Lecture 8

Scan Conversion: Lines and Curves

1 Introduction

Scan conversion is the process of converting geometric primitives to sets of pixels. This chapter discusses the scan conversion of lines and curves; the next chapter discusses the scan conversion of polygons.

This chapter begins by establishing some definitions and conventions about raster coordinate systems and sampling geometry. The various mathematical approaches towards curve definition are discussed. Scan conversion algorithms for lines and circles are discussed in detail, followed by a more general discussion of scan conversion algorithms for parametric and implicit curves. Finally, techniques for antialiasing lines are covered.

2 Conventions

Before discussing specific algorithms it is important to define what pixels should be drawn to approximate the curve. The best definition is based on overlapping domains and is motivated by the earlier discussion of signal processing, in particular, prefiltering. As mentioned, one method to prevent aliasing is to prefilter the geometry before sampling. This corresponds to convolving the curve with a shape equal to the domain of the filter and then sampling the resulting shape. The convolution is equivalent to sweeping a brush whose shape is equal to the filter's domain along the curve's trajectory. In the case of a thin curve, the result will be a thick curve with a non-zero width. An alternative definition is to consider each pixel to have a finite area and output each pixel if the thin curve overlaps the pixel. Either of these definitions will yield the same set of pixels.

Before proceeding we must agree on the raster coordinate system. There are two common raster coordinate systems in use which differ in where they consider pixel centers:

1. The most common system considers the raster coordinate system to be like graph paper. The lines of the graph paper are at integer values and the pixel are unit squares that tile the plane. In this system, the edges of pixels are at the integers and the centers are at half-integers. Filling a rectangle is very natural in this coordinate system, because we fill all the squares inside the half-open interval between the minimum and maximum coordinates.

2. The other common system is to center pixels at integer locations. This is the most natural coordinate system if we think of the pixels as a set of samples because it is most natural to think of the samples as falling on integer locations. Also, when reconstructing a function we convolve the reconstruction function with the lattice of samples. Since most reconstruction functions are symmetric and centered at the origin, this creates a pixel whose shape is equal to the domain of the reconstruction function centered at the pixel center. The graph paper metaphor works if we think of the reconstruction function as being an off-centered unit square. But reconstruction functions can have arbitrary shape

