Волгоградский государственный университет
Кафедра компьютерных наук и экспериментальной математики

Компьютерная графика. Часть I.

(краткий конспект лекций)

Е.Г.Григорьева

Волгоград 2011

Глава 1

Растровые развертки линейных объектов

1.1 Дискретная природа растровой плоскости

Графические устройства делятся на векторные и растровые. Векторные устройства (такие как графопостроители) представляют изображения в виде линейных объектов. Перо графопостроителя перемещается над бумагой к определенному месту, задаваемому компьютером. затем опускается и перемещается к другой точке оставляя след определенного цвета. Для вычерчивания линий применяются также векторные видеодисплеи. Такие устройства имеют встроенные схемы предназаначенные для развертки электронного луча, оставляющего светящийся след перемещаясь от точки к точке экрана катодно лучевой трубки. Поскольку каждый отрезок линии, подлежащий изображению занимает очень мало памяти, то векторные дисплеи могуть рисовать объекты очень быстро (тысячи векторов в секунду). Однако векторные дисплеи не могут отображать изображения с плавными переходми цветов и сканированные изображения. Закрашивание области имитируется штриховкой с различными узорами линий. Сегодня векторные диспления почти всюду вытеснены растровыми.

На большинстве ЭВМ принят растровый способ изображения графической информации. Растровые устройства имеют поверхность, на которой они формируют изображение. На этой поверхности имеется определенное количество точек (пикселей), например поверхность имеет 480 строк по 640 пикселей на строку. Т.о. изображение представляет собой прямоугольную матрицу точек, каждая из которых "окрашена"в определенный цвет, взятый из специально-

го набора – палитры цветов. для реализации этого подхода компьютер снабжается видеадаптером (видеокартой), который хранит в своей внутренней памяти изображение, отводя на каждый пиксель определенное количество бит (глубина цвета), определяющие его цвет. Кроме этого, видеадаптер периодически (50-70 раз в секунду) обеспечивает отображение изображения на монитор компьютера. Размер палитры определяется размером видеопамяти, отводимой на 1 пиксель и режимом работы видеоадаптера.

Во всех растровых дисплеях имеется встроенная система координат, которая устанавливает соответствие между заданным пикселем и физическими координатами на поверхности отображения.

Графическая программа помещается в оперативную память, которая выполняется центральным процессором. Программа вычисляет значения цвета для каждого пикселя и загружает эти значения в буфер кадра. Контроллер развертки заставляет буфер кадра пиксель за пикселем пропускать через конвертер, предназначенный для формирования светового "пятна"на поверхности изображения. Фактически любая графическая операция сводится к работе с отдельными пикселями – поставить точку с заданными координатами определенного цвета и узнать цвет точки с заданными координатами. Однако большинство бибилиотек поддерживают работу с больш**и**м набором графических объектов. Среди них выделяются такие группы:

- линейные изображения (растровые развертки линий),
- сплошные объекты (растровые образы двумерных областей),
- шрифты (растровые представления символов),
- изображения (битовые карты, прямоугольные матрицы пикселей)

Поскольку большинство графических устройств являются растровыми, то соответствующие графические бибилиотеки поддерживают достаточное количество растровых алгоритмов: растровые развертки графических примитивов (отрезков, окружностей, дуг эллипсов, кривых Безье и т.п.), алгоритмы обработки изображений.

Важным понятием для растровой сетки является понятие связности, определяющее близость точек растра. Различают два вида связности: 4-связность и 8-связность. В первом случае точки (x_1, y_1) и (x_2, y_2) называются соседними, если

$$|x_1 - x_2| + |y_1 - y_2| \leq 1,$$

а во втром случае – если

$$|x_1 - x_2| \leq 1, \quad |y_1 - y_2| \leq 1.$$

При этом растровой линией будет называться набор пикселей $P_1, P_2, ..., P_n$, в котором для всякого $i = 1, 2, ...n - 1$ точки P_i, P_{i+1} будут соседними.

1.2 Алгоритм Брезенхейма

Предположим, что нам требуется построить прямолинейный отрезок, соединяющий точки с целочислеными координатами $A = (x_a, y_a)$ и $B = (x_b, y - b)$. для простоты изложения будем предполагать выполненным условие

$$0 \leq y_b - y_a \leq x_b - x_a.$$

Тогда уравнение прямой проходящей через точки A и B будет иметь вид

$$y = y_a + \frac{y_b - y_a}{x_b - x_a}(x - x_a),$$

при этом точки требуемого отрезка удовлетворяют условию

$$x_a \leq x \leq x_b.$$

Обозначим через

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

и через

$$b = y_a - kx_a,$$

тогда уравнение отрезка примет вид

$$y = kx + b, \quad x \in [x_a, x_b].$$

простой алгоритм, представления этого отрезка состоит в том, чтобы двигаться по x от x_a до x_b с единичным шагом и на каждом шаге округлять значение y до его целой части.

```
void line(int xa,int xb,int ya,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);
}

}
```

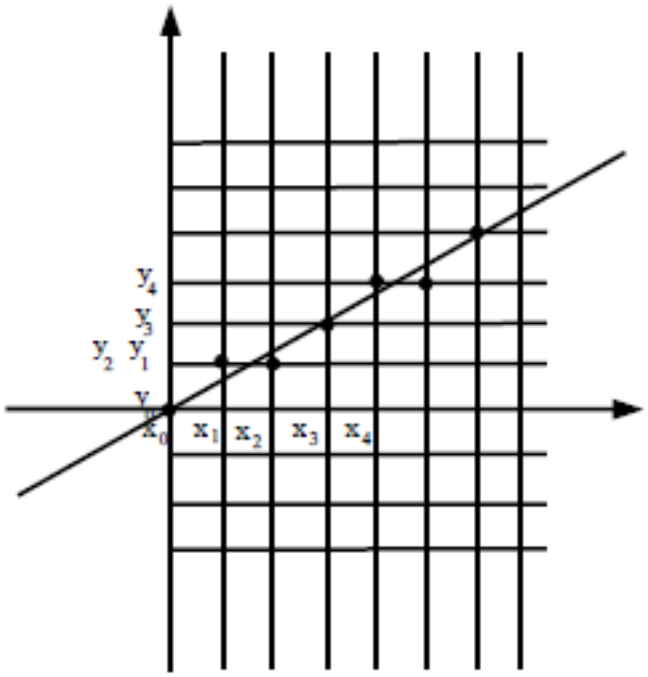
Можно избавиться от вычисления значения функции $y = kx+b$ используя рекуррентное соотношение: при изменении переменной x на 1, значение y изменяется на k. Так, что мы получим

```
void line(int xa,int xb,int ya,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 привести к не всегда корректному изображению. Улучшить внешний вид получаемого отрезка можно за счет округления значения y до ближайшего целого. Фактически это значит, что из двух возможных пикселей , между которыми проходит прямая выбирается ближайший. Что бы вычислить этот пиксель необходимо дробную часть значения y сравнить с $1/2$.

Пусть $(x_0 = x_a, y_0 = y_b), (x_1, y_1), ..., (x_n = x_b, y_n = y_b)$ – последовательность изображаемых пикселей, причем $x_{i+1} - x_i = 1$. Тогда каждому значению x_i соответствует число $kx_i + b$. Введем величины $c_i = kx_i + b - y_i$. Давайте посмотрим каким образом изменяется величина c_i при переходе от писксела к соседнему пикселу. Если $c_i + k \leq 1/2$, то должно быть $c_{i+1} = c_i + k$ и, следовательно, $y_{i+1} = y_i$. Если же $c_i + k > 1/2$, то $y_{i+1} = y_i$ и потому $c_{i+1} = c_i + k - 1$. Заметим, что $c_0 = 0$. Теперь мы можем записать

Рис. 1.1: Вычисление $y_0, y_1, y_2, ...$




```
void line(int xa,int xb,int ya,int yb, int color)
{
double k=((double)(yb-ya)/(xb-xa));
double y=ya;
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);
}

}
```

Для улучшения кода, заметим, что операция сравнения с целым нулем будет быстрее, чем сравнение с плавающим значением 0.5. С этой целью введем величину $d_i = 2c_i - 1$. Замечая, что $c_i > 0.5$ тогда и только тогда когда $d_i > 0$ и, что $d_1 = 2k - 1$, приходим к коду

```
void line(int xa,int xb,int ya,int yb, int color)
{
double k=((double)(yb-ya)/(xb-xa));
double y=ya;
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);
}

}
```

Теперь остается привести все переменные к целочисленному типу. Для этого заметим, что в программе мы имеем дело со значениями вида $p/\Delta x, p \in \mathbb{Z}$. Умножая такие значения на Δx мы получим целочисленный код. Мы введем

две величины d_1, d_2, которые будут задавать приращение величины d в зависимости от знака d. Так, что окончательно получим реализацию алгоритма Брезенхейма

```
void line(int xa,int xb,int ya,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;x<=xb;x++)
{
    if(d>0)
    {
        d+=d2;
        y++;
    }else
    {
        d+=d1;
    }
    putpixel(x,y,color);
}

}
```

1.3 Как нарисовать окружность

Для растеризации окружности мы воспользуемся ее симметрией. Пусть требуется построить окружность, заданную уравнением

$$x^2 + y^2 = R^2,$$

где R – целое, положительное число. Такая окружность симметрична относительно координатных осей и прямых $y = \pm x$. Так, что достаточно найти

пикселы одной восьмой част окружности и по ним восстановить остальные. Мы будем работать с частью окружности, координаты точек которой удовлетворяют неравенствам

$$x \geq 0,\ y \geq 0\ x \leq y.$$

Введем в расмотрение функцию $F(x,y) = x^2 + y^2 - R^2$. Эта функция обращается в нуль в точках рассматриваемой окружности. Она отрицательна внутри окружности и положительна во внешней области. так же как и в алгоритме Брезенхейма, мы обозначим через $(x_0, y_0), ..., (x_n, y_n)$ – пикселы, которые мы должны построить. Мы предполагаем, что $x_0 = 0, y_0 = R$ и $x_0 < x_1 < x_2 < ..., x_n$. Для определения значения y_{i+1} мы введем величины

$$d_i = F(x_i + 1, y_i - 0.5).$$

В случае, когда $d_i < 0$ средняя точка попадает внутрь окружности, поэтому мы полагаем $y_{i+1} = y_i$ и тогда

$$d_{i+1} = F(x_i + 2, y_i - 0.5) = F(x_i + 1, y_i - 0.5) + 2x_i + 3.$$

Если же $d_i \geq 0$, то $y_{i+1} = y_i - 1$, а, значит,

$$d_{i+1} = F(x_i + 2, y_i - 1.5) = F(x_i + 1, y_i - 0.5) + 2(x_i - y_i) + 5.$$

А поскольку,

$$d_0 = F(1, R - 0.5) = 5/4 - R,$$

приходим к такой реализации

