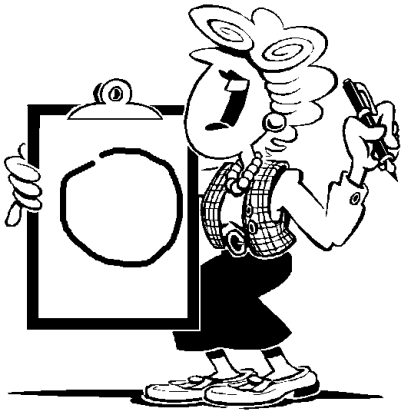
Circles and Conics



- Drawing circles
- Drawing conics
- Project #2

Hearn & Baker, pp 97-112 (Study)

Circle Drawing Algorithms

- Special case conic curves
  - frequency of use
  - symmetry
- Early algorithms also developed by Bresenham

Starting with the equation of a circle:
Solve for y in terms of x:

Simple Implementation

```
public void circleSimple(int xCenter, int yCenter, int radius, Color c)
{
    int pix = c.getRGB();
    int x, y, r2;

    r2 = radius * radius;
    for (x = -radius; x <= radius; x++) {
        y = (int) (Math.sqrt(r2 - x*x) + 0.5);
        raster.setPixel(pix, xCenter + x, yCenter + y);
        raster.setPixel(pix, xCenter + x, yCenter - y);
    }
}
```

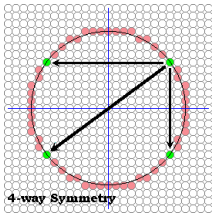
And the result would look like:

As you can see the circles look fine in areas where only one pixel is required for each column, but in areas of the circle where the local slope is greater the one the circle appears discontinuous (where have we seen this before?).

The above applet demonstrates the *circleSimple()* algorithm. Click and drag the left button to specify the circle's center and a point on its radius. Selecting the right button will clear the drawing. The circle approximation generated by the algorithm is overlaid with an ideal circle for comparison.

Exploiting Symmetry

We could compute the derivative (i.e. slope) at each point and then decide to step in the x or y. But, let's explore a different approach here.



A circle exhibits a great deal of symmetry.

- Our simple circle-drawing algorithm already exploits *2-way symmetry* about the x-axis.
- A circle has even more symmetry.
- For example about the y axis.

A circle exhibits a great deal of symmetry. We've already exploited this somewhat by plotting two pixels for each function evaluation; one for each possible sign of the square-root function. This symmetry was about the x-axis. The reason that a square-root function brings out this symmetry results from our predilection that the x-axis should be used as an independent variable in function evaluations while the y-axis is dependent. Thus, since a function can yield only one value for member of the domain, we are forced to make a choice between positive and negative square-roots. The net result is that our simple circle-drawing algorithm exploits *2-way symmetry* about the x-axis.

Obviously, a circle has a great deal more symmetry. Just as every point above an x-axis drawn through a circle's center has a symmetric point an equal distance from, but on the other side of the x-axis, each point also has a symmetric point on the opposite side of a y-axis drawn through the circle's center.

Circle Drawing with 4-way Symmetry

The following code fragment takes advantage of the 4-way symmetry of circles.

```
public void circleSym4(int xCenter, int yCenter, int radius, Color c)
{
    int pix = c.getRGB();
    int x, y, r2;

    r2 = radius * radius;
    raster.setPixel(pix, xCenter, yCenter + radius);
    raster.setPixel(pix, xCenter, yCenter - radius);
    for (x = 1; x <= radius; x++) {
        y = (int) (Math.sqrt(r2 - x*x) + 0.5);
        raster.setPixel(pix, xCenter + x, yCenter + y);
        raster.setPixel(pix, xCenter + x, yCenter - y);
        raster.setPixel(pix, xCenter - x, yCenter + y);
        raster.setPixel(pix, xCenter - x, yCenter - y);
    }
}
```

On average how much faster would you expect this algorithm to be?

We can quickly modify our previous algorithm to take advantage of this fact as shown below.

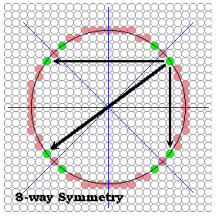
```
public void circleSym4(int xCenter, int yCenter, int radius, Color c)
{
    int pix = c.getRGB();
    int x, y, r2;

    r2 = radius * radius;
    raster.setPixel(pix, xCenter, yCenter + radius);
    raster.setPixel(pix, xCenter, yCenter - radius);
```


Same Old Problem

This improved algorithm is roughly twice as fast as our original, but it still has the same problems.

