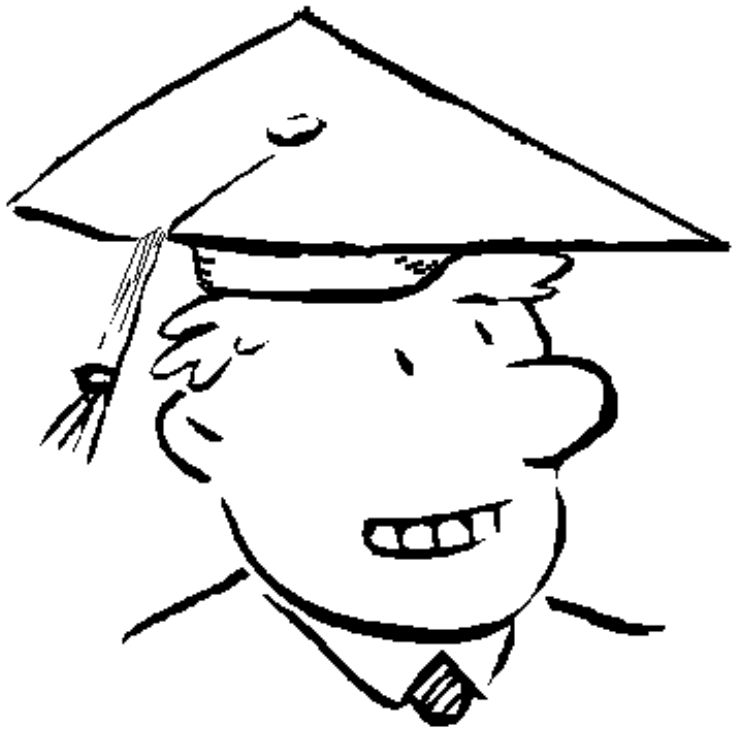
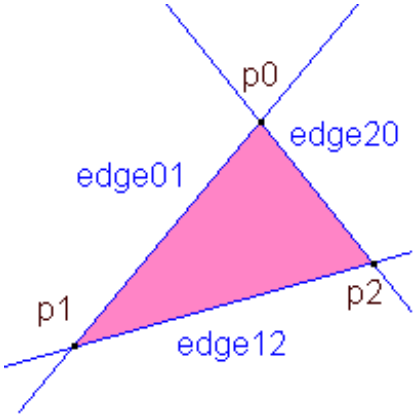
6.837 Lecture 6

Scan Converting Triangles



- Why triangles?
- Rasterizing triangles
- Interpolating parameters
- Post-triangle rendering

Triangles are Minimal:



Triangles are determined by 3 points or 3 edges

We can define a triangle in terms of three points, for example:

$$(x_1, y_1), (x_2, y_2), (x_3, y_3)$$

Or we can define a triangle interm of its three edges, for example:

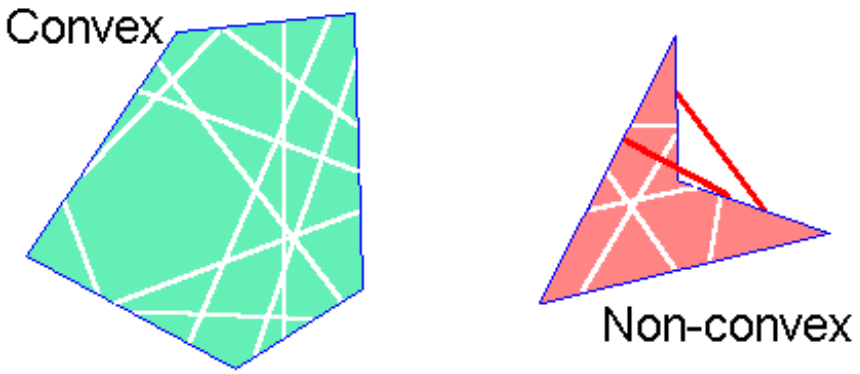
$$A_1 x + B_1 y + C_1 = 0, \quad A_2 x + B_2 y + C_2 = 0, \quad A_3 x + B_3 y + C_3 = 0$$

Why does it seem to take more parameters to represent the edges than the points?
(Hint: Edge equations are Homogeneous)

As a result, triangles are mathematically very simple. The math involved in scan converting triangles involves only simple linear equations.

Triangles are Always *Convex* Polygons

What does it mean to be a convex?



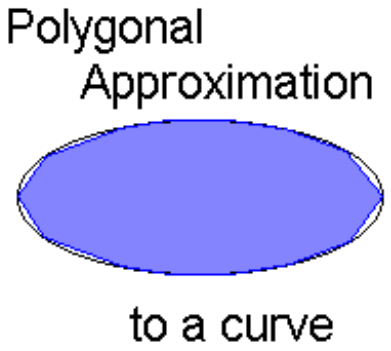
An object is convex if and only if any line segment connecting two points on its boundary is contained entirely within the object or one of its boundries.

Why is being convex important?

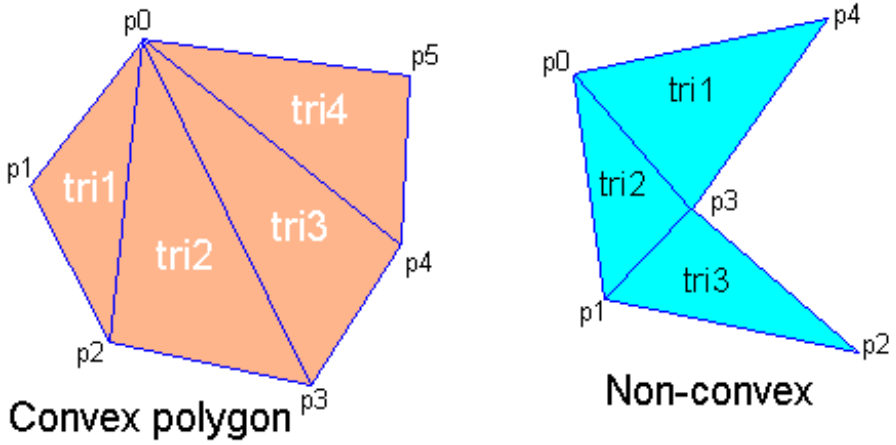
Because no matter how a triangle is oriented on the screen a given scan line will contain only a single segment or *span* of that triangle.

