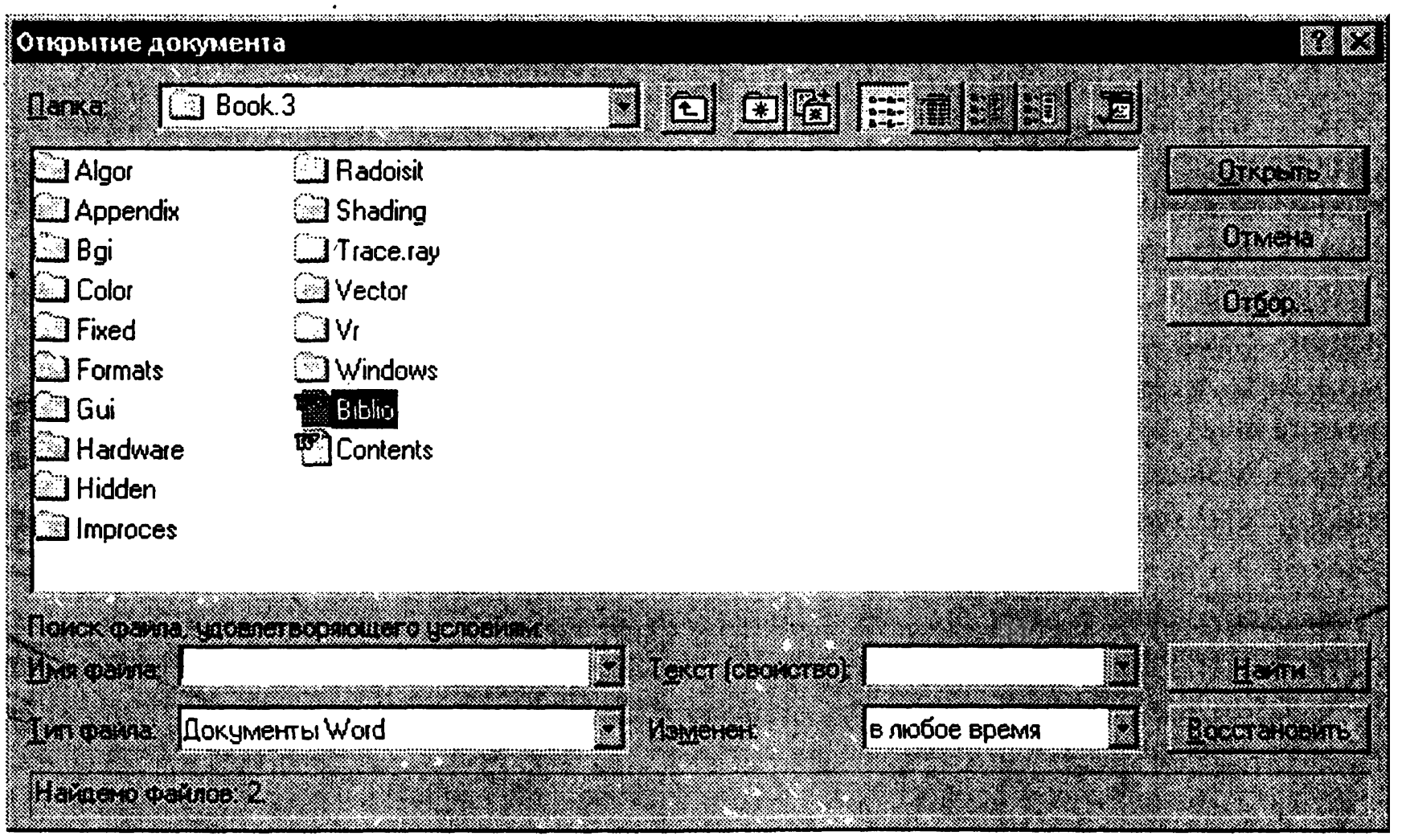


Рис. 5.5

Существует также набор специальных окон, предназначенных исключительн для использования в качестве дочерних окон. Это нажимаемые кнопки (рис. 5.6 различные виды переключателей (рис. 5.7 и 5.8), окна, служащие для ввода и реда тирования текста, окна, для отображения текста или изображения (рис. 5.9), полос прокрутки, списки (рис. 5.10), деревья и т. п.

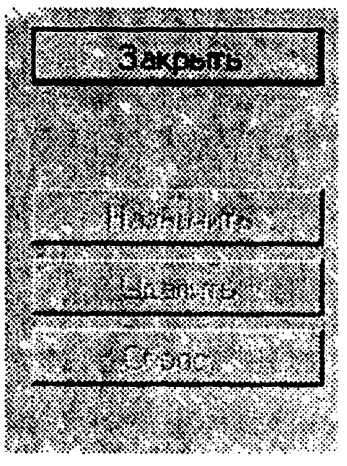


Рис. 5.6

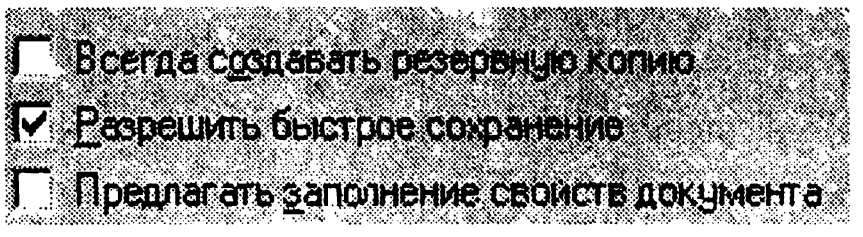


Рис. 5.7

Как видно из последних примеров, в состав окна могут входить другие окна и действовать при этом как единое целое. Например, в состав окна-списка входит полоса прокрутки.

Отличительной особенностью этих окон является то, что они предназначены для вставки в качестве дочерних в другие окна, т. е. играют роль управляющих элементов. При каком-либо воздействии на них или изменении своего состояния они посылают уведомляющее сообщение родительскому окну.



Рис. 5.8

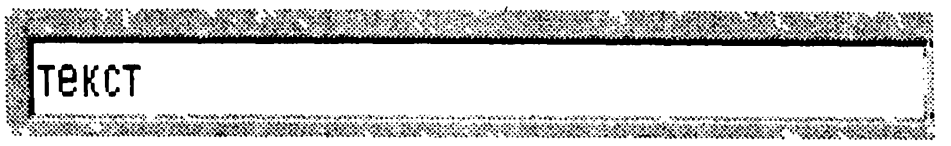


Рис. 5.9

Поскольку каждое окно является объектом, то естественной является операция наследования - создания новых классов окон на базе уже существующих путем добавления каких-то новых свойств либо переопределения части старых и наследования всех остальных.

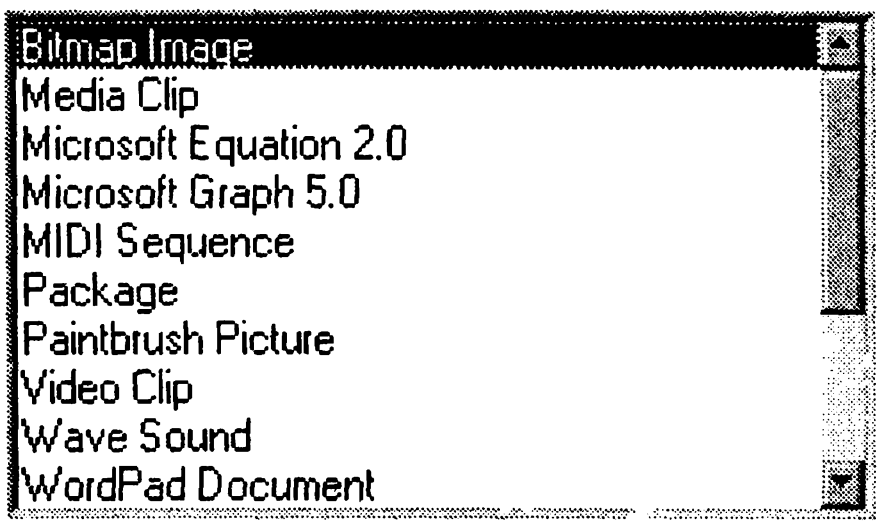


Рис. 5.10

В приводимом ниже примере создается новый тип объекта - MyWindow, происходящий от базового типа Window, но отличающийся от него иной обработкой некоторых сообщений.

```
class    MyWindow : public Window
{
    public:
        .  . . . .    .
    virtual   int  handle ( const Message& );
};
```

В приведенных примерах, как и в среде Microsoft Windows, для обработки всех поступающих окну сообщений используется всего одна функция. Фактически это приводит к появлению огромной (на несколько тысяч строк) функции, в которой обрабатываются все возможные сообщения (а их общее количество может составлять несколько сотен). Было бы намного удобнее, если бы каждому сообщению можно было поставить в соответствие свою функцию для его обработки. Многие системы (BeOS, NextStep) так и делают. Наиболее удачной является реализация такого подхода в операционной системе (ОС) NextStep (OpenStep, Rhapsody, MacOSX Server), когда вся ОС (за исключением микроядра Mach) написана на крайне гибком объектно-ориентированном языке Objective-C, представляющем удачное соединение возможностей таких языков, как C и SmallTalk. Работа с сообщениями фактически встроена в сам язык, т. е. любой вызов метода объекта заключается в посылке

ему сообщения; например, следующий фрагмент кода вызывает метод mouseMoved с аргументом theEvent у объекта obj:

```
[obj mouseMoved:theEvent];
```

Привязка посылаемого сообщения к конкретному методу осуществляется на этапе выполнения программы путем поиска соответствующего метода в таблице методов объекта (а не простой индексации, как для языка C++). За счет этого можно послать объекту фактически любое сообщение, не заботясь о том, реализован ли в этом объекте обработчик соответствующего сообщения; в случае отсутствия соответствующего метода в качестве результата будет просто возвращен NULL. Язык предоставляет также возможность спросить объект, поддерживает ли он данный метод или протокол (совокупность методов). При этом сама ОС содержит огромное количество уже готовых классов, на которых, собственно, она сама и написана, и программист может все их использовать или создавать на их основе новые. Не случайно, что именно NextStep признана самой удобной средой для разработки.

Еще одним примером удачной объектно-ориентированной системы является BeOS, целиком написанная на языке C++.

В отличие от приведенных выше систем Microsoft Windows написана фактически на языке C и объектно-ориентированной может быть названа с очень большой натяжкой.

Для облегчения программирования в Microsoft Windows существуют специальные библиотеки классов C++, облегчающие программирование в этой среде. Для обеспечения связи между номером сообщения и функцией обработки сообщения вводятся специальные макрокоманды, очень загромождающие программу и заметно понижающие ее читаемость. Пример этого приводится ниже.

```
class    CMFMenuWindow : public CFrameWnd
{
public:
    CMFMenuWindow ();

    afx_msg    void     MenuCommand ();
    afx_msg    void     ExitApp ();

    DECLARE_MESSAGE_MAP ()
};

BEGIN_MESSAGE_MAP(CMFMenuWindow, CFrameWnd)
    ON_COMMAND(ID_TEST_BEEP, MenuCommand);
    ON_COMMAND(ID_TEST_EXIT, ExitApp);
END_MESSAGE_MAP()
```

Выглядят подобные конструкции нелепо, а их появление свидетельствует о двух вещах: во-первых, язык C++ плохо подходит для написания действительно объектно-ориентированных распределенных приложений (вся подобная работа с таблицами должна неявно делаться средствами самого языка, а не посредством искусственных макросов) и, во-вторых, среда Microsoft Windows с большим трудом может быть названа действительно объектно-ориентированной, поэтому и написание объектно-ориентированных приложений под нее является таким неудобным.

5.1.1. Пример реализации основных оконных функций

Обычно для реализации основных функций для работы с окнами требуется графический пакет, поддерживающий работу с областями сложной формы. С каждым окном связываются две такие области - область отсечения, представляющая собой видимую часть окна, и область, требующая перерисовки. Менеджер окон сам определяет окна, у которых область, требующая перерисовки, не пуста, и автоматически генерирует для таких окон запрос на перерисовку соответствующей области.

В случае, когда мы работаем только прямоугольными окнами, все области, возникающие при выполнении над окнами основных операций, являются объединением нескольких прямоугольников, так что для простейшей реализации оконного интерфейса достаточно иметь графическую библиотеку с возможностью отсечения только по прямоугольным областям. В случае, когда область состоит из нескольких прямоугольников, каждый из них по очереди становится областью отсечения и для него выполняется функция перерисовки соответствующего окна.

Ниже приводится пример подобной системы. В ней весь экран разбивается на прямоугольники, являющиеся видимыми частями окон, и все рисование ведется на основе данного разбиения. Если нужно нарисовать содержимое области, то определяются все прямоугольники, имеющие с ней непустое пересечение, для каждого из них устанавливается соответствующая область отсечения (равная пересечению области и прямоугольника) и для соответствующего окна вызывается функция перерисовки.

Использование отсечения только по прямоугольным областям заметно ускоряет и упрощает процесс отсечения, но за это приходится расплачиваться несколькими вызовами функции рисования для областей, являющихся объединением нескольких прямоугольников.

```
// File view.h
//
// Simple windowing system, basic class for all window objects
//
#ifndef __VIEW__
#define __VIEW__

#include      <string.h>
#include      "point.h"
#include      "rect.h"
#include      "mouse.h"
#include      "surface.h"
#include      "object.h"
#include      "message.h"

#define IDOK    1
#define IDCANCEL 2
#define IDHELP     3
#define IDOPEN     4
#define IDCLOSE   5
#define IDSAVE     6
#define IDQUIT 7
                    // generic notifications
```

```
// (send via WM_COMMAND)
#define VN_TEXTCHANGED 0                    // view text has changed
#define VN_FOCUS       1                    // view received/lost focus style bits

#define WS_ACTIVATEABLE      0x0001   // can be activated
#define WS_REPAINTONFOCUS    0x0002   // should be repainted on focus change
#define WS_FLOATING      0x0004   // window floats above normal ones
                                      // status bits
#define WS_ENABLED       0x0001       // can receive focus, mouse &
                                      // keyboard messages
#define WS_VISIBLE       0x0002       // is visible (shown)
#define WS_COMPLETELYVISIBLE   0x0004 // not overlaid by someone hit codes

#define HT_CLIENT        0

class  View;
class  Menu;
class  Map;

extern  Surface *  screenSurface;   // surface we draw on
extern  Rect       screenRect;      // current screen rect
extern  View   *   deskTop;         // desktop
extern  View   *   focusedView;     // focused view

//////////////////////////////////////////////////////////////////
int         setFocus     ( View * );
void        redrawRect   ( Rect& );
View  *     findView     ( const Point& );
//////////////////////////////////////////////////////////////////

class  View : public Object    // base class of all windowing system
{
protected:
      char  *     text;          // text or caption
      View  *     parent;        // owner of this view
      View  *     next;          // next child of parent
      View  *     prev;          // previous chlid of this parent
      View  *     child;         // topmost child view of this
      int         style;         // view style
      int         status;        // view status
      Rect        area;          // frameRect of this view
      int         tag;           // tag of view (-1 by default)
      int         lockCount;
      View  *     delegate;      // to whom send notifications
      View  *     hook;          // view that hooked this one
public:
     View ( int x, int y, int w, int h, View * owner = NULL );
     View ();
      virtual ~View ();

      virtual char  * getClassName () const
      {
            return "View";
      }

      virtual int     put ( Store * ) const;
```

```
 virtual int    get ( Store * );
virtual void    init () {}                  // post-constructor init
 virtual void    show ();                    // show this window
 virtual void    hide ();                    // hide this window
 virtual int    handle ( const Message& );
                                            // keyboard messages
 virtual int    keyDown ( const Message& );
 virtual int    keyUp ( const Message& );
                                            // mouse events
 virtual int    mouseDown ( const Message& );
 virtual int    mouseUp ( const Message& );
 virtual int    mouseMove ( const Message& );
 virtual int    rightMouseDown ( const Message& );
 virtual int    rightMouseUp ( const Message& );
 virtual int    mouseDoubleClick ( const Message& );
 virtual int    mouseTripleClick ( const Message& );
 virtual int    receiveFocus ( const Message& );
 virtual int    looseFocus ( const Message& );
 virtual int    command ( const Message& );
 virtual int    timer ( const Message& );
 virtual int    close ( const Message& );
 virtual void    helpRequested ( const Message& ) {}
 virtual void    draw ( const Rect& ) const {}
 virtual void    getMinMaxSize ( Point& minSize,
                                 Point& maxSize ) const;
 virtual Rect    getClientRect ( Rect& ) const;
 virtual int    hitTest ( const Point& ) const
 {
       return HT_CLIENT;
 }
 virtual Menu  * getMenu ( const Message& ) const
 {
       return NULL;
 }
 virtual int    handleHooked ( const Message& )
 {
       return FALSE;       // message not processed by hook, so own message
                           // handler should be called
 }
                           // whether the view can receive input focus
virtual int    canReceiveFocus () const
{
    return TRUE;
}
void    setTag ( int newTag )
{
```

```
            tag = newTag;
        }
    View  * viewWithTag ( int theTag );
    void    addSubView ( View * subView );
    void    setFrame ( int x, int y, int width, int height );
    void    setFrame ( const Rect& r )
    {
            setFrame ( r.x1, r.y1, r.width (), r.height () );
    }

        Rect   getFrame () const
        {
              return area;
        }

        Rect       getScreenRect () const;
        Point&     local2Screen ( Point& ) const;
        Rect&      local2Screen ( Rect& ) const;
        Point&     screen2Local ( Point& ) const;
        Rect&      screen2Local ( Rect& ) const;

        void       enableView ( int = TRUE );

        void       beginDraw () const;
        void       endDraw    () const;
        void       repaint    () const;
        void       lock     ( int flag = TRUE );

        View  *    getFirstTab () const;

        void       setZOrder ( View * behind = NULL );
        View  *    hookWindow ( View * whome );

        int        containsFocus () const;

        char  *    getText () const
        {
            return text;
        }

        char  * setText ( const char * newText );

        View  * getParent () const
        {
            return parent;
        }

        int    getStyle () const
        {
            return style;
        }

        void   setStyle ( int newStyle )
        {
            style = newStyle;
        }

        int  getStatus () const
        {
```

```
    return status;
}
int    contains ( const Point& p )
{
    return p.x >= 0 && p.x < width () && p.y >= 0 &&
           p.y < height ();
}
int    width () const
{
    return area.width ();
}
int    height () const
{
    return area.height ();
}
int    isVisible () const
{
    return status & WS_VISIBLE;
}
int    isActivateable () const
{
    return style & WS_ACTIVATEABLE;
}
    int    isFloating () const
    {
        return style & WS_FLOATING;
    }
    int    isEnabled () const
    {
        return status & WS_ENABLED;
    }
    int    isLocked () const
    {
        return lockCount > 0;
    }
    int    isFocused () const
    {
        return focusedView == this;
    }
    friend class Map;
    friend class DialogWindow;
    friend int   setFocus ( View * );
};
Object * loadView ();
//////////////////////////////////////////////////////////////////
#endif
```

```
// File view.cpp
//
// Simple windowing system, basic class for all window objects
//
#include      <assert.h>
#include      <dos.h>
#include      <stdlib.h>

#include      "array.h"
#include      "store.h"
#include      "view.h"

/////////////// Global variables  ///////////////

View * deskTop         = NULL;
View * focusedView     = NULL;

Array  floaters ( 10, 10 );

/////////////// Objects to manage screen layout  ///////////////

class   Region   public Rect
{
public:
    View    * owner;
    Region * next;
};

class   Map
{
    Region  *   pool;           // pool of regions
    int         poolSize;       // # of structures allocated
    Region  *   firstAvailable; // pointer to 1st struct after all allocate
    Region  *   lastAvailable;  // last item in the pool
    Region  *   free;           // pointer to free structs list (below firs
    Region  *   start;          // first region in the list

public:
    Map ( int mapSize )
    {
        pool            = new Region [poolSize = mapSize];
        firstAvailable = pool;
        lastAvailable  = pool + poolSize - 1;
        free            = NULL;
        start           = NULL;
    }

    ~Map ()
    {
        delete [] pool;
    }

    void    freeAll ()          // free all used regions
    {
        firstAvailable = pool;
        free            = NULL;
        start           = NULL,
```

```
    }
    Region  * allocRegion ();       // allocate unused region
                                     // deallocate region (return to free list)
    void    freeRegion ( Region * reg )
    {
          reg -> next = free;
          free        = reg;
     }
     void    rebuildMap   ();    // rebuild map from beginning
                                 // redraw given screen rectangle
     void    redrawRect          ( const Rect& ) const;
                                 // find window, containg point
     View  *                     findView   ( const Point& ) const;
                                 // find 1st region of view list
Region * findViewArea            ( const View * view ) const;
                                 // add view (with all subviews) to screen map
void          addView ( View *, Rect&, int );
     friend  class View;
};
////////////////////// Map methods ////////////////////
Region * Map :: allocRegion ()
{                                // when no free spans
     if ( free == NULL )
          if ( firstAvailable <= lastAvailable )
               return firstAvailable++;
          else
               return NULL;
     Region * res = free;
     free = free -> next;
     return res;
}
void   Map   rebuildMap ()
{
     freeAll ();
    addView ( deskTop, screenRect, 0 );
}
void   Map   addView ( View * view, Rect& viewRect, int recLevel )
{
    int   updatePrev;    // whether we should update prev
    Rect    r;
    Rect    splittingRect;
    View   * owner;
    Region * reg, * next;
    Region * prev = NULL,
    if ( !view -> isVisible () )      // if not vivible
        return;                       // nothing to add
    viewRect   &   = screenRect;                    // clip area to screen
```

```
view -> status |= WS_COMPLETELYVISIBLE;  // initially not overla
                                          // by anyone

if ( viewRect.isEmpty () )
     return;
for ( reg = start; reg != NULL; reg = next )
{
     next       = reg -> next;
     updatePrev = TRUE;
                         floated views
                              // should be added
                              // after all subviews
     if ( reg -> owner -> isFloating () )
     {
          if ( recLevel >= 0 )  // if adding view then store floater
          {
               floaters.insert ( reg -> owner );
               prev = reg;        // update prev
               continue;          // skip floater
          }
     }

     splittingRect  = * reg;             // get current rect
     splittingRect &= viewRect;          // clip it to viewRect

     if ( splittingRect.isEmpty () )  // if not
     {                        // intersection
          prev = reg;        // update prev
          continue;          // then continue
}
if ( splittingRect == * reg )   // current is
{                        // overlapped
     if ( prev != NULL )     // remove region from region list
          prev -> next = next;
     freeRegion ( reg );     // free it
     updatePrev = FALSE;
}
else
{
     r        = * reg;              // save current rect
     owner = reg -> owner;   // and it's owner
     owner -> status &= ~WS_COMPLETELYVISIBLE;
                         // 1st block
     reg -> y2 = splittingRect.y1 - 1;
     if ( !reg -> isEmpty () )
     {
          prev           = reg;
          reg            = allocRegion ();
          prev -> next = reg;
     }
```

```
                    // 2nd block
reg -> x1       = r.x1;
reg -> y1       = splittingRect.y1;
reg -> x2       = splittingRect.x1 - 1;
reg -> y2       = splittingRect.y2;
reg -> owner = owner;

if ( !reg -> isEmpty () )
{
     prev        = reg;
     reg         = allocRegion ();
     prev -> next = reg;
}
                    // 3rd block
reg -> x1       = splittingRect.x2 + 1;
reg -> y1       = splittingRect.y1;
reg -> x2       = r.x2;
reg -> y2       = splittingRect.y2;
reg -> owner = owner;

if ( !reg -> isEmpty () )
{
     prev           = reg;
     reg            = allocRegion ();
     prev -> next = reg;
}
                    // 4th block
          reg -> x1       = r.x1;
          reg -> y1       = splittingRect.y2 + 1;
          reg -> x2       = r.x2;
          reg -> y2       = r.y2;
          reg -> owner = owner;

          if ( reg -> isEmpty () )// remove from chain
          {
               if ( prev != NULL )
                    prev -> next = next;

               freeRegion ( reg );

               updatePrev = FALSE;
          }
          else
               reg -> next  = next;
     }

     if ( updatePrev )
          prev = reg;
}
                    // now append viewRect to the end of list
reg = allocRegion ();          // allocate new block & add it to the list

* (Rect *) reg = viewRect;
reg -> owner  = view;
reg -> next     = NULL;
```

```
if ( prev != NULL )          // append it to the end of the list
     prev -> next = reg;
else
     start = reg;

if ( view -> isFloating () && recLevel == 0 )
     floaters.insert ( view );

if ( view -> child != NULL )    // now add it's subviews
{                               // find downmost view
     for ( View * c = view -> child; c -> prev != NULL; )
          c =.c -> prev;
                                // start adding from downmost subview
     for ( ; c != NULL; c = c -> next )
     {                          // get subview's rect
          r  = c -> getScreenRect ();
          r &= viewRect;  // clip to parent's rect
                             // add to map
          addView ( c, r, recLevel + 1 );
     }
}

if ( recLevel == 0 )
{
     for ( int i = 0; i < floaters.getCount (); i++ )
     {
          View * v = (View *) floaters [i];
          addView ( v, v -> getScreenRect (), -10000 );
     }
     floaters.deleteAll ();
  }
}

void   Map :: redrawRect ( const Rect& r ) const
{
  if ( r.isEmpty () )
       return;

  screenSurface -> beginDraw ();

  for ( Region * reg = start; reg != NULL; reg = reg -> next )
  {
       Rect   clipRect ( * reg );
       Point  org ( 0, 0 );

       clipRect &= r;

       if ( clipRect.isEmpty () )
            continue;

       screenSurface -> setClipRect ( clipRect );

       Rect   drawRect ( clipRect );

       reg -> owner -> screen2Local ( drawRect );
       reg -> owner -> beginDraw ();
       reg -> owner -> draw ( drawRect );
```

```
            reg -> owner -> endDraw ();
        }
        screenSurface -> endDraw ();
}
View *  Map :: findView ( const Point& p ) const
{
        for ( Region * reg = start; reg != NULL; reg = reg -> next )
                if ( reg -> contains ( p ) )
                    return reg -> owner;
        return NULL;
}
Region * Map :: findViewArea ( const View * view ) const
{
        for ( Region * reg = start; reg != NULL; reg = reg -> next )
                if ( reg -> owner == view )
                        return reg;
        return NULL;
}
////////////////////////////////////////////////////////
static  Map     screen ( 2000 );
////////////////////  View methods  ////////////////////
View :: View ( int x, int y, int w, int h, View * p ) :
area ( x, y, x + w - 1, y + h - 1 )
{
        style           = 0;
        status          = 0;
        lockCount       = 0;        // initially not locked
        tag             = -1;       // default tag
        child           = NULL;
        prev            = NULL;
        next            = NULL;
        delegate        = p;        // by default all notifications are sent to parent
        hook            = NULL;
        text            = (char *) malloc ( 80 );
        strcpy ( text, "" );
        if ( ( parent = p ) != NULL )
                parent -> addSubView ( this );
}
View :: View ()
{
        parent = deskTop;
        text    = (char *) malloc ( 80 );
}
View :: ~View ( )
{
        hide ();
```

```
        while ( child != NULL )        // remove all subwindows
            delete child;

        if ( parent != NULL )
            if ( parent -> child == this )
                parent -> child = prev;

        if ( prev != NULL )            // remove from chain
            prev -> next = next;

        if ( next != NULL )
            next -> prev = prev;

        free ( text );
}
int    View    put ( Store * s ) const
{
    s -> putString ( text );
    s -> putInt    ( style );
    s -> putInt    ( status );
    s -> putInt    ( area.x1 );
    s -> putInt    ( area.y1 );
    s -> putInt    ( area.x2 );
    s -> putInt    ( area.y2 );
    s -> putInt    ( tag );

    int    subViewCount = 0;

    for ( View * v = child; v -> prev != NULL;
                v = v -> prev, subViewCount++ )
        ;

    s -> putInt ( subViewCount + 1 );

    for ( ; v != NULL; v = v -> next )
        s -> putObject ( v );

    return TRUE;
}
int    View :: get ( Store * s )
{
    text        = s -> getString    ();
    style       = s -> getInt       ();
    status      = s -> getInt       () & ~WS_VISIBLE;
    area.x1     = s -> getInt       ();
    area.y1     = s -> getInt       ();
    area.x2     = s -> getInt       ();
    area.y2     = s -> getInt       ();
    tag         = s -> getInt       ();

    int    subViewCount = s -> getInt ();

    for ( int i = 0; i < subViewCount; i++ )
        addSubView ( (View *) s -> getObject () );

    return TRUE;
}
```

```
void  View  show ()
{
    if ( !isVisible () )
    {
        status |= WS_VISIBLE;
        screen.addView ( this, getScreenRect (), 0 );
        repaint ();
        if ( deskTop == NULL )
            deskTop = this;
    }
    if ( isEnabled () )
        setFocus ( this );
}
void  View :: hide ()
{
     if ( isVisible () )
     {
          if ( containsFocus () ) // move focus from this view
        {               // or it's children
            for (View * v=prev; v!=NULL; v=v -> prev)
                if (v->isVisible () && v->isEnabled())
                    break;
            setFocus ( v != NULL ? v : parent );
        }
        status &= ~WS_VISIBLE;
                    // hide all children
        for ( View * v = child; v != NULL; v = v -> prev )
        {
            v -> lockCount ++;
            v -> hide ();
            v -> lockCount --;
        }
        if ( !isLocked () )
        {
            screen.rebuildMap ();
            screen.redrawRect ( getScreenRect () );
        }
        if ( deskTop == this )
            deskTop = NULL;
    }
}
char  * View  setText ( const char * newText )
{
    free ( text );
    text = strdup ( newText );
     repaint ();
     sendMessage (delegate, WM_COMMAND, VN_TEXTCHANGED, 0, this);
```

```
        return text;
}
int    View    handle ( const Message& m )
{                           // check whether hook
                            // intercepts message
      if ( hook != NULL && hook -> handleHooked ( m ) )
          return TRUE;

       switch ( m.code )
       {
            case WM_KEYDOWN:
                 return keyDown ( m );

            case WM_KEYUP:
                 return keyUp ( m );

           case WM_MOUSEMOVE:
                 return mouseMove ( m );

            case WM_LBUTTONDOWN:
                 return mouseDown ( m );

            case WM_LBUTTONUP:
                 return mouseUp ( m );

            case WM_RBUTTONDOWN:
                 return rightMouseDown ( m );

            case WM_RBUTTONUP:
                 return rightMouseUp ( m );

            case WM_DBLCLICK:
                 return mouseDoubleClick ( m );

            case WM_TRIPLECLICK:
                 return mouseTripleClick ( m );

            case WM_RECEIVEFOCUS:
                 return receiveFocus ( m );

            case WM_LOOSEFOCUS:
                 return looseFocus ( m );

            case WM_COMMAND:
                 return command ( m );

            case WM_TIMER:
                 return timer ( m );

            case WM_CLOSE:
                 return close ( m );
      }
       return FALSE;                    // unknown message - not processed
}
int    View    keyDown ( const Message& m )    // let parent process it
{
     return parent != NULL ? parent -> keyDown ( m ) : FALSE;
}
```

```
int    View    keyUp ( const Message& m )                    // let parent process it
{
    return parent != NULL ? parent -> keyUp ( m )   FALSE;
}
int    View    mouseDown ( const Message& m )                // let parent process it
{
    return parent != NULL ? parent -> mouseDown ( m )   FALSE;
}
int    View :: mouseUp ( const Message& m )                  // let parent process it
{
    return parent != NULL ? parent -> mouseUp ( m ) : FALSE;
}
int    View :: mouseMove ( const Message& m )                // let parent process it
{
    return parent != NULL ? parent -> mouseMove ( m ) : FALSE;
}
int    View :: rightMouseDown ( const Message& m )           // let parent process it
{
    return parent != NULL ? parent -> rightMouseDown ( m )   FALSE;
}
int    View :: rightMouseUp ( const Message& m )             // let parent process it
{
    return parent != NULL ? parent -> rightMouseUp ( m ) : FALSE;
}
int    View :: mouseDoubleClick ( const Message& m )  // let parent process it
{
    return parent != NULL ? parent -> mouseDoubleClick ( m ) : FALSE;
}
int    View :: mouseTripleClick ( const Message& m )     // let parent process it
{
    return parent != NULL ? parent -> mouseTripleClick ( m ) : FALSE;
}
int    View :: receiveFocus ( const Message& m )
{
    sendMessage ( delegate, WM_COMMAND, VN_FOCUS, TRUE, this );
    if ( style & WS_REPAINTONFOCUS )
        repaint ();
    return TRUE;                                  // message processed
}
int    View :: looseFocus ( const Message& m )
{
    sendMessage ( delegate, WM_COMMAND, VN_FOCUS, FALSE, this );
    if ( style & WS_REPAINTONFOCUS )
        repaint ();
    return TRUE;                                  // message processed
```

```
}
int    View :: command ( const Message& m )
{
    return FALSE;                          // not processed
}
int    View :: timer ( const Message& m )
{
    return FALSE;
}
int    View :: close ( const Message& m )
{
    hide       ();
    autorelease ();
    return TRUE;                           // processed
}
void   View :: getMinMaxSize ( Point& minSize, Point& maxSize ) const
{
    minSize.x = 0;
    minSize.y = 0;
    maxSize.x = MAXINT;
    maxSize.y = MAXINT;
}
Rect   View :: getClientRect ( Rect& client ) const
{
     client.x1 = 0;
     client.y1 = 0;
     client.x2 = width () - 1;
     client.y2 = height () - 1;
     return client;
}
View  * View :: viewWithTag ( int theTag )
{
    if ( tag == theTag )
        return this;
    View * res = NULL;
    for ( View * v = child; v != NULL; v = v -> prev )
        if ( ( res = v -> viewWithTag ( theTag ) ) != NULL )
            return res;
    return res;
}
void   View :: addSubView ( View * subView )
{
    if ( child != NULL )
        child -> next = subView;
    subView -> prev = child;
    child           = subView;
```

```
}
void    View    setFrame ( int x, int y, int width, int height )
{
        Rect    r ( x, y, x + width - 1, y + height - 1 );
        if ( r != area )
        {
                Rect    updateRect = area;
                updateRect |= r;
                area       = r;
                if ( parent != NULL )
                        parent -> local2Screen ( updateRect );
                screen.rebuildMap ();
                screen.redrawRect ( updateRect );
        }
}
Rect    View :: getScreenRect () const
{
        Rect    r ( area );
        if ( parent != NULL )
                parent -> local2Screen ( r );
        return r;
}
Point&  View    local2Screen ( Point& p ) const
{
        for ( View * w = (View *) this; w != NULL; w = w -> parent )
        {
                p.x += w -> area.x1;
                p.y += w -> area.y1;
        }
        return p;
}
Rect&   View :: local2Screen ( Rect& r ) const
{
        for ( View * w = (View *) this; w != NULL; w = w -> parent )
                r.move ( w -> area.x1, w -> area.y1 );
        return r;
}
Point&  View :: screen2Local ( Point& p ) const
{
        for ( View * w = (View *) this; w != NULL; w = w -> parent )
        {
                p.x -= w -> area.x1;
                p.y -= w -> area.y1;
        }
        return p;
}
```

```
Rect&   View    screen2Local ( Rect& r ) const
{
        for ( View * w = (View *) this; w != NULL; w = w -> parent )
                r.move ( - w -> area.x1, - w -> area.y1 );

        return r;
}
void    View    enableView ( int flag )
{
        if ( flag )
                status |= WS_ENABLED;
        else
                status &= ~WS_ENABLED;
}
void    View :: beginDraw () const
{
     Rect    clipRect ( getScreenRect () );
     Point   org ( 0, 0 );

     clipRect &=  screenRect;

     local2Screen            ( org );
     screenSurface -> setOrg ( org );
     hideMouseCursor         ();
}
void    View :: endDraw () const
{
     showMouseCursor ();
}
void    View :: repaint () const
{
     if ( isVisible () && !isLocked () )
     {
        Rect    clipRect ( getScreenRect () );

          clipRect &= screenRect;

          if ( style & WS_COMPLETELYVISIBLE )
          {
                Rect    drawRect ( 0, 0, width () - 1,
                                                 height () - 1 );

            screenSurface -> beginDraw ();
            beginDraw ();
            draw ( drawRect );
            endDraw ();
            screenSurface -> endDraw ();
          }
          else
                screen.redrawRect ( clipRect );
     }
}
void    View :: lock ( int flag )
```

```
{
        if ( flag )
                lockCount++;
        else
                lockCount--;
}
View *  View :: getFirstTab () const
{
        if ( child == NULL )
                return NULL;
        for ( View * w = child; w -> prev != NULL; w = w -> prev )
                        ;
        while (w!=NULL && !(w->isVisible () && w->isEnabled ()))
                w = w -> next;
        return w;
}
void    View :: setZOrder ( View * behind )
{
        if ( prev != NULL )             // remove from chain
                prev -> next = next;
        if ( next != NULL )
                next -> prev = prev;
      if ( parent != NULL && parent -> child == this )
          parent -> child = prev;
      if ( behind == NULL )             // set above all children
       {
            if ( parent != NULL )
            {
              if ( parent -> child != this )
              {
                    parent -> child -> next = this;
                    prev            = parent -> child;
                    next            = NULL;
                    parent -> child    = this;
              }
            }
      }
      else
      {
            if ( ( prev = behind -> prev ) != NULL )
                 behind -> prev -> next = this;
            next          = behind;
            behind -> prev = this;
      }
      screen.rebuildMap ();
      screen.redrawRect ( getScreenRect () );
}
View  * View :: hookWindow ( View * whome )
```

```
{
     View * oldHook = whome -> hook;
      whome -> hook = this;
      return oldHook;
}
int    View    containsFocus () const
{
      for ( View * w = focusedView; w != NULL; w = w -> parent )
            if ( w == this )
                  return TRUE;
      return FALSE;
}
////////////////////////////////////////////////////////
int    setFocus ( View * view )
{
     if ( focusedView == view )            // check if already focused
          return TRUE;
                        // check whether we can set focus
                        // to this view (it's visible & enabled)
     for ( View * v = view; v != NULL; v = v -> parent )
          if ( ! v -> isVisible () )
               return FALSE;    // cannot set focus on invisible window
     if ( view != NULL && ! view -> isEnabled () )
          return FALSE;                  // cannot set focus on disabled window
     if ( ! view -> canReceiveFocus () )   // view does not want to be input focus
          return FALSE;
                        // inform current focus view
                        // that's it's loosing focus to 'view'
     View * oldFocus = focusedView;    // save focus
     focusedView = NULL;              // so isFocused return FALSE
     sendMessage ( oldFocus, WM_LOOSEFOCUS );// inform focused view that it's
                                                  // loosing input focus inform window th
                                                  // it gets input focus
     sendMessage ( focusedView = view, WM_RECEIVEFOCUS );
     return TRUE;
}
void    redrawRect ( const Rect& r )
{
      screen.redrawRect ( r );
}
View * findView ( const Point& p )
{
      return screen.findView ( p );
}
Object * loadView ()
{
     return new View;
}
```

В этих листингах представлены два основных класса для построения оконного интерфейса - класс Map, отвечающий за разбиение экрана на список видимых прямоугольников, принадлежащих различным окнам, и класс View, являющийся базовым классом для создания различных окон. Основным методом класса Map, служащим для разбиения, является метод addView, осуществляющий разбиение всех прямоугольников из имеющегося списка на части заданным прямоугольником и добавляющий заданный прямоугольник в общий список прямоугольников. В общем случае при разбиении произвольного прямоугольника другим прямоугольником возникает 5 частей (прямоугольников), некоторые из которых могут быть пустыми (рис. 5.11).

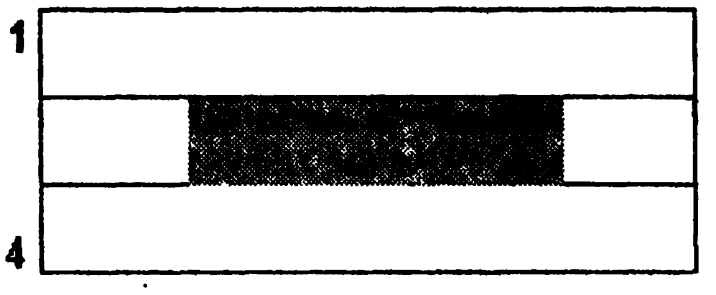


Рис. 5.11

На компакт-диске приводятся несколько программ, написанных с использованием данного модуля.

На компакт-диске вы также найдете ряд снимков системы NextStep как один из самых удачных примеров оконного интерфейса.

Упражнения

1. Для предложенной оконной системы реализуйте основные управляющие элементы (CheckBox, ScrollBar, TextEdit, ListBox и т. д.).
2. Реализуйте файловый диалог, подобный реализованому в Windows 3.1 или Windows 95.
3. На построенном графическом интерфейсе реализуйте редактор уровней для игры типа Wolfenstein 3d (см. гл. 13), позволяющий создавать лабиринты, назначать текстуры стенам и полу, размещать различные объекты - двери, оружие, боеприпасы, аптечки и т. п.
4. Реализуйте окна с прозрачными (полупрозрачными) фрагментами. Реализуйте окна с полупрозрачными тенями.

Глава 6
РАСТРОВЫЕ АЛГОРИТМЫ

Подавляющее число графических устройств являются растровыми, представляя изображение в виде прямоугольной матрицы (сетки, целочисленной решетки) пикселов (растра), и большинство графических библиотек содержат внутри себя достаточное количество простейших растровых алгоритмов, таких, как:

- переведение идеального объекта (отрезка, окружности и др.) в их растровые образы;
- обработка растровых изображений.

Тем не менее часто возникает необходимость и явного построения растровых алгоритмов.

Достаточно важным понятием для растровой сетки является связность - возможность соединения двух пикселов растровой линией, т. е. последовательным набором пикселов. Возникает вопрос, когда пикселы (x_1, y_1) и (x_2, y_2) можно считать соседними.

Вводится два понятия связности:

- 4-связность: пикселы считаются соседними, если либо их x-координаты, либо их y-координаты отличаются на единицу:

  $$|x_1 - x_2| + |y_1 - y_2| \le 1;$$

- 8-связность: пикселы считаются соседними, если их x-координаты и y-координаты отличаются не более чем на единицу:

  $$|x_1 - x_2| \le 1, \quad |y_1 - y_2| \le 1.$$

Понятие 4-связности является более сильным: любые два 4-связных пиксела являются и 8-связными, но не наоборот. На рис. 6.1 изображены 8-связная линия (*а*) и 4-связная линия (*б*).

В качестве линии на растровой сетке выступает набор пикселов $P_1, P_2, \ldots, P_n$, где любые два пиксела P_i, P_{i+1} являются соседними в смысле заданной связности.

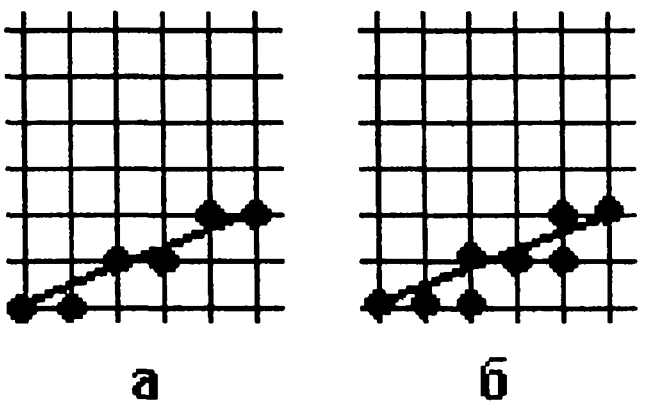


Рис. 6.1

Замечание. Так как понятие линии базируется на понятии связности, то естественным образом возникает понятие 4- и 8-связных линий. Поэтому, когда мы говорим о растровом представлении (например, отрезка), следует ясно понимать, о каком именно представлении идет речь. В общем случае растровое представление объекта не является единственным и возможны различные способы его построения.

6.1. Растровое представление отрезка. Алгоритм Брезенхейма

Рассмотрим задачу построения растрового изображения отрезка, соединяющего точки $A(x_a, y_a)$ и $B(x_b, y_b)$. Для простоты будем считать, что $0 \le y_b - y_a \le x_b - x_a$ Тогда отрезок описывается уравнением

$$y = y_a + \frac{y_b - y_a}{x_b - x_a}(x - x_a), x \in [x_a, x_b], \text{ или } y = kx + b,$$

где

$$k = \frac{y_b - y_a}{x_b - x_a},$$

$$b = y_a - kx_a$$

Отсюда получаем простейший алгоритм растрового представления отрезка:

```
// File line1.cpp
void line ( int xa, int ya, int xb, int yb, int color )
{
  double k = ((double)(yb-ya))/(xb-xa);
  double b = ya - k*xa;
  for ( int x = xa; x <= xb; x++ )
      putpixel ( x, (int)( k*x + b ), color );
}
```

Вычислений значений функции $y = kx + b$ можно избежать, используя в цикле рекуррентные соотношения, так как при изменении x на 1 значение y изменяется на k.

```
// File line2.cpp
void line ( int xa, int ya, int xb, int yb, int color )
{
  double k = ((double)(yb-ya))/(xb-xa);
  double y = ya;
  for ( int x = xa; x <= xb; x++, y += k )
      putpixel ( x, (int) y, color );
}
```

Однако взятие целой части y может приводить к не всегда корректному изображению (рис. 6.2).

Улучшить внешний вид получаемого отрезка можно за счет округления значений y до ближайшего целого. Фактически это означает, что из двух возможных кандидатов (пикселов, расположенных друг над другом так, что прямая проходит между ними) всегда выбирается тот пиксел, который лежит ближе к изображаемой прямой (рис. 6.3). Для этого достаточно сравнить дробную часть y с 1/2.

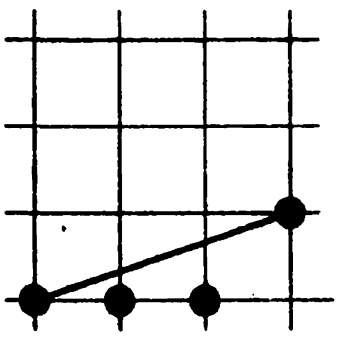


Рис. 6.2

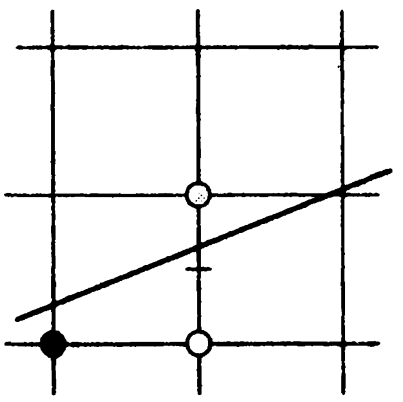


Рис. 6.3

Пусть $x_0 = x_a, y_0, \ldots, x_n = x_b, y_n = y_b$ - последовательность изображаемых пик причем $x_{i+1} - x_i = 1$. Тогда каждому значению xi соответствует число $kx_i + b$.

Обозначим через c_i дробную часть соответствующего значения фу $kx_i + b$ - $c_i = \{kx_i + b\}$

Тогда, если $c_i \le 1/2$, положим

$$y_i = [kx_i + b],$$

в противном случае - $y_i = [kx_i + b] + 1$.

Рассмотрим, как изменяется величина c_i при переходе от x_i к следующем чению x_{i+1}.

Само значение функции при этом изменяется на k.

Если $c_i + k \le 1/2$, то

$$c_{i+1} = c_i + k, y_{i+1} = y_i$$

В противном случае необходимо увеличить y на единицу и тогда прихо следующим соотношениям:

$$c_{i+1} = c_i + k - 1, y_{i+1} = y_i + 1,$$

так как

$$kx_i + b = y_i + c_i$$

$$kx_{i+1} + b = y_{i+1} + c_{i+1}$$

а y_{i+1} - целочисленная величина.

Заметим, что $c_0 = 0$, так как точка (x_0, y_0) лежит на прямой $y = kx + b$

Приходим к следующей программе:

```
// File line3.cpp
void line ( int xa, int ya, int xb, int yb, int color )
{
  double k = ((double)(yb-ya))/(xb-xa);
  double c = 0;
  int    y = ya;

  putpixel ( xa, ya, color );
  for ( int x = xa + 1; x <= xb; x++ )
  {
        if ( ( c += k ) > 0.5 )
        {
            c -= 1;
```

```
            y++;
        }
        putpixel ( x, y, color );
        }
    }
```

Замечание. *Выбор точки можно трактовать и так: рассматривается середина отрезка между возможными кандидатами и проверяется, где (выше или ниже этой середины) лежит точка пересечения отрезка прямой, после чего выбирается соответствующий пиксел. Это метод срединной точки (midpoint algorithm).*

Сравнивать с нулем удобнее, чем с 1/2, поэтому введем новую вспомогательную величину $d_i = 2c_i - 1$, заметив, что $d_i = 2k - 1$ (так как $c_1 = k$).

Получаем следующую программу:

```
// File line4.cpp
void line ( int xa, int ya, int xb, int yb, int color )
{
    double k = ((double)(yb-ya))/(xb-xa);
    double d = 2*k - 1;
    int    y = ya;

    putpixel ( xa, ya, color );
    for ( int x = xa + 1; x <= xb; x++ )
     {
            if ( d > 0 )
            {
                d += 2*k - 2;
                y++;
                }
                else
                d += 2*k;
        putpixel ( x, y, color );
        }
}
```

Несмотря на то, что и входные данные являются целочисленными величинами и все операции ведутся на целочисленной решетке, алгоритм использует операции с вещественными числами. Чтобы избавиться от необходимости их использования, заметим, что все вещественные числа, присутствующие в алгоритме, являются числами вида $\frac{p}{\Delta x}, p \in Z$. Поэтому если домножить величины d_i и k на $\Delta x = x_b - x_a$, то в результате останутся только целые числа. Тем самым мы приходим к алгоритму Брезенхейма

```
// File line5.cpp
// simplest Bresenham's alg. 0 <= y2 - y1 <= x2 - x1
void  line ( int xa, int ya, int xb, int yb, int color )
{
  int  dx = xb - xa;
  int  dy = yb - ya;
```

```
  int  d  = ( dy << 1 ) - dx;
  int  d1 = dy << 1;
  int  d2 = ( dy - dx ) << 1;
  putpixel ( xa, ya, color );
  for ( int x = xa + 1, y = y1; x <= xb; x++ )
  {
   if ( d > 0 )
   {
    d += d2;
    y += 1;
   }
   else
       d += d1;

   putpixel ( x, y, color );
  }
}
```

Известно, что этот алгоритм дает наилучшее растровое приближение отрезка.

Из предложенного примера несложно написать функцию для построения 4-связной развертки отрезка.

```
// File line_4.cpp
void  line_4 ( int x1, int y1, int x2, int y2, int color )
{
 int  dx = x2 - x1;
 int  dy = y2 - y1;
 int  d  = 0;
 int  d1 = dy << 1;
 int  d2 = - ( dx << 1 );
 putpixel ( x1, y1, color );
 for ( int x = x1, y = y1, i = 1; i <= dx + dy; i++ ) {
  if ( d > 0 )
     {
     d += d2;
       y += 1;
  }
  else
     {
    d += d1;
      x += 1;
  }
  putpixel ( x, y, color );
 }
}
```

Общий случай произвольного отрезка легко сводится к рассмотренному выше; следует только иметь в виду, что при выполнении неравенства $|\Delta y| \geq |\Delta x|$ необходимо x и y поменять местами. Полный текст соответствующей программы приводится ниже.

```
// File line6.cpp
// Bresenhames alg.
```

```
Void line ( int x1, int y1, int x2, int y2, int color )
{
    int  dx = abs ( x2 - x1 );
    int  dy = abs ( y2 - y1 );
    int  sx = x2 >= x1 ? 1 : -1;
    int  sy = y2 >= y1 ? 1   -1;

    if ( dy <= dx )
    {
        int  d  = ( dy << 1 ) - dx;
        int  d1 = dy << 1;
        int  d2 = ( dy - dx ) << 1;

        putpixel ( x1, y1, color );

        for (int x=x1+sx, y=y1, i=1; i <= dx; i++, x+=sx)
        {
            if ( d > 0 )
            {
                d += d2;
                y += sy;
            }
            else
                d += d1;

            putpixel ( x, y, color );
        }
    }
    else
    {
        int  d  = ( dx << 1 ) - dy;
        int  d1 = dx << 1;
        int  d2 = ( dx - dy ) << 1;

        putpixel ( x1, y1, color );

        for (int x=x1, y=y1+sy, i=1; i <= dy; i++, y+=sy)
        {
            if ( d > 0 )
            {
                d += d2;
                x += sx;
            }
            else
                d += d1;

            putpixel ( x, y, color );
        }
    }
}
```

6.2. Растровая развертка окружности

Для упрощения алгоритма растровой развертки стандартной окружности можно пользоваться ее симметрией относительно координатных осей и прямых $y = \pm x$

(в случае, когда центр окружности не совпадает с началом координат, эти прямые необходимо сдвинуть параллельно так, чтобы они прошли через центр окружности). Тем самым достаточно построить растровое представление для 1/8 части окружности, а все оставшиеся точки получить симметрией. С этой целью введем следующую процедуру:

```
static   void     circlePoints ( int x, int y, int color )
{
    putpixel ( xCenter + x, yCenter + y, color );
    putpixel ( xCenter + y, yCenter + x, color );
    putpixel ( xCenter + y, yCenter - x, color );
    putpixel ( xCenter + x, yCenter - y, color );
    putpixel ( xCenter - x, yCenter - y, color );
    putpixel ( xCenter - y, yCenter - x, color );
    putpixel ( xCenter - y, yCenter + x, color );
    putpixel ( xCenter - x, yCenter + y, color );
}
```

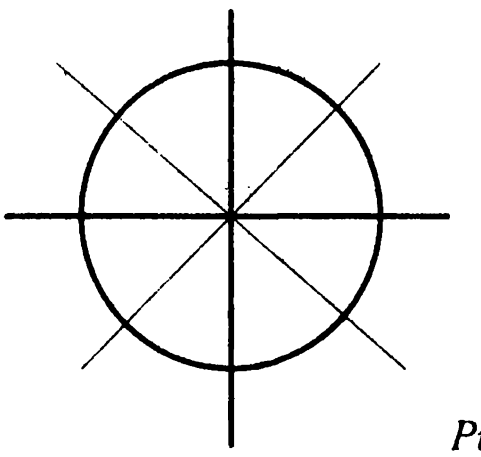


Рис. 6.4

Рассмотрим участок окружности из второго октанта

$x \in [0, R/\sqrt{2}], y \in [R/\sqrt{2}, R]$.

Особенностью данного участка является то обстоятельство, что угловой коэффициент касательной к окружности не превосходит 1 по модулю, а точнее, лежит между -1 и 0.

Применим к этому участку алгоритм средней точки (midpoint algorithm).

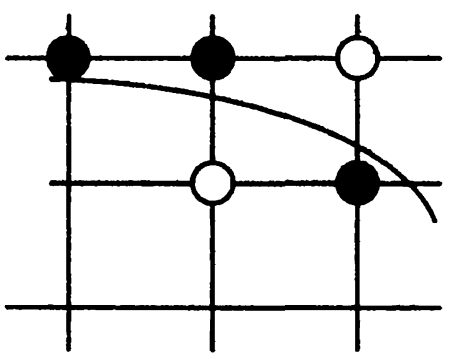


Рис. 6.5

Функция $F(x, y) = x^2 + y^2 - R^2$, определяющая окружность, обращается в нуль на самой окружности, отрицательна внутри окружности и положительна вне ее.

Пусть точка (x_i, y_i) уже поставлена. Для определения того, какое из двух значений y (y_i или) следует взять в качестве y_{i+1}, введем переменную

$d_i = F(x_i + 1, y_i - ½) = (x_i + 1)^2 + (y_i - ½)^2 - R^2$

В случае, когда $d_i < 0$, полагаем $y_{i+1} = y_i$.Тогда

$d_{i+1} = F(x_i + 2, y_i - ½) = (x_i + 2)^2 + (y_i - ½)^2 - R^2$,

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

В случае, когда $d_i \geq 0$, делаем шаг вниз, выбирая $y_{i+1} = y_i + 1$. Тогда

$d_{i+1} = F(x_i + 2, y_i - 3/2) = (x_i + 2)^2 + (y_i - 3/2)^2 - R^2$,

$\Delta d_i = 2(x_i - y_i) + 5$

Таким образом, мы определили итерационный механизм перехода от одного пиксела к другому. В качестве стартового пиксела берется. Тогда

$d_0 = F(1, R - ½) = 5/4 - R$

и мы приходим к алгоритму

```
// File circle1.cpp
static   int  xCenter;
static   int  yCenter;

static   void     circlePoints ( int x, int y, int color )
{
```

```
        putpixel ( xCenter + x, yCenter + y, color );
        putpixel ( xCenter + y, yCenter + x, color );
        putpixel ( xCenter + y, yCenter - x, color );
        putpixel ( xCenter + x, yCenter - y, color );
        putpixel ( xCenter - x, yCenter - y, color );
        putpixel ( xCenter - y, yCenter - x, color );
        putpixel ( xCenter - y, yCenter + x, color );
        putpixel ( xCenter - x, yCenter + y, color );
}
void    circle1 ( int xc, int yc, int r, int color )
{
    int  x = 0;
    int  y = r;
    float    d = 1.25 - r;

    xCenter = xc;
    yCenter = yc;

    CirclePoints ( x, y, color );

    while ( y > x )
    {
        if ( d < 0 )
        {
            d += 2*x + 3;
            x++;
        }
        else
        {
            d += 2*(x - y ) + 5;
            x++;
            y--;
        }
        circlePoints ( x, y, color );
    }
}
```

Заметим, что величина d_i всегда имеет вид

$¼ + z, z \in Z,$

и, значит, изменяется только на целое число. Поэтому дробную часть (всегда равную 1/4) можно отбросить, перейдя тем самым к полностью целочисленному алгоритму.

```
// File circle1.cpp
void    circle2 ( int xc, int yc, int r, int color )
{
    int  x     = 0;
    int  y     = r;
    int  d     = 1 - r;
    int  delta1 = 3;
    int  delta2 = -2*r + 5;

    xCenter = xc;
    yCenter = yc;
```

```
        circlePoints ( x, y, color );
        while ( y > x )
        {
            if ( d < 0 )
            {
                d     += delta1;
                delta1 += 2;
                delta2 += 2;
                x++;
            }
            else
            {
                d     += delta2;
                delta1 += 2;
                delta2 += 4;
                x++;
                y--;
            }
            circlePoints ( x, y, color );
        }
}
```

6.3. Растровая развертка эллипса

Уравнение эллипса с осями, параллельными координатным осям, имеет следующий вид:

$$\frac{x^2}{a^2}+\frac{y^2}{b^2}=1.$$

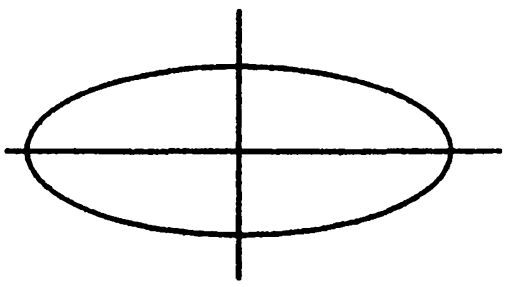


Рис. 6.6

Перепишем это уравнение несколько иначе:

$$F(x,y)=b^2x^2+a^2y^2-a^2b^2=0$$

В силу симметрии эллипса относительно координатных осей достаточно найти растровое представление только для одной из его четвертей, лежащей в первом квадранте координатной плоскости: $x \geq 0, y \geq 0$

Разобьем четверть эллипса на две части: ту, где угловой коэффициент лежит между -1 и 0, и ту, где угловой коэффициент меньше -1 (рис. 6.7).

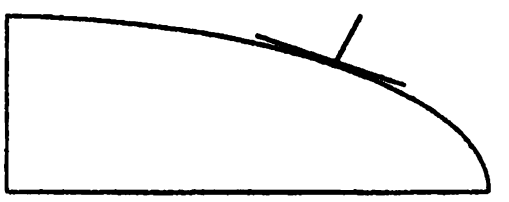


Рис. 6.7

Вектор, перпендикулярный эллипсу в точке (x, y), имеет вид

$$gradF(x,y)=\left(\frac{\partial F}{\partial x},\frac{\partial F}{\partial y}\right)=\left(2b^2x,2a^2y\right).$$

В точке, разделяющей части 1 и 2, $b^2x=a^2y$ Поэтому y-компонента градиента в области 1 больше x-компоненты (в области 2 - наоборот). Таким образом, если в следующей срединной точке

$$a^2\left(y_i-\frac{1}{2}\right)\le b^2\left(x_i+1\right),$$

то мы переходим из области 1 в область 2.

Как и в любом алгоритме средней точки, мы вычисляем значение F между кандидатами и используем знак функции для определения того, лежит ли средняя точка внутри эллипса или вне его.

- Часть 1. Если текущий пиксел равен (x_i, y_i), то

$$d_i = F\left(x_i+1, y_i-\frac{1}{2}\right)=b^2\left(x_i+1\right)^2+a^2\left(y_i-\frac{1}{2}\right)^2-a^2b^2$$

При $d_i<0$ полагаем $y_{i+1}=y_i$ и

$$d_{i+1}=F\left(x_i+2, y_i-\frac{1}{2}\right)=b^2\left(x_i+2\right)^2+a^2\left(y_i-\frac{1}{2}\right)^2-a^2b^2,$$

$$\Delta d_i=b^2\left(2x_i+3\right).$$

При $d_i\ge 0$ полагаем $y_{i+1}=y_i+1$ и

$$d_{i+1}=F\left(x_i+2, y_i-\frac{3}{2}\right)=b^2\left(x_i+2\right)^2+a^2\left(y_i-\frac{3}{2}\right)^2-a^2b^2,$$

$$\Delta d_i=b^2\left(2x_i+3\right)+a^2\left(2-2y_i\right).$$

- Часть 2. Если текущий пиксел - (x_i, y_i), то

$$d_i=F\left(x_i+\frac{1}{2}, y_i-1\right)$$

и все дальнейшие выкладки проводятся аналогично первому случаю.

Часть 2 начинается в точке (0, b), и (1, b – ½) - первая средняя точка. Поэтому

$$d_0=F\left(1, b-\frac{1}{2}\right)=b^2+a^2\left(-b+\frac{1}{4}\right).$$

На каждой итерации в части 1 мы должны не только проверять знак переменной d_i, но и, вычисляя градиент в средней точке, следить за тем, не пора ли переходить в часть 2.

6.4. Закраска области, заданной цветом границы

Рассмотрим область, ограниченную набором пикселов заданного цвета, и точку (x, y), лежащую внутри этой области.

Задача заполнения области заданным цветом в случае, когда область не является выпуклой, может оказаться довольно сложной.

Простейший алгоритм

```
// File fill1.cpp
void pixelFill ( int x, int y, int borderColor, int color )
{
  int c = getpixel ( x, y );
  if ( ( c != borderColor ) && ( c != color ) )
  {
     putpixel ( x, y, color );
     pixelFill ( x - 1, y, borderColor, color );
     pixelFill ( x + 1, y, borderColor, color );
     pixelFill ( x, y - 1, borderColor, color );
     pixelFill ( x, y + 1, borderColor, color );
  }
}
```

хотя и абсолютно корректно заполняющий даже самые сложные области, является слишком неэффективным, так как для всякого уже отрисованного пиксела функция вызывается еще 3 раза и, кроме того, этот алгоритм требует слишком большого стека из-за большой глубины рекурсии. Поэтому для решения задачи закраски области предпочтительнее алгоритмы, способные обрабатывать сразу целые группы пикселов, т. е. использовать их "связность" - если данный пиксел принадлежит области, то скорее всего его ближайшие соседи также принадлежат данной области.

Ясно, что по заданной точке (x, y) отрезок $[x_l, x_r]$ максимальной длины, проходящий через эту точку и целиком содержащийся в области, построить несложно. После заполнения этого отрезка необходимо проверить точки, лежащие непосредственно над и под ним. Если при этом мы найдем незаполненные пикселы, принадлежащие данной области, то для их обработки рекурсивно вызывается функция.

Этот алгоритм намного эффективнее предыдущего и способен работать с областями самой сложной формы (рис. 6.8).

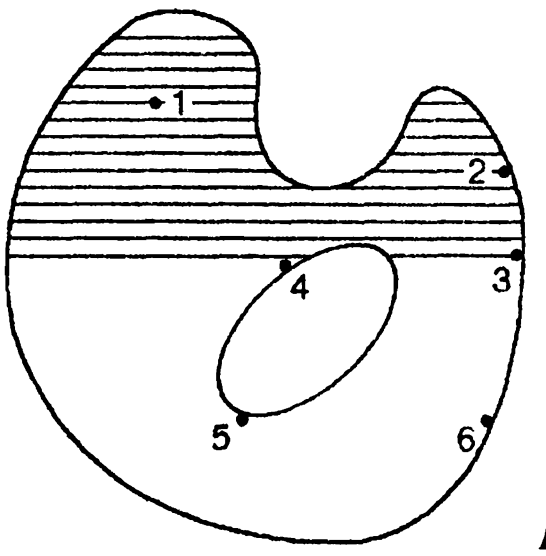


Рис. 6.8

```
// File fill2.cpp
#include  <conio.h>
#include  <graphics.h>
#include  <process.h>
#include  <stdio.h>
#include  <stdlib.h>
int  borderColor = WHITE;
int  color       = GREEN;

int  lineFill ( int x, int y, int dir, int prevXl, int prevXr )
{
  int  xl = x;
  int  xr = x;
  int  c;
       // find line segment
  do    c = getpixel ( --xl, y );
  while ( ( c != borderColor ) && ( c != color ) );
  do    c = getpixel ( ++xr, y );
  while ( ( c != borderColor ) && ( c != color ) );
  xl++;  xr--;
```

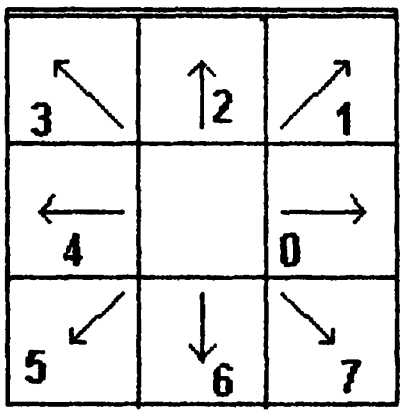
```
  line ( xl, y, xr, y );  // fill segment
       // fill adjacent segments in the same direction
  for ( x = xl; x<= xr; x++ )
  {
    c = getpixel ( x, y + dir );
    if ( ( c != borderColor ) && ( c != color ) )
      x = lineFill ( x, y + dir, dir, xl, xr );
  }
  for ( x = xl; x < prevXl; x++ )
  {
    c = getpixel ( x, y - dir );
    if ( ( c != borderColor ) && ( c != color ) )
      x = lineFill ( x, y - dir, -dir, xl, xr );
  }
  for ( x = prevXr; x < xr; x++ )
  {
    c = getpixel ( x, y - dir );
    if ( ( c != borderColor ) && ( c != color ) )
      x = lineFill ( x, y - dir, -dir, xl, xr );
  }
  return xr;
}
void  fill ( int x, int y )
{
  lineFill ( x, y, 1, x, x );
}
main ()
{
  int  driver = DETECT;
  int  mode;
  int  res;
  initgraph ( &driver, &mode, "" );
  if ( ( res = graphresult () ) != grOk )
  {
    printf("\nGraphics error: %s\n", grapherrormsg ( res) );
    exit ( 1 );
  }
  circle ( 320, 200, 140 );
  circle ( 260, 200, 40 );
  circle ( 380, 200, 40 );
  getch ();
  setcolor ( Color );
  fill ( 320, 300 );
  getch ();
  closegraph ();
}
```

Существует и другой подход к заполнению области сложной формы, заключающийся в определении ее границы и последовательном заполнении горизонтальных участков между граничными пикселами.

Такой алгоритм имеет следующую структуру:

- построение упорядоченного списка граничных пикселов (отслеживается в няя граница);
- проверка внутренности (для обнаружения в ней дыр);
- заполнение области горизонтальными отрезками, соединяющими точки границы

Занумеруем возможные ходы для перебора соседей на 8-связной решетке и проведем упорядоченный перебор пикселов, соседних с уже найденным граничным, в зависимости от направления предыдущего хода.



Ри

Ниже приводится программа заполнения области на основе этого алгоритма.

```
// File fill3.cpp
#define BLOCKED      1
#define UNBLOCKED    2
#define FALSE        0
#define TRUE         0xFFFF

int  borderColor = WHITE;
int  fillColor   = GREEN;

struct   BPStruct          // table of border pixels
{
    int  x, y;
    int  flag;
} bp [3000];

int      bpStart;
int      bpEnd = 0;
BPStruct    currentPixel;
int      D;                // current search direction
int      prevD;            // prev. search direction
int      prevV;            // prev. vertical direction

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

void     appendBPList ( int x, int y, int flag )
{
    bp [bpEnd  ].x    = x;
    bp [bpEnd  ].y    = y;
    bp [bpEnd++].flag = flag;
}

void     sameDirection ()
{
    if ( prevD == 0 )                       // moving right
        bp [bpEnd-1].flag = BLOCKED ;// block previous pixel
    else
    if ( prevD != 4 )                       // if not moving horizontally
        appendBPList ( currentPixel.x, currentPixel.y,          UNBLOCKED);
}

void     differentDirection ( int d )
```

```
{
    if ( prevD == 4 )      // previously moving left
    {
        if ( prevV == 5 ) // if from above block rightmost in line
            bp [bpEnd-1].flag = BLOCKED;
        appendBPList ( currentPixel.x, currentPixel.y,          BLOCKED);
    }
    else
    if ( prevD == 0 )      // previously moving right
    {                   // block rightmost in line
        bp [bpEnd-1].flag = BLOCKED;
        if ( d == 7 )      // if line started from above
            appendBPList ( currentPixel.x,      currentPixel.y, BLOCKED);
        else
            appendBPList ( currentPixel.x,      currentPixel.y, UNBLOCKED);
    }
    else                   // prev. moving in some
    {                      // vert. dir.
        appendBPList ( currentPixel.x,          currentPixel.y, UNBLOCKED);
                           // add pixel twice if local min or max
        if ( ( ( d >= 1 ) && ( d <= 3 ) ) &&
                        ( ( prevD >= 5 ) && ( prevD <= 7 ) ) ||
             ( ( d >= 5 ) && ( d <= 7 ) ) &&
           ( ( prevD >= 1 ) && ( prevD <= 3 ) ) )
                AppendBPList ( currentPixel.x,  currentPixel.y, UNBLOCKED);
    }
}
addBPList ( int d )
{
    if ( d == prevD )
        sameDirection ();
    else
    {
        differentDirection ( d );
        prevV = prevD; // new previous vertical direction
    }
    prevD = d;        // new previous search direction
}
void    nextXY ( int& x, int& y, int direction )
{
    switch ( direction )      // 3 2 1
    {                         // 4   0
        case 1:               // 5 6 7
        case 2:
        case 3:
            y--;      // go up
            break;
        case 5:
        case 6:
```

```
            case 7:
                y++;     // go down
                break;
        }
        switch ( direction )
        {
            case 3:
            case 4:
            case 5:
                x--;     // go left
                break;
            case 1:
            case 0:
            case 7:
                x++;     // go right
                break;
        }
}
int  findBP ( int d )
{
    int  x = currentPixel.x;
    int  y = currentPixel.y;
    nextXY ( x, y, d );          // get x,y of pixel in direction d
    if ( getpixel ( x, y ) == borderColor )
    {
        addBPList ( d ); // add pixel x,y to the table
        currentPixel.x = x;   // pixel at x,y becomes
        currentPixel.y = y;   // current pixel
        return TRUE;
    }
    return FALSE;
}
int  findNextPixel ()
{
    for ( int i = -1; i <= 5; i++ )
    {                    // find next border pixel
        int  flag = findBP ( ( D + i ) & 7 );
        if ( flag )              // flag is TRUE if found
        {
            D = ( D + i ) & 6; // (D+i) MOD 2
            return flag;
        }
    }
}
int  scanRight ( int x, int y )
{
    while ( getpixel ( x, y ) != borderColor )
```

```
        if ( ++x == 639 )
            break;
    return x;
}
void    scanRegion ( int& x, int& y )
{
    for ( int i = bpStart; i < bpEnd; )
    {                   // skip pixel if blocked
        if ( bp [i].flag == BLOCKED )
            i++;
        else                // skip last pixel in line
        if ( bp [i].y != bp [i+1].y )
            i++;
        else
        {               // if at least one pixel
                        // to fill then scan the line
            if ( bp [i].x < bp [i+1].x - 1 )
            {
                int  xr = scanRight ( bp [i].x + 1, bp [i].y );
                if ( xr < bp [i+1].x )
                {
                    x = xr;
                    y = bp [i].y;
                    break;
                }
                i += 2;
            }
        }
    bpStart = i;
}
int  compareBP ( BPStruct * arg1, BPStruct * arg2 )
{
    int  i = arg1 -> y - arg2 -> y;
    if ( i != 0 )
        return i;
    if ( ( i = arg1 -> x - arg2 -> x ) != 0 )
        return i;
    return arg1 -> flag - arg2 -> flag;
}
void    sortBP ()
{
    qsort ( bp + bpStart, bpEnd - bpStart, sizeof ( BPStruct ),
        (int (*)(const void *, const void *)) compareBP );
}
void    fillRegion ()
{
```

```
    for ( int i = 0; i < bpEnd; )
    {
        if ( bp [i].flag == BLOCKED )     // skip pixel
            i++;                  // if blocked
        else
        if ( bp [i].y != bp [i+1].y ) // skip pixel
            i++;                  // if last in line
        else                      // if at least one pixel to fill
        {                   // draw a line
            if ( bp [i].x < bp [i+1].x - 1 )
                line ( bp [i].x + 1, bp [i].y, bp [i+1].x - 1, bp [i+1].y );

            i += 2;
        }
    }
}
void    traceBorder ( int startX, int startY )
{
    int  nextFound;
    int  done;

    currentPixel.x = startX;
    currentPixel.y = startY;
    D              = 6;       // current search direction
    prevD          = 8;       // previous search direction
    prevV          = 2;       // most recent vertical direction

    do             // loop around the border
    {              // until returned to starting pixel
        nextFound = findNextPixel ();
        done      = (currentPixel.x == startX) &&          (currentPixel.y == startY);
    } while ( nextFound && !done );

    if ( !nextFound )// if only one pixel in border
    {              // add it twice to the table
        appendBPList ( startX, startY, UNBLOCKED );
        appendBPList ( startX, startY, UNBLOCKED );
    }
    else               // if last search direction was upward
                   // add the starting pixel to the table
    if ( (prevD <= 3) && (prevD >= 1) )
        appendBPList ( startX, startY, UNBLOCKED );
}

void borderFill ( int x, int y )
{
    do                 // do until entire table
    {                  //is skanned
        traceBorder ( x, y ); // trace border startint at x,y
        sortBP      ();              // sort border pixel table
        scanRegion  ( x, y );        // look for holes in the interior
    } while ( bpStart < bpEnd );

    fillRegion ();                   // use the table to fill the interior
}
```

Процедура traceBorder создает таблицу всех граничных пикселов области, которая затем сортируется по возрастанию координат *x* и *y* процедурой sortBP. После этого процедура scanRegion проверяет внутренний отрезок прямой между каждой парой граничных пикселов из таблицы. Если в отрезке обнаруживается граничный пиксел, то процедура полагает, что в области найдено отверстие, и возвращает координаты обнаруженного граничного пиксела для того, чтобы процедуры traceBorder и sortBP могли внести в таблицу координаты точек границы этого отверстия. После того как будет обследована вся внутренность области, процедура fillRegion осуществляет заполнение области на основе построенной таблицы.

Процедура traceBorder начинает работу с заданного пиксела на правой границе области и движется вдоль границы по часовой стрелке, при этом внутренность области всегда лежит справа. Если пиксел не соседствует с внутренней частью области, то алгоритм его граничным не считает. Так как алгоритм всегда проверяет первыми пикселы, лежащие справа по направлению движения, то все обнаруженные им граничные пикселы действительно прилегают к внутренней области. Подробнее об этом алгоритме можно прочитать в [11].

6.5. Заполнение многоугольника

Рассмотрим, каким образом можно заполнить многоугольник, задаваемый замкнутой ломаной линией без самопересечений.

Приведенная ниже процедура осуществляет заполнение треугольника, заданного массивом своих вершин (рис. 6.10). Для отслеживания изменения *x*-координат вдоль ребер используются числа с фиксированной точкой в формате 16.16 (подробнее об этом - в приложении).

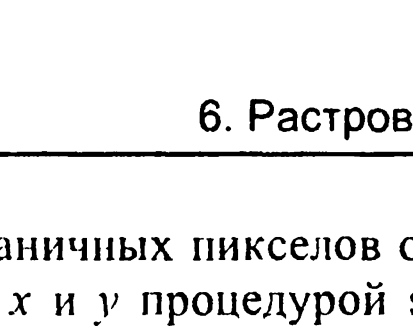


Рис. 6.10

```
// File filltr.cpp
#include    <graphics.h>
#include    "FixMath.h"

struct   Point
{
    int  x;
    int  y;
};

void     fillTriangle ( Point p [] )
{
    int  iMax = 0;
    int  iMin = 0;
    int  iMid = 0;

    for ( int i = 1; i < 3; i++ )      // find indices of top
        if ( p [i].y < p [iMin].y )    // and bottom vertices
            iMin = i;
        else
```

```
    if ( p [i].y > p [iMax].y )
        iMax = i;
iMid = 3 - iMin - iMax;          // find index of
                                 // middle item
Fixed   dx01 = p [iMax].y != p [iMin].y ?
        int2Fixed ( p [iMax].x - p [iMin].x ) /
        ( p [iMax].y - p [iMin].y ) : 0l;
Fixed   dx02 = p [iMin].y != p [iMid].y ?
        Int2Fixed ( p [iMid].x - p [iMin].x ) /
        ( p [iMid].y - p [iMin].y ) : 0l;
Fixed   dx21 = p [iMid].y != p [iMax].y ?
        Int2Fixed ( p [iMax].x - p [iMid].x ) /
        ( p [iMax].y - p [iMid].y ) : 0l;
Fixed   x1 = int2Fixed ( p [iMin].x );
Fixed   x2 = x1;
for ( i = p [iMin].y; i <= p [iMid].y; i++ )
{
    line ( fixed2Int ( x1 ), i, fixed2Int ( x2 ), i );
    x1 += dx01;
    x2 += dx02;
}
for ( i = p [iMid].y + 1; i <= p [iMax].y; i++ )
{
    x1 += dx01;
    x2 += dx21;
    line ( fixed2Int ( x1 ), i, fixed2Int ( x2 ), i );
  }
}
```

Прежде чем перейти к общему случаю, рассмотрим, как осуществляется заполнение выпуклого многоугольника. Заметим, что невырожденный выпуклый многоугольник может содержат не более двух горизонтальных отрезков - вверху и внизу.

Для заполнения выпуклого многоугольника найдем самую верхнюю точку и определим два ребра, выходящие из нее. Если одно из них (или оба) являются горизонтальными, то перейдем в соответствующем направлении к следующему ребру, и так до тех пор, пока у нас не получатся два ребра, идущие вниз (при этом они могут выходить из разных точек, но эти точки будут иметь одинаковую ординату).

Аналогично заполнению треугольника найдем приращения для x-координат для каждого из ребер и будем спускаться вниз, рисуя соответствующие линии. При этом необходимо проверять, не проходит ли следующая линия через вершину одного из двух ведущих ребер. В случае прохождения мы переходим к следующему ребру и заново вычисляем приращение для x-координаты.

Процедура, реализующая описанный алгоритм, приводится ниже.

```
// File fillconv.cpp
static   int  findEdge ( int& i, int dir, int n, const Point p [] )
{
    for ( ; ; )
```

```
    {
        int  i1 = i + dir;

        if ( i1 < 0 )
            i1 = n - 1;
        else
        if ( i1 >= n )
            i1 = 0;

        if ( p [i1].y < p [i].y ) // edge [i,i1] is going upwards
            return -1;          // must be some error
        else
        if ( p [i1].y == p [i].y )     // horizontal edge
            i = i1;
        else                       // edge [i, i1] is going downwords
            return i1;
    }
}
void    fillConvexPoly ( int n, const Point p [] )
{
    int  yMin         = p [0].y;
    int  yMax         = p [0].y;
    int  topPointIndex = 0;
                              // find y-range and
    for ( int i = 1; i < n; i++ )       // topPointIndex
        if ( p [i].y < p [topPointIndex].y )
            topPointIndex = i;
        else
        if ( p [i].y > yMax )
            yMax = p [i].y;

yMin = p [topPointIndex].y;

if ( yMin == yMax )        // degenerate polygon
{
    int  xMin = p [0].x;
    int  xMax = p [0].x;
                    // find it's x-range
    for ( i = 1; i < n; i++ )
        if ( p [i].x < xMin )
            xMin = p [i].x;
        else
        if ( p [i].x > xMax )
            xMax = p [i].x;
                    // fill it
    line ( xMin, yMin, xMax, yMin );

    return;
}

int  i1, i1Next;
int  i2, i2Next;

i1      = topPointIndex;
i1Next = findEdge ( i1, -1, n, p );
```

```
i2      = topPointIndex;
i2Next = findEdge ( i2, 1, n, p );

Fixed   x1   = int2Fixed ( p [i1].x );
Fixed   x2   = int2Fixed ( p [i2].x );
Fixed   dx1 = fraction2Fixed ( p [i1Next].x - p [i1].x, p [i1Next].y - p [i1].y );
Fixed   dx2 = fraction2Fixed ( p [i2Next].x - p [i2].x, p [i2Next].y - p [i2].y );

for ( int y = yMin; y <= yMax; y++ )
{
    line ( fixed2Int ( x1 ), y, fixed2Int ( x2 ), y );

    x1 += dx1;
    x2 += dx2;

    if ( y + 1 == p [i1Next].y )
    {
        i1 = i1Next;              // switch to next edge
        if ( --i1Next < 0 )
            i1Next = n - 1;       // check for lower
        if ( p [i1].y == p [i1Next].y )   // horizontal
            break;                        // part

        dx1 = fraction2Fixed (p[i1Next].x-p[i1].x,
                              p [i1Next].y - p [i1].y );
    }

    if ( y + 1 == p [i2Next].y )
    {
        i2 = i2Next;              // switch to next edge
        if ( ++i2Next >= n )
            i2Next = 0;
                                  // check for lower
        if ( p [i2].y == p [i2Next].y )   // horizontal
            break;                        // part

        dx2 = fraction2Fixed (p [i2Next].x - p [i2].x,
                              p [i2Next].y - p [i2].y );
    }
  }
}
```

Заполнение произвольного многоугольника, заданного выпуклой ло ей без самопересечений, будем осуществлять методом критических точе

Критическая точка - это вершина, *y*-координата которой или является локальным минимумом или представляет одну точку из последовательного набора точек, образующего локальный минимум. В многоугольнике, представленном на рис. 6.11, критическими точками являются вершины 0, 2 и 5. Алгоритм начинается с построения массива критических точек и его сортировки по *y*-координате.

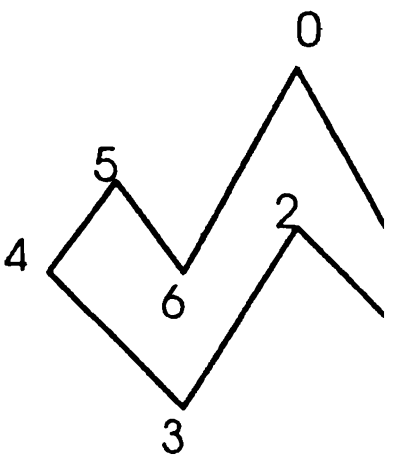


Рис. 6.11

Таблица активных ребер (Active Edge Table - AET) инициализируется как пустая. Далее находятся минимальное и максимальное значения *y*. Затем для каждой строки очередные критические точки проверяются на принадлежность к этой строке. В случае, если критическая точка лежит на строке, отслеживаются два ребра, идущих от нее вниз, которые затем добавляются в таблицу активных ребер так, чтобы она всегда была отсортирована по возрастанию *x*. Далее для каждой строки рисуются все отрезки, соединяющие попарные вершины ребер из AET. При этом проверяется, не попала ли нижняя точка какого-либо ребра на сканирующую прямую. Если попала, то ищется ребро, идущее из данной точки вниз. Если это ребро найдено, то оно заменяет собой старое ребро из AET; в противном случае соответствующее ребро из таблицы активных ребер исключается.

```
// File fillpoly.cpp
#include    <graphics.h>
#include    <mem.h>
#include    "fixmath.h"
#include    «point.h»

struct   AETEntry
{
    int      from;        // from vertex
    int      to;          // to vertex
    Fixed    x;
    Fixed    dx;
    int      dir;
};
static   AETEntry aet [20];  // active edge table
static   int   aetCount;     // # of items in AET
static   int   cr [20];      // list of critical points
static   int   crCount;      // # of critical points

static int     findEdge ( int& j, int dir, int n, Point p [] )
{
    for ( ; ; )
    {
        int  j1 = j + dir;

        if ( j1 < 0 )
            j1 = n - 1;
        else
        if ( j1 >= n )
            j1 = 0;

        if ( p [j1].y < p [j].y ) // edge j,j1 is
            return -1;          // going upwards
        else
        if ( p [j1].y == p [j].y )
            j = j1;
        else
            return j1;
    }
}
void     addEdge ( int j, int j1, int dir, Point p [] )
```

```
{
    AETEntry tmp;

    tmp.from = j;
    tmp.to   = j1;
    tmp.x    = int2Fixed (p [j].x);
    tmp.dx   = int2Fixed (p [j1].x-p [j].x)/(p [j1].y-p [j].y);
    tmp.dir  = dir;

    for ( int i = 0; i < aetCount; i++ ) // use x + dr for edges going
                                         // from the same point
        if ( tmp.x + tmp.dx < aet [i].x + aet [i].dx )
            break;

                        // insert tmp at position i
    memmove ( &aet [i+1], &aet [i], (aetCount - i) *          sizeof ( AETEntry ) );
    aet [i] = tmp;
    aetCount++;
}
void    buildCR ( int n, Point p [] )
{
    int  candidate = 0;

    crCount = 0;
                    // find all critical points
    for ( int i = 0, j = 1; i < n; i++, j++ )
    {
        if ( j >= n )       // check for overflow
            j = 0;

        if ( p [i].y > p [j].y )
            candidate = 1;
        else
        if ( p [i].y < p [j].y && candidate )
        {
            candidate = 0;
            cr [crCount++] = i;
        }
    }
    if ( candidate && p [0].y < p [1].y )
        cr [crCount++] = 0;
                    // now sort critical points
    for ( i = 0; i < crCount; i++ )
        for ( int j = i + 1; j < crCount; j++ )
            if ( p [cr [i]].y > p [cr [j]].y )
            {       // swap cr [i] and cr [j]
                int  tmp = cr [i];

                cr [i] = cr [j];
                cr [j] = tmp;
            }

}
void    fillPoly ( int n, Point p [] )
{
    int  yMin = p [0].y;
```

```
int  yMax = p [0].y;
int  k    = 0;

for ( int i = 1; i < n; i++ )
    if ( p [i].y < yMin )
        yMin = p [i].y;
    else
    if ( p [i].y > yMax )
        yMax = p [i].y;

buildCR ( n, p );

aetCount = 0;

for ( int s = yMin; s <= yMax; s++ )
{
    for ( ; k < crCount && s == p [cr [k]].y; k++ )
    {
        int  j  = cr [k];
        int  j1 = findEdge ( j, -1, n, p );

        addEdge ( j, j1, -1, p );

        j  = cr [k];
        j1 = findEdge ( j, 1, n, p );

        addEdge ( j, j1, 1, p );
    }

    for ( i = 0; i < aetCount; i += 2 )
        line ( fixed2Int ( aet [i].x ), s, fixed2Int ( aet [i+1].x ), s );

    for ( i = 0; i < aetCount; i++ )
    {
        aet [i].x += aet [i].dx;

        if ( p [aet [i].to].y == s )
        {
            int  j  = aet [i].to;
            int  j1 = findEdge ( j, aet [i].dir, n, p );

            if ( j1 >= 0 )      // adjust entry
            {
                aet [i].from = j;
                aet [i].to   = j1;
                aet [i].x    = int2Fixed ( p [j].x );
                aet [i].dx   = int2Fixed ( p [j1].x - p [j].x ) / ( p [j1].y - p [j].y );
            }
            else
            {
                aetCount--;
                memmove ( &aet [i], &aet [i+1], (aetCount - i) * sizeof ( AETEntry ) );
                i--;        // to compensate for i++
            }
        }
    }
}
```

Упражнения

Напишите программу, реализующую растровую развертку эллипса.

Напишите программу построения дуги эллипса.

Напишите программу построения прямоугольника с закругленным углами радиуса *r*.

Напишите программу, использующую алгоритм средней точки для построения закрашенных окружностей и углов.

Модифицируйте алгоритм Брезенхейма для построения линий заданной толщины с заданным шаблоном.

Добавьте в ранее введенный класс Surface поддержку линий различной толщины, поддержку шаблонов линий, заполнение многоугольников, окружностей, дуг/секторов эллипсов, чтение/запись прямоугольных фрагментов изображения, поддержку шаблонов.

Глава 7
ПРЕОБРАЗОВАНИЯ НА ПЛОСКОСТИ

Вывод изображения на экран дисплея и разнообразные действия с ним, в том числе и визуальный анализ, требуют от пользователя известной геометрической грамотности. Геометрические понятия, формулы и факты, относящиеся прежде всего к плоскому и трехмерному случаям, играют в задачах компьютерной графики особую роль. Геометрические соображения, подходы и идеи в соединении с постоянно расширяющимися возможностями вычислительной техники являются неиссякаемым источником существенных продвижений на пути развития компьютерной графики, ее эффективного использования в научных и иных исследованиях. Порой даже самые простые геометрические методики обеспечивают заметное продвижение на отдельных этапах решения большой графической задачи. С простых геометрических рассмотрений мы и начнем наш рассказ.

Заметим прежде всего, что особенности использования геометрических понятий, формул и фактов, как простых и хорошо известных, так и новых, более сложных, требуют особого взгляда на них и иного осмысления.

7.1. Аффинные преобразования на плоскости

В компьютерной графике все, что относится к двумерному случаю, принято обозначать символом (2D) (2-dimension).

Допустим, что на плоскости введена прямолинейная координатная система. Тогда каждой точке М ставится в соответствие упорядоченная пара чисел (x, y) ее координат (рис. 7.1). Вводя на плоскости еще одну прямолинейную систему координат, мы ставим в соответствие той же точке M другую пару чисел- (x^*, y^*).

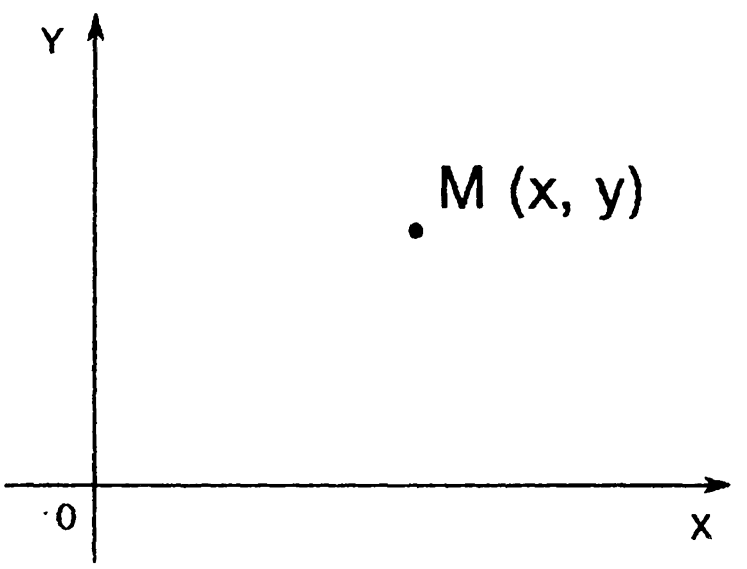


Рис. 7.1

Переход от одной прямолинейной координатной системы на плоскости к другой описывается следующими соотношениями:

$$\begin{aligned} x^* &= \alpha x + \beta y + \lambda, \\ y^* &= \gamma x + \delta y + \mu, \end{aligned} \qquad (7.1)$$

где $\alpha, \beta, \gamma, \lambda, \mu$ - произвольные числа, связанные неравенством

$$\begin{vmatrix} \alpha & \beta \\ \gamma & \delta \end{vmatrix} \neq 0.$$

Замечание. Формулы (7.1) можно рассматривать двояко: либо сохраняется точка и изменяется координатная система (рис. 7.2) - в этом случае произвольная точка М остается той же, изменяются лишь ее координаты

$$(x, y) \to (x^*, y^*),$$

либо изменяется точка и сохраняется координатная система (рис. 7.3) – в этом случае формулы (7.1) задают отображение, переводящее произвольную точку M(x, y) в точку M(x*, y*), координаты которой определены в той же координатной системе.*

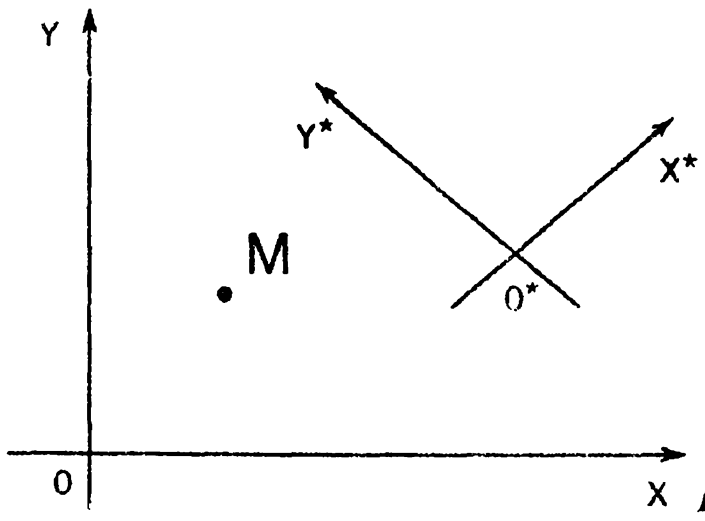


Рис. 7.2

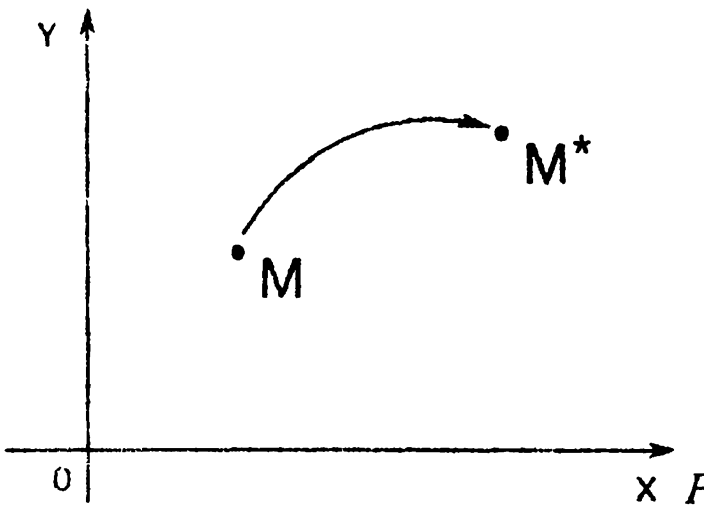


Рис. 7.3

В дальнейшем мы будем рассматривать формулы (7.1) как правила, согласно которым в заданной системе прямолинейных координат преобразуются точки плоскости.