This algorithm **has all the problems** of our previous algorithm, but, it gives the same result with **half as many function evaluations**. So much for "making it work first" before optimizing. But, we're on a roll so let's push this symmetry thing as far as it will take us.

More Symmetry



We can actually fix this problem with slopes greater than one by taking advantage symmetry.

In addition to the symmetries about the x and y axes, a circle is also symmetric about the lines $x = y$ and $x = -y$.

Thus, for every point (x,y) we can plot 7 other points:

(-x,y)
(-x,-y)
(x,-y)
(y,x)
(-y,x)
(-y,-x)
(y,-x)

Why does this eliminate our problem?

Circle Drawing with 8-way Symmetry

```
public void circleSym8(int xCenter, int yCenter, int radius, Color c)
{
    int pix = c.getRGB();
    int x, y, r2;

    r2 = radius * radius;
    raster.setPixel(pix, xCenter, yCenter + radius);
    raster.setPixel(pix, xCenter, yCenter - radius);
    raster.setPixel(pix, xCenter + radius, yCenter);
    raster.setPixel(pix, xCenter - radius, yCenter);
    for (x = 1; ; x++) {
        y = (int) (Math.sqrt(r2 - x*x) + 0.5);
        if (x > y) break;
        raster.setPixel(pix, xCenter + x, yCenter + y);
        raster.setPixel(pix, xCenter + x, yCenter - y);
        raster.setPixel(pix, xCenter - x, yCenter + y);
        raster.setPixel(pix, xCenter - x, yCenter - y);
        if (x == y) break;
        raster.setPixel(pix, xCenter + y, yCenter + x);
        raster.setPixel(pix, xCenter + y, yCenter - x);
        raster.setPixel(pix, xCenter - y, yCenter + x);
        raster.setPixel(pix, xCenter - y, yCenter - x);
    }
}
```

We get 8 points for every function evaluation (approximately 4-times faster than our initial circle-drawing algorithm).

So now we get 8 points for every function evaluation, and this routine should be approximately 4-times faster than our initial circle-drawing algorithm. What's going on with the four pixels that are set outside the loop (both at the top and bottom)? Didn't I say that every point determines 7 others?

Wait a Minute!

What has happened here?

One Last Trick

Can we eliminate that sqrt( ) call from the inner loop?

- Notice we only computes values of y where the local slope is less than 1.
- At every step in x we either leave y at its current value or change it by 1.

First, lets revisit the equation of a circle centered at the origin:

This expression will take on the following values:

We call this function, *f(x, y)*, a *discriminator*.

Evaluating the Discriminator

Assuming that we have just drawn pixel *(x, y)*, the discriminator tells us if we should advance y on the next step. Resulting in the new coordinate *(x+1, y-1)*.

When y is expressed in terms of x we make this determination by rounding each function evaluation (i.e. by adding 1/2 to y and truncating). We can do the same thing with this implicit function.
If the result >= 0 then we advance y, otherwise we stay at the old y value.

We can also express the new value of f(x,y) incrementally:
Initiallly:

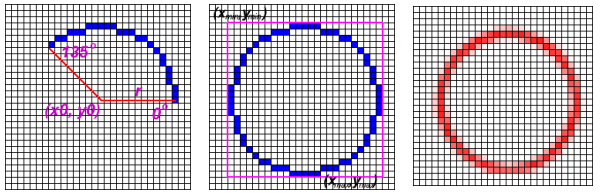
Bresenham's Midpoint Algorithm

```
public void circleMidpoint(
    int x0, int y0, int radius, Color c) {

    int pix = c.getRGB();
    int x = 0;
    int y = radius;
    int d = (5 - 4*radius)/4;

    circlePoints(x0, y0, x, y, pix);
    while (x < y) {
        if (d < 0) {
            d += 2*x+3;
            x += 1;
        } else {
            d += 2*(x-y)+5;
            x += 1;
            y -= 1;
        }
        circlePoints(x0, y0, x, y, pix);
    }
}
```

```
private void circlePoints(
    int cx, int cy,
    int x, int y, int pix) {
    if (x == 0) {
        raster.setPixel(act, cx, cy + y);
        raster.setPixel(pix, cx, cy - y);
        raster.setPixel(pix, cx + y, cy);
        raster.setPixel(pix, cx - y, cy);
    } else
    if (x == y) {
        raster.setPixel(act, cx + x, cy + y);
        raster.setPixel(pix, cx - x, cy + y);
        raster.setPixel(pix, cx + x, cy - y);
        raster.setPixel(pix, cx - x, cy - y);
        if (x < y) {
           raster.setPixel(pix, cx + y, cy + x);
           raster.setPixel(pix, cx - y, cy + x);
           raster.setPixel(pix, cx + y, cy - x);
           raster.setPixel(pix, cx - y, cy - x);
        }
    }
}
```