Triangles Can Approximate Arbitrary Shapes

Any 2-dimensional shape (or 3D surface) can be approximated by a polygon using a locally linear (planar) approximation. To improve the quality of fit we need only increase the number edges.



Any Polygon can be Decomposed into Triangles



A convex n-sided polygon, with ordered vertices $\{v_0, v_1, \ldots v_n\}$ along the perimeter, can be trivially decomposed into triangles $\{(v_0,v_1,v_2), \{v_0,v_2,v_3), (v_0,v_i,v_{i+1}), \ldots (v_0,v_{n-1},v_n)\}$.

You can usually decompose a non-convex polygon into triangles, but it is non-trivial, and in some overlapping cases you have to introduce a new vertex.

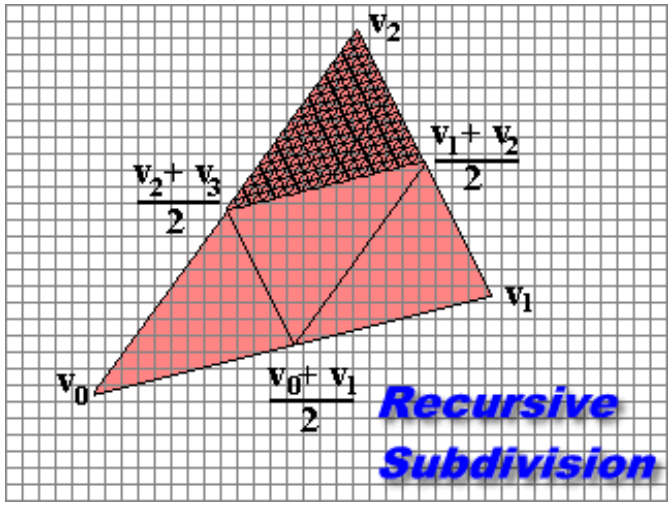
Rasterizing Triangles

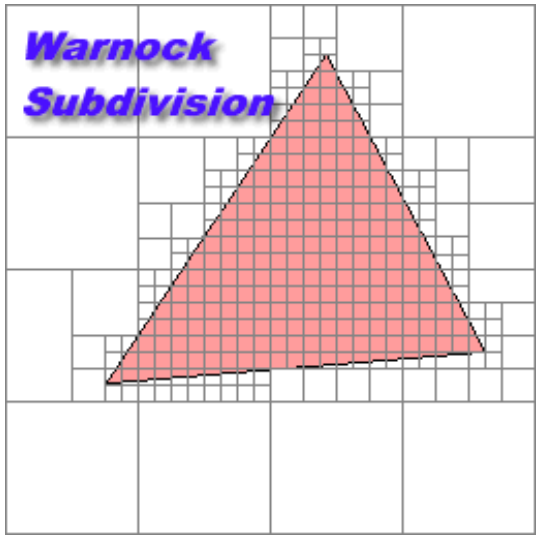
The two most common strategies for scan-converting a triangle are

edge walking and *edge equations*

There are, however, other techniques including:

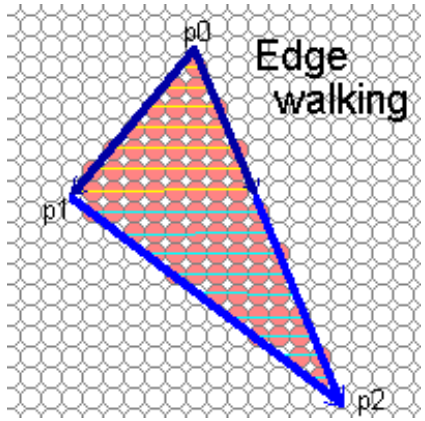
- Recursive subdivision of primitive (micropolygons)
- Recursive subdivision of screen (Warnock's algorithm)





Edge-Walking Triangle Rasterizer

Notes on edge walking:



