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

Кафедра электронных вычислительных средств

Алгоритмические основы компьютерной графики

Методическое пособие

для студентов специальности 1 – 40 02 02
«Электронные вычислительные средства»
дневной формы обучения

Минск БГУИР 2009

УДК 004.92(076)
ББК 32.973.202-018.2 я7
А45

Р е ц е н з е н т:
доц. кафедры ЭВМ БГУИР, канд. техн. наук Ал.А. Петровский

А в т о р ы:
М. З. Лившиц, Д. С. Лихачев, А. А. Петровский, М. И. Вашкевич,
М. М. Родионов

А45 **Алгоритмические** основы компьютерной графики: Методическое пособие для студ. спец. 1–40 02 02 «Электронные вычислительные средства» днев. формы обуч. / М. З. Лившиц [и др.] – Минск : БГУИР, 2009. – 39 с.: ил.
ISBN 978-985-488-402-8

В пособии описываются базовые алгоритмы компьютерной графики. В первом разделе рассматриваются растровые алгоритмы представления отрезка, развертывания окружности и эллипса. Во втором разделе подробно, с практической точки зрения, описываются аффинные преобразования на плоскости и в пространстве. Последний раздел методического пособия посвящен построению перспективных изображений пространственных объектов. Излагаемый материал сопровождается программами на языке С++.

УДК 004.92(076)
ББК 32.973.202-018.2 я7

ISBN 978-985-488-402-8

СОДЕРЖАНИЕ

1. РАСТРОВЫЕ АЛГОРИТМЫ

1.1. Растровое представление отрезка. Алгоритм Брезенхейма

Подавляющее число графических устройств являются растровыми, представляя изображение в виде прямоугольной матрицы 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 координаты отличаются не более чем на единицу: $|x_1 - x_2| \le 1,\ |y_1 - y_2| \le 1$.

Понятие 4-связности является более сильным: любые два 4-связных пиксела являются и 8-связными, но не наоборот. На рис. 1.1 изображены 8-связная линия (а) и 4-связная линия (б).

а) б)

Рис. 1.1. Растровое изображение отрезка: а) 8-связная линия, б) 4-связная линия

В качестве линии на растровой сетке выступает набор пикселов $P_1,P_2,...,P_n$, где любые два пиксела P_i, P_{i+1} являются соседними в смысле заданной связности.

Замечание. *Так как понятие линии базируется на понятии связности, то естественным образом возникает понятие 4- и 8-связных линий. Поэтому, когда говорится 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.*

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

$$y = y_a + \frac{y_b - y_a}{x_b - x_a} \cdot (x - x_a),\ x \in [x_a, x_b] \text{ или } y = kx + b,$$

где

$$k = \frac{y_b - y_a}{x_b - x_a},$$
$$b = y_a - kx_a.$$

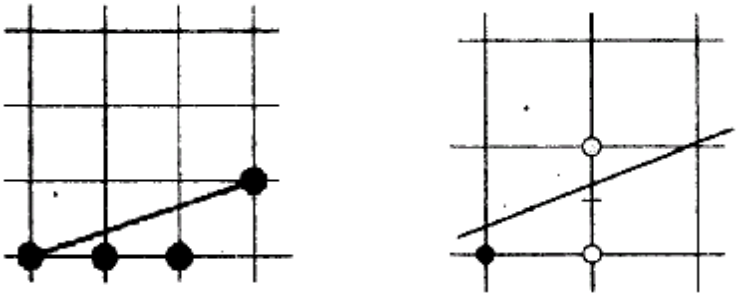
Отсюда получаем простейший алгоритм растрового представления отрезка:

```
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);
}
```

Вычисления значений функции $y = kx + b$ можно избежать, используя в цикле реккуррентные соотношения, так как при изменении x на 1 значение y изменяется на k.

```
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 получение целой части y может приводить к не всегда корректному изображению (рис. 1.2).



а) б)

Рис. 1.2. Растровое изображение отрезка

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. 1.2). Для этого достаточно сравнить дробную часть y с 1/2.

Пусть $x_0=x_a$, $y_0=y_a$, …, $x_n=x_b$, $y_n=y_b$ – последовательность изображаемых пикселей, причем $x_{i+1}-x_i=1$. Тогда каждому значению x_i соответствует число kx_i+b.

Обозначим через c_i дробную часть соответствующего значения функции $kx_i+b-c_i=\{kx_i+b\}$.

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 следующему значению x_{i+1}.

Само значение функции при этом изменяется на k. Если $c_i+k \le 1/2$, то $c_{i+1}=c_i+k$, $y_{i+1}=y$.

В противном случае необходимо увеличить y на единицу, и тогда приходим к 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:

```
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 $c_i = k$). Получаем следующую программу:

```
void line (int xa, int ya, int xb, int yb, int color)
{
   double к = ((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.

```
// Простейший алгоритм Брезенхейма 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.

```
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 $|\Delta y| \le |\Delta x|$ необходимо x и y поменять местами. Полный текст соответствующей программы приводится ниже.

```
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);
     }
   }
```

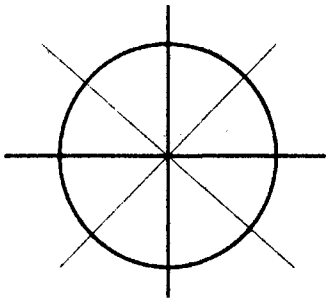
```
   else {
     int d = ( dx « 1 ) - dy;
     int d1 = dx « 1;
     int d2 = (dx-dy)«1;

     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);
    }
  }//else
}
```

1.2. 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 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:

```
...
int xCenter;
int yCenter;
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);
}
```



Рассмотрим участок окружности из второго октанта $x \in \left[0, R/\sqrt{2}\right]$, $y \in \left[R/\sqrt{2}, R\right]$. Особенностью данного участка является то обстоятельство, что угловой коэффициент касательной к окружности не превосходит 1 по модулю, а точнее, лежит между минус единицей и нулем. Применим к этому участку алгоритм средней точки (midpoint algorithm).

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 (или y_i) следует взять в качестве y_{i+1} введем переменную

$$d_i = F(x_i + 1, y_i - 1/2) = (x_i + 1)^2 + (y_i - 1/2)^2 - R^2 .$$

В случае, когда $d_i < 0$, полагаем $y_{i+1} = y_i$. Тогда

$$d_{i+1} = F(x_i + 2, y_i - 1/2 = (x_i + 2)^2 + (y_i - 1/2)^2 - R^2 ,$$

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

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, определяется итерационный механизм перехода от одного пиксела к другому. В качестве начального пиксела берется точка $(1, R)$. Тогда $d_0 = F(1, R - 1/2) = 5/4 - R$ и приходим к алгоритму:

```
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 $1/4 + z, z \in Z$ и, значит, изменяется только на целое число. Поэтому дробную часть (всегда равную 1/4) можно отбросить, перейдя тем самым к полностью целочисленному алгоритму.

```
void circle2 (int xc, int yc, int r, int color)
{
   int x    =0;
   int y    = r;
   int d    = 1 - r;
   int delta 1 = 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; deltal += 2; delta2 += 4; x++; y--;
      }
      CirclePoints ( x, y, color);
    }
}
```

1.3. Растровая развертка эллипса

Уравнение эллипса (рис. 1.3) с осями, параллельными координатным осям, имеет следующий вид:

$$\frac{x^2}{a^2}+\frac{y^2}{b^2}=1.$$

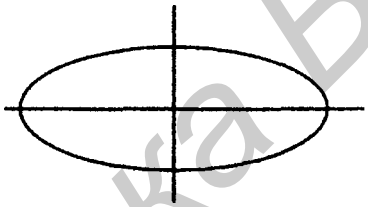


Рис. 1.3. Эллипс

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: $x>0$, $y>0$. Разобьем четверть эллипса на две части: ту, где угловой коэффициент лежит между минус единицей и нулем, и ту, где угловой коэффициент меньше минус единицы (рис. 1.4).

