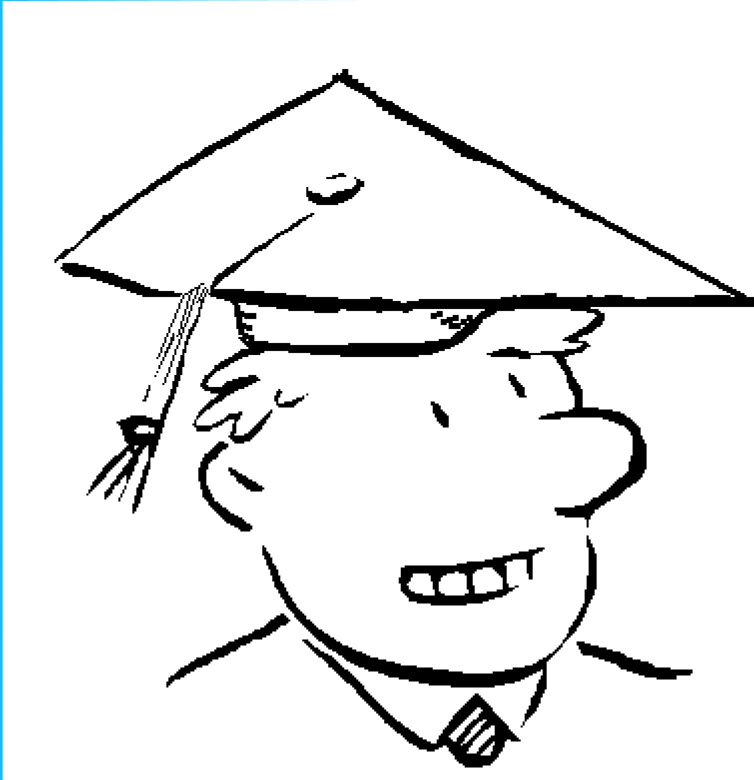
Scan Converting Triangles



Lecture 13
CISC 440/640
Spring 2015

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

Computer Graphics UD

Department of Computer and Information Science

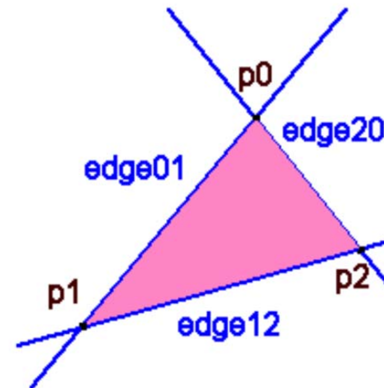
Primitive Rasterization

- 2D SAMPLING problem
- Which pixels (samples) of an "ideal" analytical primitive should be turned on? Based on
  - Closeness of the primitive to the sample to the sample point
  - Percentage coverage of an aperture
  - Whether the sample is inside or outside of a closed primitive

Why Triangles?

- Triangles are minimal!
  - Triangles are determined by 3 points or 3 edges
  - We can define a triangle in terms of three points: (x_1, y_1), (x_2, y_2), and (x_3, y_3)
  - Or we can define a triangle in terms of its three edges

$$A_1x + B_1y + C_1 = 0$$
$$A_2x + B_2y + C_2 = 0$$
$$A_3x + B_3y + C_3 = 0$$

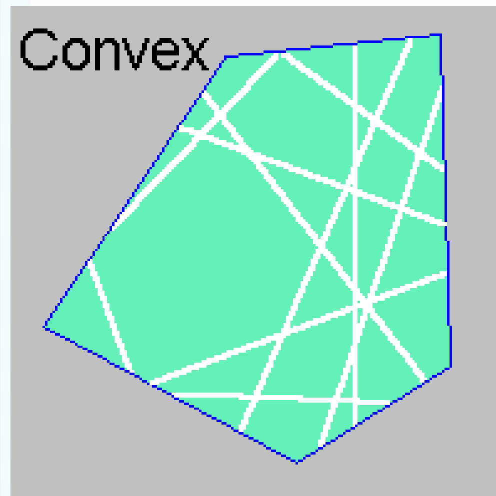


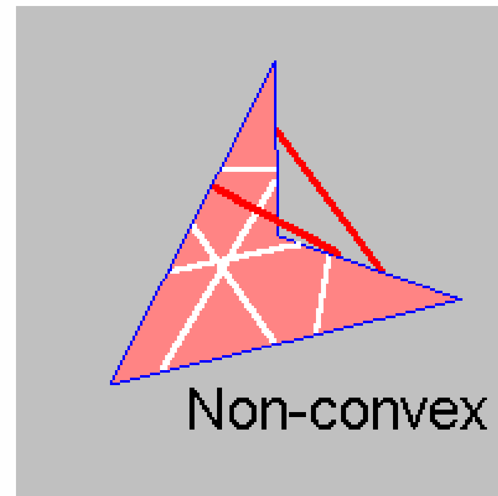
  - Why does it appear to take more parameters to represent the edges than the points?
- As a result, triangles are mathematically very simple.
- Scan converting triangles involves simple linear equations

Computer Graphics UD

Triangles are always Convex

- What does it mean to be a convex?