В аффинных преобразованиях плоскости особую роль играют несколько важных частных случаев, имеющих хорошо прослеживаемые геометрические характеристики. При исследовании геометрического смысла числовых коэффициентов в формулах (7.1) для этих случаев нам удобно считать, что заданная система координат является прямоугольной декартовой.

А. Поворот вокруг начальной точки на угол φ (рис. 7.4) описывается формулами

$$x^* = x\cos\varphi - y\sin\varphi,$$

$$y^* = x\sin\varphi + y\cos\varphi.$$

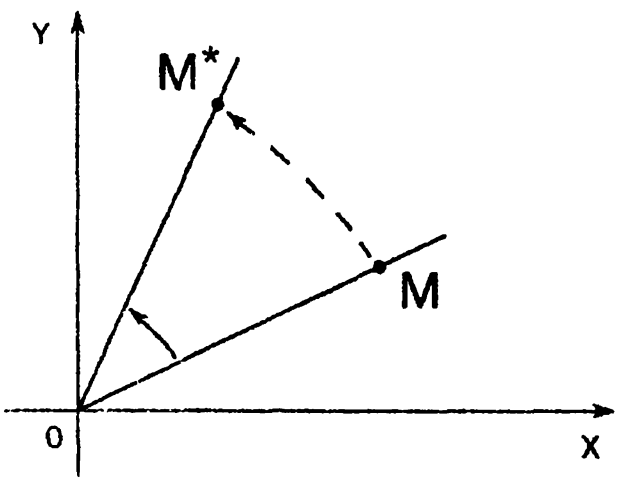


Рис. 7.4

Б. Растяжение (сжатие) вдоль координатных осей можно задать так:

$$x^* = \alpha x,$$

$$y^* = \delta y,$$

$\alpha > 0$, $\delta > 0$

Растяжение (сжатие) вдоль оси абсцисс обеспечивается при условии, что $\alpha > 1$ ($\alpha < 1$). На рис. 7.5 $\alpha = \delta > 1$.

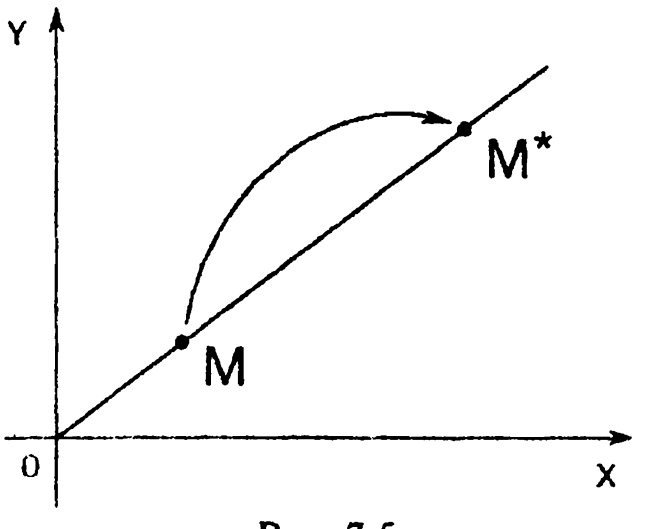


Рис. 7.5

В. Отражение (относительно оси абсцисс) (рис. 7.6) задается при помощи формул

$$x^* = x,$$

$$y^* = -y.$$

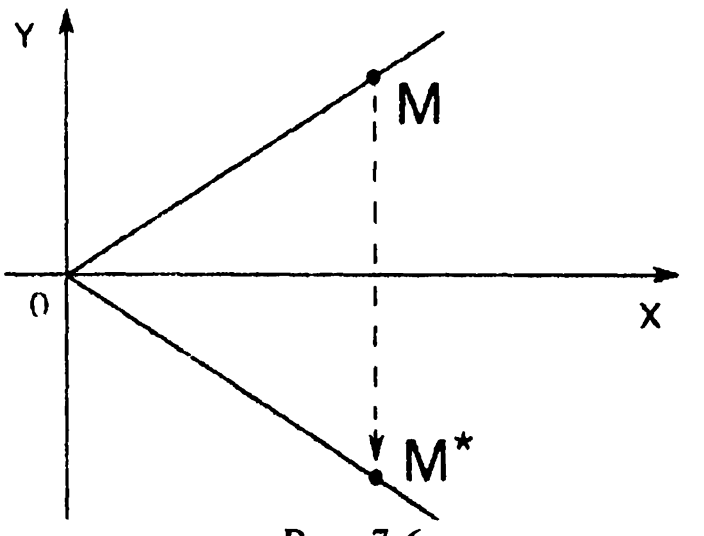


Рис. 7.6

Г. На рис. 7.7 вектор переноса MM^* имеет координаты λ и μ. Перенос обеспечивают соотношения

$$x^* = x + \lambda,$$

$$y^* = y + \mu.$$

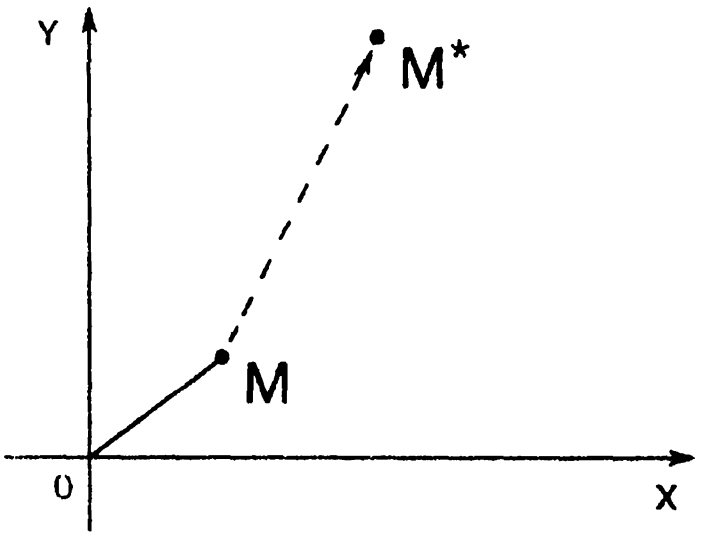


Рис. 7.7

Выбор этих четырех частных случаев определяется двумя обстоятельствами.

- каждое из приведенных выше преобразований имеет простой и наглядный геометрический смысл (геометрическим смыслом наделены и постоянные числа, входящие в приведенные формулы);
- как доказывается в курсе аналитической геометрии, любое преобразование вида (7.1) всегда можно представить как последовательное исполнение (суперпозицию) простейших преобразований вида А, Б, В и Г (или части этих преобразований).

Таким образом, справедливо следующее важное свойство аффинных преобразований плоскости: любое отображение вида (7.1) можно описать при помощи отображений, задаваемых формулами А, Б, В и Г.

Для эффективного использования этих известных формул в задачах компьютерной графики более удобной является их матричная запись. Матрицы, соответствующие случаям А, Б и В, строятся легко и имеют соответственно следующий вид:

$$\begin{pmatrix} \cos\varphi & \sin\varphi \\ -\sin\varphi & \cos\varphi \end{pmatrix}, \begin{pmatrix} \alpha & 0 \\ 0 & \delta \end{pmatrix}, \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}.$$

Ниже приводятся файлы, реализующие классы Vector2D и Matrix2D для работы с двух мерной графикой.

```
// File vector2d.h
#ifndef     __VECTOR2D__
#define     __VECTOR2D__

#include    <math.h>
```

```
enum            // values of Vector2D    classify
{
        LEFT, RIGHT, BEHIND, BEYOND, ORIGIN, DESTINATION, BETW
};
class Vector2D
{
public:
        float       x, y;
        Vector2D () {}
        Vector2D ( float px, float py )
        {
                x = px;
                y = py;
        }
        Vector2D ( const Vector2D& v )
        {
                x = v.x;
                y = v.y;
        }
        Vector2D& operator = ( const Vector2D& v )
        {
                x = v.x;
                y = v.y;
                return *this;
        }
Vector2D operator + () const
{
                return *this;
}
Vector2D operator - () const
{
                return Vector2D ( -x, -y );
}
Vector2D& operator += ( const Vector2D& v )
{
                x += v.x;
                y += v.y;
                return *this;
}
Vector2D& operator -= ( const Vector2D& v )
{
                x -= v.x;
                y -= v.y;
                return *this;
}
Vector2D& operator *= ( const Vector2D& v )
{
```

```
        x *= v.x;
        y *= v.y;
        return *this;
}
Vector2D& operator *= ( float f )
{
        x *= f;
        y *= f;
        return *this;
}
Vector2D& operator /= ( const Vector2D& v )
{
        x /= v.x;
        y /= v.y;
        return *this;
}
Vector2D& operator /= ( float f )
{
        x /= f;
        y /= f;
        return *this;
}
float& operator [] ( int index )
{
        return * ( index + &x );
}
int     operator == ( const Vector2D& v ) const
{
        return x == v.x && y == v.y;
}
int     operator != ( const Vector2D& v ) const
{
        return x != v.x || y != v.y;
}
int     operator < ( const Vector2D& v ) const
{
        return (x < v.x) || ( (x == v.x) && (y < v.y));
}
int     operator > ( const Vector2D& v ) const
{
        return (x > v.x) || ( (x == v.x) && (y > v.y));
}
float   length () const
{
        return (float) sqrt ( x * x + y * y );
}
```

```
        float       polarAngle () const
        {
                    return (float) atan2 ( y, x );
        }
        int         classify ( const Vector2D&, const Vector2D& ) const;

        friend Vector2D operator + (const Vector2D&,const Vector2D&);
        friend Vector2D operator - (const Vector2D&,const Vector2D&);
        friend Vector2D operator * (const Vector2D&,const Vector2D&);
        friend Vector2D operator * (float,                  const Vector2D&);
        friend Vector2D operator * ( const Vector2D&, float );
        friend Vector2D operator / (const Vector2D&,float );
        friend Vector2D operator / (const Vector2D&, const Vector2D&);
        friend float        operator & (const Vector2D&,    const Vector2D&);
};
inline Vector2D operator + ( const Vector2D& u, const Vector2D& v )
{
        return Vector2D ( u.x + v.x, u.y + v.y );
}
inline Vector2D operator - ( const Vector2D& u, const Vector2D& v )
{
        return Vector2D ( u.x - v.x, u.y - v.y );
}
inline Vector2D operator * ( const Vector2D& u, const Vector2D& v )
{
        return Vector2D ( u.x*v.x, u.y*v.y );
}
inline Vector2D operator * ( const Vector2D& v, float a )
{
        return Vector2D ( v.x*a, v.y*a );
}
inline Vector2D operator * ( float a, const Vector2D& v )
{
        return Vector2D ( v.x*a, v.y*a );
}
inline Vector2D operator / ( const Vector2D& u, const Vector2D& v )
{
        return Vector2D ( u.x/v.x, u.y/v.y );
}
inline Vector2D operator / ( const Vector2D& v, float a )
{
        return Vector2D ( v.x/a, v.y/a );
}
inline float operator & ( const Vector2D& u, const Vector2D& v )
{
        return u.x*v.x + u.y*v.y;
}
#endif
```

```
// File vector2d.cpp
#include "vector2d.h"

//////////////////// member functions ////////////////

int Vector2D    classify (const Vector2D& p,const Vector2D& q) const
{
        Vector2D a = q - p;
        Vector2D b = *this - p;
        float        s = a.x * b.y - a.y * b.x;

        if ( s > 0.0 )
                return LEFT;

        if ( s < 0.0 )
                return RIGHT;

        if ( a.x * b.x < 0.0 || a.y * b.y < 0.0 )
                return BEHIND;

        if ( a.length () < b.length () )
                return BEYOND;

        if ( p == *this )
                return ORIGIN;

        if ( q == *this )
                return DESTINATION;

        return BETWEEN;
}
```

```
// File matrix2d.h
#ifndef         __MATRIX2D__
#define         __MATRIX2D__

#include        "vector2d.h"

class Matrix2D
{
public:
        float       a [2][2];

        Matrix2D () {}
        Matrix2D ( float );
        Matrix2D ( const Matrix2D& );

        Matrix2D& operator =  ( const Matrix2D& )
        Matrix2D& operator =  ( float );
        Matrix2D& operator += ( const Matrix2D& );
        Matrix2D& operator -= ( const Matrix2D& );
        Matrix2D& operator *= ( const Matrix2D& );
        Matrix2D& operator *= ( float );
        Matrix2D& operator /= ( float );

        float   * operator [] ( int i )
        {
                return &x [i][0];
        }
```

```
        void invert ();
        void transpose ();

        static Matrix2D scale   ( const Vector2D& );
        static Matrix2D rotate  ( float );
        static Matrix2D mirrorX ();
        static Matrix2D mirrorY ();

        friend Matrix2D operator + (const Matrix2D&,const Matrix2D&);
        friend Matrix2D operator - (const Matrix2D&,const Matrix2D&);
        friend Matrix2D operator * (const Matrix2D&,const Matrix2D&);
        friend Matrix2D operator * (const Matrix2D&,float);
        friend Matrix2D operator * (float,          const Matrix2D&);
        friend Vector2D operator * (const Matrix2D&,const Vector2D&);
};

inline Vector2D operator * ( const Matrix2D& m,
                                                const Vector2D& v)
{
        return Vector2D ( m.a [0][0]*v.x + m.a [0][1]*v.y,
                                m.a [1][0]*v.x + m.a [1][1]*v.y );
}

#endif
```

```
// File matrix2d.cpp
#include        "matrix2d.h"

Matrix2D    Matrix2D ( float v )
{
        a [0][1] = a [1][0] = 0.0;
        a [0][0] = a [1][1] = v;
}

Matrix2D    Matrix2D ( const Matrix2D& m )
{
        a [0][0] = m.a [0][0];
        a [0][1] = m.a [0][1];
        a [1][0] = m.a [1][0];
        a [1][1] = m.a [1][1];
}

Matrix2D& Matrix2D    operator = ( const Matrix2D& m )
{
        a [0][0] = m.a [0][0];
        a [0][1] = m.a [0][1];
        a [1][0] = m.a [1][0];
        a [1][1] = m.a [1][1];

        return *this;
}

Matrix2D& Matrix2D    operator = ( float x )
{
        a [0][1] = a [1][0] = 0.0;
        a [0][0] = a [1][1] = x;
```

```
        return *this;
}
Matrix2D& Matrix2D    operator += ( const Matrix2D& m )
{
        a [0][0] += m.a [0][0];
        a [0][1] += m.a [0][1];
        a [1][0] += m.a [1][0];
        a [1][1] += m.a [1][1];

        return *this;
}
Matrix2D& Matrix2D :: operator -= ( const Matrix2D& m )
{
        a [0][0] -= m.a [0][0];
        a [0][1] -= m.a [0][1];
        a [1][0] -= m.a [1][0];
        a [1][1] -= m.a [1][1];

        return *this;
}
Matrix2D& Matrix2D :: operator *= ( const Matrix2D& m )
{
        Matrix2D c ( *this );

        a [0][0] = c.a [0][0]*m.a [0][0] + c.a [0][1]*m.a [1][0];
        a [0][1] = c.a [0][0]*m.a [0][1] + c.a [0][1]*m.a [1][1];
        a [1][0] = c.a [1][0]*m.a [0][0] + c.a [1][1]*m.a [1][0];
        a [1][1] = c.a [1][0]*m.a [0][1] + c.a [1][1]*m.a [1][1];

        return *this;
}
Matrix2D& Matrix2D :: operator *= ( float f )
{
        a [0][0] *= f;
        a [0][1] *= f;
        a [1][0] *= f;
        a [1][1] *= f;
        return *this;
}
Matrix2D& Matrix2D :: operator /= ( float f )
{
        a [0][0] /= f;
        a [0][1] /= f;
        a [1][0] /= f;
        a [1][1] /= f;

        return *this;
};
void Matrix2D :: invert ()
{
        float det = a [0][0]*a [1][1] - a [0][1]*a [1][0];
        Matrix2D m;
```

```
        m.a [0][0] =  a [1][1] / det;
        m.a [0][1] = -a [0][1] / det;
        m.a [1][0] = -a [1][0] / det;
        m.a [1][1] =  a [0][0] / det;

        *this = m;
}
void Matrix2D   transpose ()
{
        Matrix2D m;

        m.a [0][0] = a [0][0];
        m.a [0][1] = a [1][0];
        m.a [1][0] = a [0][1];
        m.a [1][1] = a [1][1];

        *this = m;
}
Matrix2D Matrix2D :: scale ( const Vector2D& v )
{
        Matrix2D m;

        m.a [0][0] = v.x;
        m.a [1][1] = v.y;
        m.a [0][1] =
        m.a [1][0] = 0.0;

        return m;
}
Matrix2D Matrix2D :: rotate ( float angle )
{
        float   cosine,sine;
        Matrix2D m ( 1.0 );

        cosine    = cos ( angle );
        sine      = sin ( angle );
        m.a [0][0] = cosine;
        m.a [0][1] = sine;
        m.a [1][0] = -sine;
        m.a [1][1] = cosine;

        return m;
}
Matrix2D Matrix2D :: mirrorX ()
{
        Matrix2D m ( 1.0 );

        m.a [0][0] = -1.0;

        return m;
}
Matrix2D Matrix2D   mirrorY ()
{
        Matrix2D m ( 1.0 );

        m.a [1][1] = -1.0;

        return m;
```

```
}
Matrix2D operator + ( const Matrix2D& a, const Matrix2D& b )
{
        Matrix2D c;

        c.x[0][0]=a.x[0][0]+b.x[0][0];
        c.x[0][1]=a.x[0][1]+b.x[0][1];
        c.x[1][0]=a.x[1][0]+b.x[1][0];
        c.x[1][1]=a.x[1][1]+b.x[1][1];

        return c;
}
Matrix2D operator - ( const Matrix2D& a, const Matrix2D& b )
{
        Matrix2D c;

        c.x[0][0]=a.x[0][0]-b.x[0][0];
        c.x[0][1]=a.x[0][1]-b.x[0][1];
        c.x[1][0]=a.x[1][0]-b.x[1][0];
        c.x[1][1]=a.x[1][1]-b.x[1][1];

        return c;
}
Matrix2D operator * ( const Matrix2D& a, const Matrix2D& b )
{
        Matrix2D c;

        c.x[0][0]=a.x[0][0]*b.x[0][0]+a.x[0][1]*b.x[1][0];
        c.x[0][1]=a.x[0][0]*b.x[0][1]+a.x[0][1]*b.x[1][1];
        c.x[1][0]=a.x[1][0]*b.x[0][0]+a.x[1][1]*b.x[1][0];
        c.x[1][1]=a.x[1][0]*b.x[0][1]+a.x[1][1]*b.x[1][1];

        return c;
}
Matrix2D operator * ( const Matrix2D& a, float b )
{
        Matrix2D c;

        c.x[0][0]=a.x[0][0]*b;
        c.x[0][1]=a.x[0][1]*b;
        c.x[1][0]=a.x[1][0]*b;
        c.x[1][1]=a.x[1][1]*b;

        return c;
}
Matrix2D operator * ( float b, const Matrix2D& a )
{
        Matrix2D c;

        c.x[0][0]=a.x[0][0]*b;
        c.x[0][1]=a.x[0][1]*b;
        c.x[1][0]=a.x[1][0]*b;
        c.x[1][1]=a.x[1][1]*b;

        return c;
}
```

Эти классы представляют собой реализацию двумерных векторов (класс Vector2D) и матриц 2x2 (класс Matrix2D). Для этих классов переопределяются основные знаки операций:

- `-` - унарный минус и поэлементное вычитание;
- `+` - поэлементное сложение;
- `*` - умножение на число;
- `*` - перемножение матриц;
- `*` - поэлементное умножение векторов;
- `*` - умножение матрицы на вектор
- `/` - деление на число;
- `/` - поэлементное деление векторов;
- `&` - скалярное произведение векторов;
- `[]` - компонента вектора.

Стандартные приоритеты операций сохраняются.

Кроме этих операций определяются также некоторые простейшие функции для работы с векторами: length () - длина вектора, polarAngle () - полярный угол для вектора, classify () - классификация вектора относительно двух других векторов (точек) (о последней функции подробнее в гл. 8).

С использованием этих классов можно в естественной и удобной форме записывать сложные векторные и матричные выражения.

7.2. Однородные координаты точки

Для решения рассматриваемых далее задач весьма желательно охватить матричным подходом все 4 простейших преобразования (в том числе и перенос), а, значит, и общее аффинное преобразование. Этого можно достичь, например, так: перейти к описанию произвольной точки плоскости не упорядоченной парой чисел, как это было сделано выше, а упорядоченной тройкой чисел.

Пусть M - произвольная точка плоскости с координатами x и y, вычисленными относительно заданной прямолинейной координатной системы.

Однородными координатами этой точки называется любая тройка одновременно неравных нулю чисел x_1, x_2, x_3, связанных с заданными числами x и y следующими соотношениями:

$$\frac{x_1}{x_3} = x, \quad \frac{x_2}{x_3} = y.$$

При решении задач компьютерной графики однородные координаты обычно вводятся так: произвольной точке $M(x, y)$ плоскости ставится в соответствие точка $M(x, y, 1)$ в пространстве (рис. 7.8).

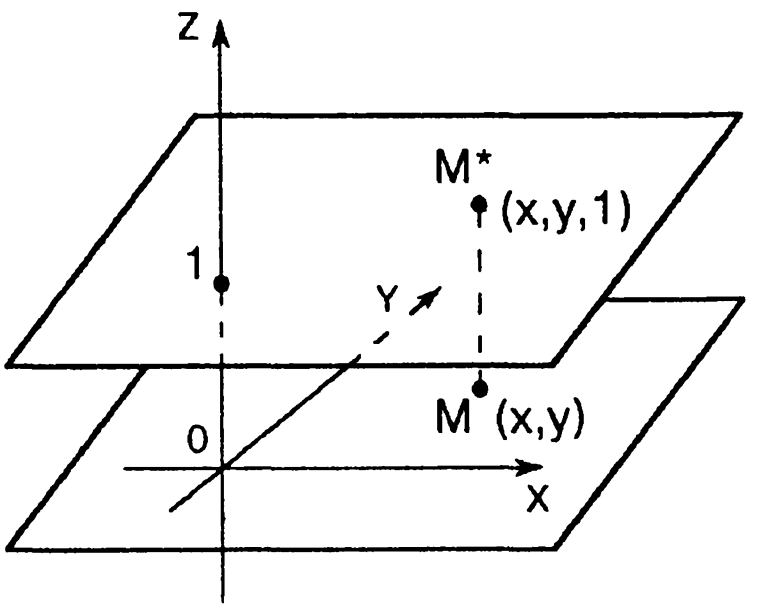


Рис. 7.8

Заметим, что произвольная точка на прямой, соединяющей начало координат, точку $O(0, 0, 0)$, с точкой $M(x, y, 1)$, может быть задана тройкой чисел вида (hx, hy, h).

Будем считать, что $h \neq 0$.

Вектор с координатами (hx, hy, h) является направляющим вектором прямой, соединяющей точки $O(0, 0,0)$ и $M(x, y, 1)$. Эта прямая пересекает плоскость $Z = 1$ в точке $(x, y, 1)$, которая однозначно определяет точку (x, y) координатной плоскости xy.

Тем самым между произвольной точкой с координатами (x, y) и множеством троек чисел вида (hx, hy, h), $h \neq 0$, устанавливается (взаимно однозначное) соответствие, позволяющее считать числа hx, hy, h новыми координатами этой точки.

Замечание. *Широко используемые в проективной геометрии однородные координаты позволяют эффективно описывать так называемые несобственные элементы (по существу, те, которыми проективная плоскость отличается от привычной нам евклидовой плоскости).*

В проективной геометрии для однородных координат принято следующее обозначение:

$x : y : 1$, или, более общо, x_1, x_2, x_3

(напомним, что здесь непременно требуется, чтобы числа x_1, x_2, x_3 одновременно в нуль не обращались).

Применение однородных координат оказывается удобным уже при решении простейших задач.

Рассмотрим, например, вопросы, связанные с изменением масштаба. Если устройство отображения работает только с целыми числами (или если необходимо работать только с целыми числами), то для произвольного значения h (например, $h = 1$) точку с однородными координатами (0.5 0.1 2.5)

представить нельзя. Однако при разумном выборе h можно добиться того, чтобы координаты этой точки были целыми числами. В частности, при $h = 10$ для рассматриваемого примера имеем (5 1 25).

Возьмем другой случай. Чтобы результаты преобразования не приводили к арифметическому переполнению, для точки с координатами (80000 40000 1000) можно взять, например, $h = 0{,}001$. В результате получим (80 40 1).

Приведенные примеры показывают полезность использования однородных координат при проведении расчетов. Однако основной целью введения однородных координат в компьютерной графике является их несомненное удобство в применении к геометрическим преобразованиям.

При помощи троек однородных координат и матриц третьего порядка можно описать любое аффинное преобразование плоскости.

В самом деле, считая $h = 1$, сравним две записи: помеченную символом * и нижеследующую матричную:

$$(x\,{*}\,y\,{*}1) = (xy1)\begin{bmatrix} \alpha & \gamma & 0 \\ \beta & \delta & 0 \\ \lambda & \mu & 1 \end{bmatrix}$$

Нетрудно заметить, что после перемножения выражений, стоящих в правой части последнего соотношения, мы получим обе формулы (7.1) и верное числовое равенство 1 = 1.

Тем самым сравниваемые записи можно считать равносильными.

Замечание. Иногда в литературе используется другая запись - запись по столбцам.

$$\begin{bmatrix} x^* \\ y^* \\ 1 \end{bmatrix} = \begin{bmatrix} \alpha & \beta & \lambda \\ \gamma & \delta & \mu \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}.$$

Такая запись эквивалентна приведенной выше записи по строкам (и получается из нее транспонированием).

Элементы произвольной матрицы аффинного преобразования не несут в себе явно выраженного геометрического смысла. Поэтому чтобы реализовать то или иное отображение, т. е. найти элементы соответствующей матрицы по заданному геометрическому описанию, необходимы специальные приемы. Обычно построение этой матрицы в соответствии со сложностью рассматриваемой задачи и с описанными выше частными случаями разбивают на несколько этапов.

На каждом этапе ищется матрица, соответствующая тому или иному из выделенных выше случаев А, Б, В или Г, обладающих хорошо выраженными геометрическими свойствами.

Выпишем соответствующие матрицы третьего порядка.

А. Матрица вращения (rotation)

$$[R] = \begin{bmatrix} \cos\varphi & \sin\varphi & 0 \\ -\sin\varphi & \cos\varphi & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Б. Матрица растяжения (сжатия) *(dilatation)*

$$[D] = \begin{bmatrix} \alpha & 0 & 0 \\ 0 & \delta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

В. Матрица отражения (reflection)

$$[M] = \begin{bmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Г. Матрица переноса (translation)

$$[T] = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ \lambda & \mu & 1 \end{bmatrix}$$

Рассмотрим примеры аффинных преобразований плоскости.

Пример 1. Построить матрицу поворота вокруг точки A(a, b) на угол φ (рис. 7.9).

1-й шаг. Перенос на вектор *A(-a,-b)* для совмещения центра поворота с началом координат;

$[T_{-A}] = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -a & -b & 1 \end{bmatrix}$ - матрица соответствующего преобразования.

2-й шаг. Поворот на угол φ;

$$[R_\varphi]=\begin{bmatrix}\cos\varphi & \sin\varphi & 0\\ -\sin\varphi & \cos\varphi & 0\\ 0 & 0 & 1\end{bmatrix}$$ -

матрица соответствующего преобразования.

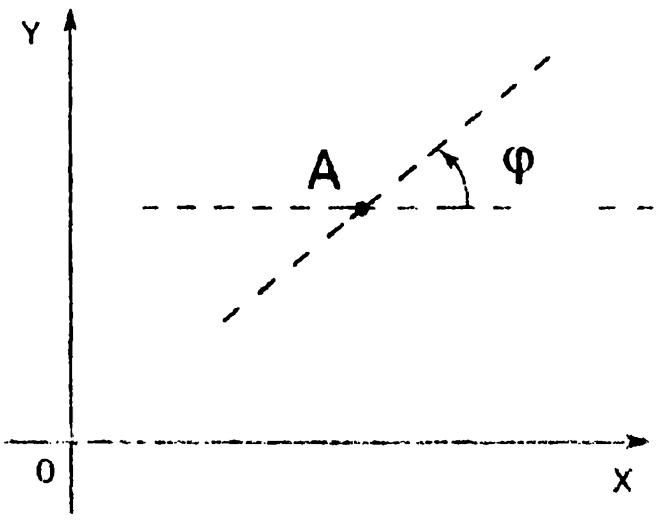


Рис. 7.9

3-й шаг. Перенос на вектор $A(a, b)$ для возвращения центра поворота в прежнее положение;

$$[T_A]=\begin{bmatrix}1 & 0 & 0\\ 0 & 1 & 0\\ a & b & 1\end{bmatrix}$$ - матрица соответствующего преобразования.

Перемножим матрицы в том же порядке, как они выписаны:

$[T_{-A}][R_\varphi][T_A]$.

В результате получим, что искомое преобразование (в матричной записи) будет выглядеть следующим образом:

$$(x^* y^* 1)=(x\,y\,1)\times$$

$$\times\begin{bmatrix}\cos\varphi & \sin\varphi & 0\\ -\sin\varphi & \cos\varphi & 0\\ -a\cos\varphi+b\sin\varphi+a & -a\sin\varphi-b\cos\varphi+b & 1\end{bmatrix}.$$

Элементы полученной матрицы (особенно в последней строке) не так легко запомнить. В то же время каждая из трех перемножаемых матриц по геометрическому описанию соответствующего отображения легко строится.

Пример 2. Построить матрицу растяжения с коэффициентами растяжения α вдоль оси абсцисс и β вдоль оси ординат и с центром в точке $A(a,b)$

1-й шаг. Перенос на вектор - $A(-a,-b)$ для совмещения центра растяжения с началом координат;

$$[T_{-A}]=\begin{bmatrix}1 & 0 & 0\\ 0 & 1 & 0\\ -a & -b & 1\end{bmatrix}$$ - матрица соответствующего преобразования.

2-й шаг. Растяжение вдоль координатных осей с коэффициентами α и δ соответственно; матрица преобразования имеет вид

$$[D]=\begin{bmatrix}\alpha & 0 & 0\\ 0 & \delta & 0\\ 0 & 0 & 1\end{bmatrix}.$$

3-й шаг. Перенос на вектор - $A(-a,-b)$ для возвращения центра растяжения в прежнее положение; матрица соответствующего преобразования -

$$[T_A]=\begin{bmatrix}1 & 0 & 0\\ 0 & 1 & 0\\ a & b & 1\end{bmatrix}$$

Перемножив матрицы в том же порядке $[T_{-A}][D][T_A]$, получим окончательно:

$$(x^* y^* 1)=(xy1)\begin{bmatrix}\alpha & 0 & 0\\ 0 & \delta & 0\\ (1-\alpha)a & (1-\delta)b & 1\end{bmatrix}$$

Замечание. Рассуждая подобным образом, т. е. разбивая предложенное преобразование на этапы, поддерживаемые матрицами
$[R],[D],[M],[T],$
можно построить матрицу любого аффинного преобразования по его геометрическому описанию.

Глава 8
ОСНОВНЫЕ АЛГОРИТМЫ ВЫЧИСЛИТЕЛЬНОЙ ГЕОМЕТРИИ

8.1. Отсечение отрезка. Алгоритм Сазерленда - Кохена

Необходимость отсечения выводимого изображения по границам некоторой ласти встречается довольно часто.

В простейших ситуациях в качестве такой области, как правило, выступает прямоугольник (рис. 8.1).

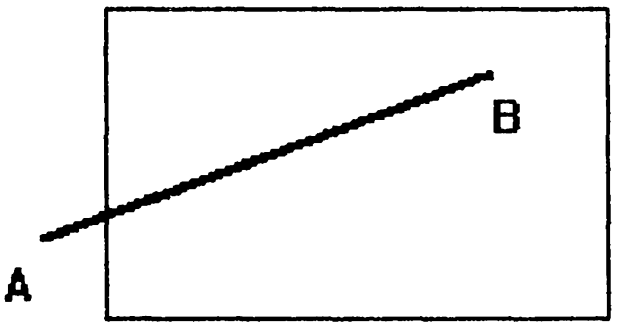


Рис. 8.1

Ниже рассматривается достаточно простой и эффективный алгоритм отсечения отрезков по границе произвольного прямоугольника.

Четырьмя прямыми вся плоскость разбивается на 9 областей (рис. 8.2). По отношению к прямоугольнику точки в каждой из этих областей расположены одинаково.

0011	0010	0110
0001	0000	0100
1001	1000	1100

Рис. 8.2

Определив, в какие области попали концы рассматриваемого отрезка, легко понять, где именно необходимо произвести отсечение. Для этого каждой области ставится в соответствие4-битовый код, где установленный

бит 0 означает, что точка лежит левее прямоугольника,
бит 1 означает, что точка лежит выше прямоугольника,
бит 2 означает, что точка лежит правее прямоугольника,
бит 3 означает, что точка лежит ниже прямоугольника.

Приведенная ниже программа реализует алгоритм Сазерленда - Кохена отс ния отрезка по прямоугольной области.

```
// File clip.cpp
inline   void  swap ( int& a, int& b )
{
        int  c;

        c = a;
        a = b;
        b = c;
    }
```

```
int  outCode ( int x, int y, int X1, int Y1, int X2, int Y2 )
{
     int code = 0;

     if ( x < X1 )
          code |= 0x01;

     if ( y < Y1 )
          code |= 0x02;

     if ( x > X2 )
          code |= 0x04,

     if ( y > Y2 )
          code |= 0x08;

     return code;
}

void  clipLine ( int x1, int y1, int x2, int y2,
      int X1, int Y1, int X2, int Y2 )
{
       int  code1   = outCode ( x1, y1, X1, Y1, X2, Y2 );
       int  code2   = outCode ( x2, y2, X1, Y1, X2, Y2 );
       int  inside  = ( code1 | code2 ) == 0;
       int  outside = ( code1 & code2 ) != 0;

while ( !outside && !inside )
{
   if ( code1 == 0 )
  {
    swap ( x1, x2 );
    swap ( y1, y2 );
    swap ( code1, code2 );
  }
  if ( code1 & 0x01 )    // clip left
  {
    y1 += (long)(y2-y1)*(X1-x1)/(x2-x1);
         x1  = X1;
  }
  else
  if ( code1 & 0x02 )    // clip above
  {
        x1 += (long)(x2-x1)*(Y1-y1)/(y2-y1);
        y1  = Y1;
  }
  else
  if ( code1 & 0x04 )    // clip right
  {
        y1 += (long)(y2-y1)*(X2-x1)/(x2-x1);
        x1  = X2;
  }
  else
  if ( code1 & 0x08 )    // clip below
```

```
    {
        x1 += (long)(x2-x1)*(Y2-y1)/(y2-y1);
        y1  = Y2;
    }
      code1   = outCode (x1, y1, X1, Y1, X2, Y2);
      code2   = outCode (x2, y2, X1, Y1, X2, Y2);
      inside  = (code1 | code2) == 0;
      outside = (code1 & code2) != 0;
  }
  line ( x1, y1, x2, y2 );
 }
```

8.2. Классификация точки относительно отрезка

Рассмотрим следующую задачу: на плоскости заданы точка и направленный отрезок. Требуется определить положение точки относительно этого отрезка (рис. 8.3).

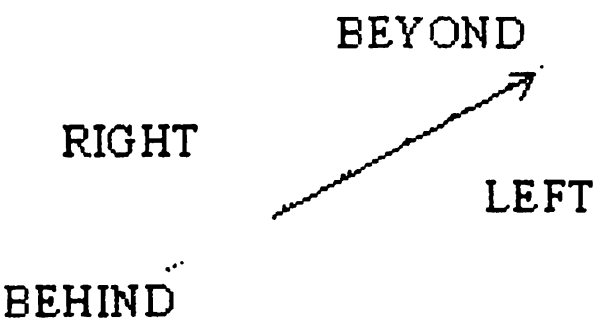


Рис. 8.3

Возможными значениями являются LEFT (слева), RIGHT (справа), BEHIND (позади), BEYOND (впереди), BETWEEN (между), ORIGIN (начало) и DESTINATION (конец).

```
// File Vector2D.cpp
#include "vector2d.h"
//////////////////// member functions ////////////////////
int  Vector2D :: classify ( Vector2D& p, Vector2D& q )
{
    Vector2D a = q - p;
    Vector2D b = *this - p;
    float        s = a.x * b.y - a.y * b.x;

    if ( s > 0.0 )
        return LEFT;

    if ( s < 0.0 )
        return RIGHT;

    if ( a.x * b.x < 0.0 || a.y * b.y < 0.0 )
        return BEHIND;

    if ( a.length () < b.length () )
        return BEYOND;

    if ( p == *this )
        return ORIGIN;

    if ( q == *this )
        return DESTINATION;

    return BETWEEN;
}
```

8.3. Расстояние от точки до прямой

Пусть заданы точка $C(c_x, c_y)$ и прямая AB, где $A(a_x, a_y)$, $B(b_x, b_y)$ и требуется найти расстояние от этой точки до прямой.

Найдем длину отрезка AB:

$$l = \sqrt{(a_x - b_x)^2 + (a_y - b_y)^2}$$

Опустим из точки C перпендикуляр на AB. Точку P пересечения этого перпендикуляра с прямой можно представить параметрически

$P = A + r\,(B - A)$,

где

$$r = \frac{(a_y - c_y)(a_y - b_y) - (a_x - c_x)(b_x - a_x)}{l^2}.$$

Положение точки C на этом перпендикуляре будет задаваться параметром s, $s < 0$ означает, что C находится слева от AB, $s > 0$, что C - справа от AB и $s = 0$ означает, что C лежит на AB.

Для вычисления S воспользуемся следующей формулой:

$$s = \frac{(a_y - c_y)(b_x - a_x) - (a_x - c_x)(b_y - a_y)}{l^2},$$

и тогда искомое расстояние $PC = sl$.

8.4. Нахождение пересечения двух отрезков

Пусть A, B, C и D - точки на плоскости. Тогда направленные отрезки AB и задаются следующими параметрическими уравнениями:

$$P = A + r(B - A), P \in AB,$$

$$Q = C + s(D - C), Q \in CD.$$

Если отрезки AB и CD пересекаются, то

$$A + r(B - A) = C + s(D - C).$$

Перепишем это векторное соотношение в координатном виде:

$$a_x + r(b_x - a_x) = c_x + s(d_x - c_x),$$

$$a_y + r(b_y - a_y) = c_y + s(d_y - c_y)$$

Эта система линейных алгебраических уравнений при

$$(b_x - a_x)(d_y - c_y) \neq (b_y - a_y)(d_x - c_x)$$

имеет единственное решение:

$$r = \frac{(a_y - c_y)(d_x - c_x) - (a_x - c_x)(d_y - c_y)}{(b_x - a_x)(d_y - c_y) - (b_y - a_y)(d_x - c_x)},$$

$$s = \frac{(a_y - c_y)(b_x - a_x) - (a_x - c_x)(b_y - a_y)}{(b_x - a_x)(d_y - c_y) - (b_y - a_y)(d_x - c_x)}.$$

Если оба получившихся значения r и s принадлежат отрезку [0,1], то отрезки AB и CD пересекаются и точка пересечения может быть найдена из параметрических уравнений. В случае, когда оба или одно из полученных значений не принадлежат отрезку [0,1], отрезки AB и CD не пересекаются, но пересекаются соответствующие прямые.

Равенство $(b_x - a_x)(d_y - c_y) = (b_y - a_y)(d_x - c_x)$ означает, что отрезки AB и CD параллельны.

8.5. Проверка принадлежности точки многоугольнику

Введем класс Polygon для представления многоугольников на плоскости. Ниже приводится h-файл, описывающий этот класс.

```
// File Polygon.h
#ifndef __POLYGON__
#define __POLYGON__
#include    <mem.h>
#include    "vector2d.h"

class   Polygon
{
public:
    int         numVertices;        // current # of vertices
    int         maxVertices;        // size of vertices array
    Vector2D * vertices;

    Polygon ()
    {
        numVertices = maxVertices = 0;
        vertices    = NULL;
    }

    Polygon ( const Vector2D * v, int size )
    {
        vertices    = new Vector2D [numVertices = size];
              maxVertices = size];

        memcpy ( vertices, v, size * sizeof ( Vector2D ) );
    }

    Polygon ( const Polygon& p )
    {
        vertices        = new Vector2D [p.maxVertices];
        numVertices  = p.numVertices;
        maxVertices  = p.maxVertices;
```

```
            memcpy ( vertices, p.vertices,
                                numVertices * sizeof ( Vector2D ) );
        }
        ~Polygon ()
        {
            if ( vertices != NULL )
                delete [] vertices;
        }
        int    addVertex ( const Vector2D& v )
        {
            return addVertex ( v, numVertices );
        }
        int         addVertex ( const Vector2D& v, int after );
        int         delVertex ( int index );
        int         isInside  ( const Vector2D& );
        Polygon * split ( int from, int to );
    protected:
        void     resize ( int newMaxVertices );
    };
    #endif
```

Одним из методов класса Polygon является isInside, служащий для проверки принадлежности точки многоугольнику (рис. 8.4).

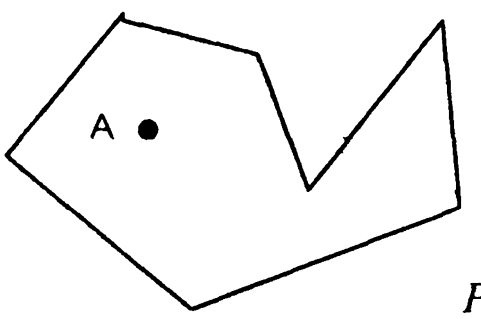


Рис. 8.4

Для решения этой задачи выпустим из точки $A(x, y)$ произвольный луч и найдем количество точек пересечения этого луча с границей многоугольника. Если отбросить случай, когда луч проходит через вершину многоугольника, то решение задачи тривиально - точка лежит внутри, если общее количество точек пересечения нечетно, и снаружи, если четно.

Ясно, что для любого многоугольника всегда можно построить луч, не проходящий ни через одну из вершин. Однако построение такого луча связано с некоторыми трудностями и, кроме того, проверку пересечения границы многоугольника с произвольным лучом провести сложнее, чем с фиксированным, например горизонтальным.

Возьмем луч, выходящий из произвольной точки A, и рассмотрим, к чему может привести прохождение луча через вершину многоугольника. Основные возможные случаи изображены на рис. 8.5.

В случае *а*, когда ребра, выходящие из соответствующей вершины, лежат по одну сторону от луча, четность количества пересечений не меняется.

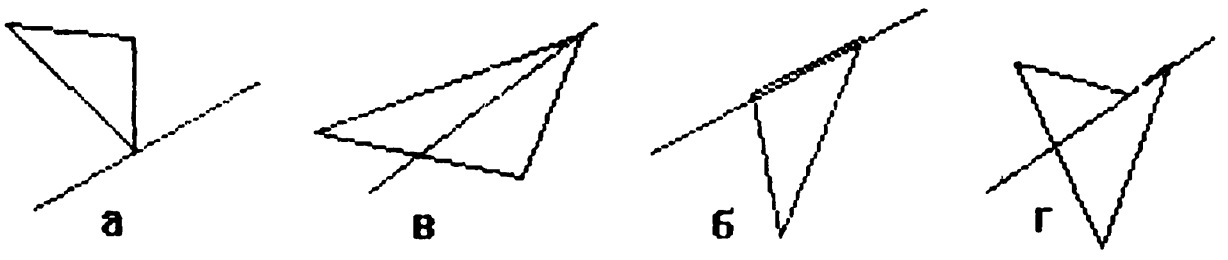


Рис. 8.5

Случай *в*, когда выходящие из вершины ребра лежат по разные стороны от луча, четность количества пересечений изменяется.

К случаям *б* и *г* такой подход непосредственно неприменим. Несколько изменим его, заметив, что в случаях *а* и *б* вершины, лежащие на луче, являются экстремальными значениями в тройке вершин соответствующих отрезков. В других же случаях экстремума нет.

Исходя из этого, можно построить следующий алгоритм: выпускаем из точки *A* горизонтальный луч в направлении оси *Ox* и все ребра многоугольника, кроме горизонтальных, проверяем на пересечение с этим лучом. В случае, когда луч проходит через вершину, т. е. формально пересекает сразу два ребра, сходящихся в этой вершине, засчитаем это пересечение только для тех ребер, для которых эта вершина является верхней. Подобный алгоритм приводит к следующей программе:

```
// File Polygon.cpp
#include    "polygon.h"

int  Polygon   addVertex ( const Vector2D& v, int after )
{
    if ( numVertices + 1 >= maxVertices )
        resize ( maxVertices + 5 );

    if ( after >= numVertices )
        vertices [numVertices] = v;
    else
    {
        memmove (vertices + after, vertices + after + 1,
                 (numVertices - after) * sizeof (Vector2D) );
        vertices [after] = v;
    }
    return ++numVertices;
}

int  Polygon   delVertex ( int index )
{
    if ( index < 0 || index >= numVertices )
        return 0;
    memmove ( &vertices [index], &vertices [index+1],
          (numVertices - index - 1) * sizeof ( Vector2D ) );
    return --numVertices;
}

int  Polygon   isInside ( const Vector2D& p )
{
    int  count = 0;   // count of ray/edge intersections
    for ( int i = 0; i < numVertices; i++ )
    {
        int  j = ( i + 1 ) % numVertices;
        if ( vertices [i].y == vertices [j].y )
            continue;
        if ( vertices [i].y > p.y && vertices [j].y > p.y )
            continue;
        if ( vertices [i].y < p.y && vertices [j].y < p.y )
            continue;
```

```
            if ( max ( vertices [i].y, vertices [j].y ) == p.y )
                count ++;
            else
            if ( min ( vertices [i].y, vertices [j].y ) == p.y )
                continue;
            else
            {
                float     t = (p.y - vertices [i].y)/
                                            (vertices [j].y - vertices [i].y);
                                if ( vertices [i].x + t * (vertices [j].x -
                                            vertices [i].x) >= p.x )
                    count ++;
            }
        }
        return count & 1;
}
Polygon * Polygon    split ( int from, int to )
{
        Polygon * p = new Polygon;
        if ( to < from )
            to += numVertices;
        for ( int i = from; i <= to; i++ )
            p -> addVertex ( vertices [i % numVertices] );
        if ( to < numVertices )
            memmove ( vertices+from+1, vertices+to+1,
                        (numVertices-to-1) * sizeof ( Vector2D ) );
        else
            memmove ( vertices, vertices+to,
                                (numVertices-to) * sizeof (Vector2D) );
        numVertices -= to - from - 1;
        return p;
}
void      Polygon    resize ( int newMaxVertices )
{
        if ( newMaxVertices < maxVertices )
            return;
        Vector2D * newVertices = new Vector2D [maxVertices = newMaxVertice
        if ( vertices != NULL )
        {
            memcpy ( newVertices, vertices, numVertices * sizeof (Vector2D) );
            delete [] vertices;
        }
}
```

8.6. Вычисление площади многоугольника

Для площади $s(P)$ многоугольника P, образованного вершинами v_0, v_1, ..., v_n, справедлива следующая формула:

$$s(P) = \frac{1}{2}\sum_{i=0}^{n-1}\left(v_{i,x}v_{i+1,y} - v_{i,y}v_{i+1,x}\right).$$

Эта формула дает площадь многоугольника со знаком, зависящим от ориентации его вершин. В случае, когда вершины упорядочены в направлении против часовой стрелки, $s(P) < 0$.

8.7. Построение звездчатого полигона

Пусть точки p и q лежат внутри некоторого многоугольника. Будем говорить, что точка q видна из точки p, если отрезок, соединяющий эти точки, целиком содержится в заданном многоугольнике.

Совокупность точек многоугольника, из которых видны все точки этого многоугольника, называется его ядром. Многоугольник называется звездчатым, если его ядро не пусто (рис. 8.6).

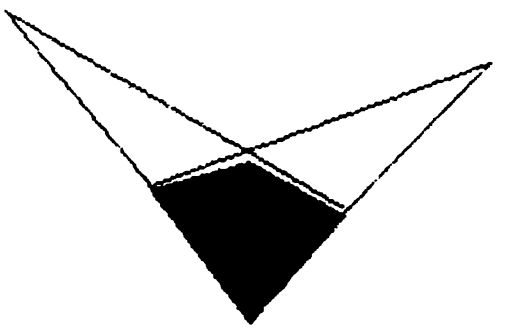


Рис. 8.6

Отметим, что ядро звездчатого многоугольника всегда выпукло.

Рассмотрим следующую задачу: на плоскости задан набор точек s_0, s_1, ..., s_{n-1}. Требуется построить "минимальный" звездчатый многоугольник так, чтобы его ядро содержало точку s_0.

Алгоритм работает итеративно, последовательно формируя из точек набора текущий многоугольник. Вначале многоугольник состоит из единственной точки s_0, и на каждой итерации в него добавляется очередная точка набора. По окончании обхода всех точек мы получим искомый звездчатый многоугольник.

Для определения того, в какое место границы текущего многоугольника нужно вставить точку s_i, заметим, что все вершины звездчатого многоугольника должны быть радиально упорядочены вокруг любой точки его ядра, а значит, и относительно точки s_0, принадлежащей ядру. Будем обходить границу многоугольника по часовой стрелке начиная с точки s_0, сравнивая каждый раз вставляемую точку и очередную точку границы. Функцию сравнения вершин polarCmp определим, основываясь на вычислении полярных координат (r, θ) точки относительно полюса s_0. Введем следующее правило сравнения: точка $p(V_p, \theta_p)$ считается меньше точки $q(V_q, \theta_q)$, если $\theta_p < \theta_q$ или $\theta_p = \theta_q$ и $V_p < V_q$. При таком упорядочении, обход текущего многоугольника по часовой стрелке будет производиться от больших точек к меньшим. Реализация алгоритма приводится ниже.

```
// File star.cpp
#include    "polygon.h"

Vector2D originPt;
```

```
int  polarCmp ( Vector2D * p, Vector2D * q )
{
    Vector2D vp       = *p - originPt;
    Vector2D vq       = *q - originPt;
    float    pAngle = vp.polarAngle ();
    float    qAngle = vq.polarAngle ();

    if ( pAngle < qAngle )
        return -1;

    if ( pAngle > qAngle )
        return 1;

    float    pLen = vp.length ();
    float    qLen = vq.length ();

    if ( pLen < qLen )
        return -1;

    if ( pLen > qLen )
        return 1;

    return 0;
}
Polygon * starPolgon ( Vector2D s [], int n )
{
   Polygon * p = new Polygon ( s, 1 );
   originPt = s [0];
   for ( int i = 1; i < n; i++ )
   {
     for ( int j = 1; polarCmp ( &s [i], &p->vertices [j] ) < 0; )
         if ( ++j >= p -> numVertices )
             j = 0;

         p -> addVertex ( s [i], j - 1 );
   }
   return p;
}
```

Несложно заметить, что временные затраты данного алгоритма составляют $O(n^2)$.

8.8. Построение выпуклой оболочки

Пусть S - конечный набор точек на плоскости. Выпуклой оболочкой $convS$ набора S называется пересечение всех выпуклых многоугольников, содержащих S. Ясно, что $convS$ - это выпуклый многоугольник, все вершины которого содержатся в S (заметим, что не все точки из S являются вершинами выпуклой оболочки).

Один из способов построения выпуклой оболочки конечного набора точек S на плоскости напоминает вычерчивание при помощи карандаша и линейки. Вначале выбирается точка $a \in S$, заведомо являющаяся вершиной границы выпуклой оболочки. В качестве такой точки можно взять самую левую точку из набора S (если таких точек несколько, выбираем самую нижнюю). Затем вертикальный луч поворачивается вокруг этой точки по направлению часовой стрелки до тех пор, пока не на-

ткнется на точку $b \in S$. Тогда отрезок ab будет ребром границы выпуклой оболочки. Для поиска следующего ребра будем продолжать вращение луча по часовой стрелке; на этот раз вокруг точки b до встречи со следующей точкой $c \in S$. Отрезок bc будет следующим ребром границы выпуклой оболочки. Процесс повторяется до тех пор, пока мы снова не вернемся в точку a. Этот метод называется методом "заворачивания подарка"

Основным шагом алгоритма является отыскание точки, следующей за точкой, вокруг которой вращается луч.

Следующая процедура реализует описанный алгоритм. Входной массив s должен иметь длину $n + 1$, где n - количество входных точек, поскольку в конец массива процедура записывает ограничивающий элемент $s[0]$.

```
// File giftwrap.cpp
#include    "polygon.h"

template <class T>
void    swap ( T a, T b )
{
    T c;

    c = a;
    a = b;
    b = a;
}
Polygon * giftWrapHull ( Vector2D s [], int n )
{
    int  a = 0;
         // find leftmost point
    for ( int i = 1; i < n; i++ )
        if ( s [i].x < s [a].x )
            a = i;
    s [n] = s [a];

    Polygon * p = new Polygon;

    for ( i = 0; i < n; i++ )
    {
        swap ( s [a], s [i] );

        p -> addVertex ( s [i], p -> numVertices );

        a = i + 1;

        for ( int j = i + 2;  j <= n; j++ )
        {
            int  c = s [j].classify ( s [i], s [a] );

            if ( c == LEFT || c == BEYOND )
                a = i;
        }
        if ( a == n )
            return p;
    }

    return NULL;
}
```

Временные затраты данного алгоритма равны $O(hn)$, где h - число вершин в границе искомой выпуклой оболочки. Работа данного алгоритма проиллюстрирована на рис. 8.7

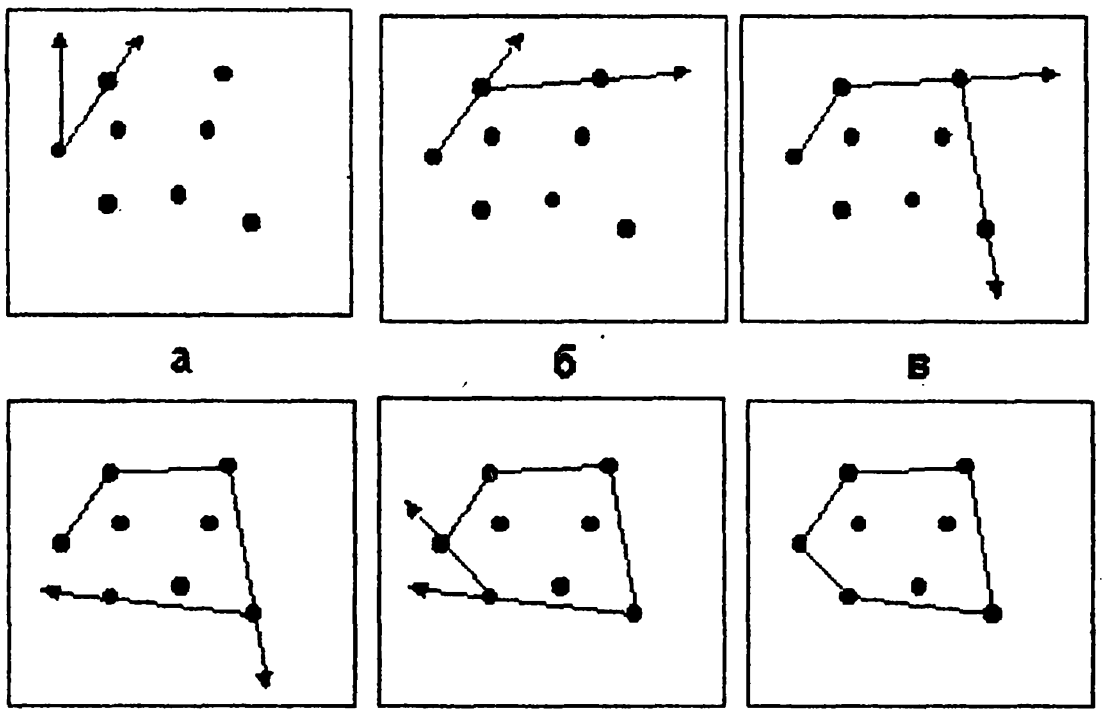


Рис. 8.7

Рассмотрим еще один алгоритм для построения выпуклой оболочки, так называемый метод обхода Грэхема. В этом методе выпуклая оболочка конечного набора точек S находится в два этапа. На первом этапе алгоритм выбирает некоторую экстремальную точку $\alpha \in S$ и сортирует все остальные точки в соответствии с углом направленного к ним из точки α луча. На втором этапе алгоритм выполняет пошаговую обработку отсортированных точек, формируя последовательность многоугольников, которые в конце концов и образуют искомую выпуклую оболочку *convS*.

Выберем в качестве экстремальной точки точку с минимальной y-координатой и поменяем ее местами с s_0. Остальные точки сортируются затем в порядке возрастания полярного угла относительно точки s_0. Если две точки имеют одинаковый полярный угол, то точка, расположенная ближе к s_0, должна стоять в отсортированном списке раньше, чем более дальняя точка. Это сравнение реализуется рассмотренной ранее функцией polarCmp.

Для определения того, какая именно точка должна быть включена в границу выпуклой оболочки после точки s_i, используется тот факт, что при обходе границы выпуклой оболочки в направлении по часовой стрелки каждая ее вершина должна соответствовать повороту влево.

```
// File graham.cpp
#include    <stdlib.h>
#include    "polygon.h"
#include    "stack.h"

template <class T>
void    swap ( T a, T b )
{
    T c;

    c = a;
    a = b;
    b = a;
}
```

```
Polygon * grahamScan ( Vector2D s [], int n )
{
                        // step 1
    for ( int i = 1, m = 0; i < n; i++ )
        if ( s [i] < s [m] )
            m = i;
    swap ( s [m], s [0] );
                        // step 2
    originPt = s [0];
    qsort ( & s [1], n - 1, sizeof ( Vector2D ), polarCmp );
                        // step 3
    for ( i = 1; s [i+1].classify (s [0], s [i]) == BEYOND; i++ )
        ;
    Stack stack;
    stack.push ( s [0] );
    stack.push ( s [i] );
                        // step 4
    for ( ++i; i < n; i++ )
    {
        while ( s [i].classify ( *.stack.nextToTop (),
                    *stack.top () ) != LEFT )
            stack.pop ();
        stack.push ( s [i] );
    }
                        // step 5
    Polygon * p = new Polygon;
    while ( !.stack.isEmpty () )
        p -> addVertex ( stack.pop (), p -> numVertices );
    return p;
}
```

Быстродействие данного алгоритма равно $O(n\log n)$.

8.9. Пересечение выпуклых многоугольников

Рассмотрим задачу об отсечении произвольного многоугольника по границе заданного выпуклого многоугольника. Одним из наиболее простых алгоритмов для решения этой задачи является алгоритм Сазерленда - Ходжмана, который мы и рассмотрим.

Алгоритм сводит исходную задачу к серии более простых задач об отсечении многоугольника вдоль прямой, проходящей через одно из ребер отсекающего многоугольника.

На каждом шаге (рис. 8.8) выбираем очередное ребро отсекающего многоугольника и поочередно проверяем положение всех вершин отсекаемого многоугольника относительно прямой, проходящей через выбранное текущее ребро. При этом в результирующий многоугольник добавляется 0, 1 или 2 вершины.

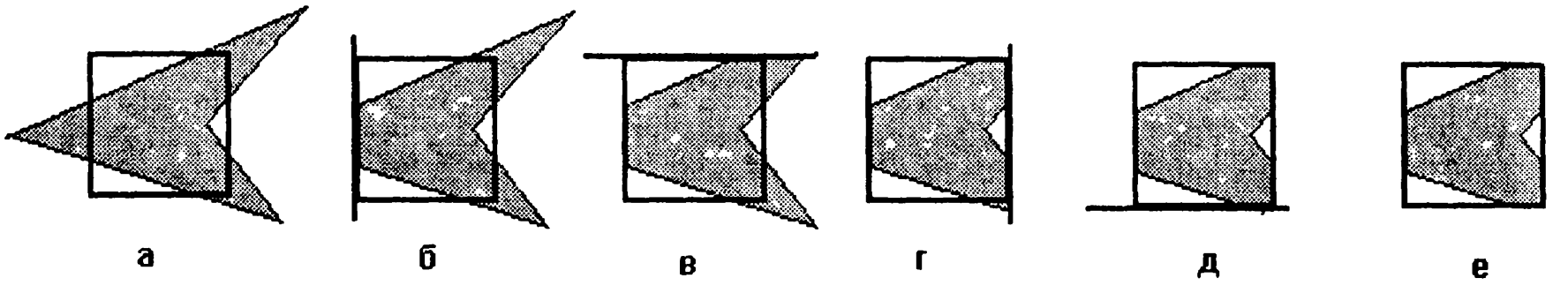


Рис. 8.8

Рассмотрим ребро отсекаемого многоугольника, соединяющее вершины *p*(*px*, *py*) и *c*(*cx*, *cy*). Возможны 4 различных ситуации (рис. 8.9).

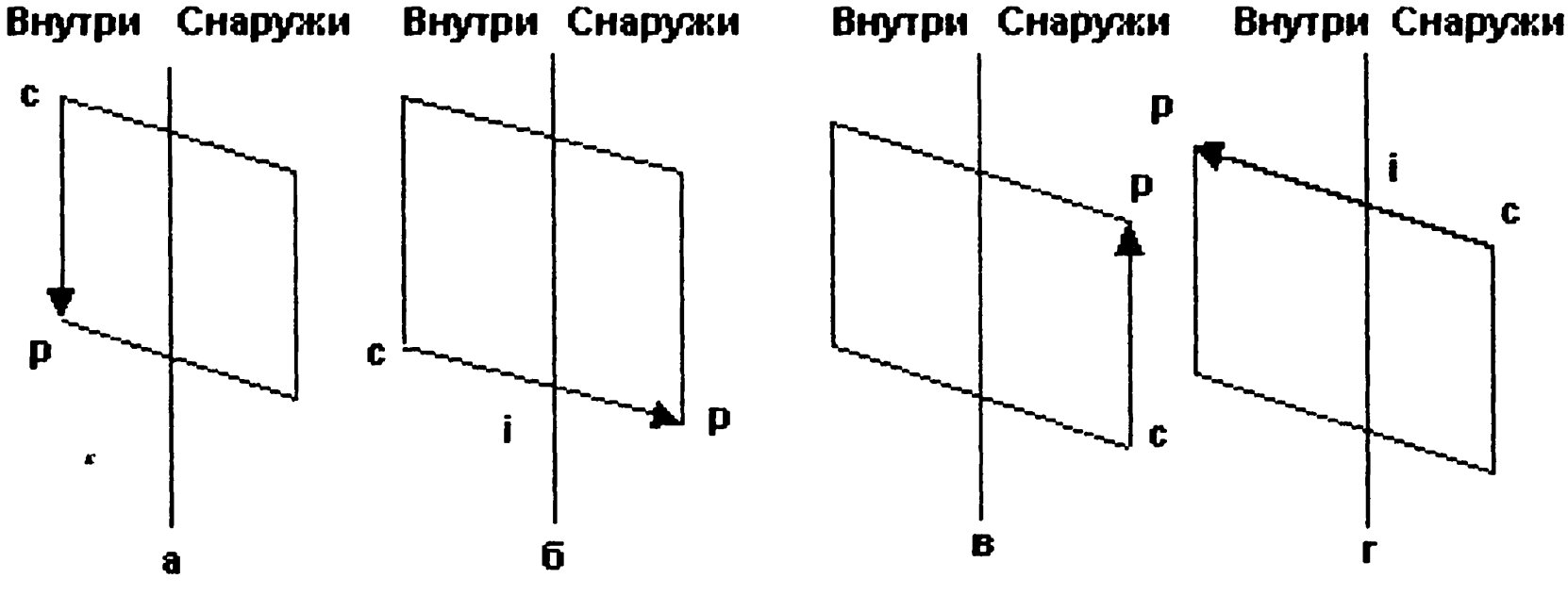


Рис. 8.9

Предположим, что точка *p* уже обработана. В случае *а* ребро целиком лежит во внутренней области и точка *c* добавляется в результирующий многоугольник. В случае *б* в результирующий многоугольник добавляется точка пересечения *i*. В случае в обе вершины лежат во внешней области и поэтому в результирующий многоугольник не добавляется ни одной точки. В случае *г* в результирующий многоугольник добавляются точка пересечения *i* и точка *c*.

Приводимая ниже функция clipPolygon реализует описанный алгоритм.

```
// File polyclip.cpp
#include     "polygon.h"

#define EPS          1e-7
#define MAX_VERTICES    100

inline    void addVertexToOutPolygon ( Vector2D * outPolygon,
     int& outVertices, int lastVertex, float x, float y )
{
    if ( outVertices == 0 ||
          (fabs (x - outPolygon [outVertices-1].x) > EPS ||
         fabs (y - outPolygon [outVertices-1].y) > EPS ) &&
         (!lastVertex || fabs (x - outPolygon [0].x) > EPS ||
         fabs (y - outPolygon [0].y) > EPS ) )
     {
          outPolygon [outVertices].x = x;
          outPolygon [outVertices].y = y;
          outVertices++;
     }
```

```
}
Polygon * clipPolygon (const Polygon& poly, const Polygon& clipPoly)
{
    Vector2D   outPoly1 [MAX_VERTICES];
    Vector2D   outPoly2 [MAX_VERTICES];
    Vector2D * inPolygon   = poly.vertices;
    Vector2D * outPolygon  = outPoly1;
    int     outVertices = 0;
    int     inVertices  = poly.numVertices;

    for ( int edge = 0; edge < clipPoly.numVertices; edge++ )
    {
       int    lastVertex = 0;        // is this the last vertex
       int    next = edge == clipPoly.numVertices-1 ? 0 : edge + 1;
       float  px = inPolygon [0].x;
       float  py = inPolygon [0].y;
       float  dx = clipPoly.vertices [next].x -
                      clipPoly.vertices [edge].x;
       float  dy = clipPoly.vertices [next].y -
                      clipPoly.vertices [edge].y;
       int    prevVertexInside = (px-clipPoly.vertices [edge].x)*dy
                 - (py - clipPoly.vertices [edge].y)*dx > 0;
       int    intersectionCount = 0;

       outVertices = 0;

       for ( int i = 1; i <= inVertices; i++ )
{
  float    cx, cy;

  if ( i < inVertices )
  {
      cx = inPolygon [i].x;
      cy = inPolygon [i].y;
  }
  else
  {
      cx = inPolygon [0].x;
      cy = inPolygon [0].y;
      lastVertex = 1;
  }

  // if starting vertex is visible then put it
  // into the output array
  if ( prevVertexInside )
      addVertexToOutPolygon ( outPolygon,
            outVertices, lastVertex, px, py );

  int curVertexInside =
         (cx-clipPoly.vertices [edge].x)*dy -
       (cy - clipPoly.vertices [edge].y)*dx > 0;

  // if vertices are on different sides of the edge
  // then look where we're intersecting
  if ( prevVertexInside != curVertexInside )
```

```
{
    double denominator = (cx-px)*dy - (cy-py)*dx;
    if ( denominator != 0.0 )
    {
        float     t =
    ((py-clipPoly.vertices [edge].y)*dx
    -(px-clipPoly.vertices [edge].x)*dy) /
     denominator;
        float     tx, ty;    // point of  intersection
        if ( t <= 0.0 )
        {
            tx = px;
            ty = py;
        }
        else
        if ( t >= 1.0 )
        {
            tx = cx;
            ty = cy;
        }
        else
        {
            tx = px + t*(cx - px);
            ty = py + t*(cy - py);
        }
            addVertexToOutPolygon ( outPolygon,
             outVertices,lastVertex, tx, ty );
        }
        if ( ++intersectionCount >= 2 )
        {
            if ( fabs (denominator) < 1 )
                intersectionCount = 0;
            else
            {    // drop out, after adding all  vertices left in input polygon
                if ( curVertexInside )
                {
                    memcpy ( &outPolygon [outVertices], &inPolygon [i],
                                (inVertices - i)* sizeof ( Vector2D ) );
                    outVertices += inVertices - i;
                }
                break;
            }
        }
    }
    px = cx;
    py = cy;
    prevVertexInside = curVertexInside;
}
```

```
            // if polygon is wiped out, break
            if ( outVertices < 3 )
                break;
            // switch input/output polygons
            inVertices = outVertices;
            inPolygon  = outPolygon;
            if ( outPolygon == outPoly2 )
                outPolygon = outPoly1;
            else
                outPolygon = outPoly2;
        }
        if ( outVertices < 3 )
            return NULL;
    return new Polygon ( outPolygon, outVertices )
}
```

Следует заметить, что приведенный алгоритм не является оптимальным по времени выполнения: доказано, что пересечение многоугольников p и q можно выполнить за время $O(n_p + n_q)$, где n_p и n_q - количество вершин многоугольников p и q, соответственно (см. [15]).

8.10. Построение триангуляции Делоне

Рассмотрим задачу триангуляции набора точек S на плоскости. Все точки набора S можно разбить на граничные - точки, лежащие на границе выпуклой оболочки *convS*, и внутренние - лежащие внутри *convS*. Ребра, полученные в результате триангуляции S, разбиваются на ребра оболочки и внутренние ребра. К ребрам оболочки относятся ребра, расположенные вдоль границы *convS*, а к внутренним - все остальные.

Любой набор точек (за исключением некоторых тривиальных случаев) допускает более одного способа триангуляции. Однако при этом выполняется следующее.

Утверждение. *Пусть набор S содержит $n \geq 3$ точек и не все из них коллинеарны. Пусть, кроме того, i из них являются внутренними (лежат внутри выпуклой оболочки convS). Тогда при любом способе триангуляции набора S будет получено $n + i - 2$ треугольников.*

Доказательство этого утверждения можно найти в [15].

Среди всех возможных триангуляций выделяется специальный вид - так называемая триангуляция Делоне. Эта триангуляция является хорошо сбалансированной в том смысле, что формируемые ей треугольники стремятся к равноугольности.

Определение. *Триангуляция набора точек S называется триангуляцией Делоне, если окружность, описанная вокруг каждого из треугольников, не будет содержать внутри себя точек набора S.*

На рис. 8.10, *а* показана окружность, не содержащая внутри себя точек набора S. Триангуляция, приведенная на рис. 10, *б*, не является триангуляцией Делоне, так как существует окружность, содержащая внутри себя одну из точек набора.

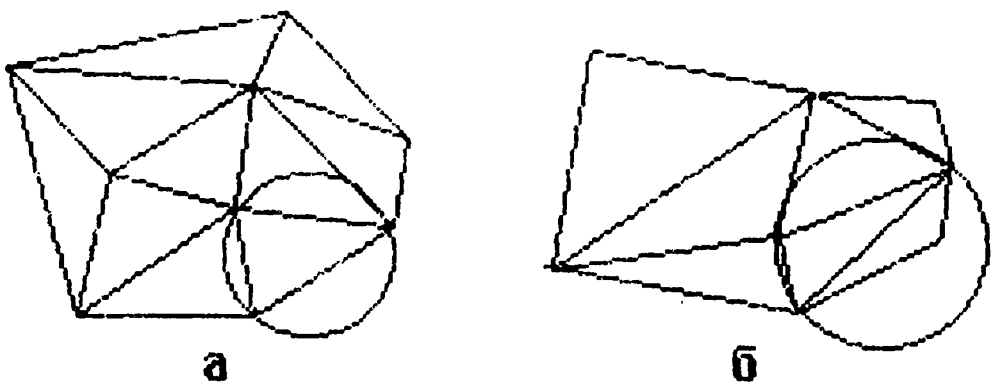


Рис. 8.10

Для упрощения алгоритма триангуляции сделаем два предположения относительно набора *S*: 1) чтобы триангуляция вообще существовала, необходимо, чтобы набор *S* содержал по крайней мере 3 неколлинеарные точки; 2) никакие 4 точки из набора *S* не лежат на одной окружности. Если последнее предположение не выполнено, то можно построить несколько различных триангуляций Делоне (см. упражнение 6).

Предлагаемый ниже алгоритм работает путем постоянного наращивания текущей триангуляции (по одному треугольнику за шаг). На каждой итерации алгоритм ищет новый треугольник, который подключается к текущей триангуляции.

В процессе триангуляции все имеющиеся ребра делятся на три класса:

- спящие ребра - ребра, которые еще не были обработаны алгоритмом;
- живые ребра - ребра, для каждого из которых известна только одна примыкающая к нему область;
- мертвые ребра - ребра, для каждого из которых известны обе примыкающие к нему области.

Вначале живым является единственное ребро, принадлежащее границе выпуклой оболочки, а все остальные ребра - спящие. По мере работы алгоритма ребра из спящих становятся живыми, а затем мертвыми.

На каждой итерации алгоритма выбирается одно из ребер границы и для него находится область, которой это ребро принадлежит. Если найденная область является треугольником, определяемым концевыми точками ребра и некоторой третьей точкой, то это ребро становится мертвым, поскольку известны обе примыкающие к нему области. Каждое из двух других ребер этого треугольника переводится или из спящего в живое или из живого в мертвое.

В случае если неизвестная область оказывается бесконечной плоскостью, то ребро просто умирает.

Для написания алгоритма понадобится класс Edge, приводимый ниже.

```
// File edge.h
#ifndef __EDGE__
#define __EDGE__

#include    "vector2d.h"

class   Edge
{
public:
    Vector2D org;
    Vector2D dest;

    Edge ( const Vector2D& p1, const Vector2D& p2 )
```

```
    {
        org  = p1,
        dest = p2;
    }
    Edge ( const Edge& e )
    {
        org  = e.org;
        dest = e.dest;
    }
    Edge () {}
    Edge&    flip ();
    Edge&    rot ();
    Vector2D point ( float t )
    {
        return org + t * (dest - org);
    }
    int   intersect ( const Edge&, float& );
};
enum        // types of edge intersection
{
    COLLINEAR, PARALLEL, SKEW
};
#endif
```

```
// File edge.cpp
#include    "edge.h"
Edge&  Edge :: rot ()
{
    Vector2D m = 0.5 * (org + dest);// center of the edge
    Vector2D n ( 0.5*(dest.y  -org.y), 0.5*(org.x - dest.x) );
    org  = m - n;
    dest = m + n;
    return *this;
}
Edge&  Edge :: flip ()
{
    Vector2D tmp = org;
    org  = dest;
    dest = tmp;
    return *this;
}
int  Edge   intersect ( const Edge& e, float& t )
{
    Vector2D n ( e.dest.y - e.org.y, e.org.x - e.dest.x );
    float      denom = n & (dest - org);
```

```
    if ( denom == 0.0 )
    {
        int   cls = org.classify ( e.org, e.dest );
        if ( cls == LEFT || cls == RIGHT )
            return PARALLEL;
        return COLLINEAR;
    }
    t = - (n & (org - e.org)) / denom;
    return SKEW;
}
```

Ниже приводится программа для построения триангуляции Делоне зада набора точек.

```
// File delaunay.cpp
#include    "polygon.h"
#include    "array.h"
#include    "dict.h"
#include    "edge.h"
#include    <values.h>

template <class T>
void     swap ( T a, T b )
{
    T c;
    c = a;
    a = b;
    b = a;
}
static int edgeCmp ( Edge * a, Edge * b )
{
    if ( a -> org < b -> org )
        return -1;
    if ( a -> org > b -> org )
        return 1;
    if ( a -> dest < b -> dest )
        return -1;
    if ( a -> dest > b -> dest )
        return 1;
    return 0;
}
static void   updateFrontier ( Dictionary& frontier, Vector2D& a, Vector2D& b )
{
    Edge  * e = new Edge ( a, b );
    if ( frontier.find ( e ) )
        frontier.del ( e );
    else
    {
```

```
            e -> flip ();
            frontier.insert ( e );
        }
    }
    static Edge * hullEdge ( Vector2D s [], int n )
    {
        int  m = 0;
        for ( int i = 1; i < n; i++ )
            if ( s [i] < s [m] )
                m = i;
        swap ( s [0], s [m] );
        for ( m = 1, i = 2; i < n; i++ )
        {
            int  cls = s [i].classify ( s [0], s [m] );
            if ( cls == LEFT || cls == BETWEEN )
                m = i;
        }
        return new Edge ( s [0], s [m] );
    }
    static Polygon * traingle ( const Vector2D& a, const Vector2D& b,
                                const Vector2D& c )
    {
        Polygon * t = new Polygon;
        t -> addVertex ( a );
        t -> addVertex ( b );
        t -> addVertex ( c );
        return t;
    }
    static int mate ( Edge& e, Vector2D s [], int n, Vector2D& p )
    {
        float    t;
        float    bestT = MAXFLOAT;
        Edge   f ( e );
        f.rot ();
        for ( int i = 0; i < n; i++ )
        {
            if ( s [i].classify ( e.org, e.dest ) == RIGHT )
            {
                Edge  g ( e.dest, s [i] );
                g.rot ();
                if ( f.intersect ( g, t ) == SKEW && t < bestT )
                {
                    bestT = t;
                    p     = s [i];
                }
            }
```

```
        }
        return t < MAXFLOAT;
    }
    Array * delaunayTriangulate ( Vector2D s [], int n )
    {
        Array      *  traingles = new Array;
        Dictionary    frontier ( edgeCmp );
        Vector2D      p;
        Edge       *  e = hullEdge ( s, n );
        fronier.insert ( e );
        while ( !frontier.isEmpty () )
        {
            e = fronier.removeAt ( 0 );
            if ( mate ( *e, s, n, p ) )
            {
                updateFrontier ( frontier, p, e -> org );
                updateFrontier ( frontier, e -> dest, p );
                traingles -> insert (
                                    triangle ( e -> org, e -> dest, p ) );
            }
            delete e;
        }
        return triangles;
    }
```

Треугольники, образуемые в процессе триангуляции, хранятся в массиве traingles. Все живые ребра хранятся в словаре frontier. Причем каждое ребро направлено так, что неизвестная для него область лежит справа.

Рассмотрим, каким образом функция updateFrontier изменяет словарь живых ребер. При добавлении к массиву triangles нового треугольника *t* изменяется состояние всех трех ребер треугольника. Ребро треугольника, примыкающее к границе, из живого становится мертвым. Каждое из двух оставшихся ребер изменяет свое состояние из спящего в живое, если оно ранее не было записано в словарь, или из живого в мертвое, если оно в словаре уже было.

На рис. 8.11 показаны оба случая. При обработке живого ребра *af*, после обнаружения того, что точка *b* является сопряженной к нему, добавляем к списку построенных треугольников треугольник *afb*. Затем ищем ребро *fb* в словаре. Поскольку оно обнаружено впервые, то его там еще нет и состояние ребра *fb* изменяется от спящего к живому.

Перед записыванием ребра *fb* в словарь, перевернем его так, чтобы примыкающая к нему неизвестная область лежала от него справа.

Ребро *ba* в словаре уже есть. Так как неизвестная для него область (треугольник *afb*) только что была обнаружена, это ребро из словаря удаляется.

Функция hullEdge строит ребро, принадлежащее границе выпуклой оболочки *convS*. В этой функции фактически реализован этап инициализации и первой итерации метода "заворачивания подарка"

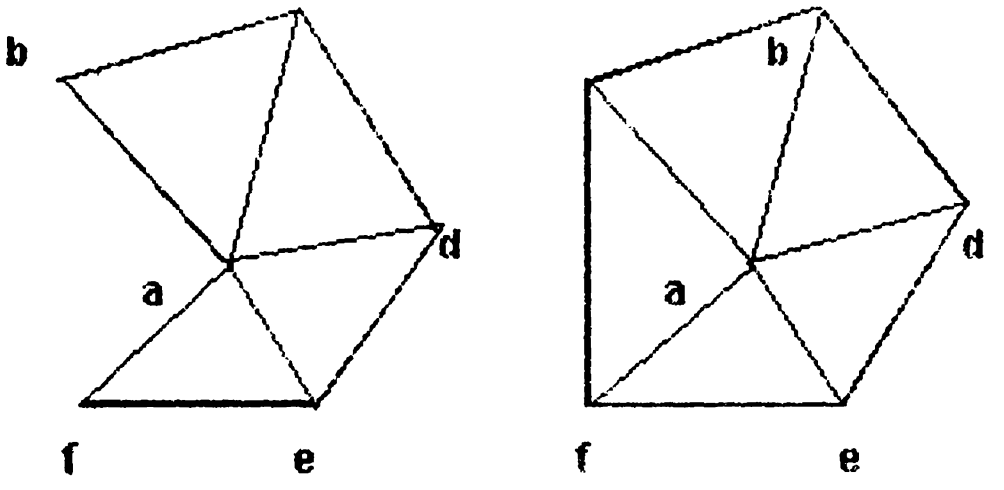


Рис. 8.11

Функция mate определяет, существует ли для данного живого ребра *e* сопряженная ему точка, и, если она есть, находит ее. Для того чтобы понять, как работает эта функция, рассмотрим множество окружностей, проходящих через концевые точки ребра *e*. Центры всех этих окружностей лежат на срединном перпендикуляре к ребру.

Рассмотрим процесс "надувания" окружности, проходящей через *e*. Если в результате такого "надувания" окружность пройдет через некоторую точку из набора *S*, то эта точка является сопряженной к ребру *e*, в противном случае ребро сопряженной точки не имеет.

Быстродействие данного алгоритма равно $O(n^2)$.

Упражнения

1. Покажите, что разность $x_a y_b - x_b y_a$ равна площади (со знаком) параллелограмма, определяемого векторами $a = (x_a y_a)$ и $b = (x_b y_b)$.
2. Напишите функцию, которая проверяет, является ли данный многоугольник выпуклым.
3. Покажите, что любая точка выпуклого многоугольника с вершинами $v_1, v_2, \ldots, v_n$ может быть записана в виде $a_1 v_1 + \ldots + a_n v_n$, где $a_1 + \ + a_n = 1$ и. $a_1 \geq 0, \ldots, a_n \geq 0$
4. Напишите функцию для нахождения пересечения двух выпуклых многоугольников p и q, работающую за время $O(n_p + n_q)$.
5. Диаметр набора точек определяется как максимальное расстояние между любыми двумя точками набора. Напишите функцию, вычисляющую за время $O(n \log n)$ диаметр набора из n точек плоскости.
6. Покажите, что триангуляция Делоне однозначно определена, если никакие 4 точки набора S не принадлежат одной окружности.

Глава 9
Преобразования в пространстве, проектирование

Обратимся теперь к трехмерному случаю (3D) (3-dimension) и начнем наше рассмотрение сразу с введения однородных координат.

Поступая аналогично тому, как это было сделано в размерности два, заменим координатную тройку (x, y, z), задающую точку в пространстве, на четверку чисел $(x, y, z, 1)$ или, более общо, на четверку

(hx, hy, hz, h), $h \neq 0$.

Каждая точка пространства (кроме начальной точки O) может быть задана четверкой одновременно не равных нулю чисел; эта четверка чисел определена однозначно с точностью до общего множителя.

Предложенный переход к новому способу задания точек дает возможность воспользоваться матричной записью и в более сложных, трехмерных задачах.

Любое аффинное преобразование в трехмерном пространстве может быть представлено в виде суперпозиции вращений, растяжений, отражений и переносов. Поэтому вполне уместно сначала подробно описать матрицы именно этих преобразований (ясно, что в данном случае порядок матриц должен быть равен четырем).

А. Матрицы вращения в пространстве.

Матрица вращения вокруг оси абсцисс на угол φ:

$$[R_x] = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\varphi & \sin\varphi & 0 \\ 0 & -\sin\varphi & \cos\varphi & \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Матрица вращения вокруг оси ординат на угол ψ:

$$[R_y] = \begin{bmatrix} \cos\psi & 0 & -\sin\psi & 0 \\ 0 & 1 & 0 & 0 \\ \sin\psi & 0 & \cos\psi & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Матрица вращения вокруг оси аппликат на угол χ:

$$[R_y] = \begin{bmatrix} \cos\chi & \sin\chi & 0 & 0 \\ -\sin\chi & \cos\chi & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Замечание. Полезно обратить внимание на место знака "-" в каждой из трех приведенных матриц.

Б. Матрица растяжения (сжатия):

$$[D]=\begin{bmatrix}\alpha & 0 & 0 & 0\\ 0 & \beta & 0 & 0\\ 0 & 0 & \gamma & 0\\ 0 & 0 & 0 & 1\end{bmatrix}$$

где $\alpha > 0$ - коэффициент растяжения (сжатия) вдоль оси абсцисс;
$\beta > 0$ - коэффициент растяжения (сжатия) вдоль оси ординат;
$\gamma > 0$ - коэффициент растяжения (сжатия) вдоль оси аппликат.

В. Матрицы отражения.

Матрица отражения относительно плоскости *xy*:

$$[M_z]=\begin{bmatrix}1 & 0 & 0 & 0\\ 0 & 1 & 0 & 0\\ 0 & 0 & -1 & 0\\ 0 & 0 & 0 & 1\end{bmatrix}$$

Матрица отражения относительно плоскости *yz*:

$$[M_x]=\begin{bmatrix}-1 & 0 & 0 & 0\\ 0 & 1 & 0 & 0\\ 0 & 0 & 1 & 0\\ 0 & 0 & 0 & 1\end{bmatrix}.$$

Матрица отражения относительно плоскости *zx*:

$$[M_y]=\begin{bmatrix}1 & 0 & 0 & 0\\ 0 & -1 & 0 & 0\\ 0 & 0 & 1 & 0\\ 0 & 0 & 0 & 1\end{bmatrix}.$$

Г. Матрица переноса (здесь (λ, μ, ν) - вектор переноса):

$$[T]=\begin{bmatrix}1 & 0 & 0 & 0\\ 0 & 1 & 0 & 0\\ 0 & 0 & 1 & 0\\ \lambda & \mu & \nu & 1\end{bmatrix}.$$

Замечание. Как и в двумерном случае, все выписанные матрицы невырожденны.

Приведем важный пример построения матрицы сложного преобразования по его геометрическому описанию.

Пример 1. Построить матрицу вращения на угол φ вокруг прямой L, проходящей через точку A(a, b, c) и имеющую направляющий вектор (l, m, n). Можно считать, что направляющий вектор прямой является единичным:

$l^2 + m^2 + n^2 = 1.$

На рис. 9.1 схематично показано, матрицу какого преобразования требуется найти.

Решение сформулированной задачи разбивается на несколько шагов. Опишем последовательно каждый из них.

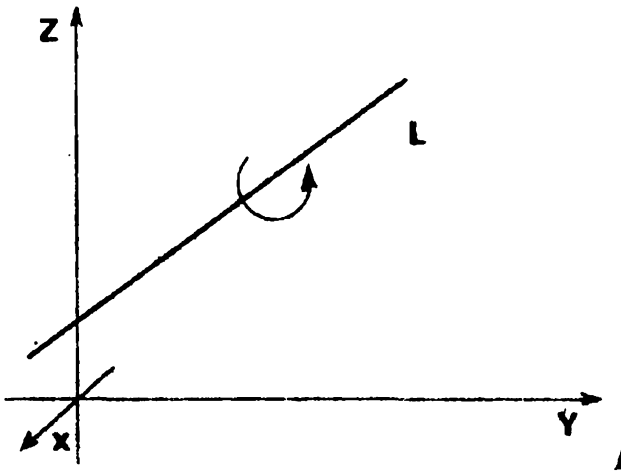


Рис. 9.1

1-й шаг. Перенос на вектор $-A(-a, -b, -c)$ при помощи матрицы

$$[T]=\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -a & -b & -c & 1 \end{bmatrix}.$$

В результате этого переноса мы добиваемся того, чтобы прямая L проходила через начало координат.

2-й шаг. Совмещение оси аппликат с прямой L двумя поворотами вокруг оси абсцисс и оси ординат.

Первый поворот - вокруг оси абсцисс на угол ψ (подлежащий определению). Чтобы найти этот угол, рассмотрим ортогональную проекцию L' исходной прямой L на плоскость $X = 0$ (рис. 9.2).

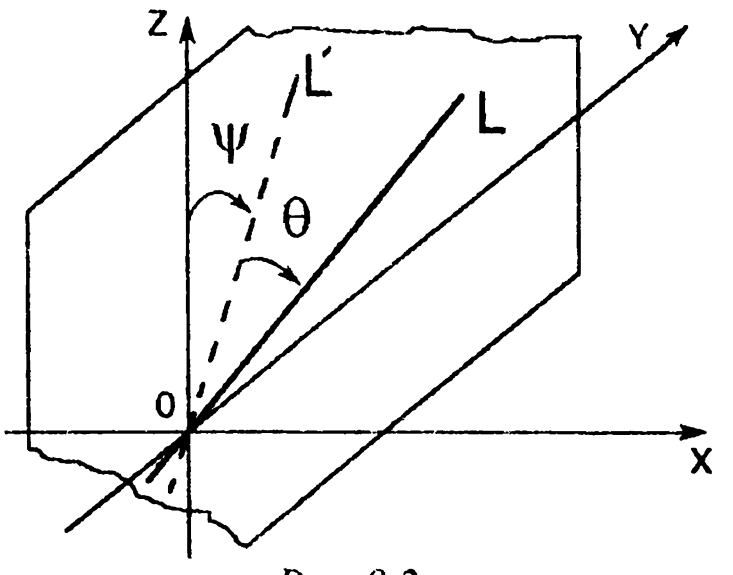


Рис. 9.2

Направляющий вектор прямой L' определяется просто - он равен $(0, m, n)$. Отсюда сразу же вытекает, что

$$\cos\psi = \frac{n}{d}, \quad \sin\psi = \frac{m}{d},$$

где $d = \sqrt{m^2 + n^2}$

Соответствующая матрица вращения имеет следующий вид:

$$[R_x]=\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \frac{n}{d} & \frac{m}{d} & 0 \\ 0 & -\frac{m}{d} & \frac{n}{d} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

Под действием преобразования, описываемого этой матрицей, координаты вектора (l, m, n) изменятся. Подсчитав их, в результате получим

$$(l, m, n, 1)\,[R_x] = (l, 0, d, 1).$$

Второй поворот - вокруг оси ординат на угол θ, определяемый соотношениями $\cos\theta = l, \sin\theta = -d$

Соответствующая матрица вращения записывается в следующем виде:

$$[R_y] = \begin{bmatrix} l & 0 & d & 0 \\ 0 & 1 & 0 & 0 \\ -d & 0 & l & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

3-й шаг. Вращение вокруг прямой L на заданный угол φ.

Так как теперь прямая L совпадает с осью аппликат, то соответствующая матрица имеет следующий вид:

$$[R_z] = \begin{bmatrix} \cos\varphi & \sin\varphi & 0 & 0 \\ -\sin\varphi & \cos\varphi & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

4-й шаг. Поворот вокруг оси ординат на угол $-\theta$.

5-й шаг. Поворот вокруг оси абсцисс на угол $-\psi$.

<u>Замечание</u>. Вращение в пространстве некоммутативно. Поэтому порядок, в котором проводятся вращения, является весьма существенным.

6-й шаг. Перенос на вектор $A(a, b, c)$.

Перемножив найденные матрицы в порядке их построения, получим следующую матрицу:

$$[T][R_x][R_y][R_z][R_y]^{-1}[R_x]^{-1}[T]^{-1}$$

Выпишем окончательный результат, считая для простоты, что ось вращения L проходит через начальную точку:

$$\begin{pmatrix} l^2 + \cos\varphi(1 - l^2) & l(1 - \cos\varphi)m + n\sin\varphi & l(1 - \cos\varphi)n - m\sin\varphi & 0 \\ l(1 - \cos\varphi)m - n\sin\varphi & m^2 + \cos\varphi(1 - m^2) & m(1 - \cos\varphi)n + l\sin\varphi & 0 \\ l(1 - \cos\varphi)n + m\sin\varphi & m(1 - \cos\varphi)n - l\sin\varphi & n^2 + \cos\varphi(1 - n^2) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

Рассматривая другие примеры подобного рода, мы будем получать в результате невырожденные матрицы вида

$$[A] = \begin{bmatrix} \alpha_1 & \alpha_2 & \alpha_3 & 0 \\ \beta_1 & \beta_2 & \beta_3 & 0 \\ \gamma_1 & \gamma_2 & \gamma_3 & 0 \\ \lambda & \mu & \nu & 1 \end{bmatrix}$$

При помощи таких матриц можно преобразовывать любые плоские и пространственные фигуры.

Пример 2. Требуется подвергнуть заданному аффинному преобразованию выпуклый многогранник.

Для этого сначала по геометрическому описанию отображения находим его матрицу [*A*]. Замечая далее, что произвольный выпуклый многогранник однозначно задается набором всех своих вершин

$$V_i(x_i, y_i, z_i), i = 1,\ldots,n,$$

строим матрицу

$$V = \begin{pmatrix} x_1 & y_1 & z_1 & 1 \\ & & & \\ x_n & y_n & z_n & 1 \end{pmatrix}.$$

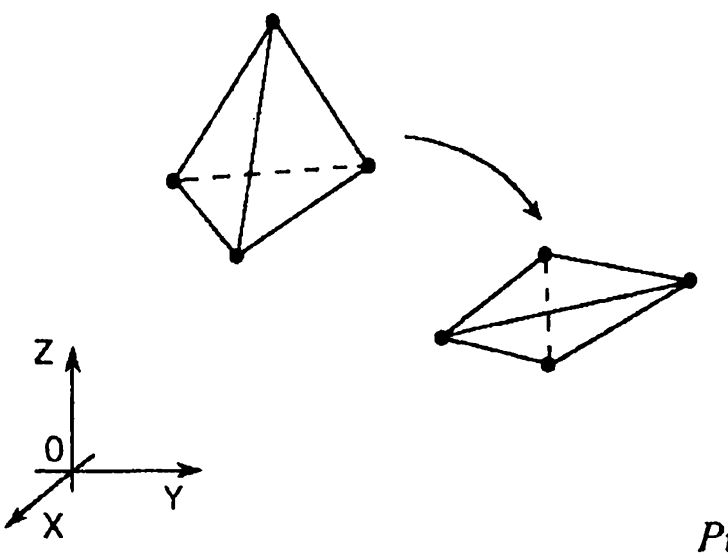


Рис. 9.3

Подвергая этот набор преобразованию, описываемому найденной невырожденной матрицей четвертого порядка [*V*][*A*], мы получаем набор вершин нового выпуклого многогранника - образа исходного (рис. 9.3).

9.1. Платоновы тела

Правильными многогранниками (платоновыми телами) называются такие выпуклые многогранники, все грани которых суть правильные многоугольники и все многогранные углы при вершинах равны между собой.

Существует ровно 5 правильных многогранников (это доказал Евклид): правильный тетраэдр, гексаэдр (куб), октаэдр, додекаэдр и икосаэдр. Их основные характеристики приведены в следующей таблице.

Название многогранника	*Число граней - Г*	*Число ребер - Р*	*Число вершин - В*
Тетраэдр	4	6	4
Гексаэдр	6	12	8
Октаэдр	8	12	6
Додекаэдр	12	30	20
Икосаэдр	20	30	12

Нетрудно заметить, что в каждом из пяти случаев числа *Г*, *Р* и *В* связаны равенством Эйлера

$$Г + В = Р + 2.$$

Правильные многогранники обладают многими интересными свойствами. Здесь мы коснемся только тех свойств, которые можно применить для построения этих многогранников.