Рис. 1.4. Разбиение четверти эллипса на две части

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, вычисляется значение *F* между кандидатами и используется 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

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

При $d_i<0$ полагаем $y_{i+1}=y_i$ и

$$d_{i+1}=F\left(x_i+2,y_i-\frac{1}{2}\right)=b^2\left(x_i+2\right)^2+a^2\left(y_i-\frac{1}{2}\right)-a^2b^2,$$

$$\Delta d_i=b^2\left(2x_i+3\right).$$

При $d_i>0$ полагаем $y_{i+1}=y_i+1$ и

$$d_{i+1}=F\left(x_i+2,y_i-\frac{3}{2}\right)=b^2\left(x_i+2\right)^2+a^2\left(y_i-\frac{3}{2}\right)-a^2b^2,$$

$$\Delta d_i=b^2\left(2x_i+3\right)+a^2\left(2-2y_i\right).$$

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 $(1,b-1/2)$ – 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 необходимо не только проверять знак переменной d_i но и, вычисляя градиент в средней точке, следить за тем, не пора ли переходить в часть 2.

1.4. 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

```
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 находятся 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. 1.5).

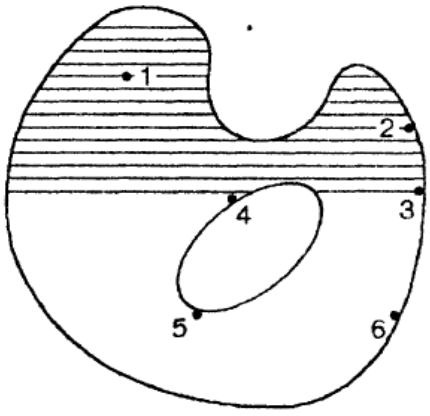


Рис. 1.5. Закраска области сложной формы

```
#include <…>

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;

                  // поиск границ отрезка xl и xr
      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 );     // закраска отрезка

      // закраска смежных отрезков в том же направлении
      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 ()
{
       …
      circle ( 320, 200, 140 );
      circle ( 260, 200, 40 );
      circle ( 380, 200, 40 );
      …
      setcolor ( color );

      fill ( 320, 300 );
     …
}
```

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:

шаг 1. построение упорядоченного списка граничных пикселов (отслеживается внешняя граница);

шаг 2. проверка внутренности (для обнаружения в ней дыр);

шаг 3. 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 (рис. 1.6).

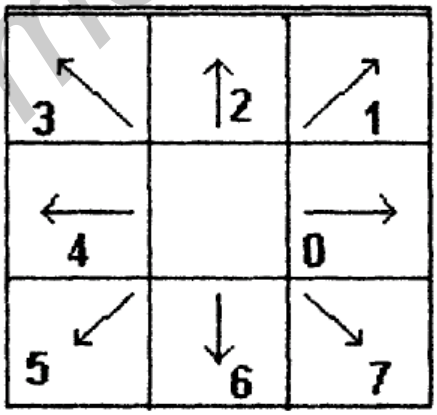


Рис. 1.6. Нумерация пикселов на 8-связной решетке

Ниже приводится программа заполнения области на основе этого алгоритма.

```
#include   <…>

#define    BLOCKED    1
#define    UNBLOCKED  2
#define    FALSE      0
#define    TRUE       0xFFFF

int  borderColor = WHITE;
```

```
int  fillColor   = GREEN;

struct BPStruct           // таблица пикселов границы
{
     int  x, y;
     int  flag;
} bp [3000];

int       bpStart;
int       bpEnd = 0;
BPStruct  currentPixel;
int       D;              // текущее направление поиска
int       prevD;          // предыдущее направление поиска
int       prevV;          // предыдущее вертикальное направление

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 )              // перемещение вправо
       bp [bpEnd-1].flag = BLOCKED;  // блокировка пред. пиксела
     else
     if ( prevD != 4 )          // если не смещаемся горизонтально
       appendBPList ( currentPixel.x, currentPixel.y, UNBLOCKED );
}

void differentDirection ( int d )
{
     if ( prevD == 4 )          // предыдущее перемещение влево
     {
          if ( prevV == 5 )
               bp [bpEnd-1].flag = BLOCKED;

          appendBPList ( currentPixel.x, currentPixel.y, BLOCKED );
     }
     else
     if ( prevD == 0 )          // предыдущее перемещение вправо
     {
          // блокировка самого правого пиксела в строке
          bp [bpEnd-1].flag = BLOCKED;
          if ( d == 7 )             // если линия началась сверху
          appendBPList (currentPixel.x, currentPixel.y, BLOCKED);
          else
          appendBPList (currentPixel.x, currentPixel.y, UNBLOCKED);
     }
     else // предыдущее перемещение было в любом вертикальном
          // направлении
     {
       appendBPList ( currentPixel.x, currentPixel.y, UNBLOCKED );
          // добавляем пиксел дважды, если локальный min и 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;   // обновляем предыдущее направление
      }

     prevD = d;            // новое значение предыдущего направления
                           // поиска
}

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--;        // переход вверх
               break;

          case 5:
          case 6:
          case 7:
               y++;        // переход вниз
               break;
     }

     switch ( direction )
     {
          case 3:
          case 4:
          case 5:
               x--;        // переход влево
               break;

          case 1:
          case 0:
          case 7:
               x++;        // переход вправо
               break;
     }
}

int  findBP ( int d )
{
```

```
      int   x = currentPixel.x;
      int   y = currentPixel.y;

      nextXY ( x, y, d );  // получаем x,y пиксела в направлении d

      if ( getpixel ( x, y ) == borderColor )
      {
            addBPList ( d );     // добавляем пиксел x,y в таблицу
            currentPixel.x = x;  // пиксел x,y становится текущим
            currentPixel.y = y;
            return TRUE;
      }

      return FALSE;
}

int   findNextPixel ()
{
      for ( int i = -1; i <= 5; i++ )
      {
            // поиск следующего пиксела границы
            int  flag = findBP ( ( D + i ) & 7 );