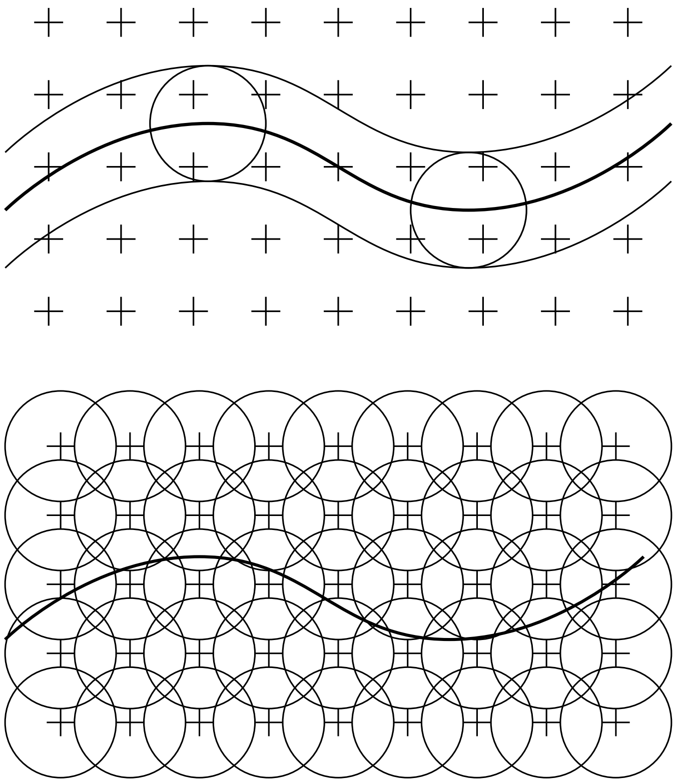


Figure 1: Sweeping vs. Intersecting a Filter with a Curve

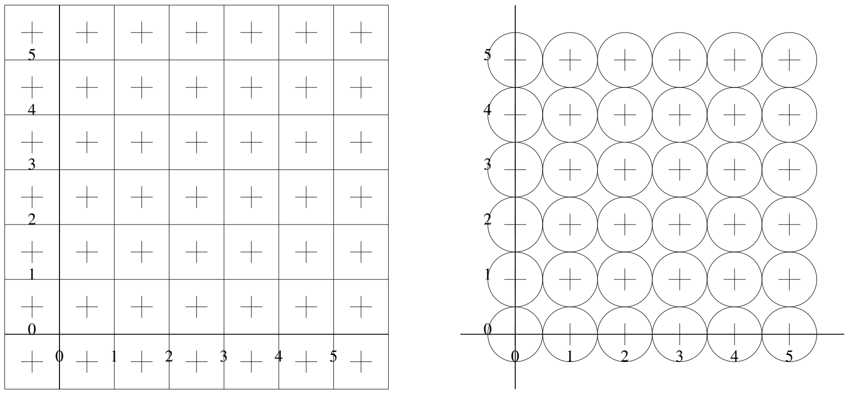


Figure 2: Half-integer vs. Integer Pixel Centers

(although the circle and square are the most common) and width (a width greater than 1 implies that the reconstruction functions overlap), so the simple graph paper metaphor breaks down.

The closest pixel center to a given (x, y) location is at $(floor(x), floor(y))$ in the first model, and $(round(x), round(y))$ in the second model. Thus, it is equally easy to use either model. We, however, advocate the use of the second raster coordinate system and that is the convention that will be followed in these notes. It is very important to agree beforehand on what coordinate system is being used, and to be consistent when implementing scan conversion algorithms. Otherwise nasty off by one artifacts are likely to occur.

The other aspect of the raster coordinate system is whether it is left or right handed. Traditionally raster graphics coordinate systems are left handed with x increasing to the right and y increasing downward. The origin is in the upper left corner. This coordinate system arises naturally when considering the raster scan process needed to refresh the display because it begins in the upper left and moves right and then down. Unfortunately this coordinate system defies common mathematical conventions where y increases upward and the origin is in the lower left. In these notes we will adopt this second convention.

3 Mathematical Curves

There are two ways to define a curve—the *parametric* and *implicit* forms.

In the parametric form, a curve is parameterized by a single number which is roughly the arc length along the curve. Points on the curve are given as functions of this parameter value.

$$(x(t), y(t))$$

In the implicit form we classify all points in the plane.

$$f(x, y) \begin{cases} < 0 & \text{inside} \\ = 0 & \text{on} \\ > 0 & \text{outside} \end{cases}$$

In this chapter we will discuss lines and circles. The parametric form for a line passing through two points is

$$x(t) \quad = x_1 + t(x_2 - x_1)$$

$$y(t) \quad = y_1 + t(y_2 - y_1).$$

The implicit form for a line is

$$f(x, y) = ax + by + c.$$

If the line passes through two points (x_1, y_1) and (x_2, y_2), the line coefficients are

$$\begin{aligned} a &= y_2 - y_1 \\ b &= x_1 - x_2 \\ c &= x_2 y_1 - x_1 y_2 \end{aligned}$$

There are many parametric forms for a circle; the most common is to treat the parameter as the angle. The equation for a circle of radius r with center at (x_c, y_c) is

$$\begin{aligned} x(t) &= x_c + r\cos 2\pi t \\ y(t) &= y_c + r\sin 2\pi t. \end{aligned}$$

The implicit form for the same circle is

$$f(x, y) = (x - x_c)^2 + (y - y_c)^2 - r^2.$$

A parametric curve is quite easy to draw as a sequence of lines. Points on the curve are evaluated at evenly spaced parameter values and successive points are connected by straight lines. Of course, we still need a method for drawing lines with pixels. And this method doesn't necessarily give the best approximation to the curve.

