Scan Conversion

Projective Rendering Pipeline

Lines and Curves
**Lines and Curves**

**Parametric**

- **Line**
  \[ x(t) = x_0 + t(x_1 - x_0) \]
  \[ y(t) = y_0 + t(y_1 - y_0) \]
  \[ t \in [0,1] \]
  \[ P(t) = P_0 + t(P_1 - P_0) \]

- **Circle**
  \[ x(\theta) = r \cos(\theta) \]
  \[ y(\theta) = r \sin(\theta) \]
  \[ \theta \in [0,2\pi] \]

**Implicit**

- **Line**
  \[ F(x, y) = y - mx - b \]
  \[ F(x, y) = 0 \text{ is on line} \]
  \[ F(x, y) > 0 \text{ is above line} \]
  \[ F(x, y) < 0 \text{ is below line} \]

- **Circle**
  \[ F(x, y) = x^2 + y^2 - r^2 \]
  \[ F(x, y) = 0 \text{ is on circle} \]
  \[ F(x, y) > 0 \text{ is inside circle} \]
  \[ F(x, y) < 0 \text{ is outside circle} \]

**Polygons**

- **Interactive graphics uses Polygons**
  - Can represent any surface with arbitrary accuracy
    - Splines, mathematical functions, ...
  - Simple, regular rendering algorithms
    - Embed well in hardware

- **Even hippos are made of polygons!**
Polygons

Basic Types

- simple convex
- simple concave
- non-simple (self-intersection)

From Polygons to Triangles

- why? triangles are planar and convex
- simple convex polygons
  - break into triangles, trivial
  - `glBegin(GL_POLYGON) ... glEnd()
- concave or non-simple polygons
  - break into triangles, more effort
  - `gluNewTess(), gluTessCallback(), ...`

What is Scan Conversion? (a.k.a. Rasterization)

- screen is discrete
- one possible scan conversion
Scan Conversion

A General Algorithm
- intersect each scanline with all edges
- sort intersections in x
- calculate parity to determine in/out
- fill the 'in' pixels

Edge Walking
past graphics hardware
- exploit continuous L and R edges on trapezoid

for (y=yB; y<=yT; y++) {
    for (x=xL; x<=xR; x++) {
        setPixel(x,y);
        xL += DxL;
        xR += DxR;
    }
}
Edge Walking Triangles

- split triangles into two regions with continuous left and right edges

\[ \text{scanTrapezoid}(x_3, x_m, y_3, y_m, \frac{1}{m_{13}}, \frac{1}{m_{12}}) \]

\[ \text{scanTrapezoid}(x_2, x_2, y_2, y_3, \frac{1}{m_{23}}, \frac{1}{m_{12}}) \]

Issues

- many applications have small triangles
  - setup cost is non-trivial
- clipping triangles produces non-triangles

Modern Rasterization

*Define a triangle as follows:*

Using Edge Equations
Computing Edge Equations

**Computing A,B,C from (x₁, y₁), (x₂, y₂)**

\[Ax_1 + By_1 + C = 0\]
\[Ax_2 + By_2 + C = 0\]
\[Ax_3 + By_3 + C = 1\]

- two equations, three unknowns
- solve for A, B in terms of C

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

\[\begin{bmatrix}A \\ B\end{bmatrix} = \frac{-C}{x_0y_1 - x_1y_0} \begin{bmatrix}y_1 - y_0 \\ x_1 - x_0\end{bmatrix}\]

- choose \(C = x_0y_1 - x_1y_0\) for convenience
- Then \(A = y_0 - y_1\) and \(B = x_1 - x_0\)

**Edge Equations**

- So...we can find edge equation from two verts.
- Given \(P_0, P_1, P_2\), what are our three edges?
  
  *How do we make sure the half-spaces defined by the edge equations all share the same sign on the interior of the triangle?*
  
  - A: Be consistent (Ex: \([P_0, P_1, P_2, P_0]\))
    
    *How do we make sure that sign is positive?*
  - A: Test, and flip if needed (\(A = -A, B = -B, C = -C\))

**Edge Equations: Code**

**Basic structure of code:**

- Setup: compute edge equations, bounding box
- (Outer loop) For each scanline in bounding box...
- (Inner loop) ...check each pixel on scanline, evaluating edge equations and drawing the pixel if all three are positive
**Edge Equations: Code**

```c
findBoundingBox(&xmin, &xmax, &ymin, &ymax);
setupEdges (&a0,&b0,&c0,&a1,&b1,&c1,&a2,&b2,&c2);

for (int y = yMin; y <= yMax; y++) {
    for (int x = xMin; x <= xMax; x++) {
        float e0 = a0*x + b0*y + c0;
        float e1 = a1*x + b1*y + c1;
        float e2 = a2*x + b2*y + c2;
        if (e0 > 0 && e1 > 0 && e2 > 0)
            Image[x][y] = TriangleColor;
    }
}
```

// more efficient inner loop

```c
// more efficient inner loop
for (int y = yMin; y <= yMax; y++) {
    float e0 = a0*xMin + b0*y + c0;
    float e1 = a1*xMin + b1*y + c1;
    float e2 = a2*xMin + b2*y + c2;
    for (int x = xMin; x <= xMax; x++) {
        if (e0 > 0 && e1 > 0 && e2 > 0)
            Image[x][y] = TriangleColor;
            e0 += a0;   e1+= a1;    e2 += a2;
    }
}
```

**Triangle Rasterization Issues**

Exactly which pixels should be lit?

A: Those pixels inside the triangle edges

What about pixels exactly on the edge?

- Draw them: order of triangles matters (it shouldn’t)
- Don’t draw them: gaps possible between triangles

We need a consistent (if arbitrary) rule

- Example: draw pixels on left or top edge, but not on right or bottom edge
Triangle Rasterization Issues

Moving Slivers

Interpolation During Scan Conversion

- interpolate between vertices: (demo)
  - \( z \)
  - \( r, g, b \) colour components
  - \( u, v \) texture coordinates
  - \( N_x, N_y, N_z \) surface normals
- three equivalent ways of viewing this (for triangles)
  1. bilinear interpolation
  2. plane equation
  3. barycentric coordinates
1. Bilinear Interpolation

- Interpolate quantity along LH and RH edges, as a function of y
  - Then interpolate quantity as a function of x

2. Plane Equation

- \( \mathbf{v} = A\mathbf{x} + B\mathbf{y} + C \)

3. Barycentric Coordinates

- Weighted combination of vertices

Building Plane Equations:

\[
A_x + B_y + C_z + D = 0
\]

1. Compute \( \mathbf{N} \) by computing a normal:
   - \( \mathbf{N} = (\mathbf{P}_3 - \mathbf{P}_2) \times (\mathbf{P}_2 - \mathbf{P}_1) \)

2. Compute \( D = -\mathbf{N} \cdot \mathbf{P} \) for any point \( \mathbf{P} \) on the triangle:
   - \( D = -\mathbf{N} \cdot \mathbf{P} \)

"Convex combination of points"
Barycentric Coordinates

• once computed, use to interpolate any # of parameters from their vertex values

\[ z = \alpha \cdot z_1 + \beta \cdot z_2 + \gamma \cdot z_3 \]
\[ r = \alpha \cdot r_1 + \beta \cdot r_2 + \gamma \cdot r_3 \]
\[ g = \alpha \cdot g_1 + \beta \cdot g_2 + \gamma \cdot g_3 \]

etc.