            if ( flag )                              {
                  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; )
      {
            // пропуск пиксела, если заканчивается блок
            if ( bp [i].flag == BLOCKED )   i++;
            else
            if ( bp [i].y != bp [i+1].y )
                // пропускаем последний пиксел в строке
                  i++;
            else
            {
   // если хотя бы один пиксел для закраски, тогда сканируем строку
                  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; )
        {
            // пропуск пиксела, если заблокирован
            if ( bp [i].flag == BLOCKED )
                i++;
            else
            // пропуск пиксела, если последний в строке
            if ( bp [i].y != bp [i+1].y )
                i++;
            else            // есди хотя бы один пиксел для закраски
            {               // рисуем линию
              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;// текущее направление поиска
     prevD          = 8;// предыдущее направление поиска
     prevV          = 2;//наиболее частое вертикальное направление

     do                    // цикл вокруг границы
     {                     // до возврата в начальную точку
          nextFound = findNextPixel ();
          done = (currentPixel.x == startX) &&
                 (currentPixel.y == startY);
     } while ( nextFound && !done );

     if ( !nextFound )     // если только один пиксел в границе
     {                     // добавляем его дважды в таблицу
          appendBPList ( startX, startY, UNBLOCKED );
          appendBPList ( startX, startY, UNBLOCKED );
     }
     else                  // если последнее направление было вверх
                           // добавляем начальный пиксел в таблицу
     if ( (prevD <= 3) && (prevD >= 1) )
          appendBPList ( startX, startY, UNBLOCKED );
}

borderFill ( int x, int y )
{
     do                    // проходим по всей таблице
     {
          traceBorder ( x, y ); // обход границы, начиная с т. x,y
          sortBP      ();       // сортировка таблицы
          scanRegion  ( x, y ); // поиск областей внутри контура
     } while ( bpStart < bpEnd );

     fillRegion ();  // использование таблицы для закраски областей
}


main ()
{
     …
      circle ( 320, 200, 140 );
     circle ( 260, 200, 40 );
     circle ( 380, 200, 40 );

     setcolor ( fillColor );