Для полного описания правильного многогранника вследствие его выпуклости достаточно указать способ отыскания всех его вершин.

Операции построения первых трех платоновых тел являются особенно простыми. С них и начнем.

Куб (гексаэдр) строится совсем несложно (рис. 9.4).

Покажем, как, используя куб, можно построить тетраэдр и октаэдр.

Для построения тетраэдра достаточно провести скрещивающиеся диагонали противоположных граней куба (рис. 9.5).

Тем самым вершинами тетраэдра являются любые 4 вершины куба, попарно не смежные ни с одним из его ребер.

Для построения октаэдра воспользуемся следующим свойством двойственности: вершины октаэдра суть центры (тяжести) граней куба (рис. 9.6).

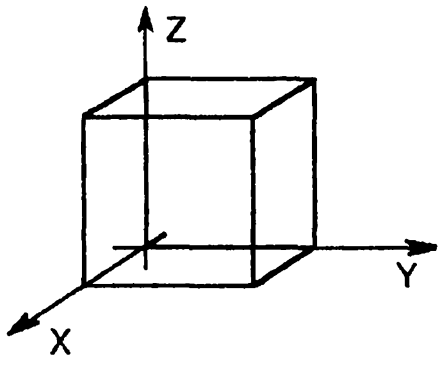


Рис. 9.4

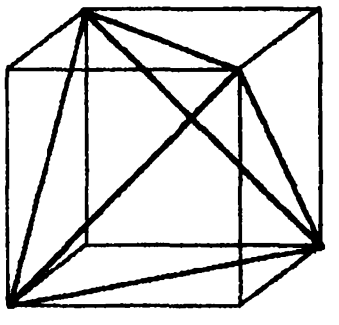


Рис. 9.5

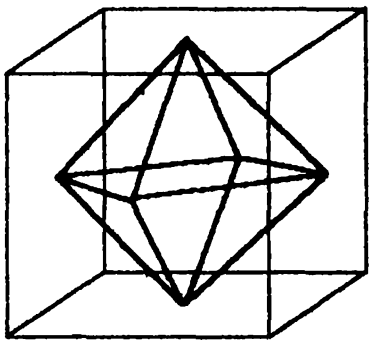


Рис. 9.6

И значит, координаты вершин октаэдра по координатам вершин куба легко вычисляются (каждая координата вершины октаэдра является средним арифметическим одноименных координат четырех вершин содержащей ее грани куба).

Додекаэдр и икосаэдр также можно построить при помощи куба. Однако существует, на наш взгляд, более простой способ их конструирования, который мы и собираемся описать здесь.

Начнем с икосаэдра.

Рассечем круглый цилиндр единичного радиуса, ось которого совпадает с осью аппликат Z двумя плоскостями Z = -0,5 и Z = 0,5 (рис. 9.7). Разобьем каждую из полученных окружностей на 5 равных частей так, как показано на рис. 9.8. Перемещаясь вдоль обеих окружностей против часовой стрелки, занумеруем выделенные 10 точек в порядке возрастания угла поворота (рис. 9.9) и затем последовательно, в соответствии с нумерацией, соединим эти точки прямолинейными отрезками (рис. 9.10).

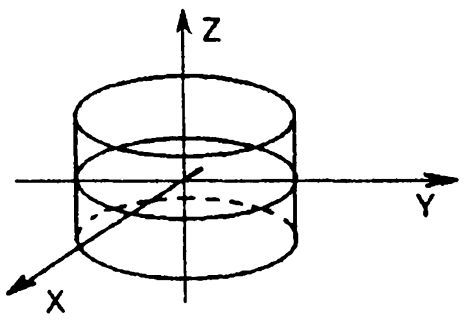


Рис. 9.7

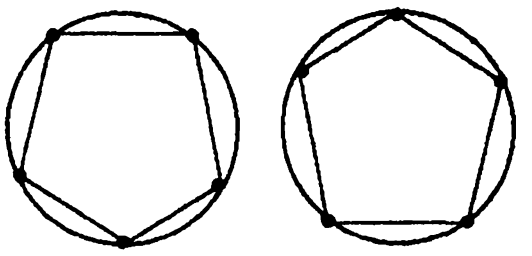


Рис. 9.8

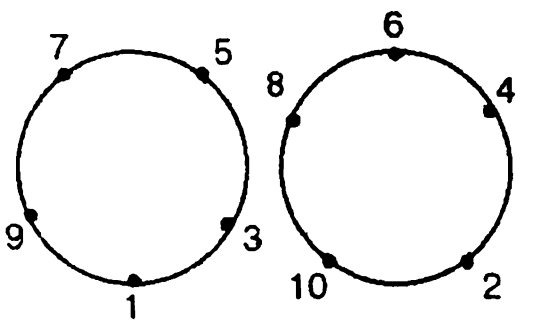


Рис. 9.9

Стягивая теперь хордами точки, выделенные на каждой из окружностей, мы получим в результате пояс из 10 правильных треугольников (рис. 9.11).

Для завершения построения икосаэдра выберем на оси Z две точки так, чтобы длины боковых ребер пятиугольных пирамид с вершинами в этих точках и основаниями, совпадающими с построенными пятиугольниками (рис. 9.12), были равны длинам сторон пояса из треугольников.

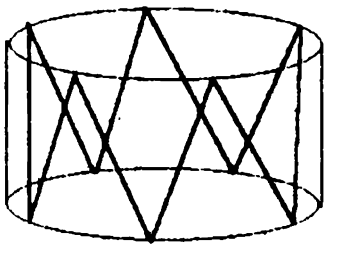


Рис. 9.10

Нетрудно видеть, что для этого годятся точки с аппликатами $\pm\frac{\sqrt{5}}{2}$

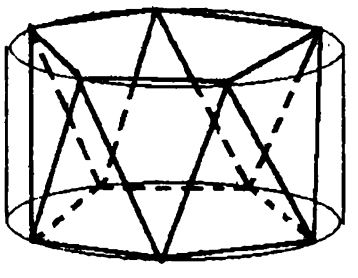


Рис. 9.11

В результате описанных построений получаем 12 точек. Выпуклый многогранник с вершинами в этих точках будет иметь 20 граней, каждая из которых является правильным треугольником, и все его многогранные углы при вершинах будут равны между собой. Тем самым результат описанного построения - икосаэдр (рис. 9.13).

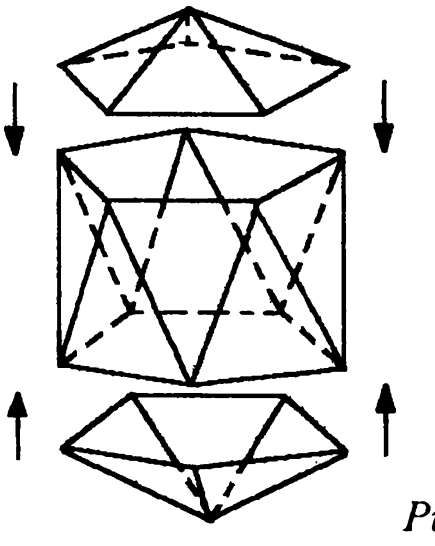


Рис. 9.12

Декартовы координаты вершин построенного икосаэдра легко вычисляются. Для двух вершин они уже найдены, а что касается остальных 10 вершин икосаэдра, то достаточно заметить, что полярные углы соседних вершин треугольного пояса разнятся на 36°, а их полярные радиусы равны единице.

Остается построить додекаэдр.

Оставляя в стороне способ, предложенный Евклидом (построение "крыш" над гранями куба), вновь воспользуемся свойством двойственности, но теперь уже связывающим додекаэдр и икосаэдр: вершины додекаэдра суть центры (тяжести) треугольных граней икосаэдра.

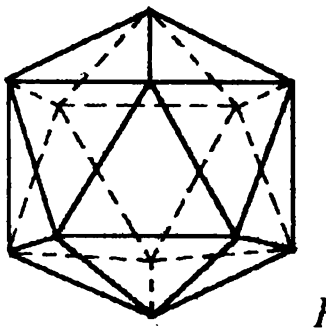


Рис. 9.13

И значит, координаты каждой вершины додекаэдра можно найти, вычислив средние арифметические соответствующих координат вершин содержащей ее грани икосаэдра (рис. 9.14).

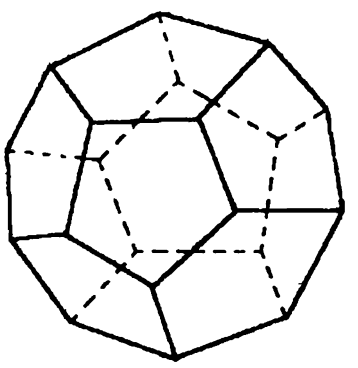


Рис. 9.14

Замечание. Подвергая полученные правильные многогранники преобразованиям вращения и переноса, можно получить платоновы тела с центрами в произвольных точках и с любыми длинами ребер.

В качестве упражнения полезно написать по предложенным способам программы, генерирующие все платоновы тела.

9.2. Виды проектирования

Изображение объектов на картинной плоскости связано с еще одной геометрической операцией - проектированием при помощи пучка прямых. В компьютерной

графике используется несколько различных видов проектирования (иногда называемого также проецированием). Наиболее употребимые на практике виды проектирования суть параллельное и центральное.

Для получения проекции объекта на картинную плоскость необходимо провести через каждую его точку прямую из заданного проектирующего пучка (собственного или несобственного) и затем найти координаты точки пересечения этой прямой с плоскостью изображения. В случае центрального проектирования все прямые исходят из одной точки - центра собственного пучка. При параллельном проектировании центр (несобственного) пучка считается лежащим в бесконечности (рис. 9 15).

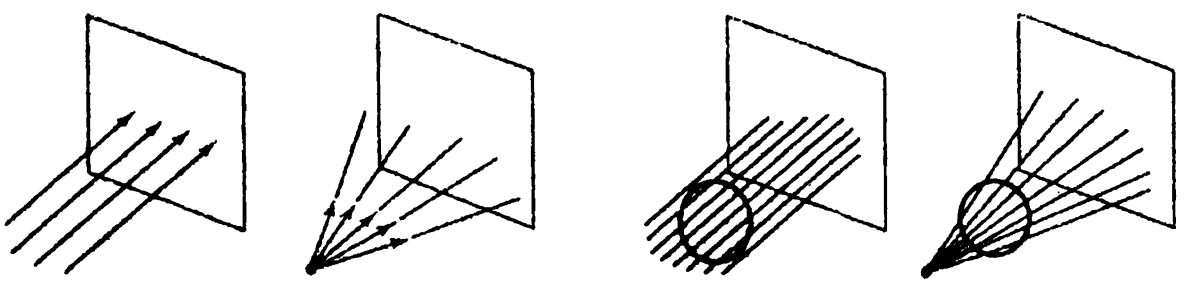


Рис. 9.15

Каждый из этих двух основных классов разбивается на несколько подклассов в зависимости от взаимного расположения картинной плоскости и координатных осей. Некоторое представление о видах проектирования могут дать приводимые ниже схемы.

Схема 1

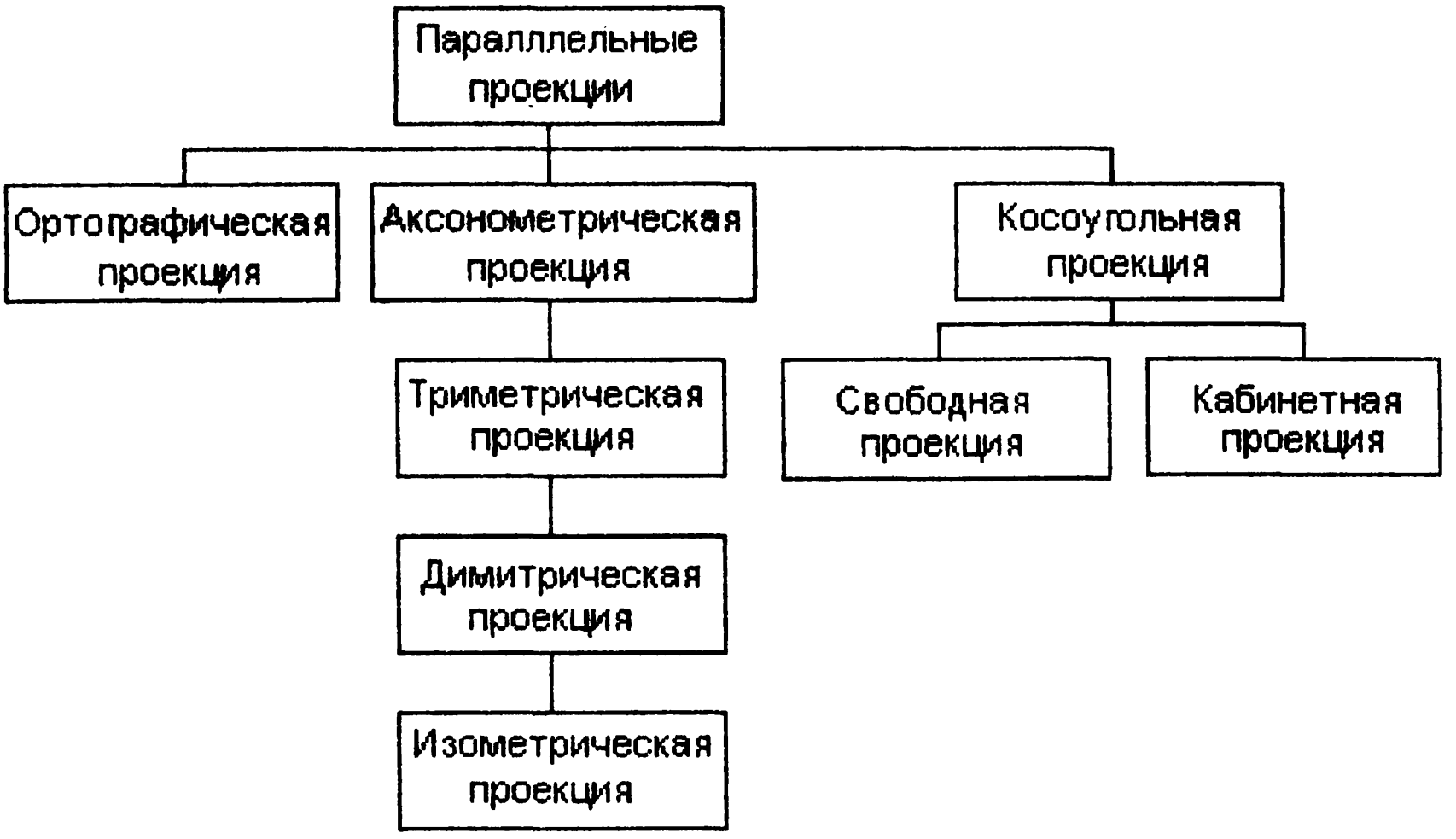
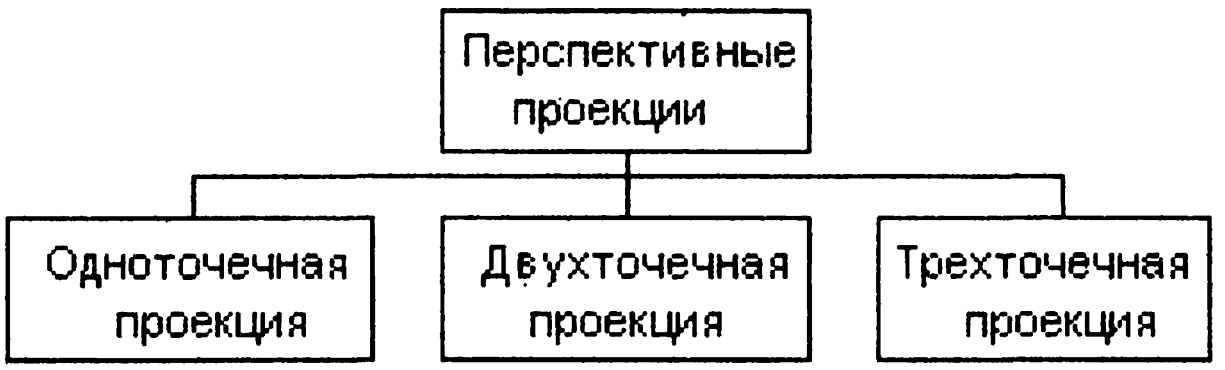


Схема 2



Важное замечание. *Использование для описания преобразований проектирования однородных координат и матриц четвертого порядка позволяет упростить изложение и зримо облегчает решение задач геометрического моделирования.*

При ортографической проекции картинная плоскость совпадает с одной из координатных плоскостей или параллельна ей (рис. 9.16). Матрица проектирования вдоль оси X на плоскость YZ имеет вид:

$$[P_x]=\begin{bmatrix}0&0&0&0\\0&1&0&0\\0&0&1&0\\0&0&0&1\end{bmatrix}$$

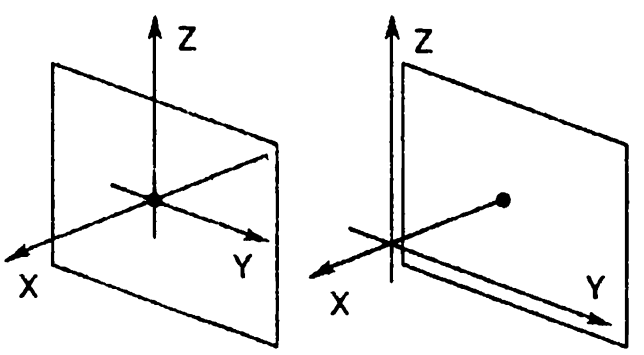


Рис. 9.16

В случае, если плоскость проектирования параллельна координатной плоскости, необходимо умножить матрицу $[P_x]$ на матрицу сдвига. В результате получаем

$$[P_x]\cdot\begin{bmatrix}1&0&0&0\\0&1&0&0\\0&0&1&0\\p&0&0&0\end{bmatrix}=\begin{bmatrix}0&0&0&0\\0&1&0&0\\0&0&1&0\\p&0&0&1\end{bmatrix}$$

Аналогично записываются матрицы проектирования вдоль двух других координатных осей:

$$\begin{bmatrix}1&0&0&0\\0&0&0&0\\0&0&1&0\\0&q&0&1\end{bmatrix},\begin{bmatrix}1&0&0&0\\0&1&0&0\\0&0&0&0\\0&0&r&1\end{bmatrix}.$$

Замечание. Все три полученные матрицы проектирования вырожденны.

При аксонометрической проекции проектирующие прямые перпендикулярны картинной плоскости.

В соответствии со взаимным расположением плоскости проектирования и координатных осей различают три вида проекций:

- триметрию - нормальный вектор картинной плоскости образует с ортами координатных осей попарно различные углы (рис. 9.17);

- диметрию - два угла между нормалью картинной плоскости и координатными осями равны (рис. 9.18);
- изометрию - все три угла между нормалью картинной плоскости и координатными осями равны (рис. 9.19).

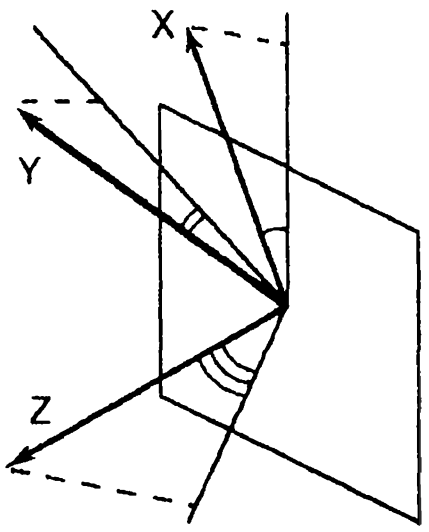


Рис. 9.17

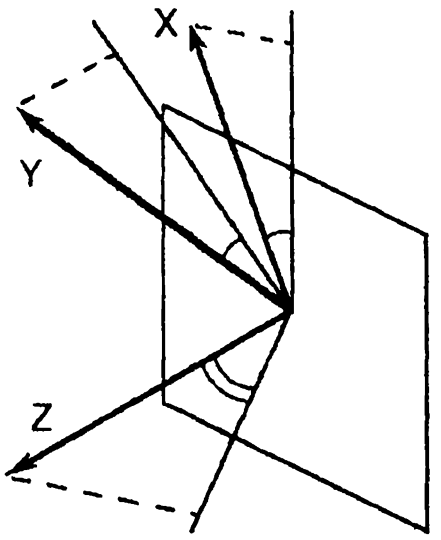


Рис. 9.18

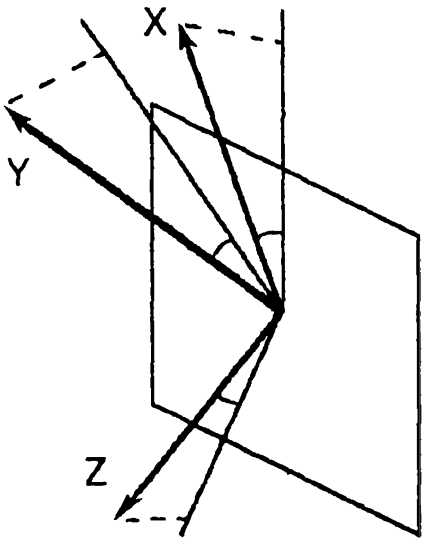


Рис. 9.19

Каждый из трех видов указанных проекций получается комбинацией поворотов, за которой следует параллельное проектирование. При повороте на угол ψ относительно оси ординат, на угол φ вокруг оси абсцисс и последующего проектирования вдоль оси аппликат возникает матрица

$$[M]=\begin{bmatrix}\cos\psi & \sin\varphi\sin\psi & 0 & 0\\ 0 & \cos\psi & 0 & 0\\ \sin\psi & -\sin\psi\cos\psi & 0 & 0\\ 0 & 0 & 0 & 1\end{bmatrix}=$$

$$=\begin{bmatrix}\cos\psi & 0 & -\sin\psi & 0\\ 0 & 1 & 0 & 0\\ \sin\psi & 0 & \cos\psi & 0\\ 0 & 0 & 0 & 1\end{bmatrix}\begin{bmatrix}1 & 0 & 0 & 0\\ 0 & \cos\varphi & \sin\varphi & 0\\ 0 & -\sin\varphi & \cos\varphi & 0\\ 0 & 0 & 0 & 1\end{bmatrix}\begin{bmatrix}1 & 0 & 0 & 0\\ 0 & 1 & 0 & 0\\ 0 & 0 & 0 & 0\\ 0 & 0 & 0 & 1\end{bmatrix}.$$

Покажем, как при этом преобразуются единичные орты координатных осей Z, Y, Z:

$$(1\quad 0\quad 0\quad 1)[M]=(\cos\psi\quad \sin\varphi\sin\psi\quad 0\quad 1),$$

$$(0\quad 1\quad 0\quad 1)[M]=(0\quad \cos\varphi\quad 0\quad 1),$$

$$(0\quad 0\quad 1\quad 1)[M]=(\sin\psi\quad -\sin\varphi\cos\psi\quad 0\quad 1).$$

Диметрия характеризуется тем, что длины двух проекций совпадают:

$$\cos^2\psi+\sin^2\varphi\sin^2\psi=\cos^2\varphi.$$

Отсюда следует, что

$$\sin^2\psi=\tan^2\varphi.$$

В случае изометрии имеем

$$\cos^2\psi + \sin^2\varphi \sin^2\psi = \cos^2\varphi,$$

$$\sin^2\psi + \sin^2\varphi \cos^2\psi = \cos^2\varphi.$$

Из последних двух соотношений вытекает, что $\sin^2\varphi = \frac{1}{3}$, $\sin^2\psi = \frac{1}{2}$

При триметрии длины проекций попарно различны.

Проекции, для получения которых используется пучок прямых, не перпендикулярных плоскости экрана, принято называть косоугольными.

При косоугольном проектировании орта оси Z на плоскость XY (рис. 9.20) имеем:
$(0 \quad 0 \quad 1 \quad 1) \to (\alpha \quad \beta \quad 0 \quad 1)$.

Матрица соответствующего преобразования имеет следующий вид:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ \alpha & \beta & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

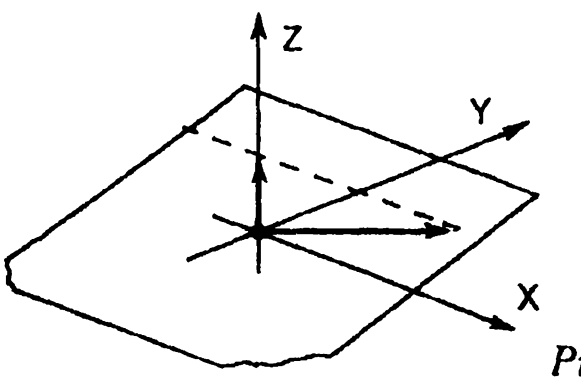


Рис. 9.20

Выделяют два вида косоугольных проекций: свободную проекцию (угол наклона проектирующих прямых к плоскости экрана равен половине прямого) и кабинетную проекцию (частный случай свободной проекции - масштаб по третьей оси вдвое меньше).

В случае свободной проекции $\alpha = \beta = \cos\frac{\pi}{4}$,

в случае кабинетной - $\alpha = \beta = \frac{1}{2}\cos\frac{\pi}{4}$.

Перспективные (центральные) проекции строятся более сложно.

Предположим для простоты, что центр проектирования лежит на оси Z в точке $C(0, 0, c)$ и плоскость проектирования совпадает с координатной плоскостью XY (рис. 9.21). Возьмем в пространстве произвольную точку $M(x, y, z)$, проведем через нее и точку C прямую и запишем соответствующие параметрические уравнения.

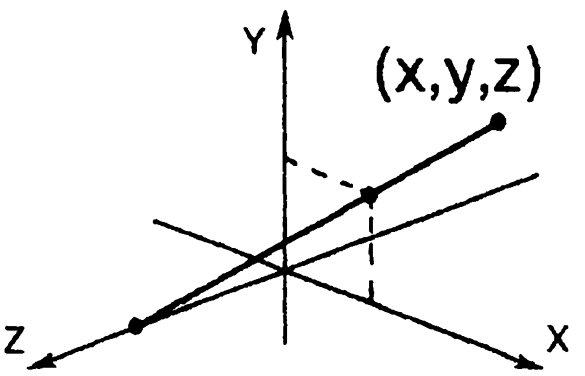


Рис. 9.21

Имеем:

$$X^* = xt, Y^* = yt, Z^* = c + (z - c)t.$$

Найдем координаты точки пересечения построенной прямой с плоскостью XY. Из условия $Z^* = 0$ получаем, что

$$t^* = \frac{1}{1 - \frac{z}{c}},$$

и далее $X^* = \frac{1}{1 - \frac{z}{c}} x,\ Y^* = \frac{1}{1 - \frac{z}{c}} y.$

Интересно заметить, что тот же самый результат можно получить, привлекая матрицу

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & -1/c \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

В самом деле, переходя к однородным координатам, прямым вычислением совсем легко проверить, что

$$\begin{pmatrix} x & y & z & 1 \end{pmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & -1/c \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{pmatrix} x & y & 0 & 1 - \frac{z}{c} \end{pmatrix}.$$

Вспоминая свойства однородных координат, запишем полученный результат в несколько ином виде:

$$\begin{pmatrix} \frac{x}{1 - \frac{z}{c}} & \frac{y}{1 - \frac{z}{c}} & 0 & 1 \end{pmatrix}$$

и затем путем непосредственного сравнения убедимся в том, что это координаты той же самой точки.

Замечание. Матрица проектирования, разумеется, вырожденна.

Матрица соответствующего перспективного преобразования (без проектирования) имеет следующий вид:

$$[Q] = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & -\frac{1}{c} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Обратим внимание на то, что последняя матрица невырожденна.

Рассмотрим пучок прямых, параллельных оси Z, и попробуем разобраться в том, что с ним происходит под действием матрицы $[Q]$.

Каждая прямая пучка однозначно определяется точкой (скажем, $M(x, y, z)$) своего пересечения с плоскостью XY и описывается уравнениями $X = x,\ Y = y,\ Z = t$.

Переходя к однородным координатам и используя матрицу $[Q]$, получаем

$$\left(x \quad y \quad t \quad 1\right)[Q]=\left(x \quad y \quad t \quad 1-\frac{t}{c}\right), \text{ или } \left(\frac{x}{1-\frac{t}{c}} \quad \frac{y}{1-\frac{t}{c}} \quad -c\frac{t}{t-c} \quad 1\right).$$

Устремим t в бесконечность.

При переходе к пределу точка (x, y, t, 1) преобразуется в (0, 0, 1, 0). Чтобы убедиться в этом, достаточно разделить каждую координату на t:

$$\left(\frac{x}{t} \quad \frac{y}{t} \quad 1 \quad \frac{1}{t}\right).$$

Точка (0, 0, -c, 1) является пределом (при t, стремящемся к бесконечности) правой части

$$\left(\frac{x}{1-\frac{t}{c}} \quad \frac{y}{1-\frac{t}{c}} \quad -c\frac{t}{t-c} \quad 1\right)$$

рассматриваемого равенства.

Тем самым бесконечно удаленный (несобственный) центр (0, 0, 1, 0) пучка прямых, параллельных оси Z, переходит в точку (0, 0, -c, 1) оси Z.

Вообще каждый несобственный пучок прямых (совокупность прямых, параллельных заданному направлению), не параллельный картинной плоскости,

$$X = x + lt, Y = y + mt, Z = z + nt, n \neq 0,$$

под действием преобразования, задаваемого матрицей [Q], переходит в собственный пучок

$$\left(x+lt \quad y+nt \quad nt \quad 1\right)[Q]=\left(x+lt \quad y+mt \quad nt \quad 1-\frac{nt}{c}\right)$$

Центр этого пучка

$$\left(-\frac{lc}{n} \quad -\frac{mc}{n} \quad -c \quad 1\right)$$

называют точкой схода.

Принято выделять так называемые главные точки схода, которые соответствуют пучкам прямых, параллельных координатным осям.

Для преобразования с матрицей [Q] существует лишь одна главная точка схода (рис. 9.22). В общем случае (когда оси координатной системы не параллельны плоскости экрана) таких точек три.

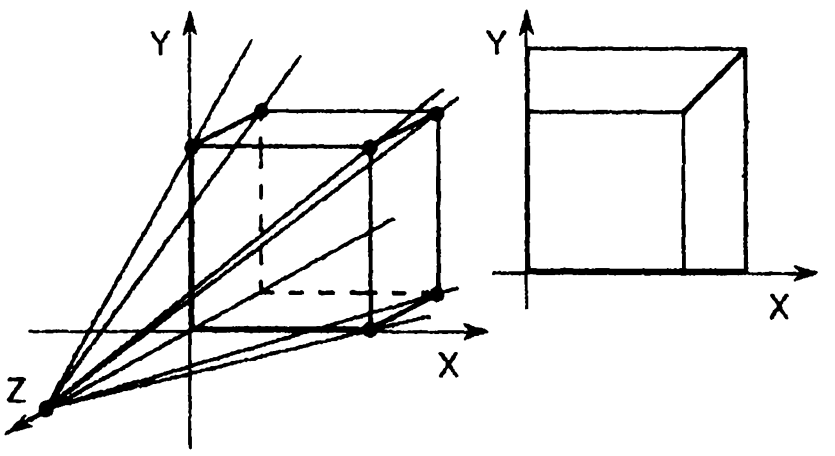


Рис. 9.22

Матрица соответствующего преобразования выглядит следующим образом:

$$\begin{bmatrix} 1 & 0 & 0 & -1/a \\ 0 & 1 & 0 & -1/b \\ 0 & 0 & 1 & -1/c \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

Пучок прямых, параллельных оси

0X	*0Y*
(1 0 0 0)	(0 1 0 0)

переходит в пучок прямых с центром.

На рис. 9.23 изображены проекции куба со сторонами, параллельными координатным осям, с одной и с двумя главными точками схода

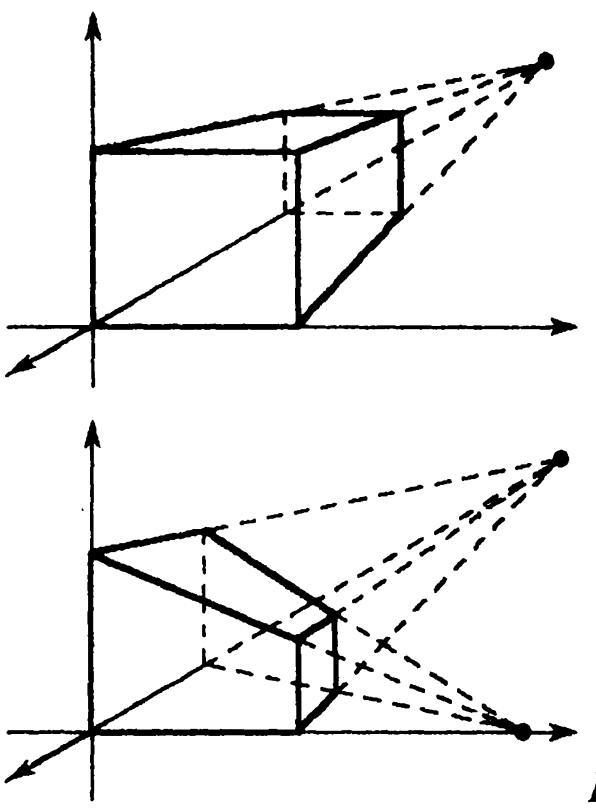


Рис. 9.23

$\left(1 \quad 0 \quad 0 \quad -\frac{1}{a}\right)\left(0 \quad 1 \quad 0 \quad -\frac{1}{b}\right)$, или (-*a* 0 0 1) (-*b* 0 0 1)

Точки (-*a*, 0, 0) и (0, -*b*, 0) суть главные точки схода.

По аналогии с двумерными объектами здесь также вводятся классы для работы с трехмерными векторами и матрицами преобразований. При этом поскольку в ряде случаев перспективное проектирование проводится отдельно и использование матриц 4x4 снижает общее быстродействие, то здесь вводится класс Vector3D для представления трехмерных векторов и два класса для работы с матрицами - класс Matrix3D, реализующий основные аффинные операции над векторами без использования однородных координат, и класс Matrix, служащий для работы с однородными координатами. Вводятся функции, возвращающие матрицы для ряда стандартных преобразований в пространстве.

```
// File vector3d.h
#ifndef         __VECTOR3D__
#define         __VECTOR3D__

#include        <math.h>

class Vector3D
{
public:
        float       x, y, z;
```

```
Vector3D () {}
Vector3D ( float px, float py, float pz )
{
        x = px;
        y = py;
        z = pz;
}
Vector3D ( const Vector3D& v )
{
        x = v.x;
        y = v.y;
        z = v.z;
}
Vector3D& operator = ( const Vector3D& v )
{
        x = v.x;
        y = v.y;
        z = v.z;
        return *this;
}
Vector3D operator + () const
{
        return *this;
}
Vector3D operator - () const
{
        return Vector3D ( -x, -y, -z );
}
Vector3D& operator += ( const Vector3D& v )
{
        x += v.x;
        y += v.y;
        z += v.z;
        return *this;
}
Vector3D& operator -= ( const Vector3D& v )
{
        x -= v.x;
        y -= v.y;
        z -= v.z;
        return *this;
}
Vector3D& operator *= ( const Vector3D& v )
{
        x *= v.x;
        y *= v.y;
        z *= v.z;
```

```
        return *this;
}
Vector3D& operator *= ( float f )
{
        x *= f;
        y *= f;
        z *= f;

        return *this;
}
Vector3D& operator /= ( const Vector3D& v )
{
        x /= v.x;
        y /= v.y;
        z /= v.z;

        return *this;
}
Vector3D& operator /= ( float f )
{
        x /= f;
        y /= f;
        z /= f;

        return *this;
}
float& operator [] ( int index )
{
        return * ( index + &x );
}
int     operator == ( const Vector3D& v ) const
{
        return x == v.x && y == v.y && z == v.z;
}
int     operator != ( const Vector3D& v ) const
{
        return x != v.x || y != v.y || z != v.z;
}
int     operator < ( const Vector3D& v ) const
{
        return (x < v.x) || ( (x == v.x) && (y < v.y));
}
int     operator > ( const Vector3D& v ) const
{
        return (x > v.x) || ( (x == v.x) && (y > v.y));
}
float   length () const
{
        return (float) sqrt ( x * x + y * y + z * z );
}
```

```
        friend Vector3D operator + (const Vector3D&,const Vector3D&);
        friend Vector3D operator - (const Vector3D&,const Vector3D&);
        friend Vector3D operator * (const Vector3D&,const Vector3D&);
        friend Vector3D operator * (float,                const Vector3D&);
        friend Vector3D operator * (const Vector3D&,float);
        friend Vector3D operator / (const Vector3D&,float);
        friend Vector3D operator / (const Vector3D&,const Vector3D&);
        friend float    operator & (const Vector3D&,const Vector3D&);
        friend Vector3D operator ^ (const Vector3D&,const Vector3D&);
};
inline Vector3D operator + ( const Vector3D& u, const Vector3D& v )
{
        return Vector3D ( u.x + v.x, u.y + v.y, u.z + v.z );
}
inline Vector3D operator - ( const Vector3D& u, const Vector3D& v )
{
        return Vector3D ( u.x - v.x, u.y - v.y, u.z - v.z );
}
inline Vector3D operator * ( const Vector3D& u, const Vector3D& v )
{
        return Vector3D ( u.x*v.x, u.y*v.y, u.z * v.z );
}
inline Vector3D operator * ( const Vector3D& v, float a )
{
        return Vector3D ( v.x*a, v.y*a, v.z*a );
}
inline Vector3D operator * ( float a, const Vector3D& v )
{
        return Vector3D ( v.x*a, v.y*a, v.z*a );
}
inline Vector3D operator / ( const Vector3D& u, const Vector3D& v )
{
        return Vector3D ( u.x/v.x, u.y/v.y, u.z/v.z );
}
inline Vector3D operator / ( const Vector3D& v, float a )
{
        return Vector3D ( v.x/a, v.y/a, v.z/a );
}
inline float operator & ( const Vector3D& u, const Vector3D& v )
{
        return u.x*v.x + u.y*v.y + u.z*v.z;
}
inline Vector3D operator ^ ( const Vector3D& u, const Vector3D& v )
{
        return Vector3D (u.y*v.z-u.z*v.y, u.z*v.x-u.x*v.z,
                                u.x*v.y-u.y*v.x);
}
#endif
```

```
// File matrix3D.h
#ifndef __MATRIX3D__
#define __MATRIX3D__

#include          "Vector3D.h"

class Matrix3D
{
public:
        float x [3][3];

        Matrix3D () {}
        Matrix3D ( float );
        Matrix3D ( const Matrix3D& );

        Matrix3D& operator =  ( const Matrix3D& );
        Matrix3D& operator =  ( float );
        Matrix3D& operator += ( const Matrix3D& );
        Matrix3D& operator -= ( const Matrix3D& );
        Matrix3D& operator *= ( const Matrix3D& );
        Matrix3D& operator *= ( float );
        Matrix3D& operator /= ( float );

        float * operator [] ( int i )
        {
                return & x[i][0];
        }
        void invert ();
        void transpose ();

        static Matrix3D scale   ( const Vector3D& );
        static Matrix3D rotateX ( float );
        static Matrix3D rotateY ( float );
        static Matrix3D rotateZ ( float );
        static Matrix3D rotate  ( const Vector3D&, float );
        static Matrix3D mirrorX ();
        static Matrix3D mirrorY ();
        static Matrix3D mirrorZ ();

        friend Matrix3D operator + (const Matrix3D&,const Matrix3D&);
        friend Matrix3D operator - (const Matrix3D&,const Matrix3D&);
        friend Matrix3D operator * (const Matrix3D&,const Matrix3D&);
        friend Matrix3D operator * (const Matrix3D&, float);
        friend Matrix3D operator * (float, const Matrix3D&);
        friend Vector3D operator * (const Matrix3D&,const Vector3D);
};

Matrix3D scale   ( const Vector3D& );
Matrix3D rotateX ( float );
Matrix3D rotateY ( float );
Matrix3D rotateZ ( float );
Matrix3D rotate  ( const Vector3D&, float );
Matrix3D mirrorX ();
Matrix3D mirrorY ();
Matrix3D mirrorZ ();
#endif
```

```
// File matrix3D.cpp
#include <math.h>
#include "matrix3D.h"

Matrix3D    Matrix3D ( float a )
{
        x [0][1] = x [0][2] = x [1][0] =
        x [1][2] = x [2][0] = x [2][1] = 0.0;
        x [0][0] = x [1][1] = x [2][2] = a;
}

Matrix3D    Matrix3D ( const Matrix3D& a )
{
        x [0][0] = a.x [0][0];
        x [0][1] = a.x [0][1];
        x [0][2] = a.x [0][2];
        x [1][0] = a.x [1][0];
        x [1][1] = a.x [1][1];
        x [1][2] = a.x [1][2];
        x [2][0] = a.x [2][0];
        x [2][1] = a.x [2][1];
        x [2][2] = a.x [2][2];
}

Matrix3D& Matrix3D    operator = ( const Matrix3D& a )
{
        x [0][0] = a.x [0][0];
        x [0][1] = a.x [0][1];
        x [0][2] = a.x [0][2];
        x [1][0] = a.x [1][0];
        x [1][1] = a.x [1][1];
        x [1][2] = a.x [1][2];
        x [2][0] = a.x [2][0];
        x [2][1] = a.x [2][1];
        x [2][2] = a.x [2][2];

        return *this;
}

Matrix3D& Matrix3D :: operator = ( float a )
{
        x [0][1] = x [0][2] = x [1][0] =
        x [1][2] = x [2][0] = x [2][1] = 0.0;
        x [0][0] = x [1][1] = x [2][2] = a;

        return *this;
}

Matrix3D& Matrix3D    operator += ( const Matrix3D& a )
{
        x [0][0] += a.x [0][0];
        x [0][1] += a.x [0][1];
        x [0][2] += a.x [0][2];
        x [1][0] += a.x [1][0];
        x [1][1] += a.x [1][1];
        x [1][2] += a.x [1][2];
```

```
        x [2][0] += a.x [2][0];
        x [2][1] += a.x [2][1];
        x [2][2] += a.x [2][2];

        return *this;
}
Matrix3D& Matrix3D    operator -= ( const Matrix3D& a )
{
        x [0][0] -=a.x [0][0];
        x [0][1] -=a.x [0][1];
        x [0][2] -=a.x [0][2];
        x [1][0] -=a.x [1][0];
        x [1][1] -=a.x [1][1];
        x [1][2] -=a.x [1][2];
        x [2][0] -=a.x [2][0];
        x [2][1] -=a.x [2][1];
        x [2][2] -=a.x [2][2];

        return *this;
}
Matrix3D& Matrix3D    operator *= ( const Matrix3D& a )
{
        Matrix3D c ( *this );

        x[0][0]=c.x[0][0]*a.x[0][0]+c.x[0][1]*a.x[1][0]+
                        c.x[0][2]*a.x[2][0];
        x[0][1]=c.x[0][0]*a.x[0][1]+c.x[0][1]*a.x[1][1]+
                        c.x[0][2]*a.x[2][1];
        x[0][2]=c.x[0][0]*a.x[0][2]+c.x[0][1]*a.x[1][2]+
                        c.x[0][2]*a.x[2][2];
        x[1][0]=c.x[1][0]*a.x[0][0]+c.x[1][1]*a.x[1][0]+
                        c.x[1][2]*a.x[2][0];
        x[1][1]=c.x[1][0]*a.x[0][1]+c.x[1][1]*a.x[1][1]+
                        c.x[1][2]*a.x[2][1];
        x[1][2]=c.x[1][0]*a.x[0][2]+c.x[1][1]*a.x[1][2]+
                        c.x[1][2]*a.x[2][2];
        x[2][0]=c.x[2][0]*a.x[0][0]+c.x[2][1]*a.x[1][0]+
                        c.x[2][2]*a.x[2][0];
        x[2][1]=c.x[2][0]*a.x[0][1]+c.x[2][1]*a.x[1][1]+
                        c.x[2][2]*a.x[2][1];
        x[2][2]=c.x[2][0]*a.x[0][2]+c.x[2][1]*a.x[1][2]+
                        c.x[2][2]*a.x[2][2];

        return *this;
}
Matrix3D& Matrix3D    operator *= ( float a )
{
        x [0][0] *= a;
        x [0][1] *= a;
        x [0][2] *= a;
        x [1][0] *= a;
        x [1][1] *= a;
        x [1][2] *= a;
```

```
        x [2][0] *= a;
        x [2][1] *= a;
        x [2][2] *= a;

        return *this;
}
Matrix3D& Matrix3D :: operator /= ( float a )
{
        x [0][0] /= a;
        x [0][1] /= a;
        x [0][2] /= a;
        x [1][0] /= a;
        x [1][1] /= a;
        x [1][2] /= a;
        x [2][0] /= a;
        x [2][1] /= a;
        x [2][2] /= a;

        return *this;
};
void Matrix3D    invert ()
{
        float          det;
        Matrix3D a;
                                                        // compute a determinant
        det = x [0][0]*(x [1][1]*x [2][2]-x [1][2]*x [2][1]) -
              x [0][1]*(x [1][0]*x [2][2]-x [1][2]*x [2][0]) +
              x [0][2]*(x [1][0]*x [2][1]-x [1][1]*x [2][0]);

        a.x [0][0] = (x [1][1]*x [2][2]-x [1][2]*x [2][1]) / det;
        a.x [0][1] = (x [0][2]*x [2][1]-x [0][1]*x [2][2]) / det;
        a.x [0][2] = (x [0][1]*x [1][2]-x [0][2]*x [1][1]) / det;
        a.x [1][0] = (x [1][2]*x [2][0]-x [1][0]*x [2][2]) / det;
        a.x [1][1] = (x [0][0]*x [2][2]-x [0][2]*x [2][0]) / det;
        a.x [1][2] = (x [0][2]*x [1][0]-x [0][0]*x [1][2]) / det;
        a.x [2][0] = (x [1][0]*x [2][1]-x [1][1]*x [2][0]) / det;
        a.x [2][1] = (x [0][1]*x [2][0]-x [0][0]*x [2][1]) / det;
        a.x [2][2] = (x [0][0]*x [1][1]-x [0][1]*x [1][0]) / det;

        *this = a;
}
void Matrix3D :: transpose ()
{
        Matrix3D a;

        a.x [0][0] = x [0][0];
        a.x [0][1] = x [1][0];
        a.x [0][2] = x [2][0];
        a.x [1][0] = x [0][1];
        a.x [1][1] = x [1][1];
        a.x [1][2] = x [2][1];
        a.x [2][0] = x [0][2];
        a.x [2][1] = x [1][2];
```

```
        a.x [2][2] = x [2][2];
        *this = a;
}
Matrix3D operator + ( const Matrix3D& a, const Matrix3D& b )
{
        Matrix3D c;
        c.x [0][0] = a.x [0][0] + b.x [0][0];
        c.x [0][1] = a.x [0][1] + b.x [0][1];
        c.x [0][2] = a.x [0][2] + b.x [0][2];
        c.x [1][0] = a.x [1][0] + b.x [1][0];
        c.x [1][1] = a.x [1][1] + b.x [1][1];
        c.x [1][2] = a.x [1][2] + b.x [1][2];
        c.x [2][0] = a.x [2][0] + b.x [2][0];
        c.x [2][1] = a.x [2][1] + b.x [2][1];
        c.x [2][2] = a.x [2][2] + b.x [2][2];
        return c;
}
Matrix3D operator - ( const Matrix3D& a, const Matrix3D& b )
{
        Matrix3D c;
        c.x [0][0] = a.x [0][0] - b.x [0][0];
        c.x [0][1] = a.x [0][1] - b.x [0][1];
        c.x [0][2] = a.x [0][2] - b.x [0][2];
        c.x [1][0] = a.x [1][0] - b.x [1][0];
        c.x [1][1] = a.x [1][1] - b.x [1][1];
        c.x [1][2] = a.x [1][2] - b.x [1][2];
        c.x [2][0] = a.x [2][0] - b.x [2][0];
        c.x [2][1] = a.x [2][1] - b.x [2][1];
        c.x [2][2] = a.x [2][2] - b.x [2][2];
        return c;
}
Matrix3D operator * ( const Matrix3D& a, const Matrix3D& b )
{
        Matrix3D c ( a );
        c.x[0][0]=a.x[0][0]*b.x[0][0]+a.x[0][1]*b.x[1][0]+
                  a.x[0][2]*b.x[2][0];
        c.x[0][1]=a.x[0][0]*b.x[0][1]+a.x[0][1]*b.x[1][1]+
                 a.x[0][2]*b.x[2][1];
        c.x[0][2]=a.x[0][0]*b.x[0][2]+a.x[0][1]*b.x[1][2]+
                  a.x[0][2]*b.x[2][2];
        c.x[1][0]=a.x[1][0]*b.x[0][0]+a.x[1][1]*b.x[1][0]+
                  a.x[1][2]*b.x[2][0];
        c.x[1][1]=a.x[1][0]*b.x[0][1]+a.x[1][1]*b.x[1][1]+
                  a.x[1][2]*b.x[2][1];
        c.x[1][2]=a.x[1][0]*b.x[0][2]+a.x[1][1]*b.x[1][2]+
                  a.x[1][2]*b.x[2][2];
        c.x[2][0]=a.x[2][0]*b.x[0][0]+a.x[2][1]*b.x[1][0]+
                  a.x[2][2]*b.x[2][0];
```

```
        c.x[2][1]=a.x[2][0]*b.x[0][1]+a.x[2][1]*b.x[1][1]+
                  a.x[2][2]*b.x[2][1];
        c.x[2][2]=a.x[2][0]*b.x[0][2]+a.x[2][1]*b.x[1][2]+
                  a.x[2][2]*b.x[2][2];

        return c;
}
Matrix3D operator * ( const Matrix3D& a, float b )
{
        Matrix3D c;

        c.x [0][0] = a.x [0][0] * b;
        c.x [0][1] = a.x [0][1] * b;
        c.x [0][2] = a.x [0][2] * b;
        c.x [1][0] = a.x [1][0] * b;
        c.x [1][1] = a.x [1][1] * b;
        c.x [1][2] = a.x [1][2] * b;
        c.x [2][0] = a.x [2][0] * b;
        c.x [2][1] = a.x [2][1] * b;
        c.x [2][0] = a.x [0][0] * b;

        return c;
}
Matrix3D operator * ( float b, const Matrix3D& a )
{
        Matrix3D c;
        c.x [0][0] = a.x [0][0] * b;
        c.x [0][1] = a.x [0][1] * b;
        c.x [0][2] = a.x [0][2] * b;
        c.x [1][0] = a.x [1][0] * b;
        c.x [1][1] = a.x [1][1] * b;
        c.x [1][2] = a.x [1][2] * b;
        c.x [2][0] = a.x [2][0] * b;
        c.x [2][1] = a.x [2][1] * b;
        c.x [2][0] = a.x [0][0] * b;

        return c;
}
Vector3D operator *= ( const Matrix3D& a, const Vector3D& b )
{
        Vector3D v;

        v.x = a.x [0][0]*b.x + a.x [0][1]*b.y + a.x [0][2]*b.z;
        v.y = a.x [1][0]*b.x + a.x [1][1]*b.y + a.x [1][2]*b.z;
        v.z = a.x [2][0]*b.x + a.x [2][1]*b.y + a.x [2][2]*b.z;

        return v;
}
////////////////////////////////////////////////////////////////
Matrix3D scale ( const Vector3D& v )
{
        Matrix3D a ( 1.0 );
```

```
        a.x [0][0] = v.x;
        a.x [1][1] = v.y;
        a.x [2][2] = v.z;

        return a;
}
Matrix3D rotateX ( float angle )
{
        Matrix3D a(1.0);
        float       cosine = cos ( angle );
        float       sine   = sin ( angle );

        a.x [1][1] = cosine;
        a.x [1][2] = sine;
        a.x [2][1] = -sine;
        a.x [2][2] = cosine;

        return a;
}
Matrix3D rotateY ( float angle )
{
        Matrix3D a ( 1.0 );
        float       cosine = cos ( angle );
        float       sine   = sin ( angle );

        a.x [0][0] = cosine;
        a.x [0][2] = sine;
        a.x [2][0] = -sine;
        a.x [2][2] = cosine;

        return a;
}
Matrix3D rotateZ ( float angle )
{
        Matrix3D a ( 1.0 );
        float       cosine = cos ( angle );
        float       sine   = sin ( angle );

        a.x [0][0] = cosine;
        a.x [0][1] = sine;
        a.x [1][0] = -sine;
        a.x [1][1] = cosine;

        return a;
}
Matrix3D rotate ( const Vector3D& v, float angle )
{
        Matrix3D a;
        float       cosine = cos ( angle );
        float       sine   = sin ( angle );

        a.x [0][0] = v.x *v.x + (1-v.x*v.x) * cosine;
        a.x [0][1] = v.x *v.y * (1-cosine) + v.z * sine;
        a.x [0][2] = v.x *v.z * (1-cosine) - v.y * sine;
        a.x [1][0] = v.x *v.y * (1-cosine) - v.z * sine;
```

```
        a.x [1][1] = v.y *v.y + (1-v.y*v.y) * cosine;
        a.x [1][2] = v.y *v.z * (1-cosine) + v.x * sine;
        a.x [2][0] = v.x *v.z * (1-cosine) + v.y * sine;
        a.x [2][1] = v.y *v.z * (1-cosine) - v.x * sine;
        a.x [2][2] = v.z *v.z + (1-v.z*v.z) * cosine;

        return a;
}
Matrix3D mirrorX ()
{
        Matrix3D a ( 1.0 );

        a.x [0][0] = -1.0;

        return a;
}
Matrix3D mirrorY ()
{
        Matrix3D a ( 1.0 );

        a.x [1][1] = -1.0;

        return a;
}
Matrix3D mirrorZ ()
{
        Matrix3D a ( 1.0 );
        a.x [2][2] = -1.0;

        return a;
}
```



```
// File matrix.h
#ifndef __MATRIX__
#define __MATRIX__

#include        <mem.h>
#include        "Vector3D.h"

class Matrix
{
public:
        float x [4][4];

        Matrix () {}
        Matrix ( float );
        Matrix ( const Matrix& m )
        {
           memcpy ( & x [0][0], &m.x [0][0], 16*sizeof ( float ) );
        }

        Matrix& operator += ( const Matrix& );
        Matrix& operator -= ( const Matrix& );
        Matrix& operator *= ( const Matrix& );
        Matrix& operator *= ( float );
        Matrix& operator /= ( float );
```

```
        float * operator [] ( int i )
        {
                return & x[i][0];
        }
        void    invert ();
        void    transpose ();
        friend  Matrix  operator + (const Matrix&, const Matrix&);
        friend  Matrix  operator - (const Matrix&, const Matrix&);
        friend  Matrix  operator * (const Matrix&, float);
        friend Matrix  operator * (float,       const Matrix&);
        friend  Matrix  operator * (const Matrix&, const Matrix&);
        friend  Vector3D operator * (const Matrix&, const Vector3D&);
};
Matrix          translate ( const Vector3D& );
Matrix          scale   ( const Vector3D& );
Matrix          rotateX  ( float );
Matrix          rotateY  ( float );
Matrix          rotateZ  ( float );
Matrix          rotate   ( const Vector3D& v, float );
Matrix          mirrorX  ();
Matrix          mirrorY  ();
Matrix          mirrorZ  ();
#endif
```

// File matrix.cpp

```
#include <math.h>
#include "Matrix.h"
Matrix :: Matrix ( float v )
{
        for ( int i = 0; i < 4; i++)
                for ( int j = 0; j < 4; j++)
                        x [i][j] = (i == j) ? v : 0.0;
        x [3][3] = 1;
}

void    Matrix :: invert ()
{
        Matrix      out ( 1 );
        for ( int i = 0; i < 4; i++ )
        {
            float    d = x [i][i];
            if ( d != 1.0)
            {
                    for ( int j = 0; j < 4; j++ )
                    {
                            out.x [i][j] /= d;
                            x [i][j]     /= d;
                    }
            }
```

```
            for ( int j = 0; j < 4; j++ )
            {
                    if ( j != i )
                    {
                       if ( x [j][i] != 0.0)
                       {
                                float       mulBy = x[j][i];
                                for ( int k = 0; k < 4, k++ )
                                {
                                        x [j][k]    -= mulBy * x [i][k];
                                    out.x [j][k] -= mulBy * out.x [i][k];
                                }
                       }
                    }
            }
        *this = out;
}
void   Matrix :: transpose ()
{
        float       t;
        for ( register int i = 0; i < 4; i++ )
                    for ( register int j = i; j < 4; j++ )
                                if ( i != j )
                                {
                                        t       = x [i][j];
                                        x [i][j] = x [j][i];
                                        x [j][i] = t;
                                }
}
Matrix&             Matrix    operator += ( const Matrix& a )
{
        for ( register int i = 0; i < 4; i++ )
                    for ( register int j = 0; j < 4; j++ )
                                x [i][j] += a.x [i][j];
        return *this;
}
Matrix&             Matrix    operator -= ( const Matrix& a )
{
        for ( register int i = 0; i < 4; i++ )
                    for ( register int j = 0; j < 4; j++ )
                                x [i][j] -= a.x [i][j];
        return *this;
}
Matrix&             Matrix    operator *= ( float v )
{
        for ( register int i = 0; i < 4; i++ )
                    for ( register int j = 0; j < 4; j++ )
```

```
                        x [i][j] *= v;
        return *this;
}
Matrix&         Matrix   operator *= ( const Matrix& a )
{
        Matrix     res ( *this );
        for ( int i = 0; i < 4; i++ )
                for ( int j = 0; j < 4; j++ )
                {
                        float sum = 0;
                        for ( int k = 0; k < 4; k++ )
                                sum += res.x [i][k] * a.x [k][j];
                        x [i][j] = sum;
                }
   return *this;
}
Matrix          operator + ( const Matrix& a, const Matrix& b )
{
        Matrix     res;
        for ( register int i = 0; i < 4; i++ )
                for ( register int j = 0; j < 4; j++ )
                        res.x [i][j] = a.x [i][j] + b.x [i][j];
        return res;
}
Matrix          operator - ( const Matrix& a, const Matrix& b )
{
        Matrix     res;
        for ( register int i = 0; i < 4; i++ )
                for ( register int j = 0; j < 4; j++ )
                        res.x [i][j] = a.x [i][j] - b.x [i][j];
        return res;
}
Matrix          operator * ( const Matrix& a, const Matrix& b )
{
        Matrix     res;
        for ( register int i = 0; i < 4; i++ )
                for ( register int j = 0; j < 4; j++ )
                {
                        float sum = 0;
                        for ( register int k = 0; k < 4; k++ )
                                sum += a.x [i][k] * b.x [k][j];
                        res.x [i][j] = sum;
                }
        return res;
```

```
}
Matrix          operator * ( const Matrix& a, float v )
{
        Matrix      res;

        for ( register int i = 0; i < 4; i++ )
                for ( register int j = 0; j < 4; j++ )
                        res.x [i][j] = a.x [i][j] * v;

        return res;
}
Matrix          operator * ( float v, const Matrix& a )
{
        Matrix      res;

        for ( register int i = 0; i < 4; i++ )
                for ( register int j = 0; j < 4; j++ )
                        res.x [i][j] = a.x [i][j] * v;

        return res;
}
Vector3D operator * ( const Matrix& m, const Vector3D& v )
{
        Vector3D res;

        res.x = m.x [0][0] * v.x + m.x [0][1] * v.y +
                m.x [0][2] * v.z + m.x [0][3];
        res.y = m.x [1][0] * v.x + m.x [1][1] * v.y +
                m.x [1][2] * v.z + m.x [1][3];
        res.z = m.x [2][0] * v.x + m.x [2][1] * v.y +
                m.x [2][2] * v.z + m.x [2][3];

        float       denom = m.x [3][0] * v.x + m.x [3][1] * v.y +
                m.x [3][2] * v.z + m.x [3][3];

        if ( denom != 1.0 )
                res /= denom;
        return res;
}
////////////////// Derived functions //////////////////
Matrix          translate ( const Vector3D& loc )
{
        Matrix      res ( 1 );

        res.x [0][3] = loc.x;
        res.x [1][3] = loc.y;
        res.x [2][3] = loc.z;

        return res;
}
Matrix          scale ( const Vector3D& v )
{
        Matrix      res ( 1 );
        res.x [0][0] = v.x;
        res.x [1][1] = v.y;
        res.x [2][2] = v.z;
```

```
        return res;
}
Matrix              rotateX ( float angle )
{
        Matrix res ( 1 );
        float  cosine = cos ( angle );
        float  sine   = sin ( angle );

        res.x [1][1] = cosine;
        res.x [1][2] = -sine;
        res.x [2][1] = sine;
        res.x [2][2] = cosine;

        return res;
}
Matrix              rotateY ( float angle )
{
        Matrix res ( 1 );
        float  cosine = cos ( angle );
        float  sine   = sin ( angle );

        res.x [0][0] = cosine;
        res.x [0][2] = -sine;
        res.x [2][0] = sine;
        res.x [2][2] = cosine;

        return res;
}
Matrix              rotateZ ( float angle )
{
        Matrix res ( 1 );
        float  cosine = cos ( angle );
        float  sine   = sin ( angle );

        res.x [0][0] = cosine;
        res.x [0][1] = -sine;
        res.x [1][0] = sine;
        res.x [1][1] = cosine;

        return res;
}
Matrix              rotation ( const Vector3D& axis, float angle )
{
        Matrix res ( 1 );
        float  cosine = cos ( angle );
        float  sine   = sin ( angle );

        res.x [0][0] = axis.x*axis.x+(1-axis.x*axis.x )*cosine;
        res.x [1][0] = axis.x*axis.y*(1-cosine)+axis.z*sine;
        res.x [2][0] = axis.x*axis.z*(1-cosine)-axis.y*sine;
        res.x [3][0] = 0;

        res.x [0][1] = axis.x*axis.y*(1-cosine)-axis.z*sine;
        res.x [1][1] = axis.y*axis.y+(1-axis.y*axis.y)*cosine;
        res.x [2][1] = axis.y*axis.z*(1-cosine)+axis.x*sine;
        res.x [3][1] = 0;
```

```
        res.x [0][2] = axis.x*axis.z*(1-cosine)+axis.y*sine;
        res.x [1][2] = axis.y*axis.z*(1-cosine)-axis.x*sine;
        res.x [2][2] = axis.z*axis.z+(1-axis.z*axis.z)*cosine;
        res.x [3][2] = 0;

        res.x [0][3] = 0;
        res.x [1][3] = 0;
        res.x [2][3] = 0;
        res.x [3][3] = 1;

        return res;
}
Matrix          mirrorX ()
{
        Matrix  res ( 1 );
        res.x [0][0] = -1;
        return res;
}
Matrix          mirrorY ()
{
        Matrix  res ( 1 );
        res.x [1][1] = -1;
        return res;
}
Matrix          mirrorZ ()
{
        Matrix  res ( 1 );
        res.x [2][2] = -1;
        return res;
}
```

9.3. Особенности проекций гладких отображений

В заключение этой главы мы остановимся на некоторых эффектах, возникающих при проектировании искривленных объектов (главным образом поверхностей) на картинную плоскость.

Важно отметить, что описываемые ниже эффекты возникают вне зависимости от того, является ли проектирование параллельным или центральным.

Будем считать для простоты, что проектирование проводится при помощи пучка параллельных прямых, идущих перпендикулярно картинной плоскости, а система координат X, Y, Z в пространстве выбрана так, что картинная плоскость совпадает с координатной плоскостью $X = 0$.

Укажем три принципиально различных случая.

1-й случай. Заданная поверхность - плоскость, описываемая уравнением $Z = X$ и проектируемая на плоскость $X = 0$ (рис. 9.24).

Записав ее уравнение в неявном виде $X \quad Z = 0$, вычислим координаты нормального вектора. Имеем

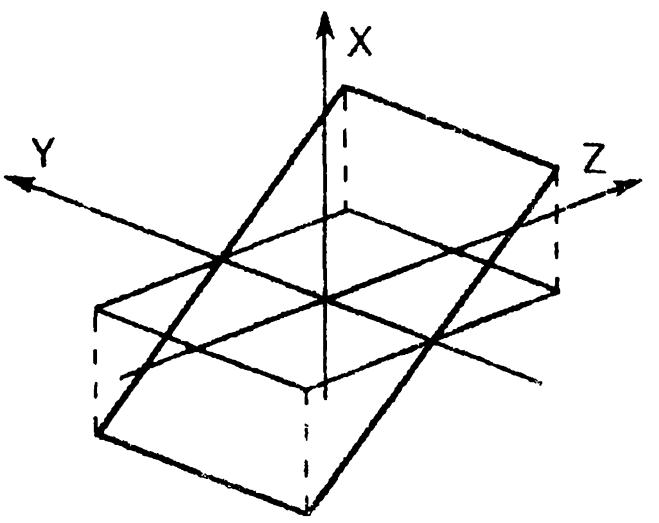
$$\vec{N} = (1, 0, -1).$$

Вектор $\vec{L}$, вдоль которого осуществляется проектирование, имеет координаты

$$\vec{L} = (1, 0, 0).$$

Легко видеть, что скалярное произведение этих двух векторов отлично от нуля:

$$\left(\vec{N}, \vec{L}\right) = 1 > 0.$$



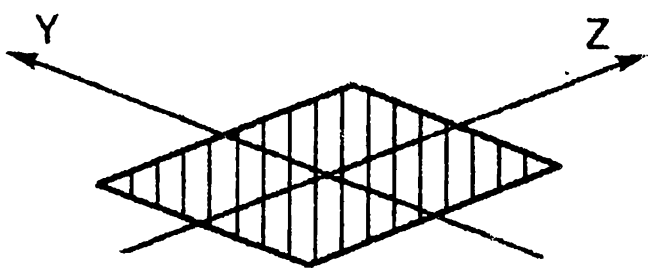


Рис. 9.24

Тем самым вектор проектирования и нормальный вектор рассматриваемой поверхности не перпендикулярны ни в одной точке.

Отметим, что полученная проекция особенностей не имеет.

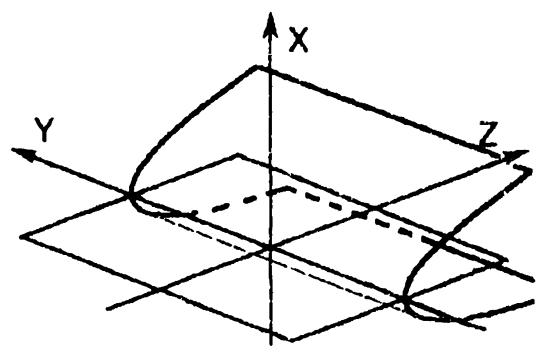
2-й случай. Заданная поверхность - параболический цилиндр с уравнением $Z = X^2$, или $X^2 - Z = 0$.

Нормальный вектор $\vec{N} = (2X, 0, -1)$ ортогонален вектору проектирования $\vec{L}$ в точках оси Y. Это вытекает из того, что $\left(\vec{N}, \vec{L}\right) = 2X$.

Здесь, в отличие от первого случая, точки плоскости $X = 0$ разбиваются на три класса:

- к первому относятся точки ($Z > 0$), у которых два прообраза (рис. 9.25 этот класс заштрихован);
- ко второму - те, у которых прообраз один ($Z = 0$);
- и наконец, к третьему классу относятся точки, у которых прообразов на цилиндре нет вовсе.

Прямая $X = 0$, $Z = 0$ является особой. Вдоль нее векторы $\vec{N}$ и $\vec{L}$ ортогональны.



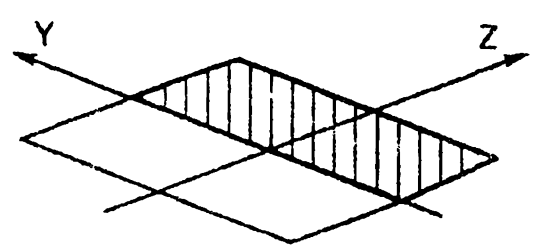


Рис. 9.25

Особенность этого типа называется складкой.

3-й случай. Рассмотрим поверхность, заданную уравнением $Z = X^3 + XY - Z = 0$.

Вычислим нормальный вектор этой поверхности

$$\vec{N} = \left(3X^2 + Y, X, -1\right)$$

и построим ее, применив метод сечений:

Пусть $Y = 1$.

Тогда $Z = X^3 + X$

(рис. 9.26).

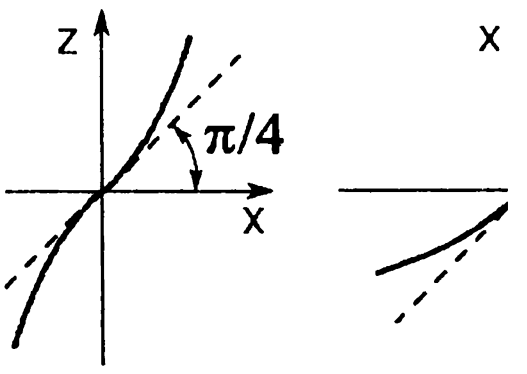


Рис. 9.26

При $Y = 0$ имеем $Z = X^3$

(рис. 9.27).

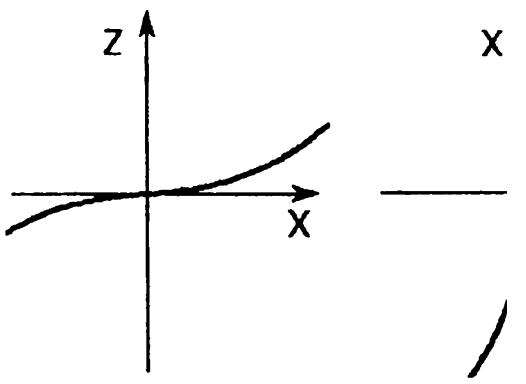


Рис. 9.27

Наконец, при $Y = -1$ получаем

$Z = X^3 - X$

(рис. 9.28).

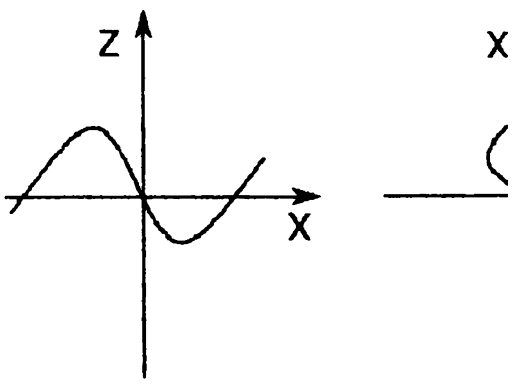


Рис. 9.28

Построенные сечения дают представление обо всей поверхности. Поэтому нарисовать ее теперь уже несложно (рис. 9.29).

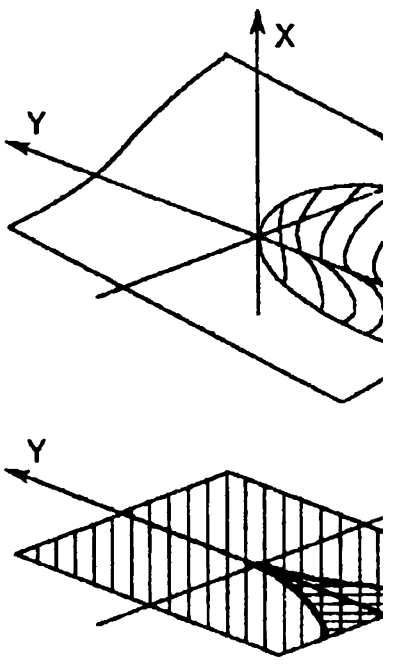


Рис. 9.29

Из условия

$$(\vec{N}, \vec{L}) = 3X^2 + Y = 0$$

и уравнения поверхности получаем, что вдоль лежащей на ней кривой с уравнениями $Y = -3X^2$, $Z = -2X^3$

вектор проектирования $\vec{L}$ и нормальный вектор $\vec{N}$ рассматриваемой поверхности ортогональны. Исключая X, получаем, что

$$\left(-Y/3\right)^3 = \left(-Z/2\right)^2,$$

или $27Z^2 = -4Y^3$

Последнее равенство задает на координатной плоскости $X = 0$ полукубическую

параболу (рис. 9.30), которая делит точки этой плоскости на три класса: к первому относятся точки, лежащие на острие (у каждой из них на заданной поверхности ровно два прообраза), внутри острия лежат точки второго класса (каждая точка имеет по три прообраза), а вне - точки третьего класса, имеющие по одному прообразу. Особенность этого типа называется сборкой.

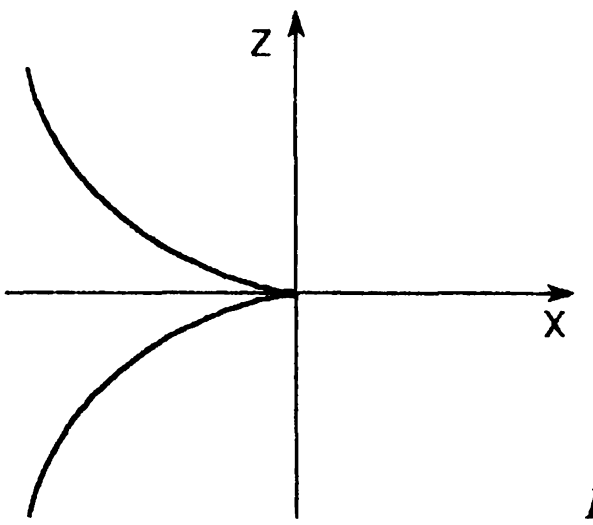


Рис. 9.30

Замечание. Возникающая в третьем случае полукубическая парабола имеет точку заострения. Однако ее прообраз

$X = X,\ Y = -3X^2,\ Z = -2X_3$

является регулярной кривой, лежащей на заданной поверхности.

В теории особенностей (теории катастроф) доказывается: при проектировании на плоскость произвольного гладкого объекта - поверхности возможны (с точностью до малого шевеления, рассыпающего более сложные проекции) только три указанных типа проекции - обыкновенная проекция, складка и сборка.

Сказанное следует понимать так: при проектировании гладких поверхностей на плоскость могут возникать и другие, более сложные особенности. Однако в отличие от трех перечисленных выше все они оказываются неустойчивыми - при малых изменениях либо направления проектирования, либо взаимного расположения плоскости и проектируемой поверхности эти особенности не сохраняются и переходят в более простые.

Замечание. По существу, в приведенных примерах рассмотрены три типа отображения 2-плоскости в 2-плоскость (рис. 9.31).

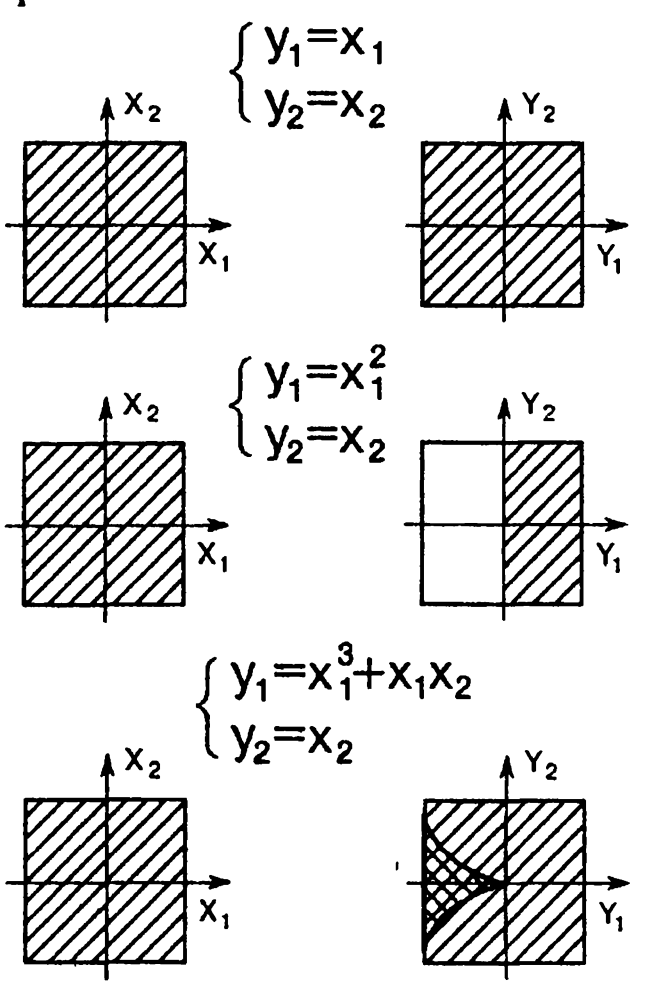


Рис. 9.31

Глава 10
УДАЛЕНИЕ НЕВИДИМЫХ ЛИНИЙ И ПОВЕРХНОСТЕЙ

Одной из важнейших задач трехмерной графики является следующая: определить, какие части объектов (ребра, грани), находящихся в трехмерном пространстве, будут видны при заданном способе проектирования, а какие будут закрыты от наблюдателя другими объектами. В качестве возможных видов проектирования традиционно рассматриваются параллельное и центральное (перспективное).

Само проектирование осуществляется на так называемую картинную плоскость (экран): через каждую точку каждого объекта проводится проектирующий луч (проектор) к картинной плоскости (рис. 10.1). Все проекторы образуют пучок либо параллельных лучей (при параллельном проектировании), либо лучей, выходящих из одной точки (центральное проектирование). Пересечение проектора с картинной плоскостью дает проекцию точки. Видимыми будут только те точки, которые вдоль направления проектирования расположены ближе всего к картинной плоскости. Все три точки P_1, P_2 и P_3 (рис. 10.2) лежат на одном и том же проекторе, т. е. проектируются в одну и ту же точку картинной плоскости.

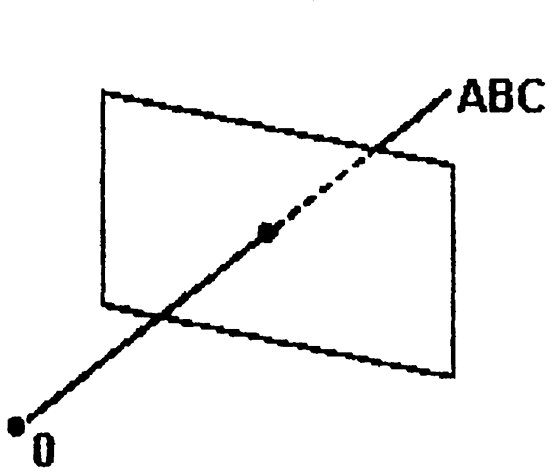


Рис. 10.1

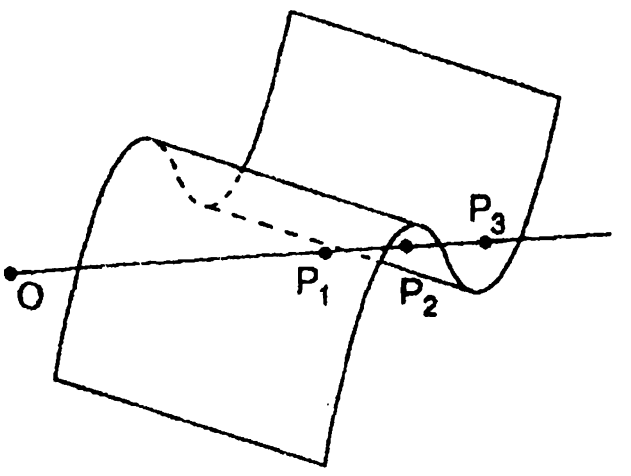


Рис. 10.2

Но так как точка P_1 лежит ближе к картинной плоскости, чем точки P_2 и P_3, и закрывает их при проектировании, то из этих трех точек именно она является видимой.

Несмотря на кажущуюся простоту, задача удаления невидимых линий и поверхностей является достаточно сложной и зачастую требует очень больших объемов вычислений. Поэтому существует целый ряд различных методов решения этой задачи, включая и методы, опирающиеся на аппаратные решения.

Эти методы различаются по следующим основным параметрам:

- способу представления объектов;
- способу визуализации сцены;
- пространству, в котором проводится анализ видимости;
- виду получаемого результата (его точности).

В качестве возможных способов представления объектов могут выступать аналитические (явные и неявные), параметрические и полигональные.

Далее будем считать, что все объекты представлены набором выпуклых плоских граней, например треугольников (полигональный способ), которые могут пересекаться одна с другой только вдоль ребер.

Координаты в исходном трехмерном пространстве будем обозначать через *(x, y, z)*, а координаты в картинной плоскости – через *(X, Y)*. Будем также считать, что на картинной плоскости задана целочисленная растровая решетка - множество точек (*i, j*), где *i* и *j* - целые числа.

Если это не оговорено особо, будем считать для простоты, что проектирование осуществляется на плоскость *Oxy*. Проектирование при этом происходит либо параллельно оси *Oz*, т. е. задается формулами

$X = x, \quad Y = y,$

либо является центральным с центром, расположенным на оси *Oz*, и задается формулами

$$X = \frac{x}{z}, \quad Y = \frac{y}{z}.$$

Существуют два различных способа изображения трехмерных тел - каркасное (wireframe - рисуются только ребра) и сплошное (рисуются закрашенные грани). Тем самым возникают два типа задач - удаление невидимых линий (ребер для каркасных изображений) и удаление невидимых поверхностей (граней для сплошных изображений).

Анализ видимости объектов можно производить как в исходном трехмерном пространстве, так и на картинной плоскости. Это приводит к разделению методов на два класса:

- методы, работающие непосредственно в пространстве самих объектов;
- методы, работающие в пространстве картинной плоскости, т. е. работающие с проекциями объектов.

Получаемый результат представляет собой либо набор видимых областей или отрезков, заданных с машинной точностью (имеет непрерывный вид), либо информацию о ближайшем объекте для каждого пиксела экрана (имеет дискретный вид).

Методы первого класса дают точное решение задачи удаления невидимых линий и поверхностей, никак не привязанное к растровым свойствам картинной плоскости.

Они могут работать как с самими объектами, выделяя те их части, которые видны, так и с их проекциями на картинную плоскость, выделяя на ней области, соответствующие проекциям видимых частей объектов, и, как правило, практически не привязаны к растровой решетке и свободны от погрешностей дискретизации. Так как эти методы работают с непрерывными исходными данными и получающиеся результаты не зависят от растровых свойств, то их иногда называют непрерывными (continuous methods).

Простейший вариант непрерывного подхода заключается в сравнении каждого объекта со всеми остальными, что дает временные затраты, пропорциональные n^2, где *n* - количество объектов в сцене.

Однако следует иметь в виду, что непрерывные методы, как правило, достаточно сложны.

Методы второго класса (point-sampling methods) дают приближенное решение задачи видимости, определяя видимость только в некотором наборе точек картинной плоскости - в точках растровой решетки. Они очень сильно привязаны к растровым свойствам картинной плоскости и фактически заключаются в определении для каждого пиксела той грани, которая является ближайшей к нему вдоль направления

проектирования. Изменение разрешения приводит к необходимости полного перерасчета всего изображения.

Простейший вариант дискретного метода имеет временные затраты порядка Cn, где C - общее количество пикселов экрана, а n - количество объектов.

Всем методам второго класса традиционно свойственны ошибки дискретизации (aliasing artifacts). Однако, как правило, дискретные методы отличаются известной простотой.

Кроме этого существует довольно большое количество смешанных методов, использующих работу как в объектном пространстве, так и в картинной плоскости, методы, выполняющие часть работы с непрерывными данными, а часть - с дискретными.

Большинство алгоритмов удаления невидимых граней и поверхностей тесно связано с различными методами сортировки. Некоторые алгоритмы проводят сортировку явно, в некоторых она присутствует в скрытом виде. Приближенные методы отличаются друг от друга фактически только порядком и способом проведения сортировки.

Очень распространенной структурой данных в задачах удаления невидимых линий и поверхностей являются различные типы деревьев - двоичные (BSP-trees), четвертичные (Quadtrees), восьмеричные (Octtrees) и др.

Методы, практически применяющиеся в настоящее время, в большинстве являются комбинациями ряда простейших алгоритмов, неся в себе целый ряд разного рода оптимизаций.

Крайне важная роль в повышении эффективности методов удаления невидимых линий и граней отводится использованию когерентности (от английского coherence - связность). Выделяют несколько типов когерентности:

- когерентность в картинной плоскости - если данный пиксел соответствует точке грани P, то скорее всего соседние пикселы также соответствуют точкам той же грани (рис. 10.3);
- когерентность в пространстве объектов - если данный объект (грань) видим (невидим), то расположенный рядом объект (грань) скорее всего также является видимым (невидимым) (рис. 10.4).

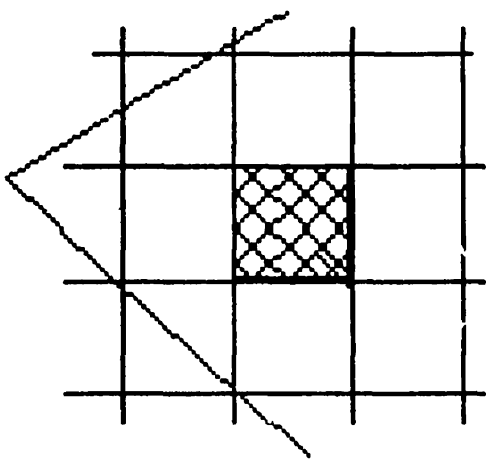


Рис. 10.3

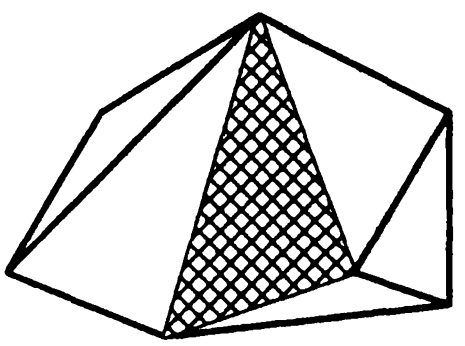


Рис. 10. 4.

- в случае построения анимации возникает третий тип когерентности – временная: грани, видимые в данном кадре, скорее всего будут видимы и в следующем; аналогично грани, невидимые в данном кадре, скорее всего будут невидимы и в следующем.

Аккуратное использование когерентности позволяет заметно сократить количество возникающих проверок и заметно повысить быстродействие алгоритма.

10.1. Построение графика функции двух переменных. Линии горизонта

Рассмотрим задачу построения графика функции двух переменных $z = f(x, y)$ в виде сетки координатных линий $x = const$ и $y = const$ (рис. 10.5).

При параллельном проектировании вдоль оси Oz проекцией вертикальной линии в объектном пространстве будет вертикальная линия на картинной плоскости (экране). Легко убедиться в том, что в этом случае точка $p(x, y, z)$ переходит в точку $((p, e_1), (p, e_2))$ на картинной плоскости, где

$$e_1 = (\cos\varphi, \sin\varphi, 0),$$

$$e_2 = (\sin\varphi\sin\psi, -\cos\varphi\sin\psi, \cos\psi),$$

а направление проектирования имеет вид

$$e_3 = (\sin\varphi\cos\psi, -\cos\varphi\cos\psi, -\sin\psi),$$

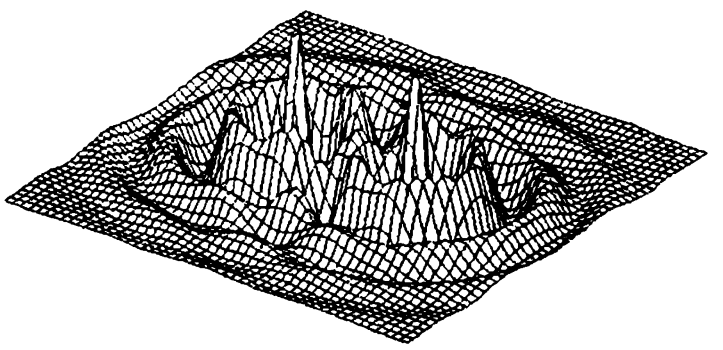


Рис. 10.5

где $\varphi \in [0, 2\pi], \psi \in \left[-\frac{\pi}{2}, \frac{\pi}{2}\right]$

Чаще всего применяется полигональный способ представления графика: функция приближается прямоугольной матрицей значений функции в узлах сетки, а сам график представляется наборами ломаных линий, соответствующих постоянным значениям x и y.

Рассмотрим сначала построение графика функции в виде набора линий, соответствующих постоянным значениям y, считая, что углы φ и ψ подобраны таким образом, что при $y_1 > y_2$ плоскость $y = y_1$ расположена ближе к картинной плоскости, чем плоскость $y = y_2$.

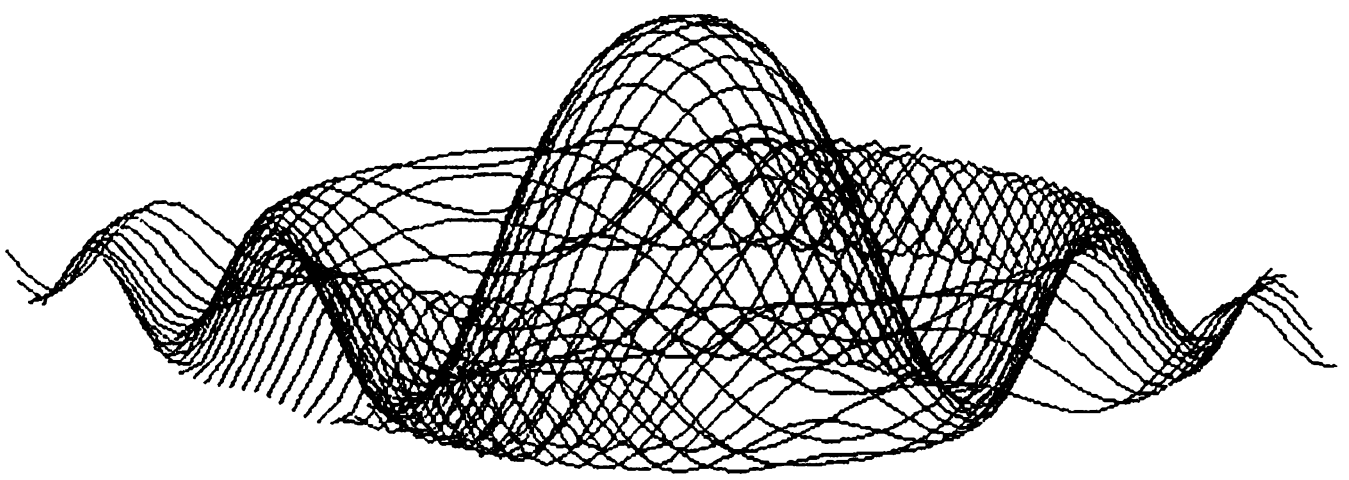


Рис. 10.6

Программу, осуществляющую построение графика функции двух переменных без удаления невидимых линий, написать несложно, однако получающееся при этом изображение зачастую оказывается слишком запутанным и непонятным (рис. 10.6). Поэтому естественным образом возникает задача о таком способе построения графика функции двух переменных, при котором невидимые линии удалялись бы (рис. 10.7).



Рис. 10.7

Каждая линия семейства $z = f(x, y_i)$ лежит в своей плоскости $y = y_i$, причем все эти плоскости параллельны и, следовательно, не могут пересекаться. Из этого следует, что при $y_j > y_i$ линия $z = f(x, y_j)$ не может закрывать собой линию $z = f(x, y_i)$ и, значит, каждая линия $z = f(x, y_j)$ может быть закрыта только предыдущими линиями $z = f(x, y_i)$, $i = 1, \ldots, j\text{-}1$.

Тем самым возможен следующий алгоритм построения графика функции $z = f(x, y)$: линии рисуются в порядке удаления (возрастания y - front-to-back) и при рисовании очередной линии выводится только та ее часть, которая ранее нарисованными линиями не закрывается.

Для определения частей линии $z = f(x, y_k)$, которые не закрывают ранее нарисованных линий, вводятся так называемые линии горизонта, или контурные линии.

Изначально линии горизонта неинициализированны, поэтому первая линия выводится полностью (так как она ближе всего расположена к наблюдателю, то закрывать ее ничто не может). После этого линии горизонта инициализируются так, что в выводимых точках они совпадают с линией, выведенной первой.

Вторая линия также выводится полностью, и линии горизонта корректируются следующим образом: нижняя линия горизонта в любой из точек равна минимуму из двух уже выведенных линий, верхняя - максимуму (рис. 10.8).



Рис. 10.8

Рассмотрим область экрана между верхней и нижней линиями горизонта - она является проекцией части графика функции, заключенной в полосе $y_1 \le y \le y_2$, и, очевидно, находится ближе, чем все остальные линии вида $z = f(x, y_i)$, $i > 2$. Поэтому те части линий, которые при проектировании попадают в эту область, указанной частью графика закрываются и при данном способе проектирования не видны.

Тем самым следующая линия будет рисоваться только в тех местах, где ее проекция лежит вне области, задаваемой контурными линиями (рис. 10.9).



Рис. 10.9

Пусть проекцией линии $z = f(x, y_k)$ на картинную плоскость является линия $Y = Y_k(X)$, где (X, Y) - координаты на картинной плоскости, причем Y - вертикальная координата. Контурные линии $Y^*_{max}(X)$ и $Y^*_{min}(X)$ определяются следующими соотношениями:

$$Y^k_{max}(X) = \max_{1 \le i \le k-1} Y_i(X),$$

$$Y^k_{min}(X) = \min_{1 \le i \le k-1} Y_i(X).$$

На экране рисуются только те части линии $Y = Y_k(X)$, которые находятся выше линии $Y^*_{max}(X)$ или ниже линии $Y^*_{min}(X)$.

Такой алгоритм называется методом плавающего горизонта.

Возможны разные способы представления линий горизонта, но одной из наиболее простых и эффективных реализаций данного метода является растровая реализация, при которой каждая линия горизонта представлена набором значений Y с шагом в 1 пиксел.

```
int  YMax [SCREEN_WIDTH];
int  YMin [SCREEN_WIDTH];
```

Для рисования сегментов этой ломаной используется модифицированный алгоритм Брезенхейма, который перед выводом очередного пиксела сравнивает его ординату с верхней и нижней контурными линиями.

Замечание. Случай отрезков с угловым коэффициентом по модулю, большему единицы, требует специальной обработки (чтобы не появлялись выпадающие пикселы (рис. 10.10)).