Drawing implicit curves is more difficult because it is harder to find points on the curve.

4 Line Drawing

Here is perhaps the simplest possible line drawing algorithm:

```
line( int x1, int y1, int x2, int y2 )
{
    int n;
    int dx, dy;
    float t, dt;
    float x, dxdt;
    float y, dydt;

    dx = x2 - x1;
    dy = y2 - y1;

    dt = n = = max(abs(dx),abs(dy));

    dxdt = dx/dt;
    dydt = dy/dt;

    x = x1;
    y = y1;
    while( n-- )
        fragment( round(x+dxdt*t), round(y+dydt*t) );
}
```

The only tricky part is deciding what parameter step size to use. If we make the parameter step too large, then the spacing between successive pixels is greater than 1 in either x or y and we end up leaving gaps. On the other hand, if we make the parameter step too small, then the spacing between pixels is less than 1 and we may output a pixel multiple times and thus perform more work than necessary.

In the case of lines this problem is easy to solve. We set the parameter value so that either dx/dt or dy/dt equals 1 and the other derivative is less than 1. The axis where the derivative is equal to 1 is often referred to as the *major axis*, and the other axis as the *minor axis*.

This algorithm can be sped up in several ways. Since we are outputting a sequence of connected points, it is easy to take advantage of this by performing the computation incrementally.

```
line( int x1, int y1, int x2, int y2 )
{
    int n;
    int dx, dy;
    float x, dxdt;
    float y, dydt;
    float dt;

    dx = x2 - x1;
    dy = y2 - y1;

    dt = n = max(abs(dx),abs(dy));

    dxdt = dx / dt;
    dydt = dy / dt;

    x = x1;
    y = y1;
    while( n-- ) {
        fragment( round(x), round(y) );
        x += dxdt;
        y += dydt;
    }
}
```

This type of algorithm is sometimes referred to as a *digital differential analyzer* or *DDA*. What a DDA does is solve a parametric equation by solving a differential equation. Any parametric equation can be converted to a differential equation using its Taylor expansion.

$$df = f(t + dt) - f(t) = f'(t)dt + \frac{f''(t)}{2!}dt^2 + ...$$

For polynomials of degree n there are at most only n non-zero derivatives. In particular for a line, there is only a single non-zero derivative so this works out very simply. This is also similar to the technique of using *finite differences* in numerical analysis.

Another way of drawing a line is to think of the line in implicit form. The idea of this approach is to follow along the curve as closely as possible on one side. If we assume that the slope of the line is between 0 and 1, and that the linear function is positive in the upper half space, then the major direction is positive x and the minor direction is positive y. Tracking the line is illustrated in Figure 3 and accomplished by the following procedure:

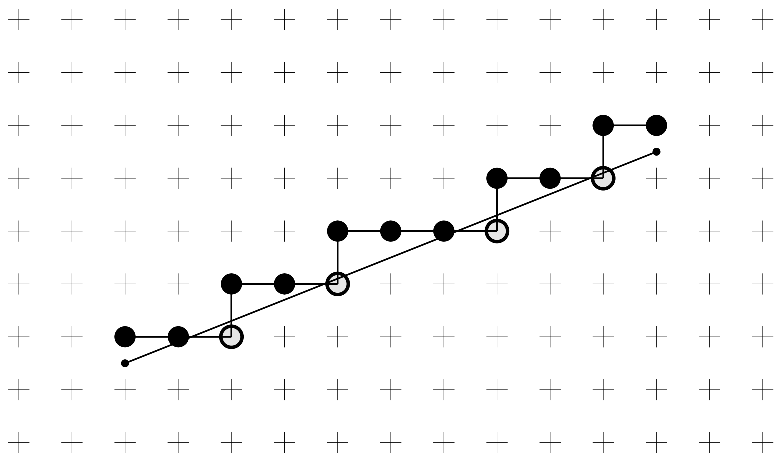


Figure 3: Tracing a Line in the 1st Octant

```
while( n--  ) {
    fragment( x, y );
    move right;
    if( below line )
        move up;
```

Note that each time through the loop we either move horizontally or diagonally.