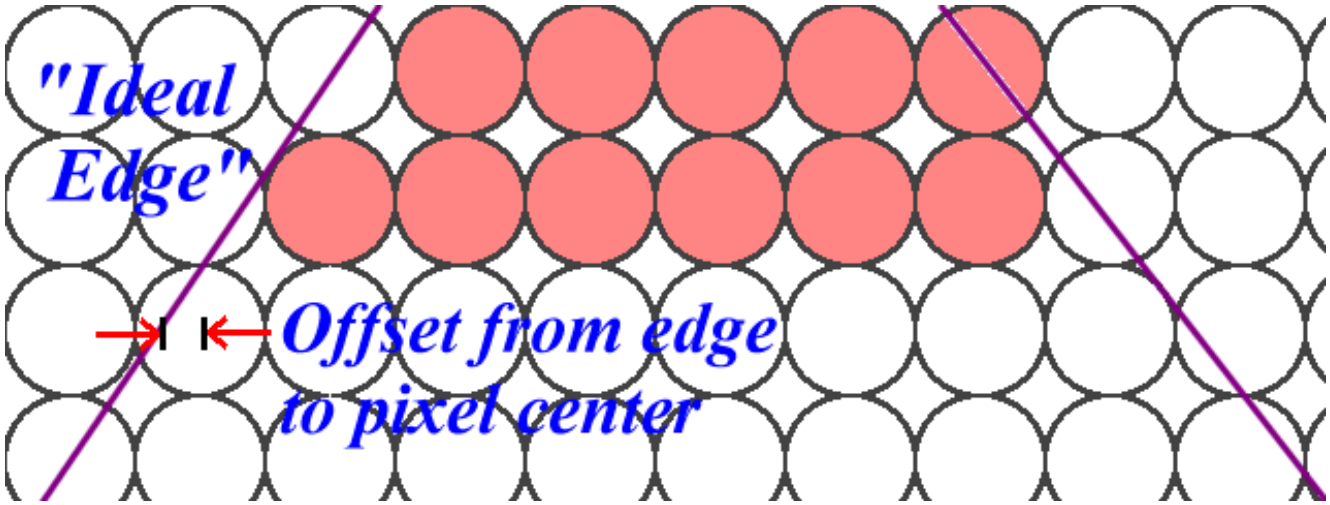
- Sort the vertices in both x and y
- Determine if the middle vertex, or *breakpoint* lies on the left or right side of the polygon. If the triangle has an edge parallel to the scanline direction then there is no breakpoint.
- Determines the left and right extents for each scanline (called *spans*).
- Walk down the left and right edges filling the pixels in-between until either a breakpoint or the bottom vertex is reached.

Advantages and Disadvantages:

- Generally very fast
- Loaded with special cases (left and right breakpoints, no breakpoints)
- Difficult to get right
- Requires computing fractional offsets when interpolating parameters across the triangle

The algorithm described in the book is an edge-walking algorithm.

Fractional Offsets



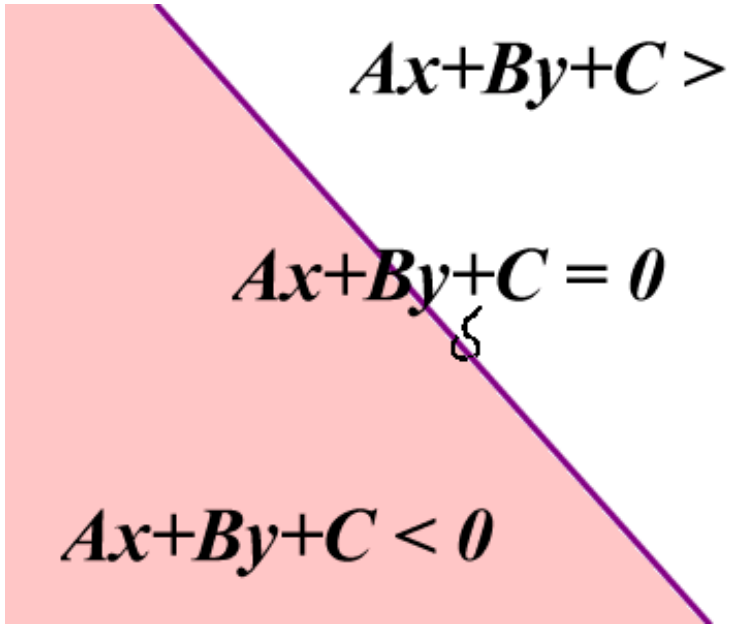
We can use *ceiling* to find the leftmost pixel in span and *floor* to find the rightmost pixel.

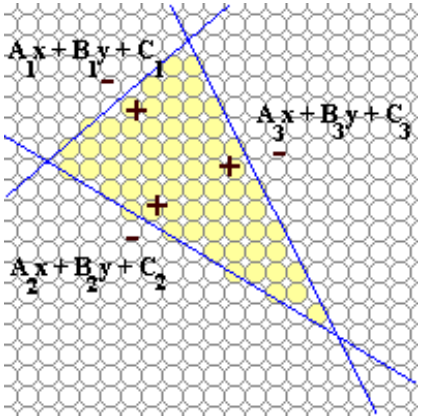
The trick comes when interpolating color values. It is straightforward to interpolate along the edges, but you must be careful when offsetting from the edge to the pixel's center.

Rasterizing Triangles using Edge Equations

An edge equation is simply a **discriminating function** like those we used in our curve and line-drawing algorithms.

An edge equation segments a planar region into three parts, a boundary, and two half-spaces. The boundary is identified by points where the edge equation is equal to zero. The half-spaces are distiguished by differences in the edge equation's sign. We can choose which half-space gives a positive sign by multiplication by -1.

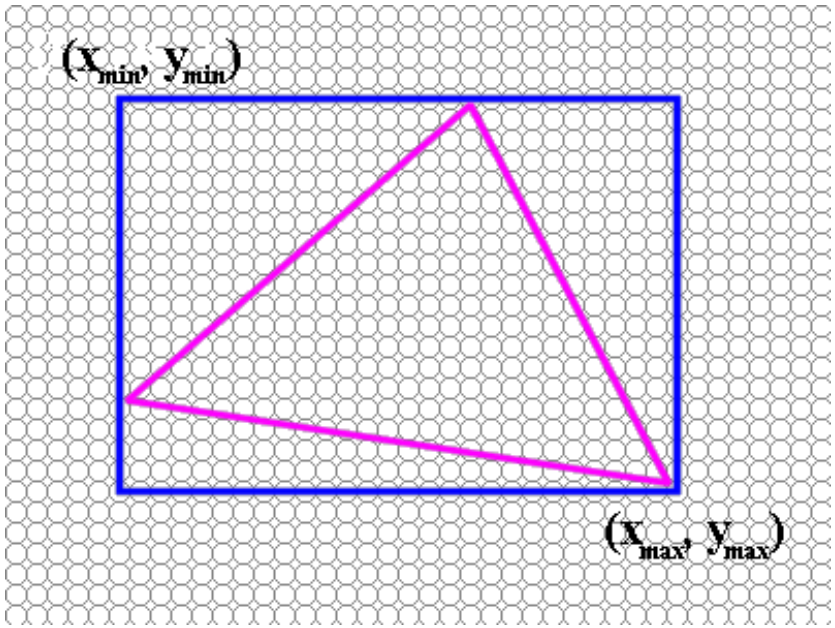




We can scale all three edges so that their negative halfspaces are on the triangle's exterior.