```
int x=0;
int y=R;
double d=1.25-R;
putpixel(x,y,color);
while(x<y)
{
   if(d<0)
   {
      d+=2*x+3;
      x++;
   }
   else
   {
      d+=2*(x-y)+5;
```

```
      x++;
      y--;
   }
   putpixel(x,y,color);
}
```

Заметим, что величина d имеет вид $0.25 + z,\ z \in \mathbb{Z}$. Причем $d > 0$ если и только если $z > 0$. Поэтому реализация получается такой

```
int x=0;
int y=R;
int d=1-R;
int delta1=3;
int delta2=-2*R+5;

while(x<y)
{
   if(d<0)
   {
      d+=delta1;
      delta1+=2;
      delta2+=2;
      x++;
   }
   else
   {
      d+=delta2;
      delta2+=4;
      delta1+=2;
      x++;
      y--;
   }
   putpixel(x,y,color);
}
```

Величины delta1, delta2 используются для вычисления приращения значения d на каждой итерации основного цикла в зависимости от ее знака.

1.4 Растровая развертка эллипса

Уравнение эллипса с осями, параллельными координатным осям имеет вид

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

Это уравнение можно переписать и в таком виде

$$F(x, y) = b^2x^2 + a^2y^2 - a^2b^2 = 0.$$

мы реализуем алгоритм построения эллипса только для той его части, которая лежит в первом квадранте: для $x \geq 0,\ y \geq 0$. Эту часть эллипса разобьем на две части. В первую часть войдут те точки, в которых угол между касательной прямой и осью Ox не превосходит $\pi/4$, а во вторую часть – остальные точки. Учитывая, что

$$\nabla F(x, y) = (2b^2x, 2a^y),$$

находим, что в первой части эллипса выполнено $b^x < a^2y$, а во второй – обратное неравенство. Условие перехода из первой части эллипса во вторую сформулируется так: если в срединной точке:

$$a^2(y_i - \frac{1}{2}) \leq b^2(x_i + 1),$$

то на следующем шаге алгоритма переходим во вторую часть. Для рисования эллипса мы воспользуемся теми же соображениями, что и при рисовании окружности. Так, например в первой части эллипса мы будем вычислять значение функции $F(x, y)$ в средней точке

$$d_i = F(x_i + 1, y_i - 0.5) = b^2(x_i + 1)^2 + a^2(y_i - 0.5)^2 - a^2b^2.$$

При $d_i < 0$ полагаем $y_{i+1} = y_i$. При этом

$$d_{i+1} - d_i = b^2(2x_i + 3).$$

При $d_i \geq 0$ полагаем $y_{i+1} = y_i - 1$ и тогда

$$d_{i+1} - d_i = b^2(2x_i + 3) + a^2(2 - 2y_i).$$

Глава 2

Алгоритмы отсечения

2.1 Взаимное расположение геометрических фигур

2.1.1 Определение пересечения двух отрезков

Указанная в заглавии задача формулируется так: даны два отрезка прямых на плоскости, требуется определеить пересекаются ли они и если пересекаются, то найти точку пересечения.

Пусть AB, CD – заданные отрезки. Содержащие их прямые могут быть заданы уравнениями

$$l(t) = A + tb, \quad b = B - A, \quad t \in [0, 1],$$

$$p(s) = C + sd, \quad d = D - C, \quad s \in [0, 1].$$

Положим $c = C - A$, тогда для нахождения пересечения прямых получим соотношение

$$bt = c + ds.$$

Умножим это равенство на вектор $d^{\perp}$, ортогональный вектору d

$$\langle b, d^{\perp} \rangle t = \langle c, d^{\perp} \rangle.$$

Таким образом, если $\langle b, d^{\perp} \rangle \neq 0$, то

$$t^* = \frac{\langle c, d^{\perp} \rangle}{\langle b, d^{\perp} \rangle}.$$

Аналогично

$$s^* \frac{\langle c, b^\perp \rangle}{\langle d, b^\perp \rangle}.$$

Отрезки будут пересекаться тогда и только тогда, когда

$$t^* \in [0,1], \quad s^* \in [0,1].$$

При этом точка пересечения может быть найдена по формуле

$$A + \frac{\langle c, d^\perp \rangle}{\langle b, d^\perp \rangle} b.$$

В случае, когда $\langle b, d^\perp \rangle = 0$ прямые, содержащие данные отрезки параллельны. В этом случае необходимо выяснить совпадают ли эти прямые и пересекаются ли отрезки AB, CD на этой прямой. В первом случае нужно проверить, например, принадлежит ли точка C прямой $l(t)$. Для этого, нужно координаты этой точки подставить в уравнение прямой

$$b_x(y - A_x) - b_y(x - A_y) = 0.$$

При условии совпадения прямых, выразим C, D через A, B

$$C = A + bt_c \quad \Rightarrow t_c = \frac{\langle C - A, b \rangle}{|b|^2},$$

$$D = A + bt_d \quad \Rightarrow t_d = \frac{\langle D - A, b \rangle}{|b|^2}.$$

Отрезки не пересекаются, если $t_c, t_d < 0$ или $t_c, t_d > 1$.

2.2 Алгоритм Сазерленда - Кохена

Постановка задачи следующая. Задан на плоскости прямоугольник $[X1, Y1] \times [X2, Y2]$. Для каждого отрезка AB требуется определить какая его часть пересекается с заданным прямоугольником.

С этой целью определим величину *code*. Положим

$$code = 0x00,\ code = 0x01,\ code = 0x02,\ code = 0x04,\ code = 0x08,$$

в зависимости от того, если точка раполагается внутри, слева сверху,снизу или вправа от прямоугольник а соответственно. Для ее вычисления реализуем процедуру

```
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;
}
```

Дополнительно нам понадобится функция обмена данными двух целочисленных переменных.

```
void swap(int & a, int & b)
{
int c;
c=a;
a=b;
b=c;

}
```

Основная идея алгоритма состоит в том, что бы отбросить очевидные случаи попадения концов отрезка в выпуклые области определяемые значением кода. Приведем его реализацию.

```
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 outside=(code1|code2)==0;
int inside = (code1&code2)!=0;

while(!outside && !inside){
   if(code1==0){
      swap(x1,x2);
```

```
      swap(y1,y2);
      swap(code1,code2);
   }
   if(code1&0x01)
   {
      y1+=(y2-y1)*(X1-x1)/(x2-x1);
      x1=X1;
   }
   if(code1&0x02)
   {
      x1+=(x2-x1)*(Y1-y1)/(y2-y1);
      y1=Y1;
   }
   if(code1&0x04)
   {
      y1+=(y2-y1)*(X2-x1)/(x2-x1);
      x1=X2;
   }
   if(code1&0x08)
   {
      x1+=(x2-x1)*(Y2-y1)/(y2-y1);
      y1=Y2;
   }
code1=outCode(x1,y1,X1,Y1,X2,Y2);
code2=outCode(x2,y2,X1,Y1,X2,Y2);
outside=(code1|code2)==0;
inside = (code1&code2)!=0;
}
line(x1,y1,x2,y2);
}
```

2.3 Алгоритм Сайруса-Бека

Здесь рассматривается задача отсечения отрезка выпуклым многоугольником на плоскости. Рассмотрим луч AC и некоторую прямую. Пусть точка B зафиксирована на прямой и точка P может пробегать всю прямую. Обозна-

чим через $\vec{n}$ – вектор нормали к этой прямой. Тогда имеет место равенство

$$\langle P - B, \vec{n} \rangle = 0.$$

Пусть $\vec{c} = C - A$. Тогда точку пересечения прямой и луча можно найти из уравнения

$$\langle \vec{n}, A + \vec{c}t - B \rangle = 0.$$

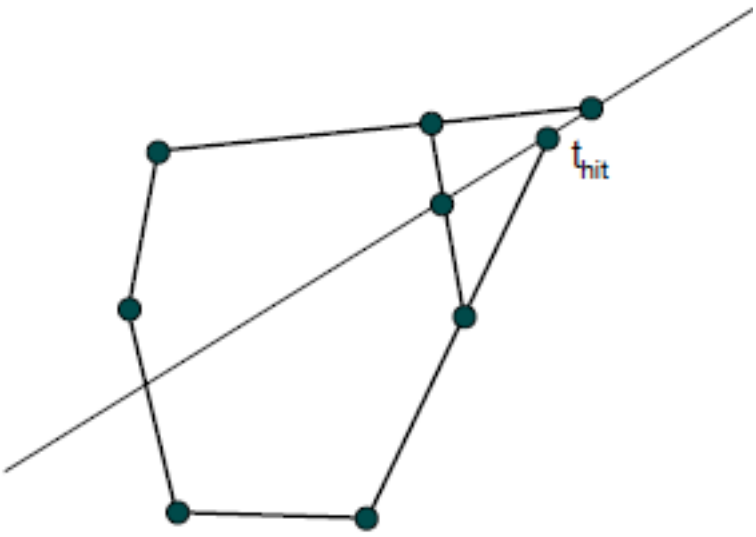
Откуда

$$t_{hit} = \frac{\langle \vec{n}, B - A \rangle}{\langle \vec{n}, \vec{c} \rangle}$$

при условии, что $\langle \vec{n}, \vec{c} \rangle \neq 0$. В случае равенства нулю этого скалярного произведения прямая и луч будут параллельными. Таким образом точка пересечения находится по формуле

$$P_{hit} = A + \vec{c}\frac{\langle \vec{n}, B - A \rangle}{\langle \vec{n}, \vec{c} \rangle}.$$

Рис. 2.1: Алгоритм Сайруса-Бека



Заметим также, что при $\langle \vec{n}, \vec{c} \rangle > 0$ луч при пересечении прямой будет выходить из полуплоскости, противоположной той, в которую направлена нормаль $\vec{n}$. В случае $\langle \vec{n}, \vec{c} \rangle > 0$ луч входит в такую полуплоскость. Пусть на плоскости задан выпуклы многоугольник M своими вершинами $P_0, P_1, ..., P_n, P_{n+1} = P_0$ пронумерованными по ходу часовой стрелки. Пусть $\vec{n_i}$ – вектор внешней нормали к стороне $P_i P_{i+1}$. Положим

$$\Pi_i = \{p : \ \langle p - p_i, \vec{n_i} \rangle < 0\}.$$

Тогда

$$M = \cap_{i=0}^{n} \Pi_i.$$

Заметим, что поскольку многоугольник выпуклый, то луч его пересекает не более чем в двух точках. Предположим, что t_{in}, t_{out} это те значения параметра на луче прикоторых он входит в многоугольник и выходит из него. Допускается случай $t_{in} = t_{out}$. Поскольку точке A соответсвует значение параметра равное 0, а точке C – значение 1, то результатом отсечения является отрезок луча соответствующий промежутку

$$[\max(0, t_{in}), \min(t_{out}, 1)].$$

Для нахождения значений t_{in}, t_{out} будем строить два массива. Перебирая стороны многоугольника M точку t_{hit} пересечния со стороной будем записываь в первый массив, если $\langle \vec{n_i}, \vec{c} \rangle < 0$, а во второй массив, если $\langle \vec{n_i}, \vec{c} \rangle 0$. Из выпуклости многоугольника получаем, что t_{in} представляет собой максимум элементов второго массива, а t_{out} – минимум элементов первого массива. Псевдокод алгоритма будет иметь вид