The procedure `vect` assumes we are given a point (x, y) near the line. This procedure then traces out n points along the line given by the line equation $ax + by + c = 0$. This is basically Bresenham's algorithm.

```
void
vect( int x, int y, int n, value a, value b, value c )
{
    value e;
    value ap, bp, cp;

    ap = 2*a;
    bp = 2*b;
    cp = 2*c+b;

    e = ap*x+bp*y+cp;
    while( n--  ) {
        assert( e == ap*x + bp*y + cp );
        assert( 0 <= e < bp );
        fragment( x, y );
        x += 1;
        e += ap;
        if( e < 0 ) {
            y += 1;
            e += bp;
        }
    }
}
```

The variable `e` equals $ax + by + c$ at all times. Note how `e` is updated incrementally by computing the forward difference of the line equation.

$$\begin{aligned} e(x+1,y) - e(x,y) &= a(x+1) + by + c - (ax + by + c) = a \\ e(x,y+1) - e(x,y) &= ax + b(y+1) + c - (ax + by + c) = b. \end{aligned}$$

The variable e in Bresenham's algorithm is often called an *error term* because it is proportional to the distance from the pixel center to the line. In fact, the vertical distance to the line is

$$d_{\uparrow} = \frac{e}{2b},$$

the perpendicular distance to the line is

$$d_{\perp} = \frac{e}{2b}\frac{b}{\sqrt{a^2+b^2}}$$

This method of tracing the line only generates pixels outside, i.e. above, the line. However, ideally we would like half of the pixels to be inside and half outside the line. This can be remedied by displacing the line downward by half a pixel. To understand how this is done, recall that the family of lines with the same a and b values, but different c values are all parallel. Thus, for any given y the line $1/2$ unit below the line must obey the following equation.

$$ax + b(y + 1/2) + c = ax + by + (c + b/2) = 0$$

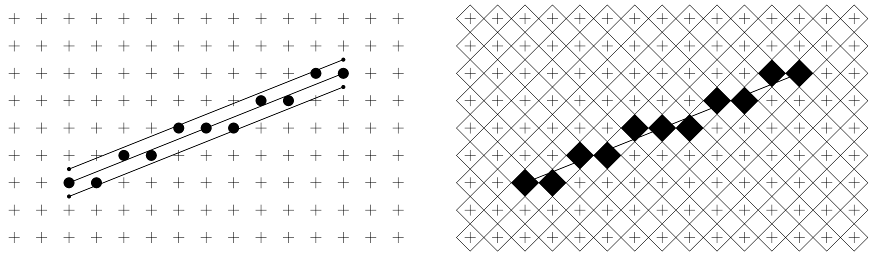


Figure 4: Inside vs. Overlapping Pixels

Thus, setting c to $c + b/2$, creates a line parallel to and half a pixel below the original line. Dividing b by 2 throws away the low order bit of b. This can be remedied by multiplying all the coefficients by 2, since this has no effect on what points satisfy the line equation.

Note that coefficients of the line equation passed to the procedure `vect` have the abstract type `value`. Possible types are integers, floats, or fixed point numbers. If the endpoints of the line are integers, then the coefficents of the line will also be integers. However, often the line's endpoints are given by floating point coordinates. In this case it is better to rasterize a high precision line that passes through the high precision vertices, than to rasterize a low precision line passing through the vertices rounded to the nearest integer location. Rounding the vertices to pixel centers will displace the line from its desired location.