Notes on using Edge Equations

- Compute edge equations from vertices
- Orient edge equations
- Compute a bounding box
- Scan through pixels in bounding box evaluating the edge equations
- When all three are positive then draw the pixel.



Example Implementation

First we define a few useful objects

```
public class Vertex2D {
    public float x, y;              // coordinate of vertex
    public int argb;                // color of vertex

    public Vertex2D() {
    }

    public Vertex2D(float xval, float yval, int cval) {
        x = xval;
        y = yval;
        argb = cval;
    }
}
```

A *Drawable* interface

```
import Raster;

public abstract interface Drawable {
    public abstract void Draw(Raster r);
}
```

The edge equations use integer coefficients for speed. This implementation uses 12 fractional bits. (Not my first version)

Edge Equation Coefficients

The edge equation coefficients are computed using the coordinates of the two vertices. Each point determines an equation in terms of our three unknowns, *A*, *B*, and *C*.

$$Ax_0 + By_0 + C = 0$$
$$Ax_1 + By_1 + C = 0$$

We can solve for A and B in terms of C by setting up the following linear system.

$$\begin{bmatrix} x_0 & y_0 \\ x_1 & y_1 \end{bmatrix} \begin{bmatrix} A \\ B \end{bmatrix} = -C \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

Multiplying both sides by the matrix inverse.

$$\begin{bmatrix} A \\ B \end{bmatrix} = \frac{-C}{x_0 y_1 - x_1 y_0} \begin{bmatrix} y_1 & -y_0 \\ -x_1 & x_0 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

If we choose $C = x_0 y_1 - x_1 y_0$, then we get $A = y_0 - y_1$ and $B = x_1 - x_0$.

Why could we just choose *C*?

Numerical Precision of the *C* Coefficient

Computers represent floating-point number internally in a format similar to scientific notation. The very worse thing that you can do with numbers represented in scientific notation is subtract numbers of similar magnitude. We loose most of the significant digits in our result (see next slide).

In the case of triangles, these sort of precision problems to occur frequently, because in general the vertices of a triangle are close to each other.

$$x_0 \approx x_1 \text{ and } y_0 \approx y_1 \text{ thus } x_0 y_1 - x_1 y_0 \approx 0$$

Thankfully, we can avoid this subtraction of large numbers when computing an expression for *C*. Given that we know *A* and *B* we can solve for *C* as follows:

$$C_0 = -Ax_0 - By_0 \text{ or } C_1 = -Ax_1 - By_1$$

To eliminate any bias toward either vertex we will average of these C values

$$C_{ave} = \frac{-(A(x_0 + x_1) + B(y_0 + y_1))}{2}$$

Loss of Precision

```
public class Jtest {
  public static void main (String args []) {
    float x, h, dydx, correct, relError;
    x = 0.1f;
    h = 10.0f;
    correct = (float)Math.cos(x);

    for(int i = 0; i < 15; i ++) {
      dydx = ((float)Math.sin(x + h) - (float)Math.sin(x - h))/ (2.0f * h);
      relError = 100.0f * (dydx - correct)/correct;
      System.out.println("i="+i+" h="+h+" dydx="+dydx+" rel. error="+relError);
      h /= 10.0f;
    }
    System.out.println("Correct value of derivative is: "+correct);
  }
}
i=0  h=10.0          dydx=-0.054130327 rel. error=-105.440216
i=1  h=1.0           dydx=0.83726716   rel. error=-15.8529
i=2  h=0.1           dydx=0.99334663   rel. error=-0.1665868
i=3  h=0.01          dydx=0.9949874    rel. error=-0.0016833
i=4  h=9.999999E-4   dydx=0.9950065    rel. error=2.3362527E-4
i=5  h=9.999999E-5   dydx=0.9949879    rel. error=-0.0016353768
i=6  h=9.999999E-6   dydx=0.99502516   rel. error=0.0021086177
i=7  h=9.999999E-7   dydx=0.9946526    rel. error=-0.035331327
i=8  h=9.999999E-8   dydx=0.9313227    rel. error=-6.400122
i=9  h=9.999999E-9   dydx=0.7450581    rel. error=-25.120102
i=10 h=9.999999E-10  dydx=0.0          rel. error=-100.0
i=11 h=9.999999E-11  dydx=0.0          rel. error=-100.0
i=12 h=9.999999E-12  dydx=0.0          rel. error=-100.0
i=13 h=9.999999E-13  dydx=0.0          rel. error=-100.0
i=14 h=9.999999E-14  dydx=0.0          rel. error=-100.0
Correct value of derivative is: 0.9950042
```