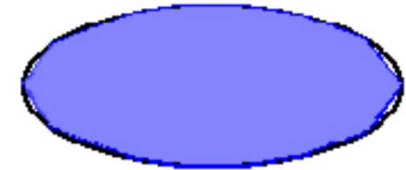




- Why is being convex important?
  - Because no matter how a triangle is oriented on the screen, a given scan line will contain only a single segment or span of that triangle

Triangles can approximate arbitrary shapes

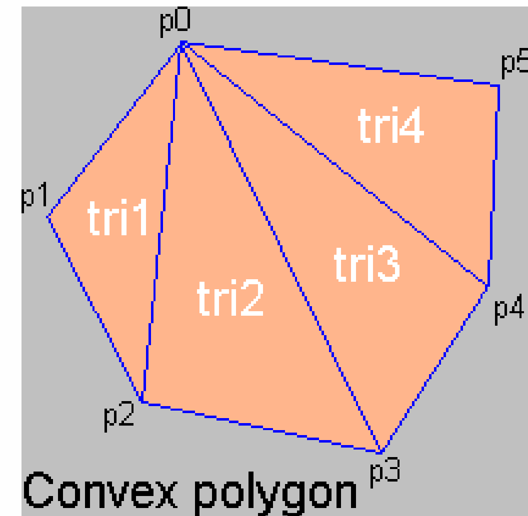
- Any 2-dimensional shape (or 3D surface) can be approximated by a polygon using a locally linear (planar) approximation. To improve the quality of fit, we need only increase the number of edges

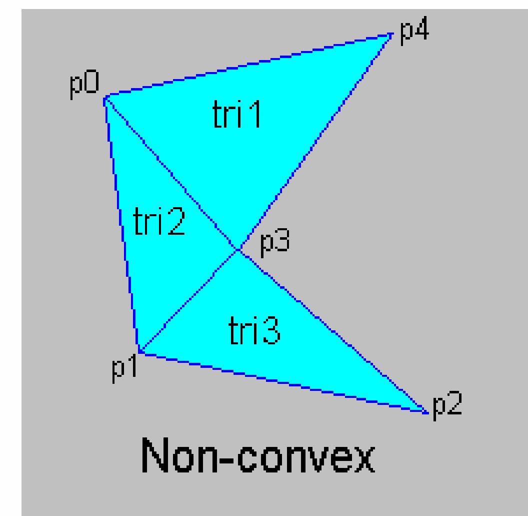


Polygons can be Decomposed into Triangles

- A convex n-sided polygon, with ordered vertices $\{p_0, p_1, \ldots p_n\}$ along the perimeter, can be trivially decomposed into triangles

  $\{(p_0,p_1,p_2), (p_0,p_2,p_3), \ldots(p_0,p_{n-1},p_n)\}$
- 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