Actually the above algorithm generates all pixels inside a wide line that has a vertical thickness equal to one pixel (see Figure 4). This means that the perpendicular thickness (which is what we would normally mean by thickness) is equal to $\cos\theta$, where θ is the angle the line makes with respect to the x axis. A horizontal line will have thickness equal to 1, whereas a $45°$ line will have thickness equal to $\frac{\sqrt{2}}{2}$. The total area of the thick line is thus $l\cos\theta$, where l is the length of the line, whereas ideally the line should have area l. Thus, the thickness of the line decreases with angle, or, said another way, lines at non-zero angles appear thinner than horizontal lines. Another way of seeing this is that the number of pixels generated is only a function of the x length of the line and does not depend on the y. As we increase y, the length increases but the number of pixels does not. In general, this variation of effective thickness with angle is undesirable.

As mentioned earlier, we can also think of scan conversion algorithms in terms of the shapes of the pixels. Forming a line with vertical distance equal to 1 can be done by convolving it with a diamond. This implies that the above algorithm is equivalent to thinking of pixels as diamonds and outputting a pixel if its diamond intersects the line (see Figure 4).

The procedure `vect` only generates lines in the 1st octant. Lines in the other eight octants can be handled by symmetry.

```
line( int x1, int y1, int x2, int y2 )
{
    value a, b, c;
    int n;
    int dx, dy;

    dx = x2 - x1;
    dy = y2 - y1;

    identity();
    translate( x1, y1 );

    if( dx < 0 ) {
        dx = -dx;
        reflectx();
    }
    if( dy < 0 ) {
        dy = -dy;
        reflecty();
    }
    if( dx < dy ) {
        dx ^= dy; dy ^= dx; dx ^= dy;
        reflectxy();
    }
    a = -dy; b =  dx; c =   0;
    n =  dx;
    vect( 0, 0, n, a, b, c );
}
```

Note that this procedure assumes that the endpoints are integers. This means that the line passes through the origin, and so c is 0.

The fragment function applies the appropriate symmetry transformations to the coordinates before outputting a pixel.

```
static int tranx, tranx
static int reflx, reflx, reflxy;

void identity( void )
{
    tranx = trany = 0;
    reflx = reflx = reflxy = 0;
}
void reflectxy( void )
{
    reflxy = !reflxy;
}
void reflectx( void )
{
    reflx = !reflx;
}
void reflecty( void )
{
    refly = !refly;
}
void translate( int dx, int dy )
{
    tranx = dx; trany = dy;
}
void fragment( int x, int y )
{
    if( reflxy ) {
        int t; t = x; x = y; y = t;
    }
    if( refly ) y = -y;
    if( reflx ) x = -x;

    point(x+tranx,y+trany);
}
```

4.1 Antialiasing Lines

The lines drawn by the above algorithms have a displeasing jagged or staircased appearance (see Figure 5). These jaggies are due to bad sampling and reconstruction. Fortunately, these artifacts can be minimized by prefiltering the lines before sampling. This causes the intensity of the pixels to change depending on how close they are or how much they overlap the line. Besides making more pleasing drawings, antialiasing also increases the effective resolution of the line and its endpoints. This trade off between intensity resolution and spatial resolution is the opposite of that occurring when dithering. In that case, we sacrifice spatial