EdgeEquation Object

```
class EdgeEqn {
    public final static int FRACBITS = 12;
    public int A, B, C;
    public int flag;            // used to compute bounding box

    public EdgeEqn(Vertex2D v0, Vertex2D v1) {
        double a = v0.y - v1.y;
        double b = v1.x - v0.x;
        double c = -0.5f*(a*(v0.x + v1.x) + b*(v0.y + v1.y));
        A = (int) (a * (1<<FRACBITS));
        B = (int) (b * (1<<FRACBITS));
        C = (int) (c * (1<<FRACBITS));
        flag = 0;
        if (A >= 0) flag += 8;
        if (B >= 0) flag += 1;
    }

    public void flip() {
        A = -A;
        B = -B;
        C = -C;
    }

    public int evaluate(int x, int y) {
        return (A*x + B*y + C);
    }
}
```

Triangle Object

```
public class FlatTri implements Drawable {
    protected Vertex2D v[];
    protected EdgeEqn edge[];
    protected int color;
    protected int area;
    protected int xMin, xMax, yMin, yMax;
    private static byte sort[][] = {{0, 1}, {1, 2}, {0, 2}, {2, 0}, {2, 1}, {1, 0}};

    public FlatTri() {      // for future extension
    }

    public FlatTri(Vertex2D v0, Vertex2D v1, Vertex2D v2) {
        v = new Vertex2D[3];
        v[0] = v0;
        v[1] = v1;
        v[2] = v2;

        //  ... Paint triangle the average of it's vertex colors ...
        int a = ((v0.argb >> 24) & 255) + ((v1.argb >> 24) & 255) + ((v2.argb >> 24) & 255);
        int r = ((v0.argb >> 16) & 255) + ((v1.argb >> 16) & 255) + ((v2.argb >> 16) & 255);
        int g = ((v0.argb >> 8) & 255) + ((v1.argb >> 8) & 255) + ((v2.argb >> 8) & 255);
        int b = (v0.argb & 255) + (v1.argb & 255) + (v2.argb & 255);

        a = (a + a + 3) / 6;
        r = (r + r + 3) / 6;
        g = (g + g + 3) / 6;
        b = (b + b + 3) / 6;

        color = (a << 24) | (r << 16) | (g << 8) | b;
    }
```

The Draw Method

```
public void Draw(Raster r) {
    int width = r.getWidth();
    int height = r.getHeight();
    if (!triangleSetup(width, height)) return;

    int x, y;
    int A0 = edge[0].A;     int B0 = edge[0].B;
    int A1 = edge[1].A;     int B1 = edge[1].B;
    int A2 = edge[2].A;     int B2 = edge[2].B;

    int t0 = A0*xMin + B0*yMin + edge[0].C;
    int t1 = A1*xMin + B1*yMin + edge[1].C;
    int t2 = A2*xMin + B2*yMin + edge[2].C;

    yMin *= width;
    yMax *= width;

    //     .... scan convert triangle ....
    for (y = yMin; y <= yMax; y += width) {
            int e0 = t0;
            int e1 = t1;
            int e2 = t2;
            boolean beenInside = false;
            for (x = xMin; x <= xMax; x++) {
                if ((e0|e1|e2) >= 0) {       // all 3 edges must be >= 0
                        r.pixel[y+x] = color;
                        beenInside = true;
                } else if (beenInside) break;
                e0 += A0;
                e1 += A1;
                e2 += A2;
            }
            t0 += B0;
            t1 += B1;
            t2 += B2;
    }
}
```

Explain Two Speed Hacks

Most everything in our *Draw( )* method is straightforward, with two exceptions.

```
int xflag = 0;
for (x = xMin; x <= xMax; x++) {
    if ((e0|e1|e2) >= 0) {
        r.pixel[y+x] = color;
        xflag++;
    } else if (xflag != 0) break;
    e0 += A0;
    e1 += A1;
    e2 += A2;
}
```

- All three edges are tested with a single comparison by oring together the three edges and checking if the result is positive. If any one of the three is negative then its sign-bit will be set to a 1, and the result of the or will be negative.

- Since triangles are convex, we can only be inside for a single interval on any given scanline. The *xflag* variable is used to keep track of when we exit the triangle's interior. If ever we find ourselves outside of the triangle having already set some pixels on the span then we can skip over the remainder of the scanline.

Triangle SetUp Method

```
protected boolean triangleSetup(int width, int height) {
    if (edge == null) edge = new EdgeEqn[3];
    //   Compute the three edge equations
    edge[0] = new EdgeEqn(v[0], v[1]);
    edge[1] = new EdgeEqn(v[1], v[2]);
    edge[2] = new EdgeEqn(v[2], v[0]);

    //    Trick #1: Orient edges so that the triangle's interior lies within all of their
    //    positive half-spaces. Assuring that the area is positive accomplishes this
    area = edge[0].C + edge[1].C + edge[2].C;
    if (area == 0) {
        return false;               // degenerate triangle
    }
    if (area < 0) {
        edge[0].flip();
        edge[1].flip();
        edge[2].flip();
        area = -area;
    }

    //    Trick #2: compute bounding box
    int xflag = edge[0].flag + 2*edge[1].flag + 4*edge[2].flag;
    int yflag = (xflag >> 3) - 1;
    xflag = (xflag & 7) - 1;

    xMin = (int) (v[sort[xflag][0]].x);
    xMax = (int) (v[sort[xflag][1]].x + 1);
    yMin = (int) (v[sort[yflag][1]].y);
    yMax = (int) (v[sort[yflag][0]].y + 1);

    //   clip triangle's bounding box to raster
    xMin = (xMin < 0) ? 0 : xMin;
    xMax = (xMax >= width) ? width - 1 : xMax;
    yMin = (yMin < 0) ? 0 : yMin;
    yMax = (yMax >= height) ? height - 1 : yMax;
    return true;
}
```

Tricks Explained

In this method we did two critical things. We orient the edge equations, and we compute the bounding box.