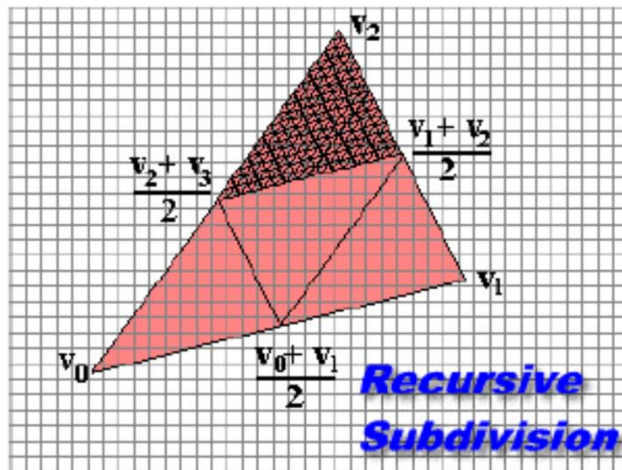


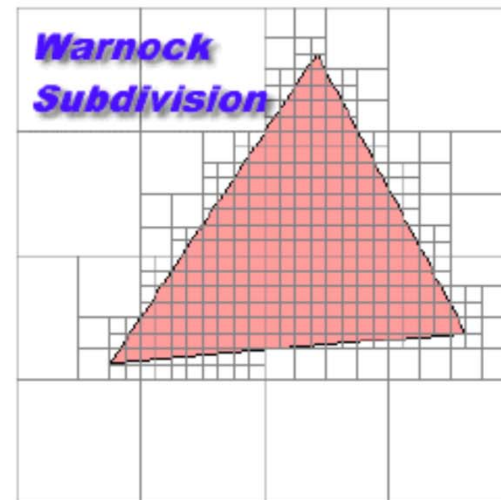


Computer Graphics UD

Scan-converting Triangles

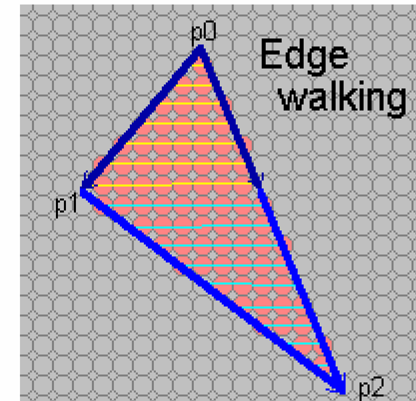
- The two most common strategies for scan-converting triangles are *edge walking* and *edge equations*
- There are, however, other techniques including:
  - Recursive subdivision of primitive (micro-polygons)
  - Recursive subdivision of screen (Warnock's algorithm)





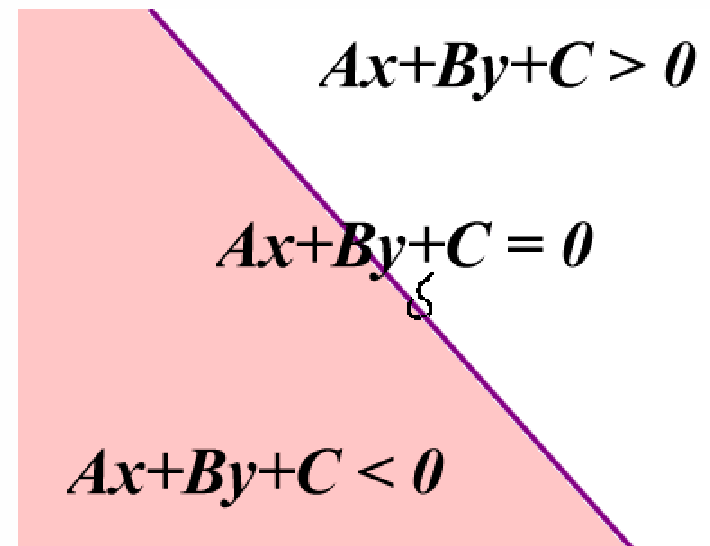
Edge-Walking Triangle Rasterizer

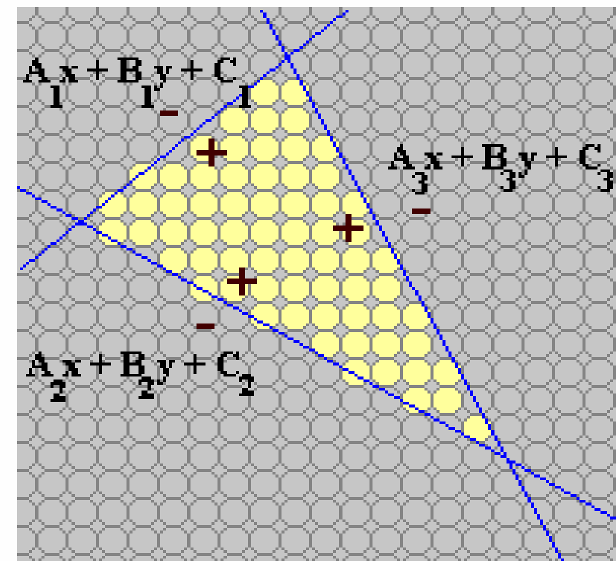
- Notes on edge walking:
  - Sort the vertices in both x and y
  - Determine if the middle vertex, or breakpoint, lies on the left or right side of the polygon.
  - If the triangle has an edge parallel to the scan line direction, then there is no breakpoint
  - Determine the left and right extents for each scan line (called *spans*).
  - Walk down the left and right edges filling the pixel 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