```
float tin=0;
float tout=1;
for(int i=0;i<=n;i++){
   float d=ni*c;

    thit=ni*(B-A)/d;
   if(d>0 && thit>tin)
   {
     tin=thit;
   }
   if(d<0 && thit<tout)
   {
     tout=thit;
```

```
   }
}
```

В результате работы этого кода отрезок параметров $[tin, tout]$ будет соответствовать отрезку отсечения.

2.4 Отсечение произвольными многоугольниками

В случае невыпуклого многоугольника луч многократно может входить внутрь многоугольника и выходить из него. Поэтому результатом отсечения будет массив отрезков, находящихся внутри заданого многоугольника. Пусть P_i – вершины многоугольника. Тогда i-ю сторону его можно задать параметрически

$$P_i + ue_i,\ e_i = P_{i+1} - P_i,\ u \in [0,1].$$

Как и выше точку пересечения луча со стороной P_iP_{i+1} находим из соотношения

$$A + t\vec{c} = P_i + ue_i.$$

Положим $b_i = P_i - A$. Получим

$$\vec{c}t = b_i + e_iu.$$

Обозначим через $v^\perp$ вектор, ортогональный вектору v. Тогда

$$t = \frac{\langle e_i^\perp, b_i\rangle}{\langle e_i, \vec{c}\rangle}, \quad u = \frac{\langle \vec{c}^\perp, b_i\rangle}{\langle e_i, \vec{c}\rangle}.$$

Теперь будем составлять список $hitList$ значений параметра t соответствующих точкам пересечения луча и стороны многоугольника. Псевдокод выглядит так