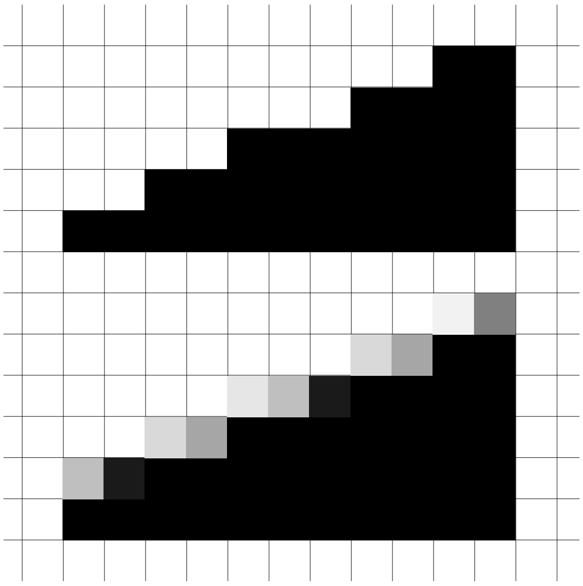


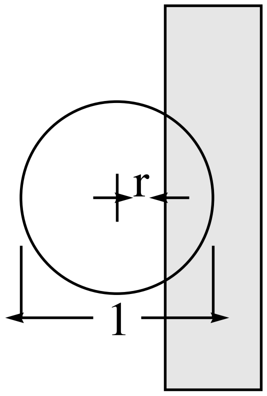
Figure 5: Jagged vs. Smooth Lines

resolution for greater intensity resolution. In this case, we increase the spatial resolution by decreasing the intensity resolution. Fortunately, lines are often drawn with a fixed constant color. Leler has estimated that displaying 1-bit lines on a 512x512 8-bit display with antialiasing is equivalent to displaying the same lines at a resolution of 4Kx4K on a 1-bit display. One way of thinking of this is to consider drawing an antialiased line between two adjacent horizontal pixels. If the line passes exactly through the left pixel, the left pixel is set to white and the right pixel is set to black. In contrast, if the line passes exactly through the right pixel, the right pixel is white and the left pixel is black. As the line moves from left to right the left pixel goes from white to black and the right pixel goes from black to white. If we are given a certain ratio of white to black, our eye, which is designed to detect edges, will estimate the subpixel position of the line corresponding to that ratio.

There are two common ways to efficiently antialias lines. In both cases we first compute the fraction, or coverage, of a thick line with the pixel, and then blend the new value in with the pixel's old value.

The first method convolves the line with a circularly symmetric filter and then samples the result. The simplest filter has a constant value. Thus, because it is radially symmetric this calculation involves computing the area of the circle covered by the line.

The fractional area of overlap between a unit circle and a line at a distance r from its center.

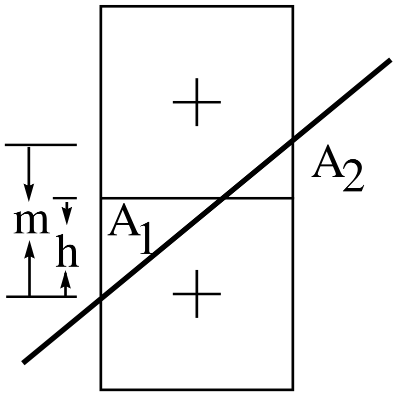


$$
\begin{aligned}
A(r) &= \frac{4}{\pi}\int_{-\frac{1}{2}}^{r}\sqrt{1-4x^2}dx \\
&= \frac{1}{2}+\frac{1}{\pi}(2r\sqrt{1-4r^2}+\sin^{-1}2r)
\end{aligned}
$$

In this equation r must lie between $-1/2$ and $1/2$.

The second method used to antialias lines is to assume the filters are rectangular. If the filters have 8-fold symmetry (so that results don't depend on which octant Bresenham's algorithm is operating within), then the filters can be tabulated and indexed by the vertical distance instead of the perpendicular distance.

The simplest filter in this case is a unit square with a constant value. This corresponds to calculating the area of a thick line within the square. Assuming the line is thick enough to cover the lower half-space, then it will cover A_1 of the upper pixel and $(1-A_2)$ of the lower pixel where A_1 and A_2 are given by the following formulas:



$$A_1 = \frac{1}{2}\frac{(m-h)^2}{m}$$

$$A_2 = \frac{1}{2}\frac{h^2}{m}$$

Where $m = a/b$ and h is the distance shown in the above figure.