Rasterizing Triangles with Edge Equations

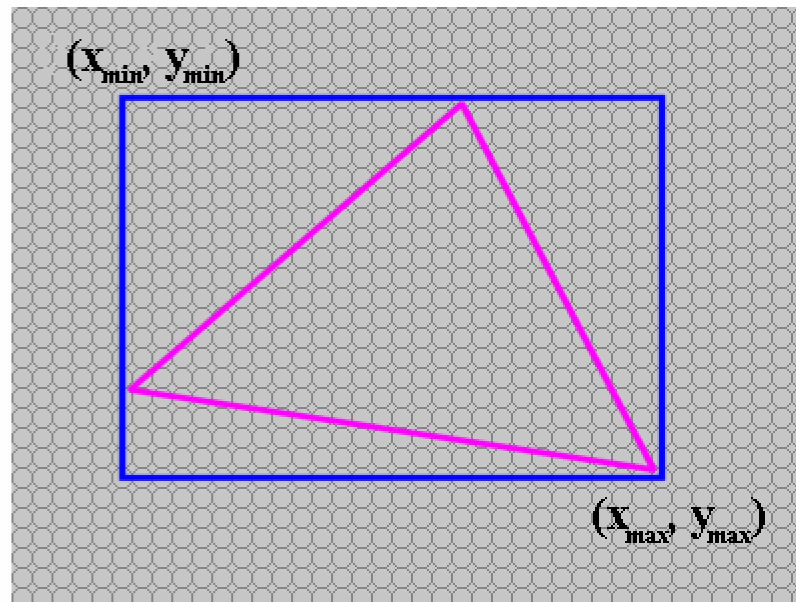
- An edge equation is simply a discriminating function
- You can think of it as answering a question about any given point (x, y)
- An edge equation segments a planar region into three parts, a boundary, and two half-spaces. The boundary is identified by points where the edge equation is equal to zero. The half spaces are distinguished by differences in the edge equation's sign. We can choose which half-space is positive by multiplication by -1.
- We can scale all three edges so that their negative half-spaces are on the triangle's exterior.





Notes on using Edge Equations

- Compute edge equations from vertices
- Orient edge equations (make the interior region positive)
- 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
- An “abstract” Drawable class
- A 2D vertex, with and without a color

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

public class Vertex2D : Drawable {
    public double x, y;
    public Color color;

    public Vertex2D(double xval, double yval) {
        x = xval;    y = yval;
    }

    public Vertex2D(double xval, double yval, Color cval) {
        x = xval;    y = yval;   color = cval;
    }

    public override void Draw(Raster r) {
        r.setPixel((int) Math.Round(x), (int) Math.Round(y), color);
    }

    public string toString() {
        return " ["+x+", "+y+"]";
    }
}
```

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 = x0y1 – x1y0, then we get A= y0 – y1 and B = x1 – x0

Why would we just choose C? what are the advantages of our choice?

Easy. However, there is still a problem.

Numerical Precision of the C coefficient

- Computers represent floating-point number internally in a format similar to a scientific notation. The very worse thing that you can do with numbers represented in scientific notation is subtract numbers of similar magnitude. Here is what happens:

$$1.234 \times 10^3 - 1.233 \times 10^3 = 1.000 \times 10^0$$

- We lose most of the significant digits in our result

- 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

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

COMPUTER GRAPHICS

Maintaining Precision in C

- 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:

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

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

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

COMPUTER GRAPHICS

An Edge Equation Object

- We compute A, B, and C as described earlier
- Notice that we also compute a few flags to remember the signs of the A and B coefficients
- We also have added a flip() method, which will be used shortly
- We also have a method to evaluate the edge equation

```
public class EdgeEqn {
    public double A, B, C;
    public int flag;

    public EdgeEqn(Vertex2D v0, Vertex2D v1) {
        A = v0.y - v1.y;
        B = v1.x - v0.x;
        C = -0.5*(A*(v0.x + v1.x) + B*(v0.y + v1.y));
        flag = 0;
        if (A >= 0) flag += 8;
        if (B >= 0) flag += 1;
    }

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

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

COMPUTER GRAPHICS

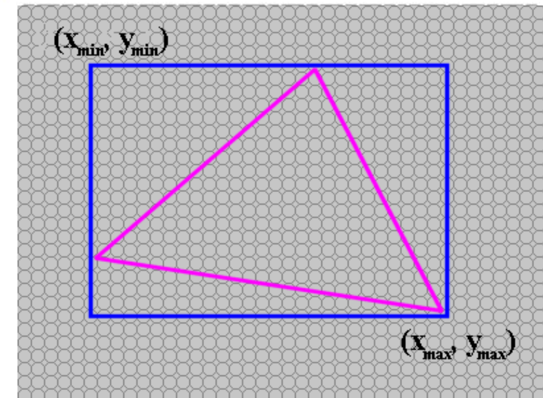
A Triangle Object

- Mostly obvious stuff here.
- A triangle has
  - 3 vertices
  - 3 edge equations
  - A color
- We use vertex colors to set the triangle color, for now

```
public class FlatTri : Drawable {
    public Vertex2D [] v;
    public Color color;
    protected EdgeEqn [] edge;

    public FlatTri() {}    // allows future extension

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

      /*
         ... Our policy is to assign a triangle
         the average of its vertex colors ...
      */
      byte r = (byte) ((2*(v0.color.r + v1.color.r + v2.color.r) + 3) / 6);
      byte g = (byte) ((2*(v0.color.g + v1.color.g + v2.color.g) + 3) / 6);
      byte b = (byte) ((2*(v0.color.b + v1.color.b + v2.color.b) + 3) / 6);
      color = new Color(r,g,b);
    }
```