```
for(int i=0; i<N; i++)
{
   //вычисляем векторы b_i,e_i для i-го ребра
   //находим соотвествующие t,u
   if(u>=0 && u<=1)
   {
      //добавляем t  к списку hitList
   }
}
```

Пусть $hitList = (t_1, t_2, ..., t_m)$ отсортированный массив по возрастанию значений. Тогда результатом отсечения будут отрезки соответствующие интервалам

$$[\max(0, t_1), t_2], [t_3, t_4], ..., [t_{m-1}, \min(1, t_m)].$$

2.5 Алгоритм Сазерленда-Ходжмана

Здесь рассматривается задача отсечения выпуклым многоугольником произвольного многоугольника. Подобно алгоритму Сайруса-Бека, данные алгоритм отсекает полигон S относительно каждой ограничивающей полигон C прямой поочередно оставляя только ту часть, которая находится внутри C.

Рассмотрим работу этого алгоритма на примере 7-угольника S и прямоугольника C. Пусть a, b, c, d, e, f, g – суть вершины S. S отсекается относительно верхнего, правого, нижнего и левого ребер C поочередно, и на каждой стадии из старого списка вершин генерируется новый список. Этот список описывает несколько полигонов и выступает в качестве отсекаемого полигона для отсечения относительно следующего ребра полигона C. Т.о. основная операция состоит в отсечении полигонов, описанных списком вершин, относительно текущего ребра и в создании выходного списка вершин. Мы просматриваем входной список, формируя последовательно ребра из смежных пар вершин.

Каждое такое ребро E имеет имеет первую и вторую концевые точки s и p. Существует 4 возможные ситуации:

1. s и p располагаются во внутренней полуплоскости отсекающего ребра,
2. s и p располагаются во внешней полуплоскости отсекающего ребра,

3,4. s и p располагаются по разные стороны от отсекающего ребра.

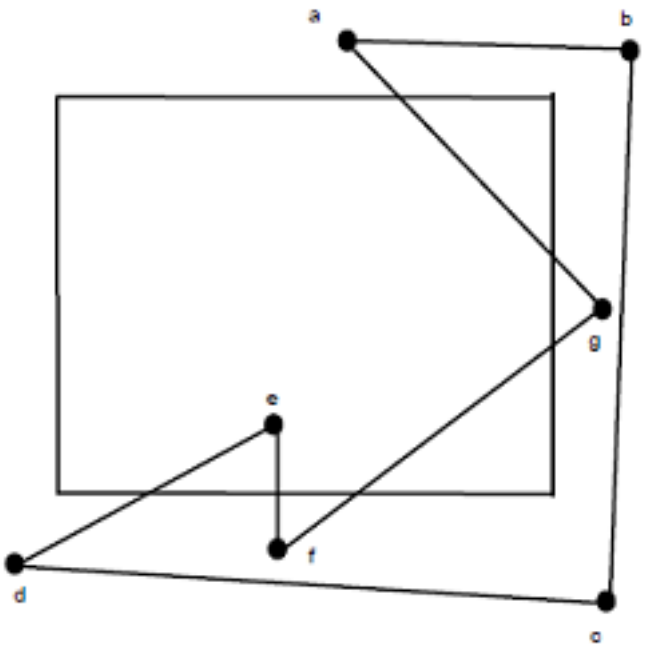
В каждом из этих случаев соответствующие точки выводятся в новый список вершин

а. s и p располагаются во внутренней полуплоскости отсекающего ребра – выводится p,

б. s и p располагаются во внешней полуплоскости отсекающего ребра – не выводится ничего,

в. s – внутри и p – снаружи от отсекающего ребра – выводится точка пересечения с ребром,

Рис. 2.2: Алгоритм Сазерленда - Ходжмана

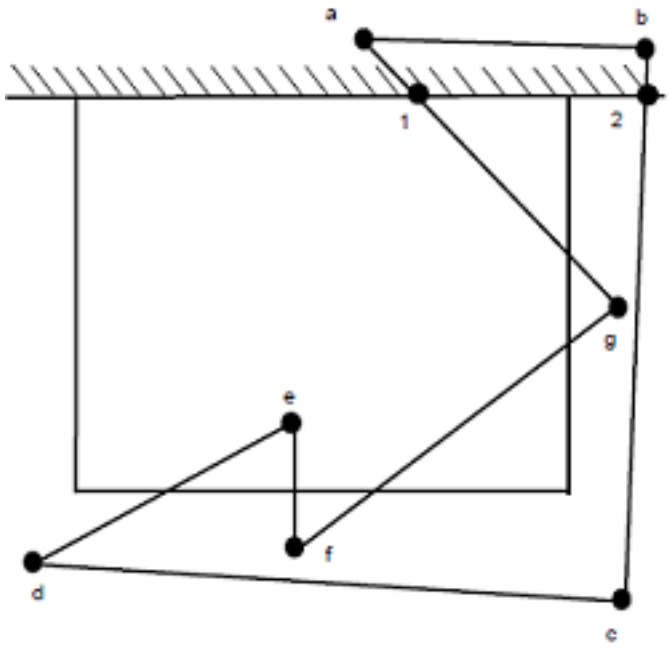


г. p – внутри и s – снаружи от отсекающего ребра – выводится точка пересечения с ребром.

Теперь обратимся к нашему примеру. Первоначально список вершин выглядит так (a, b, c, d, e, f, g). Начинаем с отсечения верхним ребром полигона C. Первым ребром рассмотрим ребро g, a (от g к a!) поскольку оно соединяет последнюю вершину и первую в списке. Полагаем $s = g$, $p = a$. Это ребро пересекает отсекающее ребро в точке 1, которое выводится в новый список. Следующим ребром будет ребро a, b. Поскольку обе точки находятся выше отсекающего ребра, то в новый список не выводится ничего. Третье ребро на выходе дает сразу две точки – точку 2, пересечения с прямой отсекающего ребра и точку c. Четвертое ребро дает одну вершину d. Поступаем таким образом получаем список $(1, 2, c, d, e, f, g)$.

На втором шаге входным списком будет $(1, 2, c, d, e, f, g)$. На выходе полу-

Рис. 2.3: Алгоритм Сазерленда - Ходжмана. Шаг 1.



чаем новый список $(3145def6)$. Получается он таким образом

$$(g,1) \rightarrow \text{ это ребро дает новые вершины } 3,1,$$

$$(1,2) \rightarrow \text{ это ребро дает новую вершину } 4,$$

$$(2,c) \rightarrow \text{ это рбро ничего не дает,}$$

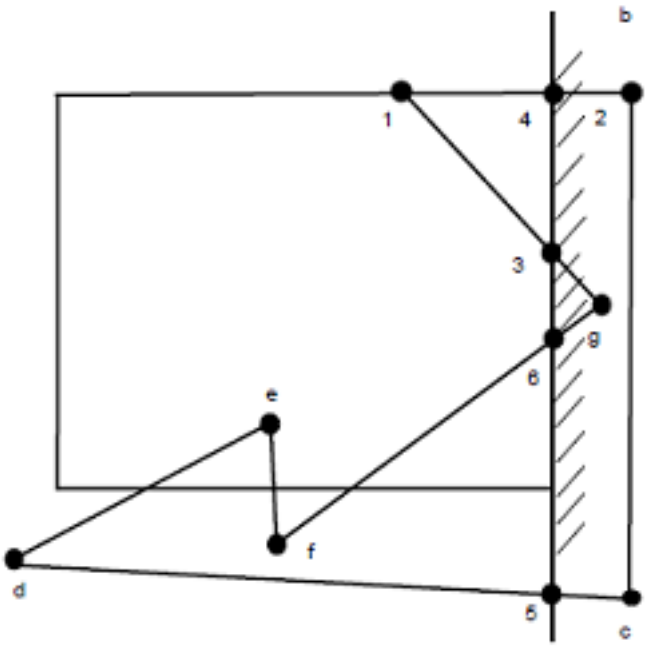
$$(c,d) \rightarrow \text{ это рбро дает вершины } 5,d,$$

$$(d,e) \rightarrow \text{ это ребро дает новую вершину } e,$$

$$(e,f) \rightarrow \text{ это ребро дает новую вершину } f,$$

$$(f,g) \rightarrow \text{ это ребро дает новую вершину } 6.$$

Рис. 2.4: Алгоритм Сазерленда - Ходжмана. Шаг 2.

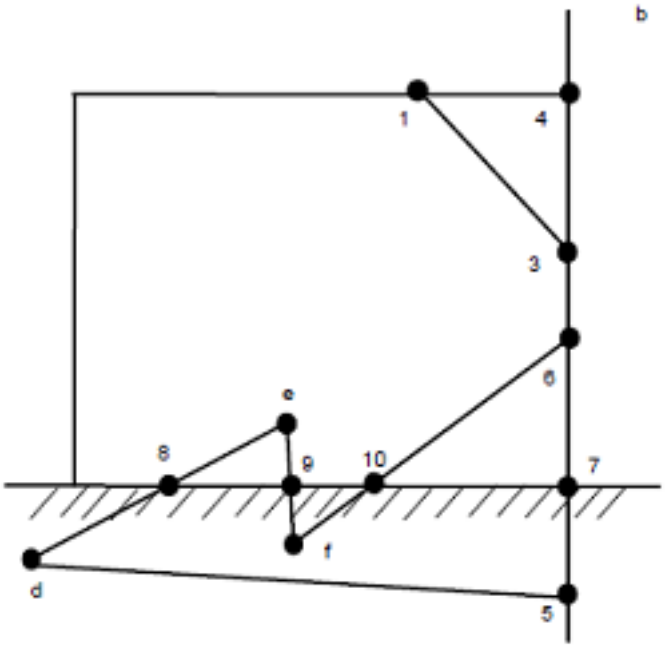


На третьем шаге мы получаем следующий список $(3, 1, 4, 7, 8, d, e, 9, 10, 6)$. Получаем это так

$$(, 3) \rightarrow \text{ это ребро дает новую вершину } 3,$$

$$(3, 1) \rightarrow \text{ это ребро дает новую вершину } 1,$$
$$(1, 4) \rightarrow \text{ это ребро дает новую вершину } 4,$$
$$(4, 5) \rightarrow \text{ это ребро дает новую вершину } 7,$$
$$(5, d) \rightarrow \text{ это ребро не дает новых вершин },$$
$$(d, e) \rightarrow \text{ это ребро дает новые вершины } 8, e,$$
$$(e, f) \rightarrow \text{ это ребро дает новую вершину } 9,$$
$$(f, 6) \rightarrow \text{ это ребро дает новые вершины } 10, 6.$$

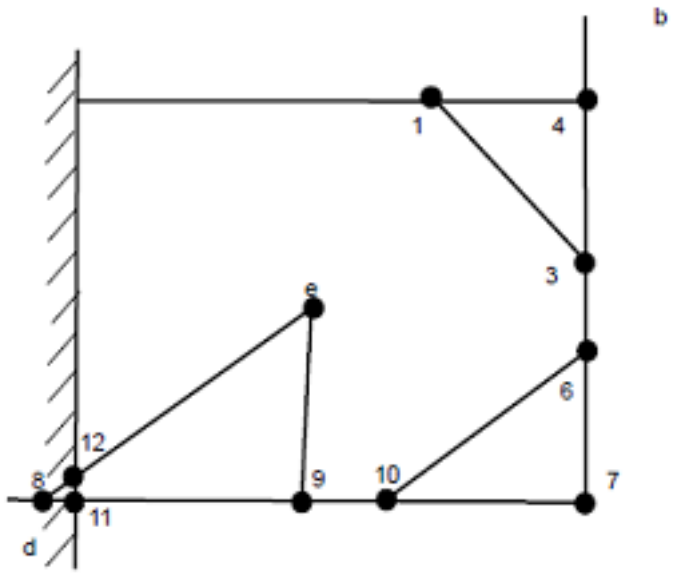
Рис. 2.5: Алгоритм Сазерленда - Ходжмана. Шаг 3.



Наконец на последнем шаге мы получаем следующий список $(3, 1, 4, 7, 11, 12, e, 9, 10,$
Получаем это так

$$(3,1) \rightarrow \text{ это ребро дает новую вершину } 1,$$
$$(1,4) \rightarrow \text{ это ребро дает новую вершину } 4,$$
$$(4,7) \rightarrow \text{ это ребро дает новую вершину } 7,$$
$$(7,8) \rightarrow \text{ это ребро дает новую вершину } 11,$$
$$(8,e) \rightarrow \text{ это ребро дает новые вершины } 12, e$$
$$(e,9) \rightarrow \text{ это ребро дает новую вершину } 9,$$
$$(9,10) \rightarrow \text{ это ребро дает новую вершину } 10,$$
$$(10,6) \rightarrow \text{ это ребро дает новые вершины } 6.$$

Рис. 2.6: Алгоритм Сазерленда - Ходжмана. Шаг 4.



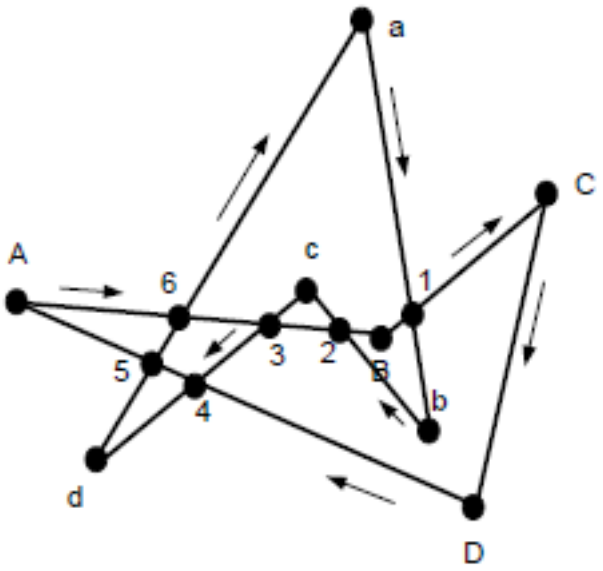
2.6 Отсечение одного полигона границей другогго. Алгоритм Вейлера - Азертона

Отсечение Вейлера-Азертона является наиболее общим механизмом отсечения из всех, которые мы изучали. Оно отсекает произвольный полигон границей любого отсекающего полигона. Эти полигоны могут содержать отверстия.

Алгоритм Сазерленда-Ходжмана использует выпуклость отсекающего полигона так как оперирует его внешними и внутренними полуплоскостями для его ребер. Однако в задачах визуализации чаще всего приходится иметь дело с отсечениями, в которых присутствуют невыпуклые полигоны.

Начнем с примера. Пусть два невыпуклых полигона $SUBJ$ и $CLIP$ заданы списками вершин (a, b, c, d) и (A, B, C, D) соответственно. Причем порядок обхода вершинсоответствует обходу по часовой стрелке – при обходе вдоль границы внутренняя область полигона остается справа.

Рис. 2.7: Отсекающий и отсекаемый полигоны



Все точки пересечения двух полигонов определяются и помещаются в специальный список. В данном примере таких пересечений 6.

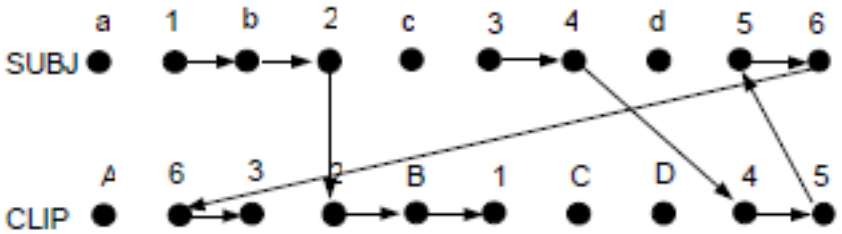
Для отсечения полигона $SUBJ$ полигоном $CLIP$ будем идти по полигону $SUBLJ$ в прямом направлении до тех пор пока не будет найдено первое "входящее" пересечние. Так мы находим точку 1 и она заносится в выходной список, в котором будет хранится информация отсечения полигона. Продолжая движение по $SUBJ$ мы найдем следующее пересечение – точку 2. Теперь в прмом направлении будем двигаться вдоль полигона $CLIP$. Проделывая такие переходы, мы в конце концов опять окажемся в точке 1. В этот момент список вывода будет таким $(1, b, 2, B)$. Далее проверяется нет ли еще каких-либо пересечний. Мы находим точку 3. Поступая как и выше приходим к еще одному списку вывода $(3, 4, 5, 6)$. Таким образом мы нашли два полигона – результат отсечения. Способ реализации основан на проходе двух списков, полученных при обходе каждого из полигонов по часовой стрелке и включением в список точек пересечения

$$SUBJ : a, 1, b, 2, c, 3, 4, d, 5, 6$$

$$CLIP : A, 6, 3, 2, B, 1, C, D, 4, 5.$$

Таким образом проход вдоль полигона соответствует перебору элементов списка, а перескакивание с одного полигона на другой соответствует смене просматриваемого списка в элементах обозначенны цифрами (точки пересечения полигонов). На рисунке показаны все такие переходы.

Рис. 2.8: Обход списков



Совершенно также обрабатываются полигоны с отверстиями. Границы отверстий ориентриуются также как и внешняя граница – при обходе полигон должен оставаться справа.

Например, рассмотрим задачу отсечения показанную на рисунке.

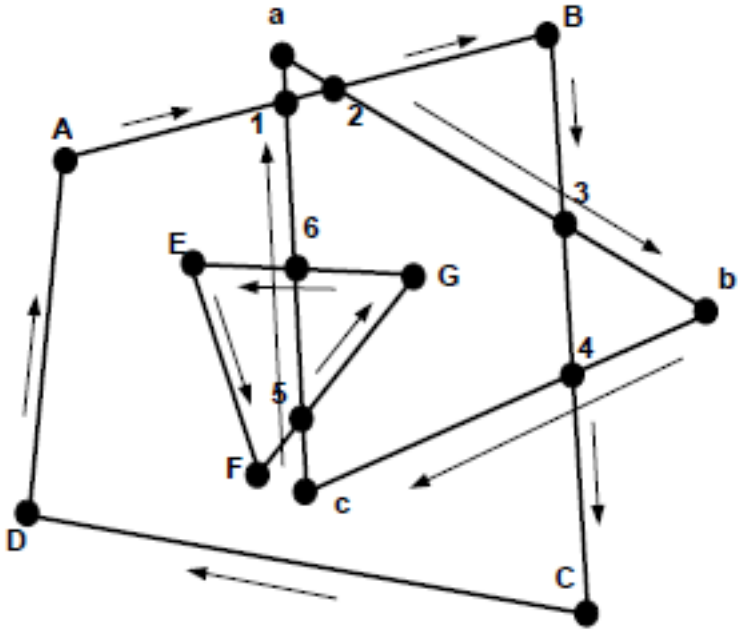
А таким образом осуществляется обход соответствующих списков.

2.7 Булевы операции с полигонами

Понятно, что результатом отсечения полигонов по алгоритму Вейлера-Азертона будет пересечением полигонов как подмножеств плоскости. Алгоритма построения объединеия двух полигонов A и B отличается от алгоритма отсечения тем, что идя по границе ппполигона A мы находим первую точку пересечения в которой происходит выход из полигона B в ее внешность.

Для получения разности $A \setminus B$ поступаем так. Идя по границе A находим входную точку пересечения. Далее переходим на полигон B и обходим его в обратном направлении. При достижении каждой точки пересечения мы

Рис. 2.9: Полигон с отверстием



перескакиваем на другой полигон и всегда полигон A обходим в прямом, а полигон B – в обратном направлении.

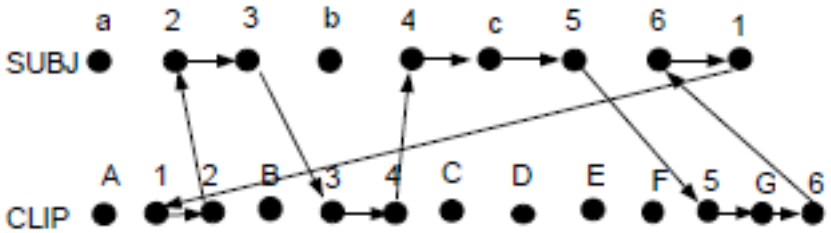
Построить соответствующие детали мы рекомендуем читателям самостоятельно.

2.8 Алгоритмы заливки областей

Фигурой мы называем плоский геометрический объект, который состоит из линий контура и точек заполнения, которые расположены внутри контура. Контуров может быть произвольное конечное количество.

Графический вывод фигуры делится на два этапа. Первый этап – это вывод контура, второй – вывод точек заполнения. Поскольку конур представляет собой объединение линий, то первая задача сводится к построению отрезков прямых или кривых из наперед заданного набора кривых, таких как дуги эллипса, кривой безье, B-сплайновой кривой.

Рис. 2.10: Обход списков



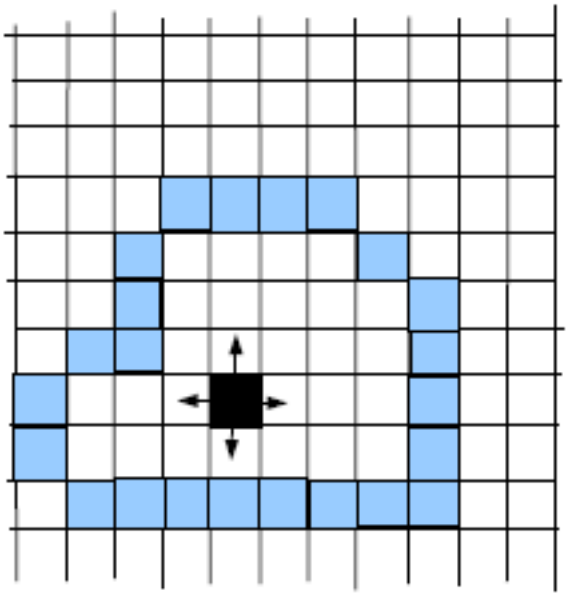
Для вывода точек заполнения известны методы, которые делятся, в основном на два типа: закрашивание от внутренних точек до границы и методы, использующие математическое описание контуров.

Рассмотрим задачу закрашивания внутренности контура, который задан последовательностью уже закрашенных пикселей. Пусть также известны координаты некоторого пикселя внутри контура. Тогда нужно закрасить его требуемым цветом и рассмотреть все соседние пиксели. Если цвет соседнего пикселя не равен цвету контура или цвету закрашивания, то он закрашивается в нужный цвет. поступаем таким образом для всех соседних пикселей. Алгоритм работает до тех пор, пока все пиксели внутри контура не закрасятся в нужный цвет. При задании пикселей контура необходимо учитывать связность его точек (4-связность или 8-связность).

Описанный выше алгоритм является рекурсивным и реализуется в следующей функции.