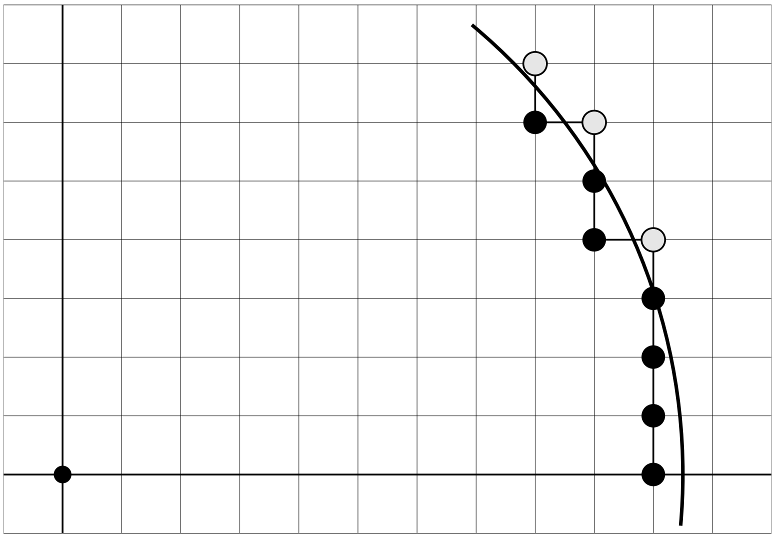


Figure 6: Circle Drawing Algorithm

5 Circle Drawing

The easiest way to scan convert a circle is to use a contour following approach (see Figure 6).

Suppose we are given the circle equation $x^2 + y^2 - r^2 = 0$. Because the circle has eight-fold symmetry, if we draw that portion of the circle within the first octant we can draw the remaining octants by symmetry. The algorithm will draw the circle by tracing along its inner edge in the first octant starting from the positive x-axis. In this octant, we always will move up and will sometimes move left, if we go outside the circle. To center the pixels we displace the circle $1/2$ unit to the right. These leads to a circle centered at the point $(1/2, 0)$ obeying the following equation:

$$f(x, y) = (x - 1/2)^2 + y^2 - r^2 = 0.$$

This approach leads to the following algorithm:

```
sector( int r )
{
    int x, y, e;

    x = r;
    y = 0;
    e = -r;
    while( x > y ) {
        assert( e == $(x-1/2)^2+y^2-r^2$ );
        fragment( x, y );
        e += 2*y+1;
        y += 1;
        if( e >= 0 ) {
            e += -2*x + 2;
            x -= 1;
        }
    }
}
```

Note that the error term is initialized to

$$e = f(r, 0) = (r - 1/2)^2 - r^2 = -r + 1/4.$$

However, since we only compare e to 0, and since e can only change by integral amounts, we can drop the $1/4$.

To evaluate the error term incrementally we write out the circle equation in terms of finite differences.

$$f(x, y + 1) - f(x, y) = (y + 1)^2 - y^2 = 2y + 1$$

$$f(x - 1, y) - f(x, y) = (x - 3/2)^2 - (x - 1/2)^2 = -2x + 2$$

Actually, note that the change in the error term can is also a function of x and y so we can write these in terms of finite differences as well. These leads to the following more efficient algorithm:

```
sector( int r )
{
    int x, y;
    int e, dedx, dedy;

    x = r;
    y = 0;
    e = -r;
    dedx = -2*r+2;
    dedy = 1;
    while( x > y ) {
        fragment( x, y );
        assert( e == $(x-1/2)^2+y^2-r^2$ );
        e += dedy; dedy += 2;
        y += 1;
        if( e > 0 ) {
            e += dedx; dedx += -2;
            x -= 1;
        }
    }
}
```

6 References

Line Drawing

Bresenham, J.E., Algorithm for computer control of a digital plotter. IBM Systems Journal 4(1) pp 35-30, 1965.

Gupta, S. and Sproull, R.F., Filtering edges for gray-scale displays. Computer Graphics 15(3) pp. 1-5, 1981.

Leler, W.J., Human vision, anti-aliasing, and the cheap 4000 line display. Computer Graphics 14(3) pp. 308-313, 1980.

Pitteway, M.L.V. and Watkinson, D.J., Bresenham's algorithm with grey scale, CACM 23(11) pp. 625-626, 1980.

Circle Drawing

Bresenham, J.E., A linear algorithm for the incremental digital display of circular arcs. CACM 20 pp. 100-106, 1977.

Horn, B.K.P., Circle generators for display devices. Computer Graphics and Image Processing 5 pp. 280-288, 1976.