To Draw() method

```
public override void Draw(Raster r) {
        if (!triangleSetup(r)) return;
        int x, y;
        double A0 = edge[0].A;   double A1 = edge[1].A;  double A2 = edge[2].A;
        double B0 = edge[0].B;   double B1 = edge[1].B;  double B2 = edge[2].B;
        double t0 = edge[0].evaluateAt(xMin, yMin);
        double t1 = edge[1].evaluateAt(xMin, yMin);
        double t2 = edge[2].evaluateAt(xMin, yMin);
        yMin *= r.width;   yMax *= r.width;
        /*    .... scan convert triangle ....     */
        for (y = yMin; y <= yMax; y += r.width) {
           double e0 = t0;   double e1 = t1;   double e2 = t2;
           bool beenInside = false;
           for (x = xMin; x <= xMax; x++) {
              if ((e0 >= 0) && (e1 >= 0) && (e2 >= 0)) {       // all 3 edges must be >= 0
                 int i = 4*(y+x);
                 r.data[i  ] = color.r;   r.data[i+1] = color.g;  r.data[i+2] = color.b;   r.data[i+3] = color.a;
                 beenInside = true;
              } else if (beenInside) break;
              e0 += A0;   e1 += A1;   e2 += A2;
           }
           t0 += B0;   t1 += B1;   t2 += B2;
        }
    }
```



Explain trick

Everything in our Draw() method is straightforward, with one exception

```
bool beenInside = false;
for (x = xMin; x <= xMax; x++) {
   if ((e0 >= 0) && (e1 >= 0) && (e2 >= 0)) {      // all 3 edges must be >= 0
      int i = 4*(y+x);
      r.data[i  ] = color.r;
      r.data[i+1] = color.g;
      r.data[i+2] = color.b;
      r.data[i+3] = color.a;
      beenInside = true;
   } else if (beenInside) break;
```

- Since triangles are convex, we can only be inside of 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.

Computer Graphics

Triangle Set-up

```
protected bool triangleSetup(Raster r) {
        if (edge == null) edge = new EdgeEqn[3];
        edge[0] = new EdgeEqn(v[0], v[1]);
        edge[1] = new EdgeEqn(v[1], v[2]);
        edge[2] = new EdgeEqn(v[2], v[0]);

        area = edge[0].C + edge[1].C + edge[2].C;
        if (area == 0) {
           return false;            // degenerate triangle
        } else if (area < 0) {
           edge[0].flip();   edge[1].flip();   edge[2].flip();   area = -area;
        }

        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);
        xMin = (xMin < 0) ? 0 : xMin;              xMax = (xMax >= r.width) ? r.width - 1 : xMax;
        yMin = (yMin < 0) ? 0 : yMin;              yMax = (yMax >= r.height) ? r.height - 1 : yMax;
        return true;
    }
```

Some more tricks

- In triangleSetup(), we did two critical things. We orient the edge equation, and we compute the bounding box.
- From analytic geometry we know the area of the triangle

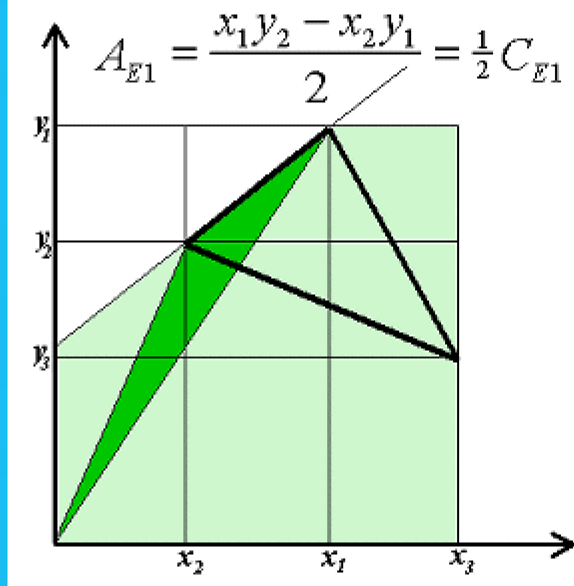
$\{(x_1, y_1), (x_2, y_2), (x_3, y_3)\}$ is:

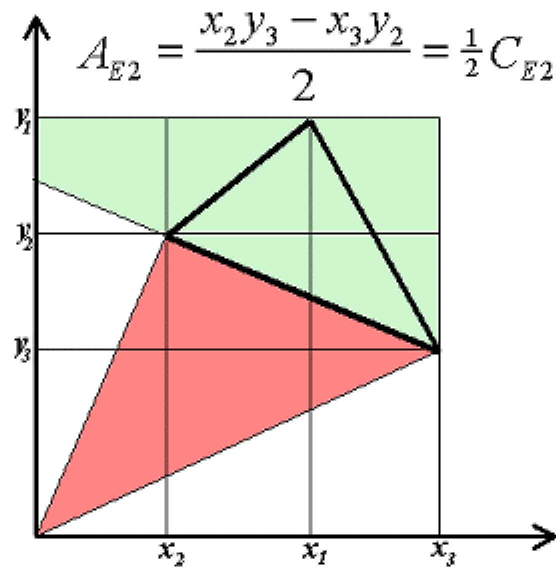
$$Area = \frac{1}{2}\det\begin{bmatrix} x_1 & x_2 & x_3 \\ y_1 & y_2 & y_3 \\ 1 & 1 & 1 \end{bmatrix}$$