```
void FillRegion(int x, int y, int bordercolor, int color)
{
   int c=getpixel(x,y);
   if((c!=bordercolor)&&(c!=color))
   {
      putpixel(x,y,color);
      FillRegion(x-1,y,bordercolor,color);
      FillRegion(x+1,y,bordercolor,color);
```

Рис. 2.11: Заполнение области



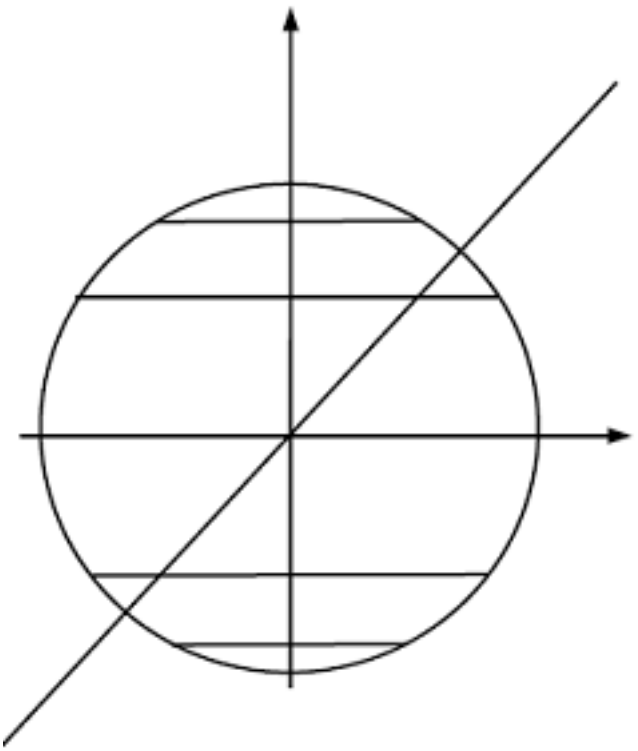
```
        FillRegion(x,y-1,bordercolor,color);
        FillRegion(x,y+1,bordercolor,color);
    }
}
```

2.8.1 Закрашивание круга

Здесь мы воспользуемся алгоритмом растровой развертки окружности. Пуст на очередной итерации циклы мы вычислсли точку (x, y) для $R > y > x > 0$ и (x_0, y_0) – центр закрашиваемого круга. Тогда перед переходом к следующей итерации цикла мы выполним следующие 4 действия

- Рисуем горизонтальную линию от $(x_0 - x, y_0 + y)$ до $(x_0 + x, y_0 + y)$,
- Рисуем горизонтальную линию от $(x_0 - y, y_0 + x)$ до $(x_0 + y, y_0 + x)$,
- Рисуем горизонтальную линию от $(x_0 - x, y_0 - y)$ до $(x_0 + x, y_0 - y)$,

Рис. 2.12: Заполнение круга

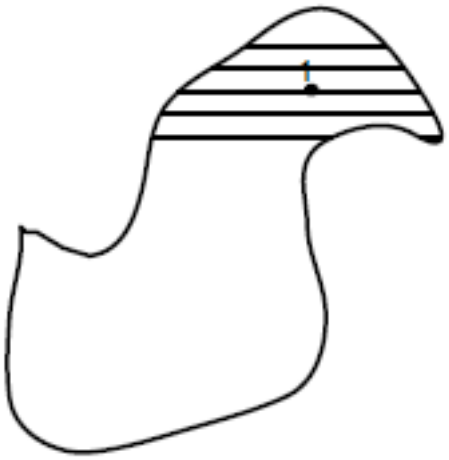


- Рисуем горизонтальную линию от $(x_0 - y, y_0 - x)$ до $(x_0 + y, y_0 - x)$.

2.9 Алгоритм закрашивания линиями

Данный алгоритм получил широкое распространение в компьютерной графике. На каждом шаге закрашивания рисуется горизонтальная линия, которая размещается между пикселами контура. Алгоритм рекурсивный,но поскольку вызов функции осуществляется не для отдельного пиксела ,а для линии, то вызов вложенных вызовов уменьшается пропорционально длине линии. Это уменьшает нагрузку на стек, и обеспечивает более высокую скорость работы.

Рис. 2.13: Заполнение области линиями



Пусть s - начальный пиксел. Первой закрашивается серия пикселов от a до b . Затем сканируется строка, лежащая выше текущей. Если такая строка найдена, то ее крайняя точка c сохраняется (ее адрес запоминается в стеке).Затем сканируется строка ниже текущей. Если она также найдена, то ее крайняя правая точка d помещается в стек. Теперь данные вытакликаваются из стека, образуется новая начальная точка d и процесс повторяется.