From analytic geometry we know that the area of a triangle $\{(x_1,y_1), (x_2,y_2), (x_3,y_3)\}$ is:

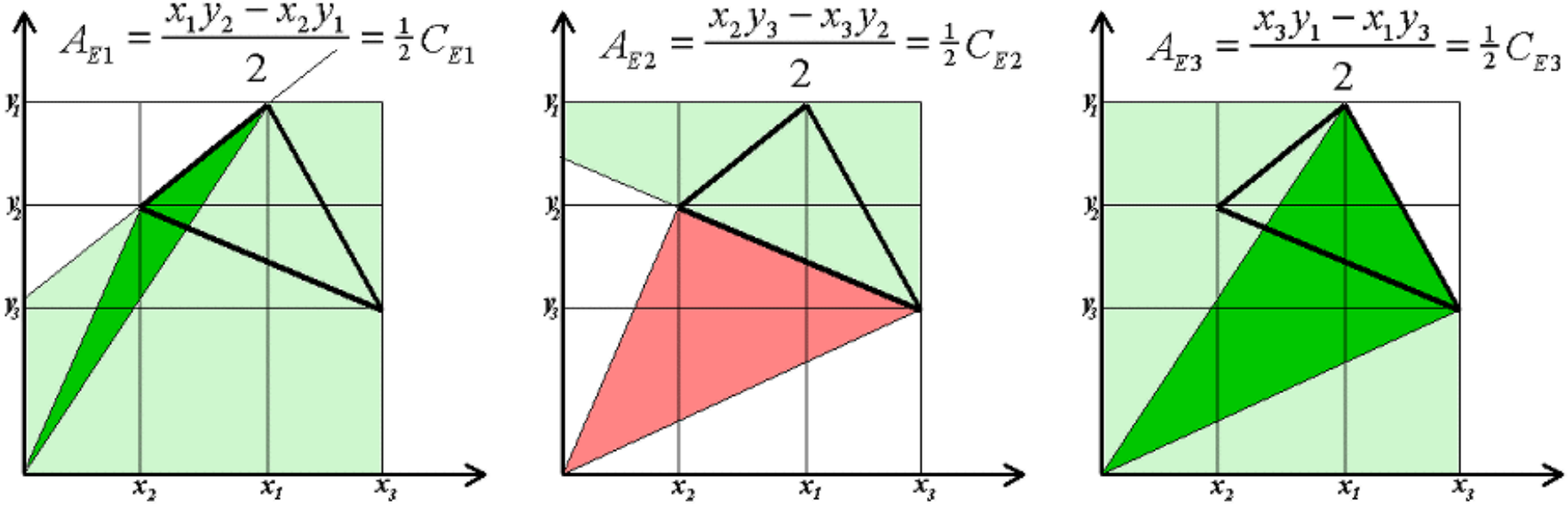
$$Area = \frac{1}{2}\det\begin{bmatrix} x_1 & x_2 & x_3 \\ y_1 & y_2 & y_3 \\ 1 & 1 & 1 \end{bmatrix}$$

The area is positive if the vertices are counterclockwise and negative if clockwise

An aside:

In terms of our discriminator, what does a positive *C* imply?

Why Positive Area Implies a Positive Interior



1. The area of each sub-triangle gives the edge equation's sign at the origin
2. Assume a positive area
   thus, the sum of the sub-triangles areas are positive
3. Each point within the triangle falls within exactly one subtriangle
   thus, subtriangles with negative areas will lie outside of the triangle
4. Since the negative-subtriangle areas are outside of the triangle
   the edge equations are positive inside

Exercise #2

Examine the code for computing the bounding box of the triangle in *EdgeEqn( )* and *FlatTri.triangleSetup( )*.

- Explain what is being saved in the *EdgeEqn.flag*
- Explain the contents *FlatTri.sort[ ]*
- Explain how the bounding box is computed
- Discuss the advantages and disadvantages of this approach
- Write down *I give up* if this exercise takes you more than 1 Hr.

Limit your discussion to one single-sided sheet of 8.5 by 11 paper.

Turn this exercise in the next time that we meet.

Triangle Rasterizer Demonstration

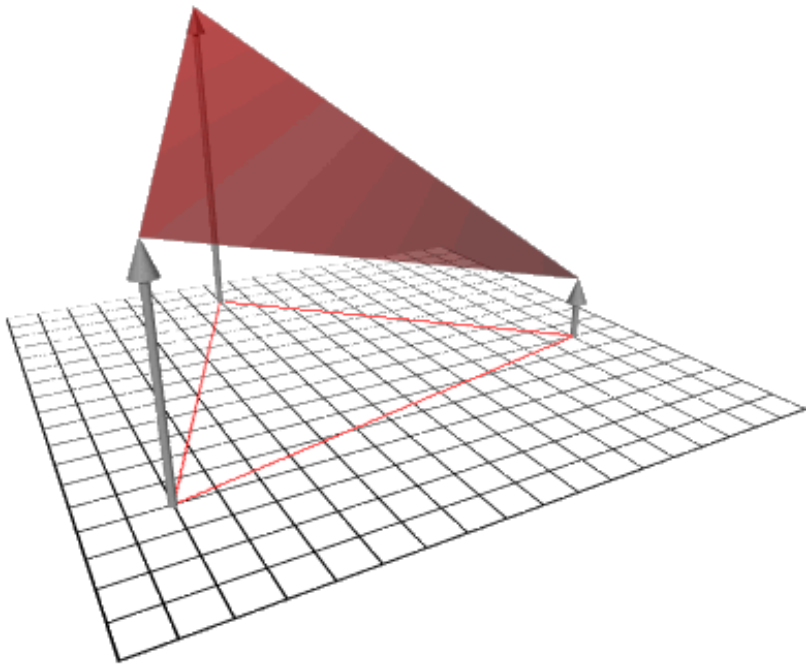
Press the left mouse button above to render a simple scene with the FlatTri rasterizer.