     borderFill ( 460, 200 );
    …
}
```

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.

2. Афинные преобразования

2.1. 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 система координат. Тогда каждой точке М ставится в соответствие упорядоченная пара чисел (x, y) ее координат (рис. 2.1). Вводя на плоскости еще одну прямолинейную систему координат, ставим в соответствие той же точке М другую пару чисел – (x^*, y^*).

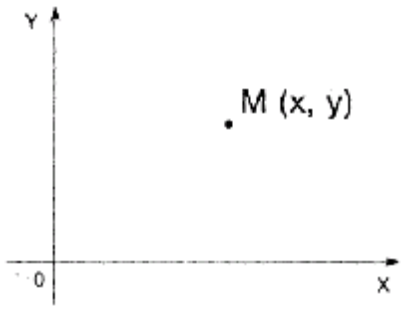


Рис. 2.1. Точка в прямолинейной системе координат

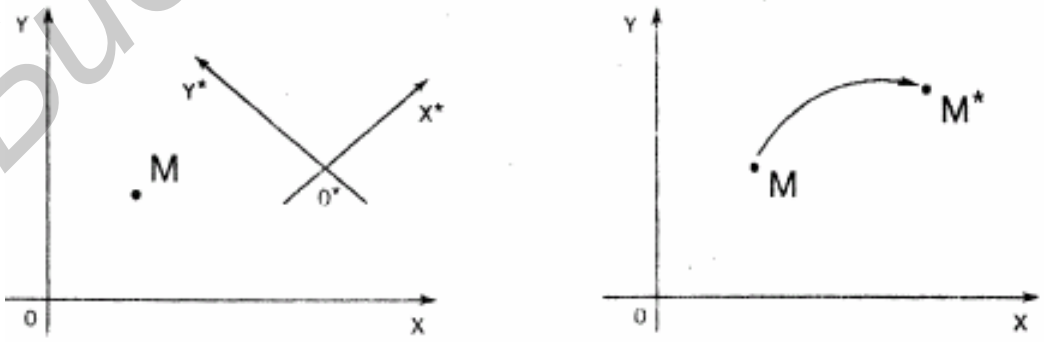
Переход от одной прямолинейной координатной системы на плоскости к другой описывается следующими соотношениями:

$$x^* = a\,x + b\,y + l,$$

$$y^* = g\,x + d\,y + m,$$

где a, b, g, l, m – произвольные числа, связанные неравенством $\begin{vmatrix} a & b \\ g & d \end{vmatrix} \neq 0$.

Преобразование на плоскости можно рассматривать двояко: либо сохраняется точка и изменяется координатная система (рис. 2.2, а) – в этом случае произвольная точка М остается той же, изменяются лишь ее координаты, либо изменяется точка и сохраняется координатная система (рис. 2.2, б).



а) б)

Рис. 2.2. Поворот системы координат или точки

В дальнейшем будем рассматривать аффинные преобразования 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:

1) поворот вокруг начальной точки на угол j описывается формулами

$$x^* = x\cos j - y\sin j,$$
$$y^* = x\sin j + y\cos j,$$

или в матричном представлении

$$\begin{pmatrix} x^* & y^* \end{pmatrix} = \begin{pmatrix} x & y \end{pmatrix} \cdot R,$$

где

$$R = \begin{pmatrix} \cos j & \sin j \\ -\sin j & \cos j \end{pmatrix};$$

2) растяжение (сжатие) вдоль координатных осей:

$$x^* = a\,x,$$
$$y^* = d\,y,$$
$$a > 0, d > 0,$$

или

$$D = \begin{pmatrix} a & 0 \\ 0 & d \end{pmatrix}.$$

Растяжение (сжатие) вдоль оси абсцисс обеспечивается при условии, что $a > 0$ ($a < 0$);

3) отражение (относительно оси абсцисс, ординат) задается при помощи формул

$$\begin{matrix} x^* = x, \\ y^* = -y, \end{matrix} \text{ (абсцисс), } \qquad \begin{matrix} x^* = -x, \\ y^* = y, \end{matrix} \text{ (ординат)}$$

или

$$M_X = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}, \; M_Y = \begin{pmatrix} -1 & 0 \\ 0 & 1 \end{pmatrix};$$

4) перенос обеспечивается следующими соотношениями:

$$x^* = x + l,$$

$$y^* = y + m.$$

Выбор этих четырех частных случаев определяется двумя обстоятельствами:

а) каждое из приведенных выше преобразований имеет простой и наглядный геометрический смысл;

б) такие преобразования всегда можно представить как последовательное исполнение (суперпозицию) простейших преобразований вида 1) –4) (или части этих преобразований).

Для эффективного использования этих известных формул в задачах компьютерной графики более удобной является их матричная запись.

2.2. Однородные координаты точки

Пусть M – произвольная точка плоскости с координатами x и y, вычисленными относительно заданной прямолинейной системы координат. 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. 2.3).

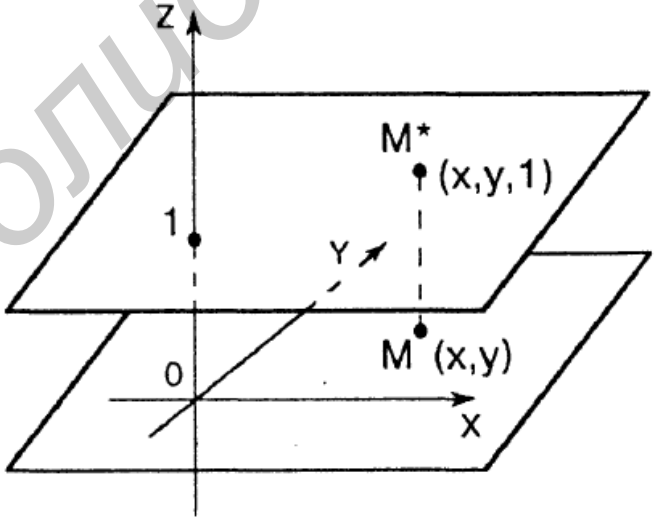


Рис. 2.3. Точка на плоскости и в пространстве

Таким образом, 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). При этом $h \neq 0$.

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.

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 аффинного преобразования в виде уравнений и в матричном представлении:

$$\left(x^*, y^*, 1\right) = \left(x, y, 1\right)\begin{pmatrix} a & g & 0 \\ b & d & 0 \\ l & m & 1 \end{pmatrix}$$

или

$$\begin{pmatrix} x^* \\ y^* \\ 1 \end{pmatrix} = \begin{pmatrix} a & b & l \\ g & d & m \\ 0 & 0 & 1 \end{pmatrix}\begin{pmatrix} x \\ y \\ 1 \end{pmatrix}.$$

Запишем матрицы элементарных преобразований с учетом введенных однородных координат:

матрица вращения: $R = \begin{pmatrix} \cos j & \sin j & 0 \\ -\sin j & \cos j & 0 \\ 0 & 0 & 1 \end{pmatrix}$,

матрица растяжения: $D = \begin{pmatrix} a & 0 & 0 \\ 0 & d & 0 \\ 0 & 0 & 1 \end{pmatrix}$,

матрица отражения относительно оси OX: $M_x = \begin{pmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$,

матрица отражения относительно оси OY: $M_y = \begin{pmatrix} -1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$,

матрица переноса: $T = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ l & m & 1 \end{pmatrix}$.

2.3. Преобразования в пространстве

Обратимся теперь к трехмерному случаю (3D) (3-dimension) и введем однородные координаты. Поступая аналогично тому, как это было сделано в случае 2D пространства, заменим координатную тройку (*x*, *y*, *z*), задающую точку в пространстве, на четверку чисел (*x*, *y*, *z*, 1) или, в общем случае, 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).

Матрицы вращения в пространстве:

1) матрица вращения вокруг оси абсцисс на угол j :

$$R_x = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos j & \sin j & 0 \\ 0 & -\sin j & \cos j & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix};$$

2) матрица вращения вокруг оси ординат на угол y :

$$R_y = \begin{pmatrix} \cos y & 0 & -\sin y & 0 \\ 0 & 1 & 0 & 0 \\ \sin y & 0 & \cos y & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix};$$

3) матрица вращения вокруг оси аппликат на угол c :