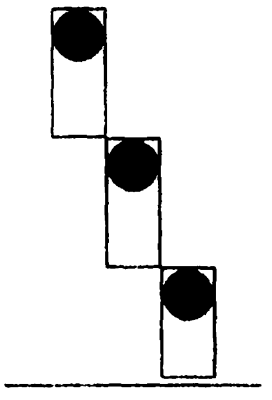


Рис. 10.10

Реализация этого алгоритма приведена ниже.

```
// File example1.cpp
#include    <conio.h>
#include    <graphics.h>
#include    <math.h>
#include    <process.h>
#include    <stdio.h>
#include    <stdlib.h>

#define NO_VALUE    7777

struct   Point              // screen point
{
    int  x, y;
};
int  YMax [640];
int  YMin [640];
int  upColor    = LIGHTGREEN;
int  downColor = LIGHTGRAY;

void     drawLine ( Point& p1, Point& p2 )
```

```
{
    int  dx = abs ( p2.x - p1.x );
    int  dy = abs ( p2.y - p1.y );
    int  sx = p2.x >= p1.x ? 1   -1;
    int  sy = p2.y >= p1.y ? 1   -1;

    if ( dy <= dx )
    {
       int    d  = -dx;
       int    d1 = dy << 1;
       int    d2 = ( dy - dx ) << 1;

       for(int x = p1.x,y = p1.y, i = 0; i <= dx; i++, x += sx)
       {
         if ( YMin [x] == NO_VALUE ) // YMin, YMax not inited
         {
            putpixel ( x, y, upColor );
            YMin [x] = YMax [x] = y;
         }
         else
         if ( y < YMin [x] )
         {
            putpixel ( x, y, upColor );
            YMin [x] = y;
         }
         else
         if ( y > YMax [x] )
         {
            putpixel ( x, y, downColor );
            YMax [x] = y;
         }
        if ( d > 0 )
        {
          d += d2;
          y += sy;
        }
        else
          d += d1;
    }
}
else
{
    int  d  = -dy;
    int  d1 = dx << 1;
    int  d2 = ( dx - dy ) << 1;
    int  m1 = YMin [p1.x];
    int  m2 = YMax [p1.x];

    for ( int x = p1.x, y = p1.y, i = 0; i <= dy; i++, y += sy )
    {
       if ( YMin [x] == NO_VALUE ) // YMin, YMax not inited
       {
         putpixel ( x, y, upColor );
         YMin [x] = YMax [x] = y;
```

```
        }
        else
        if ( y < m1 )
        {
          putpixel ( x, y, upColor );
          if ( y < YMin [x] )
               YMin [x] = y;
        }
        else
      if ( y > m2 )
        {
          putpixel ( x, y, downColor );
          if ( y > YMax [x] )
               YMax [x] = y;
        }
    if ( d > 0 )
    {
         d += d2;
         x += sx;
         m1 = YMin [x];
         m2 = YMax [x];
    }
    else
         d += d1;
  }
}
void     plotSurface ( float x1, float y1, float x2, float y2,
                       float (*f)( float, float ),
                       float fMin, float fMax, int n1, int n2 )
{
    Point *  curLine = new Point [n1];
    float    phi     = 30*M_PI/180;
    float    psi     = 10*M_PI/180;
    float    sphi    = sin ( phi );
    float    cphi    = cos ( phi );
    float    spsi    = sin ( psi );
    float    cpsi    = cos ( psi );
    float    e1 []   = { cphi, sphi, 0 };
    float    e2 []   = { spsi*sphi, -spsi*cphi, cpsi };
    float    x, y;
    float    hx      = ( x2 - x1 ) / n1;
    float    hy      = ( y2 - y1 ) / n2;
    float    xMin    = ( e1 [0] >= 0 ? x1  x2 ) * e1 [0] + ( e1 [1] >= 0 ? y1  y2 ) * e1 [1];
    float    xMax    = ( e1 [0] >= 0 ? x2  x1 ) * e1 [0] + ( e1 [1] >= 0 ? y2  y1 ) * e1 [1];
    float    yMin    = ( e2 [0] >= 0 ? x1  x2 ) * e2 [0] + ( e2 [1] >= 0 ? y1 : y2 ) * e2 [1];
    float    yMax    = ( e2 [0] >= 0 ? x2  x1 ) * e2 [0] + ( e2 [1] >= 0 ? y2 : y1 ) * e2 [1];
    int      i, j, k;

    if ( e2 [2] >= 0 )
    {
        yMin += fMin * e2 [2];
        yMax += fMax * e2 [2];
    }
```

```
else
{
     yMin += fMax * e2 [2];
     yMax += fMin * e2 [2];
}
float     ax = 10 - 600 * xMin / ( xMax - xMin );
float     bx = 600 / ( xMax - xMin );
float     ay = 10 - 300 * yMin / ( yMax - yMin );
float     by = -300 / ( yMax - yMin );

for ( i = 0; i < sizeof ( YMax ) / sizeof ( int ); i++ )
     YMin [i] = YMax [i] = NO_VALUE;

for ( i = n2 - 1; i > -1; i-- )
{
   for ( j = 0; j < n1; j++ )
   {
     x = x1 + j * hx;
     y = y1 + i * hy;
     curLine [j].x = (int)(ax+bx*(x*e1 [0]+y*e1 [1]));
     curLine [j].y = (int)(ay+by*(x*e2 [0]+y*e2 [1]+f ( x, y ) * e2 [2] ) );
   }

   for ( j = 0; j < n1 - 1; j++ )
     drawLine ( curLine [j], curLine [j + 1] );
}

delete curLine;
}
float     f ( float x, float y )
{
     float     r =  x*x + y*y;
     return 0.5*sin(2*x)*sin(2*y);
}
main ()
{
     int  driver = DETECT;
     int  mode;
     int  res;

     initgraph ( &driver, &mode, "" );
     if ( ( res = graphresult () ) != grOk )
     {
          printf("\nGraphics error: %s\n", grapherrormsg(res));
          exit ( 1 );
     }
     plotSurface ( -2, -2, 2, 2, f2, -0.5, 1, 40, 40 );

     getch ();
     closegraph ();
}
```

Функция drawLine осуществляет построение отрезка, соединяющего две ные точки *p*1 и *p*2, проводя отсечение с учетом линий горизонта YMin и YMax необходимости корректируя их. При этом часть отрезка, лежащая выше верхн

нии горизонта, рисуется цветом upColor, а часть, лежащая ниже нижней линии горизонта, рисуется цветом downColor.

Для вывода графика, состоящего из линий $z = f(x, y_i)$ и $z = f(x_j, y)$, необходимо изменить процедуру plotSurface так, чтобы отрезки ломаных выводились в порядке удаления от картинной плоскости (с сохранением порядка загораживания), ибо возможна ситуация, когда отрезок x-линии закрывает часть отрезка y-линии и наоборот. Возможно несколько типов подобного упорядочения (включая и обыкновенную сортировку), простейшим из которых является следующий: сначала выводится линия $z = f(x, y_i)$, затем отрезки, соединяющие эту линию со следующей линией, и т. д.

Ниже приводится текст функции plotSurface2, осуществляющей такое построение.

```
// File example2.cpp
void  plotSurface2 ( double x1, double y1, double x2, double y2,
      double (*f)( double, double ), double fMin, double fMax,
      int n1, int n2 )
{
  Point2 * curLine  = new Point [n1];
  Point2 * nextLine = new Point [n1];
  float   phi       = 30*M_PI/180;
  float   psi       = 10*M_PI/180;
  float   sphi      = sin ( phi );
  float   cphi      = cos ( phi );
  float   spsi      = sin ( psi );
  float   cpsi      = cos ( psi );
  float   e1 []     = { cphi, sphi, 0 };
  float   e2 []     = { spsi*sphi, -spsi*cphi, cpsi };
  float   x, y;
  float   hx        = ( x2 - x1 ) / n1;
  float   hy        = ( y2 - y1 ) / n2;
  float   xMin      = ( e1 [0] >= 0 ? x1 : x2 ) * e1 [0] + ( e1 [1] >= 0 ? y1 : y2 ) * e1 [1];
  float   xMax      = ( e1 [0] >= 0 ? x2 : x1 ) * e1 [0] + ( e1 [1] >= 0 ? y2 : y1 ) * e1 [1];
  float   yMin      = ( e2 [0] >= 0 ? x1 : x2 ) * e2 [0] + ( e2 [1] >= 0 ? y1 : y2 ) * e2 [1];
  float   yMax      = ( e2 [0] >= 0 ? x2 : x1 ) * e2 [0] + ( e2 [1] >= 0 ? y2 : y1 ) * e2 [1];
  int     i, j, k;
  if ( e2 [2] >= 0 )
  {
    yMin += fMin * e2 [2];
    yMax += fMax * e2 [2];
  }
  else
  {
    yMin += fMax * e2 [2];
    yMax += fMin * e2 [2];
  }
  float  ax = 10 - 600 * xMin / ( xMax - xMin );
  float  bx = 600 / ( xMax - xMin );
  float  ay = 10 - 400 * yMin / ( yMax - yMin );
  float  by = -400 / ( yMax - yMin );
  for ( i = 0; i < sizeof ( YMax ) / sizeof ( int ); i++ )
    YMin [i] = YMax [i] = NO_VALUE;
  for ( i = 0; i < n1, i++ )
```

```
{
  x = x1 + i * hx;
  y = y1 + ( n2 - 1 ) * hy;
  curLine [i].x = (int)(ax + bx * ( x * e1 [0] + y * e1 [1]));
  curLine [i].y = (int)(ay + by * ( x * e2 [0] + y * e2 [1] + f ( x, y ) * e2 [2] ) );
}
for ( i = n2 - 1; i > -1; i-- )
{
  for ( j = 0; j < n1 - 1; j++ )
    drawLine ( curLine [j], curLine [j + 1] );
  if ( i > 0 )
    for ( j = 0; j < n1; j++ )
    {
      x = x1 + j * hx;
      y = y1 + ( i - 1 ) * hy;
      nextLine [j].x = (int)( ax + bx * ( x * e1 [0] + y * e1 [1] ) );
      nextLine [j].y = (int)( ay + by * ( x * e2[0] + y * e2[1] + f ( x, y ) * e2 [2] ) );
      drawLine ( curLine [j], nextLine [j] );
      curLine [j] = nextLine [j];
    }
}
delete curLine;
delete nextLine;
}
```

Следует иметь в виду, что в общем случае порядок вывода отрезков зависит от выбора углов φ и ψ.

Рассмотрим теперь задачу построения полутонового изображения графика функции $z = f(x, y)$.

Как и ранее, введем сетку

$$\{(x_i, y_j), i = 1,...,n_1, j = 1,...,n_2\},$$

затем приблизим график функции набором треугольных граней с вершинами в точках $(x_i, y_j, f(x_i, y_j))$.

Для удаления невидимых граней воспользуемся методом упорядоченного вывода граней. В данном случае треугольники выводятся не по мере удаления от картинной плоскости, а по мере их приближения, начиная с дальних и заканчивая ближними: треугольники, расположенные ближе к плоскости экрана, выводятся позже и закрывают собой невидимые части более дальних треугольных граней так, что если данный пиксел накрывается сразу несколькими гранями, то в этом пикселе будет видна грань, ближайшая к картинной плоскости.

В результате применения описанной процедуры мы получаем правильное изображение поверхности.

Для определения порядка, в котором должны выводиться грани, воспользуемся тем, что треугольники, лежащие в полосе

$$\{(x,y), y_i \le y \le y_{i+1}\},$$

не могут закрывать треугольники из полосы

$$\{(x,y), y_{i-1} \le y \le y_i\}$$

Таким образом, грани можно выводить по полосам, по мере их приближения к картинной плоскости.

Приводимая программа реализует этот алгоритм с использованием 256-цветного режима. Грань окрашивается в цвет, интенсивность которого пропорциональна косинусу угла между нормалью к грани и направлением проектирования (подробнее о закрашивании - в гл. 2 и 11).

```
// File example3.cpp
#include  <conio.h>
#include  <graphics.h>
#include  <math.h>
#include  <process.h>
#include  <stdio.h>
#include  <stdlib.h>
#include  "Vector3D.h"

struct  Point      // screen point
{
  int  x, y;
};

void  plotShadedSurface ( double x1, double y1, double x2,
     double y2, double (*f)( double, double ), double fmin,
     double fmax, int n1, int n2 )
{
  Point       *     curLine  = new Point [n1];
  Point       *     nextLine = new Point [n1];
  Vector3D *      curPoint = new Vector3D [n1];
  Vector3D *      nextPoint= new Vector3D [n1];
  float     phi     = 30*M_PI/180;
  float     psi     = 20*M_PI/180;
  float     sphi    = sin ( phi );
  float     cphi    = cos ( phi );
  float     spsi    = sin ( psi );
  float     cpsi    = cos ( psi );
  Vector3D        e1 ( cphi, sphi, 0 );
  Vector3D        e2 ( spsi*sphi, -spsi*cphi, cpsi );
  Vector3D        e3 ( sphi*cpsi, -cphi*cpsi, -spsi );
  float     xMin    = ( e1 [0] >= 0 ? x1 : x2 ) * e1 [0]
                    + ( e1 [1] >= 0 ? y1   y2 ) * e1 [1];
  float     xMax    = ( e1 [0] >= 0 ? x2 : x1 ) * e1 [0]
                    + ( e1 [1] >= 0 ? y2   y1 ) * e1 [1];
  float     yMin    = ( e2 [0] >= 0 ? x1   x2 ) * e2 [0]
                    + ( e2 [1] >= 0 ? y1   y2 ) * e2 [1];
  float     yMax    = ( e2 [0] >= 0 ? x2   x1 ) * e2 [0]
                    + ( e2 [1] >= 0 ? y2   y1 ) * e2 [1];
  float     hx      = ( x2 - x1 ) / n1;
  float     hy      = ( y2 - y1 ) / n2;
  Vector3D  edge1, edge2, n;
  Point     facet [3];
  float     x, y;
  int       color;
  int       i, j, k;
```

```
if ( e2 [2] >= 0 )
{
  yMin += fMin * e2 [2];
  yMax += fMax * e2 [2];
}
else
{
  yMin += fMax * e2 [2];
  yMax += fMin * e2 [2];
}
float  ax = 20 - 600 * xMin / ( xMax - xMin );
float  bx = 600 / ( xMax - xMin );
float  ay = 40 - 400 * yMin / ( yMax - yMin );
float  by = -400 / ( yMax - yMin );
for ( i = 0; i < 64; i++ )
{
  setrgbpalette ( i, 0, 0, i );
  setrgbpalette ( 64 + i, 0, i, 0 );
}
for ( i = 0; i < n1; i++ )
{
  curPoint [i].x = x1 + i * hx;
  curPoint [i].y = y1;
  curPoint [i].z = f ( curPoint [i].x, curPoint [i].y );
  curLine [i].x = (int)( ax + bx * ( curPoint [i] & e1 ) );
  curLine [i].y = (int)( ay + by * ( curPoint [i] & e2 ) );
}
for ( i = 1; i < n2; i++ )
{
  for ( j = 0; j < n1; j++ )
  {
    nextPoint [j].x = x1 + j * hx;
    nextPoint [j].y = y1 + i * hy;
    nextPoint [j].z = f ( nextPoint [j].x, nextPoint [j].y );
    nextLine  [j].x = (int)(ax + bx * ( nextPoint [j] & e1 ));
    NextLine  [j].y = (int)(ay + by * ( NextPoint [j] & e2 ));
  }
  for ( j = 0; j < n1 - 1; j++ )
  {
        // draw 1st triangle
    edge1 = curPoint [j+1] - curPoint [j];
    edge2 = nextPoint [j] - curPoint [j];
    n = edge1 ^ edge2;
    if ( ( n & e3 ) >= 0 )
      color = 64 + (int)( 20 + 43 * ( n & e3 ) / !n );
    else
      color = (int)( 20 - 43 * ( n & e3 ) / !n );
    setfillstyle ( SOLID_FILL, color );
    setcolor ( color );
```

```
  facet [0] = curLine [j];
  facet [1] = curLine [j+1];
  facet [2] = nextLine [j];
  fillpoly ( 3, ( int far * ) facet );
       // draw 2nd triangle
  edge1 = nextPoint [j+1] - curPoint [j+1];
  edge2 = nextPoint [j] - curPoint [j+1];
  n = edge1 ^ edge2;
  if ( ( n & e3 ) >= 0 ) {
    color = 127;
    color = 64 + (int)( 20 + 43 * ( n & e3 ) / !n );
  }
  else {
    color = 63;
    color = (int)( 20 - 43 * ( n & e3 ) / !n );
  }
  setfillstyle ( SOLID_FILL, color );
  setcolor ( color );
  facet [0] = curLine [j+1];
  facet [1] = nextLine [j];
  facet [2] = nextLine [j+1];
  fillpoly ( 3, ( int far * ) facet );
 }
 for ( j = 0; j < n1; j++ )
 {
   curLine  [j] = nextLine [j];
     curPoint [j] = nextPoint [j];
    }
  }
  delete curLine;
  delete nextLine;
  delete curPoint;
  delete nextPoint;
}

double  f2 ( double x, double y )
{
  double  r = x*x + y*y;
  return cos ( r ) / ( r + 1 );
}

main ()
{
  int  driver;
  int  mode = 2;       // suggested mode 640x480x256
  int  res;
  if ((driver = installuserdriver ("VESA", NULL)) == grError)  {
    printf ( "\nCannot load VESA driver" );
    exit ( 1 );
  }
  initgraph ( &driver, &mode, "" );
  if ( ( res = graphresult () ) != grOk )  {
    printf("\nGraphics error: %s\n", grapherrormsg (res) );
```

```
    exit ( 1 );
  }
  plotShadedSurface ( -2, -2, 2, 2, f2, -0.5, 1, 30, 30 );
  getch ();
  closegraph ();
}
```

10.2. Методы оптимизации

10.2.1. Отсечение нелицевых граней

Рассмотрим многогранник, для каждой грани которого задан единичный вектор внешней нормали (рис. 10.11). Несложно заметить, что если вектор нормали грани n составляет с вектором l, задающим направление проектирования, тупой угол (вектор нормали направлен от наблюдателя), то эта грань заведомо не может быть видна (рис. 10.12). Такие грани называются нелицевыми. Если соответствующий угол является острым, грань называется лицевой.

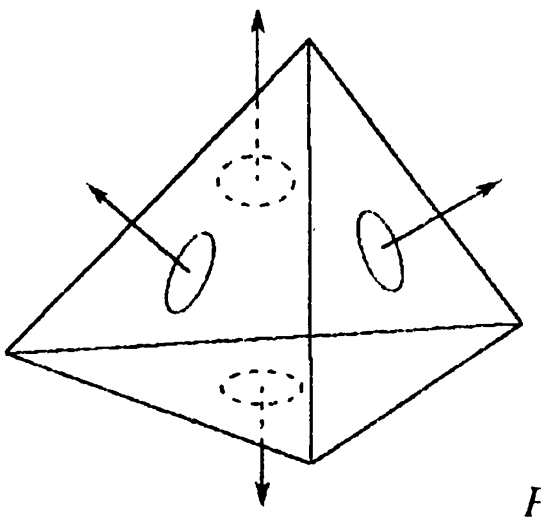


Рис. 10.11

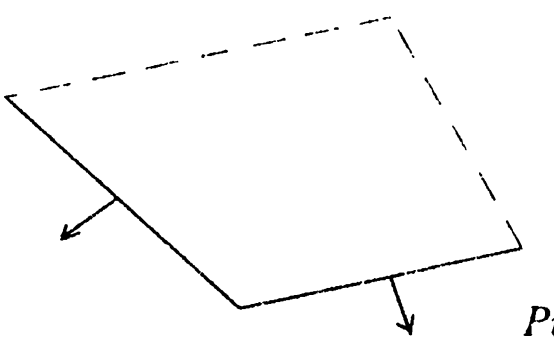


Рис. 10.12

При параллельном проектировании условие на угол можно записать в виде неравенства $(n, l) \le 0$, поскольку направление проектирования от грани не зависит.

При центральном проектировании с центром в точке c вектор проектирования для точки p будет равен $l = c - p$.

Для определения того, является заданная грань лицевой или нет, достаточно взять произвольную точку p этой грани и проверить выполнение условия $(n, l) \le 0$.

Знак этого скалярного произведения не зависит от выбора точки на грани, а определяется тем, в каком полупространстве относительно плоскости, содержащей данную грань, лежит центр проектирования.

Так как при центральном проектировании проектирующий луч зависит от грани (и не зависит от выбора точки на грани), то лицевая грань может стать нелицевой, а нелицевая лицевой даже при параллельном сдвиге. При параллельном проектировании сдвиг не изменяет углов и то, является ли грань лицевой или нет, зависит только от угла между нормалью к грани и направлением проектирования.

Заметим, что если по аналогии с определением принадлежности точки многоугольнику пропустить через произвольную точку картинной плоскости проектирующий луч к объектам сцены, то число пересечений луча с лицевыми гранями будет равняться числу пересечений луча с нелицевыми гранями.

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

```
// File cube.cpp
#include    <Bios.h>
#include    <Graphics.h>
#include    "Vector3D.h"
#include    "Matrix.h"

struct   Point
{
    int  x, y;
};

struct   Edge
{
    int  v1, v2;        // vertices indexes
    int  f1, f2;        // facet's indexes
};

struct   Facet
{
    int  v [4];         // vertices indexes
    Vector  n;          // normal
    int  flags;
};

class    Cube
{
public:
    Vector3D vertex [8];
    Edge     edge [12];
    Facet    facet [6];

    Cube ();

    void     initEdge ( int i, int v1, int v2, int f1, int f2 )
    {
        edge [i].v1 = v1;
        edge [i].v2 = v2;
        edge [i].f1 = f1;
        edge [i].f2 = f2;
    };
    void     initFacet ( int i, int v1, int v2, int v3, int v4 )
    {
        facet [i].v [0] = v1;
        facet [i].v [1] = v2;
        facet [i].v [2] = v3;
        facet [i].v [3] = v4;
    };
    void     computeNormals ();
    int  isFrontFacing ( int i , Vector& v )
  {
   return (( vertex [facet [i].v [0]] - v ) & facet [i].n ) < 0;
  }
    void     apply ( Matrix& );
    void     draw ();
};
```

```
////////////////////////////////////////////////////////////
Matrix   prj ( 1 );          // projection matrix
Vector  eye ( 0 );              // observer loc
Vector  light ( 0, 0.7, 0.7 );
////////////////////////////////////////////////////////////
Cube    Cube ()
{
    // 1. init vertices
    for ( int i = 0; i < 8; i++ )
    {
        vertex [i].x = i & 1 ? 1.0 : 0.0;
        vertex [i].y = i & 2 ? 1.0 : 0.0;
        vertex [i].z = i & 4 ? 1.0 : 0.0;
    }
    // 2. init edges
    initEdge ( 0,  0, 1, 2, 4 );
    initEdge ( 1,  1, 3, 1, 4 );
    initEdge ( 2,  3, 2, 3, 4 );
    initEdge ( 3,  2, 0, 0, 4 );
    initEdge ( 4,  4, 5, 2, 5 );
    initEdge ( 5,  5, 7, 1, 5 );
    initEdge ( 6,  7, 6, 3, 5 );
    initEdge ( 7,  6, 4, 0, 5 );
    initEdge ( 8,  0, 4, 0, 2 );
    initEdge ( 9,  1, 5, 1, 2 );
    initEdge ( 10, 3, 7, 1, 3 );
    initEdge ( 11, 2, 6, 0, 3 );
    // 3. init facets
    initFacet ( 0, 4, 6, 2, 0 );
    initFacet ( 1, 1, 3, 7, 5 );
    initFacet ( 2, 0, 1, 5, 4 );
    initFacet ( 3, 6, 7, 3, 2 );
    initFacet ( 4, 2, 3, 1, 0 );
    initFacet ( 5, 4, 5, 7, 6 );
}
void     Cube :: computeNormals ()
{
    for ( int i = 0; i < 6; i++ )
        facet [i].n = ( vertex [facet [i].v [1]] - vertex [facet [i].v [0]] ) ^
                      ( vertex [facet [i].v [2]] - vertex [facet [i].v [1]] );
}
void     Cube    apply ( Matrix& m )
{
    for ( int i = 0; i < 8; i++ )
        vertex [i] = m * vertex [i];
}
void     Cube    draw ()
{
```

```
Point    p [8];
Point    contour [4];
Vector   v;
int  color;

                // project vertices
for ( int i = 0; i < 8; i++ )
{
     v      = prj * vertex [i];
     p [i].x = (int) v.x;
     p [i].y = (int) v.y;
}
computeNormals ();

for ( i = 0; i < 6; i++ )
     facet [i].flags = isFrontFacing ( i, eye );

                // draw all faces
for ( i = 0; i < 6; i++ )
     if ( facet [i].flags )
     {
          int  color = -( facet [i].n & light )*7+8;

          for ( int j = 0; j < 4; j++ )
               contour [j] = p [facet [i].v [j]];

          setcolor ( color );
          setfillstyle ( SOLID_FILL, color );
          fillpoly ( 4, (int far *) contour );
     }
}
main ()
{
     Cube         cube;
     Matrix       trans;
     int       drv  = VGA;
     int       mode = VGAMED;
     palettetype    pal;
                // prepare projection matrix
     prj.x [2][2] = 0;
     prj.x [3][2] = 1;
     prj = Scale ( Vector3D ( 640, 350, 0 ) ) * prj;
     prj = Translate ( Vector3D ( 320, 175, 0 ) ) * prj;

     cube.apply ( Translate ( Vector ( 1, 1, 4 ) ) );

     initgraph ( &drv, &mode, "C:\\BORLANDC\\BGI" );

     getpalette ( &pal );
     for ( int i = 0; i < pal.size; i++ )
          setrgbpalette ( pal.colors [i], (63*i)/15, (63*i)/15,
                                          (63*i)/15 );
     cube.draw ();

     bioskey ( 0 );
     closegraph ();
}
```

Хотя в общем случае предложенный подход и не решает задачи удаления полностью, но тем не менее позволяет примерно вдвое сократить количество рассматриваемых граней вследствие того, что нелицевые грани всегда не видны; что же касается лицевых граней, то в общей ситуации части некоторых лицевых граней могут быть закрыты другими лицевыми гранями (рис. 10.13).

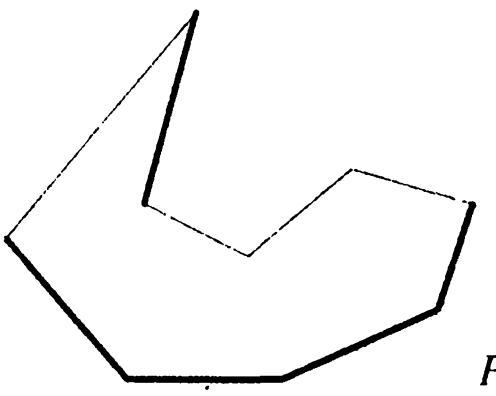


Рис. 10.13

Ребра между нелицевыми гранями также всегда не видны. Однако ребро между лицевой и нелицевой гранями вполне может быть и видимым.

10.2.2. Ограничивающие тела (Bounding Volumes)

При удалении невидимых линий и поверхностей постоянно возникает необходимость сравнения граней и объектов друг с другом. Такие задачи часто оказываются сложными и трудоемкими. Одним из средств, позволяющим заметно упростить подобные сравнения, является использование так называемых ограничивающих объемов (тел).

Опишем вокруг каждого объекта тело достаточно простого вида. Если эти тела не пересекаются, то и содержащиеся внутри них объекты пересекаться не будут. Налицо несомненный выигрыш (рис. 10.14).

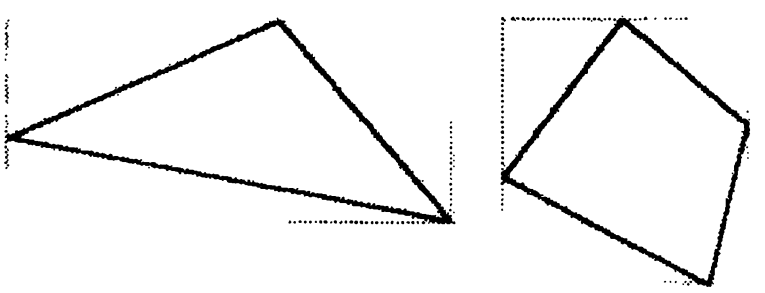


Рис. 10.14

Следует, однако, иметь в виду, что если описанные тела пересекаются, то сами объекты при этом пересекаться не обязаны (рис. 10.15).

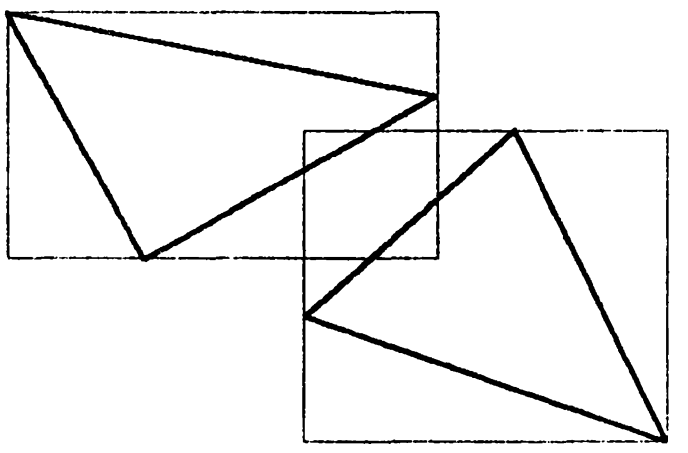


Рис. 10.15

В качестве ограничивающих тел чаще всего используются прямоугольные параллелепипеды с ребрами, параллельными координатным осям. Тогда ограничивающее тело (в данном случае его называют bounding box) описывается шестью числами:

$$(x_{min}, y_{min}, z_{min}); (x_{max}, y_{max}, z_{max}),$$

где первая тройка чисел задает одну вершину параллелепипеда, а вторая - противоположную. Сами числа представляют собой покоординатные значения минимума и максимума из координат точек исходного объекта.

Проверка на пересечение двух тел сводится просто к проверкам на пересечения промежутков

$$[x_{min}, x_{max}], [y_{min}, y_{max}], [z_{min}, z_{max}]$$

одного тела с соответствующими промежутками другого. В случае если пересечение хотя бы одной пары промежутков пусто, то можно сразу заключить, что тела, а следовательно, и содержащиеся внутри их объекты, не пересекаются.

Ограничивающие тела можно строить и для проекций объектов, причем в случае параллельного проектирования вдоль оси *Oz* ограничивающим телом для проекции будет прямоугольник, получающийся из ограничивающего тела для самого объекта отбрасыванием *z*-компоненты.

Ограничивающие тела можно описывать не только вокруг отдельных граней, но и вокруг наборов граней и сложных составных объектов, что позволяет легко отбрасывать сразу целые группы граней и объектов. При этом могут возникать сложные иерархические структуры.

10.2.3. Разбиение пространства (плоскости) (Spatial Subdivision)

Еще одним методом, позволяющим заметно облегчить сравнение объектов друг с другом и использовать когерентность как в пространстве, так и на картинной плоскости, является разбиение пространства (картинной плоскости). С этой целью разбиение пространства строится уже на этапе препроцессирования и для каждой клетки разбиения составляется список всех объектов (граней), которые ее пересекают.

Простейшим вариантом является равномерное разбиение пространства на набор равных прямоугольных клеток (рис. 10.16).

Очень эффективным является использование разбиения картинной плоскости, когда каждой клетке разбиения ставится в соответствие список тех объектов, проекции которых данную клетку пересекают. Для отыскания всех объектов, которые закрывают рассматриваемый объект при проектировании, определяются объекты, попадающие в те же клетки картинной плоскости, что и проекция данного объекта, и на закрывание проверяются только они.

Для сцен с неравномерным распределением объектов имеет смысл использовать неравномерное (адаптивное) разбиение пространства или плоскости (рис. 10.17).

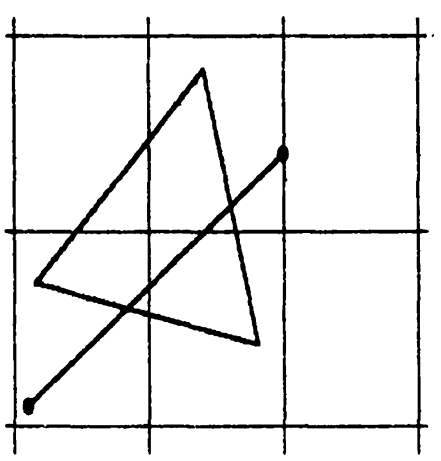


Рис. 10.16

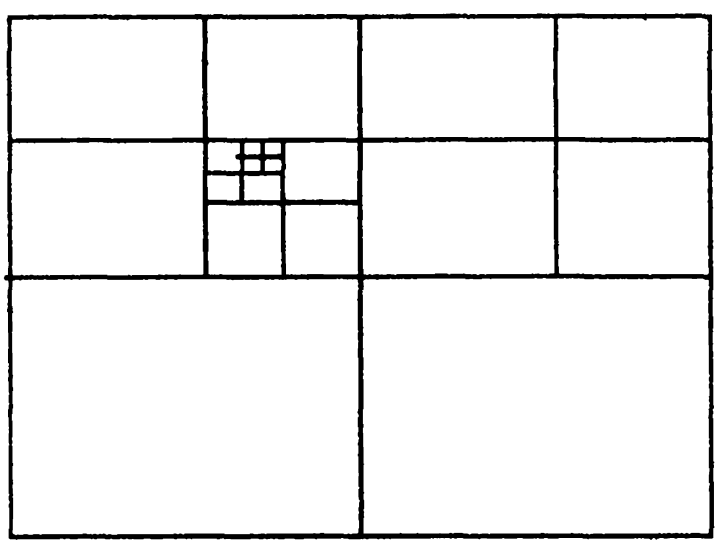


Рис. 10.17

10.2.4. Иерархические структуры (Hierarchies)

При работе с большими объемами данных весьма полезными могут оказаться различные древовидные (иерархические) структуры. Стандартными формами таких структур являются восьмеричные, тетрарные и BSP-деревья, а также деревья ограничивающих тел.

Одной из сравнительно простых структур является иерархия ограничивающих тел (Bounding Volume Hierarchy). Сначала ограничивающее тело описывается вокруг всех объектов. На следующем шаге объекты разбиваются на несколько компактных групп и вокруг каждой из них описывается свое ограничивающее тело. Далее каждая из групп снова разбивается на подгруппы, вокруг каждой из них строится ограничивающее тело и т. д. В результате получается дерево, корнем которого является тело, описанное вокруг всей сцены. Тела, построенные вокруг первичных групп образуют первичных потомков, вокруг вторичных - вторичных и т. д.

Сравнения объектов начинаются с корня. Если сравнение не дает положительного ответа, то все тела можно сразу отбросить. В противном случае проверяются все его прямые потомки, и если какой-либо из них не дает положительного ответа, то все объекты, содержащиеся в нем, сразу же отбрасываются. При этом уже на ранней стадии проверок отсечение основного количества объектов происходит достаточно быстро, ценой всего лишь нескольких проверок.

Иерархические структуры можно строить и на основе разбиения пространства (картинной плоскости): каждая клетка исходного разбиения разбивается на части (которые, в свою очередь, также могут быть разбиты, и т. д. При этом каждая клетка разбиения соответствует узлу дерева).

Иерархии (как и разбиение пространства) позволяют достаточно легко и просто производить частичное упорядочение граней. В результате получается список граней, практически полностью упорядоченный, что дает возможность применить специальные методы сортировки.

Помимо упорядочения граней иерархические структуры позволяют производить быстрое и эффективное отсечение граней, не удовлетворяющих каким-либо из поставленных условий.

10.3. Удаление невидимых линий.

10.3.1. Алгоритм Робертса

Первым алгоритмом удаления невидимых линий был алгоритм Робертса, требующий, чтобы каждая грань была выпуклым многоугольником.

Опишем этот алгоритм.

Сначала отбрасываются все ребра, обе определяющие грани которых являются нелицевыми (ни одно из таких ребер заведомо не видно).

Следующим шагом является проверка на закрывание каждого из оставшихся ребер со всеми лицевыми гранями многогранника. Возможны следующие случаи:

- грань ребра не закрывает (рис. 10.18);

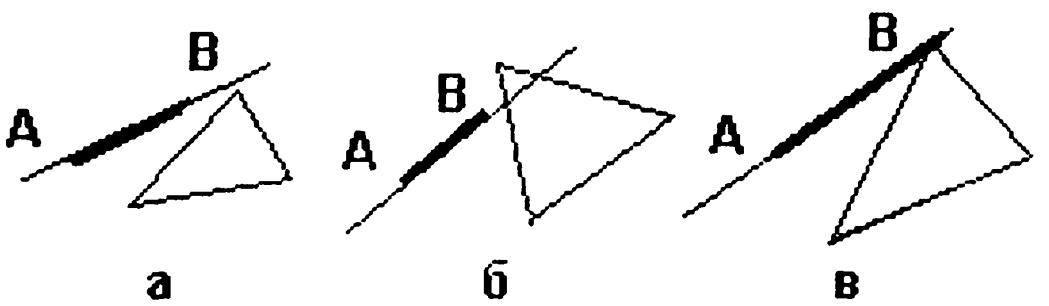


Рис. 10.18

- грань полностью закрывает ребро (тогда оно удаляется из списка рассматриваемых ребер) (рис. 10.19, *а*);
- грань частично закрывает ребро (в этом случае ребро разбивается на несколько частей, видимыми из которых являются не более двух; само ребро удаляется из списка, но в список проверенных ребер добавляются те его части, которые данной гранью не закрываются).

Рассмотрим, как осуществляются эти проверки.

Пусть задано ребро AB, где точка A имеет координаты (x_a, y_a), а точка B - (x_b, y_b).

Прямая, проходящая через отрезок AB, задается уравнениями

$$x = x_a + t(x_b - x_a),$$

$$y = y_a + t(y_b - y_a),$$

причем сам отрезок соответствует значениям параметра $0 \le t \le 1$

Данную прямую можно задать неявным образом как $F(x, y) = 0$, где

$F(x, y) = (y_b - y_a)(x - x_a) - (y_b - x_a)(y - y_a)$.

Предположим, что проекция грани задается набором проекций вершин $P_1, \ldots, P_k$ с координатами (x_i, y_i), $i = 1, \ldots, n$. Обозначим через F_i значение функции F в точке P_i и рассмотрим i-й отрезок проекции грани P_iP_{i+1}. Этот отрезок пересекает прямую AB тогда и только тогда, когда функция F принимает значения разных знаков на концах этого отрезка, а именно при

$$F_iF_{i+1} \le 0$$

Случай, когда $F_{i+1} = 0$, будем отбрасывать, чтобы дважды не засчитывать прямую, проходящую через вершину, для обоих выходящих из нее отрезков.

Итак, мы считаем, что пересечение имеет место в двух случаях:

$F_i \ge 0, F_{i+1} < 0,$

$F_i \le 0, F_{i+1} > 0.$

Точка пересечения определяется соотношениями

$x = x_i + s(x_{i+1} - x_i),$

$y = y_i + s(y_{i+1} - y_i).$

где $s = \dfrac{F_i}{F_i - F_{i+1}}$

Отсюда легко находится значение параметра t:

$$t = \begin{cases} \dfrac{x - x_x}{x_b - x_a}, |x_b - x_a| \ge |y_b - y_a|, \\ \dfrac{y - y_a}{y_b - y_a}, |y_b - y_a| > |x_b - x_a|. \end{cases}$$

Возможны следующие случаи:

1. Отрезок не имеет пересечения с проекцией грани, кроме, быть может, одной точки. Это может иметь место, когда
- прямая AB не пересекает ребра проекции (рис. 10.18, *а*);

- прямая AB пересекает ребра в двух точках t_1 и t_2, но либо $t_1 < 0$, $t_2 < 0$, либо $t_1 > 1$, $t_2 > 1$ (рис. 10.18, *б*);
- прямая AB проходит через одну вершину, не задевая внутренности треугольника (рис. 10.18, *в*).
  Очевидно, что в этом случае соответствующая грань никак не может закрывать собой ребро AB.

2. Проекция ребра полностью содержится внутри проекции грани (рис. 10.19, *а*). Тогда есть две точки пересечения прямой AB и границы грани и $t_1 < 0 < 1 < t_2$. Если грань лежит ближе к картинной плоскости, чем ребро, то ребро полностью невидимо и удаляется.

3. Прямая AB пересекает ребра проекции грани в двух точках и либо $t_1 < 0 \le t_2 \le 1$, либо $0 < t_1 \le 1 < t_2$ (рис. 10.19, *б* и *в*). Если ребро AB находится дальше от картинной плоскости, чем соответствующая грань, то оно разбивается на две части, одна из которых полностью закрывается гранью и потому отбрасывается. Проекция второй части лежит вне проекции грани и поэтому этой гранью не закрывается.

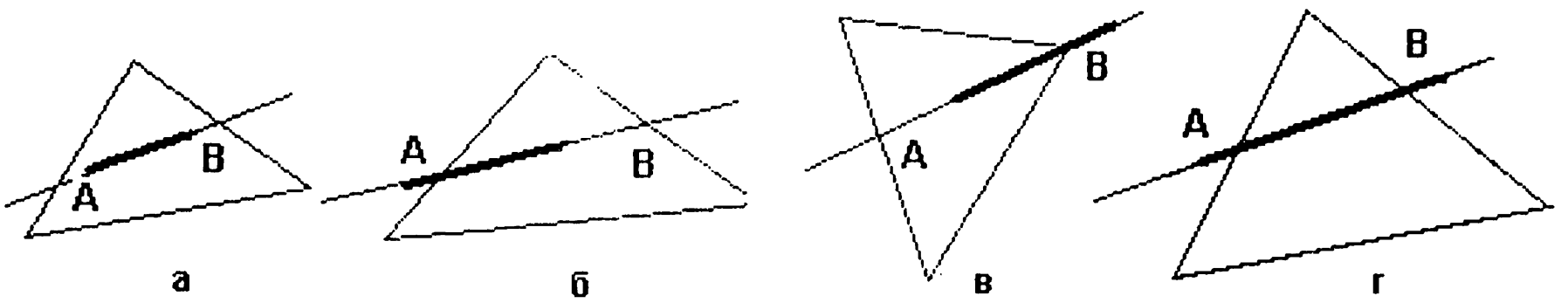


Рис. 10.19

4. Прямая AB пересекает ребра проекции грани в двух точках, причем $0 < t_1 < t_2 < 1$ (рис. 10.19, *г*). Если ребро AB лежит дальше от картинной плоскости, чем соответствующая грань, то оно разбивается на три части, средняя из которых отбрасывается.

Для определения того, что лежит ближе к картинной плоскости - отрезок AB (проекция которого лежит в проекции грани) или сама грань, через эту грань проводится плоскость

$$(n, p) + c = 0$$

(n - нормальный вектор грани), разбивающая все пространство на два полупространства. Если оба конца отрезка AB лежат в том же полупространстве, в котором находится и наблюдатель, то отрезок лежит ближе к грани; если оба конца находятся в другом полупространстве, то отрезок лежит дальше. Случай, когда концы лежат в разных полупространствах, здесь невозможен (это означало бы, что отрезок AB пересекает внутреннюю часть грани).

Если общее количество граней равно n, то временные затраты для данного алгоритма составляют $O(n^2)$.

Количество проверок можно заметно сократить, если воспользоваться разбиением картинной плоскости.

Разобьем видимую часть картинной плоскости (экран) на N_1 x N_2 равных частей (клеток) и для каждой клетки A_{ij} построим список всех лицевых граней, чьи проекции имеют с данной клеткой непустое пересечение.

Для проверки произвольного ребра на пересечение с гранями отберем сначала все те клетки, которые проекция данного ребра пересекает. Ясно, что проверять на пересечение с ребром имеет смысл только те грани, которые содержатся в списках этих клеток.

В качестве шага разбиения обычно выбирается $O(l)$, где l - характерный размер ребра в сцене. Для любого ребра количество проверяемых граней практически не зависит от общего числа граней и совокупные временные затраты алгоритма на проверку пересечений составляют $O(n)$, где n - количество ребер в сцене.

Поскольку процесс построения списков заключается в переборе всех граней, их проектировании и определении клеток, в которые попадают проекции, то затраты на составление всех списков также составляют $O(n)$.

Пример реализации этого алгоритма можно найти в [8] и [15].

10.3.2. Количественная невидимость. Алгоритм Аппеля

Рассмотрим произвольное гладкое выпуклое тело в пространстве. Взяв произвольную точку P на границе этого тела, назовем ее лицевой, если $(n, l) > 0$, где n - вектор внешней нормали к границе в этой точке. Если же $(n, l) < 0$, то данная точка является нелицевой (и, соответственно, невидимой). В силу гладкости поверхности у лицевой (нелицевой) точки существует достаточно малая окрестность, целиком состоящая из лицевых (нелицевых) точек и проектирующаяся на картинную плоскость взаимнооднозначно (не закрывая саму себя) (рис. 10.20).

У точек, для которых $(n, l) = 0$, подобной окрестности (состоящей только из лицевых или только нелицевых точек) может не существовать. Такие точки, в отличие от (регулярных) точек называются нерегулярными (особыми) точками проектирования (рис. 10.21).

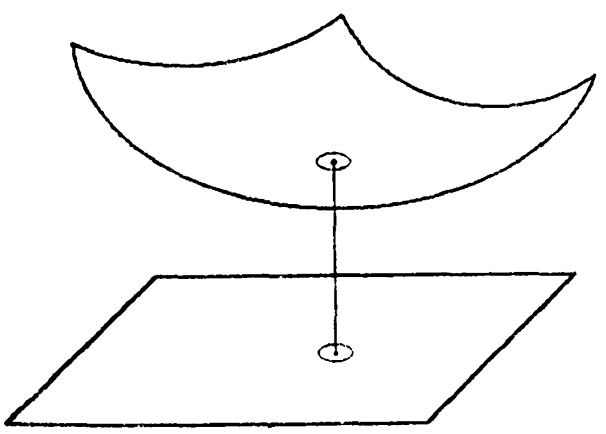


Рис. 10.20

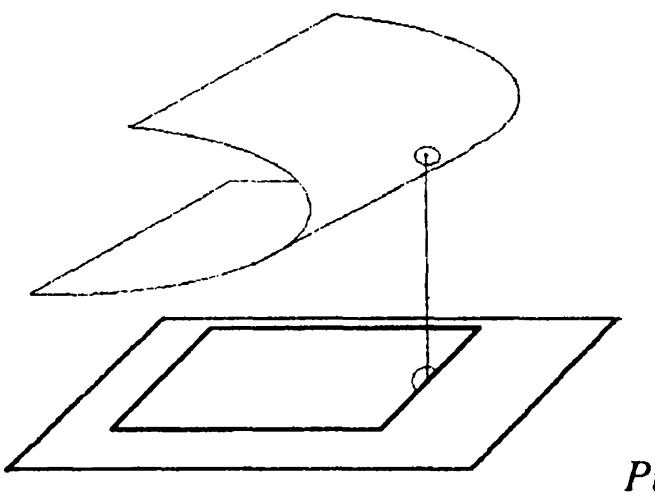


Рис. 10.21

В общем случае множество всех нерегулярных точек

$(n, l) = 0$ (13)

образует на поверхности рассматриваемого объекта гладкую кривую, называемую контурной линией. Эта линия разбивает поверхность выпуклого тела на две части, каждая из которых однозначно проектируется на картинную плоскость и целиком состоит из регулярных точек. Одна из этих частей является полностью видимой, а другая - полностью невидимой.

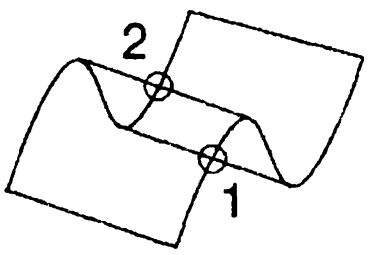
Попытаемся обобщить этот подход на случай одного или нескольких невыпуклых тел.

Множество всех контурных линий (их уже может быть несколько) разбивает границы тел на набор связных частей (компонент), каждая из которых по-прежнему взаимнооднозначно проектируется на картинную плоскость и состоит либо из лицевых, либо из нелицевых точек.

Никакая из этих частей не может закрывать себя при проектировании, однако возможны случаи, когда одна такая часть закрывает другую. Чтобы это учесть, введем числовую характеристику невидимости - так называемую количественную невидимость точки, определив ее как количество точек, закрывающих при проектировании данную точку. Точка оказывается видимой только в том случае, когда ее количественная невидимость равна нулю.

Количественная невидимость является кусочно-постоянной функцией и может изменять свое значение лишь в тех точках, проекции которых на картинную плоскость лежат в проекции одной из контурных линий.

Итак, проекции контурных линий разбивают картинную плоскость на области, каждая из которых является проекцией части объекта, а сами поверхности, ограничивающие тела, разбиваются контурными линиями на однозначно проектирующиеся фрагменты с постоянной количественной невидимостью.



В общем случае при проектировании гладких поверхностей возникает два основных типа особенностей (все остальные возможные особенности могут быть приведены к ним сколь угодно малыми "шевелениями") - линии складки, являющиеся регулярными проекциями контурных линий на картинную плоскость и представляющие собой регулярные кривые на поверхности, взаимнооднозначно проектирующиеся на картинную плоскость (рис. 10.21), и изолированные точки сборки (рис. 10.22), которые лежат на линиях складки (контурных линиях) и являются особыми точками проектирования контурных линий на картинную плоскость.

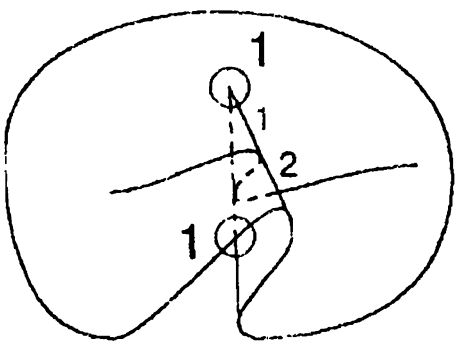


Рис. 10.22

Рассмотрим теперь изменение количественной невидимости вдоль самой контурной линии.

Можно показать, что она может измениться только в двух случаях - при прохождении позади контурной линии и в точках сборки.

В первом случае происходит загораживание складкой другого участка поверхности и количественная невидимость изменяется на два.

Во втором случае происходит загораживание поверхностью самой себя (рис. 10.22) и количественная невидимость изменяется на единицу.

Таким образом, для определения видимости достаточно найти контурные линии и их проекциями разбить всю картинную плоскость на области, являющиеся видимыми частями проекций объектов сцены.

В результате мы приходим к следующему алгоритму: на границах тел выделяется множество контурных линий *C*. Каждая из этих линий разбивается на части в тех

точках, где она закрывается при проектировании на картинную плоскость какой-либо линией этого множества, проходящей в точке закрывания ближе к картинной плоскости. Контурные линии разбиваются и в точками сборки. В результате получается множество линий, на каждой из которых количественная невидимость одна и та же (постоянна).

В случае, когда рассматриваются поверхности, не являющиеся границами сплошных тел, к множеству контурных линий *C* следует добавить и граничные линии этих поверхностей.

Если рассматриваемые объекты являются лишь кусочно-гладкими, то к множеству линий *C* следует добавить также линии излома (линии, где происходит потеря гладкости).

Данный метод может быть применен и для решения задачи удаления невидимых поверхностей - получившиеся линии разбивают картинную плоскость на области, каждая из которых соответствует видимой части одного из объектов сцены.

Его с успехом можно применить и для работы с полигональными объектами. Одним из вариантов такого применения является алгоритм Аппеля.

Аппель вводит количественную невидимость (quontative invisibility) точки как число лицевых граней, ее закрывающих. Это несколько отличается от определения, введенного ранее, однако существо подхода остается неизменным.

Контурная линия полигонального объекта состоит из тех ребер, для которых одна из проходящих граней является лицевой, а другая - нелицевой.

Так, для многогранника на рис. 10.23 контурной линией является ломаная ABCIJDEKLGA.

Рассмотрим, как меняется количественная невидимость вдоль ребра.

Для определения видимости ребер произвольного многогранника сначала берется какая-либо его вершина и ее количественная невидимость определяется непосредственно.

Далее прослеживается изменение количественной невидимости вдоль каждого из ребер, выходящих из этой вершины.

Эти ребра проверяются на прохождение позади контурной линии, и их количественная невидимость в соответствующих точках изменяется. При прохождении ребра позади контурной линии количественная невидимость точек ребра изменяется на единицу. Те части отрезка, для которых количественная невидимость равна нулю, сразу же рисуются.

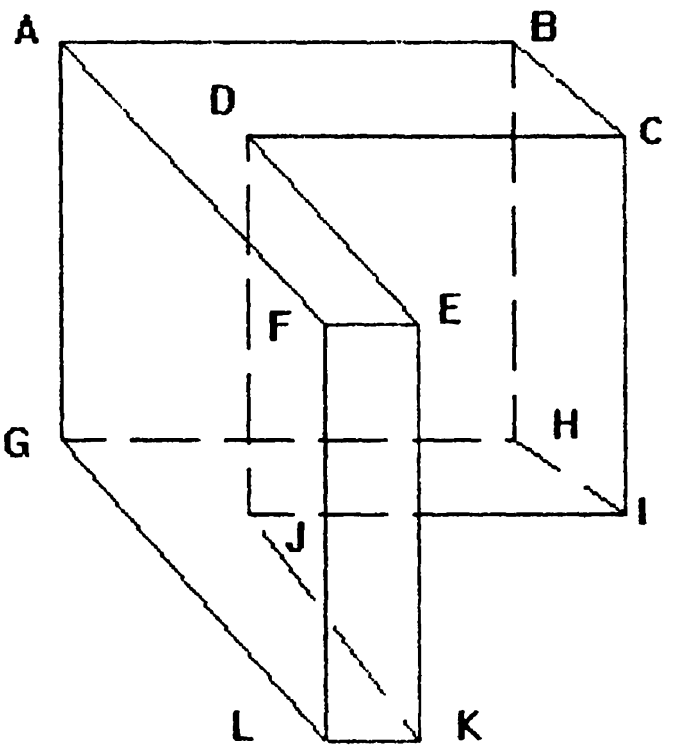


Рис. 10.23

Следующим шагом является определение количественной невидимости для ребер, выходящих из новой вершины, и т. д.

В результате определяется количественная невидимость всех ребер связной компоненты сцены, содержащей исходную вершину (и при этом видимые части ребер этой компоненты сразу же рисуются).

В случае, когда рассматривается изменение количественной невидимости вдоль ребра, выходящего из вершины, принадлежащей контурной линии, необходимо проверить, не закрывается ли это ребро одной из граней, выходящей из этой вершины (как, например, грань DEKJ закрывает ребро DJ, и это является аналогом точки сборки).

Так как для реальных объектов количество ребер, входящих в контурную линию, намного меньше общего числа ребер (если общее количество ребер равно n, то количество ребер, входящих в контурную линию, - $O(\sqrt{n})$), алгоритм Аппеля является более эффективным, чем алгоритм Робертса.

На поверхности многогранников можно выделить набор линий такой, что каждая контурная линия независимо от направления проектирования обязательно пересечет хотя бы одну из линий этого набора. Таким образом, для отыскания контурных линий необязательно перебирать все ребра - достаточно проверить заданный набор и восстановить контурные линии от точек пересечения с этим набором.

Замечание. Для повышения эффективности данного алгоритма возможно использование разбиения картинной плоскости - для каждой клетки разбиения строится список ребер контурной линии, чьи проекции пересекают данную клетку.

10.4. Удаление невидимых граней

Задача удаления невидимых граней является заметно более сложной, чем задача удаления невидимых линий, хотя бы по общему объему возникающей информации. Если практически все методы, служащие для удаления невидимых линий, работают в объектном пространстве и дают точный результат, то для удаления невидимых поверхностей существует большое число методов, работающих только в картинной плоскости, а также смешанных методов.

10.4.1. Метод трассировки лучей

Наиболее естественным методом для определения видимости граней является метод трассировки лучей (вариант, используемый только для определения видимости, без отслеживания отраженных и преломленных лучей обычно называется ray casting), при котором для каждого пиксела картинной плоскости определяется ближайшая к нему грань, для чего через этот пиксел выпускается луч, находятся все точки его пересечения с гранями и среди них выбирается ближайшая.

Данный алгоритм можно представить следующим образом:

```
for all pixels
  for all objects
    compare z
```

Одним из преимуществ этого метода является простота, универсальность (он может легко работать не только с полигональными моделями; возможно использование Constructive Solid Geometry) и возможность совмещения определения видимости с расчетом цвета пиксела.

Еще одним несомненным плюсом метода является большое количество методов оптимизации, позволяющих работать с сотнями тысяч граней и обеспечивающих временные затраты порядка $O(C\log n)$, где C - общее количество пикселов на экране, а n - общее количество объектов в сцене. Более того, существуют методы, обеспечивающие практическую независимость временных затрат от количества объектов.

10.4.2. Метод z-буфера

Одним из самых простых алгоритмов удаления невидимых граней и поверхностей является метод z-буфера (буфера глубины), где для каждого пиксела, как и в методе трассировки лучей, находится грань, ближайшая к нему вдоль направления проектирования, однако здесь циклы по пикселам и по объектам меняются местами:

```
for all objects
   for all covered pixels
      compare z
```

Поставим в соответствие каждому пикселу (x, y) картинной плоскости кроме цвета $c(x, y)$, хранящегося в видеопамяти, его расстояние до картинной плоскости вдоль направления проектирования $z(x, y)$ (его глубину).

Массив глубин инициализируется $+\infty$.

Для вывода на картинную плоскость произвольной грани она переводится в растровое представление на картинной плоскости и затем для каждого пиксела этой грани находится его глубина. В случае, если эта глубина меньше значения глубины, хранящегося в z-буфере, пиксел рисуется и его глубина заносится в z-буфер.

```
int   * zBuf = NULL;            // z-buffer, for 32-bit models

void     initZBuf ()
{
     zBuf = (int *)malloc (SCREEN_WIDTH*SCREEN_HEIGHT*
                                          sizeof(int));

     int * ptr = zBuf;
     for ( int i = 0; i < SCREEN_WIDTH * SCREEN_HEIGHT; i++ )
          * ptr++ = MAXINT;
}
void     writePixel ( int x, int y ,int z, int c )
{
     int * ptr = zBuf + x + y*SCREEN_WIDTH;

     if ( * ptr > z )
     {
          * ptr = z;
            putpixel ( x, y, c );
     }
}
```

Весьма эффективным является совмещение растровой развертки грани с выводом в z-буфер. При этом для вычисления глубины пикселов могут применяться инкрементальные методы, требующие всего нескольких сложений на пиксел.

Грань рисуется последовательно строка за строкой; для нахождения необходимых значений используется линейная интерполяция (рис. 10.24)

$$x_a = x_1 + (x_2 - x_1)\frac{y - y_1}{y_2 - y_1},$$

$$x_b = x_1 + (x_3 - x_1)\frac{y - y_1}{y_3 - y_1},$$

$$z_a = z_1 + (z_2 - z_1)\frac{y - y_1}{y_2 - y_1},$$

$$z_b = z_1 + (z_3 - z_1)\frac{y - y_1}{y_3 - y_1},$$

$$z = z_a + (z_b - z_a)\frac{x - x_a}{x_b - x_a}.$$

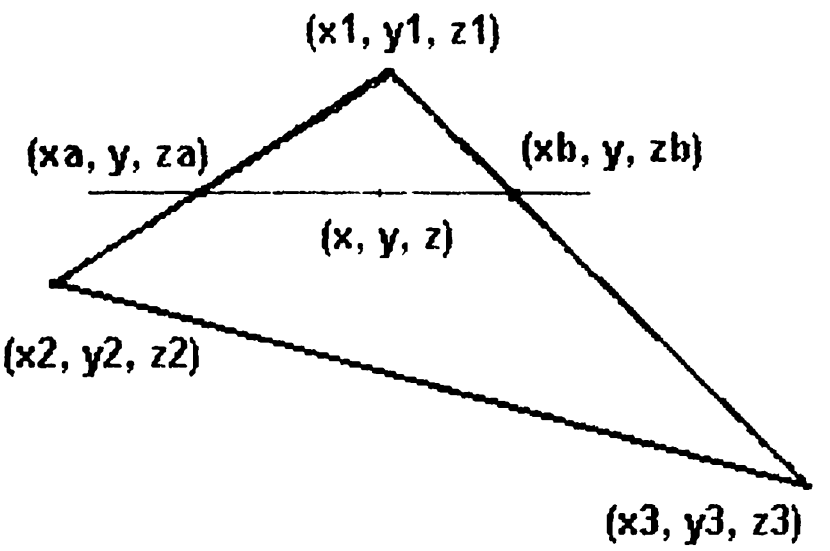


Рис. 10.24

Ниже приводится пример программы, осуществляющей вывод строки пикселов методом *z*-буфера; для вычисления глубин соседних пикселов используются рекуррентные соотношения.

Одним из преимуществ программы является то, что она работает исключительно в целых числах. Однако, так как глубины промежуточных точек могут быть нецелыми числами, для их представления используем то обстоятельство, что и шаг между глубинами соседних пикселов, и сами эти глубины являются рациональными числами вида

$$\frac{k}{x_2 - x_1}$$

Каждое такое число можно разбить на две части - целую и дробную:

$$\left[\frac{k}{x_2 - x_1}\right] + \frac{k \bmod (x_2 - x_1)}{x_2 - x_1},$$

что позволяет представлять его как два целых числа, одно из которых - целая часть числа, а другое - дробная часть, умноженная на знаменатель $x_2 - x_1$. Число, соответствующее дробной части, всегда находится в диапазоне между 0 и $x_2 - x_1 - 1$.

При сложении двух таких чисел их целые и дробные части складываются отдельно. Если дробная часть выходит из диапазона, то проводится коррекция целой и дробной частей.

```
// draw a single line
zf  = 0;       // fractional part of current z
dx  = x2 - x1;
dz  = ( z2 - z1 ) / dx;
dzf = ( z2 - z1 ) % dx;
ptr = Zbuf + y * SCREEN_WIDTH + x1;

for ( int x = x1; x <= x2; x++, ptr++ )
{
```

```
        if ( z < * ptr )
        {
          putpixel ( x, y, c );
          * ptr = z;
        }
        z  += dz;
        zf += dzf;
        if ( zf >= dx )
        {
          z++;
          zf -= dx;
        }
        else
        if ( zf < 0 )
        {
          z--;
          zf += dx;
        }
    }
```

Фактически метод *z*-буфера осуществляет поразрядную сортировку по *x* и *y*, а затем сортировку по *z*, требуя всего одного сравнения для каждого пиксела каждой грани.

Метод *z*-буфера работает исключительно в пространстве картинной плоскости и не требует никакой предварительной обработки данных. Порядок, в котором грани выводятся на экран, не играет никакой роли.

Для экономии памяти можно отрисовывать не все изображение сразу, а рисовать по частям. Для этого картинная плоскость разбивается на части (обычно это горизонтальные полосы) и каждая такая часть обрабатывается независимо. Размер памяти под буфер определяется размером наибольшей из этих частей.

Большинство современных графических станций содержат в себе графические платы с аппаратной реализацией *z*-буфера, зачастую включая и аппаратную "растеризацию" (т. е. преобразование изображения из координатного представления в растровое) граней вместе с закрашиванием Гуро. Подобные карты обеспечивают очень высокую скорость рендеринга вплоть до нескольких миллионов граней в секунду (Voodoo – 1,5 млн треугольников в секунду, Voodoo2 - до 3 млн треугольников в секунду, 90-180 мегапикселов в сек).

Средние временные затраты составляют $O(n)$, где n- общее количество граней.

Одним из основных недостатков *z*-буфера (помимо большого объема требуемой под буфер памяти) является избыточность вычислений: осуществляется вывод всех граней вне зависимости от того, видны они или нет. И если, например, данный пиксел накрывается десятью различными лицевыми гранями, то для каждого соответствующего пиксела каждой из этих десяти граней необходимо произвести расчет цвета. При использовании сложных моделей освещенности (например, модели Фонга) и текстур эти вычисления могут потребовать слишком больших временных затрат.

Рассмотрим в качестве примера модель здания с комнатами и всем, находящимся внутри их. Общее количество граней в подобной модели может составлять сотни тысяч и миллионы. Однако, находясь внутри одной из комнат этого здания, наблю-

датель реально видит только весьма небольшую часть граней (несколько тысяч). Поэтому вывод всех граней является непозволительной тратой времени.

Существует несколько модификаций метода *z*-буфера, позволяющих заметно сократить количество выводимых граней. Одним из наиболее мощных и элегантных является метод **иерархического *z*-буфера**.

Метод иерархического *z*-буфера использует сразу все три типа когерентности в сцене - в картинной плоскости (*z*-буфере), в пространстве объектов и временную когерентность.

Назовем грань скрытой (невидимой) по отношению к *z*-буферу, если для любого пиксела картинной плоскости, накрываемого этой гранью, глубина соответствующего пиксела грани не меньше значения в *z*-буфере. Ясно, что выводить скрытые грани не имеет смысла, так как они ничего не изменяют (они заведомо не являются видимыми).

Куб (прямоугольный параллелепипед) назовем скрытым по отношению к *z*-буферу, если все его лицевые грани являются скрытыми по отношению к этому *z*-буферу (рис. 10.25).

Опишем куб вокруг некоторой группы граней. Перед выводом граней, содержащихся внутри этого куба, стоит проверить, не является ли куб скрытым. Если он скрытый, то все грани, содержащиеся внутри его, можно сразу же отбрасывать.

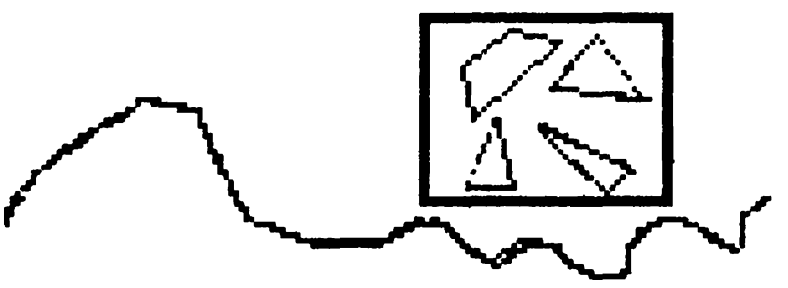


Рис. 10.25

Но даже если куб не является скрытым, часть граней, содержащихся внутри его, все равно может быть скрытой, и поэтому нужно разбить его на части и проверить каждую из частей на скрытость.

Предложенный подход приводит к следующему алгоритму.

Опишем куб вокруг всей сцены. Разобьем его на 8 равных частей. Каждую из частей, содержащую достаточно много граней, снова разобьем и т. д. Если число граней внутри частичного куба меньше заданного числа, то разбивать его нет смысла и он становится листом строящегося дерева. В результате получается восьмеричное дерево, где с каждым кубом связан список граней, содержащихся внутри его.

Вывод всей сцены можно представить следующим образом: очередной куб, начиная с корня дерева, проверяется на попадание в область видимости. Если куб в область видимости не попал, то ни одна грань из него не видна и мы его отбрасываем. В противном случае проверяем, не является ли этот куб скрытым. Скрытый куб также отбрасывается. В противном случае повторяем описанную процедуру для всех его восьми частей в порядке удаления - первым обрабатывается ближайший подкуб, последним - самый дальний. Для куба, являющегося листом, вместо вывода его частей просто выводим все содержащиеся в нем грани.

Такой подход опирается на когерентность в объектном пространстве и позволяет легко отбросить основную часть невидимых граней.

Для облегчения проверки грани на скрытость можно использовать *z*-пирамиду. Ее нижним уровнем является сам *z*-буфер. Для построения следующего уровня пикселы объединяются в группы по 4 (2 x 2) и из их глубин выбирается наибольшая. Таким образом, следующий уровень оказывается тоже буфером, но его размер уже

будет меньше исходного в 2 раза по каждому измерению. Аналогично строятся и остальные уровни пирамиды до тех пор, пока мы не придем к уровню, состоящему из единственного пиксела, являющегося вершиной z-пирамиды (рис. 10.26).

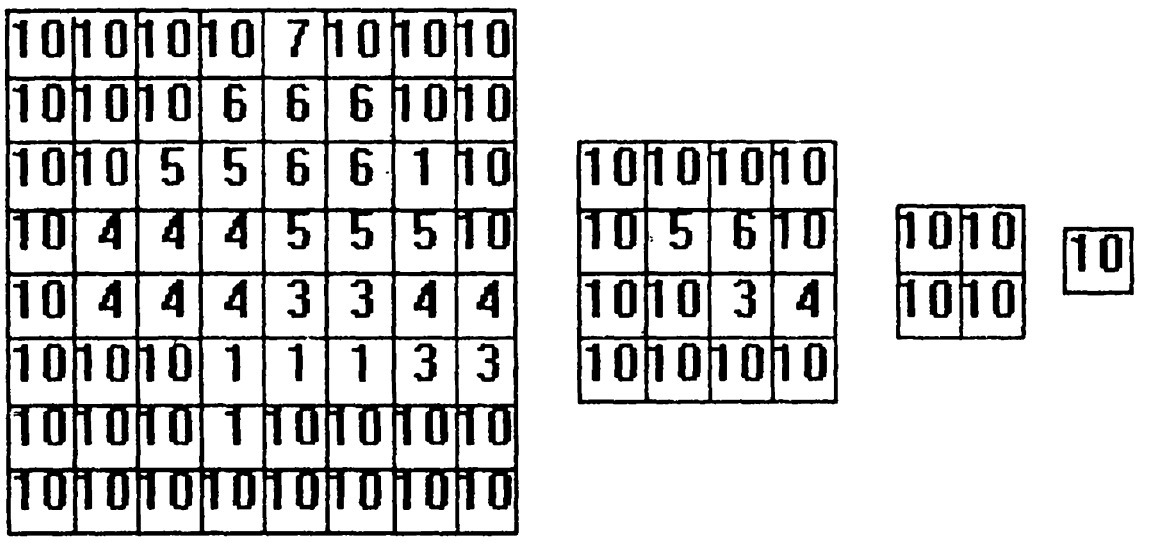


Рис. 10.26

Первым шагом проверки грани на скрытость будет сравнение ее минимальной глубины со значением в вершине z-пирамиды. Если минимальная глубина грани оказывается больше, то грань скрыта. В противном случае грань разбивается на 4 части и сравнение производится на следующем уровне пирамиды. Если ни на одном из промежуточных уровней скрытость грани установить не удалось, то осуществляется переход к последнему уровню, на котором грань растеризуется, и производится попиксельное сравнение с z-буфером. Наиболее простой является проверка на вершине пирамиды, наиболее трудоемкой - проверка в ее основании.

Применение z-пирамиды позволяет использовать когерентность в картинной плоскости - соседние пикселы скорее всего соответствуют одной и той же грани: следовательно, значения глубин в них отличаются мало.

Ясно, что чем раньше видимая грань будет выведена, тем больше невидимых граней будет отброшено сразу же. Высказанное соображение позволяет использовать когерентность по времени. Для этого ведется список тех граней, которые были видны в данном кадре, и рендеринг следующего кадра начинается с вывода именно этих граней (чтобы избежать их повторного вывода, они помечаются как уже выведенные). Только после этого осуществляется рендеринг всего дерева.

При использовании перспективного проектирования значения глубины, соответствующие пикселам одной грани, изменяются уже нелинейно, в то же время величина $1/z$ изменяется линейно и поэтому возникает понятие w-буфера, в котором вместо величины z хранится изменяющаяся линейно величина $1/z$.

Существует модификация метода z-буфера, позволяющая работать с прозрачными объектами и использовать CSG-объекты - для каждого пиксела (x, y) вместо пары (c, z) хранится упорядоченный по z список (C, z, t, ptr), где t - степень прозрачности объекта, а ptr - указатель на объект, и сначала строится буфер, затем для CSG-объектов осуществляется их раскрытие (см. метод трассировки лучей) и с учетом прозрачности рассчитываются цвета.

10.4.3. Алгоритмы упорядочения

Подход, использованный ранее для построения графика функции двух переменных и основанный на последовательном выводе на экран в определенном порядке всех граней, может быть успешно использован и для построения более сложных сцен.

Подобный алгоритм можно описать следующим образом

```
sort objects by z
for all objects
   for all visible pixels
      paint
```

Тем самым методы упорядочения выносят сравнение по глубине за пределы циклов и производят сортировку граней явным образом.

Методы упорядочения являются гибридными методами, осуществляющими сравнение и разбиение граней в объектном пространстве, а для непосредственного наложения одной грани на другую использующими растровые свойства дисплея.

Упорядочим все лицевые грани таким образом, чтобы при их выводе в этом порядке получалось корректное изображение сцены. Для этого необходимо, чтобы для любых двух граней P и Q та из них, которая при выводе может закрывать другую, выводилась позже. Такое упорядочение обычно называется back-to-front, поскольку сначала выводятся более далекие грани, а затем более близкие.

Существуют различные методы построения подобного упорядочения. Вместе с тем нередки и случаи, когда заданные грани упорядочить нельзя (рис. 10.27). Тогда необходимо произвести дополнительное разбиение граней так, чтобы получившееся после разбиения множество граней уже можно было упорядочить.

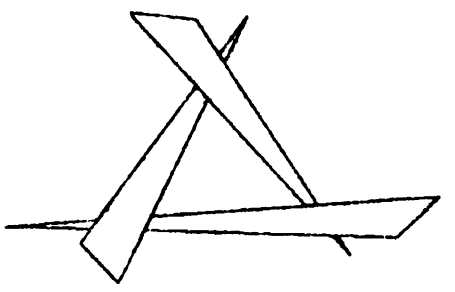


Рис. 10.27

Заметим, что две любые выпуклые грани, не имеющие общих внутренних точек, можно упорядочить всегда. Для невыпуклых граней это в общем случае неверно (рис. 10.28).

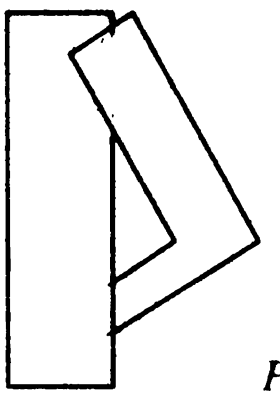


Рис. 10.28

10.4.3.1. Метод сортировки по глубине. Алгоритм художника

Этот метод является самым простым из методов, основанных на упорядочении граней. Как художник сначала рисует более далекие объекты, а затем поверх них более близкие, так и метод сортировки по глубине сначала упорядочивает грани по мере приближения к наблюдателю, а затем выводит их в этом порядке.

Метод основывается на следующем простом наблюдении: если для двух граней A и B самая дальняя точка грани A ближе к наблюдателю (картинной плоскости), чем самая ближняя точка грани B, то грань B никак не может закрыть грань A от наблюдателя.

Поэтому если заранее известно, что для любых двух лицевых граней ближайшая точка одной из них находится дальше, чем самая дальняя точка другой, то для упорядочения граней достаточно просто отсортировать их по расстоянию от наблюдателя (картинной плоскости).

Однако такое не всегда возможно: могут встретиться такие пары граней, что самая дальняя точка одной находится к наблюдателю не ближе, чем самая близкая точка другой.

На практике часто встречается следующая реализация этого алгоритма: множество всех лицевых граней сортируется по ближайшему расстоянию до картинной плоскости (наблюдателя) и потом эти грани выводятся в порядке приближения к наблюдателю. В качестве алгоритмов сортировки можно использовать либо быструю сортировку, либо поразрядную (radix sort).

Метод хорошо работает для целого ряда типичных сцен, включая, например, построение изображения нескольких непересекающихся простых тел.

Приведенная ниже программа осуществляет построение изображения тора на основе этого метода. Тор, описываемый уравнениями

$x = (R + r\cos\phi)\cos\varphi,$

$y = (R + r\sin\phi)\cos\varphi,$

$z = r\sin\phi,$

представляется в виде набора треугольных граней. После этого для сортировки граней по расстоянию до наблюдателя используется стандартная процедура qsort.

Расстояние от точки p до наблюдателя, расположенного в начале координат, при параллельном проектировании вдоль единичного вектора l задается следующей формулой: $d = (p, l)$.

```
// File Torus.cpp
#include    <conio.h>
#include    <graphics.h>
#include    <math.h>
#include    <process.h>
#include    <stdio.h>
#include    <stdlib.h>

#include    "Vector3D.h"
#include    "Matrix.h"

#define N1  40
#define N2  20

Vector  prjDir ( 0, 0, 1 );
Vector  vertex [N1*N2];
Matrix  trans = RotateX ( M_PI / 4 );

struct   Point              // screen point
{
    int  x, y;
};

struct   Facet
{
    int   index [3];
    Vector3D n;
    float     depth;
    float     coeff;
} * torus, * tmp;

int  facetComp ( const void * v1, const void * v2 )
{
    Facet * f1 = (Facet *) v1,
    Facet * f2 = (Facet *) v2;
```

```
    if ( f1 -> depth < f2 -> depth )
        return -1;
    else
    if ( f1 -> depth > f2 -> depth )
        return 1;
    else
        return 0;
}

void    initTorus ()
{
    float    r1 = 5;
    float    r2 = 1;

    // 1. Create vertices
    for ( int i = 0, k = 0; i < N1; i++ )
    {
       float  phi = i * 2 * M_PI / N1;

       for ( int j = 0; j < N2; j++, k++ )
       {
         float     psi = j * 2 * M_PI / N2;

         vertex [k].x = ( r1 + r2 * cos (psi) ) * cos (phi);
         vertex [k].y = ( r1 + r2 * cos (psi) ) * sin (phi);
         vertex [k].z = r2 * sin ( psi );

         vertex [k] = trans * vertex [k];
         vertex [k].z += 10;
       }
    }

    // 2. Create facets
    for ( i = k = 0; i < N1; i++ )
       for ( int j = 0; j < N2; j++, k += 2 )
       {
         torus [k].index [0] = i*N2 + j;
         torus [k].index [1] = ((i+1)%N1)*N2 + j;
         torus [k].index [2] = ((i+1)%N1)*N2 + (j+1)%N2;

         torus [k+1].index [0] = i*N2 + j;
         torus [k+1].index [1] = ((i+1)%N1)*N2 + (j+1)%N2;
         torus [k+1].index [2] = i*N2 + (j + 1)%N2;
       }
}

void    drawTorus ()
{
    // compute normals & distances
    for ( int i = 0, count = 0; i < N1*N2*2; i++ )
    {
       torus [i].n    = ( vertex [torus [i].index [1]] - vertex [torus [i].index [0] ] ) ^
                        ( vertex [torus [i].index [2]] - vertex [torus [i].index [1] ] );

       torus [i].coeff = ( torus [i].n & prjDir )/!torus [i].n;
       torus [i].depth = vertex [torus [i].index [0]] & prjDir;
```

```
    for ( int j = 1; j < 3; j++ )
    {
      float     d = vertex [torus [i].index [j]] & prjDir;
      if ( d < torus [i].depth )
           torus [i].depth = d;
    }
    if ( torus [i].coeff > 0 )
      tmp [count++] = torus [i];
}
// sort them
qsort ( tmp, count, sizeof ( Facet ), facetComp );
Point    edges [3];
// draw them
for ( i = 0; i < count; i++ )
{
    for ( int k = 0; k < 3; k++ )
    {
      edges [k].x = 320 + 30*vertex [tmp [i].index [k]].x;
      edges [k].y = 240 + 30*vertex [tmp [i].index [k]].y;
    }
    int     color = 64 + (int)( 20 + 43 * tmp [i].coeff );
    setfillstyle ( SOLID_FILL, color );
    setcolor     ( color );
    fillpoly     ( 3, (int far *) edges );
}
}
main ()
{
      int  driver;
      int  mode = 2;
      int  res;
      if ( ( driver = installuserdriver ( "VESA", NULL ) ) == grError )
      {
           printf ( "\nCannot load extended driver" );
           exit ( 1 );
      }
      initgraph ( &driver, &mode, "" );
      if ( ( res = graphresult () ) != grOk )
      {
           printf("\nGraphics error: %s\n", grapherrormsg(res));
           exit ( 1 );
      }
      for ( int i = 0; i < 64; i++ )
           setrgbpalette ( 64 + i, i, i, i );
      torus = new Facet [N1*N2*2];
      tmp   = new Facet [N1*N2*2];
      initTorus ();
      drawTorus ();
```

```
        getch ();
        closegraph ();

        delete tmp;
        delete torus;
    }
```

Хотя подобный подход и работает в подавляющем большинстве случаев, однако возможны ситуации, когда просто сортировка по расстоянию до картинной плоскости не обеспечивает правильного упорядочения граней (рис. 10.29), - так, грань *B* будет ошибочно выведена раньше, чем грань *A*; поэтому после сортировки желательно проверить порядок, в котором грани будут выводиться.

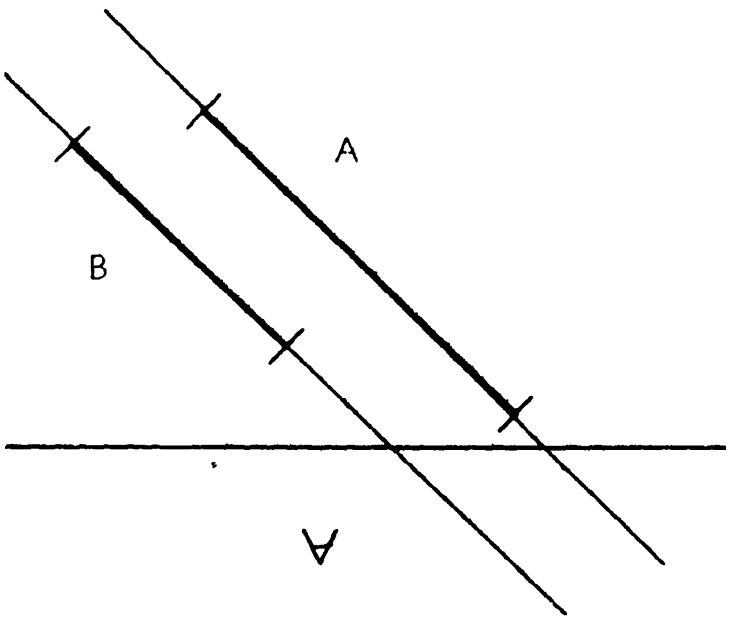


Рис. 10.29

Предлагается следующий алгоритм этой проверки. Для простоты будем считать, что рассматривается параллельное проектирование вдоль оси *Oz*.

Перед выводом очередной грани *P* следует убедиться, что никакая другая грань *Q*, которая стоит в списке позже, чем *P*, и проекция которой на ось *Oz* пересекается с проекцией грани *P* (если пересечения нет, то порядок вывода *P* и *Q* определен однозначно), не может закрываться гранью *P*. В этом случае грань *P* действительно должна быть выведена раньше, чем грань *Q*.

Ниже приведены 4 теста в порядке возрастания сложности проверки:

1. Пересекаются ли проекции этих граней на ось *Ox*?
2. Пересекаются ли проекции этих граней на ось *Oy*?
   Если хотя бы на один из этих двух вопросов получен отрицательный ответ, то проекции граней *P* и *Q* на картинную плоскость не пересекаются и, следовательно, порядок, в котором они выводятся, не имеет значения. Поэтому будем считать, что грани *P* и *Q* упорядочены верно.
   Для проверки выполнения этих условий очень удобно использовать ограничивающие тела.
   В случае, когда оба эти теста дали утвердительный ответ, проводятся следующие тесты.
3. Находятся ли грань *P* и наблюдатель по разные стороны от плоскости, проходящей через грань *Q* (рис. 10.30)?
4. Находятся ли грань *Q* и наблюдатель по одну сторону от плоскости, проходящей через грань *P*, (рис. 10.31)?

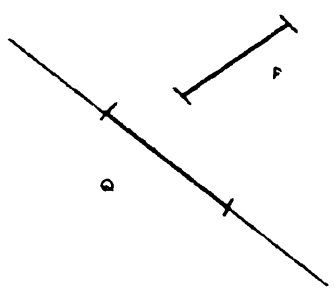


Рис. 10.30

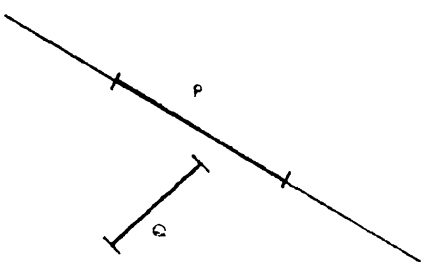


Рис. 10.31

Если хотя бы на один из этих вопросов получен утвердительный ответ, то считаем, что грани P и Q упорядочены верно, и сравниваем P со следующей гранью.

В случае, если ни один из тестов не подтвердил правильность упорядочения граней P и Q, проверяем, не следует ли поменять эти грани местами. Для этого проводятся тесты, являющиеся аналогами тестов 3 и 4 (очевидно, что снова проводить тесты 1 и 2 не имеет смысла):

- 3'. Находятся ли грань Q и наблюдатель по разные стороны от плоскости, проходящей через грань P?
- 4'. Находятся ли грань P и наблюдатель по одну сторону от плоскости, проходящей через грань Q?

В случае, если ни один из тестов 3, 4, 3', 4' не позволяет с уверенностью определить, какую из этих двух граней нужно выводить раньше, одна из них разбивается плоскостью, проходящей через другую грань и вопрос об упорядочении целой грани и частей разбитой грани легко решается при помощи тестов 3 или 4 (3' или 4').

Возможны ситуации, когда несмотря на то, что грани P и Q упорядочены верно, их разбиение все же будет произведено (алгоритм создает избыточные разбиения). Подобный случай изображен на рис. 10.32, где для каждой вершины указана ее глубина.

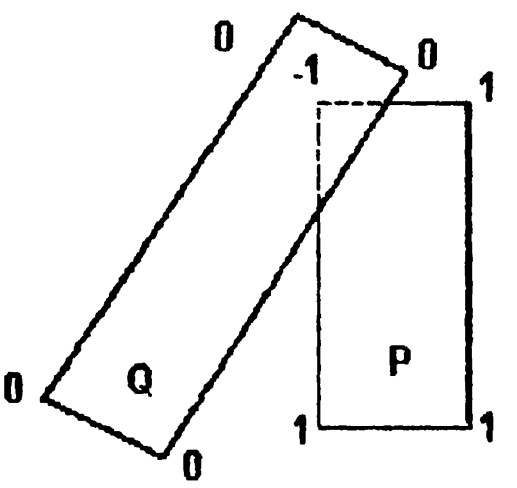


Рис. 10.32

Полную реализацию описанного алгоритма (сортировка и разбиение граней) для случая, когда сцена состоит из набора треугольных граней, можно найти в [15].

Методу упорядочения присущ тот же недостаток, что и методу z-буфера, а именно необходимость вывода всех лицевых граней. Чтобы избежать этого, можно его модифицировать следующим образом: грани выводятся в обратном порядке - начиная с самых близких и заканчивая самыми далекими (front-to-back). При выводе очередной грани рисуются только те пикселы, которые еще не были выведены. Как только весь экран будет заполнен, вывод граней можно прекратить.

Но здесь нужен механизм отслеживания того, какие пикселы были выведены, а какие нет. Для этого могут быть использованы самые разнообразные структуры, от линий горизонта до битовых масок.

Замечание. Рассмотрим подробнее, каким именно образом осуществляются проверки тестов 3 и 4.

Пусть грань P задана набором вершин

A_i, $i = 1, \ldots, n$,

а грань Q - набором вершин

B_j, $j = 1, \ldots, m$.

Тогда для определения плоскости, проходящей через грань P, достаточно знать вектор нормали n к этой грани и какую-либо ее точку, например A_1. Уравнение плоскости, проходящей через грань P, имеет следующий вид:

$(n, r) - (A_1, n) = 0$,

где вектор нормали задается формулой

$n = [A_2 - A_1, A_3 - A_2]$.

Грань *Q* лежит по ту же стороны от плоскости, проходящей через грань P, по которую находится и наблюдатель, расположенный в точке *V*, если

$$\text{sign}\{(n, B_i) - (n, A_1)\} = \text{sign}\{(n, \theta) - (n, A_1)\}, i = 1,..., m,$$

и по другую сторону, если

$$\text{sign}\{(n, B_i) - (n, A_1)\} = -\text{sign}\{(n, \theta) - (n, A_1)\}, i = 1,..., m.$$

10.4.3.2. Метод двоичного разбиения пространства

Существует другой, довольно элегантный и гибкий способ упорядочения граней.

Каждая плоскость в объектном пространстве разбивает все пространство на два полупространства. Считая, что эта плоскость не пересекает ни одну из граней сцены, получаем разбиение множества всех граней на два непересекающихся множества (кластера); каждая грань попадает в тот или иной кластер в зависимости от того, в каком полупространстве относительно плоскости разбиения эта грань находится.

Ясно, что ни одна из граней, лежащих в полупространстве, не содержащем наблюдателя, не может закрывать собой ни одну из граней, лежащих в том же полупространстве, в котором находится и наблюдатель (с небольшими изменениями это работает и для параллельного проектирования).

Для построения правильного изображения сцены необходимо сначала выводить грани из дальнего кластера, а затем из ближнего.

Применим предложенный подход для упорядочения граней внутри каждого кластера. Для этого выберем две плоскости, разбивающие каждый из кластеров на два подкластера.

Повторяя описанный процесс до тех пор, пока в каждом получившемся кластере останется не более одной грани (рис. 10.33).

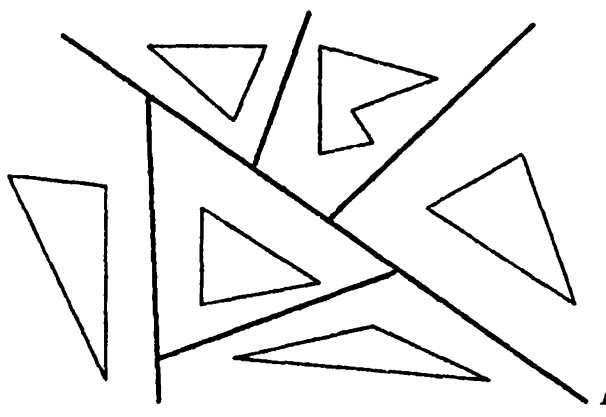


Рис. 10.33

Получаем в результате двоичное дерево (Binary Space Partitioning Tree). Узлами этого дерева являются плоскости, производящие разбиение.Пусть плоскость, производящая разбиение, задается уравнением

$(p, n) = d.$

Каждый узел дерева можно представить в виде следующей структуры

```
struct BSPNode
{
  Facet   *     facet;   // corresponding facet
  Vector3D      n;       // normal to facet ( plane )
  float         d;       // plane parameter
  BSPNode *     left;    // left subtree
  BSPNode *     right;   // right subtree
};
```

left указывает на вершину поддерева, содержащуюся в положительном полупространстве $(p, n) > d$, **right** - на поддерево, содержащееся в отрицательном полупространстве $(p, n) < d$

Обычно в качестве разбивающей плоскости выбирается плоскость, проходящая через одну из граней. Все грани, пересекаемые этой плоскостью, разбиваются вдоль

нее, а получившиеся при разбиении части помещаются в соответствующие поддеревья.

Рассмотрим сцену, представленную на рис. 10.34. Плоскость, проходящая через грань 5, разбивает грани 2 и 8 на части 2', 2", 8' и 8", и все множество граней (с учетом разбиения граней 2 и 8) распадается на два кластера (1,8',2') и (2", 3, 4, 6, 7, 8"). Выбрав для первого кластера в качестве разбивающей плоскости плоскость, проходящую через грань 6, разбиваем его на два подкластера (7,8") и (2",3,4). Каждое следующее разбиение будет лишь выделять по одной грани из оставшихся кластеров. В результате получим следующее дерево (рис. 10.35).

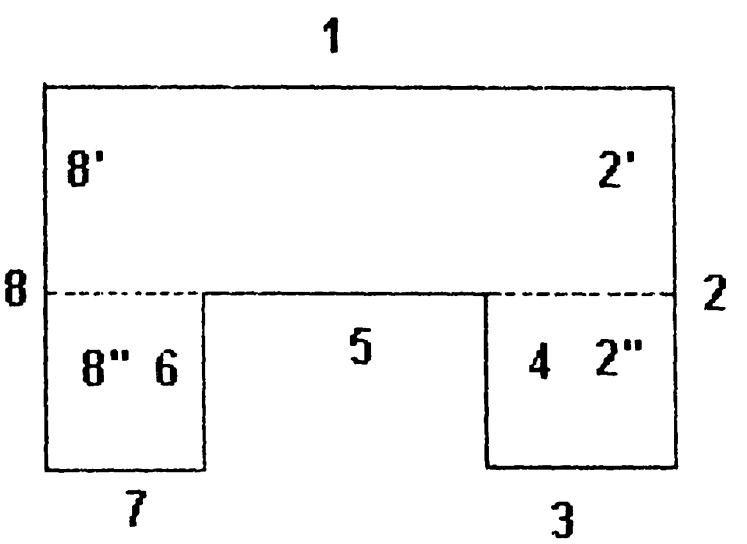


Рис. 10.34

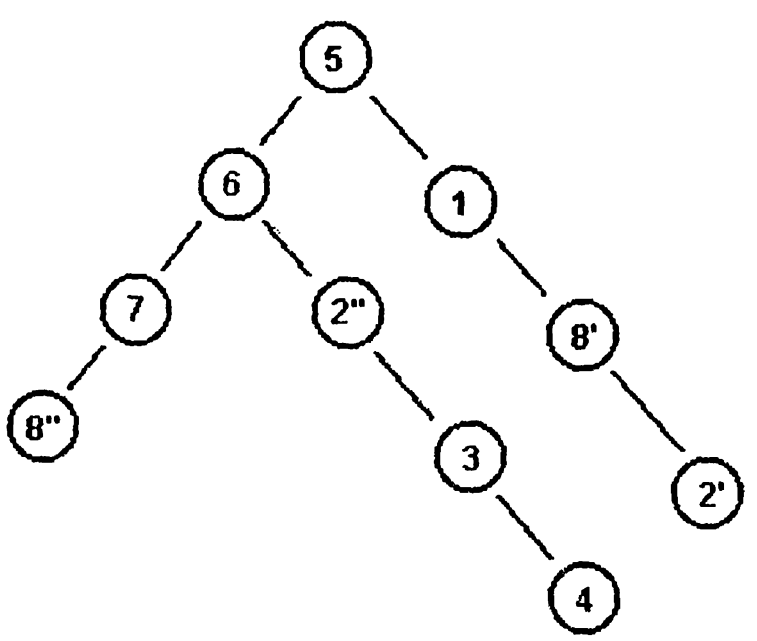


Рис. 10.35

Таким образом, процесс построения BSP-деревьев заключается в выборе разбивающей плоскости (грани), разбиении множества всех граней на две части (это может потребовать разбиения граней на части) и рекурсивного применения описанной процедуры к каждой из получившихся частей.

Замечание. Если проверяемая грань лежит в плоскости разбиения, то ее можно поместить в любую из частей. Существуют варианты метода, которые с каждым узлом дерева связывают список граней, лежащих в разбивающей плоскости.

Алгоритм построения BSP-дерева очень похож на известный метод быстрой сортировки Хоара и реализуется следующим фрагментом кода.

```
BSPNode * buildBSPTree ( const Array& facets )
{
    BSPNode   *   root = new BSPNode;        // create root node
    Facet     *   part = (Facet *) facets [0]; // use 1st facet
    Array         left, right;
    Facet     *   f1, *f2;

    root -> facet = part;
    root -> n = part -> n;
    root -> d = part -> n & part -> p [0];

    for ( int i = 1; i < facets.getCount (); i++ )
    {
        Facet   * f = (Facet *) facets [i];

        switch ( classifyFacet ( root, f ) )
        {
```

```
            case IN_PLANE:          // can put in any part
            case IN_POSITIVE:       // facet lies in +
                left.insert ( f );
                break;
            case IN_NEGATIVE:       // facet lies in -
                right.insert ( f );
                break;
            default:                // split facet
                splitFacet ( root, f, &f1, &f2 );
                left.insert ( f1 );
                right.insert ( f2 );
                break;
        }
    }
    root->left  = (left. getCount ()>0  ? buildBSPTree (left)   NULL );
    root->right = (right. getCount ()>0 ? buildBSPTree (right)  NULL );

    return root;
}
```

При этом предполагается, что каждая грань описывается следующим классом

```
class   Facet
{
    int         count;              // # of vertices
    Vector3D    n;                  // normal to facet
    Vector3D    p [MAX_POINTS];     // list of vertices
    Facet ( Vector3D *, int );
};
```

Для построения очередного разбиения приведенная выше функция использует самую первую грань из переданного списка. На самом деле в качестве разбивающей грани можно выбирать любую грань.