```
int LineFill (int x, int y,int dir, int preXL, int preXR)
{
   int xl=x,xr=x;
   COLORREF color;
     do
            color=getPixel(hdc,--xl,y);
     while(color!=Border) // цвет контура
     do
            color=getPixel(hdc,++xr,y);
     while (color!=Border);
```

```
    xl++; xr--;  //левая и правая граница текущей горизонтали

    MoveToEx(hdc,xl,y,NULL);
    LineTo(hdc,xr+1,y);//рисуем горизонтальную линию закрашивания

    for(x=xl;x<=xr;x++)
      {
        color=getPixel(hdc,x,y+dir);
          if (color!=Border)
              x=LineFill(x,y+dir,dir,xl,xr);
      }

    for(x=preXR; x<xr;x++)
      {
        color=getPixel(hdc,x,y-dir);
          if(color!=Border)
             x=LineFill(x,y-dir,-dir,xl,xr);
      }
   return xr;
  }
```

2.10 Алгоритмы заполнения многоугольников

2.10.1 Заполнение прямоугольника

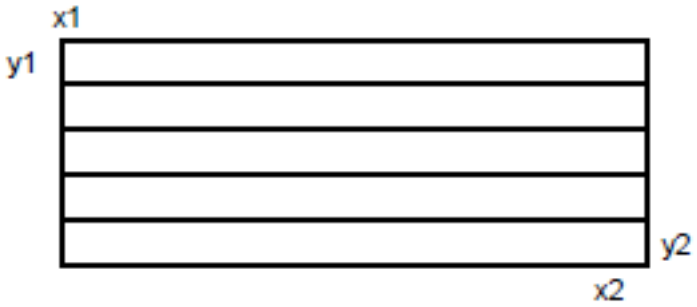
```
  for (y=y1;y<=y2;y++)
{
// рисуем горизонтальную линию с координатами  (x1,y) и (x2,y);

 }
```

2.10.2 Заполнение треугольника

Пусть треугольник задан массивом вершин $P[3]$.Определим соответствующую структуру

```
struct Point
```

Рис. 2.14: Заполнение прямоугольника

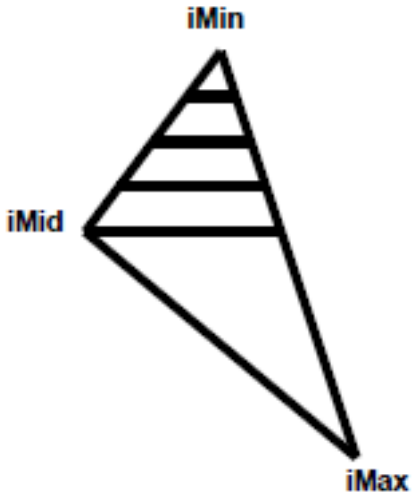



```
{
 int x;
 int y;
}
```

Для заполнения треугольника найдем самую верхнюю и самую нижнюю точки. Вычислим приращения x -координат для каждой из сторон треугольника, и будем двигаться вниз, рисуя соотвествующие горизонтальные отрезки.При этом необходимо проверять, не проходит ли следующая линия через среднюю вершину, если –да, то переходим к следующей стороне и заново вычисляем приращение для x – координаты вдоль новой стороны.

```
 void fillTriangle( Point P[])
 {
   int iMax=0;
   int iMin=0;
   int iMid=0;
 for (int i=1; i<3;  i++)
   if (P[i].y<P[iMin].y)
       iMin=i;
   else
      if (P[i].y>P[iMax].y)
        iMax=i;
   iMid=3-iMin-iMax;
```

Рис. 2.15: Заполнение треугольника



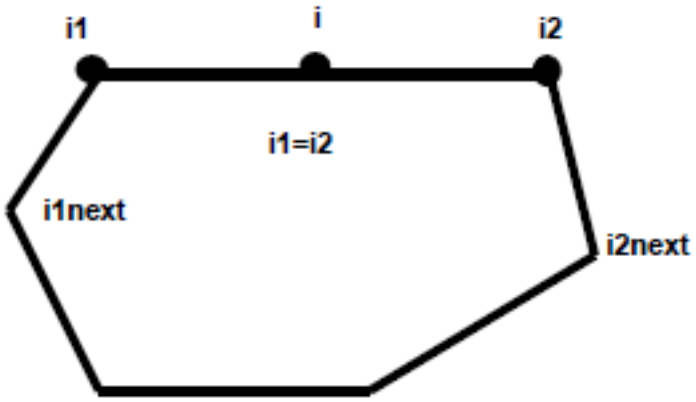

```
 long  dx01=P[iMax].y!=P[iMin].y?(P[iMax].x-P[iMin].x)/(P[iMax].y-P[iMin].y):0
// если выражения не равны между собой, то присваиваем значение приращения dx0

 long  dx02=P[iMin].y!=P[iMid].y?(P[iMin].x-P[iMid].x)/(P[iMin].y-P[iMid].y):0

 long  dx12=P[iMid].y!=P[iMax].y?(P[iMid].x-P[iMax].x)/(P[iMid].y-P[iMax].y):0

  int x1=P[iMin].x; x2=x1;

    for(i=P[iMin].y; i<=P[iMid].y;i++)
       {
               LineTo(x1,i,x2,i);
               x1+=dx01;
               x2+=dx02;
       }
    for(i=P[iMid].y+1;i<=P[iMax].y;i++)
       {
               x1+=dx01;
               x2+=dx12;
```

```
                LineTo(x1,i,x2,i);
        }
  }
```

2.10.3 Заполнение выпуклого многоугольника

Заметим,что невырожденный выпуклый многоугольник может содержать не более двух горизонтальных отрезков – вверху и внизу. Для заполнения выпуклого многоугольника найдем самую верхнюю точку и определим два ребра, выходящих 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 получится два ребра, идущие вниз. При этом они могут выходить из разных точек с одной и той же ординатой. Аналогично заполнению треугольника, будем вычислять приращения для x-координат каждого из ребер и 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 вдоль этого ребра.

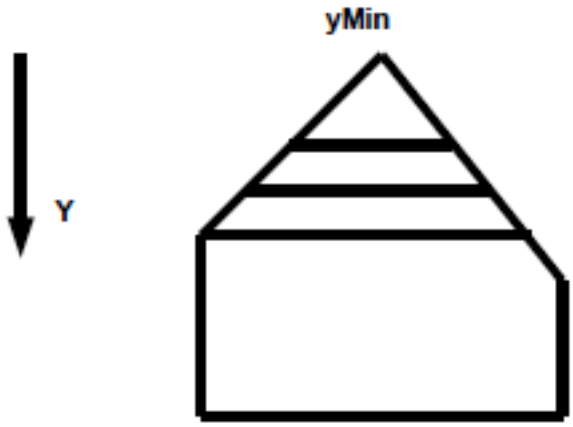
Рис. 2.16: Поиск индексов




```
int FindEdge( int &i, int dir, int n, 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) // ребро [i,i1] идет вверх
           return -1;
        else
           if (P[i1].y==P[i].y) //горизонтальное ребро
              i=i1;
           else return i1; // ребро [i,i1] идет вниз
    }
 }
```

Рис. 2.17: Заполнение многоугольника




```
void FillConvexPolygon (int n,  point P[])
{
```

```
  int yMin=P[0].y;
  int yMax=P[0].y;
  int topPointIndex=0;

  for(int i=1;i<n;i++)
     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) // вырожденный полигон
   {
    int xMin=P[0].x;
    int xMax=P[0].x;
      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;
     line(xMin,yMin,xMax,yMax);
    return;
   }
 int i1,i1next;
 int i2,i2next;

 i1=topPointIndex;
 i1next=FindEdge(i1,-1,n,P[]); //слева
 i2=topPointIndex;
 i2next=FindEdge(i2,1,n,P[]);//справа

 int x1=P[i1].x;
 int x2=P[i2].x;

 float dx1=(P[i1next].x-P[i1].x)/(P[i1next].y-P[i1].y);
 float dx2=(P[i2next].x-P[i2].x)/(P[i2next].y-p[i2].y);

  for(int y=yMin; y<=yMax; y++)
     {
       line(x1,y,x2,y);
       x1+=dx1;
       x2+=dx2;
```

```
          if(y+1==P[i1next].y) //переход к следующему ребру
            {
             i1=i1next;
                if(--i1next<0)    i1next=n-1;
                if( P[i1].y==P[i1next].y) //горизонтальная часть
                   break;
             dx1=(P[i1next].x-P[i1].x)/(P[i1next].y-P[i1].y);
             }
        if(y+1==P[i2next].y) //переход к следующему ребру
           {
             i2=i2next;
                if(++i2next>=n) i2next=0;
                if(P[i2].y==P[i2next].y)//горизонтальная часть
                  break;
            dx2=(P[i2next].x-P[i2].x)/(P[i2next].y-P[i2].y);
            }
      }