$$R_z = \begin{pmatrix} \cos c & \sin c & 0 & 0 \\ -\sin c & \cos c & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

Матрица растяжения (сжатия):

$$D = \begin{pmatrix} a & 0 & 0 & 0 \\ 0 & b & 0 & 0 \\ 0 & 0 & g & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix},$$

где a – коэффициент растяжения (сжатия) вдоль оси абсцисс, b – коэффициент растяжения (сжатия) вдоль оси ординат, g – коэффициент растяжения (сжатия) вдоль оси аппликат.

Матрица отражения:

1) матрица отражения относительно плоскости *xy*:

$$M_z = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix};$$

2) матрица отражения относительно плоскости *yz*:

$$M_x = \begin{pmatrix} -1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix};$$

3) матрица отражения относительно плоскости *zx*:

$$M_y = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

Матрица переноса:

$$T = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ l & m & n & 1 \end{pmatrix},$$

где (l, m, n) – вектор переноса.

Таким образом, любую задачу преобразования на плоскости и в пространстве можно разделить на ряд элементарных преобразований. В результате матрица, отвечающая за преобразование, будет равна произведению матриц элементарных преобразований. Для того чтобы осуществить преобразование выпуклого многогранника, необходимо каждую его вершину умножить на матрицу преобразования, и таким образом получим набор вершин нового выпуклого многогранника.

3. ПЕРСПЕКТИВНЫЕ ИЗОБРАЖЕНИЯ

В перспективном изображении параллельные горизонтальные линии встречаются в так называемой *точке схода* (рис. 3.1). Все точки схода лежат на одной прямой линии, которая называется линией *горизонта*. Заметим, что линия горизонта и точка схода являются особенностью изображения и реально не существуют в трехмерном пространстве. В течение многих веков эта концепция использовалась художниками для получения реалистичных изображений трехмерных объектов. Такие изображения обычно называются перспективными.

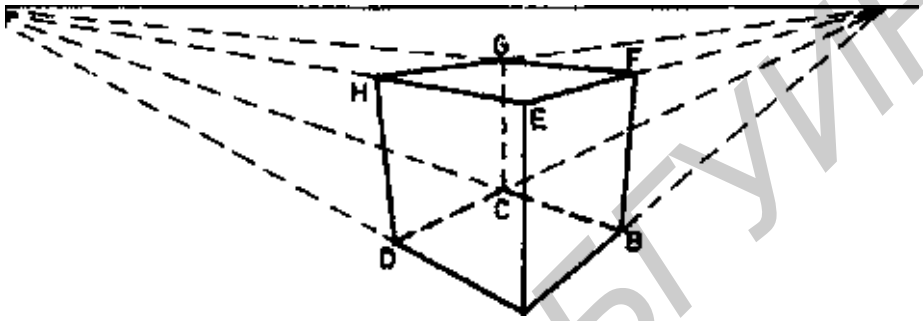
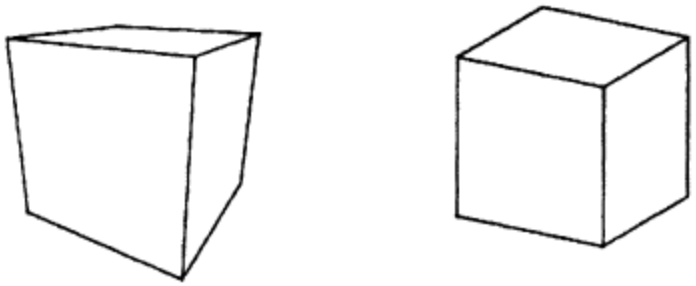


Рис. 3.1. Точки схода на горизонте

Изобретение фотографии предложило новый (более легкий) способ формирования перспективных изображений. Существует строгая аналогия между камерой, применяемой в фотографии, и человеческим глазом. Наш глаз представляет собой очень сложный инструмент, а фотокамера является его простейшей имитацией. Очевидно, что картинка будет зависеть от положения глаза. Особо важное значение имеет расстояние между глазом и объектом, поскольку «эффект перспективы» будет обратно пропорционален этому расстоянию. Если глаз расположен очень близко от объекта, то получим сильный эффект перспективы, как на рис. 3.2, а. Здесь можно четко видеть, что продолжения изображений параллельных линий на картинке пересекаются. С другой стороны, если глаз расположен далеко от объекта (по сравнению с размером объекта), то параллельные линии объекта будут казаться параллельными и на картинке (рис. 3.2, б).

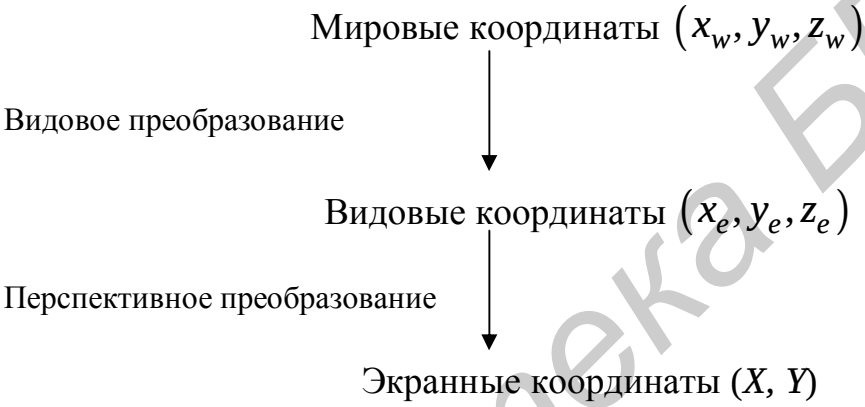


а) б)

Рис. 3.2. Расположение глаза: а) близко к объекту, б) далеко от объекта

При необходимости получения перспективной проекции задается большое количество точек P(*x, y,* z), принадлежащих объекту, для которых предстоит вычислить координаты точек изображения P'(*X, Y*) на картинке. Для этого нужно только преобразовать координаты точки P из так называемых ***мировых координат*** (*x, y,* z) в ***экранные координаты*** (*X, Y*) ее центральной проекции P'. Будем предполагать, что экран расположен между объектом и глазом E. Для каждой точки P объекта прямая линия PE пересекает экран в точке P'.

Это отображение удобно выполнять в два этапа. Первый этап заключается в видовом преобразовании – точка P остается на своем месте, но система мировых координат переходит в систему ***видовых координат***. На втором этапе осуществляется ***перспективное преобразование.*** Это точное преобразование точки P в точку P', объединенное с переходом из системы трехмерных видовых координат в систему двухмерных экранных координат.



3.1. Видовое преобразование

Для выполнения видовых преобразований должны быть заданы точка наблюдения, совпадающая с глазом, и объект. Желательно, чтобы система мировых координат была правой. Будет удобно, если начало ее координат располагается где-то вблизи центра объекта, поскольку объект наблюдается в направлении от E к O. Пусть точка наблюдения E будет задана в сферических координатах r, q, j по отношению к мировым координатам (рис. 3.3). То есть мировые координаты могут быть вычислены по формулам

$$x_E = r \sin j \cos q,$$
$$y_E = r \sin j \sin q,$$
$$z_E = r \cos j.$$

Вектор направления EO (равный минус OE) определяет направление наблюдения. Из точки наблюдения E можно видеть точки объекта только

внутри некоторого конуса, ось которого совпадает с линией EO, а вершина — с точкой E (рис. 3.4).

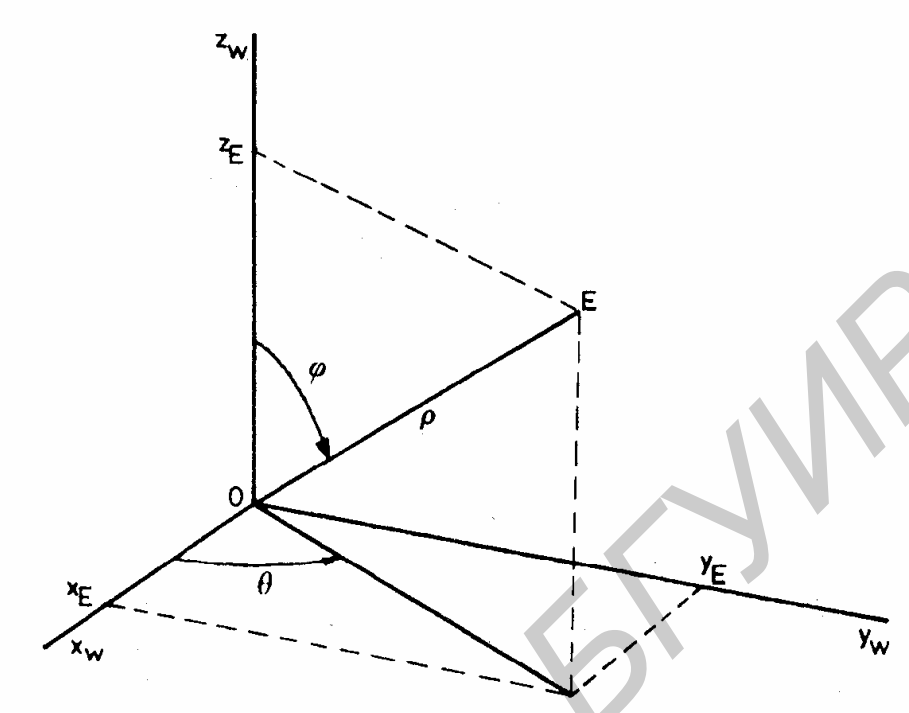


Рис. 3.3. Сферические координаты точки наблюдения E

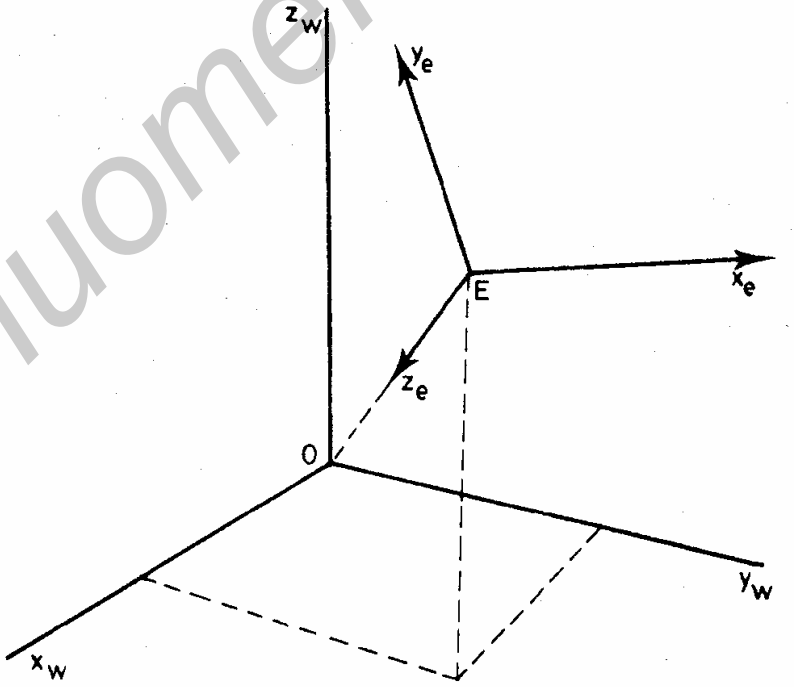


Рис. 3.4. Расположение системы видовых координат в системе мировых координат (x_w, y_w, z_w – мировые координаты; x_e, y_e, z_e – видовые координаты)