Получающиеся деревья сильно зависят от выбора разбивающих граней. На рис. 10.36 приведено дерево для сцены с рис. 10.34 с другим порядком выбора разбивающих граней. Обратите внимание на отсутствие разбиения граней в этом случае.

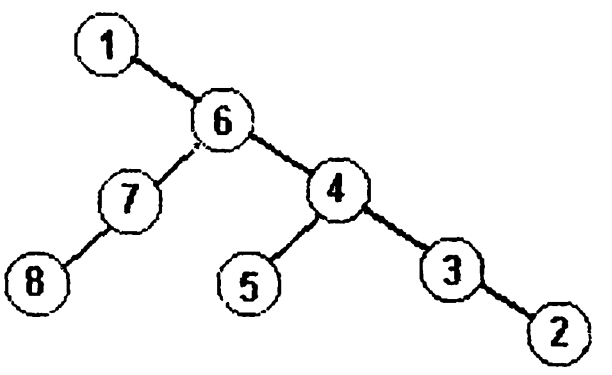


Рис. 10.36

Для классификации граней относительно плоскости можно воспользоваться следующей функцией (для учета ошибок округления в ней используется малая величина EPS).

```
int  classifyFacet (BSPNode * node, Facet& f )
{
    int  positive = 0;
    int  negative = 0;

    for ( int i = 0; i < f.count; i++ )
    {
        float     res = f.p [i] & node -> n - node -> d;
```

```
        if ( res > EPS )
            positive++;
        else
        if ( res < -EPS )
            negative++;
    }
    if ( positive > 0 && negative == 0 )
        return IN_POSITIVE;
    else
    if ( positive == 0 && negative > 0 )
        return IN_NEGATIVE;
    else
    if ( positive < 1 && negative < 1 )
        return IN_PLANE;
    else
        return IN_BOTH;
}
```

При разбиении грани плоскостью производится классификация всех вершин грани относительно этой плоскости и разбиение тех ребер, вершины которых лежат в разных полупространствах; сами точки разбиения считаются принадлежащими каждому из получившихся многоугольников (рис. 10.37).

Ниже приведена процедура разбиения грани плоскостью. Причем считается, что грань действительно разбивается плоскостью на две непустые части, и поэтому случай, когда какое-либо ребро грани лежит в плоскости разбиения, исключается в силу выпуклости граней.

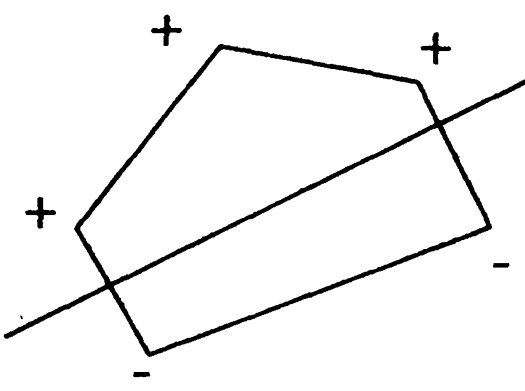


Рис. 10.37

```
void splitFacet(BSPNode * node, Facet& f, Facet **f1, Facet **f2)
{
    Vector3D   p1 [MAX_POINTS];
    Vector3D   p2 [MAX_POINTS];
    Vector     prevP  = f.p [f.count - 1];
    float      prevF  = prevP & node -> n - node -> d;
    int        count1 = 0;
    int        count2 = 0;

    for ( int i = 0; i < f.count; i++ )
    {
        Vector  curP = f.p [i];
        float   curF = curP & node -> n - node -> d;

        if ( curF >= 0 && prevF <= 0 )
            p1 [count1++] = curP;
        else
        if ( curF < 0 && prevF >= 0 )
            p2 [count2++] = curP;
        else
        if ( curF < 0 && prevF > 0 )
```

```
            {
                float      t  = - curF / ( prevF - curF );
                Vector3D sp =  curP + t * ( prevP - curP );

                p1 [count1++] = sp;
                p2 [count2++] = sp;
                p2 [count2++] = curP;
            }
            else
            if ( curF > 0 && prevF < 0 )
            {
                float      t  = - curF / ( prevF - curF );
                Vector3D sp =  curP + t * ( prevP - curP );

                p1 [count1++] = sp;
                p2 [count2++] = sp;
                p1 [count1++] = curP;
            }
            prevP = curP;
            prevF = curF;
        }

        * f1 = new Facet ( p1, count1 );
        * f2 = new Facet ( p2, count2 );
    }
```

Естественным образом возникает вопрос о построении дерева, в некотором смысле оптимального. Существует два основных критерия оптимальности:

- получение как можно более сбалансированного дерева (когда для любого узла количество граней в правом поддереве минимально отличается от количества граней в левом поддереве); это обеспечивает минимальную высоту дерева (и соответственно наименьшее количество проверок);
- минимизация количества разбиений; одним из наиболее неприятных свойств BSP-деревьев является разбиение граней, приводящее к значительному увеличению их общего числа и, как следствие, к росту затрат (памяти и времени) на изображение сцены.

К сожалению, эти критерии, как правило, являются взаимоисключающими. Поэтому обычно выбирается некоторый компромиссный вариант; например, в качестве критерия выбирается сумма высоты дерева и количества разбиений с заданными весами.

Полный анализ реальной сцены с целью построения оптимального дерева из-за очень большого количества вариантов произвести практически невозможно. Поэтому обычно поступают следующим образом: на каждом шаге разбиения случайным образом выбирается небольшое количество кандидатов на разбиение и выбор наилучшего в соответствии с выбранным критерием производится только среди этих кандидатов.

В случае, когда целью является минимизация числа разбиений, можно воспользоваться следующим приемом: на очередном шаге выбирать ту грань, использование которой приводит к минимальному числу разбиений на данном шаге.

После того, как это дерево построено, построение изображения осуществляется в зависимости от используемого проектирования. Ниже приводится процедура построения изображения для центрального проектирования с центром в точке *c*.

```
void  drawBSPTree ( BSPNode * tree )
{
    if ( ( tree -> n & c ) > tree -> d )
    {
        if ( tree -> right != NULL )
            drawBSPTree ( tree -> right );

        drawFacet ( tree -> facet );

        if ( tree -> left != NULL )
            drawBSPTree ( tree -> left );
    }
    else
    {
        if ( tree -> left != NULL )
            drawBSPTree ( tree -> left );

        drawFacet ( tree -> facet );

        if ( tree -> right != NULL )
            drawBSPTree ( tree -> right );
    }
}
```

Приведенная процедура осуществляет вывод граней в порядке back-to-front. Эту процедуру можно модифицировать для вывода только лицевых граней. Несложно также скорректировать ее и для работы с параллельным проектированием.

Если необходимо выводить грани в обратном порядке (front-to-back), то отработку левого и правого поддеревьев следует поменять местами. Но тогда потребуется механизм отслеживания уже заполненных пикселов экрана.

Как только все пикселы будут заполнены, рекурсивный обход дерева можно прекратить.

Одним из основных преимуществ этого метода является полная независимость дерева от параметров проектирования (положение центра проектирования, направление проектирования и др.), что делает его весьма удобным для построения серий изображений одной и той же сцены из разных точек наблюдения. Это обстоятельство привело к тому, что BSP-деревья стали широко использоваться в ряде систем виртуальной реальности. В частности, удаление невидимых граней в широко известных играх DOOM, Quake и Quake II основано на привлечении именно BSP-деревьев.

В ряде случаев удобнее строить дерево, листьями которого будут не грани, а выпуклые многогранники, - порядок, в котором выводятся лицевые грани выпуклых многогранников, может быть произволен и никакого влияния на конечный результат не оказывает. Причем BSP-дерево используется только для упорядочивания этих многогранников, а уже упорядочивание граней внутри каждого из них осуществляется другим способом.

К недостаткам метода BSP-деревьев относятся явно избыточная необходимость разбиения граней, особенно актуальная при работе с большими сценами, и нелокальность BSP-деревьев - даже незначительное локальное изменение сцены может повлечь за собой изменение практически всего дерева.

Вследствии их нелокальности представление больших сцен в виде BSP-деревьев оказывается слишком сложным, так как приводит к очень большому количеству разбиений. Для борьбы с этим явлением можно разделить всю сцену на несколько частей, которые можно легко упорядочить между собой, и для каждой из этих частей построить свое BSP-дерево, содержащее только ту часть сцены, которая попадает в данный фрагмент.

10.4.4. Метод построчного сканирования

Метод построчного сканирования является примером метода, удачно использующего растровые свойства картинной плоскости для упрощения исходной задачи и сведения ее к серии простых задач в пространстве меньшей размерности.

Все изображение на картинной плоскости (экране) можно представить как состоящее из горизонтальных (вертикальных) линий пикселов (строк или столбцов). Каждой такой строке пикселов соответствует сечение сцены плоскостью, проходящей через соответствующую строку и наблюдателя (для параллельного проектирования - проходящей через строку и параллельную направлению проектирования), при наших допущениях.

Пересечением секущей плоскости со сценой будет множество непересекающихся (за исключением концов) отрезков, высекаемых на гранях секущей плоскостью (рис. 10.38).

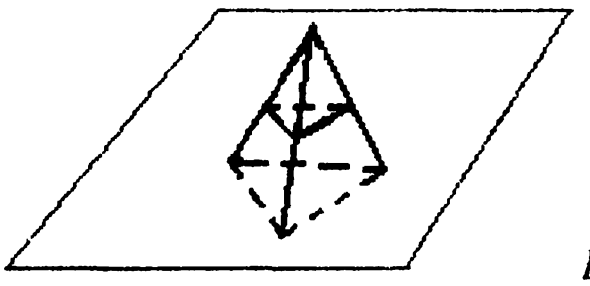


Рис. 10.38

В результате мы приходим к задаче удаления невидимых частей для отрезков на секущей плоскости при проектировании на прямую, являющуюся результатом пересечения с ней картинной плоскости. Тем самым получается задача с размерностью на единицу меньше, чем исходная задача, - вместо определения того, какие части граней закрывают друг друга при проектировании на плоскость, необходимо определить, какие части отрезков закрывают друг друга при проектировании на прямую

```
sort objects by y
sort object by x
for all x
  compare z
```

Существуют различные методы решения задачи удаления невидимых частей отрезков. Одним из наиболее простых является использование одномерного z-буфера, совмещающего крайнюю простоту с весьма небольшими затратами памяти даже при высоком разрешении картинной плоскости. К тому же существуют аппаратные реализации этого подхода. С другой стороны, для определения видимых частей можно воспользоваться и аналитическими (непрерывными) методами.

Заметим, что изменение видимости отрезков может происходить лишь в их концах. Поэтому достаточно проанализировать взаимное расположение концов отрезков с учетом глубины. Один из вариантов такого подхода использует специальные таблицы для отслеживания концов отрезков:

- таблица ребер (Edge Table), где для каждого негоризонтального ребра (горизонтальные ребра игнорируются) хранятся минимальная и максимальная y-координаты, x-координата, соответствующая вершине с наименьшей y-координатой,

шаг изменения x при переходе к следующей строке и ссылка на соответствующую грань;

- таблица граней (Facet Table), где для каждой грани помимо информации о плоскости, проходящей через эту грань, и информации, необходимой для ее закрашивания, хранится также специальный флажок, устанавливаемый в нуль при обработке очередной строки;
- таблица активных ребер (Active Edge Table), содержащая список всех ребер, пересекаемых текущей сканирующей плоскостью, и проекции точек пересечения (x-координаты при параллельном проектировании). Все ребра в таблице активных ребер сортируются по возрастанию x.

Для удобства определения ребер, пересекаемых текущей сканирующей плоскостью, список всех ребер обычно сортируется по наименьшей y-координате.

Для граней, представленных на рис. 10.39, таблица активных ребер выглядит следующим образом:

y_1	AB, AC
y_2	AB, AC, FD, FE
y_3	AB, DE, BC, FE
y_4	AB, DE, BC, FE
y_5	AB, BC, DE, FE

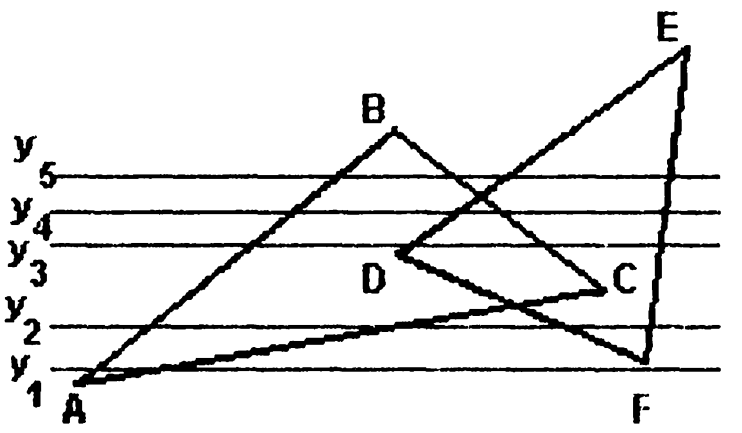


Рис. 10.39

Ребра из списка активных ребер обрабатываются по мере увеличения x.

При обработке линии y_1 таблица активных ребер состоит только из двух ребер - *AB* и *AC*. Первым обрабатывается ребро *AB*, при этом флаг соответствующей грани (*ABC*) инвертируется. Тем самым мы "входим" в эту грань, т. е. следующая группа пикселов является проекцией этой грани.

Поскольку в данной строке пересекается лишь одна грань, то очевидно, что она является видимой и, значит, весь отрезок между точками пересечения секущей плоскости с ребрами *AB* и *BC* необходимо закрасить в цвета этой грани. При обработке следующего отрезка флаг грани ABC снова инвертируется и становится равным нулю - мы выходим из этой грани.

Строка y_2 обрабатывается аналогичным образом (проекции граней для данной строки не пересекаются).

Обрабатывая строку y_3, мы сталкиваемся с перекрывающимися гранями. При прохождении через ребро *AB* флаг грани *ABC* инвертируется, и отрезок до пересечения со следующим ребром (*DF*) соответствует этой грани. При прохождении ребра *DF* флаг грани *DEF* также инвертируется и мы оказываемся сразу в двух гранях (обе грани проектируются на этот участок). Чтобы определить, какая из них видна, необходимо сравнить глубины обеих граней в этой точке; в данном случае грань *DEF* ближе и поэтому отрезок закрашивается цветом этой грани. При прохождении через ребро *BC* флаг грани *ABC* сбрасывается в нуль и мы опять оказываемся внутри только одной грани. Поэтому следующий отрезок до пересечения с ребром *EF* будет принадлежать грани *DEF*.

В общем случае при прохождении через вершину и инвертировании флага соответствующей грани проверяются все грани, для которых установлен флаг, и среди них выбирается ближайшая.

Грани не имеют внутренних пересечений, поэтому при выходе из невидимой грани нет смысла проверять видимость, так как порядок граней измениться не мог - при пересечении ребра BC видимой по-прежнему остается грань DEF (рис. 10.40).

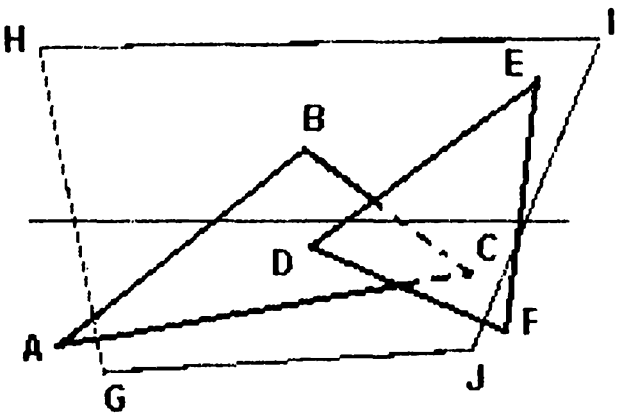


Рис. 10.40

Подобный подход существенно использует связность пикселов внутри строки (фактически анализ проводится только для точек пересечения ребер с секущей плоскостью) - видимость устанавливается сразу для целых групп пикселов. Можно также использовать когерентность пикселов и между отдельными строками. Самым простым вариантом этого является применение инкрементальных подходов для определения проекций точек пересечения секущей плоскости с ребрами (так как проекцией отрезка всегда является отрезок, эти точки отличаются сдвигом даже для перспективной проекции).

Весьма интересным является подход, основанный на следующем наблюдении: если таблица активных ребер содержит те же ребра и в том же порядке, что и для предыдущей строки, то никаких относительных изменений видимости между гранями не происходит и, следовательно, нет никакой необходимости для проверок глубины. Сказанное справедливо для строк y_3 и y_4 предыдущего примера.

Аналогично используется когерентность между соседними гранями одного тела. Заметим, что изменение видимости отрезков может происходить при переходе только через те концы отрезков, которые соответствуют ребрам, принадлежащим сразу лицевой и нелицевой граням. Тем самым возникает некоторое (сравнительно небольшое) множество концов отрезков, являющееся аналогом контурных линий, и проверку на изменение видимости можно производить только при переходе через такие точки.

Для ускорения метода применяются различные формы разбиения пространства, включая равномерное разбиение пространства, деревья и т. д. Грани обрабатываются по мере удаления, и обработки заведомо невидимых граней можно избежать.

Вариантом метода построчного сканирования является так называемый **метод *s*-буфера**.

Ключевыми элементами метода *s*-буфера являются:

- массив списков отрезков для строк экрана,
- менеджер отрезков,
- процедура вставки.

Для каждой строки экрана поддерживается список отрезков граней, задающий фактически для каждого пиксела видимую в этом пикселе грань.

Процесс заполнения массива происходит построчно. Изначально, для очередной строки, список отрезков пуст. Далее для каждой грани, пересекаемой сканирующей

плоскостью, строится соответствующий отрезок, который затем вставляется в список отрезков данной строки.

Ключевым местом алгоритма является процедура вставки отрезка в список. При этом отрезок может быть усечен, если он виден лишь частично, или вообще отброшен, если он не виден целиком.

Процедура вставки фактически сравнивает данный отрезок со всеми отрезками, уже присутствующими в списке. Ниже приводится пример простейшей процедуры вставки отрезка в упорядоченный список отрезков текущей строки.

```
//File sbuffer.h
//
// SBuffer management routines
//
#ifndef __S_BUFFER__
#define __S_BUFFER__

#include    <alloc.h>                   // for NULL
#include    "polygon.h"

struct      Span                        // span of S-Buffer
{
    float   x1, x2;                     // endpoints of the span, as floats to
// speed up calculations
    float   invZ1, invZ2;               // 1/z values for endpoints
    float   k, b;                       // coefficients of equation: 1/z=k*x+b

    Span  * next;                       // next span
Span  * prev;                           // previous span;
Polygon3D * facet;                      // facet it belongs to
float     f ( float x, float invZ ) const
{
    return k * x + b - invZ;
}
                // clip the span at the begining
void      clipAtX1  ( float newX1 )
{
    invZ1 = k * newX1 + b;
    x1    = newX1;
}
                // clip the span at the end
void      clipAtX2 ( float newX2 )
{
    invZ2 = k * newX2 + b;
    x2    = newX2;
}
                // precompute k and b coefficients
void      precompute ()
{
    if ( x1 != x2 )
    {
        k = (invZ2 - invZ1) / (x2 - x1);
        b = invZ1 - k * x1;
```

```
        }
        else
        {
            k = 0;
            b = invZ1;
        }
    }
                // whether the span is empty
    int  isEmpty ()
    {
        return x1 > x2;
    }
                // insert current span before s
    void    insertBefore ( Span * s )
    {
        prev                = s -> prev;
        s -> prev -> next   = this;
        s -> prev           = this;
        next                = s;
    }
                // insert current span after s
    void    insertAfter ( Span * s )
    {
        if ( s -> next != NULL )
            s -> next -> prev = this;
        prev        = s;
        next        = s -> next;
        s -> next   = this;
    }
                // remove current span from list
    void    remove ()
    {
        prev -> next = next;
        if ( next != NULL )
            next -> prev = prev;
    }
};
class   SpanPool
{
public:
    Span  *   pool;             // pool of Span strctures
    int       poolSize;         // # of structures allocated
    Span  *   firstAvailable;   // pointer to 1st struct after
// all allocated
    Span  *   astAvailable;     // last item in the pool
    Span  *   free;             // pointer to free structs list
// (below firstAvailable)
    SpanPool ( int maxSize )
    {
        pool            = new Span [poolSize = maxSize];
        firstAvailable  = pool;
```

```
    lastAvailable  = pool + poolSize - 1;
    free           = NULL,
}
~SpanPool ()
{
    delete [] pool;
}
void    freeAll ()          // free all spans
{
    firstAvailable = pool;
    free           = NULL;
}
Span  * allocSpan ( Span * s = NULL )   // allocate free span
{                           // when no free spans
    if ( free == NULL )
        if ( firstAvailable <= lastAvailable )
        {
            firstAvailable -> next = NULL;
            firstAvailable -> prev = NULL;

            if ( s != NULL )
                * firstAvailable = * s;

            return firstAvailable++;
        }
        else
            return NULL;

        Span  * res = free;

        free = free -> next;

        res -> next = NULL;
        res -> prev = NULL;

        if ( s != NULL )
            * res = * s;

        return res;
    }
                    // deallocate span
// (return to free span list)
    void    freeSpan ( Span * span )
    {
        span -> next = free;
        free         = span;
    }
};
extern  SpanPool * pool;

class   SBuffer
{
public:
    Span ** head;
    int     screenHeight;
```

```
        SBuffer ( int height )
        {
            head = new Span * [screenHeight = height];
            reset ();
        }
        ~SBuffer ()
        {
            delete [] head;
        }
        void    reset ()
        {
            pool -> freeAll ();
            for ( register int i = 0; i < screenHeight; i++ )
            {
                head [i]          = pool -> allocSpan ();
                head [i] -> prev = NULL;
                head [i] -> next = NULL;
            }
        }
        void    addSpan ( int line, Span * span );
        void    addPoly ( const Polygon3D& poly );
};
int  compareSpans ( const Span * s1, const Span * s2 );
#endif
```

```
//File sbuffer.cpp
//
// SBuffer management routines
//
#include    <mem.h>
#include    "SBuffer.h"
SpanPool    * pool = NULL;
//
// compare spans s1 and s2, s1 being the inserted span
// return positive if s2 cannot obscure s1 (s1 is nearer)
//
int  compareSpans ( const Span * s1, const Span * s2 )
{
            // try to check max/min depths
    float   s1MinInvZ, s1MaxInvZ;
    float   s2MinInvZ, s2MaxInvZ;
            // compute min/max for 1/z for s1
    if ( s1 -> invZ1 < s1 -> invZ2 )
    {
        s1MinInvZ = s1 -> invZ1;
        s1MaxInvZ = s1 -> invZ2;
    }
    else
```

```
    {
        s1MinInvZ = s1 -> invZ2;
        s1MaxInvZ = s1 -> invZ1;
    }
            // compute min/max for 1/z for s2
    if ( s2 -> invZ1 < s2 -> invZ2 )
    {
        s2MinInvZ = s2 -> invZ1;
        s2MaxInvZ = s2 -> invZ2;
    }
    else
    {
        s2MinInvZ = s2 -> invZ2;
        s2MaxInvZ = s2 -> invZ1;
    }
            // compare inverse depths
    if ( s1MinInvZ >= s2MaxInvZ )
        return 1;
    if ( s2MinInvZ >= s1MaxInvZ )
        return -1;
            // if everything fails then try
// direct approach
    float    f1  = s1 -> f ( s2 -> x1, s2 -> invZ1 );
    float    f2  = s1 -> f ( s2 -> x2, s2 -> invZ2 );
    int  res = 0;
    if ( f1 <= 0 && f2 <= 0 )
        res = -1;
    else
    if ( f1 >= 0 && f2 >= 0 )
        res = 1;
    else
    {
        f1 = s2 -> f ( s1 -> x1, s1 -> invZ1 );
        f2 = s2 -> f ( s1 -> x2, s1 -> invZ2 );
        if ( f1 <= 0 && f2 <= 0 )
            res = 1;
        else
            res = -1;
    }
    return res;
}
////////////////// SBuffer methods ////////////////
void    SBuffer    addSpan ( int line, Span * newSpan )
{
   newSpan -> precompute ();
   Span * prevSpan = head [line];
   Span * curSpan  = prevSpan -> next;
```

```
for ( ; curSpan != NULL, )
{
  if ( curSpan -> x2 < newSpan -> x1 )
  {
     prevSpan = curSpan;
     curSpan  = curSpan -> next;

     continue;
  }
  if ( newSpan -> x1 < curSpan -> x1 )      // cases 1, 2 or 4
  {
     if ( newSpan -> x2 <= curSpan -> x1 )      // case 1
     {
       newSpan -> insertBefore ( curSpan );

       return;
     }
     if ( newSpan -> x2 < curSpan -> x2 )  // case 2
     {
       if ( compareSpans ( curSpan, newSpan ) < 0 )
          curSpan -> clipAtX1 ( newSpan -> x2 );
       else
          newSpan -> clipAtX2 ( curSpan -> x1 );

       if ( !newSpan -> isEmpty () )
          newSpan -> insertBefore ( curSpan );

       return;
     }
     else                        // case 4
     {
       if ( compareSpans ( curSpan, newSpan ) < 0 )
       {
          Span * tempSpan = curSpan -> next;

          curSpan -> remove ();           // remove from chain
          pool    -> freeSpan ( curSpan );
          curSpan  = tempSpan;

          continue;
       }
       else
       {
          Span * tempSpan = pool -> allocSpan ( newSpan );

          tempSpan -> insertBefore ( curSpan );
          tempSpan -> clipAtX2 ( curSpan -> x1 );
          newSpan  -> clipAtX1 ( curSpan -> x2 );
       }
     }
}
else          // curSpan -> x1 <= newSpan -> x1
{
   if ( newSpan -> x2 > curSpan -> x2 )  // case 5
   {
```

```
  if ( compareSpans ( curSpan, newSpan ) < 0 )
  {
     curSpan -> clipAtX2 ( newSpan -> x1 );
     if ( curSpan -> isEmpty () )
     {
       Span * tempSpan = curSpan -> next;
       curSpan -> remove ();
       pool    -> freeSpan ( curSpan );
       curSpan = tempSpan;
       continue;
     }
  }
  else
     newSpan -> clipAtX1 ( curSpan -> x2 );
}
else                          // case 3
{
  if ( compareSpans ( curSpan, newSpan ) < 0 )
  {
     Span * tempSpan = pool -> allocSpan ( curSpan );
     curSpan  -> clipAtX2    ( newSpan -> x1 );
     newSpan  -> insertAfter ( curSpan );
     tempSpan -> clipAtX1    ( newSpan -> x2 );
     if ( !tempSpan -> isEmpty () )
          tempSpan -> insertAfter ( newSpan );
        else
          pool -> freeSpan ( tempSpan );
        return;
      }
      else
        return;
    }
  }
  if ( newSpan -> isEmpty () )
       return;
  prevSpan = curSpan;
  curSpan  = curSpan -> next;
 }
 newSpan -> insertAfter ( prevSpan );
}
static   int  findEdge ( int& i, int dir, const Polygon3D& p )
{
  for ( ; ; )
  {
       int  i1 = i + dir;
       if ( i1 < 0 )
            i1 = p.numVertices - 1;
```

```
            else
            if ( i1 >= p.numVertices )
                i1 = 0;

            if ( p.vertices [i1].y < p.vertices [i].y )
                return -1;
            else
            if ( p.vertices [i1].y == p.vertices [i].y )
                i = i1;
            else
                return i1;
        }
    }

    void    SBuffer   addPoly ( const Polygon3D& poly )
    {
        int  yMin          = (int) poly.vertices [0].y;
        int  yMax          = (int) poly.vertices [0].y;
        int  topPointIndex = 0;

        for ( int i = 1; i < poly.numVertices; i++ )
            if ( poly.vertices [i].y <
                   poly.vertices [topPointIndex].y )
                topPointIndex = i;
            else
            if ( poly.vertices [i].y > yMax )
                yMax = (int) poly.vertices [i].y;

        yMin = (int) poly.vertices [topPointIndex].y;

        if ( yMin == yMax )       // degenerate polygon
        {
            int  iMin = 0;
            int  iMax = 0;

            for ( i = 1; i < poly.numVertices; i++ )
                if ( poly.vertices [i].x <
                       poly.vertices [iMin].x )
                    iMin = i;
                else
                if ( poly.vertices [i].x >
                       poly.vertices [iMax].x )
                    iMax = i;

            Span * span   = pool -> allocSpan ();

            span -> x1      = poly.vertices [iMin].x;
            span -> x2      = poly.vertices [iMax].x;
            span -> invZ1 = 1.0 / poly.vertices [iMin].z;
            span -> invZ2 = 1.0 / poly.vertices [iMax].z;
            span -> facet  = (Polygon3D *) &poly;

            addSpan ( yMin, span );

            return;
        }

        int  i1, i1Next;
```

```
int  i2, i2Next;

i1       = topPointIndex;
i1Next   = findEdge ( i1, -1, poly );
i2       = topPointIndex;
i2Next   = findEdge ( i2, 1, poly );

float    x1     = poly.vertices [i1].x;
float    x2     = poly.vertices [i2].x;
float    invZ1  = 1.0 / poly.vertices [i1].z;
float    invZ2  = 1.0 / poly.vertices [i2].z;
float    dx1    =
   (poly.vertices [i1Next].x - poly.vertices [i1].x) /
   (poly.vertices [i1Next].y - poly.vertices [i1].y);
float    dx2    =
   (poly.vertices [i2Next].x - poly.vertices [i2].x) /
   (poly.vertices [i2Next].y - poly.vertices [i2].y);
float    dInvZ1 =
   (1.0/poly.vertices [i1Next].z - 1.0/poly.vertices [i1].z) /
   (poly.vertices [i1Next].y - poly.vertices [i1].y);
float    dInvZ2 =
   (1.0/poly.vertices [i2Next].z - 1.0/poly.vertices [i2].z) /
   (poly.vertices [i2Next].y - poly.vertices [i2].y);
int  y1Next = (int) poly.vertices [i1Next].y;
int  y2Next = (int) poly.vertices [i2Next].y;

for ( int y = yMin; y <= yMax; y++ )
{
Span * span = pool -> allocSpan ();

if ( span == NULL )
     return;

span -> x1    = x1;
span -> invZ1 = invZ1;
span -> x2    = x2;
span -> invZ2 = invZ2;
span -> facet = (Polygon3D *) &poly;

addSpan ( y, span );

x1    += dx1;
x2    += dx2;
invZ1 += dInvZ1;
invZ2 += dInvZ2;

if ( y + 1 == y1Next )
{
   i1 = i1Next;
   if ( --i1Next < 0 )
     i1Next = poly.numVertices - 1;

   y1Next = (int) poly.vertices [i1Next].y;

   if (poly.vertices[i1].y>=poly.vertices[i1Next].y)
         break;

   dx1    =
```

```
                (poly.vertices [i1Next].x - poly.vertices [i1].x) /
                (poly.vertices [i1Next].y - poly.vertices [i1].y);
               dInvZ1 =
          (1.0/poly.vertices [i1Next].z - 1.0/poly.vertices [i1].z) /
          (poly.vertices [i1Next].y - poly.vertices [i1].y);
            }
            if ( y + 1 == y2Next )
            {
                i2 = i2Next;
                if ( ++i2Next >= poly.numVertices )
                  i2Next = 0;
                y2Next = (int) poly.vertices [i2Next].y;
                if(poly.vertices[i2].y>=poly.vertices[i2Next].y)
                      break;
                 dx2    =
                (poly.vertices [i2Next].x - poly.vertices [i2].x) /
                (poly.vertices [i2Next].y - poly.vertices [i2].y);
                 dInvZ2 =
          (1.0/poly.vertices [i2Next].z - 1.0/poly.vertices [i2].z) /
          (poly.vertices [i2Next].y - poly.vertices [i2].y);
            }
        }
    }
```

Здесь процедура compareSpans ($s1$, $s2$) сравнивает два отрезка на загораживание (заметим, что для любых двух непересекающихся отрезков на плоскости это можно сделать) и возвращает положительное значение, если отрезок $s2$ не может загораживать отрезок $s1$.

Рассмотрим, как работает процедура вставки нового отрезка newSpan в s-буфер.

Если отрезков в буфере нет, то новый отрезок просто добавляется. В противном случае сначала пропускаются все отрезки, лежащие левее нового отрезка (для которых $x2$ <= newSpan->$x1$).

Пусть curSpan указывает на первый отрезок, не удовлетворяющий этому условию. Тогда возможна одна из следующих ситуаций (рис. 10.41).

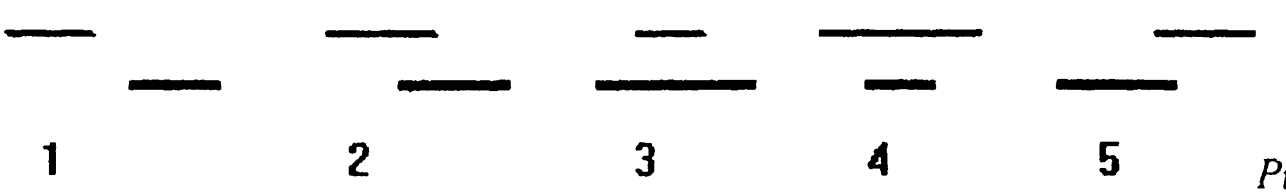


Рис. 10.41

Здесь жирным обозначен текущий отрезок (current). Тогда отрезки curSpan и newSpan сравниваются и каждый из возможных случаев отрабатывается.

Существуют и другие реализации метода s-буфера, использующие связность отрезков между собой, разбиение пространства или иерархические структуры для оптимизации процедуры вставки и отсечения заведомо невидимых отрезков.

Здесь в отличие от рассмотренного ранее метода построчного сканирования строка может быть отрисована только тогда, когда будут обработаны все ее отрезки. Таким образом, сначала строится полное разложение строки (экрана) на набор отрезков, соответствующих видимым граням, а затем уже происходит отрисовка.

Вместо списка отрезков можно использовать бинарное дерево.

Следует иметь в виду, что при работе с *s*-буфером необязательно работать только с одной строкой. Если для каждой строки завести свой указатель на список отрезков, то в *s*-буфер можно выводить по граням: грань раскладывается на набор отрезков и производится вставка каждого отрезка в соответствующий список. Тем самым в отличие от традиционного метода построчного сканирования здесь циклы по граням и по строкам меняются местами.

По аналогии с методом иерархического *z*-буфера можно рассмотреть метод иерархического *s*-буфера. Понятия невидимости грани и куба относительно *s*-буфера вводятся по аналогии, однако в этом случае производится уже не попиксельное сравнение, а сравнение на уровне отрезков. При этом отпадает необходимость в *z*-пирамиде; вместо нее для каждой строки *s*-буфера нужно запомнить максимальное значение глубины и использовать полученные значения для быстрого отсечения граней куба. Вся сцена представляется в виде восьмеричного дерева, а процедура вывода сцены носит рекурсивный характер и состоит в вызове соответствующей процедуры, где в качестве параметра выступает корень дерева.

Процедура вывода куба состоит из следующих шагов.

1. Проверка попадания куба в область видимости (если куб не попадает, то он сразу же отбрасывается).
2. Проверка видимости относительно текущего *s*-буфера (если куб невидим, то он сразу же отбрасывается).
3. Если куб является листом дерева, то последовательно выводятся все содержащиеся в нем лицевые грани. В противном случае рекурсивно обрабатываются все 8 его подкубов в порядке их удаления от наблюдателя.

Если все грани - выпуклые многоугольники, то для ускорения сравнения сегментов можно воспользоваться следующим соображением: так как любые две выпуклые грани, не имеющие общих внутренних точек, можно упорядочить, результаты сравнения соответствующих отрезков для двух произвольно взятых граней будут постоянными (зависящими только от расположения граней). Следовательно, результат сравнения отрезка текущей грани с отрезком в *s*-буфере можно кешировать для использования на следующих строках этой грани.

При построении серии кадров можно воспользоваться временной когерентностью. В этом случае запоминается список всех видимых в данном кадре граней и построение следующего кадра начинается с вывода этих граней.

В таком виде метод иерархического *s*-буфера удачно использует все три вида когерентности - в объектном пространстве (благодаря использованию восьмеричного дерева), в картинной плоскости (благодаря использованию *s*-буфера и кешированию результатов сравнения отрезков) и временную когерентность.

10.4.5. Алгоритм Варнака (Warnock)

Алгоритм Варнака является еще одним примером алгоритма, основанного на разбиении картинной плоскости на части, для каждой из которых исходная задача может быть решена достаточно просто.

Разобьем видимую часть картинной плоскости на четыре равные части и рассмотрим, каким образом могут соотноситься между собой проекции граней и получившиеся части картинной плоскости.

Возможны 4 различных случая:

1. Проекция грани полностью накрывает область (рис. 10.42, *а*);
2. Проекция грани пересекает область, но не содержится в ней полностью (рис. 10.42, *б*);
3. Проекция грани целиком содержится внутри области (рис. 10.42, *в*);
4. Проекция грани не имеет общих внутренних точек с рассматриваемой областью (рис. 10.42, *г*).

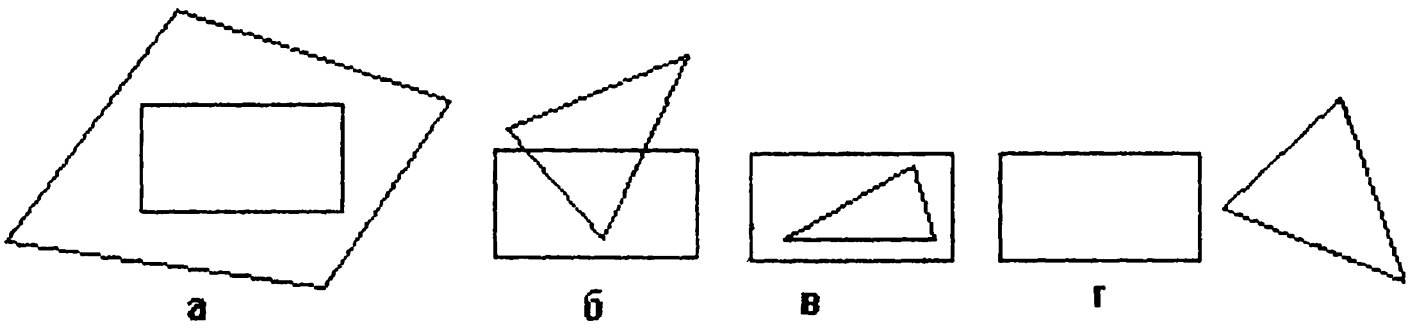


Рис. 10.42

Очевидно, что в последнем случае грань вообще никак не влияет на то, что видно в данной области.

Сравнивая область с проекциями всех граней, можно выделить случаи, когда изображение, получающееся в рассматриваемой области, определяется сразу:

- проекция ни одной грани не попадает в область;
- проекция только одной грани содержится в области или пересекает область; в этом случае грань разбивает всю область на две части, одна из которых соответствует этой грани;
- существует грань, проекция которой полностью накрывает данную область, и эта грань расположена к картинной плоскости ближе, чем все остальные грани, проекции которых пересекают данную область; в этом случае вся область соответствует этой грани.

Если ни один из рассмотренных трех случаев не имеет места, то снова разбиваем область на 4 равные части и проверяем выполнение этих условий для каждой из частей. Те части, для которых видимость таким образом определить не удалось, разбиваем снова и т. д. (рис. 10.43).

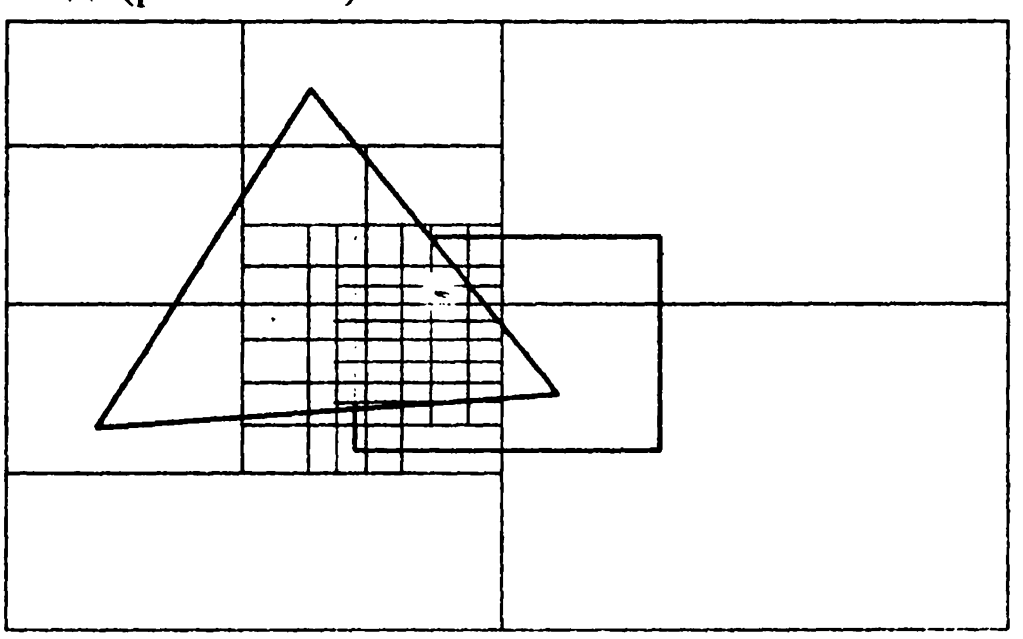


Рис. 10.43

Естественно возникает вопрос о критерии, на основании которого прекращать разбиение (иначе оно может продолжаться до бесконечности).

В качестве очевидного критерия можно взять размер области: как только размер области станет не больше размера 1 пиксела, то производить дальнейшее разбиение не имеет смысла и для данной области ближайшая к ней грань определяется явно.

10.4.6. Алгоритм Вейлера-Эйзертона (Weiler - Atherton)

Разбиение картинной плоскости можно производить не только прямыми, параллельными координатным осям, но и по границам проекций граней. В результате получается точное решение задачи.

Однако подобный подход требует эффективного способа построения пересечения (разбиения) граней (грани могут быть невыпуклыми и содержать "дыры").

Предлагаемый метод работает с проекциями граней на картинную плоскость.

В качестве первого шага производится сортировка всех граней по глубине (front-to-back).

Затем из списка оставшихся граней берется ближайшая грань A и все остальные грани обрезаются по этой грани; если проекция грани B пересекает проекцию грани A, то грань B разбивается на части так, что каждая часть либо содержится в грани A, либо не имеет с ней общих внутренних точек.

Таким образом, получаются два множества граней: F_{in} - грани, проекции которых содержатся в проекции грани A (сюда входит и сама грань A), и F_{out} - грани, проекции которых не имеют общих внутренних точек с проекцией грани A.

Множество F_{in} обычно называют множеством граней, внутренних по отношению к A.

Далее все грани из множества F_{in}, лежащие позади грани A, удаляются (грань A их полностью закрывает).

Однако в множестве F_{in} могут быть грани, лежащие к наблюдателю ближе, чем сама грань A (это возможно, например, при циклическом наложении граней). В этом случае каждая такая грань используется для рекурсивного разбиения всех граней из множества F_{in} (включая и исходную грань A). Когда рекурсивное разбиение завершится, то все грани из первого множества выводятся и из набора оставшихся граней выбрасываются (их уже ничто не может закрывать). Затем из набора оставшихся граней F_{out} берется очередная грань и процедура повторяется.

Для учета циклического наложения граней используется стек граней, по которым проводится разбиение. Когда какая-то грань оказывается ближе, чем текущая разбивающая грань, то сначала проверяется, нет ли уже этой грани в стеке. Если она там присутствует, то рекурсивного вызова не производится.

Рассмотрим простейший случай двух граней A и B (рис. 10.44).

Будем считать, что грань A расположена ближе, чем грань B. Тогда на первом шаге для разбиения используется именно грань A. Грань B разбивается на две части. Часть B_1 попадает в первое множество и, так как она лежит дальше грани A, удаляется.

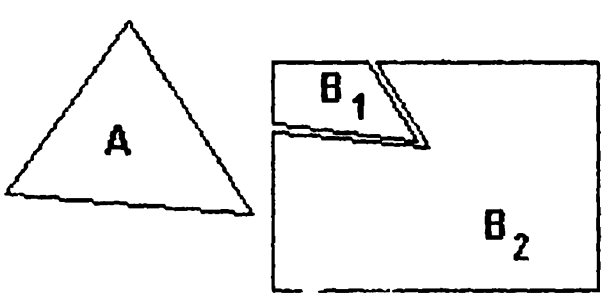


Рис. 10.44

После этого выводится грань A и в списке оставшихся граней остается только грань B_2. Так как кроме нее других граней не осталось, то эта грань выводится, и на этом работа завершается.

На рис. 10.45 приведена заметно более сложная сцена, требующая рекурсивного разбиения.

Предположим, что вначале грани отсортированы в порядке *A, B, C*.

На первом шаге берется грань *A* и все оставшиеся грани (*B* и *C*) разбиваются по границе проекции грани *A* (рис. 10.45, *а*). Грань *B* разбивается на части B_1 и B_2, а грань *C* - на части C_1 и C_2.

При этом $F_{in} = \{A, B_1, C_1\}$,

$F_{out} = \{B_2, C_2\}$.

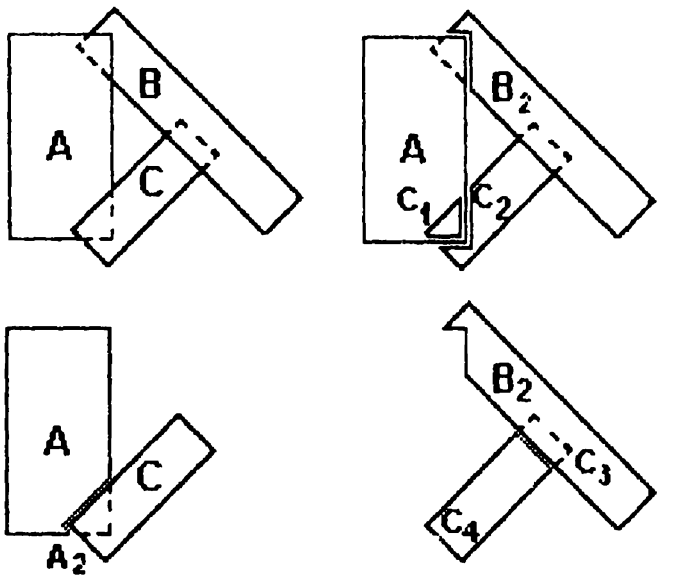


Рис. 10.45

Так как грань B_1 лежит дальше, чем грань *A*, то она отбрасывается. Взяв следующую грань из F_{in}, а именно грань C_1, мы обнаруживаем, что она лежит ближе к наблюдателю, чем грань *A*. В этом случае мы производим рекурсивное разбиение множества F_{in} при помощи грани C_1 (или *C*). В результате этого грань *A* разбивается на две части A_1 и A_2 и $F_{in} = \{C, A_2\}$. Так как грань A_2 лежит дальше, чем грань C_1, то она отбрасывается, а грань C_1 выводится. Исчерпав таким образом F_{in}, мы возвращаемся из рекурсии и получаем $F_{in} = \{A_1, C_1\}$. Эти части можно вывести сразу же, так как они не закрывают друг друга и никакая другая грань их закрывать не может.

Далее берем множество внешних граней F_{out}. В качестве грани, производящей разбиение, выберем грань B_2 и разобъем грань C_2 на две части - C_3 и C_4.

Получаем $F_{in} = \{B_2, C_3\}$, $F_{out} = \{C_4\}$. Так как грань C_3 лежит дальше, чем грань B_2, то она удаляется, а оставшаяся грань B_2 выводится.

На последнем шаге выводится грань C_4 из F_{out}, так как она, очевидно, ничем закрываться не может.

Таким образом видимыми являются грани A_1, B_2 и C_4 (рис. 10.46).

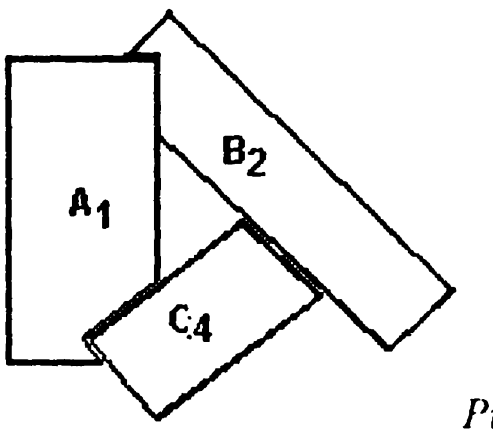


Рис. 10.46

10.5. Специальные методы оптимизации

Часто встречается имеющая свою специфику задача визуализации внутренности архитектурных сооружений (помещения, здания, лабиринты и т. п.). Одной из основных особенностей этой задачи является то, что при очень большом общем числе граней количество граней, видимых из данной точки, обычно оказывается сравнительно малым.

Поэтому существуют специальные методы, основанные на как можно более раннем отбрасывании большинства заведомо невидимых граней.

10.5.1. Потенциально видимые множества граней

В подобных задачах сцену можно достаточно легко разбить на набор выпуклых областей, например комнат, и для каждой из таких областей составить список тех граней, которые могут быть видны из данной области. Подобный список называется

PVS (Potentially Visible Set) и для каждой из областей на этапе препроцессинга обычно строится заранее. PVS позволяет получить достаточно быструю визуализацию постоянной сцены из различных точек наблюдения, хотя и требует больших затрат на этапе препроцессирования.

Для удаления невидимых граней из PVS и их частей используется один из традиционных методов удаления невидимых поверхностей (например, метод z-буфера).

10.5.2. Метод порталов

Существует подход, позволяющий строить PVS прямо на ходу. Разобьем сцену на набор выпуклых областей и рассмотрим, как эти области соединены между собой. Те соединения, через которые можно видеть (окна, дверные проемы), называются порталами.

Ясно, что все грани, принадлежащие той ячейке, в которой находится наблюдатель, могут быть видны и поэтому автоматически попадают в PVS. Рассмотрим порталы, соединяющие данную ячейку с соседними. Если какие-то грани и могут быть видны, то только через эти порталы. Поэтому выделим области, соединенные с текущей областью порталами, и в них те грани, которые видны через соединяющие их порталы. Далее для областей, соседних с начальной, рассмотрим соседние области. Они также могут быть видны только через соединяющие порталы. Поэтому выделим те грани, которые могут быть видны (теперь уже через два последовательных портала), и т. д. Подобным путем можно легко построить некоторое множество граней, потенциально видимых из данной точки. Возможно, этот список окажется несколько избыточным, но тем не менее он будет заметно меньше, чем общее число граней.

Рассмотрим сцену, представленную на рис. 10.47. Порталы обозначены пунктирными линиями.

Пусть наблюдатель находится в комнате 0-1-2-27-28. Очевидно, что он видит все лицевые грани в этой комнате. Кроме того, через портал 3-26 он видит комнату 3-4-25-26, а через портал 4-25 - комнату 4-5-6-7-16-17-24-25 и т. д.

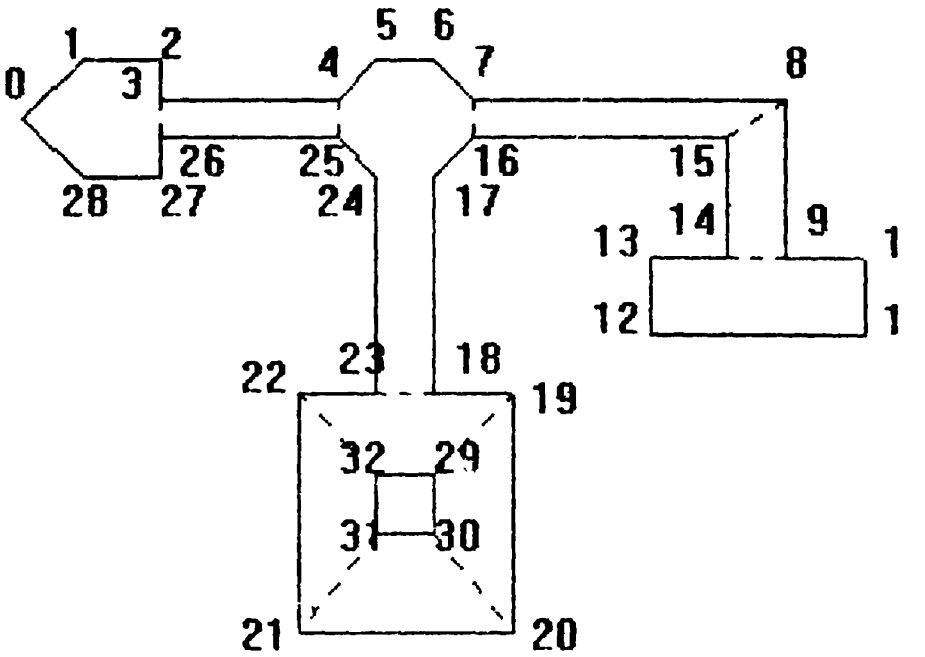


Рис. 10.47

Сначала достаточно нарисовать лицевые грани из текущей комнаты, затем для каждого портала, принадлежащего этой комнате, нужно нарисовать видимые сквозь портал части лицевых граней смежных комнат, используя при этом соответствующий портал как область отсечения.

Рассмотрим комнаты, соединенные порталами с комнатами, соседними с начальной, и нарисуем те их лицевые грани, которые видны через суперпозицию (пересечение) сразу двух порталов, ведущих в соответствующую комнату, и т. д. Если пересечение порталов пустое множество, то данная комната из начальной точки, где находится наблюдатель, не видна.

Получившийся алгоритм реализует следующий фрагмент псевдокода:

```
void renderScene ( const Polygon clippingPoly, Cell activeCell )
{
   for each wall of activeCell
      if ( wall intersects viewingFrustrum )
         {
          Polygon clippedWall = clipPolygon ( wall, clippingPoly);
            drawPolygon ( clippedWall );
         }
        for each portal of activeCell
          if ( portal intersects viewingFrustrum )
          {
             Polygon clippedPortal = clipPolygon ( portal,  clippingPoly);
             renderScene ( clippedPortal, adjacentCell ( activeCell, portal ) );
          }
    }
```

Так как в большинстве случаев все многоугольники являются выпуклыми, то в результате требуется процедура отсечения одного выпуклого многоугольника по другому выпуклому многоугольнику.

10.5.3. Метод иерархических подсцен

Традиционный метод порталов можно несколько видоизменить, отказавшись от условия выпуклости областей, на которые порталы разбивают сцену.

Разобьем всю сцену на ряд отдельных областей (подсцен) при помощи порталов. Каждую такую подсцену можно рассматривать как некоторый абстрактный объект со своей, присущей именно ему внутренней структурой, умеющей строить свое изображение в заданном многоугольнике картинной плоскости. И мы приходим к тому, что каждая подсцена является объектом, унаследованным от приведенного ниже абстрактного класса SubScene.

```
//File subscene.h
class SubScene : public Object
{
public:
    SubScene () {}
    virtual ~SubScene () {}
    virtual void render ( const Polygon& area )= 0;
};
```

Внутренняя организация данных и метод рендеринга для каждой конкретной подсцены могут быть уникальными и наиболее подходящими к ее структуре. Так, для подсцен, которые являются внутренними помещениями, может использоваться метод BSP-деревьев или метод *s*-буфера. Для подсцен, представляющих собой открытые участки пространства (к примеру, ландшафт), может быть удобна другая организация данных и рендеринга, например использование явной сортировки граней. При этом каждая подсцена, в свою очередь, может состоять из других подсцен.

В результате мы приходим к концепции иерархической сцены как объекта, состоящего из набора подсцен, соединенных порталами. Это является неким аналогом агрегатного объекта в методе трассировки лучей.

Упражнения

Модифицируйте алгоритмы построения графика функции двух переменных для работы при произвольных углах φ и ψ.

Покажите, почему в алгоритме Робертса нельзя отбрасывать все ребра, составляющие границу нелицевой грани.

Реализуйте один из алгоритмов удаления невидимых линий (Робертса или Аппеля). Посмотрите, к какому выигрышу во времени приводит использование равномерного разбиения картинной плоскости.

Реализуйте алгоритм z-буфера. Напишите процедуру вывода грани в z-буфер, максимально использующую связность пикселов грани.

Покажите, что при использовании перспективного проектирования значения глубины для пикселов одной грани уже нелинейно зависят от координаты пиксела, в то время как величина, обратная глубине, изменяется линейно.

Реализуйте алгоритм визуализации сцены с использованием BSP-деревьев (по заданному набору граней строится BSP-дерево, которое затем визуализируется для различных углов зрения и положений наблюдателя).

Модифицируйте алгоритм s-буфера для работы с полупрозрачными гранями.

Реализуйте в s-буфере функцию проверки того, изменяет ли данная грань содержимое s-буфера, и на основе этого реализуйте метод иерархического s-буфера.

Реализуйте метод порталов для визуализации внутренности лабиринта типа, используемого в игре Descent.

Приведите пример, когда метод упорядочения и расщепления граней для сцены, состоящей из n граней, приводит к $O(n^2)$ граням.

Глава 11
ПРОСТЕЙШИЕ МЕТОДЫ РЕНДЕРИНГА ПОЛИГОНАЛЬНЫХ МОДЕЛЕЙ

В большинстве случаев модели задаются набором плоских выпуклых граней. Поэтому при построении изображения естественно воспользоваться этой простотой модели. Существует три простейших метода рендеринга полигональных моделей, дающих достаточно приемлемые результаты, - метод плоского (постоянного) закрашивания, метод Гуро и метод Фонга.

11.1. Метод постоянного закрашивания

Это самый простой метод из всех трех. Он заключается в том, что на грани берется произвольная точка и определяется ее освещенность, которая и принимается за освещенность всей грани.

В качестве модели освещенности обычно используются простейшие модели вида

$$I = K_a I_a + K_d (n,l) \qquad (11.1)$$

или

$$I = K_a I_a + K_d (n,l) + K_s (n,h)^p \qquad (11.2)$$

Получающееся при этом изображение носит ярко выраженный полигональный характер - сразу видно, что модель состоит из отдельных плоских граней. Это связано с тем, что если рассматривать освещенность вдоль поверхности какого-либо объекта, то она претерпевает разрывы на границах граней (фактически освещенность является кусочно-постоянной функцией).

Более высокого уровня реалистичности можно добиться, если воспользоваться методом, обеспечивающим непрерывность освещенности вдоль границ объектов. Именно такой метод и рассматривается ниже.

11.2. Метод Гуро

Метод Гуро обеспечивает непрерывность освещенности за счет использования билинейной интерполяции.

Пусть задана плоская грань $v_1v_2v_3$ (рис. 11.1). Найдем значение освещенности в каждой ее вершине, используя формулу (11.1) или (11.2).

Обозначим получившиеся значения через $I_1I_2I_3$.

Рисуя грань $v_1v_2v_3$ построчно, будем находить значения освещенности в концах каждого горизонтального отрезка путем линейной интерполяции значений вдоль ребер. Так, освещенность в точке A (рис. 11.1) вычисляется по следующей формуле

$$I(A) = I_1(1-t) + I_2 t, t = \frac{|Av_1|}{|v_1v_2|}$$

При рисовании очередного отрезка *AB* будем считать, что интенсивность изменяется от *I(A)* до *I(B)* линейно.

Ниже приводится функция, осуществляющая построение горизонтального отрезка с изменением интенсивности от I_a до I_b.

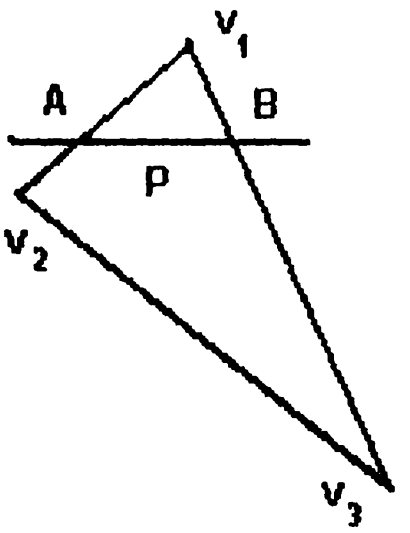


Рис. 11.1

```
// File drawLine.cpp
void  drawLine (int xa, int xb, int y, int ia, int ib)
{
        long        i  = ((long) ia) << 16;
        long        di = (((long)(ib - ia)) << 16 ) / (xb- xa + 1);

        for ( int x = xa; x <= xb; x++, i += di )
                    putpixel ( x, y, (int)(i >> 16) );
}
```

Эта функция рассчитана на то, что цветам от *ia* до *ib* соответствует набор промежуточных цветов с линейно возрастающей интенсивностью, например когда вся палитра состоит из оттенков одного или нескольких цветов.

Подобный метод билинейной интерполяции используется в ряде графических систем, например в OpenGL.

Метод Гуро гарантирует создание непрерывного поля освещенности вдоль объекта, но это поле не является гладким (дифференцируемым). Следующий метод строит поле освещенности, полностью имитируя гладкость объекта.

11.3. Метод Фонга

Гладкий объект отличается от негладкого тем, что на его поверхности задано непрерывное поле единичных векторов нормали. Попытаемся построить такое поле искусственно. Для этого применим описанную ранее процедуру билинейной интерполяции не к значениям освещенности, а к значениям вектора нормали. В результате мы получим непрерывное поле векторов нормали, но, так как эти векторы не всегда оказываются единичными, нужна нормировка.

При помощи полученного единичного вектора нормали в соответствующей точке по формулам (11.1) или (11.2) находится значение освещенности. Это позволяет создавать достаточно реалистически выглядящие изображения с корректными бликами.

Метод Фонга требует намного больше вычислений, чем метод Гуро, так как вычисление вектора нормали и освещенности производится отдельно в каждой точке.

Методы Гуро и Фонга являются инкрементальными методами, использующими значения параметров в предыдущей точке для вычисления значений параметров в следующей.

Тем не менее этим методам присущи и определенные недостатки: в некоторых случаях они способны давать некорректный результат, зависящий от положения наблюдателя. Рассмотрим грань, приведенную на рис. 11.2.

Грань является квадратом. Цифрами обозначены значения освещенности в вершинах. Найдем значение освещенности в середине квадрата.

При той ориентации, которая приведена на рис. 11.2, *а*), это значение равно единице, однако если повернуть грань вокруг вектора нормали на 90° (рис. 11.2, *б*), то оно станет равным нулю. Тем самым мы получаем, что освещенность квадрата зависит от его ориентации.

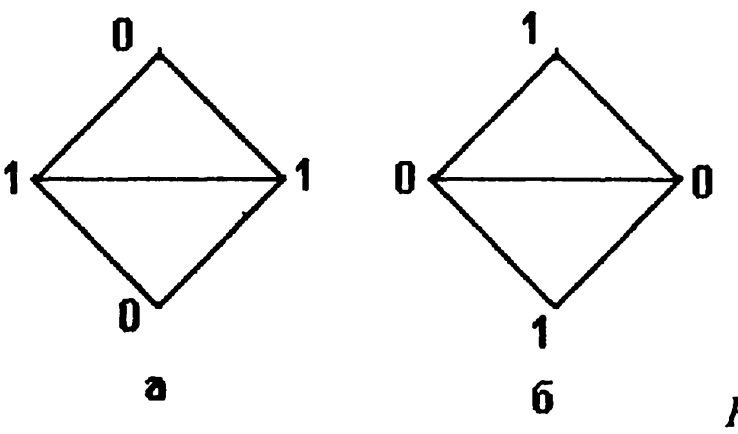


Рис. 11.2

Замечание. Методы Гуро и Фонга используют только векторы нормали, заданные в вершинах грани.

Обычно для нахождения вектора нормали в вершине используют нормированную взвешенную сумму векторов нормали граней, которым эта вершина принадлежит:

$$n = \frac{a_1 n_1 + \ldots + a_k n_k}{\left\| a_1 n_1 + \ldots a_k n_k \right\|}.$$

Упражнения

1. Напишите процедуру закраски треугольной грани методом Гуро. Переделайте программу построения изображения тора из гл. 9 для построения изображения методом Гуро. Сравните результаты.
2. Напишите процедуру закраски треугольной грани методом Фонга, считая, что источник света и наблюдатель находятся в бесконечно удаленных точках (векторы l и v не зависят от точки на грани).

Глава 12
РАБОТА С БИБЛИОТЕКОЙ OpenGL

На данный момент в Windows существует два стандарта для работы с трехмерной графикой: OpenGL, являющийся стандартом де-факто для всех графических рабочих станций, и Direct3D - стандарт, предложенный фирмой Microsoft. Здесь мы рассмотрим только стандарт OpenGL, так как, по мнению авторов (и не только авторов, аналогичного мнения придерживается один из основателей фирмы id Software Джон Кармак, считающий стандарт Direct 3D крайне неудобным для практического применения), он является намного более продуманным и удобным, нежели постоянно изменяющийся Direct 3D.

Существенным достоинством OpenGL является его широкая распространенность - он является стандартом в мире графических рабочих станций типа Sun, Silicon Graphics и др.

Стандарт OpenGL был разработан и утвержден в 1992 г. девятью ведущими фирмами, среди которых Digital Equipment Corparation, Evans & Sutherland, Hewlett Packard Co., IBM Corp., Intel Corp., Intergraph Corp., Silicon Graphics Inc., Sun Microsystems и Microsoft. В основу стандарта была положена библиотека IRIS GL, разработанная фирмой Silicon Graphics Inc.

OpenGL представляет собой программный интерфейс к графическому оборудованию (хотя существуют и чисто программные реализации OpenGL). Интерфейс насчитывает около 120 различных команд, которые программист использует для задания объектов и операций над ними (необходимых для написания интерактивных трехмерных приложений).

OpenGL был разработан как эффективный, аппаратно-независимый интерфейс, пригодный для реализации на различных аппаратных платформах. Поэтому OpenGL не включает в себя никаких специальных команд для работы с окнами или ввода информации от пользователя.

OpenGL позволяет:

1. Создавать объекты из геометрических примитивов (точки, линии, грани и битовые изображения).
2. Располагать объекты в трехмерном пространстве и выбирать способ и параметры проектирования.
3. Вычислять цвет всех объектов. Цвет может быть как явно задан, так и вычисляться с учетом источников света, параметров освещения, текстур.
4. Переводит математическое описание объектов и связанной с ними информации о цвете в изображение на экране.

При этом OpenGL может осуществлять дополнительные операции, такие, как удаление невидимых фрагментов изображения.

Команды OpenGL реализованы как модель 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 и клиент, так и на другом.

Хотя OpenGL и поддерживает палитровые режимы, мы их рассматривать не будем, считая, что работа ведется в режиме непосредственного задания цвета (HiColor или TrueColor).

Все команды (процедуры и функции) OpenGL начинаются с префикса gl, а все константы - с префикса GL_. Кроме того, в имена функций и процедур OpenGL входят суффиксы, несущие информацию о числе передаваемых параметров и их типе. В таблице приводятся вводимые OpenGL типы данных, стандартные типы языка C, которым они соответствуют, и суффиксы, которые им соответствуют.

Суффикс	*Описание*	*Тип в C*	*Тип в OpenGL*
b	8-битовое целое	signed char	GLbyte
s	16-битовое целое	short	GLshort
i	32-битовое целое	long	GLint, GLsizei
f	32-битовое число с плавающей точкой	float	GLfloat,GLclampf
d	64-битовое число с плавающей точкой	double	GLdouble, GLclampd
ub	8-битовое беззнаковое целое	unsigned char	GLubyte, GLboolean
us	16-битовое беззнаковое целое	unsigned short	GLushort
ui	32-битовое беззнаковое целое	unsigned long	GLUint, GLenum, GLbitfield
		void	GLvoid

Некоторые команды OpenGL оканчиваются на букву v. Это говорит о том, что команда получает указатель на массив значений, а не сами эти значения в виде отдельных параметров. Многие команды имеют как векторные, так и не векторные версии; например конструкции

```
glColor3f ( 1.0, 1.0, 1.0);
```

и

```
GLfloat color [] = {1.0, 1.0, 1.0};
glColor3fv (color);
```

эквивалентны.

OpenGL можно рассматривать как машину, находящуюся в одном из нескольких состояний. Внутри OpenGL содержится целый ряд переменных, например текущий цвет. Если установить текущий цвет, то все последующие объекты будут этого цвета до тех пор, пока текущий цвет не будет изменен. По умолчанию каждая системная переменная имеет свое значение и в любой момент значение каждой из этих переменных можно узнать. Обычно для этого используется одна из следующих функций: glGetBooleanv (), glGetDoublev (), glGetFloatv () и glGetIntergerv (). Для определения значения некоторых переменных служат специальные функции.

OpenGL предоставляет пользователю достаточно мощный, но низкоуровневый набор команд, и все операции высокого уровня должны выполняться в терминах этих команд. Обычно для облегчения работы вместе с OpenGL поставляется библиотека дополнительных команд, каждая из которых начинается с префикса glu. В данной главе будет рассмотрена часть этих команд.

12.1. Рисование геометрических объектов

OpenGL содержит внутри себя несколько различных буферов. Среди них фреймбуфер (куда строится изображение), z-буфер, служащий для удаления невидимых поверхностей, буфер трафарета и аккумулирующий буфер.

Для очистки окна (экрана, внутренних буферов) служит процедура

```
void glClear ( GLbitfield mask );
```

очищающая буферы, заданные переменной mask. Параметр mask является комбинацией следующих констант:

GL_COLOR_BUFFER_BIT - очистить буфер изображения (фреймбуфер),
GL_DEPTH_BUFFER_BIT - очистить z-буфер,
GL_ACCUM_BUFFER_BIT - очистить аккумулирующий буфер,
GL_STENCIL_BUFFER_BIT - очистить буфер трафарета.

Цвет, которым очищается буфер изображения, задается процедурой

```
void glClearColor ( GLclampf red, GLclampf green, GLclampf blue, GLclampf alpha );
```

Значение, записываемое в z-буфер при очистке, - процедурой

```
void glClearDepth ( GLclampd depth );
```

значение, записываемое в буфер трафарета, - процедурой

```
void glClearStencil ( GLint s );
```

цвет, записываемый в аккумулирующий буфер, - процедурой

```
void glClearAccum ( GLfloat red, GLfloat green, GLfloat blue, GLfloat alpha );
```

Сама команда glClear очищает одновременно все заданные буферы, заполняя их соответствующими значениями.

Для задания цвета объекта служит процедура

```
void glColor3{b s i f d ub us ui}(TYPE r,TYPE g,TYPE b);
void glColor4{b s i f d ub us ui}(TYPE r,TYPE g,TYPE b,TYPE a);
void glColor3{b s i f d ub us ui}v(const TYPE * v );
void glColor4{b s i f d ub us ui}v(const TYPE * v );
```

Если α-значение не задано, то оно автоматически полагается равным единице. Версии процедуры glColor*, где параметры являются переменными с плавающей точкой, автоматически обрезают переданные значения в отрезок [0,1]. Значения остальных типов приводятся (масштабируются) в этот отрезок для беззнаковых типов (при этом наибольшему возможному значению соответствует значение, равное единице) и в отрезок [-1,1] для типов со знаком.

Процедура

```
void glFlush ();
```

вызывает немедленное рисование ранее переданных команд. При этом ожидания завершения всех ранее переданных команд не происходит.

Команда
void glFinish ();
ожидает, пока не будут завершены все ранее переданные команды.

Если нужно включить удаление невидимых поверхностей методом *z*-буфера, то *z*-буфер необходимо очистить и подать команду
glEnable (GL_DEPTH_TEST);

Все геометрические примитивы задаются в терминах вершин. Каждая вершина задается набором чисел.

OpenGL работает с однородными координатами (*x*, *y*, *z*, *w*). Если координата *z* не задана, то она считается равной нулю. Если координата *w* не задана, то она считается равной единице.

Под линией в OpenGL подразумевается отрезок, заданный своими начальной и конечной вершинами.

Под гранью (многоугольником) в OpenGL подразумевается замкнутый выпуклый многоугольник с несамопересекающейся границей.

Все геометрические объекты в OpenGL задаются посредством вершин, а сами вершины - процедурой
void glVertex{2 3 4}{s i f d}[v](TYPE x,);
где реальное количество аргументов определяется первым суффиксом (2, 3 или 4), а суффикс *v* означает, что в качестве единственного аргумента выступает массив, содержащий необходимое количество координат. Например:

```
glVertex2s (1, 2);
glVertext3f ( 2.3, 1.5, 0.2 );
GLdouble vect [] ={ 1.0, 2.0, 3.0, 4.0};
glVertext4dv ( vect );
```

Для задания геометрических примитивов необходимо как-то выделить набор вершин, определяющих этот объект. Для этого служат процедуры glBegin() и glEnd().

Процедура
void glBegin (GLenum mode);
обозначает начало списка вершин, описывающих геометрический примитив. Тип примитива задается параметром *mode*, который принимает одно из следующих значений:

GL_POINTS - набор отдельных точек;

GL_LINES - пары вершин, задающих отдельные отрезки: $[v_0v_1v_2\ldots v_n]$ и т. д.;

GL_LINE_STRIP - незамкнутая ломаная $v_0v_1v_2\ldots v_n$;

GL_LINE_LOOP - замкнутая ломаная $v_0v_1v_2\ldots v_nv_0$;

GL_POLYGON - простой выпуклый многоугольник;

GL_TRIANGLES - тройки вершин, интерпретируемые как вершины отдельных треугольников: $v_0v_1v_2$, $v_3v_4v_5$,

GL_TRIANGLE_STRIP - связанная полоса треугольников: $v_0v_1v_2$, $v_2v_1v_3$, $v_2v_3v_4$

GL_TRIANGLE_FAN - веер треугольников: $v_0v_1v_2$, $v_0v_2v_3$, $v_0v_3v_4$, ...;

GL_QUADS - четверки вершин, задающие выпуклые четырехугольники: $v_0v_1v_2v_3$, $v_4v_5v_6v_7$,..

GL_QUAD_STRIP - полоса четырехугольников: $v_0v_1v_3v_2$, $v_2v_3v_5v_4$, $v_4v_5v_7v_6$,...

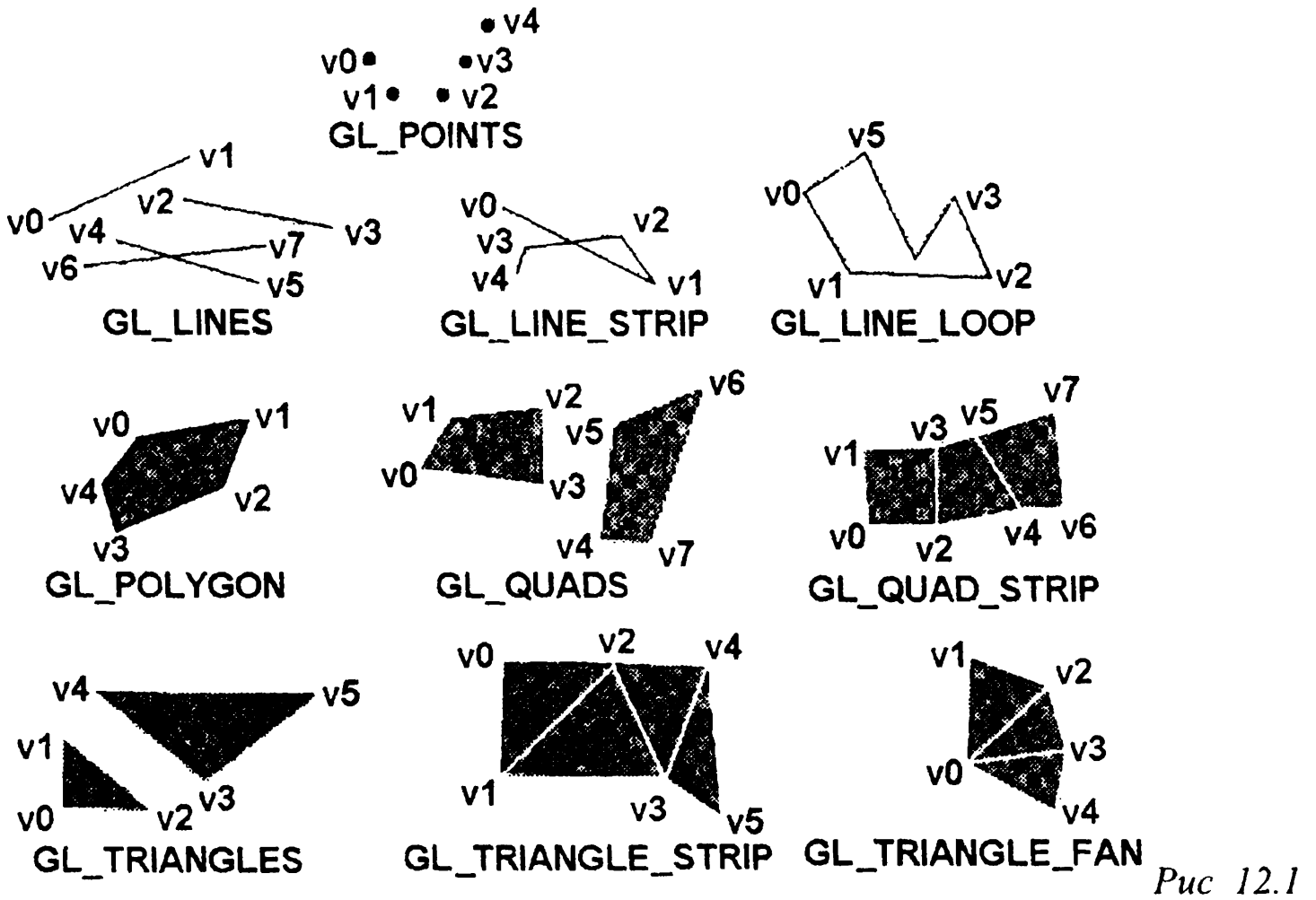


Рис 12.1

Процедура
void glEnd ();

отмечает конец списка вершин.

Между командами glBegin () и glEnd () могут находиться команды задания различных атрибутов вершин glVertex* (), glColor* (), glNormal* (), glCallList (), glCallLists (), glTexCoord* (), glEdgeFlag () и glMaterial* (). Между командами glBegin () и glEnd () все остальные команды OpenGL недопустимы и приводят к возникновению ошибок.

Рассмотрим в качестве примера задание окружности.

```
glBegin ( GL_LINE_LOOP );
for ( int i = 0; i < N; i++ )
{
    float angle =2*M_PI*i/N;
    glVertex2f ( cos (angle), sin (angle) );
}
glEnd ();
```

Хотя многие команды могут находиться между glBegin () и glEnd (), вершины генерируются при вызове glVertex* (). В момент вызова glVertex* () OpenGL присваивает создаваемой вершине текущий цвет, координаты текстуры, вектор нормали и т. д. Изначально вектор нормали полагается равным (0,0,1), цвет полагается равным (1, 1, 1, 1), координаты текстуры полагаются равными нулю.

12.2. Рисование точек, линий и многоугольников

Для задания размеров точки служит процедура
void glPointSize (GLfloat size);

которая устанавливает размер точки в пикселах, по умолчанию он равен единице.

Для задания ширины линии в пикселах служит процедура
void glLineWidth (GLfloat width);

Шаблон, которым будет рисоваться линия, можно задать при помощи процедуры
void glLineStipple (GLint factor, GLushort pattern);

Шаблон задается переменной pattern и растягивается в factor раз.
Использование шаблонов линий необходимо разрешить при помощи команды
glEnable (GL_LINE_STIPPLE);

Запретить использование шаблонов линий можно командой
glDisable (GL_LINE_STIPPLE);

Многоугольники рисуются как заполненные области пикселов внутри границы, хотя их можно рисовать либо только как граничную линию, либо просто как набор граничных вершин.

Многоугольник имеет две стороны, переднюю и заднюю, и может быть отрисован по-разному в зависимости от того, какая сторона обращена к наблюдателю. По умолчанию обе стороны рисуются одинаково. Для задания того, как именно следует рисовать переднюю и заднюю стороны многоугольника, служит процедура
void glPolygonMode (GLenum face, GLenum mode);

Параметр face может принимать значения GL_FRONT_AND_BACK (обе стороны), GL_FRONT (передняя сторона) или GL_BACK (задняя сторона); параметр mode может принимать значения GL_POINT, GL_LINE или GL_FILL, обозначая, что многоугольник должен рисоваться как набор граничных точек, граничная ломаная или заполненная область, например
glPolygonMode (GL_FRONT, GL_FILL);
glPolygonMode (GL_BACK, GL_LINE);

По умолчанию вершины многоугольника, которые появляются на экране в направлении против часовой стрелки, называются лицевыми (передними). Это можно изменить при помощи процедуры
void glFrontFace (GLenum mode);

По умолчанию параметр mode равняется GL_CCW, что соответствует направлению обхода против часовой стрелки. Если задать этот параметр равным GL_CW, то лицевыми будут считаться многоугольники с направлением обхода вершин по часовой стрелке.

При помощи процедуры
void glCullFace (GLenum mode);

вывод лицевых или нелицевых многоугольников можно запретить. Параметр mode принимает одно из значений GL_FRONT (оставить только лицевые грани). GL_BACK (оставить нелицевые) или GL_FRONT_AND_BACK (оставить все грани). Для отсечения граней необходимо разрешить отсечение при помощи команды
glEnable (GL_CULL_FACE);

Шаблон для заполнения грани можно задать при помощи процедуры
void glPolygonStipple (const GLubyte * mask);

где mask задает массив битов размером 32 на 32. Для разрешения использования шаблонов при выводе многоугольников служит команда
glEnable (GL_POLYGON_STIPPLE);

Свой вектор нормали для каждой вершины можно задать при помощи одной из следующих процедур:

```
void glNormal3{b s i d f}( TYPE nx, TYPE ny, TYPE nz );
void glNormal3{b s i d f}v ( const TYPE * v );
```

В версиях с суффиксами *b*, *s* или *i* значения аргументов масштабируются в отрезок [-1, 1].

В качестве примера приведем процедуру, строящую прямоугольный параллелепипед с ребрами, параллельными координатным осям, по диапазонам изменения x, y и z.

```
// File drawbox.cpp
void    drawBox ( GLfloat x1, GLfloat x2, GLfloat y1, GLfloat y2, GLfloat z1, GLfloat z2 )
    {
        glBegin ( GL_POLYGON );        // front face
          glNormal3f ( 0.0, 0.0, 1.0 );
          glVertex3f ( x1, y1, z2 );
          glVertex3f ( x2, y1, z2 );
          glVertex3f ( x2, y2, z2 );
          glVertex3f ( x1, y2, z2 );
        glEnd ();

        glBegin ( GL_POLYGON );        // back face
          glNormal3f ( 0.0, 0.0, -1.0 );
          glVertex3f ( x2, y1, z1 );
          glVertex3f ( x1, y1, z1 );
          glVertex3f ( x1, y2, z1 );
          glVertex3f ( x2, y2, z1 );
        glEnd ();

        glBegin ( GL_POLYGON );        // left face
          glNormal3f ( -1.0, 0.0, 0.0 );
          glVertex3f ( x1, y1, z1 );
          glVertex3f ( x1, y1, z2 );
          glVertex3f ( x1, y2, z2 );
          glVertex3f ( x1, y2, z1 );
        glEnd ();

        glBegin ( GL_POLYGON );        // right face
          glNormal3f ( 1.0, 0.0, 0.0 );
          glVertex3f ( x2, y1, z2 );
          glVertex3f ( x2, y1, z1 );
          glVertex3f ( x2, y2, z1 );
          glVertex3f ( x2, y2, z2 );
        glEnd ();

        glBegin ( GL_POLYGON );        // top face
          glNormal3f ( 0.0, 1.0, 0.0 );
          glVertex3f ( x1, y2, z2 );
          glVertex3f ( x2, y2, z2 );
          glVertex3f ( x2, y2, z1 );
          glVertex3f ( x1, y2, z1 );
        glEnd ();

        glBegin ( GL_POLYGON );        // bottom face
          glNormal3f ( 0.0, -1.0, 0.0 );
```

```
        glVertex3f ( x2, y1, z2 );
        glVertex3f ( x1, y1, z2 );
        glVertex3f ( x1, y1, z1 );
        glVertex3f ( x2, y1, z1 );
      glEnd ();
}
```

12.3. Преобразования объектов в пространстве. Камера

В процессе построения изображения координаты вершин подвергаются определенным преобразованиям.

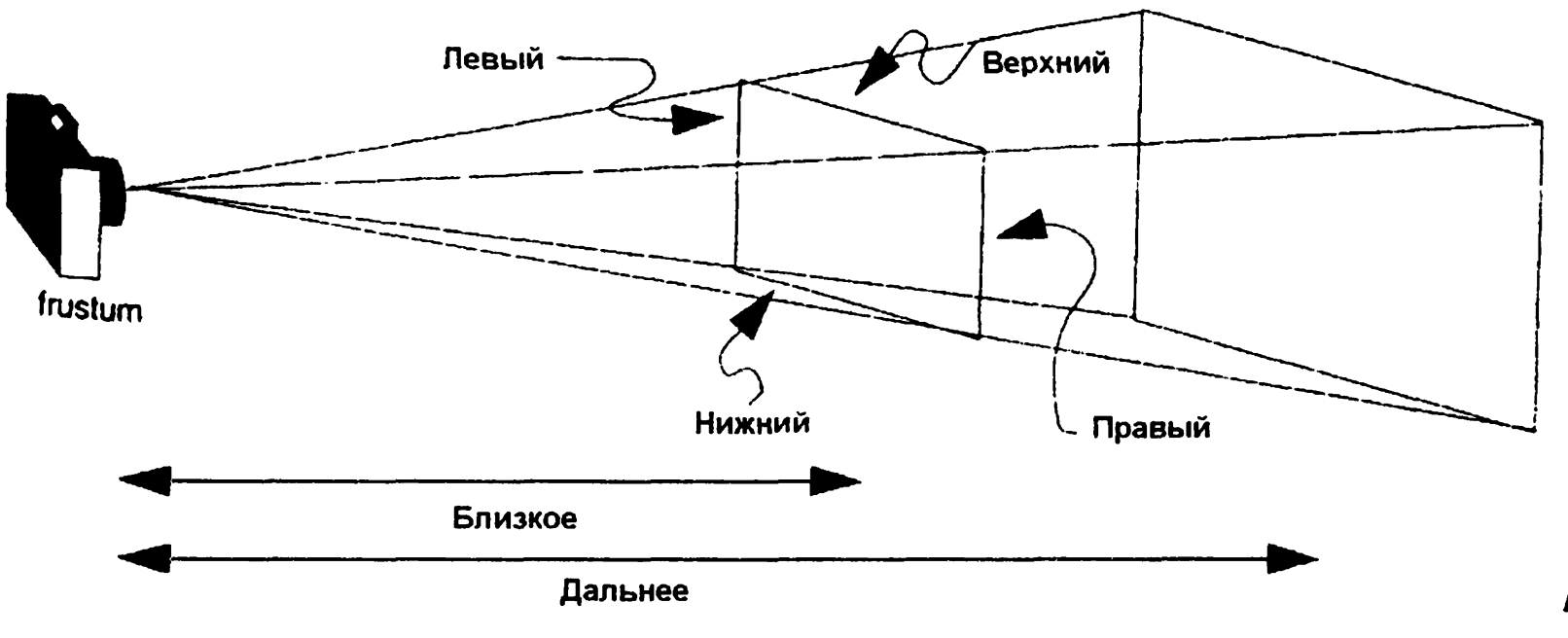


Рис. 12.2

Подобным преобразованиям подвергаются заданные векторы нормали.

Изначально камера находится в начале координат и направлена вдоль отрицательного направления оси *Oz*.

В OpenGL существуют две матрицы, последовательно применяющиеся в преобразовании координат. Одна из них - матрица моделирования (modelview matrix), а другая - матрица проектирования (projection matrix). Первая служит для задания положения объекта и его ориентации, вторая отвечает за выбранный способ проектирования. OpenGL поддерживает два типа проектирования - параллельное и перспективное.

Существует набор различных процедур, умножающих текущую матрицу (моделирования или проектирования) на матрицу выбранного геометрического преобразования. Текущая матрица задается при помощи процедуры

```
void glMatrixMode ( GLenum mode );
```

Параметр mode может принимать значения GL_MODELVIEW, GL_TEXTURE или GL_PROJECTION, позволяя выбирать в качестве текущей матрицы матрицу моделирования (видовую матрицу), матрицу проектирования или матрицу преобразования текстуры.

Процедура

```
void glLoadIdenity ();
```

устанавливает единичную текущую матрицу.

Обычно задание соответствующей матрицы начинается с установки единичной и последовательного применения матриц геометрических преобразований.

Преобразование сдвига задается процедурой

```
void glTranslate {fd}(TYPE x, TYPE y, TYPE z);
```

обеспечивающей сдвиг объекта на величину (x, y, z).

преобразование поворота - процедурой

```
void glRotate {fd}(TYPE angle, TYPE x, TYPE y, TYPE z);
```

обеспечивающей поворот на угол angle в направлении против часовой стрелки вокруг прямой с направляющим вектором (x, y, z).

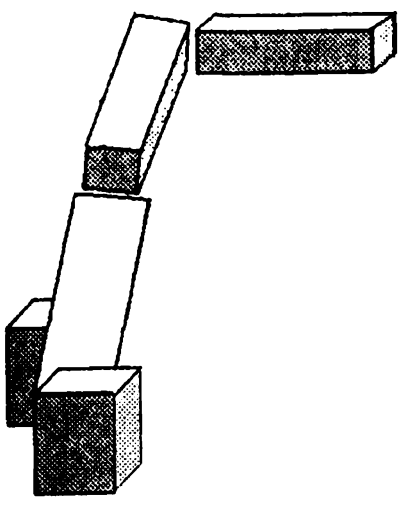


Рис. 12.3

преобразование масштабирования - процедурой

```
void glScale {fd}(TYPE x, TYPE y, TYPE z);
```

Если указано несколько преобразований, то текущая матрица в результате будет последовательно умножена на соответствующие матрицы.

Рассмотрим, каким образом можно построить изображение руки робота (рис. 12.3).

Рука состоит из двух боковых опор и трех последовательно сочлененных фрагментов.

Для построения этого объекта воспользуемся приведенной ранее процедурой drawBox.

```
// File drawarm.cpp
void drawArm ()
{
    glClear ( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );
    glMatrixMode ( GL_MODELVIEW );
    glLoadIdentity ();
    glTranslatef ( 0.0, 0.0, -64.0 );
    glRotatef ( 30.0, 1.0, 0.0, 0.0 );
    glRotatef ( 30.0, 0.0, 1.0, 0.0 );          // draw the block anchoring the arm
    drawBox ( 1.0, 3.0, -2.0, 2.0, -2.0, 2.0 );
    drawBox ( -3.0, -1.0, -2.0, 2.0, -2.0, 2.0 );

                // rotate the coordinate system and draw the arm's base member
    glRotatef ( (GLfloat) angle, 1.0, 0.0, 0.0 );
    drawBox ( -1.0, 1.0, -1.0, 1.0, -5.0, 5.0 );

                // translate the coordinate system to the end of base member, rotate it,
                // and draw the second member
    glTranslatef ( 0.0, 0.0, -5.0 );
    glRotatef ( -(GLfloat) angle / 2.0, 1.0, 0.0, 0.0 );
    drawBox ( -1.0, 1.0, -1.0, 1.0, -10.0, 0.0 );

                // translate and rotate coordinate system again and draw arm's
                // third member
    glTranslatef ( 0.0, 0.0, -5.0 );
    glRotatef ( -(GLfloat) angle / 2.0, 1.0, 0.0, 0.0 );
    drawBox ( -1.0, 1.0, -1.0, 1.0, -10.0, 0.0 );
}
```

Сначала инициализируется моделирующая матрица и локальная система координат переносится в точку, которая будет служит опорной точкой для руки:

```
glMatrixMode ( GL_MODELVIEW );
glLoadIdentity ();
glTranslatef ( 0.0, 0.0, -64.0 );
glRotatef ( 30.0, 1.0, 0.0, 0.0 );
glRotatef ( 30.0, 0.0, 1.0, 0.0 );
```

Далее рисуются два опорных блока, по обе стороны от опорной точки:

```
drawBox ( 1.0, 3.0, -2.0, 2.0, -2.0, 2.0 );
drawBox ( -3.0, -1.0, -2.0, 2.0, -2.0, 2.0 );
```

Осуществляется поворот локальной системы координат на угол *angle* вокруг оси *Ox* и рисуется первый блок руки:

```
glRotatef ( (GLfloat) angle, 1.0, 0.0, 0.0 );
drawBox ( -1.0, 1.0, -1.0, 1.0, -5.0, 5.0 );
```

Перед рисованием следующего блока начало локальной системы координат переносится из центра первого блока на 5 единиц в отрицательном направлении оси *Oz*. Это помещает центр локальной системы координат в конце первого члена руки - опорной точки для второго члена - точки, где первый и второй блоки сочленяются. Затем локальная система координат поворачивается вокруг оси *Ox* и рисуется следующий блок:

```
glTranslatef ( 0.0, 0.0, -5.0 );
glRotatef ( -(GLfloat) angle / 2.0, 1.0, 0.0, 0.0 );
drawBox ( -1.0, 1.0, -1.0, 1.0, -10.0, 0.0 );
```

Команды для рисования третьего члена аналогичны:

```
glTranslatef ( 0.0, 0.0, -5.0 );
glRotatef ( -(GLfloat) angle / 2.0, 1.0, 0.0, 0.0 );
drawBox ( -1.0, 1.0, -1.0, 1.0, -10.0, 0.0 );
```

Для задания матрицы проектирования сначала надо выполнить следующие команды:

```
glMatrixMode ( GL_PROJECTION );
glLoadIdentity ();
```

Преобразование проектирования определяет, как именно объекты будут проектироваться на экран и какие части объектов будут отсечены, как не попадающие в поле зрения.

Поле зрения при перспективном преобразовании является усеченной пирамидой (рис. 12.4).