Circular Discussions

- Our version of the Midpoint algorithm is slightly different than the version in the text. How and why?
- Can the circle algorithms we've discussed be modified to render circular arcs?
- Alternate circle specifications
  How would they impact the algorithm?
- As in lines, can we remove the *jaggies*?



Drawing Conic Curves

Circles are special cases of a class of curves known as *conics*. Conics are distingished by second-degree descriminating functions of the form:

$$f(x, y) = Ax^2 + Bxy + Cy^2 + Dx + Ey + F$$

Note: Equation 3-50 (pg. 110) is wrong in the book!
(Actually it is just inconsistent with equation 3-51).

The values of the constants, *A, B, C, D, E,* and *F* determines the type of curve as follows:

$B^2 - 4AC < 0$ circle if ($A=C$ and $B = 0$), ellipse otherwise
$B^2 - 4AC = 0$ line if ($A=B=C=0$), parabola otherwise
$B^2 - 4AC > 0$ hyperbola

To make things confusing, mathematicians often refer to the term $B^2 - 4AC$ as the conic discriminant. Here we will stick to the computer graphics definition of a discriminant as a function that partitions interior and exterior half-spaces.

Why and How?

Conics arise frequently in phyical simulations, such as plotting the path of a projectiles under the influence of gravity (a parabola), or the near collisions of like-charged particles (hyperbolas).

Conics, like circles posses symmetry, but not nearly to the same extent. A circle is a very special case of conic, it is so special that it is often considered *non-generic*. Typically a conic will have only one (parabola) or two (ellipse or hyperbola) symmetric axes. As a result, we won't be able to use the same tricks to avoid slope calculations at each point, that we used when drawing circles.

In order to compute the slope at each point we'll need to find derivatives of the disccriminating equation:

$$\frac{\partial f(x,y)}{\partial x} = 2Ax + By + D$$
$$\frac{\partial f(x,y)}{\partial y} = Bx + 2Cy + E$$

Drawing Conics

Using these equations we can compute the instantaneous slope at every point on the conic curve. The next thing we need is an incremental formulations for the descriminating function

$$f(x \pm 1, y) = f(x,y) \pm \frac{\partial f(x,y)}{\partial x} + A$$
$$f(x, y \pm 1) = f(x,y) \pm \frac{\partial f(x,y)}{\partial y} + C$$
$$f(x \pm 1, y \pm 1) = f(x,y) \pm \frac{\partial f(x,y)}{\partial x} \pm \frac{\partial f(x,y)}{\partial y} + A + B + C$$
$$f(x \pm 1, y \mp 1) = f(x,y) \pm \frac{\partial f(x,y)}{\partial x} \mp \frac{\partial f(x,y)}{\partial y} + A - B + C$$

and its derivatives.

$$\frac{\partial f(x \pm 1, y)}{\partial x} = \frac{\partial f(x,y)}{\partial x} \pm 2A, \qquad \frac{\partial f(x \pm 1, y)}{\partial y} = \frac{\partial f(x,y)}{\partial y} \pm B$$
$$\frac{\partial f(x, y \pm 1)}{\partial x} = \frac{\partial f(x,y)}{\partial x} \pm B, \qquad \frac{\partial f(x, y \pm 1)}{\partial y} = \frac{\partial f(x,y)}{\partial y} \pm 2C$$
$$\frac{\partial f(x \pm 1, y \pm 1)}{\partial x} = \frac{\partial f(x,y)}{\partial x} \pm 2A \pm B, \qquad \frac{\partial f(x \pm 1, y \pm 1)}{\partial y} = \frac{\partial f(x,y)}{\partial y} \pm B \pm 2C$$
$$\frac{\partial f(x \pm 1, y \mp 1)}{\partial x} = \frac{\partial f(x,y)}{\partial x} \pm 2A \mp B, \qquad \frac{\partial f(x \pm 1, y \mp 1)}{\partial y} = \frac{\partial f(x,y)}{\partial y} \pm B \mp 2C$$

Conic Drawing Algorithm

So now we know how to compute the discriminator, and the slope at every point along a conic and we can write the conic-drawing routine.