Видовое преобразование может быть записано в форме $(x_e\ y_e\ z_e) = (x_w\ y_w\ z_w)V$, где V – матрица видового преобразования размерами 4x4. Видовое преобразование состоит из четырех элементарных преобразований:

1) Перенос начала из т.О в т.Е (рис. 3.5):

$$T = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -x_E & -y_E & -z_E & 1 \end{pmatrix};$$

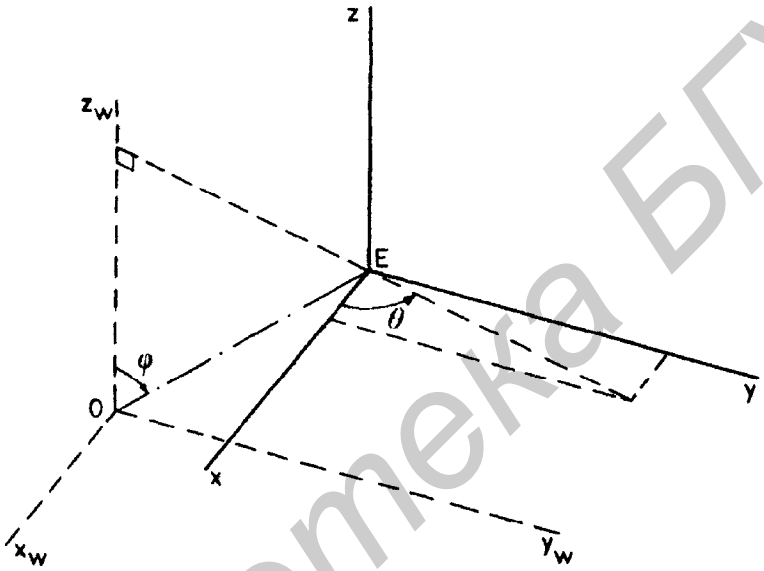


Рис. 3.5. Новые оси после переноса

2) поворот координатной системы вокруг оси z:

обращаясь к рис. 3.5, повернем систему координат вокруг оси z на угол $p/2 - q$ в отрицательном направлении. В результате ось y совпадет по направлению с горизонтальной составляющей отрезка OE, а ось x будет расположена перпендикулярно отрезку OE. Матрица для такого изменения координат будет совпадать с матрицей для поворота точки на такой же угол в положительном направлении. Матрица 3x3 для этого поворота равна

$$R_x = \begin{pmatrix} \sin q & \cos q & 0 & 0 \\ -\cos q & \sin q & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix};$$

новое положение осей представлено на рис. 3.6.

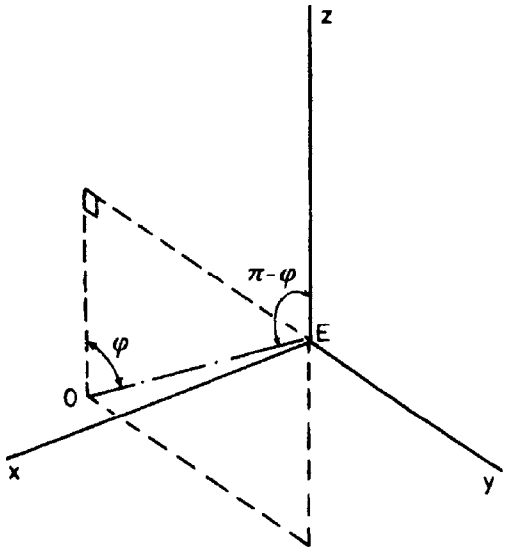


Рис. 3.6 Положение осей после поворота вокруг оси z

3) поворот системы координат вокруг оси x:

поскольку новая ось z должна совпадать по направлению с отрезком EO, повернем систему координат вокруг оси x на угол $p - j$ в положительном направлении, что соответствует повороту точки на угол $-(p - j) = j - p$:

$$R_x = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & -\cos q & -\sin q & 0 \\ 0 & \sin q & -\cos q & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix};$$

4) изменение направления оси *x* осуществляется при помощи матрицы

$$M_{yz} = \begin{pmatrix} -1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

Таким образом, матрица видового преобразования может быть представлена в следующем виде: $V = T \cdot R_z \cdot R_x \cdot M_{yz}$.

3.2. Перспективное преобразование

На рис. 3.7 выбрана точка Q, видовые координаты которой равны (0, 0, d) для некоторого положительного числа d. Плоскость z = d определяет экран, который будем использовать (картинную плоскость). Таким образом, экран — это плоскость, проходящая через точку Q и перпендикулярная оси z.

Экранные координаты определяются привязкой начала к точке Q, а оси X и Y имеют такие же направления, как оси x и y соответственно. Для каждой точки объекта P точка изображения P' определяется как точка пересечения прямой линии PE и экрана. Чтобы упростить рис. 3.7, будем считать, что точка P имеет нулевую *y*-координату. Но все последующие уравнения для вычисления ее *y*-координаты также пригодны и для любых других значений координаты X.

На рис. 3.8 треугольники EPR и EP'Q подобны. Тогда из соотношения $P'Q/EQ = PR/ER$ получим $X/d = x/z$ или $X = d \cdot x/z$, аналогично получим $Y = d \cdot y/z$.

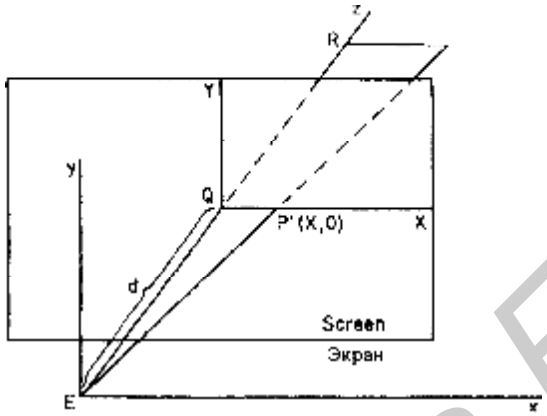


Рис. 3.7. Экран и видовые координаты

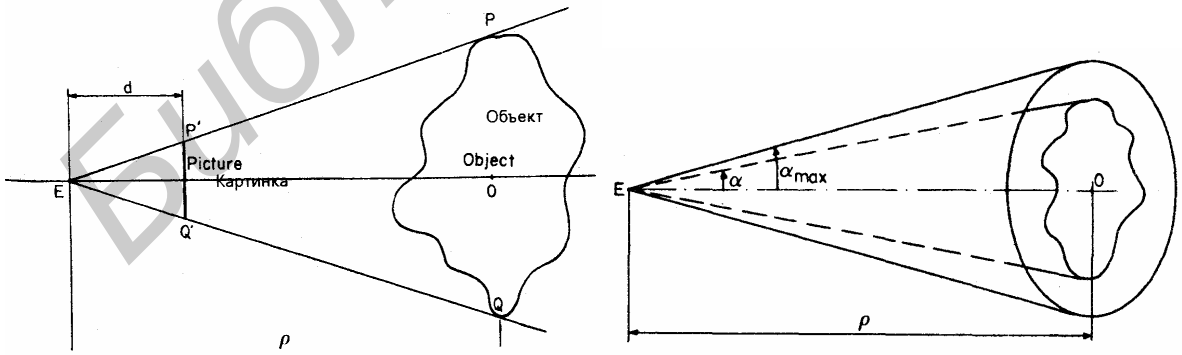
Расстояние между точкой наблюдения E и экраном ориентировочно определяется из соотношения:

$$\text{размер картинки}/d = \text{размер объекта}/r,$$

что следует из подобия треугольников EP'Q' и EPQ (рис. 3.8,а).

При вычислениях лучше ограничивать не фактическое значение угла a, показанное на рис. 3.8,б, а примерно выдерживать отношение

$$tga = 0.5 \cdot \text{размер объекта}/r.$$



а) б)

Рис. 3.8. Принцип построения перспективного изображения: а) размеры картинки и объекта, б) конус наблюдения

4. Приложение А
Пример выполнения индивидуального задания

Исходные данные

Тема: Изучение стандартных средств отображения графической информации интегрированной среды разработки C++ Builder.

Цель: Используя стандартные средства вывода графической информации среды C++ Builder, построить график кривой высшего порядка, обеспечить возможность изменения масштаба по осям X и Y, поворота осей координат на произвольный угол, смещения начала координат в любую точку относительно исходной позиции.

Вариант №5. Эпициклоиды

Направляющая кривая L – окружность радиуса b, окружность K радиуса a катится без скольжения вне ее.

В параметрической форме:

$$x = (a+b)\cos j - a\cos((a+b)j/a),$$
$$y = (a+b)\sin j - a\sin((a+b)j/a),$$
$$-\infty < j < \infty,\ j = \angle COA_1.$$

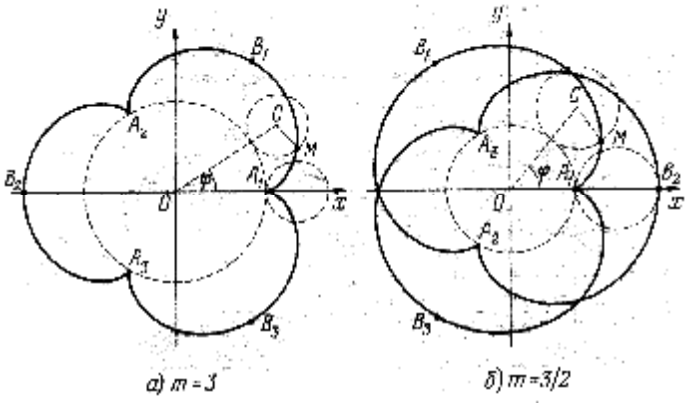
Вид кривых зависит от отношения $m = b/a$.

а) m – целое положительное число. Кривые состоят из m равных друг другу дуг, «обходящих» направляющую окружность L (а). Достаточно рассмотреть изменение j от нуля до $2p$, так как кривые далее переходят сами в себя.

б) $m = p/q$, p и q – положительные целые взаимно простые числа. Кривые состоят из p равных друг другу пересекающихся дуг (б). Кривые замкнуты. Интервал изменения параметра: $0 \le j < 2qp$.

в) если m – иррациональное, то кривые состоят из бесконечного числа равных друг другу дуг. Кривые не замкнуты. Радиус кривизны $R(j) = (4a(a+b)\sin((bj)/(2a)))/(2a+b)$, в вершинах B_k:

$$R_{B_k} = 4a(a+b)/(2a+b)$$



Реализуем данное задание в среде Borland C++ Builder 6.0. Расположим на форме компоненты, необходимые для реализации интерфейса пользователя. Необходимо предусмотреть возможность ввода параметров эпициклоиды, коэффициентов масштабирования по осям, угол поворота, а также смещение начала координат по оси X и Y.

Далее приводится исходный код программы с подробными комментариями, реализующий задание.