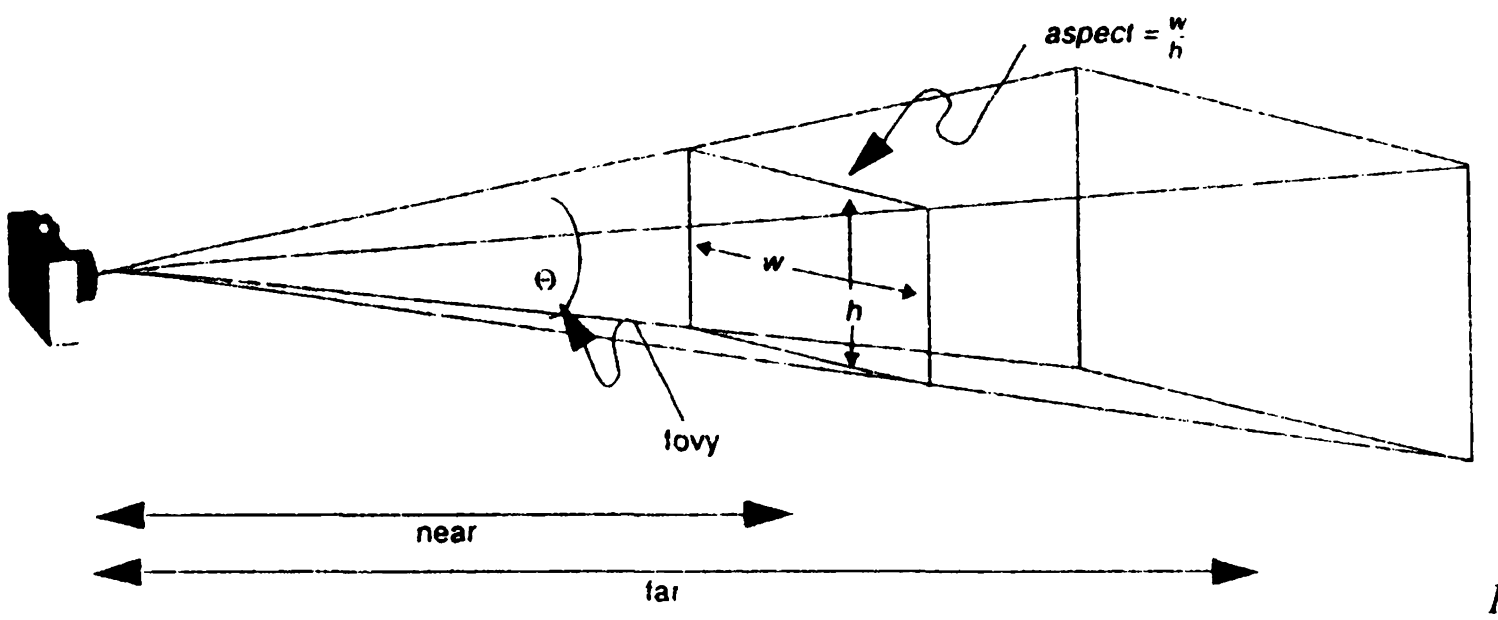


Рис. 12.4

Для задания перспективного преобразования в OpenGL служит процедура

```
void glFrustrum ( GLdouble left, GLdouble right, GLdouble bottom,
                  GLdouble top, GLdouble near, GLdouble far );
```

Смысл передаваемых параметров ясен из рисунка. Обратите внимание на то, что в момент применения матрицы проектирования координаты объектов уже переведены в систему координат камеры. Величины near и far должны быть неотрицательными.

Соответствующая матрица преобразования имеет вид:

$$\begin{pmatrix} \frac{2near}{right-left} & 0 & \frac{right+left}{right-left} & 0 \\ 0 & \frac{2near}{top-bottom} & \frac{top+bottom}{top-bottom} & 0 \\ 0 & 0 & -\frac{far+near}{far-near} & -\frac{2far*near}{far-near} \\ 0 & 0 & -1 & 0 \end{pmatrix}.$$

Иногда для задания перспективного преобразования удобнее воспользоваться следующей процедурой из библиотеки утилит OpenGL

```
void gluPerspective ( GLdouble fovy, GLdouble aspect,
                      GLdouble zNear, GLdouble zFar );
```

Эта процедура создает матрицу для задания симметричного поля зрения и умножает текущую матрицу на нее. Здесь *fovy* - угол зрения камеры в плоскости *Oxz*, лежащий в диапазоне [0,180]. Параметр aspect - отношение ширины области к ее высоте, zNear и zFar - расстояния вдоль отрицательного направления оси *Oz*, определяющие ближнюю и дальнюю плоскости отсечения.

Существует еще одна удобная функция для задания перспективного проектирования

```
void gluLookAt (GLdouble eyeX,GLdouble eyeY,GLdouble eyeZ,
                Gldouble centerX,GLdouble centerY,GLdouble centerZ,
                GLdouble upX, GLdouble upY, GLdouble upZ );
```

Вектор (*eyeX*, *eyeY*, *eyeZ*) задает положение наблюдателя, вектор (*centerX*, *centerY*, *centerZ*) -направление на центр сцены, а вектор (*upX*, *upY*, *upZ*) - направление вверх.

В случае параллельного проектирования полем зрения является прямоугольный параллелепипед. Для задания параллельного проектирования служит процедура

```
void glOrtho ( GLdouble left, GLdouble right, GLdouble bottom,
               GLdouble top, GLdouble near, GLdouble far );
```

Параметры left и right определяют координаты левой и правой вертикальных плоскостей отсечения, а bottom и top - нижней и верхней горизонтальных.

Соответствующая матрица преобразования имеет вид:

$$\begin{pmatrix} \frac{2}{right-left} & 0 & 0 & -\frac{right+left}{right-left} \\ 0 & \frac{2}{top-bottom} & 0 & -\frac{top+bottom}{top-bottom} \\ 0 & 0 & \frac{-2}{far-near} & -\frac{far+near}{far-near} \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Следующим шагом в задании проектирования (после выбора параллельного или перспективного преобразования) является задание области в окне, в которую будет помещено получаемое изображение. Для этого служит процедура

```
void glViewport ( GLint x, GLint y, GLsizei width,
                  GLsizei height );
```

Здесь (x, y) задает нижний левый угол прямоугольной области в окне, а width и height являются ее шириной и высотой.

OpenGL содержит стек матриц для каждого из трех типов преобразований. При этом текущую матрицу можно поместить в стек или снять матрицу с вершины стека и сделать ее текущей.

Для помещения текущей матрицы в стек служит процедура

```
void glPushMatrix ();
```

для снятия матрицы со стека - процедура

```
void glPopMatrix ();
```

Ниже приводится пример программы, использующей работу со стеком матриц, для построения изображения машины с колесами, причем каждое колесо крепится пятью болтами.

```
void drawWheelAndBolts ()
{
   drawWheel ();
     for ( int i = 0; i < 5; i++ )
     {
        glPushMatrix ();
        glRotatef ( 72.0*i, 0.0, 0.0, 1.0 );
        glTranslatef ( 3.0, 0.0, 0.0 );
        drawBolt ();
        glPopMatrix ();
   }
}
void drawBodyAndWheelAndBolts ()
{
   drawCarBody ();
   glPushMatrix ();
     glTranslatef ( 40.0, 0.0, 30.0 );
     drawWheelAndBolts ();
     glPopMatrix ();
     glPushMatrix ();
     glTranslatef ( 40.0, 0.0, -30.0 );
     drawWheelAndBolts ();
     glPopMatrix ();
                  // draw last 2 wheels similarly
}
```

12.4. Дисплейные списки

В традиционных языках программирования существуют функции и процедуры можно выделить конкретную последовательность команд, запомнить ее в определенном месте и вызывать всякий раз, когда в ней возникает потребность. Подобная

возможность существует и в OpenGL - набор команд OpenGL можно запомнить в так называемый дисплейный список (display list)(все команды и данные переводятся в некоторое внутреннее представление, наиболее удобное для данной реализации OpenGL) и затем вызывать при помощи всего лишь одной команды.

Каждому дисплейному списку соответствует некоторое целое число, идентифицирующее этот список.

Узнать, занят ли данный номер каким-либо дисплейным списком, можно при помощи функции

```
GLboolean glIsList ( GLuint list );
```

а зарезервировать range свободных идущих подряд номеров для идентификации дисплейных списков при помощи функции

```
GLuint glGenLists ( GLsizei range );
```

Эта функция возвращает первый свободный номер из блока зарезервированных данной командой.

Для того чтобы выделить последовательность команд в дисплейный список, служат процедуры glNewList () и glEndList (). Все команды, находящиеся между ними, помещаются в соответствующий дисплейный список автоматически.

```
void glNewList ( GLuint list, GLenum mode );
```

Здесь list - уникальное положительное число, служащее идентификатором списка, а параметр mode принимает одно из двух значений - GL_COMPILE или GL_COMPILE_AND_EXECUTE. Первое из этих значений обеспечивает только запоминание (компиляцию) дисплейного списка, второе - запоминание и выполнение этого списка.

Не все команды OpenGL могут быть запомнены в дисплейном списке. Ниже приводится список команд, которые не могут быть запомнены в нем.

glDeleteLists ()	glIsEnabled ()
glFeedbackBuffer ()	glIsList ()
glFinish ()	glPixelStore ()
glGenLists ()	glRenderMode ()
glGet* ()	glSelectBuffer ()

Для того чтобы вызвать (выполнить) дисплейный список, служит процедура

```
void glCallList ( GLuint list );
```

Возможно построение иерархических дисплейных списков, когда внутри определения одного списка вызываются и другие списки.

Для уничтожения дисплейных списков служит процедура

```
void glDeleteLists ( GLuint list, GLsizei range );
```

Замечание. *Дисплейные списки нельзя изменять. Список можно уничтожить или создать заново. Команды в дисплейном списке 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;
```

12.5. Задание моделей закрашивания

Линия или заполненная грань могут быть нарисованы одним цветом (плоское закрашивание, GL_FLAT) или путем интерполяции цветов в вершинах (закрашивание Гуро, GL_SMOOTH).

Для задания режима закрашивания служит процедура

```
void glShadeModel ( GLenum mode );
```

где параметр mode принимает значение GL_SMOOTH или GL_FLAT.

12.6. Освещение

OpenGL использует модель освещенности, в которой свет приходит из нескольких источников, каждый из которых может быть включен или выключен. Кроме того, существует еще общее фоновое (ambient) освещение.

Для правильного освещения объектов необходимо для каждой грани задать материал, обладающий определенными свойствами. Материал может испускать свой собственный свет, рассеивать падающий свет во всех направлениях (диффузное отражение) или подобно зеркалу отражать свет в определенных направлениях (см. гл. 2).

Пользователь может определить до восьми источников света и их свойства, такие, как цвет, положение и направление. Для задания этих свойств служит процедура

```
void glLight{if}[v](GLenum light, GLenum pname, TYPE param );
```

которая задает параметры для источника света light, принимающего значения GL_LIGHT0, GL_LIGHT1, ..., GL_LIGHT7. Параметр pname определяет характеристику источника света, которая задается последним параметром. Возможные значения для pname приведены в таблице.

Значение	*Знач. по умолчанию*	*Комментарий*
GL_AMBIENT	(0, 0, 0, 1)	Фоновая RGBA-освещенность
GL_DIFFUSE	(1, 1, 1, 1)	Диффузная RGBA-освещенность
GL_SPECULAR	(1, 1, 1, 1)	Бликовая (Фонга) RGBA-освещенность
GL_POSITION	(0, 0, 1, 0)	(x, y, z, w) - позиция источника света
GL_SPOT_DIRECTION	(0, 0, -1)	(x, y, z) - направление для конических источников света
GL_SPOT_EXPONENT	0	Показатель степени в формуле Фонга
GL_SPOT_CUTOFF	180	Половина угла для конических источников света
GL_CONSTANT_ATTENUATION		
GL_LINEAR_ATTENUATION		
GL_QUADRATIC_ATTENUATION		

Замечание. Значения GL_DIFFUSE и GL_SPECULAR в таблице относятся только к источнику света GL_LIGHT0, для остальных источников света обычно (0, 0, 0, 1).

Для употребления источников света использование расчета освещенности надо разрешить командой

```
glEnable ( GL_LIGHTING );
```

а применение соответствующего источника света разрешить (включить) командой glEnable, например:

```
glEnable ( GL_LIGHT0 );
```

Источник света можно рассматривать как имеющий вполне определенные координаты и светящий во всех направлениях или как направленный источник, находящийся в бесконечно удаленной точке и светящий в заданном направлении (x, y, z).

Если параметр w в команде GL_POSITION равен нулю, то соответствующий источник света - направленный и светит в направлении (x, y, z). Если же w отлично от нуля, то это позиционный источник света, находящийся в точке с координатами (x/w, y/w, z/w).

Заданием параметров GL_SPOT_CUTOFF и GL_SPOT_DIRECTION можно создавать источники света, которые будут иметь коническую направленность. По умолчанию значение параметра GL_SPOT_CUTOFF равно 180°, т. е. источник светит во всех направлениях с равной интенсивностью. Параметр GL_SPOT_CUTOFF определяет максимальный угол от направления источника, в котором распространяется свет от него; он может принимать значение 180° (не конический источник) или от 0 до 90°

Интенсивность источника с расстоянием, вообще говоря, убывает (параметры этого убывания задаются при помощи параметров GL_CONSTANT_ ATTENUATION, GL_LINEAR_ATTENUATION и GL_ QUADRATIC_ ATTENUATION). Только собственное свечение материала и глобальная фоновая освещенность с расстоянием не ослабевают.

Глобальное фоновое освещение можно задать при помощи команды

```
glLightModel{if}v ( GL_LIGHT_MODEL_AMBIENT, ambientColor);
```

Местонахождение наблюдателя оказывает влияние на блики на объектах. По умолчанию при расчетах освещенности считается, что наблюдатель находится в бесконечно удаленной точке, т. е. направление на наблюдателя постоянно для любой вершины. Можно включить более реалистическое освещение, когда направление на наблюдателя будет вычисляться для каждой вершины отдельно; для этого служит команда

```
glLightModeli (GL_LIGHT_MODEL_LOCAL_VIEWER, GL_TRUE );
```

Для задания освещения как лицевых, так и нелицевых граней (для нелицевых граней вектор нормали переворачивается) служит следующая команда:

```
glLightModeli (GL_LIGHT_MODEL_TWO_SIDE, GL_TRUE );
```

Причем существует возможность отдельного задания свойств материала для каждой из сторон.

Свойства материала, из которого сделан объект, задаются при помощи процедуры

```
void glMaterial{if}[v] (GLenum face, GLenum pname, TYPE param );
```

Параметр face указывает, для какой из сторон грани задается свойство, и принимает одно из следующих значений: GL_BACK, GL_FRONT_AND_BACK, GL_FRONT

Параметр pname указывает, какое именно свойство материала задается. Возможные значения представлены в таблице.

Значение	*Значение по умолчанию*	*Комментарий*
GL_AMBIENT	(0.2, 0.2, 0.2, 1.0)	Фоновый цвет материала
GL_DIFFUSE	(0.8, 0.8, 0.8, 1.0)	Диффузный цвет материала
GL_AMBIENT_AND_DIFFUSE		фоновый и диффузный цвета материала
GL_SPECULAR	(0, 0, 0, 1)	Цвет бликов
GL_SHININESS	0	Коэффициент Фонга для бликов
GL_EMISSION	(0, 0, 0, 1)	Цвет свечения материала

Итоговый цвет вычисляется по следующей формуле:

$$Color = E + I_a K_a + \sum S_i \frac{1}{k_c + k_l d + k_q d^2} \times$$

$$\left(I_{ai} K_a + \max\{(l,n),0\} I_{di} K_d + \left(\max\{(h,n),0\}\right)^p I_{si} K_s \right)$$

где

E - собственная светимость материала (GL_EMISSION);

I_a - глобальная фоновая освещенность;

K_a - фоновый цвет материала (GL_AMBIENT);

S_i - коэффициент, отвечающий за ослабление света в силу того, что источник имеет коническую направленность, и принимающий следующие значения:

1, если источник не конический,

0, если источник конический и вершина лежит вне конуса освещенности,

$(\max\{(v, l), 0\})^c$, где v - единичный вектор от источника света к вершине, l - единичный вектор направления для источника света (GL_SPOT_DIRECTION);

e - коэффициент GL_SPOT_EXPONENT;

d - расстояние до источника света;

k_c - коэффициент GL_CONSTANT_ATTENUATION;

k_l - коэффициент GL_LINEAR_ATTENUATION;

k_q- коэффициент GL_QUADRATIC_ATTENUATION;

I_{ai}- фоновая освещенность от i-го источника света;

l- единичный вектор направления на источник света;

n - единичный вектор нормали;

I_{di}- диффузная освещенность от i-го источника света;

K_d- диффузный цвет (GL_DIFFUSE);

p - коэффициент Фонга (GL_SHININESS);

I_{si} - бликовая освещенность от i-го источника света;

K_s- цвет бликов (GL_SPECULAR).

После проведения всех вычислений цветовые компоненты отсекаются по отрезку [0, 1].

Замечание. *Расчет освещенности в OpenGL не учитывает затенения одних объектов другими.*

12.7. Полупрозрачность. Использование α-канала

До сих пор мы не рассматривали α-канал (в RGBA-представлении цвета) и значение соответствующей компоненты во всех примерах всегда равнялось единице. Задавая значения, отличные от единицы, можно смешивать цвет выводимого пиксела с цветом пиксела, уже находящегося в соответствующем месте на экране, создавая тем самым эффект прозрачности.

При этом наиболее естественно думать об этом, считая что RGB-компоненты задают цвет фрагмента, а α-значение - его непрозрачность (степень поглощения фрагментом проходящего через него света). Так, если у стекла установить значение α, равное 0.2, то в результате вывода цвет получившегося фрагмента будет на 20 % состоять из собственного цвета стекла и на 80 % - из цвета фрагмента под ним.

Для использования α-канала необходимо сначала разрешить режим прозрачности и смешения цветов командой

```
glEnable ( GL_BLEND );
```

В процессе смешения цветов цветовые компоненты выводимого фрагмента $R_sG_sB_sA_s$ смешиваются с цветовыми компонентами уже выведенного фрагмента $R_dG_dB_dA_d$ по формуле

$$(R_sS_r + R_dD_r, G_sS_g + G_dD_g, B_sS_b + B_dD_b, A_sS_a + A_dD_a),$$

где (S_r, S_g, S_b, S_a) и (D_r, D_g, D_b, D_a) - коэффициенты смешения.

Для задания связи этих коэффициентов с α-значениями используется следующая функция:

```
void glBlendFunc (GLenum sfactor, GLenum dfactor );
```

Здесь параметр sfactor задает то, как нужно вычислять коэффициенты(S_r, S_g, S_b, S_a), а параметр dfactor - коэффициенты(D_r, D_g, D_b, D_a). Возможные значения для этих параметров приведены в таблице.

Значение	*Задействованные коэффициенты*	*Значение коэффициентов*
GL_ZERO	S, D	(0, 0, 0, 0)
GL_ONE	S, D	(1, 1, 1, 1)
GL_DST_COLOR	S	(R_d, G_d, B_d, A_d)
GL_SRC_COLOR	D	(R_s, G_s, B_s, A_s)
GL_ONE_MINUS_DST_COLOR	S	$(1, 1, 1, 1)-(R_d, G_d, B_d, A_d)$
GL_ONE_MINUS_SRC_COLOR	D	$(1, 1, 1, 1) - (R_s, G_s, B_s, A_s)$
GL_SRC_ALPHA	S, D	(A_s, A_s, A_s, A_s)
GL_ONE_MINUS_SRC_ALPHA	S, D	$(1, 1, 1, 1) - (A_s, A_s, A_s, A_s)$

GL_DST_ALPHA	S, D	(A_d, A_d, A_d, A_d)
GL_ONE_MINUS_DST_ALPHA	S, D	$(1, 1, 1, 1) - (A_d, A_d, A_d, A_d)$
GL_SRC_ALPHA_SATURATE	S	$(f, f, f, 1)$ $f = \min(A_s, 1 - A_d)$

12.8. Вывод битовых изображений

OpenGL поддерживает вывод битовых масок (изображений), когда на 1 пиксел приходится 1 бит.

Для вывода битовых масок служит процедура

```
void glBitmap ( GLsizei width, GLsizei height, GLfloat xo,
                GLfloat yo, GLfloat xi, GLfloat yi,
                const GLubyte * bitmap );
```

Эта процедура выводит изображение, задаваемое параметром bitmap. Битовое изображение выводится начиная с текущей растровой позиции. Параметры width и height задают размер битового изображения в пикселах. Параметры xo и yo используются для задания положения нижнего левого угла выводимого изображения относительно текущей растровой позиции, параметры xi и yi представляют собой величины, прибавляемые к текущей растровой позиции после вывода изображения.

Для задания текущей растровой позиции служит процедура

```
void glRasterPos {234}{sifd}[v]( TYPE x, TYPE y, TYPE z );
```

Для задания того, в каком формате хранятся пикселы в передаваемом изображении, служит команда

```
void glPixelStore{if}(Glenum pname, Glint param );
```

Аргумент pname определяет устанавливаемый параметр и может принимать одно из 12 значений (по 6 на чтение и запись пикселов).

pname	
GL_PACK_SWAP_BYTES GL_UNPACK_SWAP_BYTES	Если param равен GL_TRUE, то байты в многобайтовых компонентах цвета, глубины, индекса трафарета упорядочены в обратном порядке
GL_PACK_LSB_FIRST GL_UNPACK_LSB_FIRST	Если param равен GL_TRUE, то биты внутри байта упорядочены от младшего разряда к старшему. Этот параметр применим только к битовым массивам
GL_PACK_ROW_LENGTH GL_UNPACK_ROW_LENGTH	Если значение param больше нуля, то оно определяет число пикселов в строке
GL_PACK_ALIGNMENT GL_UNPACK_ALIGNMENT	Значение параметра определяет кратность выравнивания значений пикселов (1, 2, 4, 8)
GL_PACK_SKIP_PIXELS GL_PACK_SKIP_ROWS GL_UNPACK_SKIP_PIXELS GL_UNPACK_SKIP_ROWS	Значение параметра позволяет пропускать заданное количество пикселов или строк

Параметры GL_PACK_* используются при работе с командой glReadPixel, а параметр GL_UNPACK_* действует только для команд glDrawPixel, glTexImage1D, glTexImage2D, glBitmap и glPolygonStipple.

12.9. Ввод/вывод цветных изображений

OpenGL поддерживает вывод и полноцветных изображений, когда для каждого пиксела задаются все величины RGBA или только некоторые из них.

Для копирования изображения из фреймбуфера в обычную память служит процедура

```
void glReadPixels ( GLint x, GLint y, GLsizei width,
                    GLsizei height, GLenum format, GLenum type,
                    GLvoid * pixels );
```

Здесь параметры (x, y) задают координаты левого нижнего угла, а параметры width и height - размеры копируемого изображения.

Параметр format отражает то, какие данные о пикселе заносятся в буфер; возможными значениями являются GL_RGB, GL_RGBA, GL_RED, GL_GREEN, GL_BLUE, GL_ALPHA, GL_LUMINANCE_ALPHA, GL_LUMINANCE, GL_STENCIL_INDEX и GL_DEPTH_COMPONENT.

Параметр type задает тип каждого из записываемых значений. Возможными значениями являются GL_UNSIGNED_BYTE, GL_BYTE, GL_BITMAP, GL_UNSIGNED_SHORT, GL_SHORT, GL_UNSIGNED_INT, GL_INT и GL_FLOAT.

Для вывода изображения в фреймбуфер из оперативной памяти служит следующая процедура:

```
void glDrawPixels ( GLsizei width, GLsizei height, Glenum format,
                    GLenum type, const GLvoid * pixels );
```

Изображение выводится начиная с текущей растровой позиции.

12.10. Наложение текстуры

Текстурирование позволяет наложить изображение на многоугольник и вывести этот многоугольник с наложенной на него текстурой, соответствующим образом преобразованной. OpenGL поддерживает одно- и двумерные текстуры и различные способы наложения (применения) текстуры.

Для использования текстуры надо сначала разрешить одно- или двумерное текстурирование при помощи команд

```
glEnable ( GL_TEXTURE_1D );
```

или

```
glEnable ( GL_TEXTURE_2D );
```

Для задания двумерной текстуры служит процедура

```
void glTexImage2D ( GLenum target, GLint level,
    GLint component, GLsizei width, GLsizei height,
    GLint border, GLenum format, GLenum type,
    const GLvoid * pixels );
```

Параметр target зарезервирован для будущего использования и в нынешней версии должен быть равен GL_TEXTURE_2D.

Параметр level используется в том случае, если задается несколько разрешений данной текстуры; при ровно одном разрешении он должен быть равным нулю.

Следующий параметр - component - целое число от 1 до 4, показывающее, какие из RGBA-компонент выбраны для использования. Значение 1 выбирает компоненту R, значение 2 выбирает R и A компоненты, 3 соответствует R, G и B, а 4 соответствует компонентам RGBA.

Параметры width и height задают размеры текстуры, border задает размер границы (бортика), обычно равный нулю. Как параметр width, так и параметр height, должны иметь вид $2^n + 2b$, где n - целое число, а b - значение параметра border. Максимальный размер текстуры зависит от реализации OpenGL, но он не менее 64*64.

Смысл параметров format и type аналогичен их смыслу в процедурах glReadPixels и glDrawPixels.

При текстурировании OpenGL поддерживает использование пирамидального фильтрования (mip-mappping). Для этого необходимо иметь текстуры всех промежуточных размеров, являющихся степенями двух, вплоть до 1x1, и для каждого такого разрешения вызвать glTexImage2D с соответствующими параметрами level, width, height и image. Кроме того, необходимо задать способ фильтрования, который будет применяться при выводе текстуры.

Под фильтрованием здесь подразумевается способ, которым для каждого пиксела будет выбираться подходящий элемент текстуры (тексел). При текстурировании возможна ситуация, когда 1 пикселу соответствует небольшой фрагмент тексела (увеличение) или же, наоборот, когда 1 пикселу соответствует целая группа текселов (уменьшение). Способ выбора соответствующего тексела как для увеличения, так и для уменьшения (сжатия) текстуры необходимо задать отдельно. Для этого используется процедура

```
void glTexParameteri ( GL_TEXTURE_2D, GLenum p1, GLenum p2 );
```

где параметр p1 показывает, задается ли фильтр для сжатия или для растяжения текстуры, принимая значение GL_TEXTURE_MIN_FLITER или GL_TEXTURE_MAG_FILTER.; параметр p2 задает способ фильтрования, возможные значения приведены в таблице.

Параметр p1	*Параметр p2*
GL_TEXTURE_MAG_FILTER	GL_NEAREST, GL_LINEAR
GL_TEXTURE_MIN_FILTER	GL_NEAREST, GL_LINEAR, GL_NEAREST_MIPMAP_NEAREST, GL_NEAREST_MIPMAP_LINEAR, GL_LINEAR_MIPMAP_NEAREST, GL_LINEAR_MIPMAP_LINEAR

Значение GL_NEAREST соответствует выбору тексела с координатами, ближайшими к центру пиксела. Значению GL_LINEAR соответствует выбор взвешенной линейной комбинации из массива 2x2 текселов, лежащих ближе всего к рассматриваемому пикселу. При использовании пирамидального фильтрования помимо

выбора текселя на одном слое текстуры появляется возможность либо выбрать один соответствующий слой, либо проинтерполировать результаты выбора между двумя соседними слоями.

Для правильного применения текстуры каждой вершине следует задать соответствующие ей координаты текстуры при помощи процедуры

```
void glTexCoord{1234}{sifd}[v] (TYPE coord,    );
```

Этот вызов задает значения индексов текстуры для последующей команды glVertex * ().

Если размер грани больше, чем размер текстуры, то для циклического повторения текстуры служат команды

```
glTexParameteri ( GL_TEXTURE_2D, GL_TEXTURE_S_WRAP, GL_REPEAT );
glTexParameteri ( GL_TEXTURE_2D, GL_TEXTURE_T_WRAP, GL_REPEAT );
```

Координаты текстуры обычно подвергаются преобразованию при помощи матрицы текстурирования. По умолчанию она совпадает с единичной матрицей, но пользователь сам имеет возможность задать преобразования текстуры, например следующим образом:

```
glMatrixMode ( GL_TEXTURE );
glRotatef (    );
. . .  . . . .
glMatrixMode ( GL_MODELVIEW );
```

Замечание. *При выводе текстуры OpenGL может использовать линейную интерполяцию (аффинное текстурирование) (см. гл. 13) или же точно учитывать перспективное искажение. Для задания точного текстурирования служит команда*

```
glHint (GL_PERSPECTIVE_CORRECTION_HINT, GL_NICEST );
```

Если качество не играет большой роли, а нужна высокая скорость рендеринга, то в качестве последнего аргумента следует использовать константу GL_FASTEST.

12.11. Работа с OpenGL в Windows

OpenGL представляет собой универсальную графическую библиотеку, которая может быть реализована в любой оконной среде. Существует ее реализация и для Windows 95 и для Windows NT.

Для работы OpenGL в Windows используется понятие контекста воспроизведения (rendering context), который связывает OpenGL с оконной системой Windows. Если обычный контекст устройства (device context) содержит информацию, относящуюся к графическим компонентам GDI, то контекст воспроизведения содержит информацию, относящуюся к OpenGL.

Таким образом, чтобы начать работать с командами OpenGL, приложение должно создать как минимум один контекст воспроизведения и сделать его текущим.

Перед созданием контекста воспроизведения необходимо установить формат пикселов.

Для установки формата пикселов используется функция

```
int ChoosePixelFormat ( HDC, const PIXELFORMATDESCRIPTOR * );
```

выбирающая наиболее подходящий формат исходя из информации, переданной в полях структуры PIXELFORMATDESCRIPTOR.

После того как найден подходящий формат пикселов, нужно установить его в контексте устройства при помощи функции

```
BOOL SetPixelFormat ( HDC hDC, int pixelFormat,
    const PIXELFORMATDESCRIPTOR * );
```

Для работы с контекстом воспроизведения в Windows существуют функции

```
HGLRC wglCreateContext ( HDC hDC );
```

и BOOL wglMakeCurrent (HDC hDC, HGLRC hGLRC);

Первая из них создает новый контекст воспроизведения OpenGL, который подходит для рисования на устройстве, задаваемом контекстом hDC. Вторая функция устанавливает текущий контекст воспроизведения.

По окончании работы с OpenGL созданный контекст воспроизведения необходимо уничтожить. Для этого существует функция

```
BOOL wglDeleteContext ( HGLRC hGLRC );
```

Текущий контекст воспроизведения можно узнать при помощи функции

```
HGLRC wglGetCurrentContext ();
```

Ниже приводится пример программы, рисующей в окне ранее рассмотренную руку робота.

```
// File arm1.cpp
#include      <windows.h>
#include      <gl\gl.h>
#include      <gl\glu.h>

LONG WINAPI wndProc ( HWND, UINT, WPARAM, LPARAM );
void      setDCPixelFormat ( HDC );
void      initializeRC ();
void      drawBox ( GLfloat, GLfloat, GLfloat, GLfloat, GLfloat, GLfloat );
void      drawScene ( HDC, int );

int   WINAPI WinMain ( HINSTANCE hInstance, HINSTANCE hPrevInstance,
                       LPSTR cmdLine, int cmdShow )
{
      static char     appName [] = "Robot Arm";
      WNDCLASS  wc;
      HWND      hWnd;
      MSG       msg;

      wc.style          = CS_HREDRAW | CS_VREDRAW;
      wc.lpfnWndProc    = (WNDPROC) wndProc;
      wc.cbClsExtra     = 0;
      wc.cbWndExtra     = 0;
      wc.hInstance      = hInstance;
      wc.hIcon          = LoadIcon ( NULL, IDI_APPLICATION );
      wc.hCursor        = LoadCursor ( NULL, IDC_ARROW );
      wc.hbrBackground  = (HBRUSH) (COLOR_WINDOW + 1);
      wc.lpszMenuName   = NULL;
      wc.lpszClassName  = appName;

      RegisterClass ( &wc );

      hWnd = CreateWindow ( appName, appName,
                            WS_OVERLAPPEDWINDOW |
                            WS_CLIPCHILDREN | WS_CLIPSIBLINGS,
```

```
                CW_USEDEFAULT, CW_USEDEFAULT,
                                CW_USEDEFAULT,
                CW_USEDEFAULT, HWND_DESKTOP, NULL, hInstance, NULL );

        ShowWindow ( hWnd, cmdShow );
        UpdateWindow ( hWnd );

        while ( GetMessage ( &msg, NULL, 0, 0 ) )
        {
                TranslateMessage ( &msg );
                DispatchMessage  ( &msg );
        }

        return msg.wParam;
}
LONG WINAPI wndProc ( HWND hWnd, UINT msg, WPARAM wParam,
                                LPARAM lParam )
{
        static HDC          hdc;
        static HGLRC        hrc;
        PAINTSTRUCT         ps;
        GLdouble            aspect;
        GLsizei             width, height;
        static int          angle = 0;

        switch ( msg )
        {
                case WM_CREATE:
                hdc = GetDC ( hWnd );
                setDCPixelFormat ( hdc );
                hrc = wglCreateContext ( hdc );
                ReleaseDC ( hWnd, hdc );
                return 0;

        case WM_SIZE:
                hdc = GetDC ( hWnd );
                wglMakeCurrent ( hdc, hrc );

                width  = (GLsizei) LOWORD ( lParam );
                height = (GLsizei) HIWORD ( lParam );
                aspect = (GLdouble) width /
                                        (GLdouble) height;

                glMatrixMode ( GL_PROJECTION );
                glLoadIdentity ();
                gluPerspective ( 30.0, aspect, 1.0, 100.0 );

                glViewport ( 0, 0, width, height );

                wglMakeCurrent ( NULL, NULL );
                ReleaseDC ( hWnd, hdc );

                return 0;

        case WM_PAINT:
                hdc = BeginPaint ( hWnd, &ps );
                wglMakeCurrent ( hdc, hrc );
                drawScene  ( hdc, angle );
```

```
                wglMakeCurrent ( NULL, NULL );
                EndPaint   ( hWnd, &ps );

                return 0;

            case WM_DESTROY:
                wglMakeCurrent ( NULL, NULL );
                wglDeleteContext ( hrc );

                PostQuitMessage ( 0 );

                return 0;
        }

        return DefWindowProc ( hWnd, msg, wParam, lParam );
}

void    setDCPixelFormat ( HDC hdc )
{
        static PIXELFORMATDESCRIPTOR pfd =
        {
            sizeof ( PIXELFORMATDESCRIPTOR ),
            1,
            PFD_DRAW_TO_WINDOW | PFD_SUPPORT_OPENGL,// flags
            PFD_TYPE_RGBA,        // RGBA pixel values
            24,                   // 24-bit color
            0, 0, 0, 0, 0, 0,     // don't care about these
            0, 0,                 // no alpha buffer
            0, 0, 0, 0, 0,        // no accumulation buffer
            32,                   // 32-bit depth buffer
            0,                    // no stencil buffer
            0,                    // no auxiliary buffers
            PFD_MAIN_PLANE,       // layer type
            0,                    // reserved (must be 0)
            0, 0, 0               // no layer masks
        };
        int     pixelFormat;

        pixelFormat = ChoosePixelFormat ( hdc, &pfd );
        SetPixelFormat ( hdc, pixelFormat, &pfd );
        DescribePixelFormat ( hdc, pixelFormat,
                              sizeof (PIXELFORMATDESCRIPTOR), &pfd );

        if ( pfd.dwFlags & PFD_NEED_PALETTE )
            MessageBox ( NULL, "Wrong video mode", "Attention", MB_OK );
}

void    initializeRC ()
{
        GLfloat lightAmbient  [] = { 0.1, 0.1, 0.1, 1.0 };
        GLfloat lightDiffuse  [] = { 0.7, 0.7, 0.7, 1.0 };
        GLfloat lightSpecular [] = { 0.0, 0.0, 0.0, 1.0 };

        glFrontFace  ( GL_CCW );
        glCullFace   ( GL_BACK );
        glEnable     ( GL_CULL_FACE );

        glDepthFunc  ( GL_LEQUAL );
```

```
        glEnable    ( GL_DEPTH_TEST );
        glClearColor ( 0.0, 0.0, 0.0, 0.0 );

        glLightfv ( GL_LIGHT0, GL_AMBIENT, lightAmbient );
        glLightfv ( GL_LIGHT0, GL_DIFFUSE, lightDiffuse );
        glLightfv ( GL_LIGHT0, GL_SPECULAR, lightSpecular );
        glEnable  ( GL_LIGHTING );
        glEnable  ( GL_LIGHT0 );
}

void    drawScene ( HDC hDC, int angle )
{
        GLfloat blue [] = { 0.0, 0.0, 1.0, 1.0 };
        GLfloat yellow [] = { 1.0, 1.0, 0.0, 1.0 };

        glClear ( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );

        glMatrixMode ( GL_MODELVIEW );
        glLoadIdentity ();
        glTranslatef ( 0.0, 0.0, -64.0 );
        glRotatef ( 30.0, 1.0, 0.0, 0.0 );
        glRotatef ( 30.0, 0.0, 1.0, 0.0 );
                        // draw the block anchoring the arm
        glMaterialfv ( GL_FRONT, GL_AMBIENT_AND_DIFFUSE, yellow );
        drawBox ( 1.0, 3.0, -2.0, 2.0, -2.0, 2.0 );
        drawBox ( -3.0, -1.0, -2.0, 2.0, -2.0, 2.0 );
                        // rotate the coordinate system and draw the arm's base member
        glMaterialfv ( GL_FRONT, GL_AMBIENT_AND_DIFFUSE, blue );
        glRotatef ( (GLfloat) angle, 1.0, 0.0, 0.0 );
        drawBox ( -1.0, 1.0, -1.0, 1.0, -5.0, 5.0 );
                        // translate the coordinate system to the end of base member, rotate it,
                        // and draw the second member
        glTranslatef ( 0.0, 0.0, -5.0 );
        glRotatef ( -(GLfloat) angle / 2.0, 1.0, 0.0, 0.0 );
        drawBox ( -1.0, 1.0, -1.0, 1.0, -10.0, 0.0 );

                        // translate and rotate coordinate
                        // system again and draw arm's third member
        glTranslatef ( 0.0, 0.0, -5.0 );
        glRotatef ( -(GLfloat) angle / 2.0, 1.0, 0.0, 0.0 );
        drawBox ( -1.0, 1.0, -1.0, 1.0, -10.0, 0.0 );

        glFlush ();    // render the scene to pixel buffer
}
```

При помощи OpenGL можно создавать анимации. При этом для изображения используется режим работы с двумя буферами, когда содержимое одного из них показывается, а в другом осуществляется построение. После окончания построения специальная команда меняет буферы местами (по аналогии с двухстраничным режимом работы). Приводимая программа для Windows строит анимацию движения руки робота. Обратите внимание на использование флага PFD_DOUBLE_BUFFER при задании формата пикселов и команду SwapBuffers, меняющую буферы местами (по умолчанию вывод происходит в невидимый буфер).

```
// File arm2.cpp
#include    <windows.h>
#include    <gl\gl.h>
#include    <gl\glu.h>

LONG WINAPI wndProc ( HWND, UINT, WPARAM, LPARAM );
void        setDCPixelFormat ( HDC );
void        initializeRC ();
void        drawBox ( GLfloat, GLfloat, GLfloat, GLfloat, GLfloat, GLfloat );
void        drawScene ( HDC, int );

int  WINAPI WinMain ( HINSTANCE hInstance, HINSTANCE hPrevInstance,
                      LPSTR cmdLine, int cmdShow )
{
    static char appName [] = "Robot Arm";
    WNDCLASS    wc;
    HWND        hWnd;
    MSG         msg;

    wc.style            = CS_HREDRAW | CS_VREDRAW;
    wc.lpfnWndProc      = (WNDPROC) wndProc;
    wc.cbClsExtra       = 0;
    wc.cbWndExtra       = 0;
    wc.hInstance        = hInstance;
    wc.hIcon            = LoadIcon ( NULL, IDI_APPLICATION );
    wc.hCursor          = LoadCursor ( NULL, IDC_ARROW );
    wc.hbrBackground    = (HBRUSH) (COLOR_WINDOW + 1);
    wc.lpszMenuName     = NULL;
    wc.lpszClassName    = appName;

    RegisterClass ( &wc );

    hWnd = CreateWindow ( appName, appName,
                WS_OVERLAPPEDWINDOW | WS_CLIPCHILDREN |
                            WS_CLIPSIBLINGS,
                CW_USEDEFAULT, CW_USEDEFAULT, CW_USEDEFAULT,
                CW_USEDEFAULT, HWND_DESKTOP, NULL,
                hInstance, NULL );

    ShowWindow ( hWnd, cmdShow );
    UpdateWindow ( hWnd );

    while ( GetMessage ( &msg, NULL, 0, 0 ) )
    {
        TranslateMessage ( &msg );
        DispatchMessage  ( &msg );
    }

    return msg.wParam;
}

LONG WINAPI wndProc ( HWND hWnd, UINT msg,
                      WPARAM wParam, LPARAM lParam )
{
    static HDC   hdc;
    static HGLRC hrc;
    PAINTSTRUCT  ps;
```

```
GLdouble    aspect;
GLsizei     width, height;
static int  angle = 0;
static BOOL up = TRUE;
static UINT timer;

switch ( msg )
{
    case WM_CREATE:
        hdc = GetDC ( hWnd );
        setDCPixelFormat ( hdc );
        hrc = wglCreateContext ( hdc );
        wglMakeCurrent ( hdc, hrc );
        initializeRC ();
        timer = SetTimer ( hWnd, 1, 50, NULL );

        return 0;

    case WM_SIZE:
        width  = (GLsizei) LOWORD ( lParam );
        height = (GLsizei) HIWORD ( lParam );
        aspect = (GLdouble) width / (GLdouble) height;

        glMatrixMode ( GL_PROJECTION );
        glLoadIdentity ();
        gluPerspective ( 30.0, aspect, 1.0, 100.0 );

        glViewport ( 0, 0, width, height );

        return 0;

    case WM_PAINT:
        BeginPaint ( hWnd, &ps );
        drawScene  ( hdc, angle );
        EndPaint   ( hWnd, &ps );

        return 0;

    case WM_TIMER:
        if ( up )
        {
            angle += 2;
            if ( angle == 90 )
                up = FALSE;
        }
        else
        {
            angle -= 2;
            if ( angle == 0 )
                up = TRUE;
        }

        InvalidateRect ( hWnd, NULL, FALSE );

        return 0;
    case WM_DESTROY:
        wglMakeCurrent ( NULL, NULL );
        wglDeleteContext ( hrc );
```

```
            KillTimer ( hWnd, timer );
            PostQuitMessage ( 0 );

            return 0;

        }

        return DefWindowProc ( hWnd, msg, wParam, lParam );
}
void    setDCPixelFormat ( HDC hdc )
{
        static PIXELFORMATDESCRIPTOR pfd =
        {
            sizeof ( PIXELFORMATDESCRIPTOR ), // sizeof this structure
            1,          // version number
            PFD_DRAW_TO_WINDOW | PFD_SUPPORT_OPENGL,
            PFD_TYPE_RGBA,      // RGBA pixel values
            24,                 // 24-bit color
            0, 0, 0, 0, 0, 0,   // don't care about these
            0, 0,               // no alpha buffer
            0, 0, 0, 0, 0,      // no accumulation buffer
            32,                 // 32-bit depth buffer
            0,                  // no stencil buffer
            0,                  // no auxiliary buffers
            PFD_MAIN_PLANE,     // layer type
            0,                  // reserved (must be 0)
            0, 0, 0             // no layer masks
        };
        int  pixelFormat;

        pixelFormat = ChoosePixelFormat ( hdc, &pfd );
        SetPixelFormat ( hdc, pixelFormat, &pfd );
        DescribePixelFormat ( hdc, pixelFormat,
                    sizeof (PIXELFORMATDESCRIPTOR), &pfd );

        if ( pfd.dwFlags & PFD_NEED_PALETTE )
            MessageBox ( NULL, "Wrong video mode", "Attention", MB_OK );
}
void    initializeRC ()
{
        GLfloat lightAmbient  [] = { 0.1, 0.1, 0.1, 1.0 };
        GLfloat lightDiffuse  [] = { 0.7, 0.7, 0.7, 1.0 };
        GLfloat lightSpecular [] = { 0.0, 0.0, 0.0, 1.0 };

        glFrontFace  ( GL_CCW );
        glCullFace   ( GL_BACK );
        glEnable     ( GL_CULL_FACE );

        glDepthFunc  ( GL_LEQUAL );
        glEnable     ( GL_DEPTH_TEST );
        glClearColor ( 0.0, 0.0, 0.0, 0.0 );

        glLightfv ( GL_LIGHT0, GL_AMBIENT, lightAmbient );
        glLightfv ( GL_LIGHT0, GL_DIFFUSE, lightDiffuse );
        glLightfv ( GL_LIGHT0, GL_SPECULAR, lightSpecular );
```

```
    glEnable  ( GL_LIGHTING );
    glEnable  ( GL_LIGHT0 );
}
void     drawScene ( HDC hDC, int angle )
{
    GLfloat blue [] = { 0.0, 0.0, 1.0, 1.0 };
    GLfloat yellow [] = { 1.0, 1.0, 0.0, 1.0 };

    glClear ( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );

    glMatrixMode ( GL_MODELVIEW );
    glLoadIdentity ();
    glTranslatef ( 0.0, 0.0, -64.0 );
    glRotatef ( 30.0, 1.0, 0.0, 0.0 );
    glRotatef ( 30.0, 0.0, 1.0, 0.0 );
            // draw the block anchoring the arm
    glMaterialfv ( GL_FRONT, GL_AMBIENT_AND_DIFFUSE, yellow );
    drawBox ( 1.0, 3.0, -2.0, 2.0, -2.0, 2.0 );
    drawBox ( -3.0, -1.0, -2.0, 2.0, -2.0, 2.0 );

            // rotate the coordinate system and draw the arm's base member
    glMaterialfv ( GL_FRONT, GL_AMBIENT_AND_DIFFUSE, blue );
    glRotatef ( (GLfloat) angle, 1.0, 0.0, 0.0 );
    drawBox ( -1.0, 1.0, -1.0, 1.0, -5.0, 5.0 );

            // translate the coordinate system to
            // the end of base member, rotate it, and draw the second member
    glTranslatef ( 0.0, 0.0, -5.0 );
    glRotatef ( -(GLfloat) angle / 2.0, 1.0, 0.0, 0.0 );
    drawBox ( -1.0, 1.0, -1.0, 1.0, -10.0, 0.0 );

            // translate and rotate coordinate
            // system again and draw arm's third member
    glTranslatef ( 0.0, 0.0, -5.0 );
    glRotatef ( -(GLfloat) angle / 2.0, 1.0, 0.0, 0.0 );
    drawBox ( -1.0, 1.0, -1.0, 1.0, -10.0, 0.0 );

    glFlush ();   // render the scene to pixel buffer
}
```

Это лишь краткое изложение основных возможностей OpenGL. Достаточно подное описание ее потенциала содержится в [16].

Упражнения

Напишите программу, строящую правильные многогранники с правильным освещением.

Модифицируйте предыдущую программу для анимации сцены - все многогранники должны двигаться по различным траекториям.

Глава 13
ЭЛЕМЕНТЫ ВИРТУАЛЬНОЙ РЕАЛЬНОСТИ

Под виртуальной реальностью обычно понимается создание эффекта присутствия человека в несуществующем пространстве. Причем этот эффект должен быть как можно более полным. Для достижения этого используются различные аппаратные и программные средства.

Программа, которая создает наиболее полную иллюзию присутствия в имитируемом пространстве так, что на экране наблюдатель видит все то, что он мог бы увидеть на самом деле, и качество изображения настолько высоко, чтобы поддерживать эту иллюзию, - подобная программа должна включать в себя:

- эффективное и быстрое средство для удаления невидимых поверхностей;
- быстрый алгоритм для текстурирования граней с элементами расчета интенсивности.

При этом крайне желательно избежать обработки всех поверхностей, так как их обычно бывает очень много, т. е. иметь порядок $o(n)$, где n - общее количество граней.

Подавляющее большинство программ виртуальной реальности использует полигональное представление сцены. Для представления отдельных объектов привлекаются как полигональные модели, так и спрайты. Применение спрайтов позволяет заметно повысить быстродействие программы, правда за счет качества изображения (это особенно заметно вблизи объектов). Часто для спрайтов задают виды с нескольких сторон.

Практически все программы виртуальной реальности используют перспективную проекцию.

Важным требованием является возможность обеспечения достаточно высокой скорости рендеринга - количества кадров в секунду должно быть не менее 10-12 (иначе теряется иллюзия непрерывности движения).

13.1. Wolfenstein 3-D. Ray Casting

Одной из самых первых серьезных игр для IBM PC, использующей принципы (элементы) виртуальной реальности (не считая различных имитаторов полетов), мойжно считать игру W*o*lfe*n*stei*n* 3-D фирмы id*S*oftware. Эта игра имела огромный успех, и на протяжении длительного времени многие программисты обсуждали, как это можно было сделать. К данному моменту времени все исходные тексты этой игры уже известны, однако их большой объем затрудняет понимание принципов, лежащих в ее основе. Поэтому неудивительно, что мы начнем именно с этой игры.

В W*o*lfe*n*stei*n* 3-D вся сцена представляет собой прямоугольный лабиринт (вершины стен лежат на прямоугольной сетке, и стены могут быть либо вертикальными, либо горизонтальными) с постоянной высотой пола и потолка. Размер лабиринта (количество клеток) в самой игре был 64x64, однако мы рассмотрим уменьшенный вариант.

Лабиринт задается прямоугольной матрицей, где каждая клетка либо свободна, либо содержит стену с заданной текстурой (рис. 13.1). Можно считать, что лабиринт

как бы строится из одинаковых кубиков с текстурированными боковыми гранями. При этом пол и потолок текстуры не имеют.

При этом игрок может свободно перемещаться по всему лабиринту (рис. 13.2), однако высота глаз над уровнем пола все время остается неизменной.

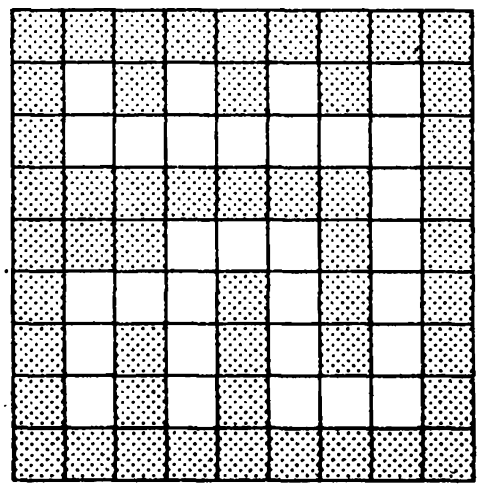


Рис. 13.1

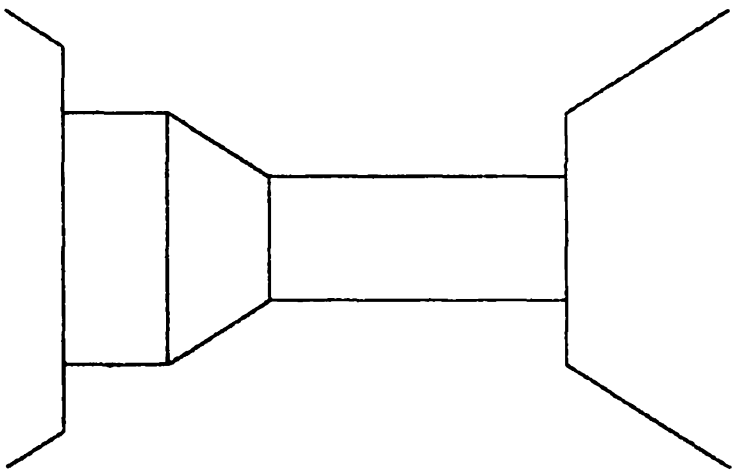


Рис. 13.2

Использованный алгоритм фактически является модификацией метода построчного сканирования, однако если традиционный алгоритм осуществляет построение изображения по строкам, то в данном случае более эффективным является подход, основанный на построении изображения по столбцам. Это связано с тем, что при пересечении лабиринта плоскостью, соответствующей горизонтальной строке пикселов, возникает большое количество различных отрезков, подавляющая часть которых игроку не видна. Но если вместо строк использовать столбцы, то ситуация намного упрощается (рис. 13.3) - не только заметно сокращается количество пересечений, но существенным является только ближайшее из них (ясно, что пересечение с полом и с потолком на самом деле можно не рассматривать), так как оно закроет собой все остальные пересечения. Так как высота пола постоянна, для правильной отрисовки столбца достаточно знать лишь расстояние до ближайшей вдоль плоскости стены (вопрос о текстуре будет рассмотрен позже).

Посмотрим на сцену сверху: стенам соответствуют вертикальные и горизонтальные отрезки, а каждой секущей плоскости - луч, выходящий из положения наблюдателя. Тем самым задачу определения ближайшей стены для каждого столбца можно решать не в пространстве, а на плоскости *Oxy*.

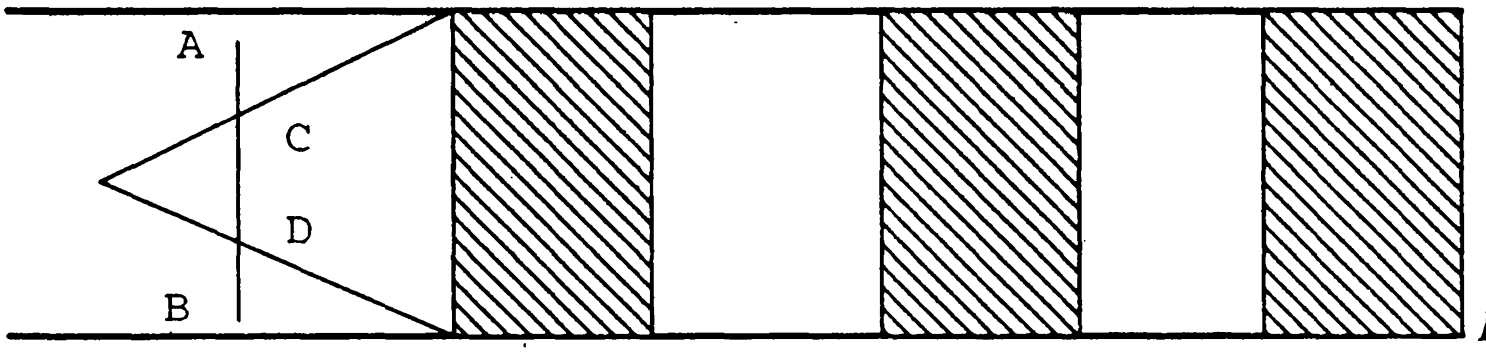


Рис. 13.3

Фактически мы приходим к двумерной трассировке лучей (ray casting) - из точки, соответствующей положению игрока на карте, выпускается пучок лучей и для каждого из них находится ближайшее пересечение со стенами лабиринта (рис. 13.4).

Самый простой подход к задаче о нахождении ближайшего пересечения луча со стенами лабиринта (проверить все стены на пересечение с лучом и выбрать среди точек пересечения ближайшую) является слишком медленным и неэффективным. Удобнее воспользоваться тем, что стены являются сегментами линий сетки. Для этого пересекаемые

лучом клетки отслеживаются в порядке распространения луча из начальной клетки, содержащей игрока, до тех пор, пока не будет встречена непустая клетка.

Наиболее простой реализацией подобного подхода является следующая: сначала на пересечение с лучом проверяются вертикальные стены (параллельные оси Ox), затем проверяются горизонтальные (параллельные оси Oy) стены и выбирается ближайшее пересечение.

Рассмотрим этот процесс подробнее.

Пусть игрок находится в точке (x^*, y^*) и угол между лучом и положительным направлением оси Ox равен α. Будем считать, что клетка, содержащая в себе игрока, имеет индекс (i^*, j^*), а шаг сетки равен h.

Найдем точки пересечения луча с вертикальными линиями $x = ih$. Если направляющий вектор луча имеет положительную x-составляющую ($\cos\alpha > 0$, т. е. $-\pi/2 \le \alpha \le \pi/2$), то ближайшая точка будет иметь координаты

$$x_1 = (i^* + 1)h,$$

$$y_1 = y^* + \Delta x \tan\alpha = y^* + (x_1 - x^*)\tan\alpha$$

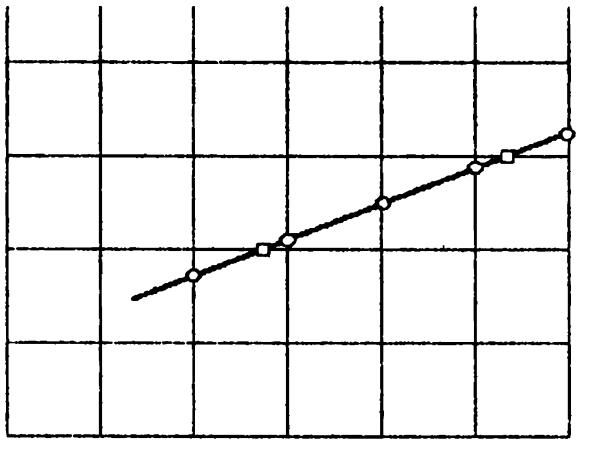


Рис. 13.4

где $\Delta x = h - (x^* - i^*h) = x_1 - x^*$ - расстояние вдоль оси Ox от положения игрока до ближайшей вертикальной линии справа.

При этом эта точка лежит на границе клетки $\left(i^* + 1, \left[\frac{y_1}{h}\right]\right)$

Если эта клетка занята стеной, то мы сразу получаем пересечение со стеной в точке (x_1, y_1). При этом расстояние до точки пересечения будет равно

$$d = \frac{\Delta x}{\cos\alpha} = \frac{x_1 - x^*}{\cos\alpha}.$$

Если эта клетка пуста, то проверяем следующие точки:

$x_{i+1} = x_i + h,$

$y_{i+1} = y_i + h\tan a;$

здесь i-я точка соответствует клетке $\left(i^* + i, \left[\frac{y_i}{h}\right]\right)$;

проверка проводится до тех пор, пока мы либо не наткнемся на занятую клетку, либо не выйдем за пределы лабиринта.

Если клетка, соответствующая точке пересечения (x_i, y_i), занята, то расстояние до этой точки задается формулой

$$d = \frac{\Delta x + (i-1)h}{\cos\alpha} = \frac{x_i - x^*}{\cos\alpha}$$

Рассмотрим теперь случай, когда

$$\cos\alpha < 0, \left(\frac{\pi}{2} \le \alpha \le \frac{3\pi}{2}\right).$$



Рис. 13.5

Ближайшая точка пересечения луча с линией вида $x = ih$ описывается формулами

$x_1 = i^* h$,

$y_1 = y^* - \Delta x \tan a = y^* + (x_1 - x^*) \tan a$;

где $\Delta x = x^* - x_1$.

Первой проверяется клетка $\left(i^* - 1, \left[\frac{y_1}{h}\right]\right)$.

Если она занята, то пересечение найдено и расстояние до точки пересечения равно

$$d = \frac{-\Delta x}{\cos\alpha} = \frac{x_1 - x^*}{\cos\alpha}$$

В противном случае необходимо проверить остальные точки пересечения:

$x_{i+1} = x_i - h$,

$y_{i+1} = y_i - h \tan a$;

Для точки (x_i, y_i) следует проверить клетку $\left(i^* - i, \left[\frac{y_i}{h}\right]\right)$, и в случае, если занята, расстояние до точки пересечения составит

$$d = \frac{\Delta x + (i-1)h}{-\cos\alpha} = \frac{x_i - x^*}{\cos\alpha}$$

В результате мы приходим к следующему алгоритму:

```
float     checkVWalls ( float angle )
{
    int       xTile = (int) locX;    // cell indices
    int       yTile = (int) locY;
    float     xIntercept;            // intercept point
    float     yIntercept;
    float     xStep;                 // intercept steps
    float     yStep;
    int       dxTile;
                                     // check for vertical rays
    if ( fabs ( cos ( angle ) ) < 1e-7 )
        return 10000.0;
    if ( angle >= ANGLE_270 || angle <= ANGLE_90 )
    {
      xTile ++;
      xStep = 1;
      yStep = tan ( angle );
      xIntercept = xTile;
      dxTile = 1;
```

```
        }
        else
        {
          xTile --;
          xStep = -1;
          yStep = -tan ( angle );
          xIntercept = xTile + 1;
          dxTile = -1;
        }
                              // find interception point
        yIntercept = locY + ( xIntercept - locX ) * tan ( angle );

        for ( ; ; )
        {
          yTile = yIntercept;
          if ( xTile<0 || xTile>MAX_XTILE )
            return 10000.0;
          if ( yTile<0 || yTile>MAX_YTILE )
            return 10000.0; // out of map

          if ( map [yTile][xTile] != ' ' )
            return ( xIntercept - locX ) / cos ( angle );

          xIntercept   += xStep;
          yIntercept   += yStep;
          xTile        += dxTile;
        }
    }
```

Аналогично определяется ближайшая точка пересечения луча с горизонтал ми линиями сетки $y = jh$.

Если $0 \le a \le \pi$, то

$$x_1 = x^* + \left(y_1 - y^*\right)ctg\alpha,$$

$$y_1 = \left(j^* + 1\right)h,$$

$$x_{i+1} = x_i + hctg\alpha,$$

$$y_{i+1} = y_i + h,$$

а в случае $\pi \le \alpha \le 2\pi$

$$x_1 = x^* + \left(y_1 - y^*\right)ctg\alpha,$$

$$y_1 = \left(j^* - 1\right)h,$$

$$x_{i+1} = x_i - hctg\alpha,$$

$$y_{i+1} = y_i - h.$$

Соответствующая проверка осуществляется процедурой

```
float     checkHWalls ( float angle )
{
     int  xTile = (int) locX;
```

```
int  yTile = (int) locY;
float     xIntercept;
float     yIntercept;
float     xStep;
float     yStep;
int  dyTile;

if ( fabs ( sin ( angle ) ) < 1e-7 )
     return 10000.0;

if ( angle <= ANGLE_180 )
{
     yTile ++;
     xStep = 1 / tan ( angle );
     yStep = 1;
     yIntercept = yTile;
     dyTile = 1;
}
else
{
     yTile --;
     yStep = -1;
     xStep = -1 / tan ( angle );
     yIntercept = yTile + 1;
     dyTile = -1;
}

xIntercept = locX + ( yIntercept - locY ) / tan ( angle );

     for ( ; ; )
     {
          xTile = xIntercept;
          if ( xTile<0 || xTile >52 || yTile<0 || yTile>9 )
               return 10000.0;

          if ( map [yTile][xTile] != ' ' )
               return ( yIntercept - locY ) / sin ( angle );

          xIntercept += xStep;
          yIntercept += yStep;
          yTile      += dyTile;
     }
}
```

После того как найдены ближайшие точки пересечения луча как с вертикальными, так и с горизонтальными стенами, среди них выбирается наименее удаленная от игрока.

Существует модификация описанного подхода, когда сразу проверяются все пересечения луча, без отдельного рассмотрения пересечений с вертикальными и горизонтальными стенами. При этом последовательный переход из клетки в клетку осуществляется до тех пор, пока не будет найдена непустая клетка.

Подобный подход и был реализован в игре Wolfenstein 3-D. Вариант подобного подхода можно найти на компакт-диске.

Однако данный подход заметно сложнее предложенного выше, почти не отличаясь от него по быстродействию.

Рассмотрим теперь, каким образом по найденному расстоянию d до точки пересечения строится соответствующий столбец пикселов.

Пусть глаз наблюдателя расположен по высоте посередине между полом и потолком и расстояние от игрока до картинной плоскости равно f (см. рис. 13.3). Тогда столбцу пикселов на картинной плоскости соответствует отрезок AB, состоящий из трех частей: проекции потолка AC, проекции пола DB и собственно проекции стены CD, причем $AC = DB$. Длина отрезка CD (высота проекции стены в пикселах) легко находится из равенства

$$|CD| = \frac{SCREEN_HEIGHT * f}{d},$$

$$|AC| = |DB| = \frac{SCREEN_HEIGHT - |CD|}{2}$$

Простейший вариант программы, реализующий описанный алгоритм, приведен ниже.

```
void    drawView ()
{
    float    phi;
    float    distance;

    totalFrames++;
    for ( int col = 0; col < SCREEN_WIDTH; col++ )
    {
        phi = angle + rayAngle [col];
        if ( phi < 0 )
            phi += 2 * M_PI;
        else
        if ( phi >= 2 * M_PI )
            phi -= 2 * M_PI;

        float    d1 = checkVWalls ( phi );
        float    d2 = checkHWalls ( phi );

        distance = d1;

        if ( d2 < distance )
            distance = d2;

        distance *= cos ( phi - angle );
        drawSpan ( distance, WALL_COLOR, col );
    }
}
```

Для задания углов лучей здесь используется массив ray*A*ngle - для каждого луча указан угол между направлением луча и направлением взгляда игрока. Наиболее простым способом описания этого массива является задание углов с постоянным шагом, но, так как этот способ вызывает некоторые искажения, рекомендуется следующий метод:

rayAngle[i]=arctg(tg(VIEW_ANGLE/2)*(1+i/(SCREEN_WIDTH-1)))

Приводимая ниже программа реализует данный алгоритм.

```
// File wolf1.cpp
```

```
#include    <bios.h>
#include    <dos.h>
#include    <math.h>
#include    <stdio.h>
#include    <time.h>

#define  VIEW_WIDTH      ( M_PI / 3 )    // viewing angle ( 60 degrees )
#define  SCREEN_WIDTH    320             // size of rendering window
#define  SCREEN_HEIGHT   200             // basic colors
#define  FLOOR_COLOR     8
#define  CEILING_COLOR   3
#define  WALL_COLOR      1               // angles
#define  ANGLE_90        M_PI_2          // 90 degrees
#define  ANGLE_180       M_PI            // 180 degrees
#define  ANGLE_270       (M_PI*1.5)      // 270 degrees
                                         // bios key defintions
#define  ESC             0x011b
#define  UP              0x4800
#define  DOWN            0x5000
#define  LEFT            0x4b00
#define  RIGHT           0x4d00
                                         // labyrinth
char  * worldMap [] =
{
    "**************************************************",
    "*                  *                   *",
    "*  * * * * * * * *  ************************* *",
    "*                  *                   *",
    "*  * * * * * * * *  * ********************** *",
    "*                                  * *",
    "*  * * * * * * * *  ************************* *",
    "*                  *                   *",
    "**************************************************"
};

float       swing;
float       locX;
float       locY;
float       angle;
long        totalFrames = 0l;
char far *  screenPtr    = (char far *) MK_FP ( 0xA000, 0 );
float       rayAngle [SCREEN_WIDTH];
                                         // angles for rays from main
                                         // viewing direction

////////////////////////// Functions ////////////////////////////////

void     drawSpan ( float dist, char wallColor, int x )
{
    char far * vptr = screenPtr + x;
    int     h    = dist >= 0.5 ? (int)(SCREEN_HEIGHT*0.5 / dist )
 SCREEN_HEIGHT;
    int     j1   = ( SCREEN_HEIGHT - h ) / 2;
    int     j2   = j1 + h;
```

```
    for ( int j = 0; j < j1; j++, vptr += 320 )
        * vptr = CEILING_COLOR;

    if ( j2 > SCREEN_HEIGHT )
        j2 = SCREEN_HEIGHT;

    for ( , j < j2; j++, vptr += 320 )
        * vptr = wallColor;

    for ( ; j < SCREEN_HEIGHT; j++, vptr += 320 )
        * vptr = FLOOR_COLOR;
}

float   checkVWalls ( float angle )
{
    int     xTile = (int) locX;         // cell indices
    int     yTile = (int) locY;
    float   xIntercept;                 // intercept point
    float   yIntercept;
    float   xStep;                      // intercept steps
    float   yStep;
    int     dxTile;

    if ( fabs ( cos ( angle ) ) < 1e-7 )
        return 10000.0;

    if ( angle >= ANGLE_270 || angle <= ANGLE_90 )
    {
        xTile ++;
        xStep       = 1;
        yStep       = tan ( angle );
        xIntercept  = xTile;
        dxTile      = 1;
    }
    else
    {
        xTile --;
        xStep       = -1;
        yStep       = -tan ( angle );
        xIntercept  = xTile + 1;
        dxTile      = -1;
    }
                                        // find interception point
    yIntercept = locY + (xIntercept - locX) * tan ( angle );

    for ( ; ; )
    {
        yTile = yIntercept;
        if ( xTile < 0 || xTile > 52 || yTile < 0 || yTile > 9 )
            return 10000.0;

        if ( worldMap [yTile][xTile] != ' ' )
            return ( xIntercept - locX ) / cos ( angle );

        xIntercept  += xStep;
        yIntercept  += yStep;
        xTile       += dxTile;
```

```
    }
}
float     checkHWalls ( float angle )
{
    int  xTile = (int) locX;
    int  yTile = (int) locY;
    float     xIntercept;
    float     yIntercept;
    float     xStep;
    float     yStep;
    int  dyTile;

    if ( fabs ( sin ( angle ) ) < 1e-7 )
        return 10000.0;

    if ( angle <= ANGLE_180 )
    {
        yTile ++;
        xStep = 1 / tan ( angle );
        yStep = 1;
        yIntercept = yTile;
        dyTile = 1;
    }
    else
    {
        yTile --;
        yStep = -1;
        xStep = -1 / tan ( angle );
        yIntercept = yTile + 1;
        dyTile = -1;
    }

    xIntercept = locX + (yIntercept - locY) / tan ( angle );

    for ( ; ; )
    {
        xTile = xIntercept;
        if ( xTile < 0 || xTile > 52 || yTile < 0 || yTile > 9 )
            return 10000.0;

        if ( worldMap [yTile][xTile] != ' ' )
            return ( yIntercept - locY ) / sin ( angle );

        xIntercept += xStep;
        yIntercept += yStep;
        yTile      += dyTile;
    }
}
void      drawView ()
{
    float     phi;
    float     distance;

    totalFrames++;
    for ( int col = 0; col < 320; col++ )
```

```
    {
        phi = angle + rayAngle [col];

        if ( phi < 0 )
            phi += 2 * M_PI;
        else
        if ( phi >= 2 * M_PI )
            phi -= 2 * M_PI;

        float     d1 = checkVWalls ( phi );
        float     d2 = checkHWalls ( phi );

        distance = d1;

        if ( d2 < distance )
            distance = d2;

        distance *= cos ( phi - angle );   // adjustment for fish-eye
        drawSpan ( distance, WALL_COLOR, col );
    }
}
void    setVideoMode ( int mode )
{
    asm {
        mov     ax, mode
        int  10h
    }
}
void    initTables ()
{
    for ( int i = 0; i < SCREEN_WIDTH; i++ )
        rayAngle [i] = atan ( -swing + 2 * i * swing / ( SCREEN_WIDTH - 1 ) );
}
main ()
{
    int  done = 0;

    angle = 0;
    locX  = 1.5;
    locY  = 1.5;
    swing = tan ( VIEW_WIDTH / 2 );

    initTables ();
    setVideoMode ( 0x13 );

    int  start = clock ();

    while ( !done )
    {
        drawView ();

        if ( bioskey ( 1 ) )
        {
            float     vx = cos ( angle );
            float     vy = sin ( angle );
            float     x, y;
```

```
switch ( bioskey ( 0 ) )
{
    case LEFT:
        angle -= 5.0*M_PI/180.0;
        break;

    case RIGHT:
        angle += 5.0*M_PI/180.0;
        break;

    case UP:
        x = locX + 0.3 * vx;
        y = locY + 0.3 * vy;

        if ( worldMap [(int) y][(int) x] == ' ' )
        {
            locX = x;
            locY = y;
        }

        break;

    case DOWN:
        x = locX - 0.3 * vx;
        y = locY - 0.3 * vy;

        if ( worldMap [(int) y][(int) x ] == ' ' )
        {
            locX = x;
            locY = y;
        }

        break;

    case ESC:
        done = 1;
    }

    if ( angle < 0 )
        angle += 2 * M_PI;
    else
    if ( angle >= 2 * M_PI )
        angle -= 2 * M_PI;
  }
}

float    totalTime = ( clock () - start ) / CLK_TCK;

setVideoMode ( 0x03 );

printf ( "\nFrames rendered   : %ld",  totalFrames );
printf ( "\nTotal time ( sec )   %7.2f", totalTime );
printf ( "\nFPS                  %7.2f", totalFrames / totalTime );
}
```

Для получения каркасного изображения приведенную программу несложно дифицировать.

```
// File wolf2.cpp
```

```
#include    <bios.h>
#include    <dos.h>
#include    <math.h>
#include    <stdio.h>
#include    <time.h>

#define   VIEW_WIDTH        ( M_PI / 3 )    // viewing angle ( 60 degrees )
#define   SCREEN_WIDTH      320             // size of rendering window
#define   SCREEN_HEIGHT     200             // basic colors
#define   FLOOR_COLOR       0
#define   CEILING_COLOR     0
#define   WALL_COLOR        1
                                            // angles
#define   ANGLE_90          M_PI_2          // 90 degrees
#define   ANGLE_180         M_PI            // 180 degrees
#define   ANGLE_270         (M_PI*1.5)      // 270 degrees
                                            // bios key defintions
#define   ESC       0x011b
#define   UP        0x4800
#define   DOWN      0x5000

#define   LEFT      0x4b00
#define   RIGHT     0x4d00
                                            // labyrinth
char  * worldMap [] =
{
    "**********************************************************"
    "**                 *                    *"
    "**  * * * * * * * *  ************************* **"
    "**                 *                    *"
    "**  * * * * * * * *  * ********************** **"
    "**                                  * **"
    "**  * * * * * * * *  ************************* **"
    "**                 *                    *"
    "**********************************************************"
};

float       swing;
float       locX;
float       locY;
float       angle;
long        totalFrames = 0l;
char far * screenPtr   = (char far *) MK_FP ( 0xA000, 0 );
float       rayAngle [SCREEN_WIDTH];      // angles for rays from main
// viewing direction
int     lastCell = -1;
int     lastJ1   = 0;
int     lastJ2   = 0;

//////////////////////// Functions ////////////////////////////////

void      drawSpan ( float dist, char wallColor, int x, int cell )
{
    char far * vptr = screenPtr + x;
```

```
int     h    = dist >= 0.5 ? (int)(SCREEN_HEIGHT*0.5 / dist )  SCREEN_HEIGHT;
int     j1   = ( SCREEN_HEIGHT - h ) / 2;
int     j2   = j1 + h;

if ( x > 0 && cell != lastCell )// check for wall boundary
{
     if ( lastJ1 > j1 )
          lastJ1 = j1;

     if ( lastJ2 < j2 )
           lastJ2 = j2;

     for ( int j = 0; j < lastJ1; j++, vptr += 320 )
          * vptr = CEILING_COLOR;

     for ( ; j < lastJ2; j++, vptr += 320 )
          * vptr = wallColor;

     for ( ; j < SCREEN_HEIGHT; j++, vptr += 320 )
          * vptr = CEILING_COLOR;
}
else
{
     for ( int j = 0; j < SCREEN_HEIGHT; j++, vptr += 320 )
          * vptr = CEILING_COLOR;

     * ( screenPtr + x + 320*j1 ) = wallColor;
     * ( screenPtr + x + 320*j2 ) = wallColor;
}
     lastCell = cell;
     lastJ1   = j1;
     lastJ2   = j2;
}

float     checkVWalls ( float angle, int& cell )
{
     int     xTile = (int) locX;
     int     yTile = (int) locY;
     float   xIntercept;
     float   yIntercept;
     float   xStep;
     float   yStep;
     int     dxTile;

     if ( fabs ( cos ( angle ) ) < 1e-7 )
          return 10000.0;

     if ( angle >= ANGLE_270 || angle <= ANGLE_90 )
     {
          xTile ++;
          xStep        = 1;
          yStep        = tan ( angle );
          xIntercept   = xTile;
          dxTile       = 1;
     }
     else
     {
```

```
            xTile --;
            xStep       = -1;
            yStep       = -tan ( angle );
            xIntercept  = xTile + 1;
            dxTile      = -1;
        }

        yIntercept = locY + ( xIntercept - locX ) * tan ( angle );

        for ( ; ; )
        {
            yTile = yIntercept;
            if ( xTile < 0 || xTile > 52 || yTile < 0 || yTile > 9 )
                return 10000.0;

            if ( worldMap [yTile][xTile] != ' ' )
            {
                cell = xTile;
                return ( xIntercept - locX ) / cos ( angle );
            }

            xIntercept += xStep;
            yIntercept += yStep;
            xTile      += dxTile;
        }
    }

    float    checkHWalls ( float angle, int& cell )
    {
    int    xTile = (int) locX;
    int    yTile = (int) locY;
    float  xIntercept;
    float  yIntercept;
    float  xStep;
    float  yStep;
    int    dyTile;

    if ( fabs ( sin ( angle ) ) < 1e-7 )
        return 10000.0;

    if ( angle <= ANGLE_180 )
    {
        yTile ++;
        xStep       = 1 / tan ( angle );
        yStep       = 1;
        yIntercept  = yTile;
        dyTile      = 1;
    }
    else
    {
        yTile --;
        yStep       = -1;
        xStep       = -1 / tan ( angle );
        yIntercept  = yTile + 1,
        dyTile      = -1;
    }
```

```
    xIntercept = locX + ( yIntercept - locY ) / tan ( angle );

    for ( ; ; )
    {
        xTile = xIntercept;
        if ( xTile < 0 || xTile > 52 || yTile < 0 || yTile > 9 )
            return 10000.0;

        if ( worldMap [yTile][xTile] != ' ' )
        {
            cell = 1000 + yTile;

            return ( yIntercept - locY ) / sin ( angle );
        }

        xIntercept += xStep;
        yIntercept += yStep;
        yTile      += dyTile;
    }
}
void    drawView ()
{
    float    phi;
    float    distance;

    totalFrames++;
    lastCell = -1;

    for ( int col = 0; col < 320; col++ )
    {
        phi = angle + rayAngle [col];
        if ( phi < 0 )
            phi += 2 * M_PI;
        else
        if ( phi >= 2 * M_PI )
            phi -= 2 * M_PI;

        int  c1, c2;
        float    d1 = checkVWalls ( phi, c1 );
        float    d2 = checkHWalls ( phi, c2 );

        distance = d1;

        if ( d2 < distance )
        {
            distance = d2;
            c1       = c2;
        }

        distance *= cos ( phi - angle );   // adjustment for fish-eye

        drawSpan ( distance, WALL_COLOR, col, c1 );
    }
}
void    setVideoMode ( int mode )
{
    asm {
```

```
            mov     ax, mode
            int  10h
        }
}
void    initTables ()
{
    for ( int i = 0; i < SCREEN_WIDTH; i++ )
        rayAngle [i] = atan ( -swing + 2 * i * swing / ( SCREEN_WIDTH - 1 ) );
}
main ()
{
    int  done = 0;

    angle = 0.61;
    locX  = 2.25;
    locY  = 4.85;
    swing = tan ( VIEW_WIDTH / 2 );

    initTables ();
    setVideoMode ( 0x13 );

    int  start = clock ();

    while ( !done )
    {
        drawView ();

        if ( bioskey ( 1 ) )
{
    float    vx = cos ( angle );
    float    vy = sin ( angle );
    float    x, y;

    switch ( bioskey ( 0 ) )
    {
        case LEFT:
            angle -= 5.0*M_PI/180.0;
            break;

        case RIGHT:
            angle += 5.0*M_PI/180.0;
            break;

        case UP:
            x = locX + 0.3 * vx;
            y = locY + 0.3 * vy;

            if ( worldMap [(int) y][(int) x] == ' ' )
            {
                locX = x;
                locY = y;
            }

            break;

        case DOWN:
            x = locX - 0.3 * vx;
```

```
                    y = locY - 0.3 * vy;
                    if ( worldMap [(int) y][(int) x ] == ' ' )
                    {
                        locX = x;
                        locY = y;
                    }
                    break;
                case ESC:
                    done = 1;
            }
            if ( angle < 0 )
                angle += 2 * M_PI;
            else
            if ( angle >= 2 * M_PI )
                angle -= 2 * M_PI;
        }
    }
    float    totalTime = ( clock () - start ) / CLK_TCK;
    setVideoMode ( 0x03 );
    printf ( "\nFrames rendered   : %ld",  totalFrames );
    printf ( "\nTotal time ( sec ) : %7.2f", totalTime );
    printf ( "\nFPS                %7.2f", totalFrames / totalTime );
}
```

Одним из подводных камней, встречающихся при реализации описанного алгоритма, является так называемый fish-eye-эффект - изображение на экране сильно искажается и вместо прямых линий в местах соединения пола (потолка) со стенами видны дуги. Причина подобного явления кроется в неправильном осуществлении перспективного проектирования (при перспективном проектировании образом прямой линии всегда является прямая).

Рассмотрим несколько столбцов пикселов и соответствующие им лучи (рис. 13.6). Вследствие того что изображение строится на плоском экране, высоты столбцов пикселов, соответствующих стенам, получаются как отношение расстояния до стены к расстоянию до экрана f.

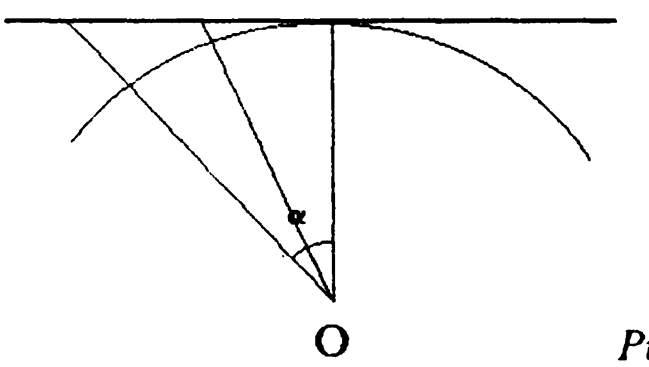


Рис. 13.6

Если столбец соответствует центральному лучу, то оба эти расстояния совпадают. Если же луч отличается от центрального на угол α, то правильное фокусное расстояние равно $f/\cos\alpha$.

Самым простым способом борьбы с этим явлением является коррекция найденного расстояния *d* - оно домножается на косинус угла между лучом и направлением взгляда.

Основной процедурой этой программы является *d*rawView, строящая изображение сцены, видимой из точки (*locX*, *locY*), в заданном направлении ang*l*e; конечно, ее быстродействие оставляет желать лучшего. Причины этого кроются как в неоптимизированности алгоритма, так и в использовании операций с плавающей точкой (f*lo*at).

В данной программе можно легко отказаться от использования вещественных (*float*) чисел (что и было сделано в самой игре), применяя числа с фиксированной точкой (см. прил.). Соответствующая программа приводится ниже. В связи с широким использованием 32-битовых чисел желательно в опциях компилятора указать генерацию кода для 386-го процессора.



```
// File wolf3.cpp

#include    <bios.h>
#include    <dos.h>
#include    <math.h>
#include    <stdio.h>
#include    <time.h>
#include    "fixmath.h"

#define   VIEW_WIDTH        (M_PI / 3)    // viewing angle ( 60 degrees )
#define   SCREEN_WIDTH      320           // size of rendering window
#define   SCREEN_HEIGHT     200           // basic colors
#define   FLOOR_COLOR       8
#define   CEILING_COLOR     3
#define   WALL_COLOR        1             // bios key defintions
#define   ESC               0x011b
#define   UP                0x4800
#define   DOWN              0x5000
#define   LEFT              0x4b00
#define   RIGHT             0x4d00          // labyrinth
char  * worldMap [] =
{
    "**************************************************",
    "*                  *                   *",
    "*  * * * * * * * *  ************************* *",
    "*                  *                   *",
    "*  * * * * * * * *  * ********************** *",
    "*                                 * *",
    "*  * * * * * * * *  ************************* *",
    "*                  *                   *",
    "**************************************************"
};

float       swing;
Fixed       locX;
Fixed       locY;
Angle       angle;
long         totalFrames = 0l;
char far * screenPtr   = (char far *) MK_FP ( 0xA000, 0 );
Angle       rayAngle [SCREEN_WIDTH]; // angles for rays from main
// viewing direction

//////////////////////////// Functions ////////////////////////////////////

void      drawSpan ( Fixed dist, char wallColor, int x )
{
    char far * vptr = screenPtr + x;
    int     h    = dist >= (ONE / 2) ? fixed2Int (
              (int2Fixed (SCREEN_HEIGHT/2) << 8) / (dist >> 8))
```

```
            SCREEN_HEIGHT;
    int     j1   = ( SCREEN_HEIGHT - h ) / 2;
    int     j2   = j1 + h;

    for ( int j = 0; j < j1; j++,.vptr += 320 )
        * vptr = CEILING_COLOR;

    if ( j2 > SCREEN_HEIGHT )
        j2 = SCREEN_HEIGHT;

    for ( ; j < j2; j++, vptr += 320 )
        * vptr = wallColor;

    for ( ; j < SCREEN_HEIGHT; j++, vptr += 320 )
        * vptr = FLOOR_COLOR;
}

Fixed   checkVWalls ( Angle angle )
{
    int       xTile = fixed2Int ( locX );       // cell indices
    int       yTile = fixed2Int ( locY );
    Fixed     xIntercept;                       // intercept point
    Fixed     yIntercept;
    Fixed     xStep;                            // intercept steps
    Fixed     yStep;
    int       dxTile;

    if ( fixAbs ( coTang ( angle ) ) < 2 )
        return MAX_FIXED;

if ( angle >= ANGLE_270 || angle <= ANGLE_90 )
{
    xTile ++;
    xStep       = ONE;
    yStep       = tang     (angle);
    xIntercept  = int2Fixed (xTile);
    dxTile      = 1;
}
else
{
    xTile --;
    xStep       = -ONE;
    yStep       = -tang     (angle);
    xIntercept  =  int2Fixed (xTile + 1);
    dxTile      = -1;
}
                    // find interception point
yIntercept = locY + ((xIntercept - locX) << 15) / (coTang (angle) >> 1);

for ( ; ; )
{
    yTile = fixed2Int (yIntercept);
    if ( xTile < 0 || xTile > 52 || yTile < 0 || yTile > 9 )
        return MAX_FIXED;

    if ( worldMap [yTile][xTile] != ' ' )
        return ((xIntercept - locX) >> 8) * (invCosine ( angle ) >> 8);
```

```
            xIntercept += xStep;
            yIntercept += yStep;
            xTile      += dxTile;
        }
    }
    Fixed   checkHWalls ( Angle angle )
    {
        int     xTile = fixed2Int ( locX );
        int     yTile = fixed2Int ( locY );
        Fixed   xIntercept;
        Fixed   yIntercept;
        Fixed   xStep;
        Fixed   yStep;
        int     dyTile;

        if ( fixAbs ( tang ( angle ) ) < 2 )
            return MAX_FIXED;

        if ( angle <= ANGLE_180 )
        {
            yTile ++;
            xStep       = coTang (angle);
            yStep       = ONE;
            yIntercept  = int2Fixed (yTile);
            dyTile      = 1;
        }
        else
        {
            yTile --;
            yStep       = -ONE;
            xStep       = -coTang ( angle );
            yIntercept  = int2Fixed (yTile + 1);
            dyTile      = -1;
        }

        xIntercept = locX + ((yIntercept - locY) << 15) / (tang (angle) >> 1);

        for ( ; ; )
        {
            xTile = fixed2Int (xIntercept);
            if ( xTile < 0 || xTile > 52 || yTile < 0 || yTile > 9 )
                return MAX_FIXED;

            if ( worldMap [yTile][xTile] != ' ' )
                return ((yIntercept - locY) >> 8) * (invSine ( angle ) >> 8);

            xIntercept += xStep;
            yIntercept += yStep;
            yTile      += dyTile;
        }
    }
    void    drawView ()
    {
        Angle   phi;
```

```
    Fixed    distance;

    totalFrames++;
    for ( int col = 0; col < 320; col++ )
    {
        phi = angle + rayAngle [col];

        Fixed    d1 = checkVWalls ( phi );
        Fixed    d2 = checkHWalls ( phi );

        distance = d1;

        if ( d2 < distance )
            distance = d2;
                        // adjustment for fish-eye
        distance = (distance >> 8) * (cosine ( phi - angle ) >> 8);

        drawSpan ( distance, WALL_COLOR, col );
    }
}
void     setVideoMode ( int mode )
{
    asm {
        mov     ax, mode
        int  10h
    }
}
void     initTables ()
{
    initFixMath ();

    for ( int i = 0; i < SCREEN_WIDTH; i++ )
        rayAngle [i] = rad2Angle ( atan ( -swing + 2 * i * swing /
  (SCREEN_WIDTH - 1) ) );
}
main ()
{
    int  done = 0;
    Fixed    ct   = float2Fixed ( 0.3 );
    Angle    da   = 5 * (ANGLE_90 / 90);

    angle = 0;
    locX  = float2Fixed ( 1.5 );
    locY  = float2Fixed ( 1.5 );
    swing = tan ( VIEW_WIDTH / 2 );
    locX  = 375476;
    locY  = 358224;
    angle = 60084U;

    initTables   ();
    setVideoMode ( 0x13 );

    int  start = clock ();

    while ( !done )
    {
```

```
    drawView ();
    if ( bioskey ( 1 ) )
    {
        Fixed   vx = cosine ( angle );
        Fixed   vy = sine    ( angle );
        Fixed   x, y;

        switch ( bioskey ( 0 ) )
        {
            case LEFT:
                angle -= da;
                break;

            case RIGHT:
                angle += da;
                break;
            case UP:
                x = locX + (ct >> 8) * (vx >> 8);
                y = locY + (ct >> 8) * (vy >> 8);

                if ( worldMap [fixed2Int (y)][fixed2Int (x)] == ' ' )
                {
                    locX = x;
                    locY = y;
                }
                break;

            case DOWN:
                x = locX - (ct >> 8) * (vx >> 8);
                y = locY - (ct >> 8) * (vy >> 8);

                if ( worldMap [fixed2Int (y)][fixed2Int (x)] == ' ' )
                {
                    locX = x;
                    locY = y;
                }
                break;

            case ESC:
                done = 1;
            }
        }
    }
    float     totalTime = ( clock () - start ) / CLK_TCK;

    setVideoMode ( 0x03 );

    printf ( "\nFrames rendered    : %ld",   totalFrames );
    printf ( "\nTotal time ( sec )   %7.2f", totalTime );
    printf ( "\nFPS                  %7.2f", totalFrames / totalTime );
}
```

Обратите внимание на вычисление первой точки пересечения луча с вертикальной и горизонтальной стенами - там используются несколько иные формулы для ления чисел с фиксированной точкой 16.16, чем в приложении. Несложно, одна

убедиться в правильности этих формул, а также и в том, что их использование в этом месте оправданно - они обеспечивают большую точность.

Замечание. Ряд современных процессоров (Pentium, Pentium II, Pentium III, PowerPC) отличаются специальной оптимизаций операций с вещественными числами, поэтому для этих процессоров использование вещественных чисел может оказаться более предпочтительным.

Следующим шагом будет использование текстур. Для этого назначим каждой стене картинку некоторого фиксированного размера (удобно если этот размер будет являться степенью двух). В следующем примере будем считать, что размер картинки равен 128 на 128 пикселов.

В общем случае задача перспективного проектирования текстуры является нелинейной и требует два деления на каждый пиксел (см. далее).

Рассмотрим сначала простейший частный случай: стена параллельна экрану. Тогда, как несложно заметить, все изображение просто равномерно сжимается/растягивается. Этот случай практически не требует никаких делений - можно, например, использовать алгоритм Брезенхейма.

В общем случае, когда стена не параллельна экрану, любой столбец пикселов экрана (вертикальный отрезок картинной плоскости) соответствует вертикальному отрезку на стене и тем самым параллелен этому отрезку. Отсюда следует, что отображение отрезка пикселов на стене в соответствующий отрезок пикселов на картинной плоскости будет растяжением (сжатием).

Таким образом, для построения текстурированной проекции стены для каждого столбца пикселов на экране следует определить соответствующий столбец пикселов текстуры и смасштабировать его до требуемого размера, для чего необходимо найти расстояние t от точки пересечения до угла клетки.

Так как стены параллельны координатным осям, то в качестве этого расстояния выступает $y \bmod h$ для пересечения с вертикальной стеной и $x \bmod h$ для пересечения с горизонтальной стеной. Обратите внимание на рис. 13.7. Для наблюдателя, находящегося в точке A, самый первый (левый) луч попадает в угол клетки и для него $t = 0$. Для наблюдателя в точке B самый первый луч даст значение $t = h$. Тем самым наблюдатель увидит изображение, перевернутое слева направо. А чтобы оба наблюдателя увидели одно и то же изображение, можно в зависимости от величины угла наблюдения "перевернуть" t, т. е. заменить его на $h - t$.

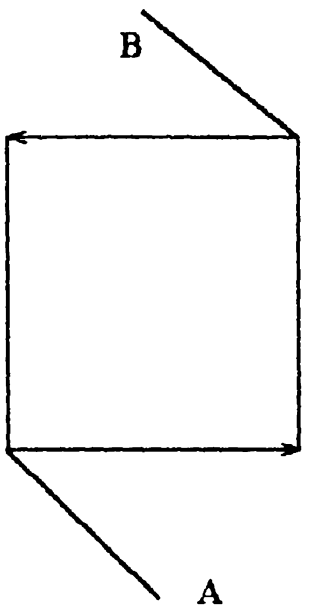


Рис. 13.7

В случае, когда имело место пересечение с горизонтальной стеной, параметр t переворачивается, если ордината точки пересечения меньше ординаты наблюдателя. для вертикальной стены - если абсцисса точки пересечения меньше абсциссы наблюдателя.

Индекс столбца текстуры определяется по формуле

$$j = \frac{(t\%h) * picWidth}{h}$$

В данном случае достаточно просто произвести побитовый сдвиг; к примеру, если t задано в представлении 16.16, $h = 1$, а $picWidth = 64$, то

j = (int)(t >> 10).

Для осуществления сжатия можно воспользоваться либо алгоритмом Брезенхейма, либо дробными числами. Последний подход предпочтительнее, так как в большинстве случаев коэффициент сжатия больше единицы и алгоритм Брезенхейма приводит к многократным записям в один и тот же пиксел экрана.

Несложно видеть, что подобный подход дает действительно правильное перспективное проектирование текстуры, причем ценой совсем небольших затрат - все необходимые параметры для произведения сжатия можно взять из таблиц.

Модифицированная соответствующим образом программа приведена ниже.