Press the left mouse button above to render a more complicated scene with the FlatTri rasterizer.

Interpolating Parameters Within a Triangle

Currently, our triangle scan-converter draws only solid colored triangles. Next we'll discuss how to smoothly vary parameters as we fill the triangle. In this case the parameters that are interpolated are the red, green, and blue components of the color. Later on, when we get to 3D techniques, we'll also interpolate other parameters such as the depth at each point on the triangle.

First, let's frame the problem. At each vertex of a triangle we have a parameter, say its redness. When we actually draw the vertex, the specified shade of red is exactly what we want, but at other points we'd like some sort of smooth transition between the values given. This situation is shown to the right:



Notice that the shape of our desired redness function is planar. Actually, it is a special class of plane where there exists a corresponding point for every x-y coordinate. Planes of this type can always be expressed in the following form:

$$z = Ax + By + C$$

This equation should appear familiar. It has the same form as our edge equations.

Given the redness of the three vertices, we can set up the following linear system.

$$\begin{bmatrix} r_0 \\ r_1 \\ r_2 \end{bmatrix} = \begin{bmatrix} x_0 & y_0 & 1 \\ x_1 & y_1 & 1 \\ x_2 & y_2 & 1 \end{bmatrix} \begin{bmatrix} A_r \\ B_r \\ C_r \end{bmatrix}$$

with the solution:

$$\frac{1}{2area} \begin{bmatrix} y_1 - y_2 & y_2 - y_0 & y_0 - y_1 \\ x_2 - x_1 & x_0 - x_2 & x_1 - x_0 \\ x_1 y_2 - x_2 y_1 & x_2 y_0 - x_0 y_2 & x_0 y_1 - x_1 y_0 \end{bmatrix} \begin{bmatrix} r_0 \\ r_1 \\ r_2 \end{bmatrix} = \begin{bmatrix} A_r \\ B_r \\ C_r \end{bmatrix}$$

By the way, we've already computed these matrix entries, *they're exactly the coefficients of our edge equations*.

$$\frac{1}{2area} \begin{bmatrix} A_2 & A_3 & A_1 \\ B_2 & B_3 & B_1 \\ C_2 & C_3 & C_1 \end{bmatrix} \begin{bmatrix} r_0 \\ r_1 \\ r_2 \end{bmatrix} = \begin{bmatrix} A_r \\ B_r \\ C_r \end{bmatrix}$$

So the all the additional work that we need to do to interpolate is a single matrix multiplication and compute the equivalent of an extra edge equation for each parameter.

Smooth Triangle Object

```
import Raster;
import Drawable;
import Vertex2D;
import FlatTri;

public class SmoothTri extends FlatTri implements Drawable {
    boolean isFlat;
    double scale;

    public SmoothTri(Vertex2D v0, Vertex2D v1, Vertex2D v2) {
        v = new Vertex2D[3];
        v[0] = v0;
        v[1] = v1;
        v[2] = v2;

        /*
            check if all vertices are the same color
        */
        isFlat = (v0.argb == v1.argb) && (v0.argb == v2.argb);

        if (isFlat) {
            color = v0.argb;
        } else {
            color = 0;
        }

        /*
            Scale is always non zero and positive. This zero
            value indicates that it has not been computed yet
        */
        scale = -1;
    }
```

Computing Plane Equations

We've added two new instance variables. The first is simply an optimization that detects the case when all three vertices are the same color. In this case we'll call the slightly faster FlatTri methods that we inherited. The second is a scale factor that we'll disscuss next.

Next we add a new method to compute the plane equations of our parameters. The *PlaneEqn()* method performs the required matrix multiply and avoids computing the inverse of the triangle area more than once.

```
public void PlaneEqn(int eqn[], int p0, int p1, int p2) {
    int Ap, Bp, Cp;
    if (scale <= 0) {
        scale = (1 << EdgeEqn.FRACBITS) / ((double) area);
    }
    double sp0 = scale * p0;
    double sp1 = scale * p1;
    double sp2 = scale * p2;
    Ap = (int)(edge[0].A*sp2 + edge[1].A*sp0 + edge[2].A*sp1);
    Bp = (int)(edge[0].B*sp2 + edge[1].B*sp0 + edge[2].B*sp1);
    Cp = (int)(edge[0].C*sp2 + edge[1].C*sp0 + edge[2].C*sp1);
    eqn[0] = Ap;
    eqn[1] = Bp;
    eqn[2] = Ap*xMin + Bp*yMin + Cp + (1 << (EdgeEqn.FRACBITS - 1));
}
```

Modified *Draw( )* Method

Compute the plane equation for each parameter.

```
public void Draw(Raster raster) {
    if (isFlat) {
        super.Draw(raster);
        return;
    }

    int width = raster.getWidth();
    int height = raster.getHeight();
    if (!triangleSetup(width, height)) return;

    int alpha[] = new int[3];
    int red[] = new int[3];
    int green[] = new int[3];
    int blue[] = new int[3];

    int t0 = v[0].argb;
    int t1 = v[1].argb;
    int t2 = v[2].argb;
    PlaneEqn(blue, t0 & 255, t1 & 255, t2 & 255);
    t0 >>= 8;   t1 >>= 8;   t2 >>= 8;
    PlaneEqn(green, t0 & 255, t1 & 255, t2 & 255);
    t0 >>= 8;   t1 >>= 8;   t2 >>= 8;
    PlaneEqn(red, t0 & 255, t1 & 255, t2 & 255);
    t0 >>= 8;   t1 >>= 8;   t2 >>= 8;
    PlaneEqn(alpha, t0 & 255, t1 & 255, t2 & 255);
```