 }
```

2.10.4 XY -алгоритм заполнения полигона

Основная идея XY-алгоритма – закрашивание фигуры отрезками прямых линий. При этом удобнее использовать горизонтальные прямые. Алгоритм представляет собой цикл вдоль оси y, в ходе которого выполняется поиск точек пересечения линии контура с соответствующими горизонталями.

Общая схема алгоритма :

1)Найти значения $\min\{y_i\}$ и $\max\{y_i\}$ среди вершин P_i ;

2)Выполнить цикл по переменной y от $y = \min$ до $y = \max$

{ 3) Найти точки пересечения всех ребер контура с горизонталью y. Координаты x_i точек пересечения записать в массив.

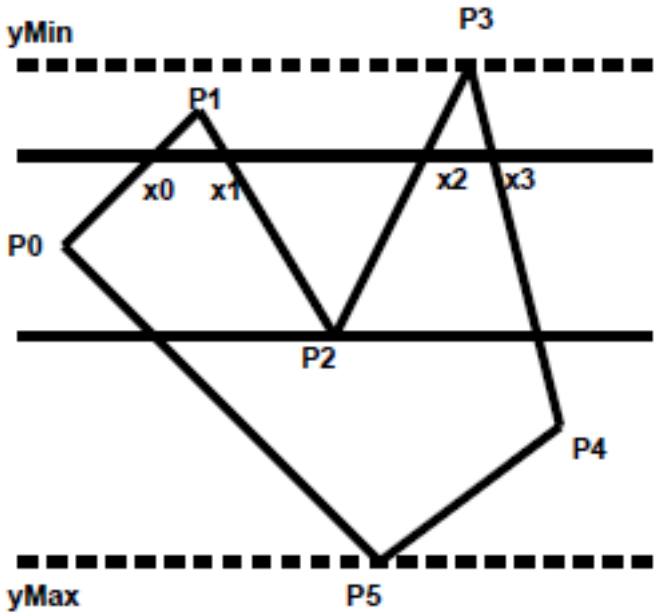
4) Отсортировать массив $\{x_i\}$ по увеличению x.

5) Выводим горизонтальные отрезки с координатами $(x_0, y)-(x_1, y)$,$(x_2, y)-(x_3, y)$,...,$x_{2k}, y) - (x_{2k+1}, y)$. Каждый отрезок рисуется цветом заполнения.

}

В этом алгоритме использовано топологическое свойство фигуры: любая прямая пересекает любой замкнутый контур четное число раз. Для выпуклых фигур число точек пересечения всегда равно двум.Таким образом, на

Рис. 2.18: Заполнение многоугольника

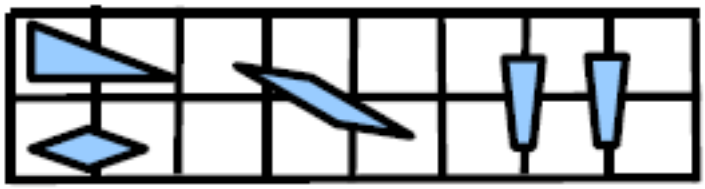


третьем шаге алгоритма в массив $\{x_i\}$ всегда должно записываться четное число точек пересечения. При нахождении точек пересечения горизонтали с контуром необходимо принимать во внимание *особые точки* . Если горизонталь имеет координату $y = y_i$ для вершины P_i, тогда следует анализировать , как горизонталь проходит через вершину. Если горизонталь пересекает контур (как на рисунке в точках P_0 или P_4), то в массив записывается одна точка пересечения. Если горизонталь касается вершины (вершины P_1, P_2, P_3, P_5 на рисунке), тогда координата точки касания или не записывается , или записывается два раза. Сложность алгоритма :$o[(y_{max} - y_{min}) \cdot n]$, где n –общее число вершин.

2.11 Технологии сглаживания ступенчатости

Ступеньки – одна из форм дефектов изображения, которая является неотъемлемым свойством растровых дисплеев. Ступечатость возникает из-за дискретной природы пикселов: пикселы появляются на дисплее в виде фиксированного прямоугольного массива.

Рис. 2.19: Проблемы мерцания

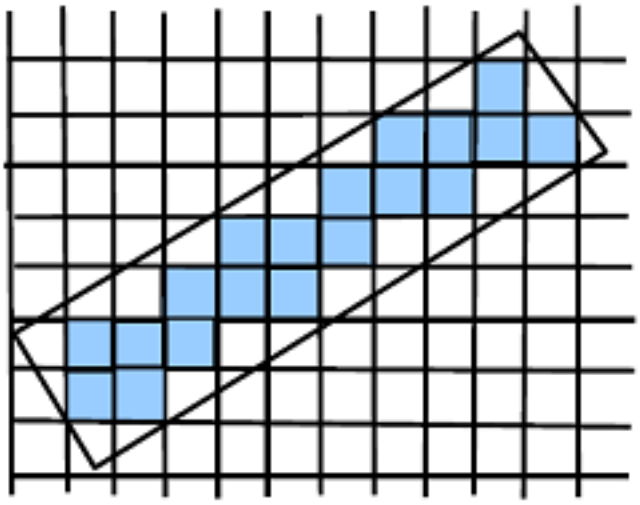


Если прямоугольник охватывает большое количество пикселов, то его граница будет выглядеть относительно гладкой. Однако, если он занимает небольшое число пикселов, то его "ступеньки "будут очень заметны. На рисунке каждый пиксел, образующий прямоугольник, окрашивается в черный цвет в том случае, если прямоугольник проходит через центр этого пиксела. Фактически, каждый пиксел "опрашивает"прямоугольник в единственной точке – центре пиксела, выясняя, имеется ли прямогугольник в этой точке, и в зависимости от ответа, вся площадь пиксела окрашивается в белый или черный цвет. Такой опрос может привести к тому, что маленькие объекты польностью исчезнут. Например, если маленький объект располагается между центрами двух или более пикселов, то он не отобразится вовсе. Объект может мерцать при анимации (см.рисунок).

Возникает вопрос: как уменьшить ступенчатость, обусловленную неэффективным опросом?

Технологии сглаживания включают в себя ту или иную форму "размытия"границ. В случае черного прямоугольника на белом фоне резкий переход от черного к белому можно смягчить путем прорисовки вблизи границы прямугольников из смеси черных и серых пикселов. В этом случае, при взгляде на изображении издали , глаз смешивает плавно изменяющиеся тени и видит

Рис. 2.20: Метод опроса



более ровное ребро.

Обычно используют 3 метода сглаживания: предфильтрацию, сверхопрос и постфильтрацию.

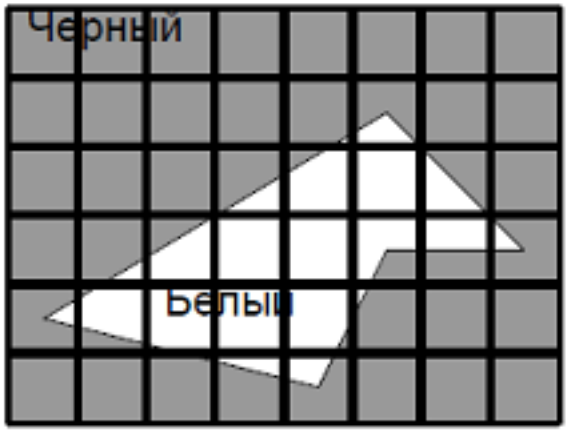
2.11.1 Предфильтрация

В технологиях предфильтрации цвета пикселов вычисляются на основе степени их покрытия объектом: вычисляется часть площади пиксела, которая покрывается объектом.

Рассмотрим сканирующее преобразование белого полигона на черном фоне. Центр каждого квадрата сетки соответствует центру пиксела на дисплее. Пикселу, половина которого покрывается полигоном, присваивается интенсивность 1/2, пикселу покрытому на треть – интенсивность 1/3 и так далее. Если буфер кадров использует 4 бита на пиксел, то в этом случае черному цвету будет соответствовать число 0, а белому -15. Пикселу, покрытому на 1/4 будет соответствовать число $1/4 \cdot 15 \approx 4$. На рисунке приведена таблица значений, полученных после вычисления степени покрытия каждого пиксела.

Геометрические вычисления, необходимые для нахождения степени покрытия каждого пиксела могут занять много времени, поэтому рассмотрим на примере соответствующую модификацию алгоритма Брезенхейма для пря-

Рис. 2.21: Белый полигон на черном фоне



мых, разработанную Питвеем и Уоткинсом (Pitteway, Watkinson []).

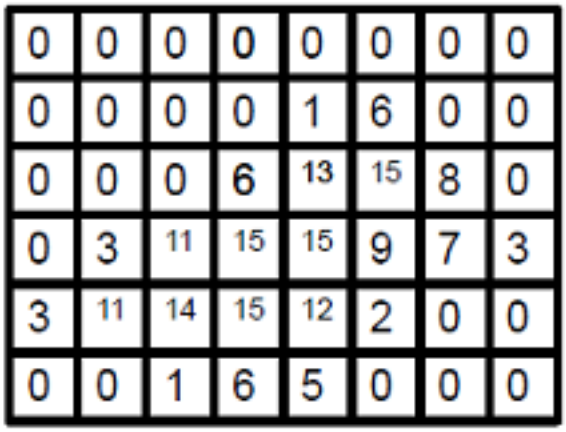
Пусть наклон ребра полигона равен 4/10. Точками на рисунке отмечены пикселы, определенные по стандартному алгоритму Брезенхейма, а затененная область – площадь внутри полигона. Из геометрических соображений легко увидеть, что по мере движения вправо пиксел за пикселом затененная область или увеличивается на $m = 0.4$, если значение y не меняется, или увеличивается на $1 - m = 0.6$, если значение y увеличивается. Следовательно, нетрудно инкрементно обновлять эту область в рамках алгоритма Брезенхейма.

Пусть $MaxLevel$ –максимальное значение интенсивности, поддерживаемое буфером кадров. Для ребра с наклоном m находим ближайшее целое $inc1 = MaxLevel * m$ и или увеличиваем интенсивность каждого пиксела на $inc1$, или уменьшаем ее на $inc2 = MaxLevel - inc1$.

Основной цикл алгоритма Брезенхейма требуется слегка изменить:

если $F(x, y) = -2W(y-a_y)+2H(x-a_x) < 0$, где рисуется отрезок от a до b, $W = b_x - a_x, H = b_y - a_y$, то кроме обновления невязки на $F+ = 2*H$, нужно обновить интенсивность $inc = +inc1$. При $F \geq 0$ $F+ = 2 * (H - W), inc = +inc2$. Для форм, отличных от полигонов, предфильтрация может оказаться дорогостоящей технологией.

Рис. 2.22: Таблица предфильтрации



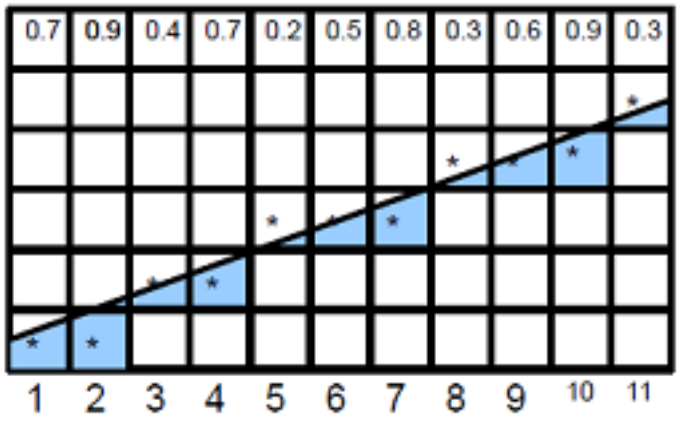
2.11.2 Сверхопрос

Сверхопрос означает проведение опросов сцены с большей частотой, чем эта сцена отображается на экране. Каждое значение пиксела формируется как усреднение нескольких опросов. Для объекта (в нашем случае это наклонная полоса) замеры объекта делаются в два раза плотнее, чем этот объект отображается на дисплее. Каждый результирующий пиксел может быть сформирован как среднее из 9 соседних замеров: центра и 8 окружающих пикселов. Пиксел в точке A формируется из 6 замеров внутри полосы и 3 замеров фона. Поэтому его цвет состоит из 2/3 цвета полосы и 1/3 цвета фона. Цвет пиксела в точке B основан на 9 замерах внутри полосы и равен цвету полосы.

2.11.3 Постфильтрация

В этом случае каждый пиксел вычисляется как средне взвешенное соответствующего набора соседних пикселов. Рассмотрим двойной опрос. Каждое числовое значение описывает интенсивность опроса сцены. На каждый квадрат накладывается квадратная маска. Вес каждой клетки умножается на соответствующий замер и складывается, получаем интенсивность пиксела.Например, для 30

Рис. 2.23: Метод Питвея-Уоткинсона



$$\frac{30}{2} + \frac{28 + 16 + 4 + 42 + 17 + 53 + 60 + 62}{16} = 32,625 \approx 33$$

Эта маска придает центральному замеру вес в 8 раз больше, чем веса остальных восьми соседей. Если w_i –веса из маски, w_0 – вес центра, то $w_0 + w_1 + ... + w_m = 1$ и

$$= \sum_{i=0}^{m} w_i \cdot c_i,$$

где c_i–цвета соседних пикселов. Приведем наиболее часто используемые маски:

Рис. 2.24: Сверхопрос

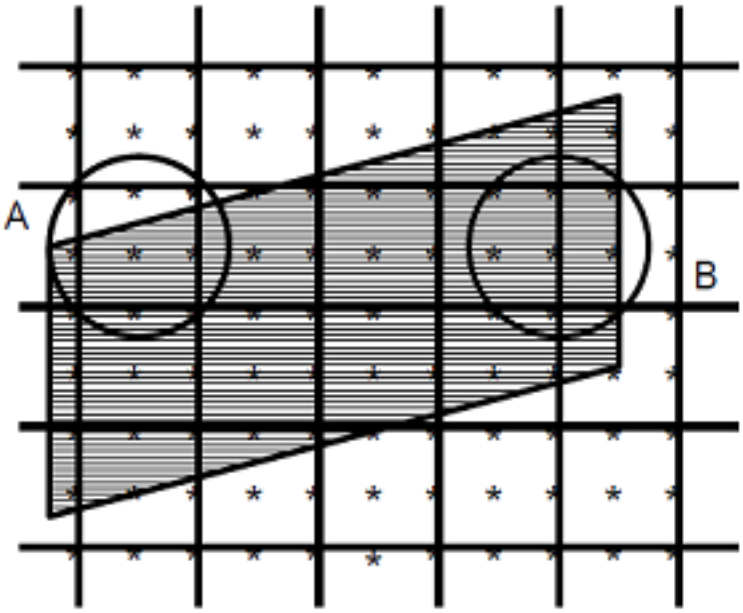


Рис. 2.25: Маска для постфильтрации

Рис. 2.26: Маски для постфильтрации

1\8

0	1	0
1	4	1
0	1	0

1\16

1	2	1
2	4	2
1	2	1

Рис. 2.27: Маски для постфильтрации

1\81

1	2	3	2	1
2	4	6	4	2
3	6	9	6	3
2	4	6	4	2
1	2	3	2	1

Глава 3

Рекурсивные алгоритмы

3.1 Черепашья графика

3.1.1 Инструменты

Часто бывает удобно начинать рисование в текущих координатах и описывать другие позиции относительно текущей точки. Поэтому было бы удобным разработать функции, параметры которых обозначают изменения координат.