```
// File wolf4.cpp
#include    <bios.h>
#include    <dos.h>
#include    <math.h>
#include    <stdio.h>
#include    <time.h>
#include    "fixmath.h"
#include    "bmp.h"
#define   VIEW_WIDTH        (M_PI / 3)    // viewing angle ( 60 degrees )
#define   SCREEN_WIDTH      320           // size of rendering window
#define   SCREEN_HEIGHT     200           // basic colors
#define   FLOOR_COLOR       140
#define   CEILING_COLOR     3             // bios key defintions
#define   ESC               0x011b
#define   UP                0x4800
#define   DOWN              0x5000
#define   LEFT              0x4b00
#define   RIGHT             0x4d00          // labyrinth
char  * worldMap [] =
{
    "**********************************************************",
    "*                 *                    *",
    "*  * * * * * * * *  ************************* *",
    "*                 *                    *",
    "*  * * * * * * * *  * ********************** *",
    "*                                  * *",
    "*  * * * * * * * *  ************************* *",
    "*                 *                    *",
    "**********************************************************"
};
float           swing;
Fixed           locX;
Fixed           locY;
Angle           angle;
```

```
Fixed           xIntV;                                  // interception point for
Fixed           yIntV;                                  // vertical walls
Fixed           xIntH;                                  // interception point for
Fixed           yIntH;                                  // horizontal walls
long            totalFrames  = 0l;
char far *      screenPtr    = (char far *) MK_FP ( 0xA000, 0 );
BMPImage *      pic          = new BMPImage ( "WALL.BMP" );
Angle           rayAngle [SCREEN_WIDTH];   // angles for rays from main
                                           // viewing direction

//////////////////////// Functions ////////////////////////////////

void    drawSpan (.Fixed dist, int textOffs, int x )
{
    char far * vptr = screenPtr + x;
    int    h   = dist >= (ONE / 2) ?
           fixed2Int ((int2Fixed (SCREEN_HEIGHT/2) << 8) / (dist >> 8)) :
           SCREEN_HEIGHT;
    int        j1     = ( SCREEN_HEIGHT - h ) / 2;
    int        j2     = j1 + h;
    char   *   img    = pic -> data + textOffs;
    Fixed      y      = 0l;
    Fixed      dy     = (2 * 128 * dist) / SCREEN_HEIGHT;
                                                  // draw ceiling
    for ( register int j = 0; j < j1; j++, vptr += 320 )
        * vptr = CEILING_COLOR;
    if ( j2 > SCREEN_HEIGHT )
        j2 = SCREEN_HEIGHT;
                                  // skip invisible part of wall
    for ( j = j1; j < 0; j++, y += dy )
        ;
                                  // draw wall
                                  // Note: a << 7 == a * 128
                                  // where pic -> width == 128
    for ( ; j < j2; j++, vptr += 320, y += dy )
        * vptr = img [fixed2Int(y)<<7];
                                  // draw floor
    for ( ; j < SCREEN_HEIGHT; j++, vptr += 320 )
        * vptr = FLOOR_COLOR;
}

Fixed   checkVWalls ( Angle angle )
{
    int        xTile = fixed2Int ( locX );     // cell indices
    int        yTile = fixed2Int ( locY );
    Fixed      xIntercept;                     // intercept point
    Fixed      yIntercept;
    Fixed      xStep;                          // intercept steps
    Fixed      yStep;
    int        dxTile;

    if ( fixAbs ( coTang ( angle ) ) < 2 )
        return MAX_FIXED;
```

```
if ( angle >= ANGLE_270 || angle <= ANGLE_90 )
{
    xTile ++;
    xStep       = ONE;
    yStep       = tang     (angle);
    xIntercept  = int2Fixed (xTile);
    dxTile      = 1;
}
else
{
    xTile --;
    xStep       = -ONE;
    yStep       = -tang     (angle);
    xIntercept  =  int2Fixed (xTile + 1);
    dxTile      = -1;
}
                                        // find interception point
yIntercept = locY + ((xIntercept - locX) << 15) / (coTang (angle) >> 1);

for ( ; ; )
{
    yTile = fixed2Int (yIntercept);
    if ( xTile < 0 || xTile > 52 || yTile < 0 || yTile > 9 )
        return MAX_FIXED;

    if ( worldMap [yTile][xTile] != ' ' )
    {
        yIntV = yIntercept;  // store interception point for
        xIntV = xIntercept;  // computing texture offset

        return ((xIntercept - locX) >> 8) * (invCosine ( angle ) >> 8);
    }

    xIntercept += xStep;
    yIntercept += yStep;
    xTile      += dxTile;
  }
}

Fixed   checkHWalls ( Angle angle )
{
    int     xTile = fixed2Int ( locX );
    int     yTile = fixed2Int ( locY );
    Fixed   xIntercept;
    Fixed   yIntercept;
    Fixed   xStep;
    Fixed   yStep;
    int     dyTile;

    if ( fixAbs ( tang ( angle ) ) < 2 )
        return MAX_FIXED;

    if ( angle <= ANGLE_180 )
    {
        yTile ++;
        xStep       = coTang (angle);
```

```
        yStep       = ONE;
        yIntercept   = int2Fixed (yTile);
        dyTile      = 1;
    }
    else
    {
        yTile --;
        yStep       = -ONE;
        xStep       = -coTang ( angle );
        yIntercept  = int2Fixed (yTile + 1);
        dyTile      = -1;
    }
    xIntercept = locX + ((yIntercept - locY) << 15) / (tang (angle) >> 1);

    for ( ; ; )
    {
        xTile = fixed2Int (xIntercept);
        if ( xTile < 0 || xTile > 52 || yTile < 0 || yTile > 9 )
            return MAX_FIXED;
        if ( worldMap [yTile][xTile] != ' ' )
        {
            xIntH = xIntercept;         // store interception point
            yIntH = yIntercept;         // for computing texture offset

            return ((yIntercept - locY) >> 8) * (invSine ( angle ) >> 8);
        }
        xIntercept += xStep;
        yIntercept += yStep;
        yTile      += dyTile;
    }
}
void    drawView ()
{
    Angle   phi;
    Fixed   distance;

    totalFrames++;
    for ( int col = 0; col < 320; col++ )
    {
        phi = angle + rayAngle [col];

        Fixed   d1 = checkVWalls ( phi );
        Fixed   d2 = checkHWalls ( phi );
        int     textureOffset;

        distance = d1;

        if ( d2 < distance )
        {
            distance     = d2;
            textureOffset = fixed2Int ( frac ( xIntH ) << 7 );

            if ( yIntH < locY )
                textureOffset = 127 - textureOffset;
        }
```

```
            else
            {
                textureOffset = fixed2Int ( frac ( yIntV ) << 7 );
                if ( xIntV < locX )
                    textureOffset = 127 - textureOffset;
            }
                                // adjustment for fish-eye
            distance = (distance >> 8) * (cosine ( phi - angle ) >> 8);
            drawSpan ( distance, textureOffset, col );
        }
    }
    void    setVideoMode ( int mode )
    {
        asm {
            mov     ax, mode
            int  10h
        }
    }
    void    setPalette ( RGB * palette )
    {
        for ( int i = 0; i < 256; i++ )        // convert from 8-bit to 6-bit values
        {
            palette [i].red   >>= 2;
            palette [i].green >>= 2;
            palette [i].blue  >>= 2;
        }
        asm {                                  // really load palette via BIOS
            push    es
            mov     ax, 1012h
            mov     bx, 0                      // first color to set
            mov     cx, 256                    // # of colors
            les     dx, palette                // ES:DX == table of color values
            int     10h
            pop     es
        }
    }
    void    initTables ()
    {
        initFixMath ();
        for ( int i = 0; i < SCREEN_WIDTH; i++ )
            rayAngle [i] = rad2Angle ( atan ( -swing + 2 * i * swing /
                                              (SCREEN_WIDTH - 1) ) );
    }
    main ()
    {
        int     done = 0;
        Fixed   ct   = float2Fixed ( 0.3 );
        Angle   da   = 5 * (ANGLE_90 / 90);
```

```
angle   = 0;
locX    = float2Fixed ( 1.5 );
locY    = float2Fixed ( 1.5 );
swing   = tan ( VIEW_WIDTH / 2 );

initTables   ();
setVideoMode ( 0x13 );
setPalette   ( pic -> palette );

int  start = clock ();

while ( !done )
{
     drawView ();

     if ( bioskey ( 1 ) )
     {
          Fixed    vx = cosine ( angle );
          Fixed    vy = sine   ( angle );
          Fixed    x, y;

          switch ( bioskey ( 0 ) )
          {
               case LEFT:
                    angle -= da;
                    break;

               case RIGHT:
                    angle += da;
                    break;

          case UP:
               x = locX + (ct >> 8) * (vx >> 8);
               y = locY + (ct >> 8) * (vy >> 8);

               if ( worldMap [fixed2Int (y)][fixed2Int (x)] == ' ' )
               {
                    locX = x;
                    locY = y;
               }

               break;

          case DOWN:
               x = locX - (ct >> 8) * (vx >> 8);
               y = locY - (ct >> 8) * (vy >> 8);

               if ( worldMap [fixed2Int (y)][fixed2Int (x)] == ' ' )
               {
                    locX = x;
                    locY = y;
               }

               break;

          case ESC:
               done = 1;
     }
}
```

```
    }
    float     totalTime = ( clock () - start ) / CLK_TCK;
    setVideoMode ( 0x03 );
    printf ( "\nFrames rendered    : %ld",   totalFrames );
    printf ( "\nTotal time ( sec )   %7.2f", totalTime );
    printf ( "\nFPS                  %7.2f", totalFrames / totalTime );
}
```

Для повышения качества анимации желательно воспользоваться каким-либо многостраничным *X*-режимом адаптера VGA.

К сожалению, столь элегантный и эффективный метод, непосредственно для текстурирования горизонтальных поверхностей (пола и потолка) неприменим. Правда, существует его модификация, но о ней будет рассказано позже.

Для представления различных объектов в лабиринте в Wolfenstein 3-D используются спрайты, в частности даже светлое пятно под лампами делается как спрайт.

Обычный плоский спрайт представляет собой набор картинок, соответствующих разным фазам движения или действия объекта. Для придания объекту трехмерности воспользуемся следующим приемом: для каждой из фаз зададим вид на объект с заданного количества сторон, например с восьми сторон.

Тогда такому объекту приписывается определенный угол - направление, куда смотрит (направлен) соответствующий объект.

Картинка выбирается с того направления, которое определяется разностью углов между направлением взгляда объекта и взгляда игрока на объект (рис. 13.8).

Как нетрудно убедиться,

$$\alpha = \pi + \varphi - \psi$$

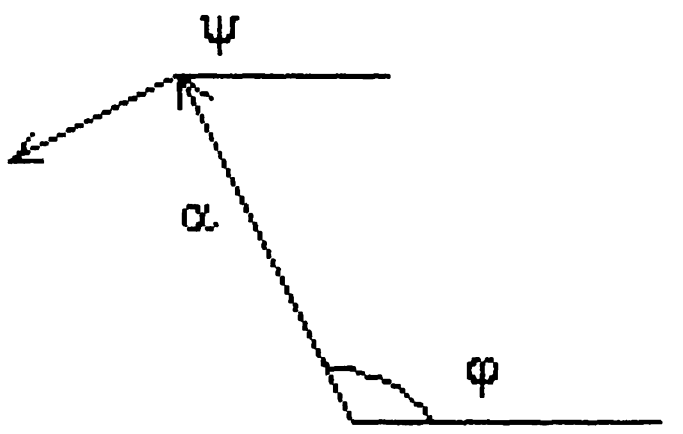


Рис. 13.8

Путем несложных модификаций программы реально добавить в нее возможность работы со спрайтами. Для этого при трассировании лучей пометим все видимые клетки и для каждого луча запомним расстояние до ближайшей стены, затем составим список всех спрайтов, находящихся в помеченных клетках (можно клетки и не помечать, а использовать полный список спрайтов), отсортируем его по расстоянию до игрока и выведем спрайты в порядке близости к игроку (back-to-front). Каждый спрайт масштабируется и выводится по столбцам, при этом для каждого столбца перед выводом необходимо сравнить расстояние до спрайта с расстоянием до ближайшей стены - своего рода одномерный аналог *z*-буфера.

Двери и секреты можно рассматривать как специальный тип клетки. При этом результат проверки клетки на занятость (может ли сквозь нее пройти луч) зависит от точки пересечения луча с границей клетки и времени, что может потребовать изменения процедуры вычисления расстояния (рис. 13.9).

В принципе метод ray casting применим и для создания заметно более сложных игр (например, DOOM). Ниже мы рассмотрим некоторые модификации этого алгоритма для работы со стенами произвольной ориентации.

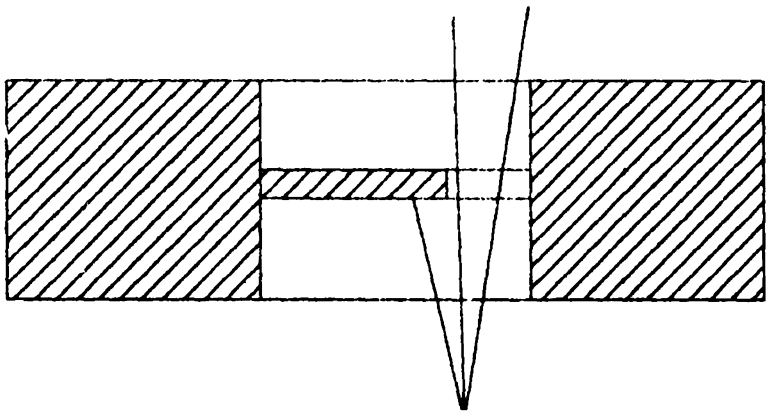


Рис. 13.9

Пусть на плоскости задан набор отрезков, определяющих вертикальные стены в трехмерном пространстве. На ориентацию этих стен мы не будем накладывать каких-либо ограничений - оставаясь вертикальными, они могут быть расположены под произвольными углами к оси Ox

Самым простым вариантом применения ray casting для такой сцены по-прежнему будет следующий: для каждого столбца экрана выпускается по лучу и находится его ближайшее пересечение со стенами лабиринта. Однако проверка всех стен на пересечение с лучом неприемлема ввиду того, что проверка на пересечение с произвольно ориентированной стеной достаточно сложна и стен может быть много.

Для оптимизации такого подхода можно воспользоваться ранее применявшимся разбиением плоскости на равные части (клетки), при котором каждой клетке ставится в соответствие список всех пересекающих ее стен. Тогда, выпустив луч, мы отслеживаем его переход из клетки в клетку и для каждой очередной клетки проверяем связанные с ней стены на пересечение с лучом. Ближайшее пересечение в пределах данной клетки (в одной клетке может быть найдено несколько пересечений; кроме того, пересечение на самом деле может принадлежать другой клетке) и становится искомым.

Рассмотрим, каким образом осуществляется проверка на пересечение луча, выпущенного из точки (x^*, y^*) под углом φ, с отрезком, соединяющим точки (x_1, y_1) и (x_2, y_2).

Прямая, проходящая через точку (x^*, y^*) под углом φ, задается следующим уравнением:

$$(x - x^*)\sin\varphi - (y - y^*)\cos\varphi = 0$$

Его можно переписать в виде $ax + by - c = 0$

Прямая разбивает всю плоскость на две полуплоскости и может пересекать отрезок только в том случае, если его концы находятся в разных полуплоскостях. Для определения, в какой полуплоскости находится данная точка, введем функцию

$$F(x, y) = ax + by - c;$$

Одной полуплоскости соответствуют точки, где $F(x, y) > 0$, а другой - где $F(x, y) < 0$.

Условие пересечения отрезка прямой имеет вид $F(x_1, y_1)F(x_2, y_2) \le 0$.

Если пересечение имеет место, то точка пересечения ищется в виде

$$x = tx_1 + (1 - t)x_2,$$

$$y = ty_1 + (1 - t)y_2,$$

где $t = \dfrac{F(x_2, y_2)}{F(x_2, y_2) - F(x_1, y_1)}$

Это пересечение прямой и луча. Чтобы получить пересечение отрезка с лучом, достаточно сравнить координаты точки пересечения с концами отрезка.

Вследствие того, что проверка на пересечение луча с произвольным отрезком **существенно сложнее, чем проверка на пересечение с горизонтальным или вертикальным отрезком, быстродействие данного алгоритма не так высоко.**

Для повышения быстродействия можно попытаться использовать когерентность между соседними столбцами пикселов - если данный столбец соответствует некоторой стене, то скорее всего несколько последующих столбцов также будут соответствовать этой же стене. **Изменим процедуру проверки луча на пересечение с заданной стеной - сразу найдем, каким столбцам** $c1$ **и** $c2$ **соответствуют концы отрезка, и** запомним их, и, если потребуется проверить на пересечение луч с данной стеной, достаточно просто проверить, принадлежит ли номер луча (столбца) диапазону $[c1,c2]$. **Удобно сразу же рассчитать высоты для проекций соответствующих столбцов этой** стены (они изменяются линейно).

Тем самым мы приходим к следующей процедуре проверки.

```
float     checkWall ( int wallno )
{
    Wall     * wptr = &walls [wallno];
    if ( !wptr -> flags )          // check if data not computed
{
  float  u1 = -(wptr->x1-locX)*sin(angle)+(wptr->y1-locY)*cos(angle);
  float  v1 = (wptr->x1-locX)*cos(angle)+(wptr->y1-locY)*sin(angle);
  float  u2 = -(wptr->x2-locX)*sin(angle)+(wptr->y2-locY)*cos(angle);
  float  v2 = (wptr->x2-locX)*cos(angle)+(wptr->y2-locY)*sin(angle);
                    // check if wall is behind
  if ( v1 < V_MIN && v2 < V_MIN )
  {
    wptr -> c1 = w -> ptr -> c2 = 10000;
    wptr -> flags = 1;
    return -1;
  }
  if ( v1 < V_MIN )
  {
    u1 += ( V_MIN - v1 )*( u2 - u1 )/( v2 - v1 );
    v1  = V_MIN;
  }
  if ( v2 < V_MIN )
  {
    u2 += ( V_MIN - v2 )*(u2 - u1 )/( v2 - v1 );
    v2  = V_MIN;
  }

  wptr -> flags = 1;
  wptr -> c1    = (int)( 160 + (u1*HSCALE)/v1 );
  wptr -> c2    = (int)( 160 + (u2*HSCALE)/v2 );
```

```
        float  h1    = VSCALE / v1;
        float  h2    = VSCALE / v2;
        float  dh    = ( h2 - h1 ) / ( c2 - c1 );

        for ( int i = wptr->c1;i <= wptr->c2; i++, h1 += dh )
            wptr -> h [i - wptr -> c1] = h1;
    }

    if ( col < wptr -> c1 || col > wptr -> c2 )
        return -1;

    return wptr -> h [col - wptr -> c1];
}
```

Количество операций умножения и деления на всю стену составляет 15, так что если средняя ширина проекции стены больше 3-4 пикселов, то этот подход оказывается заметно выгоднее. Обратите внимание на использование когерентности для вычисления высот соответствующих столбцов пикселов для соседних лучей.

Замечание. *Количество операций умножения для поворота пиксела (что используется для проектирования вершин отрезка) можно сократить вдвое. Для этого для каждой вершины заранее вычисляется произведение ее координат xy, а также для угла поворота вычисляется величина*

$$\sin\varphi\cos\varphi = \frac{1}{2}\sin 2\varphi$$

После этого используются соотношения

$$y\sin\varphi - x\cos\varphi = (x+\sin\varphi)(y-\cos\varphi) - xy + \sin\varphi\cos\varphi,$$
$$x\sin\varphi + y\cos\varphi = (x+\cos\varphi)(y+\sin\varphi) - xy - \sin\varphi\cos\varphi.$$

13.2. Текстурирование горизонтальных поверхностей

Попытаемся изменить подход, столь успешно примененный при текстурировании вертикальных поверхностей (и заключающийся в рисовании их вертикальными отрезками), для работы с горизонтальными поверхностями.

Для этого рассмотрим произвольный горизонтальный отрезок на экране, соответствующий некоторой горизонтальной поверхности. Его прообразом при проектировании является отрезок, лежащий на прямой, получающейся при пересечении поверхности с плоскостью, проходящей через наблюдателя и отрезок экрана. Как несложно заметить, эти два отрезка будут параллельны друг другу и, следовательно, их отображение будет масштабированием с коэффициентом, постоянным вдоль всего отрезка.

Тем самым горизонтальные поверхности следует рисовать на экране горизонтальными линиями на экране.

Рассмотрим упрощенную задачу: наблюдатель находится на высоте h над плоскостью с нанесенной на нее текстурой, и больше ничего на плоскости нет. Пусть направление взгляда наблюдателя составляет угол φ с осью параметра u (рис. 13.10).

Прообразом произвольной строки экрана на плоскости будет отрезок прямой, чем угол этой прямой с направлением оси *u* будет $\varphi + \pi/2$.

Найдем расстояние вдоль плоскости *Oxy* до этой прямой и ближайшую точку на

Имеем: $d = \dfrac{hd_0}{h - k\delta} = hctg\alpha_k$,

где k - номер строки от конца экрана, а δ - шаг по вертикали между строками экрана.

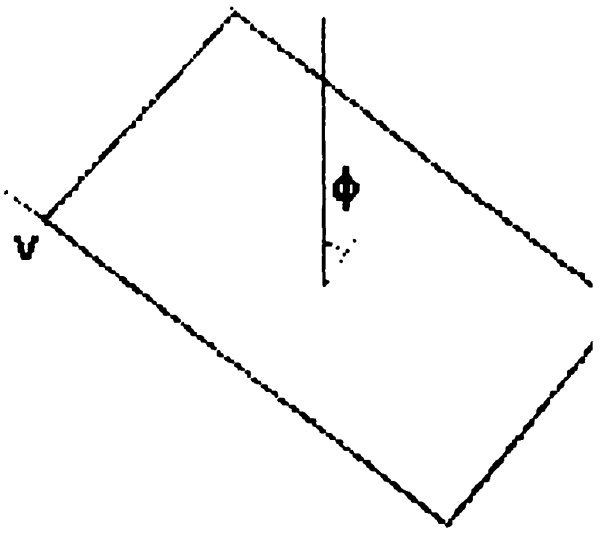


Рис. 13.10

Ближайшая точка находится из соотношений

$x = x^* + d \cos \varphi,$

$y = y^* + d \sin \varphi.$

Коэффициент сжатия пропорционален расстоянию, т. е. равен Cd, где C - некоторый масштабирующий коэффициент.

Для построения строки необходимо найти индекс текстуры (u, v) для какой-либо точки данной строки и шаги для обоих индексов при переходе к соседнему пикселу строки.

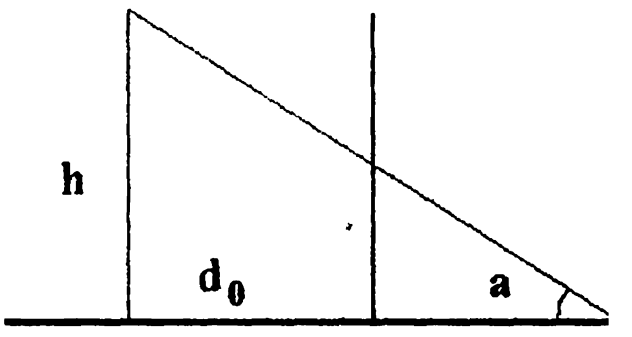


Рис. 13.11

Индекс легко находится из формулы

u = [x * picWidth] % picWidth

v = [y * picHeight] % picHeight

Шаги вдоль индексов текстуры определяются из следующих соотношений

$\Delta u = Cd \sin \varphi,$

$\Delta v = -Cd \cos \varphi.$

Ниже приводится вещественная версия программы, осуществляющая текс рование горизонтальной плоскости.

```
// File Floor.cpp
#include    <bios.h>
#include    <dos.h>
#include    <math.h>
#include    <mem.h>
#include    <stdio.h>
#include    <time.h>
#include    "bmp.h"

#define SKY_COLOR   3

#define   ESC       0x011b
#define   UP        0x4800
#define   DOWN      0x5000
#define   LEFT      0x4b00
#define   RIGHT     0x4d00

#define   H         100.0
#define   DELTA     1.0             // note   DELTA * NumLines == H
#define   C         0.01
#define   D0        100.0
```

```
long            totalFrames   = 0l;
BMPImage *      pic           = new BMPImage ( "FLOOR.BMP" );
char far *      screenPtr     = (char far *) MK_FP ( 0xA000, 0 );

float             x, y;                 // viewer loc
float             angle;

int  mod ( float x, int y )
{
    int  res = (int) fmod ( x, y );

    if ( res < 0 )
        res += y;

    return res;
}

void     drawView ()
{
    char     far * videoPtr = screenPtr + 100*320;

    totalFrames++;

    _fmemset ( screenPtr, SKY_COLOR, 100*320 );     // draw the sky

    for ( int row = 0; row < 100; row++ )
    {
        float  dist = H * D0 / ( ( 1 + row ) * DELTA );
        int    i0   = mod ( x + dist * cos ( angle ), pic -> width );
        int    j0   = mod ( y + dist * sin ( angle ), pic -> height );
        float  di   = C * dist * sin ( angle );
        float  dj   = -C * dist * cos ( angle );
        float  ii   = i0;
        float  jj   = j0;
        int    i, j;

        videoPtr += 160;

        for ( int col = 159; col >= 0; col-- )
        {
            i = mod ( ii, pic -> width );
            j = mod ( jj, pic -> height );

            * videoPtr-- = pic -> data [ i + j * pic -> width ];

            ii -= di;
            jj -= dj;
        }

        videoPtr += 160;
        ii = i0 + di;
        jj = j0 + dj;

        for ( col = 160; col < 320; col++ )
        {
            i = mod ( ii, pic -> width );
            j = mod ( jj, pic -> height );

            * videoPtr++ = pic -> data [ i + j * pic -> width ];
            ii += di;
```

```
                jj += dj;
            }
        }
    }
    void    setVideoMode ( int mode )
    {
        asm {
            mov     ax, mode
            int  10h
        }
    }
    void    setPalette ( RGB  * palette )
    {
        for ( int i = 0; i < 256; i++ )        // convert from 8-bit to
        {                                      // 6-bit values
            palette [i].red   >>= 2;
            palette [i].green >>= 2;
            palette [i].blue  >>= 2;
        }
        asm {                                  // really load palette via BIOS
            push    es
            mov     ax, 1012h
            mov     bx, 0                      // first color to set
            mov     cx, 256                    // # of colors
            les     dx, palette                // ES:DX == table of color values
            int     10h
            pop     es
        }
    }
    main ()
    {
        int  done  = 0;
        int  start  = clock ();

        angle   = 0;
        x       = 29;
        y       = 0;

        setVideoMode ( 0x13 );
        setPalette   ( pic -> palette );

        while ( !done )
        {
            drawView ();

            if ( bioskey ( 1 ) )
            {
                float     vx = cos ( angle ) * 10;
                float     vy = sin ( angle ) * 10;

                switch ( bioskey ( 0 ) )
                {
                    case LEFT:
```

```
                angle += 10 * M_PI / 180;
                break;
            case RIGHT:
                angle -= 10 * M_PI / 180;
                break;
            case UP:
                x += vx;
                y += vy;
                break;
            case DOWN:
                x -= vx;
                y -= vy;
                break;
            case ESC:
                done = 1;
                break;
        }
    }
}
float    totalTime = ( clock () - start ) / CLK_TCK;
setVideoMode ( 0x03 );
printf ( "\nFrames rendered   : %ld",  totalFrames );
printf ( "\nTotal time ( sec )    %7.2f", totalTime );
printf ( "\nFPS                   %7.2f", totalFrames / totalTime );
}
```

После перехода к числам с фиксированной точкой и ряда оптимизаций процедура *d*raw*V*iew принимает следующий вид:

```
// Phragment of Floor3.cpp
void        drawView ()
{
    char far *   videoPtr      = screenPtr + 100*320;
    long         widthMask     = MAKELONG ( pic -> width - 1, 0xFFFF );
    long         heightMask    = MAKELONG ( pic -> height - 1, 0xFFFF );
    int          widthShift    = getShift ( pic -> width );
    char    *    picData       = pic -> data;

    totalFrames++;

    drawSky ();

    for ( int row = 0; row < 100; row++ )
    {
        Fixed   dist = ( H * distTable [row] ) >> 8;
        Fixed   u0   = ( locX + dist * ( cosine ( angle ) >> 8 ) ) & widthMask;
        Fixed   v0   = ( locY + dist * ( sine ( angle ) >> 8 ) ) & heightMask;
        Fixed   du   = (  dist * CSinTable [angle >> 6] ) >> 8;
        Fixed   dv   = ( -dist * CCosTable [angle >> 6] ) >> 8;
        Fixed   u    = u0;
        Fixed   v    = v0;
```

```
            videoPtr += 160;
            for ( int col = 159; col >= 0; col-- )
            {
                * videoPtr-- = picData [fixed2Int ( u ) +
                                ( fixed2Int ( v ) << widthShift )];
                u = ( u - du ) & widthMask;
                v = ( v - dv ) & heightMask;
            }
            videoPtr += 160;
            for ( col = 160, u = u0, v = v0; col < 320; col++ )
            {
                u = ( u + du ) & widthMask;
                v = ( v + dv ) & heightMask;
                videoPtr++ = picData [ fixed2Int ( u ) +
                ( fixed2Int ( v ) << widthShift ) ];
            }
        }
    }
```

13.3. DOOM

Одной из самых серьезных и известных игр, сделанных в стиле виртуальной реальности, является игра DOOM, которая прочно заняла ведущие места в рейтингах самых популярных игр сразу же после своего выхода.

Игра была революционной в целом ряде аспектов, сочетая в себе сложную геометрию сцены, текстурирование горизонтальных поверхностей и многое другое, продемонстрировав трехмерную графику, невиданную ранее на персональных компьютерах.

Рассмотрим структуру типичного уровня (пример уровня приведен на рис. 13.12).

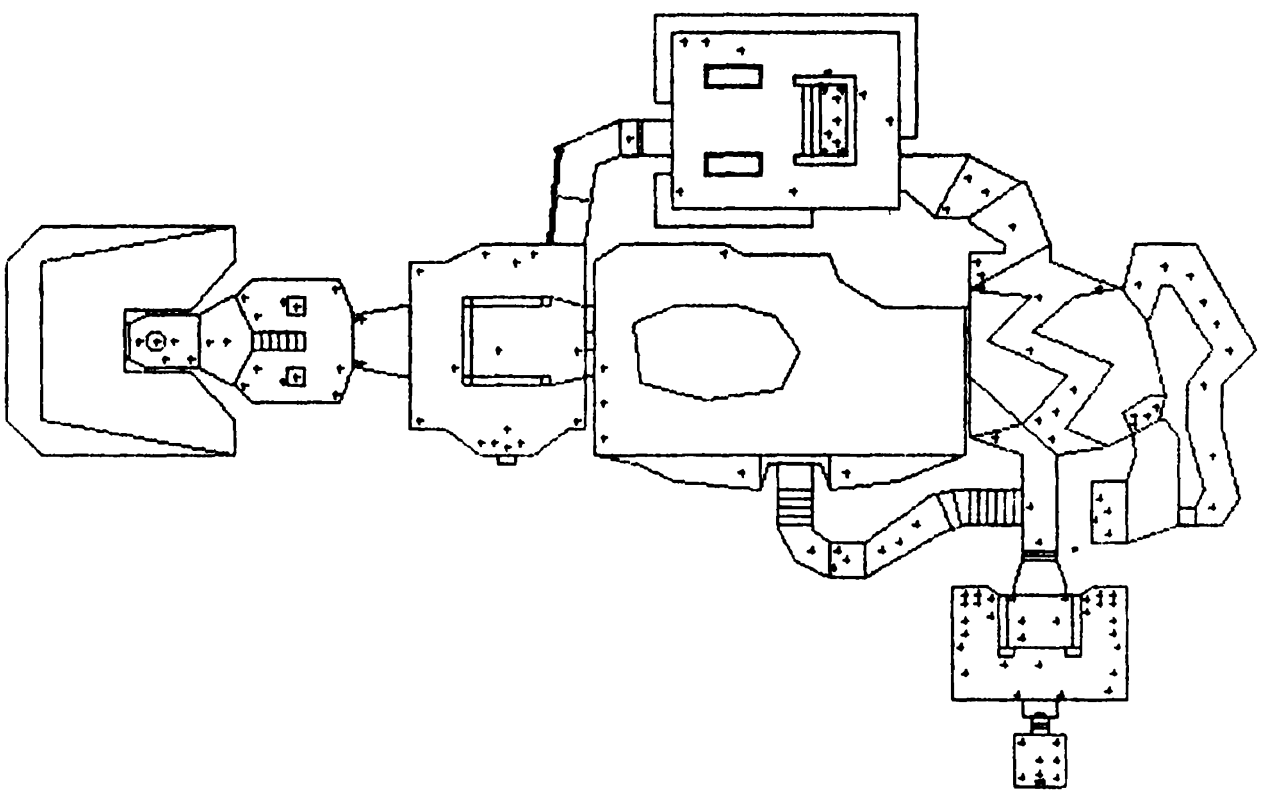


Рис. 13.12

Несмотря на кажущуюся трехмерность, игра эта скорее двумерная, представляя собой набор плоских слоев (такие сцены иногда называют 2,5-мерными). Заметнее

всего это при создании нового уровня - сначала на плоскости Oxy строится карта уровня, состоящая из вершин (Vertex) и соединяющих их отрезков (LineDef). Отрезки разбивают плоскость на ряд областей (Sector), для которых задаются высота пола, потолка и необходимые текстуры. Фактически для любой точки карты лабиринта пересечение сцены вертикальной прямой, проходящей через эту точку, состоит либо из двух точек, либо из отрезка (возможен случай, когда это пересечение состоит из двух отрезков). При этом отрезки возникают только в том случае, когда луч проходит сквозь стену. Тем самым возможность расположения двух комнат одна над другой полностью исключается.

Отрезки направлены (порядок вершин выбран) таким образом, что справа от отрезка всегда находится сектор. Существуют и такие отрезки, для которых секторы находятся с обеих сторон, - так называемые двусторонние отрезки.

Рассмотрим сцену, представленную на рис. 13.13 и состоящую из двух секторов 0-1-4-5 и 1-2-3-4 с разными высотами пола и потолка (или разными текстурами для них). В данном случае отрезок 1-4 является двусторонним, так как служит линией раздела двух секторов. С каждым отрезком связана одна или несколько сторон (SideDef).

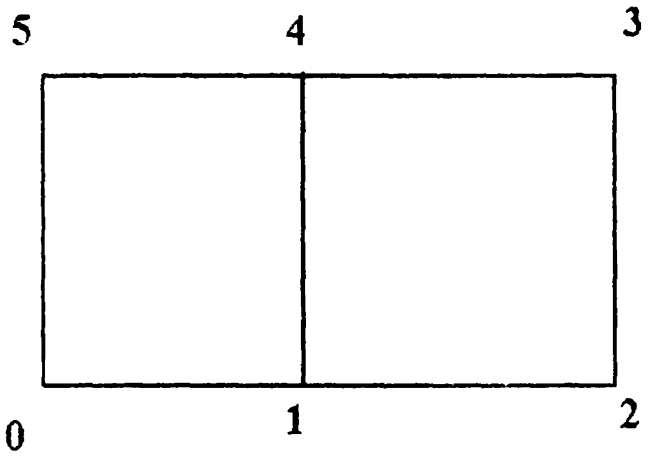


Рис. 13.13

Сторона служит для определения текстуры вдоль стены, и в случае, когда отрезок служит разделителем двух секторов, с ним могут быть связаны две стороны, каждая из которых определяет текстуру, видимую со своей стороны. Выделяется несколько типов текстуры. Простейшая из них - это регулярная (main), служащая для закрашивания нормальной стены. В случае двусторонних отрезков возникают специальные текстуры - верхняя и нижняя.

Нижняя текстура используется для закрашивания фрагментов стен, соединяющих пол одного сектора с полом другого вдоль линии их раздела (например, для ступенек). Верхняя текстура используется для закрашивания фрагмента стены, соединяющей потолок одного сектора с потолком другого (рис. 13.14).

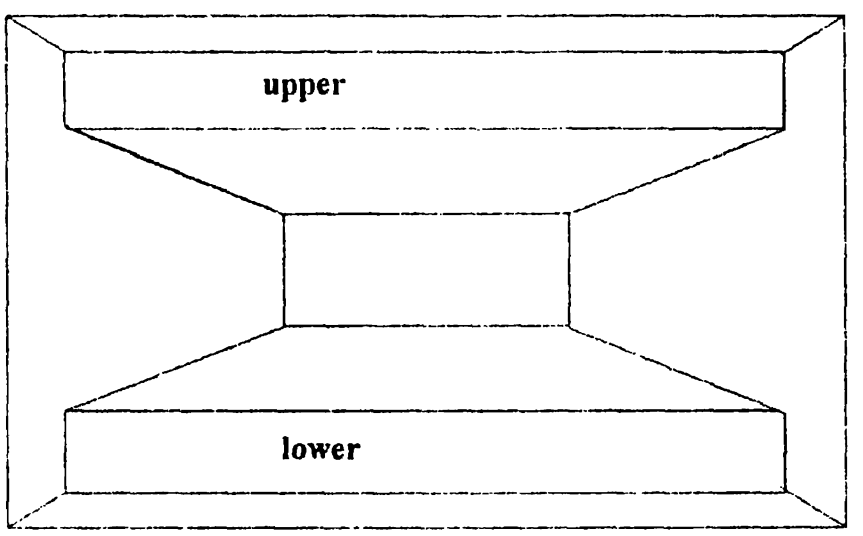


Рис. 13.14

Для работы рендерера все секторы разбиваются на выпуклые части, называемые SubSector.

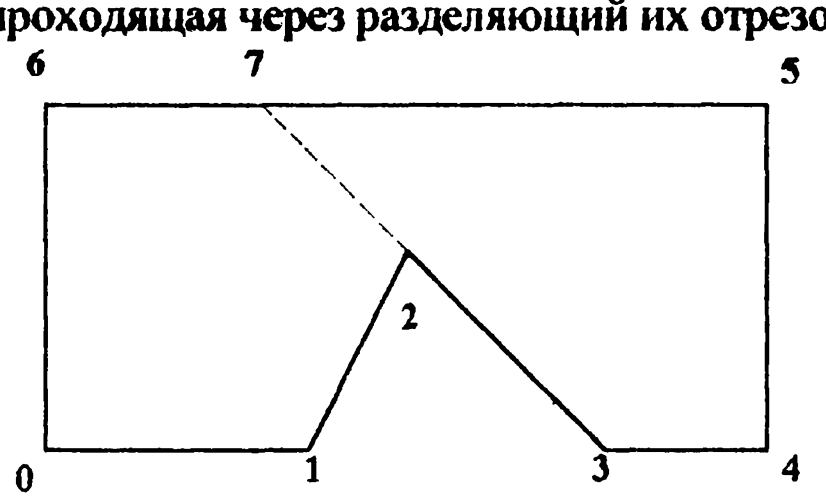
При построении изображения происходит упорядочение всех SubSector'ов по мере их удаления (front-to-back) и последовательный вывод.

Для построения такого разбиения и упорядочения SubSector используется специальный вариант BSP-деревьев.

Замечание. *На самом деле используется еще структура равномерного разбиения плоскости (BLOCKMAP), но она применяется не для рендеринга, а для определения возможности перемещения игрока (проверок на пересечение со стенами)*
В данной структуре весь лабиринт разбивается на одинаковые квадратные *клетки и для каждой такой клетки хранится список всех пересекающих ее отрезков (LineDef).*

Рассмотрим несколько простейших сцен.

Пусть сцена (рис. 13.15) состоит всего из одного сектора, но он невыпуклый. Разобьем его на две части прямой *l*, проходящей через вершины 3 и 2. В результате появится одна дополнительная вершина 7 и сектор будет разбит на две выпуклые части (0-1-2-7-6 и 2-3-4-5-7), они и станут SubSector. По результатам разбиения по**строим двоичное дерево, узлами которого являются разбивающие плоскости, а листьями - SubSector. Получившееся в результате дерево изображено на рис. 13.16. Сцена с рис.** 13.13 состоит уже из двух выпуклых секторов, поэтому узлом (корнем) дерева будет пря**мая, проходящая через разделяющий их отрезок.**



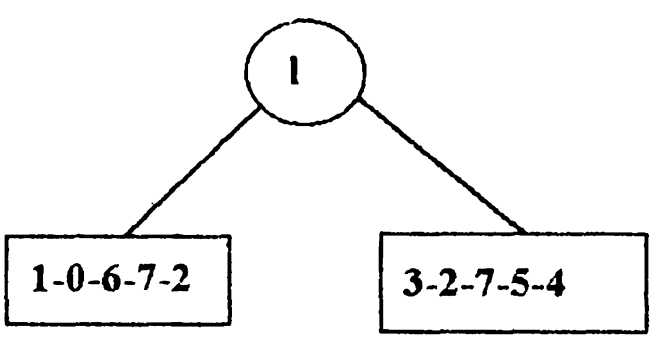
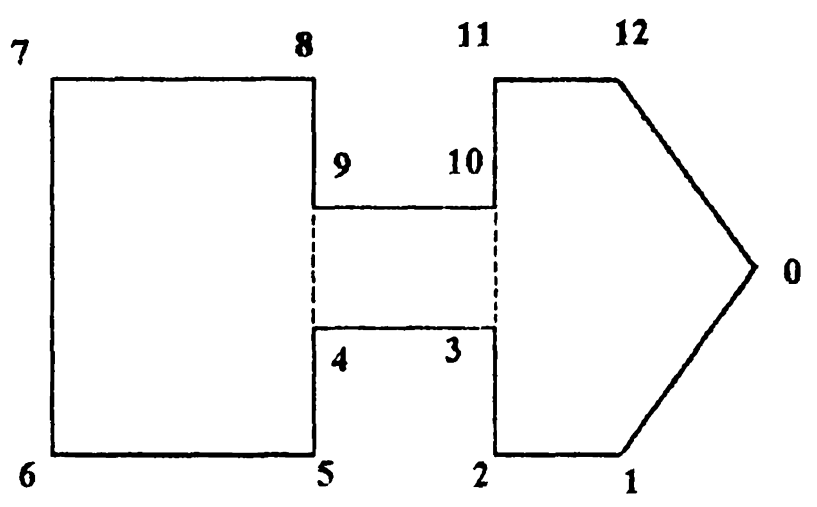


Рис. 13.15 *Рис. 13.16*

Для сцены, приведенной на рис. 13.17, одного разбиения недостаточно, так как одна из частей, которые получились после разбиения прямой, проходящей через отрезок 11-10, по-прежнему является невыпуклой. Разобьем ее еще раз, приходя тем самым к дереву, представленному на рис. 13.18.



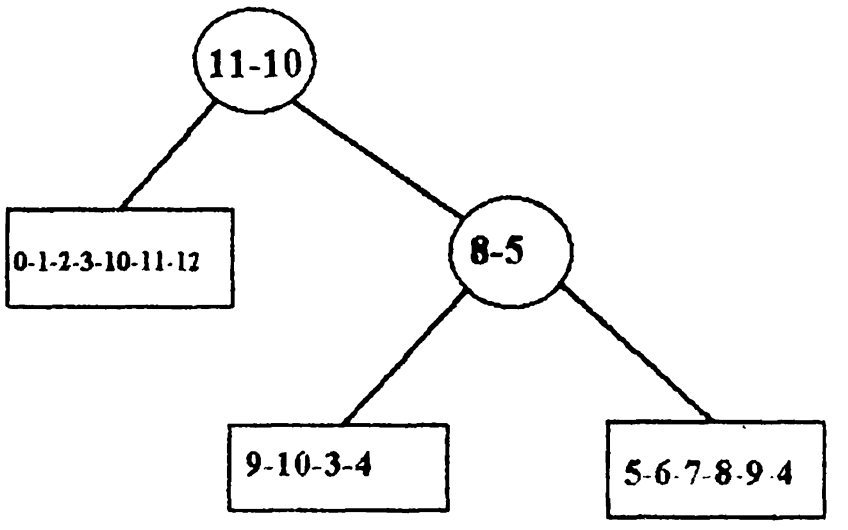


Рис. 13.17 *Рис. 13.18*

В общем случае процедура разбиения выглядит следующим образом: выбирается некоторая прямая (обычно проходящая через один из отрезков) и все секторы, через которые она проходит, разбиваются ею на части. Получаем два множества

многоугольников - лежащих справа от прямой, производящей разбиение, и лежащих слева от этой прямой.

Далее для каждого из получившихся множеств, которые не являются одним выпуклым многоугольником, повторяем описанную процедуру.

В результате мы получаем разбиение всей сцены на набор SubSector и соответствующее двоичное дерево, узлами которого являются прямые, производящие разбиение, а листьями - выпуклые многоугольники - SubSector. Связь этого метода с классическим BSP-деревом несомненна. В получившемся дереве все узлы нумеруются начиная с нуля, так что корень дерева имеет наибольший номер. Листьям дерева присваиваются номера соответствующих SubSector, где (чтобы его можно было отличить от узла) устанавливается старший бит (0x8000).

Границами каждого SubSector в общем случае являются части отрезков, называемые сегментами (Seg). При этом все сегменты упорядочиваются так, чтобы при их последовательном обходе соответствующий SubSector всегда находился справа. Приведем список Seg для сцены с рис. 13.15: [5, 4], [4, 1], [1, 0], [0, 5], [1, 4], [4, 3], [3, 2], 2, 1].

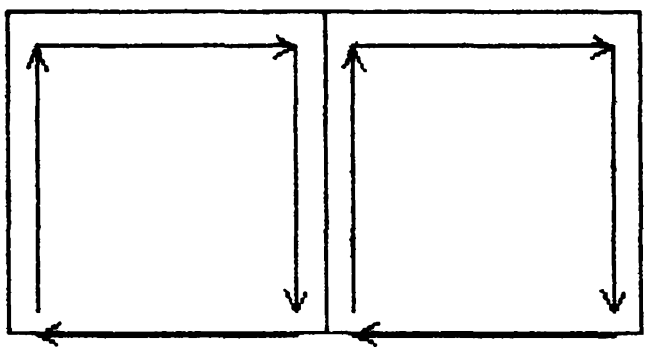


Рис. 13.19

Для сцены с рис. 13.17 возникают следующие сегменты: [7, 5], [5, 4], [4, 3], [3, 2], [6, 7], [2, 1], [1, 0], [0, 6]. Обратите внимание на отсутствие сегмента между точками 2 и 7.

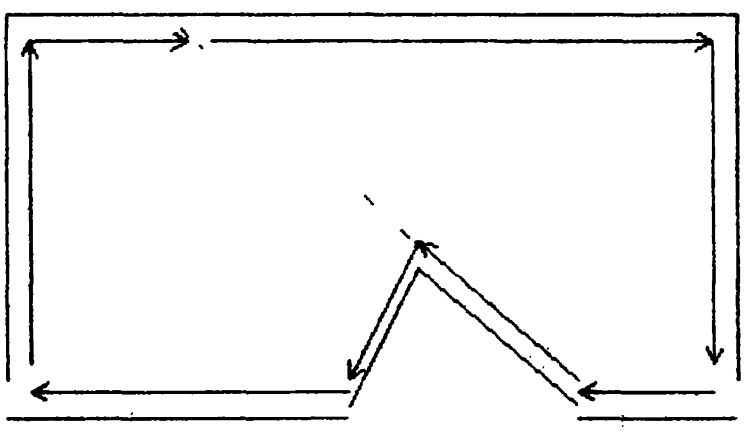


Рис. 13.20

Для упорядочения всех SubSector используется рекурсивная процедура, полностью аналогичная процедуре вывода BSP-деревьев.

```
void    drawNode ( unsigned node )
{
    if ( node & 0x8000 )      // if it's a ssector - draw it
        drawSubSector ( node & 0x7FFF );
    else
    if ( viewerOnRight ( node ) )
    {
        drawNode ( Nodes [node].rightNode );
        drawNode ( Nodes [node].leftNode );
    }
    else
    {
        drawNode ( Nodes [node].leftNode );
        drawNode ( Nodes [node].rightNode );
    }
}
```

Данная процедура обеспечивает вывод всех SubSector в порядке front-to-back, причем первым будет выведен тот SubSector, где находится игрок. Для вывода всей сцены достаточно вызвать drawNode с номером корневого узла дерева.

Поскольку каждый SubSector является выпуклым многоугольником, то никакого специального упорядочения ограничивающих его отрезков не требуется, достаточно только отбросить нелицевые отрезки.

Таким образом, вывод SubSector'а состоит в выводе всех ограничивающих его лицевых сегментов и, при необходимости, фрагментов пола и потолка.

```
void     drawSubSector ( int s )
{
    int             firstSeg = subSectors [s].firstSeg;
    int             numSegs    = subSectors [s].numSegs;
    Segment *  seg = &Segs [firstSeg];

    if ( curSector == NULL )// 1st ssector contains viewer
    {
        SideDef * side = &sides[lines[seg->lineDef].sideDefs[seg->lineSide]];
                    // find sector with viewer
        curSector = &subSectors [s];
        locZ = 40 + sectors [side -> sector].floorHeight;
    }

    for ( int i = 0; i < numSegs; i++, seg++ )
        if ( frontFacing ( seg -> from, seg -> to ) )
            drawSeg ( seg );
}
```

При выводе SubSector (сегментов) в порядке удаления от наблюдателя необходим механизм для отслеживания тех частей экрана, которые уже были заполнены. В силу структуры сцены наиболее подходящим для этой цели является метод плавающего горизонта. Линии горизонта (два массива размером в ширину экрана) topLine и bottomLine вводятся так, чтобы была незаполненной только область между этими линиями. При выводе стены (пола, потолка) выводится только та часть, которая находится между линиями горизонта, а сами эти линии соответствующим образом корректируются.

Если для какого-либо столбца c имеет место неравенство

topLine [c] > bottomLine [c],

то это означает, что данный столбец уже полностью заполнен и его можно пропустить.

Наиболее простым является вывод односторонней стены - выводится только та ее часть, которая находится между линиями горизонта. Так как эта стена проходит от пола до потолка без всяких отверстий, то она полностью закрывает собой все, что расположено за ней. Таким образом, для всех столбцов *c*, в которые она попадает, после ее вывода можно установить

topLine [c] > bottomLine [c]

Стена рисуется по столбцам; для облегчения вычислений используется тот факт, что высоты столбцов каждой стены изменяются линейно. Сначала производится вычисление размеров очередного столбца [top,bottom], затем этот столбец отсекается по линиям горизонта, т. е. по отрезку [topLine[col], bottomLine[col]].

Если верхний конец отрезка стены top оказывается лежащим ниже соответствующей линии горизонта, то на этом месте должен находиться фрагмент потолка.

Аналогично если нижний конец отрезка стены bottom находится выше нижней линии горизонта, то между ними должен находиться фрагмент пола.

Ниже приведена процедура для построения простой (односторонней) стены.

```
void    drawSimpleWall ( int col1, int col2, int bottomHeight,
                         int topHeight, int v1, int v2 )
{
    Fixed   b1 = int2Fixed ( 100l + ( bottomHeight * VSCALE ) / v1 );
    Fixed   b2 = int2Fixed ( 100l + ( bottomHeight * VSCALE ) / v2 );
    Fixed   t1 = int2Fixed ( 100l - ( topHeight * VSCALE ) / v1 );
    Fixed   t2 = int2Fixed ( 100l - ( topHeight * VSCALE ) / v2 );
    Fixed   db = ( b2 - b1 ) / ( col2 - col1 );
    Fixed   dt = ( t2 - t1 ) / ( col2 - col1 );
    int     top, bottom;

    if ( col1 < 0 )          // clip invisible part
    {
        b1  -= col1 * db;
        t1  -= col1 * dt;
        col1 = 0;
    }

    if ( col2 > 320 )
        col2 = 320;

for ( int col = col1; col < col2; col++ )
{
    top         = fixed2Int ( t1 );
    bottom      = fixed2Int ( b1 );
    t1          += dt;
    b1          += db;

    if ( topLine [col] > bottomLine [col] ) // nothing to draw here
        continue;

    if ( top > bottomLine [col] )
        top = bottomLine [col];

    if ( bottom < topLine [col] )
        bottom = topLine [col];

    if ( top >= topLine [col] )      // draw ceiling
    {
        drawVertLine (col, topLine [col], top,
                      CEILING_COLOR);
        topLine [col] = ++top;
    }
    else                             // otherwise correct to
        top = topLine [col];         // draw only visible part

    if ( bottom <= bottomLine [col] ) // draw floor
    {
        drawVertLine ( col, bottom, bottomLine [col], FLOOR_COLOR );
        bottomLine [col] = --bottom;
```

```
            }
            else
                bottom = bottomLine [col];

            topLine [col]    = bottom + 1;
            bottomLine [col] = top - 1;
                                    // now draw visible part of wall
            drawVertLine ( col, top, bottom, WALL_COLOR );

        }
    }
```

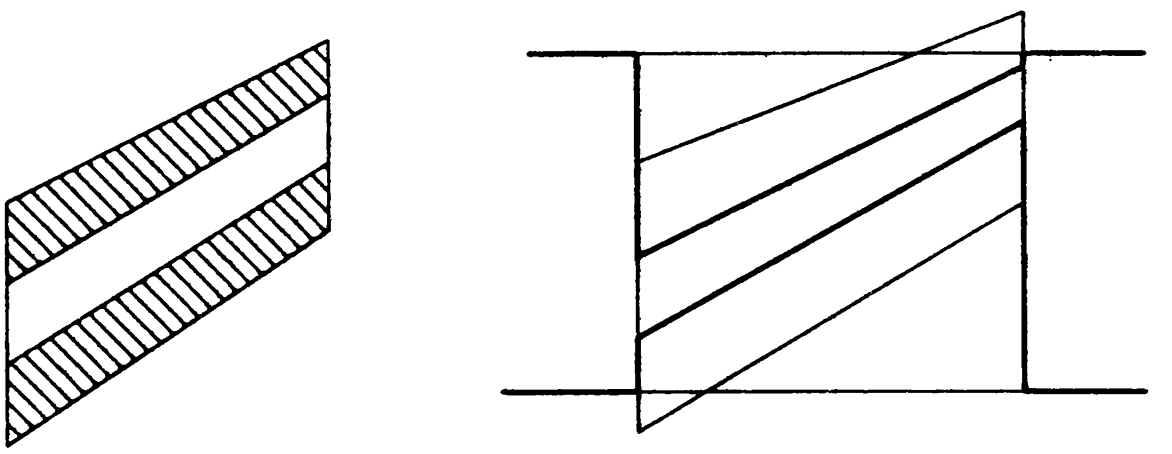


Рис. 13.21

Жирными линиями выделены (рис. 13.21) линии горизонта после вывода грани.

Для написания простейшего рендерера, способного работать с WAD-файл потребуется класс для чтения данных из него, а также описание основных его ст тур.

```
// File Wad.h
#ifndef __WAD__
#define __WAD__

#include    "fixmath.h"

////////////////////////// WAD-file structures ////////////////////////////

struct      WadHeader               // header of .WAD file
{
    char    sign [4];               // "IWAD" or "PWAD" signature
    long    dirEntries;             // # of directory entries
    long    dirOffset;              // offset of directory in a file
};

struct      DirEntry                // entry description
{
    long    offset;                 // data offset
    long    size;                   // data size
    char    name [8];               // resource name
};

struct  Vertex
{
    short   x;
    short   y;
};

struct      LineDef             // line from one vertex to another
{
```

```
    short   fromVertex;
    short   toVertex;
    short   attrs;              // attributes
    short   type;               // type
    short   tag;
    short   sideDefs [2];       // right sidedef, left sidedef (or -1)
};
struct      SideDef             // defines wall texture along LineDef
{
    short   xOffset, yOffset;   // texture offsets
    char    upper_tx [8];       // texture names   upper
    char    lower_tx [8];       // lower
    char    main_tx [8];        // main ( regular part if the wall )
    short   sector;             // sector
};
struct      Segment             // SEG, part of linedef,
{                               // bounding SubSector
    short   from, to;
    Angle   angle;              // angle of segment ( in BAM )
    short   lineDef;            // linedef
    short   lineSide;           // 0 if this seg lies on the right of linedef
                                // 1 if on the left
    short   lineOffset;         // offset distance along LineDef to
// the start of this seg
};
struct      SSector             // SubSector - convex part of Sector
{
    short   numSegs;            // # of segments for this ssector
    short   firstSeg;
};
struct      BBox                // bounding box for BSP trees
{
    short   y2, y1, x1, x2;
};
struct      Node                // BSP node struct
{
    short   x, y;               // from point
    short   dx, dy;             // in direction
    BBox    rightBox;           // bounding box for right subtree
    BBox    leftBox;            // bounding box for left subtree
    short   rightNode, leftNode;    // subtree ( or SSector ) pointers
};
struct      Sector              // area of map with constant heights & textures
{
    short   floorHeight;        // heights:
    short   ceilingHeight;
    char    floor_tx [8];       // textures:
    char    ceiling_tx [8];
    short   light;              // brightness of a sector (0-total dark, 255 - max)
```

```
    short    type;
    short    trigger;
};

struct   Thing
{
    short    x, y;
    short    angle;          // in degrees with 45 deg step
    short    type;
    short    attrs;
};

struct   RGB
{
    char     red;
    char     green;
    char     blue;
};

struct   PicHeader           // internal WAD picture format
{
    short    width;
    short    height;
    short    leftOffset;
    short    topOffset;
    long     colPtr [];      // pointers to PicColumn in WAD file
};

struct   PicColumn
{
    char     row;
    char     nonTransparentPixels;
    char     pixels [];
};

struct   TexturePatch
{
    short    xOffset;
    short    yOffset;
    short    pnamesNo;       // # of this patch in PNAMES resource
    short    one;            // unused, should be 1
    short    zero;           // unsused, should be 0
};

struct   TextureEntry        // in TEXTURE1 and TEXTURE2 resources
{
    char             textureName [8];
    short            zero1;  // unused, should always be 0
    short            zero2;  // unused, should always be 0
    short            width;
    short            height;
    short            zero3;
    short            zero4;
    short            numPatches;
    TexturePatch     patch [];
```

```
};
//////////////////////// non-WAD structures ////////////////////////////
struct   Pic
{
    short    width;
    short    height;
    short    leftOffset;
    short    topOffset;
    long  * colOffsets;
    char  * data;
};
struct   Texture
{
    char     name [8];
    int      width;
    int      height;
    char  *  data;
};
//////////////////////// WadFile class ////////////////////////////////
class   WadFile
{
public:
    WadHeader  hdr;
    DirEntry *     directory;
    char           fileName [128];
    int            file;

    WadFile ( const char * );
    ~WadFile ();

    void        loadLevel        ( int episode, int level );
    void        loadPlayPal      ();
    void        loadColorMap     ();
    Texture   * loadTexture      ( const char * name );
    Texture   * loadFloorTexture ( const char * name );
    Pic       * loadPic          ( const char * name );
    int         locateResource   ( const char * name );
    void        loadTexture1     ();
    void        loadPNames       ();
    void        applyPatch       ( Texture * tex, TexturePatch& patch );
};
////////////////////////////////////////////////////////////////////////

void    freePic  ( Pic * pic );
void    freeTexture ( Texture * tex );

//////////////////////////// global vars ////////////////////////////////

extern  Vertex *      vertices;
extern  LineDef *     lines;
extern  SideDef *     sides;
extern  Sector  *     sectors;
```

```
extern  Segment *     segs;
extern  SSector *     subSectors;
extern  Node  *       nodes;
extern  Thing  *      things;
extern  long   *      texture1;
extern  char   *      pnames;
extern  RGB           playPal [14][256];
extern  char          colorMap [34][256];

extern  int       numVertices;
extern  int       numLines;
extern  int       numSides;
extern  int       numSectors;
extern  int       numSegs;
extern  int       numSubSectors;
extern  int       numNodes;
extern  int       numThings;
extern  short     numPNames;
#endif
```

```
// File Wad.cpp
#include      <fcntl.h>
#include      <io.h>
#include      <malloc.h>
#include      <process.h>
#include      <stdio.h>
#include      <string.h>
#include      <sys\stat.h>
#include      "Wad.h"

Vertex  *     vertices      = NULL;
LineDef *     lines         = NULL;
SideDef *     sides         = NULL;
Sector  *     sectors       = NULL;
Segment *     segs          = NULL;
SSector *     subSectors    = NULL;
Node    *     nodes         = NULL;
Thing   *     things        = NULL;
long    *     texture1      = NULL;
char    *     pnames        = NULL;

RGB       playPal [14][256];
char      colorMap [34][256];

int       numVertices;
int       numLines;
int       numSides;
int       numSectors;
int       numSegs;
int       numSubSectors;
int       numNodes;
int       numThings;
short     numPNames;

//////////////////////////// WadFile methods ////////////////////////////
```

```
WadFile   WadFile ( const char * name )
{
      if ( ( file = open (strcpy (fileName, name), O_RDONLY|O_BINARY)) == -1)
      {
            printf ( "\nCannot open %s.", name );
            exit ( 1 );
      }

      read ( file, &hdr, sizeof ( hdr ) );

      if ( !strnicmp ( hdr.sign, "IWAD", 4 ) &&
          !strnicmp ( hdr.sign, "PWAD", 4 ) )
      {
            printf ( "\nNot a valid WAD file" );
            exit ( 1 );
      }

      directory = new DirEntry [hdr.dirEntries];

      if ( directory == NULL )
      {
            printf ( "\nInsufficint memory to load directory." );
            exit ( 1 );
      }

      lseek ( file, hdr.dirOffset, SEEK_SET );
      read  ( file, directory, hdr.dirEntries * sizeof ( DirEntry ) );
}

WadFile   ~WadFile ()
{
      close ( file );

      if ( directory != NULL )
            delete directory;
}

void   WadFile   loadLevel ( int episode, int level )
{
      char   levelName [5] = { 'E', '0' + episode, 'M', '0' + level, '\0' };

      for ( int i = 0; i < hdr.dirEntries; i++ )
            if ( !strcmp ( directory [i].name, levelName ) )
                  break;

      if ( i >= hdr.dirEntries )
            return;

      for ( i++; i < hdr.dirEntries; i++ )
      {
            if ( !strnicmp ( directory [i].name, "VERTEXES", 8 ) )
            {
                  if ( vertices != NULL )
                        delete  vertices;

                  numVertices = directory [i].size / sizeof ( Vertex );
                  vertices        = new Vertex [numVertices];

                  lseek ( file, directory [i].offset, SEEK_SET );
```

```
        read  ( file, vertices, directory [i].size );
}
else
if ( !strnicmp ( directory [i].name, "LINEDEFS", 8 ) )
{
        if ( lines != NULL )
                delete lines;

        numLines = directory [i].size / sizeof ( LineDef );
        lines        = new LineDef [numLines];

        lseek ( file, directory [i].offset, SEEK_SET );
        read  ( file, lines, directory [i].size );
}
else
if ( !strnicmp ( directory [i].name, "SIDEDEFS", 8 ) )
{
        if ( sides != NULL )
                delete sides;

        numSides = directory [i].size / sizeof ( SideDef );
        sides        = new SideDef [numSides];

        lseek ( file, directory [i].offset, SEEK_SET );
        read  ( file, sides, directory [i].size );
}
else
if ( !strnicmp ( directory [i].name, "SEGS", 8 ) )
{
        if ( segs != NULL )
                delete segs;

        numSegs = directory [i].size / sizeof ( Segment );
        segs         = new Segment [numSegs];

        lseek ( file, directory [i].offset, SEEK_SET );
        read  ( file, segs, directory [i].size );
}
else
if ( !strnicmp ( directory [i].name, "SECTORS", 8 ) )
{
        if ( sectors != NULL )
                delete sectors;

        numSectors = directory [i].size / sizeof ( Sector );
        sectors        = new Sector [numSectors];

        lseek ( file, directory [i].offset, SEEK_SET );
        read  ( file, sectors, directory [i].size );
}
else
if ( !strnicmp ( directory [i].name, "SSECTORS", 8 ) )
{
        if ( subSectors != NULL )
                delete subSectors;

        numSubSectors = directory [i].size / sizeof (SSector);
```

```
        subSectors      = new SSector [numSubSectors];
        lseek ( file, directory [i].offset, SEEK_SET );
        read  ( file, subSectors, directory [i].size );
}
else
if ( !strnicmp ( directory [i].name, "NODES", 8 ) )
{
        if ( nodes != NULL )
                delete nodes;
        numNodes = directory [i].size / sizeof ( Node );
        nodes       = new Node [numNodes];
        lseek ( file, directory [i].offset, SEEK_SET );
        read  ( file, nodes, directory [i].size );
}
else
if ( !strnicmp ( directory [i].name, "THINGS", 8 ) )
{
        if ( things != NULL )
                delete things;
        numThings = directory [i].size / sizeof ( Thing );
        things       = new Thing [numThings];
        lseek ( file, directory [i].offset, SEEK_SET );
        read  ( file, things, directory [i].size );
}
        else
                return;
        }
}
void   WadFile   loadPlayPal ()
{
        int    index = locateResource ( "PLAYPAL" );
        if ( index > -1 )
        {
                lseek ( file, directory [index].offset, SEEK_SET );
                read  ( file, playPal, sizeof ( playPal ) );
        }
}
void   WadFile :: loadColorMap ()
{
        int    index = locateResource ( "COLORMAP" );
        if ( index > -1 )
        {
                lseek ( file, directory [index].offset, SEEK_SET );
                read  ( file, colorMap, sizeof ( colorMap ) );
        }
}
int*  WadFile   locateResource ( const char * name )
{
```

```
        for ( register int i = 0; i < hdr.dirEntries; i++ )
            if ( !strnicmp ( directory [i].name, name, 8 ) )
                return i;

        return -1;
}

Pic  * WadFile   loadPic ( const char * picName )
{
        int    index = locateResource ( picName );

        if ( index < 0 )                        // not found
            return NULL;
                                                // allocate space for resource
        char     *     data    = (char *) malloc ( directory [index].size );
        PicHeader * picHdr = (PicHeader *) data;

        if ( data == NULL )
            return NULL;
                                                // read resource completely
        lseek ( file, directory [index].offset, SEEK_SET );
        read  ( file, data, directory [index].size );

        Pic * pic = new Pic;

        if ( pic == NULL )
        {
            free ( data );

            return NULL;
        }

        pic -> width        = picHdr -> width;
        pic -> height       = picHdr -> height;
        pic -> leftOffset   = picHdr -> leftOffset;
        pic -> topOffset    = picHdr -> topOffset;
        pic -> colOffsets   = picHdr -> colPtr;
        pic -> data         = data;

        return pic;
}

Texture * WadFile :: loadTexture ( const char * texName )
{
    if ( texture1 == NULL )             // will use only TEXTURE1
        loadTexture1 ();

    if ( texture1 == NULL )
        return NULL;

    if ( pnames == NULL )
        loadPNames ();

    if ( pnames == NULL )
        return NULL,

    for ( int i = 0; i < texture1 [0]; i++ )
    {
        TextureEntry * entry = (TextureEntry *)( texture1 [i+1] + (char *) texture1 );
```

```
    if ( !strnicmp ( entry -> textureName, texName, 8 ) )
    {
        Texture * tex = new Texture;    // init Texture

        if ( tex == NULL )
            return NULL;

        strncpy ( tex -> name, texName, 8 );

        tex -> width  = entry -> width;
        tex -> height = entry -> height;
        tex -> data   = (char *) malloc ( entry -> width * entry -> height );

        if ( tex -> data == NULL )
        {
            delete tex;

            return NULL;
        }

        memset ( tex -> data, 0, entry -> width * entry -> height );
                                // apply patches
        for ( int j = 0; j < entry -> numPatches; j++ )
            applyPatch ( tex, entry -> patch [j] );

        return tex;
    }
}

return NULL;                                // texture not found
}
void   WadFile   applyPatch ( Texture * tex, TexturePatch& patch )
{
    Pic   * pic = loadPic ( pnames + 2 + patch.pnamesNo * 8 );

    if ( pic == NULL )
        return;

    if ( tex -> data == NULL )
        return;

    for ( int col = 0; col < pic -> width; col++ )
    {
        if ( col + patch.xOffset < 0 )
            continue;

        char  * colData  = pic -> data + pic -> colOffsets [col];
        char  * texData;
        int     curRow;

        if ( *colData == '\xFF' )
            continue;

        do
        {
            int    row                  = *colData++;
            int    nonTransparentPixels = *colData++;
            int    count                = nonTransparentPixels;

            curRow = patch.yOffset + row;
```

```
                if ( count + curRow >= tex -> height )
                    count = tex -> height - 1 - curRow;

                colData++;                      // skip 1st pixel

                texData = tex->data + col + patch.xOffset + curRow * tex -> width;

                for ( register int y = 0; y < count; y++, curRow++ )
                {
                    if ( curRow >= 0 )
                        *texData = colData [y];

                    texData += tex -> width;
                }

                colData += nonTransparentPixels + 1;
            } while ( *colData != '\xFF' );
        }

        freePic ( pic );
}

Texture * WadFile :: loadFloorTexture ( const char * name )
{
        int  index = locateResource ( name );

        if ( index < 0 )
            return NULL;

        Texture * tex = new Texture;

        if ( tex == NULL )
            return NULL;

        strncpy ( tex -> name, name, 8 );

        tex -> width  = 64;
        tex -> height = 64;
        tex -> data   = (char *) malloc ( 4096 );

        if ( tex -> data == NULL )
        {
            delete tex;

            return NULL;
        }
        lseek ( file, directory [index].offset, SEEK_SET );
        read  ( file, tex -> data, 4096 );

        return tex;
}

void    WadFile :: loadTexture1 ()
{
        int     index = locateResource ( "TEXTURE1" );

        texture1 = (long *) malloc ( directory [index].size );

        if ( texture1 == NULL )
            return;

        lseek ( file, directory [index].offset, SEEK_SET );
```

```
    read  ( file, texture1, directory [index].size );
}
void   WadFile   loadPNames ()
{
    int    index = locateResource ( "PNAMES" );
    if ( index < 0 )
        return;
    lseek ( file, directory [index].offset, SEEK_SET );
    read  ( file, &numPNames, sizeof ( numPNames ) );
     pnames = (char *) malloc ( numPNames * 8 );
     if ( pnames == NULL )
        return;
     read ( file, pnames, numPNames * 8 );
}
////////////////////////////////////////////////////////////////////////////
void   freePic ( Pic * pic )
{
     free ( pic -> data );
     delete pic;
}
void   freeTexture ( Texture * tex )
{
     free ( tex -> data );
     delete tex;
}
```

Замечание. В игре DOOM принята своя система измерения углов. Полный угол в 2π радиан соответствует 65536 единицам. Значению угла 0 соответствует направлению на восток, значению 16384 (0x4000) - на север и т. д. Одним из преимуществ подобного подхода является то обстоятельство, что угол всегда нормирован, т. е. находится в отведенном диапазоне при использовании 16-битовых чисел для работы с углами.

Ниже приведена простейшая программа, реализующая просмотр заданного уровня без использования текстур.

```
// File doom1.cpp
#include    <alloc.h>
#include    <bios.h>
#include    <dos.h>
#include    <math.h>
#include    <stdio.h>
#include    <time.h>

#include    "Wad.h"

#define   SCREEN_WIDTH     320
#define   SCREEN_HEIGHT    200
#define   V_MIN            20
#define   HSCALE           160l
```

```
#define   VSCALE              160l

#define   FLOOR_COLOR     8
#define   CEILING_COLOR   7
#define   WALL_COLOR      1
#define   LOWER_WALL_COLOR 2
#define   UPPER_WALL_COLOR  3

#define   ESC         0x011b
#define   UP          0x4800
#define   DOWN        0x5000
#define   LEFT        0x4b00
#define   RIGHT       0x4d00
#define   CtrlLeft    0x7300
#define   CtrlRight   0x7400

////////////////////// Static Data ////////////////////////////

int         locX;                           // viewer's location
int         locY;
int         locZ;                           // viewer height
Angle       angle;                          // viewing direction
SSector *   curSector;                      // ssector, viewer located in
long        totalFrames = 0;

int         topLine [SCREEN_WIDTH];         // top horizon line
int         bottomLine [SCREEN_WIDTH];      // bottom horizon line

WadFile     wad ( "E:\\GAMES\\DOOM.ULT\\DOOM.WAD" );

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

void      drawView          ();
int       viewerOnRight     ( int node );
int       frontFacing       ( int from, int to );
void      drawNode          ( unsigned node );
void      drawSubSector     ( int s );
void      drawSeg           ( Segment * seg );
void      drawSimpleWall    ( int col1, int col2, int bottomHeight, int topHeight,
                              int v1, int v2 );
void      drawLowerWall     ( int col1, int col2, int bottomHeight, int topHeight,
                              int v1, int v2 );
void      drawUpperWall     ( int col1, int col2, int bottomHeight, int topHeight,
                              int v1, int v2 );
void      setVideoMode      ( int mode );
void      drawVertLine      ( int col, int top, int bottom, int color );

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

void          setVideoMode ( int mode )
{
    asm {
        mov     ax, mode
        int  10h
    }
}

void          drawVertLine ( int col, int top, int bottom, int color )
```

```
{
    char far * vptr = (char far *)MK_FP ( 0xA000, col + 320 * top );
    while ( top <= bottom )
    {
        * vptr = color;
        top++;
        vptr += 320;
    }
}
void        dumpLevel ()
{
    FILE * fp = fopen ( "level", "w" );
    fprintf ( fp, "\n\t %d Vertices\n", numVertices );
    for ( int i = 0; i < numVertices; i++ )
        fprintf ( fp, " %3d   (%5d, %5d)\n", i, vertices [i].x, vertices [i].y );
    fprintf ( fp, "\n\t %d LineDefs\n", numLines );
    for ( i = 0; i < numLines; i++ )
        fprintf ( fp, "%3d   From %3d to %3d   Right SideDef %3d   Left SideDef %3d\n'
            i, lines [i].fromVertex, lines [i].toVertex,
          lines [i].sideDefs [0], lines [i].sideDefs [1] );
    fprintf ( fp, "\n\t %d SideDefs\n", numSides );
    for ( i = 0; i < numSides; i++ )
        fprintf ( fp, "%3d   Upper %8s   Lower %8s   Main %8s   Sector %d\n",
            i, sides [i].upper_tx, sides [i].lower_tx,
            sides [i].main_tx, sides [i].sector );
    fprintf ( fp, "\n\t %d Segs\n", numSegs );
    for ( i = 0; i < numSegs; i++ )
        fprintf ( fp, "%3d From %3d to %3d Side %d\n", i, segs [i].from,
            segs [i].to, segs [i].lineSide );
    fprintf ( fp, "\n\t %d SSectors\n", numSubSectors );
    for ( i = 0; i < numSubSectors; i++ )
        fprintf ( fp, "%3d numSegs %4d FirstSeg %4d\n", i,
            subSectors [i].numSegs, subSectors [i].firstSeg );
    fprintf ( fp, "\n\t %d Nodes\n", numNodes );
    for ( i = 0; i < numNodes; i++ )
        fprintf ( fp, "%3d x %4d y %4d dx %4d dy %4d Right %4d(%4d) Left
%4d(%4d)\n",
            i, nodes [i].x, nodes [i].y, nodes [i].dx, nodes [i].dy,
            nodes [i].rightNode, nodes [i].rightNode & 0x7FFF,
            nodes [i].leftNode, nodes [i].leftNode & 0x7FFF );
    fclose ( fp );
}
main ()
{
    int  done = 0;
    initFixMath ();
    wad.loadLevel ( 1, 1 );
```

```
            // set viewer to start of level
for ( int i = 0; i < numThings; i++ )
    if ( things [i].type == 1 )
    {
        locX  = things [i].x;
        locY  = things [i].y;
        locZ  = 0;
        angle = (Angle) ( ( 0x4000l * things [i].angle ) / 90 );
        break;
    }
setVideoMode ( 0x13 );
clock_t start = clock ();
while ( !done )
{
    drawView ();
    if ( bioskey ( 1 ) )
        switch ( bioskey ( 0 ) )
        {
            case UP:
                locX += fixed2Int ( 8l * cosine ( angle ) );
                locY += fixed2Int ( 8l * sine   ( angle ) );
                break;
            case DOWN:
                locX -= fixed2Int ( 8l * cosine ( angle ) );
                locY -= fixed2Int ( 8l * sine   ( angle ) );
                break;
        case LEFT:
            angle += 1024;
            break;
        case RIGHT:
            angle -= 1024;
            break;
        case CtrlLeft:
            locX += fixed2Int ( 8l *
                    cosine (angle+ANGLE_90) );
            locY += fixed2Int ( 8l *
                    sine   (angle+ANGLE_90) );
            break;
        case CtrlRight:
            locX += fixed2Int ( 8l *
                    cosine (angle-ANGLE_90) );
            locY += fixed2Int ( 8l *
                    sine   (angle-ANGLE_90) );
            break;
        case ESC:
            done = 1;
            break;
}
```

```
    }
    float     totalTime = ( clock () - start ) / CLK_TCK;
    setVideoMode ( 0x03 );
    printf ( "\nFrames rendered    : %7ld",  totalFrames );
    printf ( "\nTotal time ( sec )    %7.2f", totalTime );
    printf ( "\nFPS                    %7.2f", totalFrames / totalTime );
}
void          drawView ()
{
    curSector = NULL;                                  // now unknown
    for ( int i = 0; i < SCREEN_WIDTH; i++ )     // reset horizon lines
    {
        topLine [i]    = 0;
        bottomLine [i] = SCREEN_HEIGHT - 1;
    }
    drawNode ( numNodes - 1 );                        // start drawing with root
    totalFrames++;
}
int  viewerOnRight ( int node )
{
    Node    * n = &nodes [node];
    return n->dy * (long)( locX - n->x ) >= n->dx *  (long)( locY - n->y );
}
int  frontFacing ( int from, int to )
{
    return  (long)( vertices [from].x - vertices [to].x ) *
        (long)( locY - vertices [to].y ) >=
          (long)( vertices [from].y - vertices [to].y ) *
        (long)( locX - vertices [to].x );
}
void          drawNode ( unsigned node )
{
    if ( node & 0x8000 )      // if it's a ssector => draw it
        drawSubSector ( node & 0x7FFF );
    else
    if ( viewerOnRight ( node ) )// otherwise draw tree in
    {                          // front-to-back order
        drawNode ( nodes [node].rightNode );
        drawNode ( nodes [node].leftNode );
    }
    else
    {
        drawNode ( nodes [node].leftNode );
        drawNode ( nodes [node].rightNode );
    }
}
void          drawSubSector ( int s )
```

```
{
    int    firstSeg = subSectors [s].firstSeg;  // get segments, forming  ssector
    int    numSegs  = subSectors [s].numSegs;
    Segment * seg  = &segs [firstSeg];          // pointer to current segment

    if ( curSector == NULL )                    // 1st ssector to be drawn
    {                                           // front-to-back will contain viewer
        SideDef * side = &sides [lines [seg->lineDef].sideDefs [ seg->lineSide]];

        curSector = &subSectors [s];            // so remember it &  adjust height
        locZ = 40 + sectors [side -> sector].floorHeight;
    }

    for ( int i = 0; i < numSegs; i++, seg++ ) // draw all front-facing  segments
        if ( frontFacing ( seg -> from, seg -> to ) )
            drawSeg ( seg );
}

void          drawSeg ( Segment * seg )
{
    int  x1 = vertices [seg->from].x - locX;
    int  y1 = vertices [seg->from].y - locY;
    int  x2 = vertices [seg->to].x - locX;
    int  y2 = vertices [seg->to].y - locY;
                                    // convert to local coords
    int  u1 = fixed2Int ( -x1*sine (angle)    + y1*cosine (angle) );
    int  v1 = fixed2Int (  x1*cosine (angle) + y1*sine (angle) );
    int  u2 = fixed2Int ( -x2*sine (angle)    + y2*cosine (angle) );
    int  v2 = fixed2Int (  x2*cosine (angle) + y2*sine (angle) );

    if ( v1 < V_MIN && v2 < V_MIN )          // segment behind the image plane
        return;

    if ( v1 < V_MIN )                        // clip 1st point
    {
        u1 += (int)(((long)(V_MIN - v1) * (long)(u2 - u1)) / (v2 - v1));
        v1  = V_MIN;
    }

    if ( v2 < V_MIN )                        // clip 2nd point
    {
        u2 += (int)(((long)(V_MIN - v2) * (long)(u2 - u1)) / (v2 - v1));
        v2  = V_MIN;
    }
                                    // project vertices to screen
    int  c1 = (int) ( 160l - ( u1 * HSCALE ) / v1 );
    int  c2 = (int) ( 160l - ( u2 * HSCALE ) / v2 );
                                    // reject invisible segs
    if ( c1 >= SCREEN_WIDTH || c2 < 0 || c2 <= c1 )
        return;

    SideDef * side = &sides [ lines [seg -> lineDef].sideDefs [
                      seg -> lineSide]];

    int  floorHeight   = sectors [side->sector].floorHeight;
    int  ceilingHeight = sectors [side->sector].ceilingHeight;

    if ( side -> main_tx [0] != '-' )        // simple wall from floor to ceiling
```

```
                                // no lower or upper
        drawSimpleWall ( c1, c2, locZ - floorHeight, ceilingHeight - locZ, v1, v2 );
    else
    {                           // otherwise we'll need adjacent sector heights
        SideDef * otherSide = &sides [lines [seg -> lineDef].sideDefs [
                                                seg -> lineSide ^ 1]];
                                // draw lower part
        if ( side -> lower_tx [0] != '-' )
            drawLowerWall ( c1, c2, locZ - floorHeight, locZ - sectors [otherSide ->
                                                        sector].floorHeight, v1, v2 );
        else
            drawLowerWall ( c1, c2, locZ - floorHeight, locZ - floorHeight, v1, v2 );
                                // draw upper part
        if ( side -> upper_tx [0] != '-' )
            drawUpperWall ( c1, c2, sectors [
                    otherSide->sector].ceilingHeight - locZ,
                    ceilingHeight - locZ, v1, v2 );
        else
            drawUpperWall ( c1, c2, ceilingHeight - locZ,
                    ceilingHeight - locZ, v1, v2 );

    }
}
// Draw walls using horizon lines - only parts between horizon lines are drawn
void    drawSimpleWall ( int col1, int col2, int bottomHeight, int topHeight,
                int v1, int v2 )
{
    Fixed   b1 = int2Fixed ( 100l + ( bottomHeight * VSCALE ) / v1 );
    Fixed   b2 = int2Fixed ( 100l + ( bottomHeight * VSCALE ) / v2 );
    Fixed   t1 = int2Fixed ( 100l - ( topHeight * VSCALE ) / v1 );
    Fixed   t2 = int2Fixed ( 100l - ( topHeight * VSCALE ) / v2 );
    Fixed   db = ( b2 - b1 ) / ( col2 - col1 );
    Fixed   dt = ( t2 - t1 ) / ( col2 - col1 );
    int  top, bottom;

    if ( col1 < 0 )         // clip invisible part
    {
        b1  -= col1 * db;
        t1  -= col1 * dt;
        col1 = 0;
    }

    if ( col2 > 320 )
        col2 = 320;

    for ( int col = col1; col < col2; col++ )
    {
        top    = fixed2Int ( t1 );
        bottom = fixed2Int ( b1 );
        t1    += dt;
        b1    += db;

        if ( topLine [col] > bottomLine [col] ) // nothing to draw here
```

```
        continue;
    if ( top > bottomLine [col] )
        top = bottomLine [col];
    if ( bottom < topLine [col] )
        bottom = topLine [col];
    if ( top >= topLine [col] )      // draw ceiling
    {
        drawVertLine ( col, topLine [col], top, CEILING_COLOR );
        topLine [col] = ++top;
    }
    else                             // otherwise correct to
        top = topLine [col];         // draw only visible part
    if ( bottom <= bottomLine [col] ) // draw floor
    {
        drawVertLine ( col, bottom, bottomLine [col], FLOOR_COLOR );
        bottomLine [col] = --bottom;
    }
    else
        bottom = bottomLine [col];
    topLine [col]    = bottom + 1;
    bottomLine [col] = top - 1;
                             // now draw visible part  of wall
    drawVertLine ( col, top, bottom, WALL_COLOR );
        }
}
void    drawLowerWall ( int col1, int col2, int bottomHeight, int topHeight,
                        int v1, int v2 )
{
    Fixed   b1 = int2Fixed ( 100l + ( bottomHeight * VSCALE ) / v1 );
    Fixed   b2 = int2Fixed ( 100l + ( bottomHeight * VSCALE ) / v2 );
    Fixed   t1 = int2Fixed ( 100l + ( topHeight * VSCALE ) / v1 );
    Fixed   t2 = int2Fixed ( 100l + ( topHeight * VSCALE ) / v2 );
    Fixed   db = ( b2 - b1 ) / ( col2 - col1 );
    Fixed   dt = ( t2 - t1 ) / ( col2 - col1 );
    int  top, bottom;
    if ( col1 < 0 )          // clip invisible part
    {
        b1  -= col1 * db;
        t1  -= col1 * dt;
        col1 = 0;
    }
    if ( col2 > 320 )
        col2 = 320;
    for ( int col = col1; col < col2; col++ )
    {
        top    = fixed2Int ( t1 );
        bottom = fixed2Int ( b1 );
        t1     += dt;
```

```
        b1      += db;

        if ( topLine [col] > bottomLine [col] )
            continue;

        if ( top < topLine [col] )
            top = topLine [col];        // adjust top

        if ( bottom < topLine [col] )
            bottom = topLine [col];

        if ( bottom <= bottomLine [col] ) // draw floor
            drawVertLine ( col, bottom, bottomLine [col]++, FLOOR_COLOR );
        else
            bottom = bottomLine [col];

        if ( top < bottom )                 // now draw visible part
        {                                   // of wall
            drawVertLine ( col, top, bottom, LOWER_WALL_COLOR );
            if ( top < bottomLine [col] )
                bottomLine [col] = top - 1;
        }
        else
            if ( bottom < bottomLine [col] )
                bottomLine [col] = bottom - 1;
    }
}

void    drawUpperWall ( int col1, int col2, int bottomHeight, int topHeight,
                        int v1, int v2 )
{
    Fixed   b1 = int2Fixed ( 100l - ( bottomHeight * VSCALE ) / v1 );
    Fixed   b2 = int2Fixed ( 100l - ( bottomHeight * VSCALE ) / v2 );
    Fixed   t1 = int2Fixed ( 100l - ( topHeight * VSCALE ) / v1 );
    Fixed   t2 = int2Fixed ( 100l - ( topHeight * VSCALE ) / v2 );
    Fixed   db = ( b2 - b1 ) / ( col2 - col1 );
    Fixed   dt = ( t2 - t1 ) / ( col2 - col1 );
    int  top, bottom;

    if ( col1 < 0 )         // clip invisible part
    {
        b1  -= col1 * db;
        t1  -= col1 * dt;
        col1 = 0;
    }

    if ( col2 > 320 )
        col2 = 320;

    for ( int col = col1; col < col2; col++ )
    {
        top     = fixed2Int ( t1 );
        bottom = fixed2Int ( b1 );
        t1      += dt;
        b1      += db;

        if ( topLine [col] > bottomLine [col] )
            continue;
```

```
            if ( top > bottomLine [col] )
                top = bottomLine [col];

            if ( top >= topLine [col] )              // draw ceiling
                drawVertLine ( col, topLine [col]++, top, CEILING_COLOR );
            else                                     // otherwise correct to
                top = topLine [col];                 // draw only visible part

            if ( bottom > bottomLine [col] )
                bottom = bottomLine [col];

            if ( top < bottom )                      // now draw visible part
            {                                        // of wall
                drawVertLine ( col, top, bottom, UPPER_WALL_COLOR );
                if ( bottom > topLine [col] )
                    topLine [col] = bottom + 1;
            }
            else
                if ( top > topLine [col] )
                    topLine [col] = top + 1;
        }
    }
```

Рассмотрим некоторые приемы, позволяющие повысить быстродействие данной программы без переписывания ее на ассемблер. Основная идея заключается в как можно более раннем отбрасывании тех фрагментов сцены (SubSector и Seg), которые заведомо не могут быть видны.

В первую очередь можно воспользоваться тем, что для любого узла дерева в WAD-файле хранится минимальный прямоугольник (ограничивающее тело), полностью содержащий соответствующее поддерево внутри себя (фактически заданы минимальные и максимальные значения x и y) и в процедуре drawNode перед рекурсивным вызовом себя для обработки поддерева, и проверить, находится ли это поддерево (его ограничивающий прямоугольник) внутри области видимости (угла с вершиной в положении наблюдателя и биссектрисой, совпадающей с направлением взгляда наблюдателя). Если поддерево с областью видимости общих точек не имеет, то его можно сразу же отбросить.

Существуют разные методы проверки того, попадает ли прямоугольник в заданный угол. Мы воспользуемся самым простым из них - проведем через позицию наблюдателя прямые, параллельные координатным осям. Они разобьют всю плоскость на 4 части.

Дальше проверим, в какие части попадает область видимости и попадает ли в эти части заданный прямоугольник. Так прямоугольники A и D на рис. 13.22 сразу же отбрасываются, прямоугольник *B* действительно попадает в рассматриваемую область, а прямоугольник *C* остается, хотя в область видимости и не попадает.

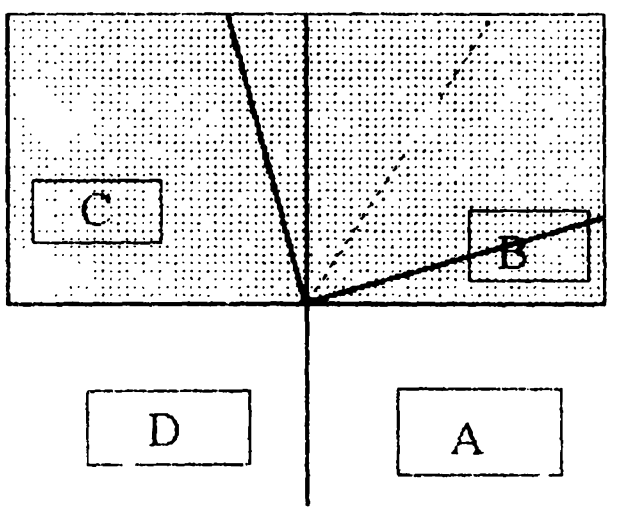


Рис. 13.22

Хотя этот метод все не попадающие в область видимости прямоугольники и не отбрасывает, однако он крайне прост и умножений и делений не требует.

С не меньшей эффективностью можно использовать учет уже полностью заполненных столбцов на экране, причем сразу в двух местах - при выводе очередной стены (если все столбцы, соответствующие этой стене, уже заполнены, то стены не выводятся) и при обходе дерева (если все столбцы экрана уже выведены, то обход дерева прекращается).

Рассмотрим, каким образом осуществляется вычисление параметров текстуры для вертикальных стен произвольной ориентации. Основным моментом здесь является вычисление параметра текстуры (номера столбца текстуры) для соответствующего вертикального столбца стены. Рассмотрим, каким путем это можно сделать.

Пусть у нас на плоскости (x, y) расположен отрезок стены с начальной точкой (x_1, z_1) и концом в точке (x_2, z_2).

Пусть перспективное проектирование осуществляется по следующей формуле: $c = x/z$, где c - экранная координата точки (x, z), принадлежащей отрезку стены. Интересующий нас параметр текстуры t - это расстояние вдоль стены от точки (x_1, z_1) до точки (x, z) (рис. 13.23.).

Покажем, что величины $1/z$ и t/z линейно зависят от c. Точку (x, z) можно выразить следующим образом:

$$\begin{cases} x = x_1 + t\cos\alpha, \\ z = z_1 + t\sin\alpha. \end{cases}$$

Выразим t через z из этих соотношений:

$$t = \frac{z - z_1}{\sin\alpha}.$$

Отсюда выразим x через z, используя предыдущее соотношение:

$$x = x_1 + (z - z_1)ctg\alpha$$

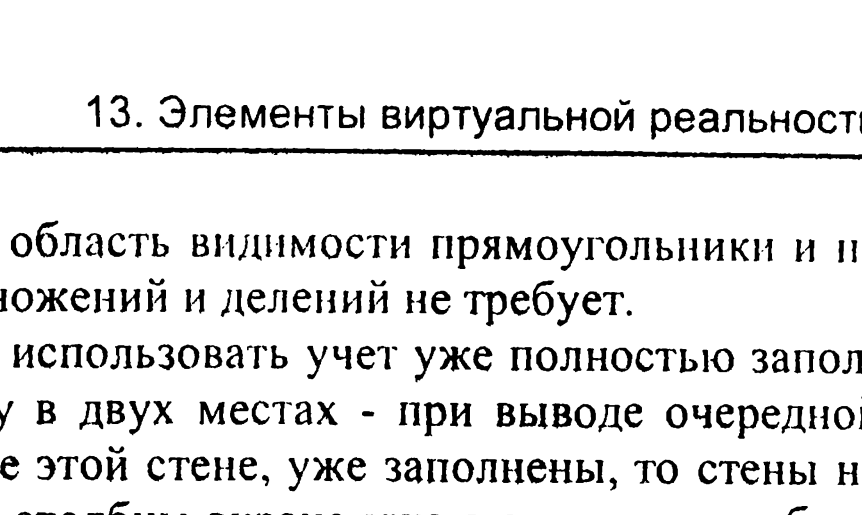


Рис. 13.23

Используя выражение для c, получим выражение для $1/z$:

$$\frac{1}{z} = \frac{c - ctg\alpha}{x_1 - z_1 ctg\alpha}$$

Отсюда легко получить выражение для t/z:

$$\frac{t}{z} = \frac{z - z_1}{z\sin\alpha} = \frac{1}{\sin\alpha} - \frac{z_1}{\sin\alpha}\frac{1}{z} = \frac{1}{\sin\alpha} - \frac{z_1}{\sin\alpha}\frac{c - ctg\alpha}{x_1 - z_1 ctg\alpha}$$

Легко видеть, что выличины $1/z$ и t/z действительно линейно зависят от c.

Таким образом, для вывода текстурированной стены достаточно кроме вычисления столбцов, занимаемых стеной, найти еще и искомые величины в концах стены. При выводе стены от столбца к столбцу параметр t находится как частное этих величин, а сами эти величины линейно изменяются от столбца к столбцу.

Ниже приводим пример программы, реализующей текстурирование верти ных поверхностей.