The code is rather long. It must decide at every step whether to step in positive or negative directions in x or y, and whether the slope is less than or greater than 1. This gives us 8 choices at every point corresponding to octants of the coordinate system.

One of the most difficult aspects of using the conic routine is specifying the desired conic. Directly specifying the coefficients, *A, B, C, D, E,* and *F*, is not very intuitive. The interface seen on the right was designed for specifying ellipses. The user gives three points, the first two are on the ellipse and the third specifies the intersection of the major and minor axes.


```
final static int OCTANTS = 0x12650374;

public void conic(
    int x0, int y0,             // starting point
    int x1, int y1,             // ending point
    Color color,                // color of curve
    float A, float B, float C,  // coefficients of conic
    float D, float E, float F)
{
    int pix = color.getRGB();
    int dxDiag, dyDiag, dxLine, dyLine;
    int octant = 0;
    float d, u, v;

    F = 0;
    D = 2*A*x0 + B*y0 + D;
    E = B*x0 + 2*C*y0 + E;

    if (D >= 0) {
        dxDiag = dxLine = 1;
        octant += 1;
    } else {
        dxDiag = dxLine = -1;
    }

    if (E >= 0) {
        dyDiag = dyLine = 1;
        octant += 2;
```

```
                } else {
                    dyDiag = dyLine = -1;
                }

                if (Math.abs(E) > Math.abs(D)) {
                    dxLine = 0;
                    u = dxDiag*dyDiag*B/2 + C + dyDiag*E;
                    d = u + A/4 + dxDiag*D/2 + F;
                    v = u + dxDiag*D;
                    octant += 4;
                } else {
                    dyLine = 0;
                    u = dxDiag*dyDiag*B/2 + A + dxDiag*D;
                    d = u + C/4 + dyDiag*E/2 + F;
                    v = u + dyDiag*E;
                }
                octant = (OCTANTS >> (4*octant)) & 7;

                float k1 = 2*(A + dyLine*dyDiag*(C - A));
                float k2 = dxDiag*dyDiag*B;
                float k3 = 2*(A + C + k2);
                k2 += k1;

        loop:   do {
                    if ((octant & 1) == 0) {
                        while (2*v <= k2) {
                            raster.setPixel(pix, x0, y0);
                            if ((x0 == x1) & (y0 == y1)) break loop;
                            if (d < 0) {
                                x0 += dxLine;
                                y0 += dyLine;
                                u += k1;
                                v += k2;
                                d += u;
                            } else {
                                x0 += dxDiag;
                                y0 += dyDiag;
                                u += k2;
                                v += k3;
                                d += v;
                            }
                        }
                        d = d - u + v/2 - k2/2 + 3*k3/8;
                        u = -u + v - k2/2 + k3/2;
                        v = v - k2 + k3/2;
                        k1 = k1 - 2*k2 + k3;
                        k2 = k3 - k2;
                        int t = dxLine; dxLine = -dyLine; dyLine = t;
                    } else {
                        while (2*u < k2) {
                            raster.setPixel(pix, x0, y0);
                            if ((x0 == x1) & (y0 == y1)) break loop;
                            if (d > 0) {
```



```
                            x0 += dxLine;
                            y0 += dyLine;
                            u += k1;
                            v += k2;
                            d += u;
                        } else {
                            x0 += dxDiag;
                            y0 += dyDiag;
                            u += k2;
                            v += k3;
                            d += v;
                        }
                    }
                    float dk = k1 - k2;
                    d = d + u - v + dk;
                    v = 2*u - v + dk;
                    u = u + dk;
                    k3 = k3 + 4*dk;
                    k2 = k1 + dk;
                    int t = dxDiag; dxDiag = -dyDiag; dyDiag = t;
                }
                octant = (octant + 1) & 7;
            } while (true);
        }
```


Generalizing the Scan Conversion Process

Thus far we have developed several scan conversion algorithms using the same approach:

1. Identify a discriminating function for the primitive
2. Exploit available symmetry of the primitive
3. Figure out how to evaluate the discriminating function incrementally

What's the Future?

This approach is classic as well as very common. However, developments in modern computer systems lead us to question some of these techniques. For example:

- How do we divide up the scan conversion task for parallel implemetation?
- Can we exploit symmetry on parallel processors?
- Does incremental evaluation map well to parallel processors?

Project #2

Next Time

(psst... What IS he talking about?)