More SmoothTri.Draw( )

Add accumulators for each parameter

```
int x, y;
int A0 = edge[0].A; int A1 = edge[1].A; int A2 = edge[2].A;
int Aa = alpha[0];
int Ar = red[0];    int Ag = green[0];  int Ab = blue[0];

int B0 = edge[0].B; int B1 = edge[1].B; int B2 = edge[2].B;
int Ba = alpha[1];
int Br = red[1];    int Bg = green[1];  int Bb = blue[1];

t0 = A0*xMin + B0*yMin + edge[0].C;
t1 = A1*xMin + B1*yMin + edge[1].C;
t2 = A2*xMin + B2*yMin + edge[2].C;
int ta = alpha[2];
int tr = red[2];    int tg = green[2];  int tb = blue[2];
yMin *= width;
yMax *= width;
```

Even More SmoothTri.Draw( )

The inner loop

```
/*
    .... scan convert triangle ....
*/
for (y = yMin; y <= yMax; y += width) {
    int e0 = t0;    int e1 = t1;    int e2 = t2;    int xflag = 0;
    int a = ta;     int r = tr;     int g = tg;     int b = tb;
    for (x = xMin; x <= xMax; x++) {
        if ((e0|e1|e2) >= 0) {      // all 3 edges must be >= 0
            int pixa = (a >> EdgeEqn.FRACBITS);
            int pixr = (r >> EdgeEqn.FRACBITS);
            int pixg = (g >> EdgeEqn.FRACBITS);
            int pixb = (b >> EdgeEqn.FRACBITS);
            pixa = ((pixa & ~255) == 0) ? pixa << 24 : ((a < 0) ? 0 : 0xff000000);
            pixr = ((pixr & ~255) == 0) ? pixr << 16 : ((r < 0) ? 0 : 0x00ff0000);
            pixg = ((pixg & ~255) == 0) ? pixg << 8  : ((g < 0) ? 0 : 0x0000ff00);
            pixb = ((pixb & ~255) == 0) ? pixb       : ((b < 0) ? 0 : 0x000000ff);
            raster.pixel[y+x] = (pixa | pixr | pixg | pixb);
            xflag++;
        } else if (xflag != 0) break;
            e0 += A0;   e1 += A1;   e2 += A2;
            a += Aa;    r += Ar;    g += Ag;    b += Ab;
    }
    t0 += B0;   t1 += B1;   t2 += B2;
    ta += Ba;   tr += Br;   tg += Bg;   tb += Bb;
}
```

Smooth Triangle Results

Press the left mouse button above to render a simple scene with the SmoothTri rasterizer.

Press the left mouse button above to render a more complicated scene with the SmoothTri rasterizer.

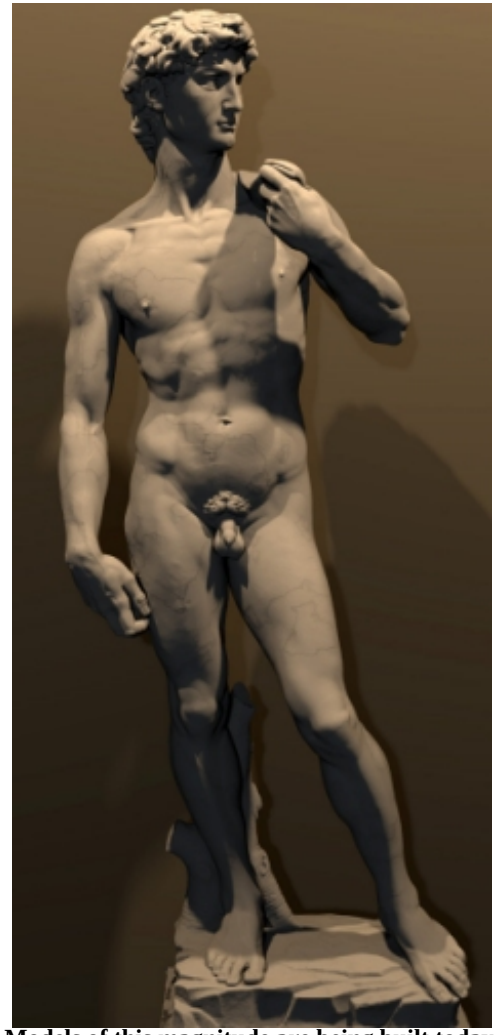
Today's Code: Vertex2D.java, Drawable.java, FlatTri.java, SmoothTri.java, and Triangles.java

A Post-Triangle World

Are triangles really the best rendering primitive?

100,000,000 primitive models displayed on 2,000,000 pixel displays.

Even even if we assume that only 10% of the primities are visible, and they are uniformly distributed over the whole screen, thats still 5 primitives/pixel. Remember, that in order to draw a single triangle we must specify 3 vertices, determine three colors, and interpolte within 3 edges. On average, these triangle will impact only a fraction of a pixel.



**Models of this magnitude are being built today.
The leading and most ambitious work in this area is Stanford's "Digital Michelangelo Project".
Click on the image above to find out more.**

Point-Cloud Rendering

A new class of rendering primitives have recently been introduced to address this problem. Key Attibutes:

- Hierarchy
- Incremental Refinement
- Compact Reperesentation (differental encoding)


130,712 Splats, 132 mS


259,975 Splats, 215 mS


1,017,149 Splats, 722 mS


14,835,967 Splats, 8308 mS

Next Time