```
// File doom4.cpp
#ifdef __WATCOMC__
#include      <i86.h>
#endif

#include      <bios.h>
#include      <dos.h>
#include      <malloc.h>
#include      <math.h>
#include      <stdio.h>
#include      <time.h>

#include      "wad.h"
#include      "texdict.h"

#define SCREEN_WIDTH          320
#define SCREEN_HEIGHT         200
#define V_MIN                 20
#define HSCALE                160l
#define VSCALE                160l

#define FLOOR_COLOR           8
#define CEILING_COLOR         7
#define WALL_COLOR            1
#define LOWER_WALL_COLOR      2
#define UPPER_WALL_COLOR      3

#define     ESC       0x011b
#define     UP        0x4800
#define     DOWN      0x5000
#define     LEFT      0x4b00
#define     RIGHT     0x4d00
#define     CtrlLeft  0x7300
#define     CtrlRight 0x7400

struct  VertexCache
{
      int    u;
      int    v;
      int    c;
      int    frame;
};

struct DPMIDosPtr
{
      unsigned short segment;
      unsigned short selector;
};

////////////////////// Static Data ////////////////////////////
struct
{
      unsigned long   edi;
      unsigned long   esi;
```

```
        unsigned long   ebp;
        unsigned long   esp;
        unsigned long   ebx;
        unsigned long   edx;
        unsigned long   ecx;
        unsigned long   eax;
        unsigned short  flags;
        unsigned short  es, ds, fs, gs, ip, cs, sp, ss;
} rmRegs;

REGS        regs;
SREGS       sregs;

int         locX;                           // viewer's location
int         locY;
int         locZ;                           // viewer height
Angle       angle;                          // viewing direction
SSector *   curSector;                      // ssector, viewer located in
long        totalFrames = 0;

int         topLine [SCREEN_WIDTH];         // top horizon line
int         bottomLine [SCREEN_WIDTH];      // bottom horizon line
int         colsLeft;                       // # on unfinished columns
char        colFilled [SCREEN_WIDTH];       // whether column is already filled

VertexCache     *       vertexCache;
WadFile                 wad ( "E:\\GAMES\\DOOM.ULT\\DOOM.WAD" );
TextureDictionary       texDict ( &wad );

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

void    drawView        ();
void    drawNode        ( unsigned node );
void    drawSubSector   ( int s );
void    drawSeg         ( Segment * seg );
void    drawSimpleWall ( Texture * tex, SideDef * side, int col1, int col2,
                         int bottomHeight, int topHeight, Fixed t1, Fixed t2,
                         int v1, int v2 );
void    drawLowerWall  ( Texture * tex, SideDef * side, int col1, int col2,
                         int bottomHeight, int topHeight, Fixed t1, Fixed t2,
                         int v1, int v2 );
void    drawUpperWall  ( Texture * tex, SideDef * side, int col1, int col2,
                         int bottomHeight, int topHeight, Fixed t1, Fixed t2,
                         int v1, int v2 );
void    setVideoMode    ( int mode );
void    drawVertLine    ( int col, int top, int bottom, int color );

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

Fixed   fixedDist ( int u1, int v1, int u2, int v2, Angle angle )
{
        int     du = u2 - u1;
        int     dv = v2 - v1;
        Fixed   dist;

        if ( abs ( du ) > abs ( dv ) )
                dist = du * invCosine ( angle );
```

```
        else
            dist = dv * invSine ( angle );
        return dist;
}
inline  int     viewerOnRight ( int node )
{
        Node    * n = &nodes [node];
        return n->dy * (long)( locX - n->x ) >=
            n->dx * (long)( locY - n->y );
}
inline  int     frontFacing ( int from, int to )
{
        return (long)( vertices [from].x - vertices [to].x ) *
            (long)( locY - vertices [to].y ) >=
            (long)( vertices [from].y - vertices [to].y ) *
            (long)( locX - vertices [to].x );
}
void    RMVideoInt ()               // execute real-mode video interrupt
{
        segread ( &sregs );
        regs.w.ax  = 0x0300;
        regs.w.bx  = 0x0010;
        regs.w.cx  = 0;
        regs.x.edi = FP_OFF ( &rmRegs );
        sregs.es   = FP_SEG ( &rmRegs );
        int386x ( 0x31, &regs, &regs, &sregs );
}
void    DPMIAllocDosMem ( DPMIDosPtr& ptr, int paras )
{
        regs.w.ax = 0x0100;         // allocate DOS memory
        regs.w.bx = paras;          // # of memory in paragraphs
        int386 ( 0x31, &regs, &regs );
        ptr.segment  = regs.w.ax;   // real-mode segment of the block
        ptr.selector = regs.w.dx;   // selector for the allocated block
}
void    DPMIFreeDosMem ( DPMIDosPtr& ptr )
{
        regs.w.ax = 0x0101;         // free DOS memory
        regs.w.dx = ptr.selector;
        int386 ( 0x31, &regs, &regs );
}
void    setVideoMode ( int mode )
{
        _asm {
            mov     ax, word ptr mode
            int     10h
```

```
    }
}
void   setPalette ( RGB * palette )
{
    DPMIDosPtr    palPool;
    DPMIAllocDosMem ( palPool, 3*256 / 16 );
    RGB * tmpPal = (RGB *) (palPool.segment * 16);
    for ( int i = 0; i < 256; i++ )        // convert from 8-bit to
    {                          // 6-bit values
        tmpPal [i].red   = palette [i].red   >> 2;
        tmpPal [i].green = palette [i].green >> 2;
        tmpPal [i].blue  = palette [i].blue  >> 2;
    }
    rmRegs.eax = 0x1012;              // set palette
    rmRegs.ebx = 0;                   // starting index
    rmRegs.ecx = 256;                 // # of pal entries
    rmRegs.es  = palPool.segment;
    rmRegs.edx = 0;
    RMVideoInt ();                    // execute video interrupt
    DPMIFreeDosMem ( palPool );   // free memory
}
void   drawVertLine ( int col, int top, int bottom, int color )
{
    char * vptr = col + 320 * top + (char *)0xA0000l;
    while ( top <= bottom )
    {
        * vptr = color;
        vptr  += 320;
        top++;
    }
}
void   drawTexturedLine ( int col, int top, int bottom, Texture * tex,
                  int textureOffset, Fixed texY, Fixed dTexY )
{
    char * vptr = col + 320 * top + (char *)0xA0000l;
   while ( top <= bottom )
   {
       * vptr = tex -> data [textureOffset + tex -> width *
                   (fixed2Int (texY) & (tex->height-1))];
       vptr  += SCREEN_WIDTH;
       texY  += dTexY;
       top++;
   }
}
main ()
{
   int    done = 0;
```

```
initFixMath          ();
wad.loadLevel        ( 1, 1 );
wad.loadPlayPal      ();
wad.loadColorMap     ();

vertexCache = new VertexCache [numVertices];

                     // set viewer to start of level
for ( int i = 0; i < numThings; i++ )
    if ( things [i].type == 1 )
    {
        locX  = things [i].x;
        locY  = things [i].y;
        locZ  = 0;
        angle = (Angle) ( ( 0x4000l * things [i].angle ) / 90 );

        break;
    }
                     // load all textures
 for ( i = 0; i < numSides; i++ )
 {
      texDict.getTexture ( sides [i].main_tx );
      texDict.getTexture ( sides [i].lower_tx );
      texDict.getTexture ( sides [i].upper_tx );
 }

 setVideoMode ( 0x13 );
 setPalette   ( playPal [0] );
   clock_t start = clock ();
while ( !done )
{
      drawView ();
        if ( _bios_keybrd ( _KEYBRD_READY ) )
            switch ( _bios_keybrd ( _KEYBRD_READ ) )
            {
                  case UP:
                        locX += fixed2Int (8l*cosine(angle));
                        locY += fixed2Int (8l*sine  (angle));
                        break;

                  case DOWN:
                        locX -= fixed2Int (8l*cosine (angle));
                        locY -= fixed2Int (8l*sine   (angle));
                        break;

                  case LEFT:
                        angle += 1024;
                        break;

                  case RIGHT:
                        angle -= 1024;
                        break;

                  case CtrlLeft:
                        locX += fixed2Int ( 8l *
```

```
                                cosine (angle+ANGLE_90));
                        locY += fixed2Int ( 8l *
                                sine   (angle+ANGLE_90));
                        break;

                case CtrlRight:
                        locX += fixed2Int (8l *
                                cosine (angle-ANGLE_90));
                        locY += fixed2Int ( 8l *
                                sine   (angle-ANGLE_90));
                        break;

                case ESC:
                        done = 1;
                        break;
                }
        }

        float   totalTime = ( clock () - start ) / CLK_TCK;

        setVideoMode ( 0x03 );

        printf ( "\nFrames rendered      %7ld",  totalFrames );
        printf ( "\nTotal time ( sec )   %7.2f", totalTime );
        printf ( "\nFPS                  %7.2f", totalFrames / totalTime );
}

void    drawView ()
{
        curSector = NULL;                       // now unknown
        colsLeft  = SCREEN_WIDTH;

        for ( int i = 0; i < SCREEN_WIDTH; i++ )  // reset horizon lines
        {
                colFilled  [i]    = 0;
                topLine    [i]    = 0;
                bottomLine [i] = SCREEN_HEIGHT - 1;
        }

        drawNode ( numNodes - 1 );              // start drawing with root
        totalFrames++;
}

// check whether the bounding box is in the viewing frustrum
int     boxInVCone ( BBox& box )
{
        Angle   leftAngle = angle + 0x2000;

        if ( leftAngle < ANGLE_90 )
        {
                if ( box.x2 > locX )
                        return 1;
        }
        else
        if ( leftAngle < ANGLE_180 )
        {
                if ( box.y2 > locY )
                        return 1;
```

```
        }
        else
        if ( leftAngle < ANGLE_270 )
         {
              if ( box.x1 < locX )
                   return 1;
        }
        else
        {
              if ( box.y1 < locY )
                   return 1;
        }
        return 0;
}
void    drawNode ( unsigned node )
{
        if ( node & 0x8000 )          // if it's a ssector => draw it
             drawSubSector ( node & 0x7FFF );
        else
        if ( viewerOnRight ( node ) )  // otherwise draw tree in front-to-back order
{
             if ( colsLeft > 0 && boxInVCone ( nodes [node].rightBox ) )
                   drawNode ( nodes [node].rightNode );

             if ( colsLeft > 0 && boxInVCone ( nodes [node].leftBox ) )
                   drawNode ( nodes [node].leftNode );
        }
        else
        {
              if ( colsLeft > 0 && boxInVCone ( nodes [node].leftBox ) )
                   drawNode ( nodes [node].leftNode );

              if ( colsLeft > 0 && boxInVCone ( nodes [node].rightBox ) )
                   drawNode ( nodes [node].rightNode );
        }
}
void    drawSubSector ( int s )
{
     int            firstSeg     = subSectors [s].firstSeg;    // get segments, forming ssect
     int            numSegs      = subSectors [s].numSegs;
     Segment *      seg          = &segs [firstSeg];           // pointer to current segment

     if ( curSector == NULL )                                  // 1st ssector to be drawn usin
      {                                                  // front-to-back will contain viewer
           SideDef * side = &sides [lines [seg->lineDef].sideDefs
                                    [seg->lineSide]];

           curSector = &subSectors [s];          // so remember it & adjust height
           locZ = 40 + sectors [side -> sector].floorHeight;
      }

      for ( int i = 0; i < numSegs; i++, seg++ )     // draw all front-facing segments
           if ( frontFacing ( seg -> from, seg -> to ) )
```

```
                drawSeg ( seg );
}
void    drawSeg ( Segment * seg )
{
      if ( vertexCache [seg -> from].frame != totalFrames )   // vertex has
      {       // not been projected this frame
            int             x1   = vertices [seg->from].x - locX;
            int             y1   = vertices [seg->from].y - locY;
            VertexCache *   ptr  = &vertexCache [seg -> from];

            ptr -> frame         = totalFrames;
            ptr -> u             = fixed2Int (-x1*sine(angle) +y1*cosine(angle));
            ptr -> v             = fixed2Int ( x1*cosine(angle)+y1*sine(angle));

            if ( ptr -> v >= V_MIN )
                  ptr -> c = (int) (160l - (ptr->u*HSCALE) / ptr->v);
      }
      if ( vertexCache [seg -> to].frame != totalFrames )
      {
            int             x2  = vertices [seg->to].x - locX;
            int             y2  = vertices [seg->to].y - locY;
            VertexCache *   ptr  = &vertexCache [seg -> to];

            ptr -> frame         = totalFrames;
            ptr -> u             = fixed2Int (-x2*sine(angle)  +y2*cosine(angle));
            ptr -> v             = fixed2Int ( x2*cosine(angle)+y2*sine(angle));

            if ( ptr -> v >= V_MIN )
                  ptr -> c = (int) (160l - (ptr->u*HSCALE) / ptr->v);
}
                                          // convert to local coords
int     v1 = vertexCache [seg -> from].v;
int     v2 = vertexCache [seg -> to].v;

if ( v1 < V_MIN && v2 < V_MIN )      // segment behind the return;
                                      // image plane

int       u1 = vertexCache [seg -> from].u;
int       u2 = vertexCache [seg -> to].u;
Fixed   t1 = int2Fixed ( seg -> lineOffset );
Fixed   t2 = t1 + fixedDist ( u1, v1, u2, v2,
                        ANGLE_90 - seg -> angle + angle );

if ( v1 < V_MIN )                          // clip 1st point
{
      int oldU = u1;
      int oldV = v1;

      u1 += (int)(((long)(V_MIN - v1)*(long)(u2 - u1)) / (v2 - v1));
      v1  = V_MIN;
      t1  = int2Fixed ( seg -> lineOffset ) + fixedDist ( oldU, oldV,
                  u1, v1, ANGLE_90 - seg->angle + angle);
}
if ( v2 < V_MIN )                          // clip 2nd point
{
```

```
        u2 += (int)(((long)(V_MIN - v2)*(long)(u2 - u1)) / (v2 - v1));
        v2  = V_MIN;
        t2  = t1 + fixedDist ( u1, v1, u2, v2,
                        ANGLE_90 - seg -> angle + angle );
}
                                        // project vertices to screen
int     c1 = (int) ( 160l - ( u1 * HSCALE ) / v1 );
int     c2 = (int) ( 160l - ( u2 * HSCALE ) / v2 );
                                        // reject invisible segs
if ( c1 >= SCREEN_WIDTH || c2 < 0 || c2 <= c1 )
        return;

int     canExit = 1;        // check whether any column in range [c1, c2]
                            // remains unfilled
for ( register int i = c1; i < c2; i++ )
        if ( !colFilled [i] )
        {
                canExit = 0;
                break;
        }

if ( canExit )
        return;

SideDef * side = &sides [ lines [seg -> lineDef].sideDefs [
                        seg -> lineSide]];

int     floorHeight   = sectors [side->sector].floorHeight;
int     ceilingHeight = sectors [side->sector].ceilingHeight;

if ( side -> main_tx [0] != '-' )           // simple wall from floor to ceiling
                                            // no lower or upper
        drawSimpleWall ( texDict.getTexture ( side -> main_tx ), side,
                        c1, c2, locZ - floorHeight,
                        locZ - ceilingHeight, t1, t2, v1, v2 );
else
 {                                          // otherwise we'll need adjacent sector heights
          SideDef * otherSide = &sides [lines [seg -> lineDef].sideDefs·
                                [seg -> lineSide ^ 1]];

                                            // draw lower part
        if ( side -> lower_tx [0] != '-' )
                drawLowerWall ( texDict.getTexture (side->lower_tx),
                                side, c1, c2, locZ - floorHeight,
                                locZ - sectors [otherSide
                                ->sector].floorHeight, t1, t2,
                                v1, v2 );
        else
                drawLowerWall ( NULL, side, c1, c2, locZ - floorHeight,
                                locZ - floorHeight, t1, t2, v1, v2 );

                                            // draw upper part
        if ( side -> upper_tx [0] != '-' )
                drawUpperWall ( texDict.getTexture(side->upper_tx),
                                side, c1, c2,
                                sectors [otherSide->sector].
```

```
                        ceilingHeight - locZ,
                        ceilingHeight - locZ, t1, t2, v1, v2 );
            else
                drawUpperWall ( NULL, side, c1, c2,
                        ceilingHeight - locZ,
                        ceilingHeight - locZ, t1, t2, v1, v2 );

        }
}
// Draw walls using horizon lines - only parts between horizon lines are drawn
void    drawSimpleWall ( Texture * tex, SideDef * side, int col1, int col2,
                int bottomHeight, int topHeight,
                Fixed tOffs1, Fixed tOffs2, int v1, int v2 )
{
        Fixed   b1      = int2Fixed ( 100l + ( bottomHeight * VSCALE ) / v1 );
        Fixed   b2      = int2Fixed ( 100l + ( bottomHeight * VSCALE ) / v2 );
        Fixed   t1      = int2Fixed ( 100l + ( topHeight * VSCALE ) / v1 );
        Fixed   t2      = int2Fixed ( 100l + ( topHeight * VSCALE ) / v2 );
        Fixed   db      = ( b2 - b1 ) / ( col2 - col1 );
        Fixed   dt      = ( t2 - t1 ) / ( col2 - col1 );
        Fixed   invV1   = (ONE<<6) / v1;
        Fixed   invV2   = (ONE<<6) / v2;
        Fixed   tInvV1  = (tOffs1<<6) / v1;
        Fixed   tInvV2  = (tOffs2<<6) / v2;
        Fixed   dInvV   = (invV2 - invV1) / (col2 - col1);
        Fixed   dtInvV  = (tInvV2 - tInvV1 ) / (col2 - col1);
        Fixed   invV    = invV1;
        Fixed   tInvV   = tInvV1;
        Fixed   texY;
        Fixed   dTexY;
        int     top, bottom;
        int     newTop;
        int     tOffset;
        if ( col1 < 0 )         // clip invisible part
        {
                b1      -= col1 * db;
                t1      -= col1 * dt;
                invV    -= col1 * dInvV;
                tInvV -= col1 * dtInvV;
                col1    = 0;
        }
        if ( col2 > 320 )
                col2 = 320;
        for ( int col = col1; col < col2; col++ )
        {
                top         = fixed2Int ( t1 );
                bottom      = fixed2Int ( b1 );
                newTop      = top;
                dTexY       = int2Fixed (bottomHeight - topHeight + 1) / (bottom - top + 1);
            texY    = int2Fixed (side->yOffset) - (bottom - top + 1) * dTexY;
```

```
tOffset = tInvV / invV;
      t1     += dt;
      b1     += db;
invV         += dInvV;
      tInvV += dtInvV;

      if ( tex != NULL )
            tOffset &= tex -> width - 1;

      if ( topLine [col] > bottomLine [col] ) // nothing to draw here
            continue;

      if ( top > bottomLine [col] )
            top = bottomLine [col];

      if ( bottom < topLine [col] )
            bottom = topLine [col];

      if ( top >= topLine [col] )            // draw ceiling
{
            drawVertLine (col, topLine [col], top, CEILING_COLOR);
           topLine [col] = ++top;
      }
      else                                    // otherwise correct to
           top = topLine [col];               // draw only visible part

    if ( bottom <= bottomLine [col] )        // draw floor
    {
           drawVertLine ( col, bottom, bottomLine [col],
                          FLOOR_COLOR );
                bottomLine [col] = --bottom;
          }
          else
                bottom = bottomLine [col];

          topLine [col]    = bottom + 1;
          bottomLine [col] = top - 1;
                                          // now draw visible part of wall
        texY -= (newTop-top)*dTexY;

        if ( tex != NULL )
          drawTexturedLine ( col, top, bottom, tex, tOffset, texY, dTexY );
      else
          drawVertLine ( col, top, bottom, WALL_COLOR );

          colFilled [col] = 0xFF;
          colsLeft--;
      }
}
void   drawLowerWall ( Texture * tex, SideDef * side, int col1, int col2,
                int bottomHeight, int topHeight,
                Fixed tOffs1, Fixed tOffs2, int v1, int v2 )
{
      Fixed  b1      = int2Fixed ( 100l + ( bottomHeight * VSCALE ) / v1 );
      Fixed  b2      = int2Fixed ( 100l + ( bottomHeight * VSCALE ) / v2 );
      Fixed  t1      = int2Fixed ( 100l + ( topHeight * VSCALE ) / v1 );
      Fixed  t2      = int2Fixed ( 100l + ( topHeight * VSCALE ) / v2 );
```

```
Fixed  db      = ( b2 - b1 ) / ( col2 - col1 );
Fixed  dt      = ( t2 - t1 ) / ( col2 - col1 );
Fixed  invV1   = (ONE<<6) / v1;
Fixed  invV2   = (ONE<<6) / v2;
Fixed   tInvV1 = (tOffs1<<6) / v1;
Fixed  tInvV2  = (tOffs2<<6) / v2;
Fixed  dInvV   = (invV2 - invV1) / (col2 - col1);
Fixed  dtInvV   = (tInvV2 - tInvV1 ) / (col2 - col1);
Fixed  invV    = invV1;
Fixed  tInvV   = tInvV1;
Fixed  texY;
Fixed  dTexY;
int    top, bottom;
int    newTop;
int    tOffset;

if ( col1 < 0 )        // clip invisible part
{
      b1    -= col1 * db;
      t1    -= col1 * dt;
      invV  -= col1 * dInvV;
      tInvV -= col1 * dtInvV;
      col1   = 0;
}

if ( col2 > 320 )
      col2 = 320;

for ( int col = col1; col < col2; col++ )
{
      top        = fixed2Int ( t1 );
      bottom     = fixed2Int ( b1 );
      newTop     = top;
      dTexY      = int2Fixed (bottomHeight - topHeight + 1) / (bottom - top + 1);
      texY       = int2Fixed ( side -> yOffset ) - (bottom - top + 1) * dTexY;
      tOffset    = tInvV / invV;
      t1         += dt;
      b1         += db;
      invV       += dInvV;
      tInvV      += dtInvV;

   if ( tex != NULL )
            tOffset &= tex -> width - 1;

   if ( topLine [col] > bottomLine [col] )
      continue;

     if ( top < topLine [col] )
           top = topLine [col];          // adjust top

     if ( bottom < topLine [col] )
           bottom = topLine [col];

     if ( bottom <= bottomLine [col] )      // draw floor
           drawVertLine ( col, bottom, bottomLine [col]++,
                       FLOOR_COLOR );
```

```
            else
                bottom = bottomLine [col];

            if ( top < bottom )                    // now draw visible
            {                           // part of wall
                texY -= (newTop-top)*dTexY;

                if ( tex != NULL )
                    drawTexturedLine ( col, top, bottom, tex,
                                tOffset, texY, dTexY );
                else
                    drawVertLine ( col, top, bottom,
                                LOWER_WALL_COLOR );

                if ( top < bottomLine [col] )
                    bottomLine [col] = top - 1;
            }
            else
                if ( bottom < bottomLine [col] )
                    bottomLine [col] = bottom - 1;

            if ( topLine [col] > bottomLine [col] )
            {
                colsLeft--;
                colFilled [col] = 0xFF;
            }
        }
    }
void   drawUpperWall ( Texture * tex, SideDef * side, int col1, int col2,
                int bottomHeight, int topHeight,
                Fixed tOffs1, Fixed tOffs2, int v1, int v2 )
{
    Fixed  b1      = int2Fixed ( 100l - ( bottomHeight * VSCALE ) / v1 );
    Fixed  b2      = int2Fixed ( 100l - ( bottomHeight * VSCALE ) / v2 );
    Fixed  t1      = int2Fixed ( 100l - ( topHeight * VSCALE ) / v1 );
    Fixed  t2      = int2Fixed ( 100l - ( topHeight * VSCALE ) / v2 );
    Fixed  db      = ( b2 - b1 ) / ( col2 - col1 );
    Fixed  dt      = ( t2 - t1 ) / ( col2 - col1 );
    Fixed  invV1   = (ONE<<6) / v1;
    Fixed  invV2   = (ONE<<6) / v2;
    Fixed  tInvV1  = (tOffs1<<6) / v1;
    Fixed  tInvV2  = (tOffs2<<6) / v2;
    Fixed  dInvV   = (invV2 - invV1) / (col2 - col1);
    Fixed  dtInvV  = (tInvV2 - tInvV1 ) / (col2 - col1);
    Fixed  invV    = invV1;
    Fixed  tInvV   = tInvV1;
    Fixed  texY;
    Fixed  dTexY;
    int    top, bottom;
    int    newTop;
    int    tOffset;

    if ( col1 < 0 )        // clip invisible part
    {
```

```
        b1    -= col1 * db;
        t1    -= col1 * dt;
        invV  -= col1 * dInvV;
        tInvV -= col1 * dtInvV;
        col1   = 0;
}
if ( col2 > 320 )
        col2 = 320;
for ( int col = col1; col < col2; col++ )
{
    top         = fixed2Int ( t1 );
    bottom      = fixed2Int ( b1 );
    newTop      = top;
    dTexY       = int2Fixed (bottomHeight - topHeight + 1) / (bottom - top + 1);
    texY        = int2Fixed ( side -> yOffset ) - (bottom - top + 1) * dTexY;
  tOffset       = tInvV / invV;
    t1          += dt;
    b1          += db;
    invV        += dInvV;
    tInvV       += dtInvV;
      if ( tex != NULL )
            tOffset &= tex -> width - 1;
      if ( topLine [col] > bottomLine [col] )
            continue;
if ( top > bottomLine [col] )
      top = bottomLine [col];
if ( top >= topLine [col] )           // draw ceiling
      drawVertLine (col,topLine [col]++,top,CEILING_COLOR);
else                                  // otherwise correct to
      top = topLine [col];            // draw only visible part
if ( bottom > bottomLine [col] )
      bottom = bottomLine [col];
if ( top < bottom )                   // now draw visible
{                           // part of wall
      texY -= (newTop-top)*dTexY;
      if ( tex != NULL )
            drawTexturedLine ( col, top, bottom, tex,
                        tOffset, texY, dTexY );
      else
            drawVertLine ( col, top, bottom,
                        UPPER_WALL_COLOR );
      if ( bottom > topLine [col] )
            topLine [col] = bottom + 1;
}
else
      if ( top > topLine [col] )
            topLine [col] = top + 1;
if ( topLine [col] > bottomLine [col] )
```

```
            {
                colsLeft--;
                colFilled [col] = 0xFF;
            }
        }
    }
```

Формат WAD-файла можно найти по адресу:
www.gamers.org/dEngine/doom/spec/uds.666.txt

13.4. Descent

Игра Descent, появившаяся спустя год после выхода игры DOOM, является полностью трехмерной - действие происходит в трехмерном лабиринте, причем коридоры этого лабиринта находятся под всевозможными углами друг к другу и не лежат в одной плоскости. Фактически в ходе игры понятия пола и потолка (а также правая и левая стена) постоянно меняются местами.

Лабиринт в этой игре представлен в виде набора полигональных фрагментов. Каждый такой фрагмент является либо комнатой, либо деформированным кубом. Для удаления невидимых поверхностей используется метод порталов, великолепно работающий в системе длинных коридоров. Все грани текстурируются и применяется достаточно мощная схема освещения. Освещенность непрерывно интерполируется внутри грани. При этом помимо стандартной освещенности ряд объектов (например, ракеты) также выступают в качестве источников света. Для повышения скорости рендеринга достаточно удаленные грани не текстурируются, а закрашиваются постоянным цветом. Основные объекты представлены в виде полигональных моделей и для каждой из них также строится свое BSP-дерево. Достаточно удаленные объекты изображаются спрайтами.

Подобный подход позволил получить крайне динамические трехмерные сцены, состоящие из полигональных объектов.

Для рендеринга каждый объект как точка сравнивается со всеми разбивающими плоскостями и в нужном месте выводится как независимое BSP-дерево.

13.5. Текстурирование в общем случае

Рассмотрим, каким образом можно осуществить текстурирование произвольно ориентированной грани при перспективном проектировании (рис. 13.24).

Пусть грань P задана в пространстве набором своих вершин P_1, P_2, P_3, P_4. Будем считать ее прямоугольной и определим вектор a как P_1, а также два боковых вектора e_1 и e_2.

$e_1 = P_2 - P_1$ $e_2 = P_4 - P_1$

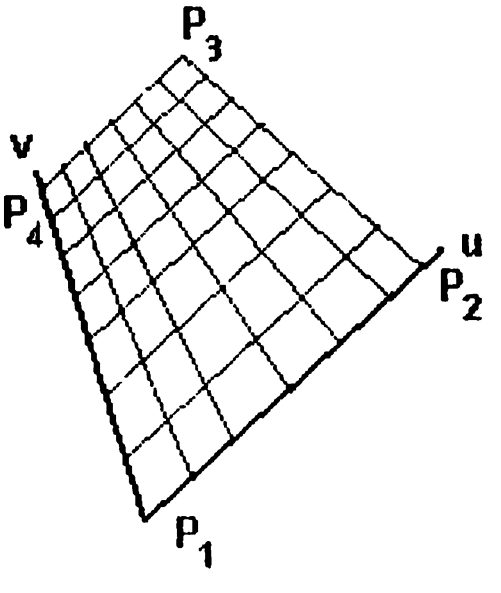


Рис. 13.24

Нормаль n к грани определяется из соотношения $n = [e_1, e_2]$.

Будем считать, что перспективное преобразование осуществляется по формулам

$$X = \frac{x}{z}, \quad Y = \frac{y}{z}.$$

т. е. в данном случае z соответствует глубине.

Найдем образ произвольной точки грани $p = a + ue_1 + ve_2$, где u и v - параметры текстуры. После несложных преобразований получаем:

$$X = \frac{a_x + ue_{1x} + ve_{2x}}{a_z + ue_{1z} + ve_{2z}},$$

$$Y = \frac{a_y + ue_{1y} + ve_{2y}}{a_z + ue_{1z} + ve_{2z}}.$$

Однако для текстурирования грани удобнее использовать обратное отображение. Найдем прообраз (u, v) произвольной точки (X, Y) экрана. Для этого разрешим предыдущие соотношения относительно u и v. Умножив их на знаменатель дроби, получим систему линейных уравнений относительно u и v:

$$\begin{cases} u(Xe_{1z} - e_{1x}) + v(Xe_{2z} - e_{2x}) = a_x - a_z X \\ u(Ye_{1z} - e_{1y}) + v(Ye_{2z} - e_{2y}) = a_y - a_z Y \end{cases}$$

Применим для ее решения правило Крамера:

$$u = \frac{\Delta_u}{\Delta}, \quad v = \frac{\Delta_v}{\Delta},$$

где $\Delta = Xn_x + Yn_y + n_z$

$\Delta_u = Xm_x + Ym_y + m_z$

$\Delta_v = Xl_x + Yl_y + l_z$

Вспомогательные векторы l и m определяются следующими векторными произведениями:

$m = [e_2, a]$, $l = [a, e_1]$.

Эти формулы можно переписать с использованием однородных координат в виде

$$\begin{pmatrix} u\Delta \\ v\Delta \\ \Delta \end{pmatrix} = \begin{pmatrix} m_x & m_y & m_z \\ l_x & l_y & l_z \\ n_x & n_y & n_z \end{pmatrix} \begin{pmatrix} X \\ Y \\ 1 \end{pmatrix}$$

В общем случае перспективное проектирование задается формулой

$$\begin{pmatrix} Xh \\ Yh \\ h \end{pmatrix} = P \begin{pmatrix} x \\ y \\ z \end{pmatrix},$$

где P - невырожденная матрица; например следующему преобразованию:

$$X = \frac{k_1 x}{z} + b_1,$$

$$Y = \frac{k_2 y}{z} + b_2.$$

соответствует матрица

$$\begin{pmatrix} k_1 & 0 & b_1 \\ 0 & k_2 & b_2 \\ 0 & 0 & 1 \end{pmatrix}.$$

Для определения параметров текстуры можно воспользоваться такой формулой:

$$\begin{pmatrix} u\Delta \\ v\Delta \\ \Delta \end{pmatrix} = TP^{-1} \begin{pmatrix} X \\ Y \\ 1 \end{pmatrix}$$

где T - матрица, составленная из векторов m, l и n.

Величины u и v определяют точку текстуры в единичном квадрате, т. е. вершине P_1 соответствует $u - 0$, $v = 0$, вершине $P_2 - u = 1$, $v = 0$, вершине $P_3 - u = 1$, $v = 1$ и вершине $P_4 - u = 0$, $v = 1$. Чтобы получить индексы для текстуры, например размера 64 на 64, следует взять [64*u] и [64*v].

В результате мы приходим к следующей схеме текстурирования:

```
for ( row = row1; row <= row2; row++ )
    for ( col = col1; col <= col2; col++ )
    {
        d  = n.z + n.y * row + n.x * col;
        du = m.z + m.y * row + m.x * col;
        dv  = l.z + l.y * row + l.x * col;
        u  = 64 * du / d;
        v  = 64 * dv / d;
        putpixel ( col, row, textureMap [v][u] );
    }
```

Эту процедуру, вынося наиболее сложные части за пределы внутреннего цикла, можно несколько оптимизировать.

```
for ( row = row1; row <= row2; row++ )
{
    d  = n.z+n.y*row+n.x*col1;
    du = 64 * (m.z+m.y*row+m.x*col1);
    dv  = 64 * (l.z+l.y*row+l.x*col1);

    for ( col = col1; col <= col2; col++ )
    {
        u  = du / d;
        v  = dv / d;
        putpixel ( col, row, textureMap [v][u] );
        d  += n.x;
        du += m.x;
```

```
            dv  += l.x;
        }
    }
```

Можно вместо явного вычисления соответствующих векторов и матриц в каждой вершине грани вычислить величины $u' = u/z$, $v' = v/z$ и $w = 1/z$, которые изменяются линейно в пространстве картинной плоскости. Докажем это.

Плоскость, в которой лежит грань P, можно описать следующим уравнением: $xn_x + yn_y + zn_z = (a, n)$.

Разделим это уравнение на z и выразим отсюда $1/z$:

$$\frac{1}{z} = \frac{\left(Xn_x + Yn_y + n_z\right)}{(a,n)} = \frac{\Delta}{(a,n)}$$

Аналогично найдем u/z и v/z:

$$\frac{u}{z} = u\frac{1}{z} = \frac{\Delta_u}{\Delta}\frac{\Delta}{(a,n)} = X\frac{m_x}{(a,n)} + Y\frac{m_y}{(a,n)} + \frac{m_z}{(a,n)},$$

$$\frac{v}{z} = \frac{\Delta_v}{\Delta}\frac{\Delta}{(a,n)} = X\frac{l_x}{(a,n)} + Y\frac{l_y}{(a,n)} + \frac{l_z}{(a,n)}$$

Тогда эти величины линейно интерполируются для каждого пиксела и для вычисления точных параметров текстуры используются следующие формулы: $u = u'/w$ и $v = v'/w$.

Тем не менее все равно потребуется два дорогостоящих деления на каждый пиксел. Рассмотрим несколько частных случаев.

1. Грань вертикальная (рис. 13.25). В этом случае $n_z = 0$ и определитель Δ не зависит от Y. Тем самым если поменять циклы местами (вынести цикл по столбцам наружу) и вынести деление за пределы внутреннего цикла, то делений практически не потребуется. Фактически это означает вывод грани по столбцам, что и делалось ранее.
2. Грань горизонтальная (рис. 13.26). Тогда $n_x = n_y = 0$ и определитель Δ не зависит от X. В этом случае вывод пикселов следует осуществлять по строкам; как и ранее, потребуется всего несколько делений на строку.

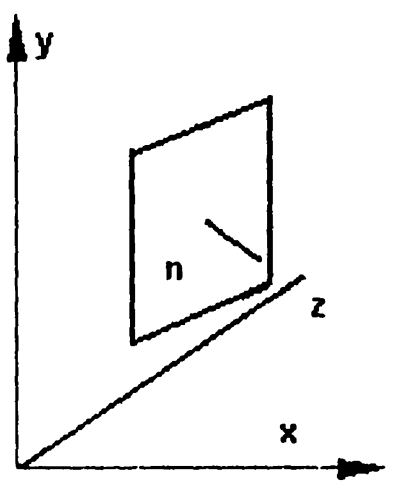


Рис. 13.25

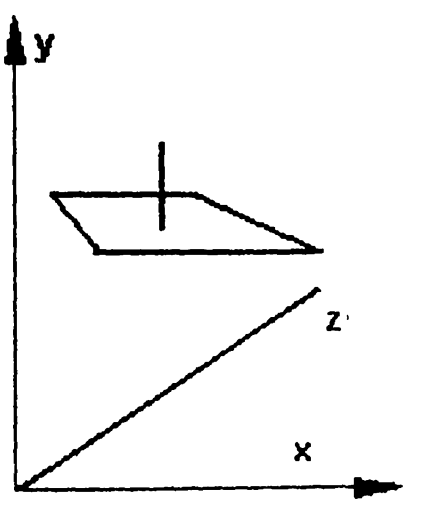


Рис. 13.26

Попытаемся перенести этот подход на общий случай. Для этого необходимо сгруппировать пикселы таким образом, чтобы для всех пикселов каждой группы величина Δ была постоянной.

Это приводит к уравнению $Xn_x + Yn_y + n_z = \text{const}$, т. е. каждая построенная группа пикселов является отрезком прямой картинной плоскости с нормалью (n_x, n_y). Несложно заметить, что все точки грани, проектирующиеся на этот отрезок, обладают одинаковой глубиной, т. е. являются пересечением грани с плоскостью, параллельной экрану.

В результате мы приходим к следующему алгоритму: проекция грани разбивается на набор параллельных отрезков с заданным угловым коэффициентом и по этим отрезкам рисуется.

Мы опять получаем минимум операций деления и умножения на пиксел, но за это приходится расплачиваться заметно более сложной схемой рисования.

Кроме того, в отличие от вертикальных и горизонтальных граней, когда перебирались пикселы, действительно лежащие на линии постоянства глубины, в данном случае перебираемые пикселы лежат на растровой развертке соответствующей линии, что приводит к заметным искажениям.

Описанный метод называется методом постоянной глубины (constant-z).

Еще одним точным методом текстурирования является гиперболический метод. Запишем выражение для u в зависимости от x в пределах одной строки:

$$u(x) = \frac{ax + b}{cx + d}$$

Правильное значение u удовлетворяет следующему уравнению: $f(x,u) = 0$, где $f(x,u) = ax + b - u(cx + d)$.

Уравнение $f(x, u) = 0$ описывает гиперболу на плоскости переменных u и x. Поскольку величина x изменяется каждый раз на единицу, то для вычисления соответствующего значения u можно воспользоваться аналогом алгоритма Брезенхейма для построения гиперболы. Этот алгоритм обеспечивает достаточно высокое качество, но довольно сложен в реализации.

В ряде случаев прибегают к использованию интерполяционных текстур, когда вместо точных значений u и v для каждого пиксела экрана находятся их приближенные значения.

Существует довольно большое количество различных схем интерполяции, отличающихся как по затратам, так и по качеству получаемого изображения.

Простейшим вариантом интерполяции является билинейная интерполяция - аналог закрашивания Гуро, когда параметры текстуры сначала линейно интерполируются вдоль ребер, а затем - по горизонтальным линиям.

Получающееся при этом изображение приведено на рис. 13.27.

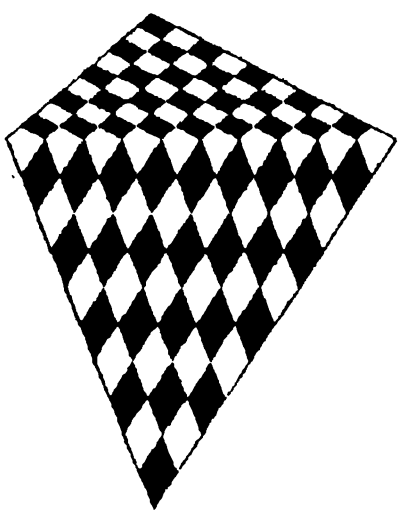


Рис. 13.27

Сразу бросается в глаза сильное искажение изображения вблизи линии, соединяющей противоположные вершины.

Это типичная ошибка закрашивания Гуро; отметим, что при интерполяции освещенности эти ошибки гораздо менее заметны, чем при интерполировании текстуры, так как интенсивность обычно является гладкой, медленно изменяющейся величиной, тогда как текстура часто имеет сильные скачки и разрывы. Наиболее сильно искажения заметны для текстур с ярко выраженной регулярной структурой.

Далее мы подробнее рассмотрим несколько интерполяционных методов в простейшем примере: построении горизонтальной строки пикселов, соответствующей некоторой грани (рис. 13.28).

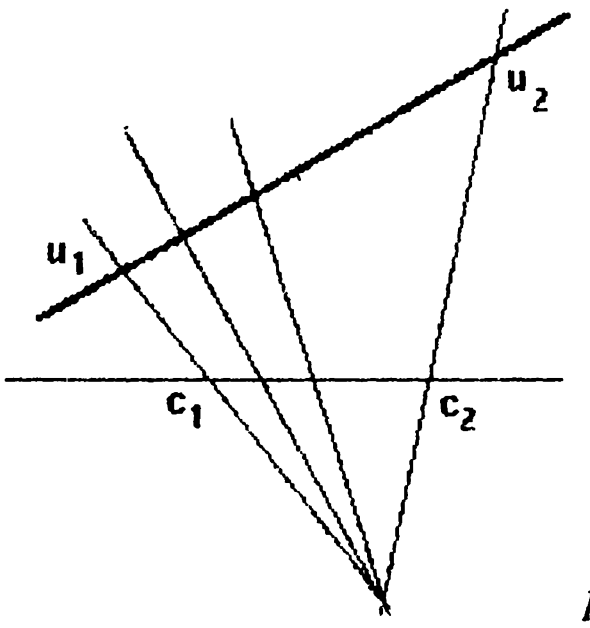


Рис. 13.28

Будем считать, что концам отрезка соответствуют столбцы x_1 и x_2 со значениями параметра текстурирования u_1 и u_2 соответственно.

Простейшим вариантом является линейная интерполяция

$u(x) = a_0 + a_1 x,$

где значения параметров a_0 и a_1 легко определяются из соотношений

$u(x_1) = u_1,, u(x_2) = u_2.$

Несмотря на получающиеся при интерполяции ошибки, данный метод в целом ряде случаев оказывается вполне приемлемым.

Следующим возможным вариантом интерполяции является квадратичная: $u(x) = a_0 + a_1 x + a_2 x^2$

Значение параметра u_3 обычно вычисляется в некоторой промежуточной точке, например в точке $x_3 = \dfrac{x_1 + x_2}{2}$

Если точкам x_1 и x_2 соответствуют значения $\Delta_1, \Delta_{u1}, \Delta_2, \Delta_{u2}$, то промежуточной точке x_3 соответствует значение

$$u_3 = \frac{\Delta_{u1} + \Delta_{u2}}{\Delta_1 + \Delta_2}.$$

Тогда значения параметров a_0, a_1 и a_2 находятся из соотношений

$$a_2 = \frac{2k}{(x_2 - x_1)^2},$$

$$a_1 = \frac{u_2 - u_1}{x_2 - x_1} - \frac{2k}{(x_2 - x_1)^2}(x_2 + x_1),$$

$$a_0 = u_1 - a_1 x_1 - a_2 x_1^2,$$

где $k = u_1 + u_2 - 2u_3$.

Поскольку нас интересует только набор значений u, то для облегчения расчетов можно произвести нормирование координат и считать, что $x_0 = 0$, $x_1 = 1$ и x изменяется с шагом h. Тогда получаются заметно более простые уравнения для коэффициентов:

$a_0 = u_1$,

$a_1 = -3u_1 - u_2 + 4u_3$,

$a_2 = 2(u_1 + u_2 - 2u_3)$.

При этом для вычисления очередного значения u к операции умножения можно и не прибегать, достаточно воспользоваться рекуррентными соотношениями

$$u(x_1 + ih) = u(x_1 + (i-1)h) + d_i^1,$$

$$d_i^1 = d_{i-1}^1 + 2a_2h^2,$$

$$d_0^1 = a_1h + a_2hx_1.$$

Обратите внимание, что $d_0^1 = hu'(x_1)$.

Для построения текстурированного изображения четырехугольника можно воспользоваться следующей функцией:

```
void        quadratic ()
{
    float    dx01 = (float)(pr[1].x - pr[0].x)/(float)(pr[1].y - pr[0].y);
    float    dx12 = (float)(pr[2].x - pr[1].x)/(float)(pr[2].y - pr[1].y);
    float    dx03 = (float)(pr[3].x - pr[0].x)/(float)(pr[3].y - pr[0].y);
    float    dx32 = (float)(pr[2].x - pr[3].x)/(float)(pr[2].y - pr[3].y);
float    x1    = pr [0].x;
float    x2    = pr [0].x;

for ( int y = pr [0].y; y < pr [2].y; y++ )
{
    float    du1 = t.x [0][0] * x1 + t.x [0][1] * y + t.x [0][2];
    float    dv1 = t.x [1][0] * x1 + t.x [1][1] * y + t.x [1][2];
    float    d1  = t.x [2][0] * x1 + t.x [2][1] * y + t.x [2][2];
    float    du2 = t.x [0][0] * x2 + t.x [0][1] * y + t.x [0][2];
    float    dv2 = t.x [1][0] * x2 + t.x [1][1] * y + t.x [1][2];
    float    d2  = t.x [2][0] * x2 + t.x [2][1] * y + t.x [2][2];
    float    u1  = du1 / d1;
    float    u2  = du2 / d2;
    float    v1   = dv1 / d1;
    float    v2   = dv2 / d2;
    float    um  = ( du1 + du2 ) / ( d1 + d2 );
    float    vm  = ( dv1 + dv2 ) / ( d1 + d2 );

    float    invDx = x2 > x1 ? 1.0 / ( x2 - x1 ) : 0.0;
    float    a0 = u1;
    float    a1 = (-3*u1 + 4*um - u2 ) * invDx;
    float    a2 = 2*( u1 - 2*um + u2 ) * invDx * invDx;

    float    b0 = v1;
    float    b1 = (-3*v1 + 4*vm - v2 ) * invDx;
    float    b2 = 2*( v1 - 2*vm + v2 ) * invDx * invDx;

    float    deltaU1 = a1 + a2;
    float    deltaU2 = 2 * a2;
```

```
        float    deltaV1 = b1 + b2;
        float    deltaV2 = 2 * b2;

        float    u = u1;
        float    v = v1;

        for ( int x = x1; x < x2; x++ )
        {
            puttexel ( x, y, u, v );
            u          += deltaU1;
            v          += deltaV1;
            deltaU1 += deltaU2;
            deltaV1 += deltaV2;
        }
        if ( y < pr [1].y )
            x1 += dx01;
        else
            x1 += dx12;
        if ( y < pr [3].y )
            x2 += dx03;
        else
            x2 += dx32;
    }
}
```

Квадратичная интерполяция сложнее линейной, но она обеспечивает замет более высокое качество.

Ниже приводятся изображения, получаемые в процессе применения нескольк методов интерполяции.

В качестве текстуры применяется стандартная клеточная текстура.

На рис. 13.29 приведено изображение, получающееся при корректном проектировании грани.

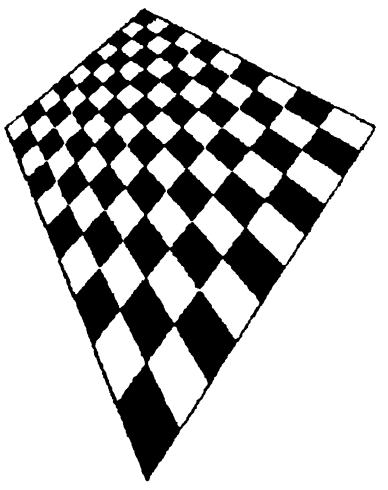


Рис. 13.29

Более точным является подход, при котором грань рисуется по строкам (столбцам) с применением линейной интерполяции внутри каждой строки, но значения параметров для каждой строки вычисляются точно.

Получающееся при этом изображение приведено на рис. 13.30.

Более высокой точности можно добиться, применяя квадратичную интерполяцию. Получающееся при этом изображение приведено на рис. 13.31.

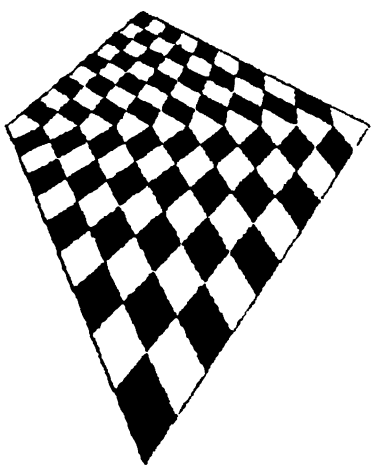


Рис. 13.30

Можно использовать и кубическую интерполяцию:

$$u = a_0 + a_1 x + a_2 x^2 + a_3 x^3,$$

где вместо задания значений в промежуточных точках обычно задаются значения производных в конечных точках

$u(x_1) = u_1,$
$u'(x_1) = u'_1,$
$u(x_2) = u_2,$
$u'(x_2) = u'_2.$

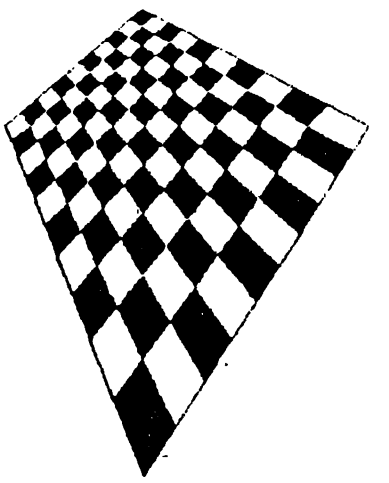


Рис. 13.3

Для вычисления значений производных можно воспользоваться следующими соотношениями:

$$u'(x) = \frac{d}{dx}\frac{\Delta_u}{\Delta} = \frac{y(m_x n_y - n_x m_y) + m_x n_z - n_x m_z}{\Delta^2},$$

$$u'_1 = \frac{(\Delta_{u2} - \Delta_{u1})\Delta_1 - (\Delta_2 - \Delta_1)\Delta_{u1}}{\Delta_1^2}$$

$$u'_2 = \frac{(\Delta_{u2} - \Delta_{u1})\Delta_2 - (\Delta_2 - \Delta_1)\Delta_{u2}}{\Delta_2^2}$$

Так как $\Delta_2 - \Delta_1 = n_x$, $\Delta_{u2} - \Delta_{u1} = m_x$

Полагая, как и ранее, что аргумент x изменяется на отрезке [0, 1], получаем дующие формулы для интерполяционных коэффициентов:

$a_0 = u_1,$
$a_1 = u'_1,$
$a_2 = 3(u_2 - u_1) - 2u'_1 - u'_2,$
$a_3 = -2(u_2 - u_1) + u'_1 + u'_2.$

Для расчета текстуры вдоль строки пикселов можно воспользоваться сле щим фрагментом программы:

```
void        cubic ()
{
    float    dx01 = (float)(pr[1].x - pr[0].x)/(float)(pr[1].y - pr[0].y);
    float    dx12 = (float)(pr[2].x - pr[1].x)/(float)(pr[2].y - pr[1].y);
    float    dx03 = (float)(pr[3].x - pr[0].x)/(float)(pr[3].y - pr[0].y);
    float    dx32 = (float)(pr[2].x - pr[3].x)/(float)(pr[2].y - pr[3].y);
    float    x1   = pr [0].x;
    float    x2   = pr [0].x;

    for ( int y = pr [0].y; y < pr [2].y; y++ )
    {
        float    du1 = t.x [0][0] * x1 + t.x [0][1] * y + t.x [0][2];
        float    dv1 = t.x [1][0] * x1 + t.x [1][1] * y + t.x [1][2];
        float    d1  = t.x [2][0] * x1 + t.x [2][1] * y + t.x [2][2];
        float    du2 = t.x [0][0] * x2 + t.x [0][1] * y + t.x [0][2];
        float    dv2 = t.x [1][0] * x2 + t.x [1][1] * y + t.x [1][2];
        float    d2  = t.x [2][0] * x2 + t.x [2][1] * y + t.x [2][2];
        float    invDx = x2 > x1 ? 1.0 / ( x2 - x1 )   0.0;
        float    tt1 = ( du2 * d1 - du1 * d2 ) * invDx;
        float    tt2 = ( dv2 * d1 - dv1 * d2 ) * invDx;
```

```
float    tt3 = tt1 / ( d2 * d2 );
float    tt4 = tt2 / ( d2 * d2 );
float    du  = tt1 / ( d1 * d2 );
float    dv  = tt2 / ( d1 * d2 );
float    a0  = du1 / d1;
float    a1  = tt1 / ( d1 * d1 );
float    a2  = ( 3 * du - 2 * a1 - tt3 ) * invDx;
float    a3  = ( -2 * du + a1 + tt3 ) * invDx * invDx;
float    b0  = dv1 / d1;
float    b1  = tt2 / ( d1 * d1 );
float    b2  = ( 3 * dv - 2 * b1 - tt4 ) * invDx;
float    b3  = ( -2 * dv + b1 + tt4 ) * invDx * invDx;

float    deltaU1 = a1 + a2 + a3;
float    deltaU2 = 6 * a3 + 2 * a2;
float    deltaU3 = 6 * a3;
float    deltaV1 = b1 + b2 + b3;
float    deltaV2 = 6 * b3 + 2 * b2;
float    deltaV3 = 6 * b3;
float    u       = a0;
float    v       = b0;

for ( int x = x1; x < x2; x++ )
{
    puttexel ( x, y, u, v );

    u       += deltaU1;
    v       += deltaV1;
            deltaU1 += deltaU2;
            deltaV1 += deltaV2;
            deltaU2 += deltaU3;
            deltaV2 += deltaV3;
        }
        if ( y < pr [1].y )
            x1 += dx01;
        else
            x1 += dx12;

        if ( y < pr [3].y )
            x2 += dx03;
        else
            x2 += dx32;
    }
}
```

При этом получается изображение, как на рис. 13.32.

Поскольку значения параметров текстур на ребрах задаются точно, то искажения происходят только внутри граней и проблем со стыковкой граней между собой не возникает.

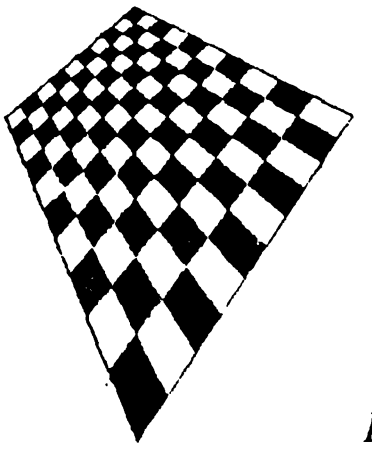


Рис. 13.32

Возможно и дальнейшее повышение порядка интерполяции, однако при этом заметно возрастает сложность вычисления необходимых коэффициентов, а получающийся выигрыш оказывается небольшим.

Искажения связаны с ориентацией грани и расстоянием до нее.

Интерполяция может проходить как по столбцам, так и по строкам.

Для построения достаточно качественного изображения удобно применять адаптивные схемы, которые в зависимости от ориентации грани и расстояния до нее выбирают наиболее оптимальный вариант - рисование по строкам или столбцам, вид интерполяции.

Повысить качество интерполяции можно за счет введения промежуточных точек, для чего внутри требуемого интервала выбирается несколько вспомогательных точек и вычисляются значения параметров текстуры в этих точках. После этого параметры интерполируются уже внутри получившихся частичных интервалов, обеспечивая повышение качества интерполяции. Таким образом можно, например, вычислять точное значение текстуры через каждые 8-16 пикселов и применять интерполяцию в промежуточных точках.

Поскольку в большинстве случаев вертикальные и горизонтальные стены составляют большинство стен, то для них имеет смысл применять точные методы.

Пример программы, реализующей основные методы текстурирования (точное, аффинное, линейное, квадратическое и кубическое) можно найти на прилагающемся к книге компакт-диске.

Еще одним возможным вариантом разбиения, дающим очень высокую скорость, является разбиение исходного многоугольника на небольшие фрагменты (например, треугольники) и использование аффинного текстурирования для каждого из получившихся фрагментов.

Дополнительную информацию по текстурированию можно найти в Интернете по следующим адресам:

www.geocities.com/SiliconValley/2151/tmap.html
www.geocities.com/SiliconValley/Park/9784/perspect.html
www.unm.edu/~strider7/texture_mapping.html

13.6. Пирамидальное фильтрование (mipmapping)

Обратите внимание на вид текстурированной поверхности, находящейся на большом расстоянии, когда коэффициент сжатия заметно больше единицы (например, в программе текстурирования плоскости на дальнюю часть плоскости).

В этом случае мы видим фактически случайный набор точек, имеющий мало общего с тем, что мы хотели бы увидеть. Это связано с тем, что здесь одному пикселу экрана соответствует сразу несколько пикселов текстуры, поэтому для корректного отображения данного пиксела его цвет должен быть усреднением цветов этих пикселов (вместо этого выбирается практически случайным образом один из этих пикселов текстуры).

Ясно, что нереально проводить точное усреднение пикселов в реальном времени практически: это потребовало бы слишком больших затрат. Однако существует простой метод, позволяющий бороться с подобными ошибками сравнительно легко.

Пусть исходное изображение имеет размер 2^n на 2^n пикселов. Построим по нему изображение 2^{n-1} на 2^{n-1} пикселов следующим образом: объединим пикселы исходного изображения в группы по 4 (2 на 2), усредним их цвета и назначим получившийся цвет (ближайшее к нему значение из палитры) соответствующей точке нового изображения. Затем по полученному изображению тем же методом построим изображение размеров 2^{n-2} на 2^{n-2}

Путем последовательных применений описанной процедуры мы приходим к серии изображений, получающихся из исходного сжатием в 2, 4, 8 и т. д. раз. Размер последнего изображения будет 1 на 1 пиксел. Дальнейшее сжатие проводить бессмысленно.

В результате получается пирамида, где на 0-м (нижнем) уровне находится исходное изображение, а каждый следующий уровень получается из предыдущего сжатием в 2x2 раза.

```
// File Pyramid.h
//
// class ImagePyramid realizes mip-mapping
// for a  given image

#ifndef __PYRAMID__
#define __PYRAMID__

#include    "bmp.h"

#define MAX_LEVELS  10        // maximum # of levels in pyramid

class    ImagePyramid
{
public:
    char  *  data;
    RGB  * palette;
    int      palSize;
    int      width;
    int      offs [MAX_LEVELS];

    ImagePyramid ( BMPImage * );
    ~ImagePyramid ();
};

int  findClosestEntry ( int, int, int, RGB *, int );
#endif
```

```
// File Pyramid.cpp
//
// class ImagePyramid realizes mip-mapping
// for a  given image

#include    <alloc.h>
#include    <mem.h>
#include    <stdlib.h>
#include    "pyramid.h"

ImagePyramid    ImagePyramid ( BMPImage * im )
{
    width    = im -> width;
```

```
palSize  = 256;
data     = (char *) malloc ( ( 4 * width * width ) / 3 );
palette  = new RGB [palSize];
offs [0] = 0;
                // copy palette
memcpy ( palette, im -> palette, palSize * sizeof ( RGB ) );

                // copy 0-th layer
memcpy ( data, im -> data, width * width );

for ( int level = 1, w = width; w > 0; level ++ )
{
    char  * prev = data + offs [level - 1];
    char  * cur  = prev + w*w;

    offs [level] = offs [level - 1] + w*w;
    w          /= 2;

    for ( int i = 0; i < w; i++ )
        for ( int j = 0; j < w; j++ )
        {
            int  index = 2*i + 2*j*2*w;
            int  r, g, b;

            r = ( (int)palette [prev [index]].red +
                (int)palette [prev [index+1]].red +
                 (int)palette [prev [index+2*w]].red +
                (int)palette [prev [index+2*w+1]].red ) >> 2;

            g = ( (int)palette [prev [index]].green +
                (int)palette [prev [index+1]].green +
                 (int)palette [prev [index+2*w]].green +
                (int)palette [prev [index+2*w+1]].green ) >> 2;

            b = ( (int)palette [prev [index]].blue +
                (int)palette [prev [index+1]].blue +
                 (int)palette [prev [index+2*w]].blue +
                (int)palette [prev [index+2*w+1]].blue ) >> 2;

            cur [i+j*w] = findClosestEntry ( r, g, b, palette,
                                             palSize );
        }
    }
}
ImagePyramid   ~ImagePyramid ()
{
    free ( data );
    delete palette;
}
int  findClosestEntry ( int r, int g, int b, RGB * palette, int palSize )
{
    int  minDist = 1024;
    int  index;

    for ( int i = 0; i < palSize; i++ )
    {
        int  d = abs (r - palette [i].red) + abs (g - palette [i].green) +
```

```
                abs ( b - palette [i].blue );
            if ( d < minDist )
            {
                minDist = d;
                index   = i;
            }
        }
        return index;
}
```

Рассмотрим применение пирамид для текстурирования.

Предположим, что нужно найти пиксел из изображения, соответствующего сжатию в k раз. Если k больше размера изображения, то возьмем пиксел с изображения **в вершине пирамиды - размером 1 на 1 пиксел. В противном случае возьмем пиксел** с такого уровня i, что $2^i \le k < 2^{i+1}$

Полученный пиксел соответствует усреднению исходной текстуры в $2^{[\log_2 k]}$ раз.

Ниже приводится модификация программы текстурирования горизонтальной плоскости с использованием пирамидального фильтрования. Сразу бросается в глаза отсутствие случайных точек вблизи линии горизонта.

При этом для хранения всей пирамиды требуется всего на 1/3 больше памяти, чем для хранения только исходного изображения.

```
// File Floor3.cpp
//
// Drawing textured floor
// with pyramid filtering (mip-mapping)
#include    <alloc.h>
#include    <bios.h>
#include    <dos.h>
#include    <math.h>
#include    <mem.h>
#include    <stdlib.h>
#include    <stdio.h>
#include    <time.h>
#include    "bmp.h"
#include    "pyramid.h"
#include    "fixmath.h"

#define     ESC         0x011b
#define     UP          0x4800
#define     DOWN        0x5000
#define     LEFT        0x4b00
#define     RIGHT       0x4d00

#define     SCREEN_HEIGHT 200
#define     SCREEN_WIDTH  320

#define     H           (100*256l)
#define     DELTA       1.0             // note   DELTA * NumLines == H
#define     C           0.01
#define     D0          100.0
```

```
#define MAKELONG(high, low) (((long)low) | (((long)high)<<16))

BMPImage     *  pic            = new BMPImage ( "FLOOR.BMP" );
BMPImage     *  sky            = new BMPImage ( "SKY.BMP" );
ImagePyramid *  pyr            = new ImagePyramid ( pic );
char far     *  screenPtr      = (char far *) MK_FP ( 0xA000, 0 );
long            totalFrames    = 0;

Fixed   distTable [SCREEN_WIDTH/2];
Fixed   CSinTable [1024];
Fixed   CCosTable [1024];

Fixed   locX, locY;            // viewer loc
Angle   angle;                 // viewer angles

int  getShift ( int val )
{
    for ( int s = 0; val != 0; s++, val >>= 1 )


    return s - 1;
}

void  drawSky ()
{
    char far * videoPtr   = screenPtr;
    int          angleShift = angle >> 7;
    char     * skyPic      = sky -> data;

    for ( int row = 0; row < 100; row++, skyPic += sky -> width )
    {
        for ( int col = 0; col < 320; col++ )
            *videoPtr++ = skyPic [ (col - angleShift) & (sky->width-1)];
    }
}

void      drawView ()
{
    char far * videoPtr    = screenPtr + 100*320;
    long     widthMask   = MAKELONG ( pic -> width - 1, 0xFFFF );
    long      heightMask = MAKELONG ( pic -> height - 1, 0xFFFF );
    char    * picData      = pic -> data;

    totalFrames++;

    drawSky ();

    for (int row = 0; row < 100; row++ )
    {
        Fixed   dist          = ( H * distTable [row] ) >> 8;
        Fixed   u0            = ( locX + dist * ( cosine ( angle ) >> 8 ) ) & widthMask
        Fixed   v0            = ( locY + dist * ( sine ( angle ) >> 8 ) ) & heightMask;
        Fixed   du            = ( dist * CSinTable [angle >> 6] ) >> 8;
        Fixed   dv            = ( -dist * CCosTable [angle >> 6] ) >> 8;
        long    widthMask  = MAKELONG ( pic -> width - 1, 0xFFFF );
        long    heightMask = MAKELONG ( pic -> height - 1, 0xFFFF );
        int     widthShift   = getShift ( pic -> width );

        in      scale         = dist / (256l*100 );  // Dist * C
```

```
int       level          = 0;      // start with level 0

while ( scale / 2 > 1 )
{
     u0        /= 2;
     v0        /= 2;
     du        /= 2;
     dv        /= 2;
     scale      /= 2;
     widthMask  /= 2;
     heightMask /= 2;
     widthShift --;
     level ++;
}

Fixed    u    = u0;
Fixed    v    = v0;

picData   = pyr -> data + pyr -> offs [level];
videoPtr += 160;

for ( int col = 159; col >= 0; col-- )
{
     * videoPtr-- = picData [fixed2Int ( u ) + (fixed2Int ( v ) << widthShift )];

     u = ( u - du ) & widthMask;
     v = ( v - dv ) & heightMask;
}
          videoPtr += 160;

          for ( col = 160, u = u0, v = v0; col < 320; col++ )
          {
               u = ( u + du ) & widthMask;
               v = ( v + dv ) & heightMask;

               * videoPtr++ = picData [ fixed2Int ( u ) + (fixed2Int ( v ) << widthShift)];
          }
     }
}

void       setVideoMode ( int mode )
{
     asm {
          mov      ax, mode
          int   10h
     }
}

void       setPalette ( RGB * palette )
{
     for ( int i = 0; i < 256; i++ )        // convert from 8-bit to
     {                                      // 6-bit values
          palette [i].red   >>= 2;
          palette [i].green >>= 2;
          palette [i].blue  >>= 2;
     }
```

```
    asm {                           // really load palette via BIOS
        push    es
        mov     ax, 1012h
        mov     bx, 0               // first color to set
        mov     cx, 256             // # of colors
        les  dx, palette            // ES:DX == table of color values
        int  10h
        pop es
    }
}

void    initTables ()
{
    initFixMath ();

    for ( int i = 0; i < SCREEN_HEIGHT / 2; i++ )
        distTable [i] = float2Fixed ( D0 / ( ( i + 1 ) * DELTA ) ) >> 8;
    for ( i = 0; i < 1024; i++ )
    {
        float   x = i * 64.0 * M_PI / 32768;

        CSinTable [i] = (long)( C * 65536.0 * sin ( x ) );
        CCosTable [i] = (long)( C * 65536.0 * cos ( x ) );
    }
}
void    draw ( int x, int y, char * data, int w )
{
    for ( int i = 0; i < w; i++ )
        for ( int j = 0; j < w; j++ )
            *(screenPtr + x + i + (y+j)*320) = data [i+j*w];
}

main ()
{
    int  done = 0;

    angle = 0;
    locX  = 0;
    locY  = 0;

    setVideoMode ( 0x13 );
    setPalette   ( pic -> palette );
    initTables   ();

    draw ( 0,   0, pyr -> data, 64 );
    draw ( 70,  0, pyr -> data + pyr -> offs [1], 32 );
    draw ( 110, 0, pyr -> data + pyr -> offs [2], 16 );
    draw ( 130, 0, pyr -> data + pyr -> offs [3], 8 );
    draw ( 150, 0, pyr -> data + pyr -> offs [4], 4 );
    draw ( 160, 0, pyr -> data + pyr -> offs [5], 2 );
    draw ( 170, 0, pyr -> data + pyr -> offs [6], 1 );
    bioskey ( 0 );

    int  start = clock ();

    while ( !done )
    {
```

```
drawView ();
if ( bioskey ( 1 ) )
{
    .Fixed   vx = cosine ( angle ) * 1000;
     Fixed   vy = sine ( angle ) * 1000;

     switch ( bioskey ( 0 ) )
     {
          case LEFT:
               angle += ANGLE_90 / 20;
               break;

          case RIGHT:
               angle -= ANGLE_90 / 20;
               break;

          case UP:
               locX += vx;
               locY += vy;
               break;

          case DOWN:
               locX -= vx;
               locY -= vy;
               break;

          case ESC:
               done = 1;
               break;
          }
     }
     }
     float     totalTime = ( clock () - start ) / CLK_TCK;

     setVideoMode ( 0x03 );

     printf ( "\nFrames rendered     %7ld",  totalFrames );
     printf ( "\nTotal time ( sec ) :    %7.2f", totalTime );
     printf ( "\nFPS                     %7.2f", totalFrames / totalTime );
}
```

13.7. Освещение

Одним из часто встречающихся элементов является затенение с увеличением расстояния - чем дальше находится точка, тем ближе ее цвет к некоторому заданному.

Один из наиболее простых вариантов описывается формулой

$$c(d) = c_0(1 - k(d)) + c_\infty k(d),$$

где $k(d) = \begin{cases} 0, d \le d_0, \\ \dfrac{d - d_0}{d}, d > d_0, \end{cases}$; c_0- цвет объекта, c_∞ - цвет на бесконечности.

Для всех объектов, расположенных ближе чем d_0, затенения не происходит.

Поскольку на практике в качестве цвета всегда используется индекс в палитре, то непосредственно данная формула неприменима и обычно поступают следующим образом: весь диапазон расстояний разбивается на несколько интервалов (может оказаться, что удобнее разбивать не диапазон расстояний, а диапазон обратных величин), внутри каждого из которых затенение считается постоянным. Затем для каждого из этих интервалов вводится своя таблица перекодировки цветов палитры, которая и используется при выводе пикселов:

```
char  * table = distTable [ dist / DIST_STEP ];
putPixel ( x, y, table [color] );
```

Одним из недостатков подобного подхода является появление четкой границы раздела, когда одна половина грани находится в одном интервале, а другая - в другом. При этом обе половины заметно отличаются друг от друга.

Чтобы этого не возникало, можно к расстоянию добавлять небольшую случайную величину. Если это делать для каждого выводимого пиксела, то граница раздела интервалов как бы размывается и становится гораздо менее заметной. Сами случайные значения во избежание длительных расчетов можно брать из готовой таблицы.

13.8. Quake

Продолжением линии Wolfensetin3*d* - DOOM является игра Quake, основными отличительными особенностями которой является полная трехмерность, 6 степеней свободы, великолепно сделанное освещение (освещенность плавно изменяется, около факелов светлее, в углах темнее, т. е. освещенность непрерывно распределена вдоль поверхностей).

Еще одной отличительной особенностью Quake является использование вместо спрайтов трехмерных объектов (Alias mo*d*els).

Ряд идей, использованных в Quake, очень напоминает то, что было в игре DOOM, но вместе с тем появились и принципиально новые подходы.

Рассмотрим в общих чертах работу графического ядра Quake.

Все разделено на две группы:

- статический (неподвижный) мир;
- все остальное (движущиеся объекты, включая игроков, противников, двери, платформы, оружие, боеприпасы и т. д.).

Статический мир построен из большого (до 10 Кбайт) количества текстурируемых граней с произвольным количеством источников света. На основе данного набора граней строится единое BSP-дерево, разбивающее все пространство на набор выпуклых многогранников, являющихся листьями этого дерева (полная аналогия с разбиением плоского мира в игре DOOM на набор выпуклых SubSe*c*tor, только здесь все происходит в пространстве). Построенное дерево можно легко обходить в любом порядке. При этом для каждого листа дерева задаются ограничивающий его параллелепипед и набор поверхностей (под поверхностью - surfa*c*e - понимается текстурированная грань с наложенной на нее освещенностью).

Принципиально новым является использование PVS - для каждого листа дерева на этапе препроцессирования строится список всех тех листьев, которые могут быть видны из данного листа.

Таким образом, обходя дерево в порядке front-to-back, мы сперва попадаем в лист, содержащий наблюдателя внутри себя, получаем список всех видимых листьев и сортируем их при помощи BSP-дерева.

Структура PVS не только сильно ускоряет рендеринг, но и выполняет еще функции структуры BLOCKMAP в игре DOOM, определяя объекты, которые могут видеть игрока и которые может видеть игрок.

При обходе дерева (листьев) в порядке front-to-back все лицевые грани каждого листа (порядок граней внутри листа неважен, так как каждый лист - выпуклый многогранник) выводятся в s-буфер.

Таким образом, в конце обхода дерева получается готовый s-буфер, содержащий разложение всего экрана на горизонтальные отрезки, упорядоченные сверху вниз и слева направо. После этого по построенному s-буферу осуществляется рисование статической сцены. При этом повторный вывод в один и тот же пиксел экрана полностью исключается.

Приведем алгоритм, строящий изображение статической части сцены (всех BSP-моделей), используя соответствующие структуры pak-файла.

```
int  BSPFile :: isSurfaceFrontFacing ( Surface& surface ) const
    {
        return planes [surface.planeNum].normal & loc ) >= 0;
    }
        nt   BSPFile    isLeafVisible ( int leaf ) const
    {
        int  v = curLeaf -> visList;          // start of visibility list for curLeaf
        for ( register int i = 1; i < numLeaves; i++ )
            if ( visLists [v] == 0 )
            {
                i += 8 * visLists [v + 1];
                v++;
            }
            else
            {
                for ( register int bit = 0x80; bit > 0; bit >>= 1, i++ )
                    if ( ( i == leaf ) && ( visLists [v] & bit ) )
                        return 1;
            }
        return 0;
}
int  BSPFile    viewerInFront ( Plane& plane ) const
{
    return ( plane.normal & loc ) >= plane.dist;
}
void     BSPFile    traverseBSPTree ( long node )
{
    Plane * plane = &planes [nodes [node].planeNum];
    if ( node & 0x8000 )      // is it a leaf
    {
        if ( curLeaf == NULL )
            curLeaf = &leaves [~node];
        if ( isLeafVisible ( ~node ) )
```

```
                visLeaves.insert ( &leaves [~node] );
            return;
        }
                        // check whether node lies
                        // in viewing frustrum
        if ( !boxInFrustrum ( nodes [node].boundBox ) )
            return;
        if ( viewerInFront ( *plane ) )
        {
            traverseBSPTree ( nodes [node].front );
            traverseBSPTree ( nodes [node].back  );
        }
        else
        {
            traverseBSPTree ( nodes [node].back  );
            traverseBSPTree ( nodes [node].front );
        }
}
void    BSPFile   renderLeaf ( BSPLeaf& leaf )
{
    if ( !boxInFrustrum ( leaf.boundBox ) )
        return;

    int  firstSurface = leaf.firstSurface;
    int  lastSurface  = leaf.firstSurface + leaf.nuymSurfaces - 1;

    for ( int i = firstSurface; i <= lastSurface; i++ )
        if ( isSurfaceFrontFacing ( surfaces [surfaceList [i]] ) )
            sBuffer -> addSurface ( surfaces [surfaceList [i]] );
}
void    BSPFile :: render ( Hull& hull )
{
    visLeaves.deleteAll ();
    sBuffer.reset      ();

    curLeaf = NULL;

    traverseBSPTree ( hull.node );

    for ( int i = visLeaves.getCount () - 1; i >= 0; i-- )
        renderLeaf ( *(BSPLeaf *) visLeaves [i] );

    sBuffer.render ();
}
void    BSPFile   render ()
{
    resetZBuffer ();

    for ( int i = 0; i < numHulls; i++ )
        render ( hulls [i] );
}
```

Для создания правильно выглядящего освещения вводятся так называемы ты освещенности (light maps). Карта освещенности представляет собой матриц чений освещенности, построенную с шагом в 16 пикселов как по горизонтали, по вертикали. Она создается на этапе построения уровня, когда из каждой

карты освещенности выпускаются лучи ко всем источникам света, определяя тем самым первичную (непосредственную) освещенность точки.

Для наложения карты освещенности на текстуру она сначала билинейно интерполируется (для получения значений освещенности во всех промежуточных точках) и накладывается на текстуру. Для повышения качества изображения, применяется пирамидальное фильтрование (mip-mapping) со степенями сжатия 1, 2, 4 и 8.

Получившаяся текстура с наложенной на нее картой освещенности называется поверхностью (surface) и кешируется (используется когерентность по времени, так как набор поверхностей, видимых в одном кадре, вряд ли будет сильно отличаться от набора поверностей, видимых в следующем).

Поскольку грани могут иметь произвольную ориентацию, то методы текстурирования горизонтальных и вертикальных поверхностей уже не годятся. Вместо этого используется интерполяционный метод, когда значения индексов текстуры через каждые 8 или 16 пикселов (в зависимости от разрешения) вычисляются точно, а в промежутках используется линейная интерполяция. Подобный подход позволяет при сравнительно небольших затратах получать изображение с достаточно высокой степенью реалистичности.

Все остальное (кроме статического мира) представляет собой BSP-модели (двери, платформы), спрайты и полигональные модели (оружие, боеприпасы, игроки, противники).

Отдельные BSP-модели как бы "вклеиваются" в общее BSP-дерево, затем составляющие их грани отрезки добавляются в s-буфер.

Более сложные объекты, такие, как игроки и противники, в виде BSP-деревьев уже не могут быть представлены (их форма все время меняется). Поэтому они представлены в виде полигональных моделей (Alias models) - наборов большого количества текстурированных треугольных граней. Каждый такой объект имеет одну общую текстуру, и для каждой вершины треугольной грани на этой общей для всех граней текстуре задается соответствующая точка, и эта точка не меняется при изменении положения вершин треугольника в пространстве.

Это можно себе представить как если бы с реального объекта сняли кожу, распрямили ее, "натянули" полученную плоскую текстуру на каркасную модель объекта. При этом соответствие вершин граней и точек на текстуре при движении (изменении формы) модели не меняется.

Таким образом, alias model представляет собой набор вершин, набор треугольных граней, построенных на этих вершинах, текстуру и соответствие между вершинами и определенными точками текстуры.

Анимация модели заключается в изменении координат точек (вершин) модели, соответствующим образом меняются треугольные грани, но образы вершин на "шкуре" объекта не изменяются.

Для вывода всех не-BSP-объектов используется z-буфер. Когда по построенному s-буферу рисуется статическая сцена, одновременно по нему строится и z-буфер, на который потом накладываются все не-BSP модели.

Одной из особенностей игры является широкое использование в ней систем частиц (particle systems) для взрывов, разлетания брызг крови, дымовых следов от ракет и т. д., а также широкое использование в ней операций с плавающей точкой (вместо

операций с фиксированной точкой). Правда, это дает выигрыш лишь для процессоров типа Pentium, на которые, впрочем, и рассчитана игра.

На компакт-диске можно найти файлы, определяющие все основные структуры данных, используемых в игре, и необходимые классы для работы с ними.

Информацию о структуре файлов данных для игры Quake можно найти по следующему адресу:

www.gamers.org/dEngine/quake/spec/quake-spec34/qkmenu.htm

Упражнения

1. Модифицируйте программу wolf4.cpp, добавив в нее работу со спрайтами и поддержку сразу нескольких текстур.
2. Модифицируйте файл doom4.cpp, добавив в него текстурирование горизонтальных поверхностей и поддержку спрайтов (их следует брать из wad-файла).
3. Напишите процедуру текстурирования произвольной выпуклой грани, используя один из интерполяционых методов. При этом считается, что для каждой вершины грани заданы ее текстурные координаты. Процедура должна поддерживать использование mip-mapping.
4. На основе классов для работы с pak-файлом напишите программы для показа уровня игры. С этой целью потребуется изменить класс SBuffer, убрав оттуда сравнения отрезков, поскольку все грани будут поступать в порядке front-to-back. Класс SBuffer должен поддерживать вывод в него произвольных (не обязательно выпуклых) граней и вывод текстурированных отрезков в z-буфер.

Приложение
ВЫЧИСЛЕНИЯ С ФИКСИРОВАННОЙ ТОЧКОЙ

Несмотря на все преимущества вычислений с плавающей точкой (большой диапазон представимых величин, легкость использования)? в ряде случаев использование вычислений с плавающей точкой нежелательно. Основные причины этого заключаются в том, что, во-первых, не на всех процессорах вычисления с плавающей точкой реализованы достаточно эффективно и, во-вторых, преобразования между целыми и вещественными числами занимают слишком много времени. Если первый недостаток для ряда современных процессоров (Pentium, Pentium II, PowerPC) преодолен, то второй недостаток все еще остается в силе. Поэтому для получения достаточно высокого быстродействия в ряде случаев целесообразно заменить часть операций с вещественными числами на работу с целыми числами. Причем желательно, чтобы можно было по-прежнему работать с дробными числами.

Стандартный подход заключается в использовании 32-битовых целых чисел (long), где младшие 16 бит отводятся под дробную часть числа, а старшие 16 бит - под целую. Таким образом, каждому вещественному числу f ставится в соответствие целое число $l = [2^{16} f]$. Подобное представление обозначается как 16.16, показывая сколько бит отводится под целую и дробную части. Возникающая погрешность не превышает 2^{16} Рассмотрим, каким образом осуществляются операции над такими числами.

Очевидно, что сумме (разности) чисел f_1 и f_2 соответствует сумма (разность) чисел l_1 и l_2, так как

$$l = \left[2^{16}(f_1 \pm f_2)\right] \approx \left[2^{16} f_1\right] \pm \left[2^{16} f_2\right] = l_1 \pm l_2$$

Формула для произведения чисел несколько сложнее:

$$l = \left[2^{16}(f_1 f_2)\right] = \left[\frac{\left(2^{16} f_2\right)\left(2^{16} f_2\right)}{2^{16}}\right] = \frac{l_1 l_2}{2^{16}} = \left(\frac{l_1}{2^8}\right)\left(\frac{l_2}{2^8}\right)$$

Аналогично для деления чисел имеем:

$$l = \left[2^{16}\,\frac{f_1}{f_2}\right] \approx \left[2^{16}\,\frac{2^{16} f_1}{2^{16} f_2}\right] \approx 2^{16}\,\frac{l_1}{l_2} \approx \frac{2^8 l_1}{2^{-8} l_2}$$

Умножение и деление на степени двойки легко заменяются побитовыми сдвигами.

Таким образом, сложению, вычитанию и сравнению чисел с фиксированной точкой соответствуют сложение, вычитание и сравнение соответствующих 32-битовых чисел. Умножению и делению соответствуют умножение и деление с дополнительными сдвигами.

При умножении следует обратить внимание на то, что при перемножении достаточно больших чисел результат может легко выйти за отведенные 32 бита. Для компенсации этого производятся предварительные сдвиги, а не только лишь сдвиги результата операции.

Естественным шагом для повышения быстродействия является также отказ от вычисления "на ходу" значений тригонометрических и других сложных функций. Вместо этого удобнее использовать таблицы, где хранятся значения требуемых функций (с некоторым шагом).

Ниже приводятся тексты простейшей библиотеки, реализующей работу с тригонометрическими функциями в описанном представлении.

```
// File Fixmath.h
#ifndef   __FIXED_MATH__
#define   __FIXED_MATH__

#define   ANGLE_90   16384U          // angle of 90 degrees
#define   ANGLE_180  32768U          // angle of 180 degress
#define   ANGLE_270  49152U          // angle of 270 degress
#define   MAX_FIXED  0x7FFFFFFFl     // maximum possible Fixed number
#define   ONE        0x10000l        // 1.0

typedef   long           Fixed;
typedef unsigned short   Angle;

extern  Fixed * sinTable;
extern  Fixed * cosTable;
extern  Fixed * tanTable;
extern  Fixed * cotanTable;
extern  Fixed *  invSinTable;
extern  Fixed * invCosTable;

inline   Fixed   int2Fixed ( int x )
{
    return ( (long) x ) << 16;
}
inline   int  fraction2Fixed ( int a, int b )
{
    return (long)( (((long) a) << 16) / (long) b );
}
inline   Fixed   float2Fixed ( float x )
{
    return (long)( 65536.0 * x );
}
inline   int  fixed2Int ( Fixed x )
{
    return (int) ( x >> 16 );
}
inline   float    fixed2Float ( Fixed x )
{
    return ((float)x) / 65536.0;
}
inline   Fixed   fixAbs ( Fixed x )
{
    return x > 0 ? x   -x;
}
inline   Fixed   frac ( Fixed x )
{
    return x & 0xFFFFl;
```

```
}
inline    Fixed    sine ( Angle angle )
{
    return sinTable [ angle >> 3 ];
}
inline    Fixed    cosine ( Angle angle )
{
    return cosTable [ angle >> 3 ];
}
inline    Fixed    tang ( Angle angle )
{
    return tanTable [ angle >> 3 ];
}
inline    Fixed    coTang ( Angle angle )
{
    return cotanTable [ angle >> 3 ];
}
inline    Fixed    invSine ( Angle angle )
{
    return invSinTable [ angle >> 3 ];
}
inline    Fixed    invCosine ( Angle angle )
{
    return invCosTable [ angle >> 3 ];
}
inline    Angle    rad2Angle ( float angle )
{
    return (Angle)(32768.0 * angle / M_PI);
}
inline    float     angle2Rad ( Angle angle )
{
    return ((float) angle) * M_PI / 32768.0;
}
void      initFixMath ();
#endif
```



```
// File Fixmath.cpp
#include     <alloc.h>
#include     <math.h>
#include     "FixMath.h"

Fixed * sinTable;
Fixed * cosTable;
Fixed * tanTable;
Fixed * cotanTable;
Fixed * invSinTable;
Fixed * invCosTable;

void      initFixMath ()
{
    sinTable       = new Fixed [8192];
    cosTable       = new Fixed [8192];
    tanTable       = new Fixed [8192];
    cotanTable     = new Fixed [8192];
```

```
invSinTable   = new Fixed [8192];
invCosTable   = new Fixed [8192];

for ( int i = 0; i < 8192; i++ )
{
    float     x  = i * 2 * M_PI / (8192.0);
    float     sx = sin ( x );
    float     cx = cos ( x );
    float     tx = tan ( x );

    sinTable [i] = float2Fixed ( sx );
    cosTable [i] = float2Fixed ( cx );
    tanTable [i] = float2Fixed ( tx );

    if ( tx > 0.0001 || tx < -0.0001 )
        cotanTable [i] = (long)( 65536.0 / tx );
    else
        cotanTable [i] = ( tx > 0 ? 65536.0 * 10000.0
                                  -65536.0 * 10000.0 );

    if ( sx > 0.0001 || sx < -0.0001 )
        invSinTable [i] = (long)( 65536.0 / sx );
    else
        invSinTable [i] = ( sx > 0 ? 65536.0 * 10000.0
                                   -65536.0 * 10000.0 );

    if ( cx > 0.0001 || cx < -0.0001 )
        invCosTable [i] = (long)( 65536.0 / cx );
    else
        invCosTable [i] = ( cx > 0 ? 65536.0 * 10000.0
                                   -65536.0 * 10000.0 );
    }
}
```

Замечание. *Помимо предложенной формы представления чисел 16.16 возможны и другие (8.24, 2.30 и т.д.). Выбор той или иной формы определяется требуемой точностью и диапазоном представляемых чисел. При этом возможно одновременное использование чисел в разных форматах.*

Использование таблиц является одним из широко распространенных приемов, позволяющих заметно повысить быстродействие программы, и годится не только для вычисления тригонометрических функций.

Литература

1. **Роджерс Д., Адамс Дж.** Математические основы машинной графики. М.: Машиностроение, 1980.
2. **Гилой В.** Интерактивная машинная графика. - М.: Мир, 1982.
3. **Фокс Ф., Пратт М.** Вычислительная геометрия. Применение в проектировании и на производстве. - М.: Мир, 1982.
4. **Ньюмен У., Спрулл Р.** Основы интерактивной графики. - М.: Мир, 1985.
5. **Фоли Дж., ван Дэм Ф.** Основы интерактивной машинной графики. - М.: Мир, 1985.
6. Математика и САПР:. В 2 кн. - М.: Мир, 1988.
7. **Павлидис У.** Алгоритмы машинной графики и обработка изображений. М.: Радио и связь, 1988.
8. **Аммерал Л.** Машинная графика на языке С: В 4 кн. - Сол Систем, 1992.
9. **Иванов В. П., Батраков А. С.** Трехмерная компьютерная графика. - М.: Радио и связь, 1994.
10. **Хейни, Лорен.** Построение изображений методом слежения луча. - М., 1994.
11. **Уилтон Р.** Видеосистемы персональных компьютеров IBM PC и PS/2. Руководство по программированию. - М.: Радио и связь, 1994.
12. **Шикин Е. В., Боресков А. В., Зайцев А. А.** Начала компьютерной графики. - М.: Диалог-МИФИ, 1993.
13. **Шикин Е. В., Боресков А. В.** Компьютерная графика. Динамика, реалистические изображения. - М.: Диалог-МИФИ, 1995.
14. **Абраш, Майкл.** Программирование графики. Таинства. – Киев: ЕвроСиб, 1995.
15. **Майкл, Ласло.** Вычислительная геометрия и компьютерная графика на С++. - М.: Бином, 1997.
16. **Тихомиров Ю.** Программирование трехмерной графики. – СПб.: BHV, 1998.
17. **Шикин Е. В., Плис А. И.** Кривые и поверхности на экране компьютера. - М.: Диалог-МИФИ,1996.
18. Секреты программирования игр / **А. ла Мот, Д. Ратклифф, М. Семинаторе, Д. Тайлер.** - СПб.: Питер-Пресс, 1995.
19. UNIX, X Window, Motif. Основы программирования. - М.: АО "Аналитик", 1994.
20. **Доймлинг Ф., Силлеску Д.** Язык программирования PostScript. - М.: Физ.-мат. лит., 1993.
21. **Barsky B.** Computer graphics and geometric modeling using Beta-splines. - Springer Verlag, 1988.
22. **Farin G.** Curves and surfaces for computer aided geometric design. A practical guide. - Academic Press, 1990.
23. Computer graphics. Principles and practice/ **D.J. Foley, A. van Dam, S. K. Feiner, J. F. Hughes.** - Addison-Wesley, 1991.
24. **Hall R.** Illumination and color in computer generated imagenary. 1991.
25. PostScript Language Reference Manual, Second Edition, Adobe System Incorporated. - Addisson-Wesley,1990

Авторы считают целесообразным обратить внимание на ряд журналов, доступных российскому читателю, которые с известной регулярностью публикуют статьи, посвященные различным вопросам компьютерной графики. Это, в частности, "Компьютер-Пресс" и "Мир ПК"

Оглавление

Предисловие ... 3

Глава 1. СВЕТ. ЦВЕТОВОСПРИЯТИЕ. ЦВЕТОВЫЕ МОДЕЛИ ... 4

Упражнения ... 15

Глава 2. РАСПРОСТРАНЕНИЕ СВЕТА. ОСВЕЩЕННОСТЬ ... 16

2.1. Зеркальное отражение ... 16

2.2. Диффузное отражение ... 17

2.3. Идеальное преломление ... 17

2.4. Диффузное преломление ... 18

2.5. Распределение энергии ... 18

2.6. Микрофасетная модель поверхности ... 19

Упражнения ... 22

Глава 3. ГРАФИЧЕСКИЕ ПРИМИТИВЫ В ЯЗЫКАХ ПРОГРАММИРОВАНИЯ ... 23

3.1. Инициализация и завершение работы с библиотекой ... 24

3.2. Работа с отдельными точками ... 26

3.3. Рисование линейных объектов ... 26

3.3.1. Рисование прямолинейных отрезков ... 27

3.3.2. Рисование окружностей ... 27

3.3.4. Рисование дуг эллипса ... 27

3.4. Рисование сплошных объектов ... 27

3.4.1. Закрашивание объектов ... 27

3.4.2. Работа с изображениями ... 28

3.5. Работа со шрифтами ... 29

3.6. Понятие режима (способа) вывода ... 30

3.7. Понятие окна (порта вывода) ... 31

3.8. Понятие палитры ... 31

3.9. Понятие видеостраниц и работа с ними ... 33

3.10. Подключение нестандартных драйверов устройств ... 35

3.11. Построение графика функции ... 36

Упражнения ... 41

Глава 4. РАБОТА С ОСНОВНЫМИ ГРАФИЧЕСКИМИ УСТРОЙСТВАМИ ... 42

4.1. Клавиатура ... 42

4.2. Мышь ... 44

4.2.1. Инициализация и проверка наличия мыши ... 51

4.2.2. Высветить на экране курсор мыши ... 51

4.2.3. Убрать (сделать невидимым) курсор мыши ... 51

4.2.4. Прочесть состояние мыши (ее координаты и состояние кнопок) 51

4.2.5. Передвинуть курсор мыши в точку с заданными координатами 52

4.2.6. Установка области перемещения курсора ... *52*
4.2.7 Задание формы курсора *52*
4.2.8. Установка области гашения ... *52*
4.2.9. Установка обработчика событий ... *52*
4.3. Джойстик ... 57
4.4. Сканер ... 60
4.5. Принтер ... 60
4.5.1. Девятиигольчатые принтеры ... *62*
4.5.2. Двадцатичетырехигольчатые (LQ) принтеры ... *65*
4.5.3. Лазерные принтеры ... *66*
4.5.4. PostScript-устройства ... *68*
4.6. Видеокарты EGA и VGA ... 70
4.7. Шестнадцатицветные режимы адаптеров EGA и VGA ... 72
4.7.1. Graphics Controller (порты 3CE-3CF) ... *75*
4.7.2. Sequencer (порты 3C4-3C5) ... *75*
4.8. Режимы чтения ... 76
4.8.1. Режим чтения 0 ... *76*
4.8.2. Режим чтения 1 ... *76*
4.9. Режимы записи ... 77
4.9.1. Режим записи 0 ... *77*
4.9.2. Режим записи 1 *78*
4.9.3. Режим записи 2 ... *79*
4.9.4. Режим адаптера VGA с 256-цветами ... *81*
4.9.5. Спрайты и работа с ними ... *82*
4.9.6. Нестандартные режимы адаптера VGA (X-режимы) ... *91*
4.10. Программирование SVGA-адаптеров ... 99
4.10.1. Непалитровые режимы адаптеров SVGA ... *110*
4.10.2. Стандарт VBE 2.0 (VESA BIOS Extension 2.0) ... *113*
Упражнения ... 125
а 5. Принципы построения пользовательского интерфейса ... 126
5.1. Основные типы окон ... 131
5.1.1. Пример реализации основных оконных функций ... *135*
Упражнения ... 155
а 6. РАСТРОВЫЕ АЛГОРИТМЫ 156
6.1. Растровое представление отрезка. Алгоритм Брезенхейма ... 157
6.2. Растровая развертка окружности ... 161
6.3. Растровая развертка эллипса ... 164
6.4. Закраска области, заданной цветом границы ... 165
6.5. Заполнение многоугольника ... 173
Упражнения ... 180
а 7. ПРЕОБРАЗОВАНИЯ НА ПЛОСКОСТИ 181
7.1. Аффинные преобразования на плоскости ... 181
7.2. Однородные координаты точки ... 192

8. ОСНОВНЫЕ АЛГОРИТМЫ ВЫЧИСЛИТЕЛЬНОЙ ГЕОМЕТРИИ 197
8.1. Отсечение отрезка. Алгоритм Сазерленда - Кохена ... 197
8.2. Классификация точки относительно отрезка ... 199
8.3. Расстояние от точки до прямой 200
8.4. Нахождение пересечения двух отрезков ... 200
8.5. Проверка принадлежности точки многоугольнику ... 201
8.6. Вычисление площади многоугольника ... 205
8.7. Построение звездчатого полигона ... 205
8.8. Построение выпуклой оболочки ... 206
8.9. Пересечение выпуклых многоугольников ... 209
8.10. Построение триангуляции Делоне ... 213
Упражнения ... 219
9. Преобразования в пространстве, проектирование ... 220
9.1. Платоновы тела ... 224
9.2. Виды проектирования ... 226
9.3. Особенности проекций гладких отображений ... 250
10. УДАЛЕНИЕ НЕВИДИМЫХ ЛИНИЙ И ПОВЕРХНОСТЕЙ ... 254
10.1. Построение графика функции двух переменных. Линии горизонта ... 257
10.2. Методы оптимизации ... 268
10.2.1. Отсечение нелицевых граней ... 268
10.2.2. Ограничивающие тела (Bounding Volumes) ... 272
10.2.3. Разбиение пространства (плоскости) (Spatial Subdivision) ... 273
10.2.4. Иерархические структуры (Hierarchies) 273
10.3. Удаление невидимых линий ... 274
10.3.1. Алгоритм Робертса ... 274
10.3.2. Количественная невидимость. Алгоритм Аппеля ... 277
10.4. Удаление невидимых граней ... 280
10.4.1. Метод трассировки лучей 280
10.4.2. Метод z-буфера 281
10.4.3. Алгоритмы упорядочения 285
10.4.3.1. Метод сортировки по глубине. Алгоритм художника ... 286
10.4.3.2. Метод двоичного разбиения пространства ... 292
10.4.4. Метод построчного сканирования ... 298
10.4.5. Алгоритм Варнака (Warnock) 311
10.4.6. Алгоритм Вейлера-Эйзертона (Weiler - Atherton) ... 313
10.5. Специальные методы оптимизации ... 314
10.5.1 Потенциально видимые множества граней ... 314
10.5.2. Метод порталов ... 315
10.5.3. Метод иерархических подсцен 316
Упражнения ... 317

Глава 11. ПРОСТЕЙШИЕ МЕТОДЫ РЕНДЕРИНГА ПОЛИГОНАЛЬНЫХ МОДЕЛЕЙ ... 318
11.1. Метод постоянного закрашивания ... 318
11.2. Метод Гуро ... 318
11.3. Метод Фонга ... 319
Упражнения ... 320
Глава 12. РАБОТА С БИБЛИОТЕКОЙ OpenGL ... 321
12.1. Рисование геометрических объектов ... 323
12.2. Рисование точек, линий и многоугольников ... 325
12.3. Преобразования объектов в пространстве. Камера ... 328
12.4. Дисплейные списки ... 332
12.5. Задание моделей закрашивания ... 334
12.6. Освещение ... 334
12.7. Полупрозрачность. Использование α-канала ... 337
12.8. Вывод битовых изображений ... 338
12.9. Ввод/вывод цветных изображений ... 339
12.10. Наложение текстуры ... 339
12.11. Работа с OpenGL в Windows ... 341
Упражнения ... 349
Глава 13. ЭЛЕМЕНТЫ ВИРТУАЛЬНОЙ РЕАЛЬНОСТИ ... 350
13.1. Wolfenstein 3-D. Ray Casting ... 350
13.2. Текстурирование горизонтальных поверхностей ... 383
13.3. DOOM ... 388
13.4. Descent ... 430
13.5. Текстурирование в общем случае ... 430
13.6. Пирамидальное фильтрование (mipmapping) ... 440
13.7. Освещение ... 447
13.8. Quake ... 448
Упражнения ... 452
Приложение. ВЫЧИСЛЕНИЯ С ФИКСИРОВАННОЙ ТОЧКОЙ ... 453
Литература ... 457