```
...
void __fastcall TForm1::Button1Click(TObject *Sender)
{
   float fi;
   float ScX,ScY;
   float a,b,t;
  int x,y;
  int xr,yr;
  int Xc,Yc;
  TColor Color;


  Canvas->Rectangle(0,0,ClientWidth,ClientHeight);

  a=StrToFloat(Edit1->Text);  /*вводим параметр a эпициклоиды*/
  b=StrToFloat(Edit2->Text);  /*вводим параметр b эпициклоиды*/
  fi=-M_PI*(StrToFloat(Edit3->Text)/180); /*вводим угол поворота
                                          и переводим его в радианы*/

  ScX=StrToFloat(Edit6->Text);/*вводим коэффициент масштабирования по X*/
  ScY=StrToFloat(Edit7->Text);/*вводим коэффициент масштабирования по Y*/

    /* Если выбран автоматический режим,
       то за начало координат принимается центр формы */
    if (CheckBox1->Checked)
    {
        Xc=ClientWidth/2;
        Yc=ClientHeight/2;
    }
    else { //иначе вводятся заданные пользователем координаты
        Xc=StrToInt(Edit4->Text);
        Yc=StrToInt(Edit5->Text);
    }

    /* Устанавливаем ширину линии */
    Canvas->Pen->Width=2;
    /* Выбираем цвет линии */
   switch (RadioGroup1->ItemIndex)
   { case 0 : Color = clRed;break;
     case 1 : Color = clGreen;break;
     case 2 : Color = clBlue;break;
     case 3 : Color = clBlack;break;
   }

   /* Устанавливаем цвет линии */
   Canvas->Pen->Color=Color;

   /* Рассчитываем координаты начальной точки кривой */
   t=0;
   x=(a+b)*cos(t)-a*cos((a+b)*t/a);
   y=(a+b)*sin(t)-a*sin((a+b)*t/a);

   /* Производим масштабирование осей на заданные коэффициенты и
      поворот на заданный угол */
```

```
    xr=ScX*x*cos(fi)-ScY*y*sin(fi);
    yr=ScX*x*sin(fi)+ScY*y*cos(fi);

    /* Устанавливаем перо в начальную точку */
    Canvas->MoveTo(xr+Xc,yr+Yc);

    /* В цикле рассчитываем и выводим точки кривой */
    for (t=0; t<=2*M_PI*a; t+=0.001)
    {   /*расчет координат очередной точки кривой*/
     x=((a+b)*cos(t)-a*cos((a+b)*t/a));
     y=((a+b)*sin(t)-a*sin((a+b)*t/a));
     /*масштабирование и поворот на заданный угол*/
     xr=ScX*x*cos(fi)-ScY*y*sin(fi);
     yr=ScX*x*sin(fi)+ScY*y*cos(fi);

     Form1->Canvas->LineTo(xr+Xc,yr+Yc);
    }

    /*Устанавливаем перо в черный цвет*/
    Canvas->Pen->Color=clBlack;