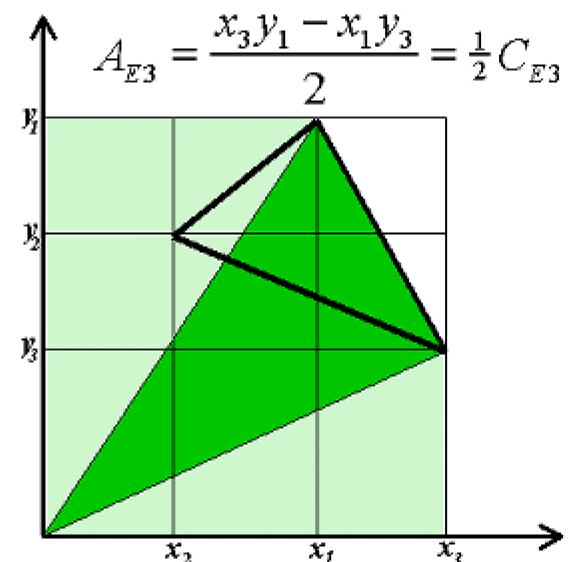
- The area is positive if the vertices are counterclockwise and negative if clockwise.
- An aside: In terms of our discriminator, what does a positive C imply?

COMPUTER GRAPHICS UD

Why a positive area implies a positive interior







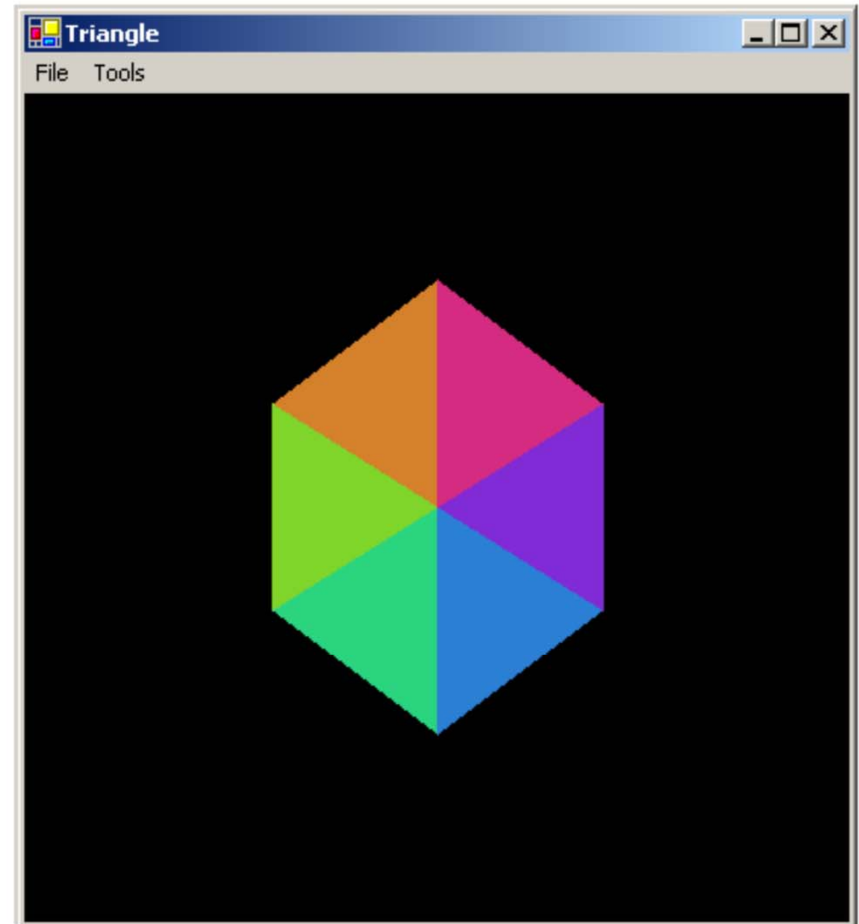
1. The area of each sub-triangle gives the edge equation's sign at the origin
2. Assume a positive area, thus the sum of the sub-triangle areas are positive
3. Each point within the triangle falls within exactly one sub-triangle, thus, subtriangles with negative areas will lie outside of the triangle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