```
void lineRel( float dx, float dy)
{
float x=CP.x +dx;
float y=CP.y+dy;
lineTo(x,y);
moveTo(x,y);
}
```

Черепашья графика – инструмент, который не только отслеживает с помощью текущей точки (CP), где мы находимся , но и направление, куда нас ведут.

Идея черепашьей графики в том, что черепаха подобно перу помещается на странице, оставляя за собой след в виде отрезка прямой линии. Черепаха помещается в CP , указывая текущее направление. Текущее направление отсчитывается в градусах от положительного направления оси x против часовой стрелки.

Будем обозначать текущее положение через CP, а текущее направление – CD.

Чтобы управлять черепахой добавим новые функции:

```
turnTo(float angle)
{
CD=angle;// задание текущего направления
}
turn(float angle)
{
CD+=angle;//поворот черепахи против часовой стрелки на angle градусов
}
forward(float dist, int isVisible)
{
float r=0.017453393;// для вычисления  угла в радианах
float x=CP.x+dist*cos(r*CD);
float y=CP.y=dist*sin(r*CD);
if (isVisible)
  lineTo(x,y);
else
  moveTo(x,y);
}
```

Последняя команда перемещает черепаху по прямой из текущей точки на расстояние dist в текущем направлении CD, после чего обновляет CP.

ПРИМЕРЫ

Полиспираль – ломаная линия, каждый последующий отрезок больше (или меньше) предыдущего на постоянную величину, причем последующий отрезок повернут по отношению к предыдущему на постоянный угол.

```
for (;;)
{
forward(length,1);
turn(angle);
length+=increment;
}
```

Фигура из крючка

```
{
forward(3*L,1);// L -- длина коротких участков
```

```
turn(90);
forward(L,1);
turn(90);
forward(L,1);
turn(90);
}
```

Рисование меандров. Меандр– ломаная линия, извивающаяся вдоль некоторой траектории. Его можно увидеть на греческих вазах, китайских блюдцах.

3.2 Фракталы и самоподобие

Фрактал можно определить как объект довольно сложной формы, которая получена в результате выполнения простого итерационного цикла. Итерационность процедуры создания обусловливает такие свойства фракталов как самоподобие – отдельные части похожи по форме на весь фрактал в целом. Одни кривые в точности самоподобны(какое бы сильное увеличение мы ни делали, увеличенное изображение выглядит в точности как оригинал), другие кривые – статически самоподобны.

В природе существуют примеры статического самоподобия. Классический пример – береговая линия. Другие примеры– ветви дерева, трещины на тротуаре, поверхность губки, кровеносная система животных.

В 1975г. Бенуа Мандельброт обобщил исследования природы самоподобия. Он назвал различные формы самоподобных кривых фракталами(от farctus –составленный из фрагментов, дробный). Известно, что прямая является одномерной, а плоскость – двумерной, однако были обнаружены "создания"в промежутке между ними. Например, мы можем определить кривые бесконечной длины, лежащие внутри конечного прямоугольника, их размерность располагается между 1 и 2. Работа Мандельброта и других исследователей породила невероятное количество исследований природы фракталоподобных объектов, как в области математики так и в компьютерной графике.

3.2.1 Кривая Коха

Очень сложные кривые можно получить рекурсивно, посредством многократного "усложнения "простой кривой. Простейшим примеров является кривая

Коха, открытая в 1904г. математиком Хельгом фон Кохом. Эта кривая вызвала огромный интерес, поскольку образует бесконечно длинную линию внутри области конечной площади.

Обозначим $K_0, K_1, K_2...$ – последовательные поколения кривой Коха. Форма нулевого поколения K_0 – отрезок прямой линии единичной длины. Для создания K_1 разделим прямую K_0 на 3 равные части и заменим среднюю из них треугольным "зубцом"со сторонами длиной $1/3$. Очевидно, что длина всей линии равна $4/3$.Кривая K_2 образована путем построения зубцов на каждом из четырех отрезков линии K_1.

Для создания кривой K_{n+1} из K_n необходимо разделить каждый отрезок кривой K_n на три равные части и заменить среднюю часть зубцом в форме равностороннего треугольника.

В течение этого процесса длина каждого отрезка увеличивается в $4/3$ раза, следовательно, общая длина кривой в $4/3$ больше длины кривой предыдущего поколения. Тогда общая длина кривой K_i

$$K_i = (\frac{4}{3})^i \to \infty, \quad i \to \infty.$$

Снежинкой Коха называют кривую, полученную из трех соединенных кривых Коха. Алгоритм рисования кривой Коха:

```
void DrawKoch(double dir, double len, int n)
{
double dirRad=0.0174533*dir;
if( n==0)
  lineRel(len*cos(dirRad),len*sin(dirRad));
else
  {
    n--;
    len/=3;
    DrawKoch(dir,len,n);
    dir+=60;
    DrawKoch(dir,len,n);
    dir-=120;
    DrawKoch(dir,len,n);
    dir+=60;
    DrawKoch(dir,len,n);
   }
}
```

3.2.2 Дробная размерность

В пределе кривая Коха имеет бесконечную длину, хотя занимает на плоскости ограниченную область. Чему равна ее размерность?

Математик Феликс Хаусдорф на основе анализа простых самоподобных объектов типа прямых, квадратов и кубов ввел понятие дробных размерностей. Предположим, что отношение длины каждого из отрезков к длине исходной прямой равно $1/N = r($ при этом прямая единичной длины разбита на N равных отрезков). Проделаем тоже самое с квадратом: разделим его на N равных квадратов. Тогда отношение стороны маленького квадрата к исходному равно $r = 1/\sqrt{N}$. Аналогичная процедура для куба приведет к тому, что отношение стороны каждого подкуба к стороне исходного будет равно $r = 1/N^{1/3}$. Таким образом, можно обнаружить следующую закономерность: размерность объекта фигурирует в показателе степени при основании N. Будем говорить, что *объект обладает размерностью* D , если при делении его на N равных частей, каждая часть будет иметь сторону меньшую, чем сторона исходного объекта в $r = 1/N^{1/D}$ раз. После логарифмирования обеих частей получим

$$D = fraclogNlog\frac{1}{r}.$$

Чему же равна размерность кривой Коха? При переходе от одного поколения к следующему из каждого отрезка создается $N = 4$ новых отрезка, а их длины получаются из длин отрезков предыдущего поколения умножением на $r = 1/3$. Тогда

$$D = \frac{log4}{log3} = 1.26...$$

Если кривая A обладает большей размерностью, чем кривая B, то кривая A обязательно будет более извилистой. Это означает, что она меньше похожа на линию, ей присуща большая "заполняемость"плоскости. Так, фрактальная кривая Пеано полностью заполняет плокость и приобретает размерность 2.

Усложнение каждого очередного поколения кривой можно осуществить многими способами:

Оглавление