    /*Выводим оси координат*/
    Canvas->MoveTo(cos(fi)*ClientWidth+Xc,sin(fi)*ClientHeight+Yc);
    Canvas->LineTo(-cos(fi)*ClientWidth+Xc,-sin(fi)*ClientHeight+Yc);
    Canvas->MoveTo(-sin(fi)*ClientWidth+Xc,cos(fi)*ClientHeight+Yc);
    Canvas->LineTo(sin(fi)*ClientWidth+Xc,-cos(fi)*ClientHeight+Yc);
}
...
```

Окно приложения представлено на рис.А.1

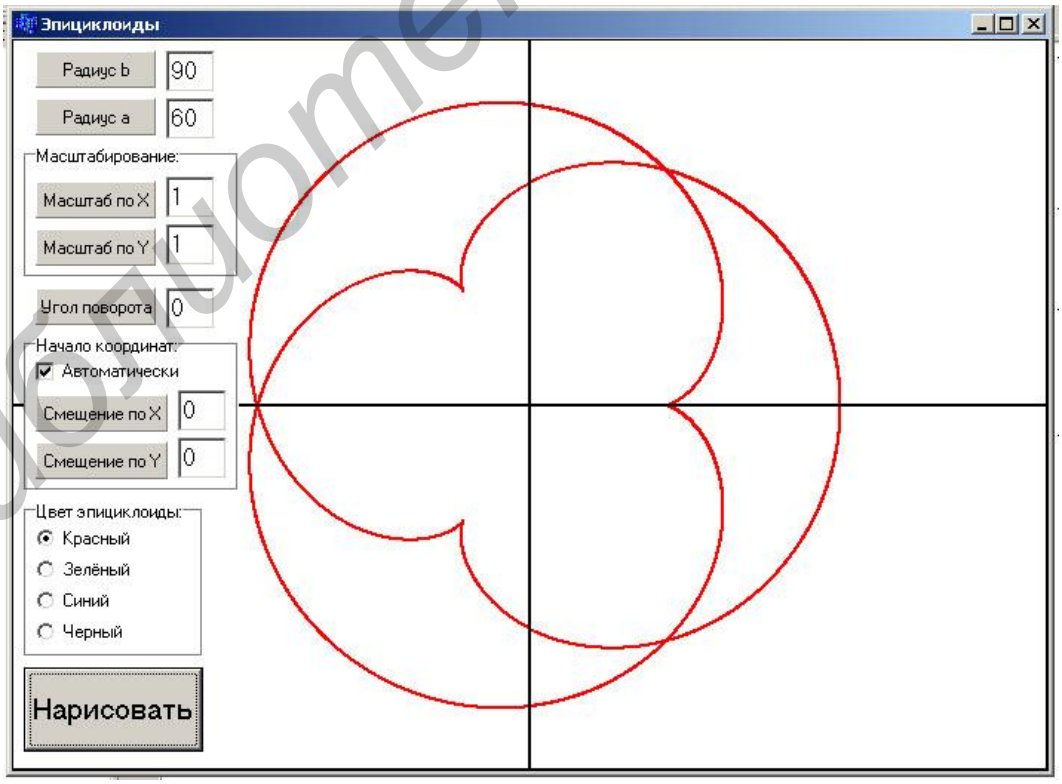


Рис. А.1. Окно приложения

Литература

1. Аммерал, Л. Интерактивная трехмерная машинная графика / Л. Аммерал. – М. : Сол Систем, 1992.
2. Аммерал, Л. Принципы программирования в машинной графике / Л. Аммерал. – М.: Сол Систем, 1992.
3. Аммерал, Л. Машинная графика на персональных компьютерах / Л. Аммерал. – М.: Сол Систем, 1992.
4. Энджел, Й. Практическое введение в машинную графику / Й. Энджел. – М. : Радио и связь, 1984.
5. Роджерс, Д. Алгоритмические основы машинной графики / Д. Роджерс. – М. : Мир, 1989.
6. Гилой, В. Интерактивная машинная графика: структуры данных, алгоритмы, языки / В. Гилой. – М. : Мир, 1981.
7. Хирн, Д. Микрокомпьютерная графика / Д. Хирн, М. Бейкер. – М. : Мир, 1987.
8. Гардан, И. Машинная графика и автоматизация конструирования / И. Гардан, М. Люка. – М. : Мир, 1987.
9. Шикин, Е. В. Компьютерная графика: динамика, реалистические изображения / Е. В. Шикин. – М. : Диалог-МИФИ, 1995.
10. Скляров, В. А. Язык С++ и объектно-ориентированное программирование / В.А. Скляров.– Минск: Высш. шк., 1997.
11. Прата, С. Язык программирования С++. Лекции и упражнения. Учебник / С. Прата; пер. с англ. – К.: Издательство «Диасофт», 2001.
12. Шикин, Е.В. Компьютерная графика: полигональные модели / Е.В. Шикин, А.В. Боресков. – М. : Диалог-МИФИ, 2000.
13. Роджерс, Д. Математические основы машинной графики / Д. Роджерс, Дж. Адамс. – М. : Мир, 2001.
14. Павлидис, Е. Алгоритмы машинной графики и обработка изображений / Е. Павлидис. – М. : Радио и связь, 1988.
15. Фоли, Дж. Основы интерактивной машинной графики / Дж. Фоли, А. Ван Дэм. – М. : Мир, 1985.
16. Фокс, А. Вычислительная геометрия / А. Фокс, М. Пратт. – М. : Мир, 1982.
17. Ватолин, Д. С. Алгоритмы сжатия изображений / Д. С. Ватолин. – М. : ВМК МГУ, 1999.
18. Блинова, Т. Компьютерная графика / Т. Блинова. – М. : Юниор, 2006, 520 с.
19. Херн, Д. Компьютерная графика и стандарт OpenGL / Д. Херн. – М. : Вильямс, 2005, 1168 с.
20. Пирогов, В. Компьютерная графика: учеб. пособие / В. Пирогов. – К. : BHV, 2002, 432 с.
21. Никулин, Е. Компьютерная геометрия и инструменты машинной графики / Е. Никулин.– К. : BHV, 2003, 560 с.

Св. план 2008, поз. 19

Учебное издание

Лившиц Михаил Зенадьевич,
Лихачев Денис Сергеевич,
Петровский Александр Александрович и др.

АЛГОРИТМИЧЕСКИЕ ОСНОВЫ КОМПЬЮТЕРНОЙ ГРАФИКИ

МЕТОДИЧЕСКОЕ ПОСОБИЕ

по курсу
«Алгоритмические основы компьютерной графики»
для студентов специальности I – 40 02 02
«Электронные вычислительные средства»
дневной формы обучения

Редактор Т.П. Андрейченко
Корректор

Подписано в печать	Формат 60x84 1/16.	Бумага офсетная.
Гарнитура «Таймс»	Печать ризографическая.	Усл. печ. л.
Уч.- изд. л. 2,0	Тираж 100 экз.	Заказ

Издатель и полиграфическое исполнение: Учреждение образования
«Белорусский государственный университет информатики и радиоэлектроники»
ЛИ №02330/0056964 от 01.04. 2004. ЛП №02330/0131518 от 30.04. 2004.
220013, Минск, П. Бровки, 6.