COMPUTER GRAPHICS

Demonstration

- The OpenGL code is simple

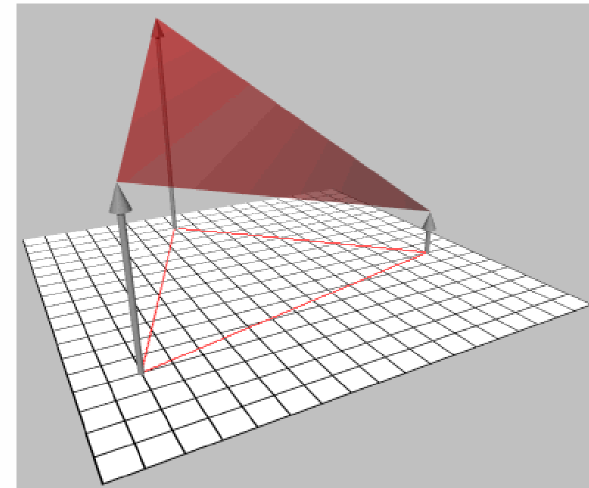
```
public override void Init( ) {
   if ((panelSize.Width != width) || (panelSize.Height != height)) {
      width = panelSize.Width;
      height = panelSize.Height;
      data = new PUInt8(4*width*height);
   }
   glViewport(0, 0, width, height);
   glClearColor(0.0f, 0.0f, 0.0f, 1.0f);
   glMatrixMode(GL_PROJECTION);
   glLoadIdentity();
   gluOrtho2D(0, width, 0, height);
   glMatrixMode(GL_MODELVIEW);
   glLoadIdentity();
}

public override void Draw() {
   clearRaster(0,0,0,255);
   foreach (Drawable t in triList) {
      t.Draw(this);
   }
   glRasterPos2i(0,0);
   glDrawPixels(width,height,GL_RGBA,GL_UNSIGNED_BYTE,data);
   glFlush();
}
```



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. We might also want to 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:

$$Z = Ax + By + C$$

COMPUTER GRAPHICS

A Familiar Equation

- It has the same form as our edge equation. 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

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

- The desired plane equation is just a linear combination of our three edge equations! Obvious?
- So 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.

A Smooth Triangle

- Most are the same
- A triangle has
  - 3 vertices
  - 3 edge equations
- We interpolate vertex color
  - Unless they are all the same

```
public class SmoothTri : FlatTri {
    bool 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.color == v1.color) && (v0.color == v2.color);
        if (isFlat) color = v0.color;

        /*
            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 fasterFlatTri methods that we inherited. The second is a scale factor that we'll discuss next.

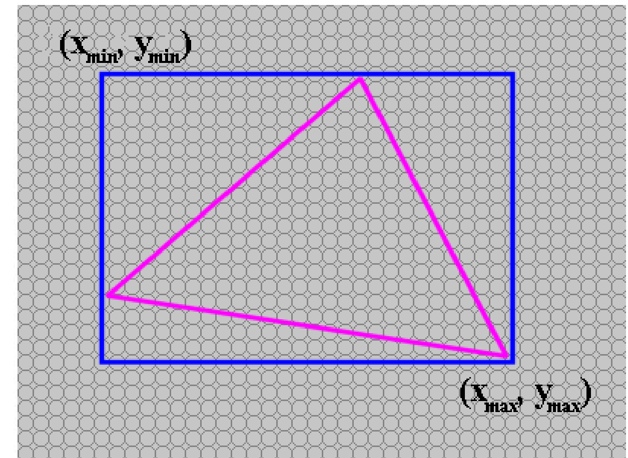
Next we add a new method to compute the plane equations of our parameters. ThePlaneEqn() method performs the required matrix multiply and avoids computing the inverse of the triangle area more than once.

```
public double [] PlaneEqn(int p0, int p1, int p2) {
    double Ap, Bp, Cp;
    double [] eqn = new double[3];
    if (scale <= 0) {
        scale = 1 / ((double) area);
    }
    double sp0 = scale * p0;
    double sp1 = scale * p1;
    double sp2 = scale * p2;
    Ap = edge[0].A*sp2 + edge[1].A*sp0 + edge[2].A*sp1;
    Bp = edge[0].B*sp2 + edge[1].B*sp0 + edge[2].B*sp1;
    Cp = 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;
    return eqn;
}
```

COMPUTER GRAPHICS

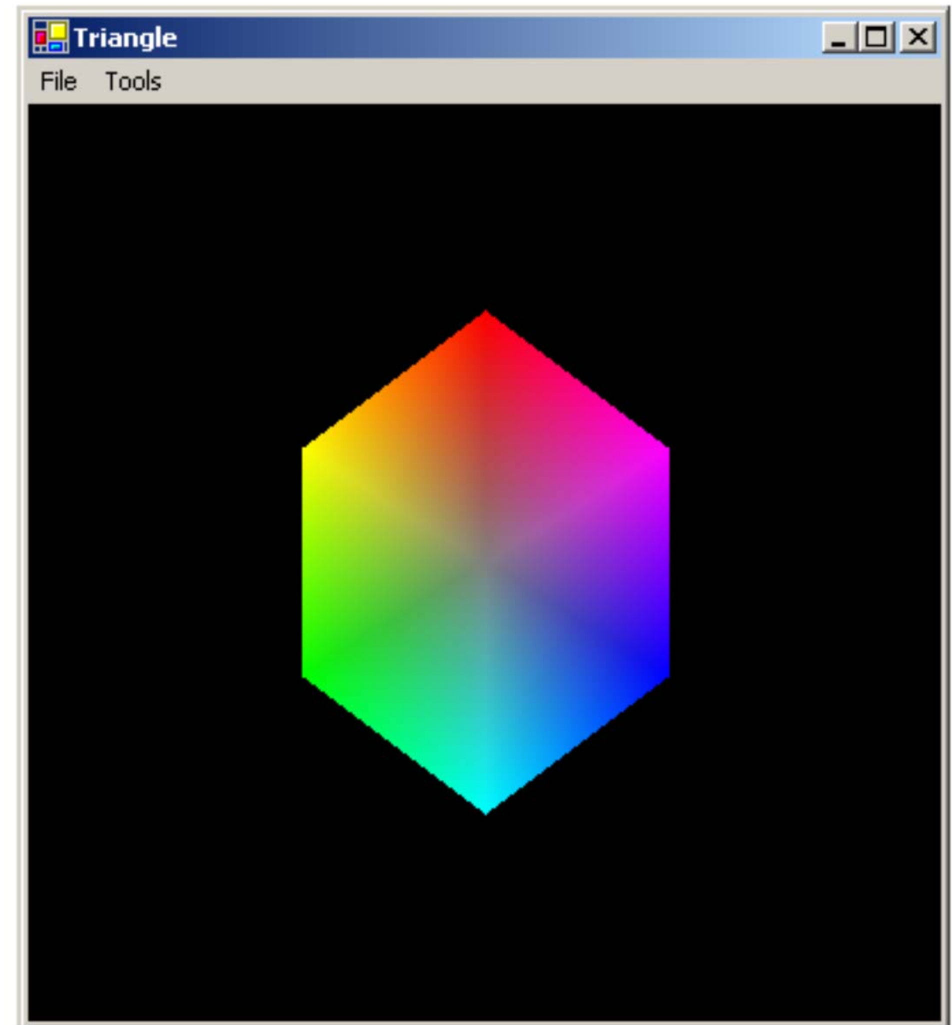
Modified Draw() Method

```
public override void Draw(Raster r) {
        if (!triangleSetup(r)) return;
        int x, y;
        double A0 = edge[0].A;   double A1 = edge[1].A;  double A2 = edge[2].A;
        double B0 = edge[0].B;   double B1 = edge[1].B;  double B2 = edge[2].B;
        double t0 = edge[0].evaluateAt(xMin, yMin);
        double t1 = edge[1].evaluateAt(xMin, yMin);
        double t2 = edge[2].evaluateAt(xMin, yMin);
        yMin *= r.width;   yMax *= r.width;
        /*    .... scan convert triangle ....     */
        for (y = yMin; y <= yMax; y += r.width) {
           double e0 = t0;   double e1 = t1;   double e2 = t2;
           bool beenInside = false;
           for (x = xMin; x <= xMax; x++) {
              if ((e0 >= 0) && (e1 >= 0) && (e2 >= 0)) {       // all 3 edges must be >= 0
                 int i = 4*(y+x);
                 r.data[i ] = color.r;    r.data[i+1] = color.g;  r.data[i+2] = color.b;    r.data[i+3] = color.a;
                 beenInside = true;
              } else if (beenInside) break;
              e0 += A0;    e1 += A1;    e2 += A2;
           }
           t0 += B0;    t1 += B1;   t2 += B2;
        }
    }
```



Demo: Smooth Triangles

- Smooth shading with per-vertex colors
- Can interpolate arbitrary parameters over the triangle
- At the heart of modern rendering H/W



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 primitives are visible, and they are uniformly distributed over the whole screen, that's still 5 primitives/pixel. Remember, that in order to draw a single triangle we must specify 3 vertices, determine three colors, and interpolate 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 Stanfords "Digital Michelangelo Project".

COMPUTER GRAPHICS UD

Point-Cloud Rendering

- A new class of rendering primitives have recently been introduced to address this problem.
- Key Attributes:
  - –Hierarchy
  - –Incremental Refinement
  - –Compact Representation (differential 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

- We dive into 3-D
- Read in and display 3D model

COMPUTER GRAPHICS UD