Scan Conversion

Projective Rendering Pipeline

OCS - object coordinate system
WCS - world coordinate system
VCS - viewing coordinate system
CCS - clipping coordinate system
NDCS - normalized device coordinate system
DCS - device coordinate system

Lines and Curves

Explicit
- line
  \[ y = mx + b \]
- circle
  \[ y = \pm \sqrt{r^2 - x^2} \]
Lines and Curves

Parametric

**line**

\[ x(t) = x_0 + t(x_1 - x_0) \]
\[ y(t) = y_0 + t(y_1 - y_0) \]
\[ t \in [0,1] \]
\[ P(t) = P_0 + t(P_1 - P_0) \]
\[ P(t) = (1-t)P_0 + tP_1 \]

**circle**

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

Implicit

**line**

\[ F(x, y) = (x-x_0)dy - (y-y_0)dx \]
\[ F(x, y) = 0 \quad (x,y) \text{ is on line} \]
\[ F(x, y) > 0 \quad (x,y) \text{ is below line} \]
\[ F(x, y) < 0 \quad (x,y) \text{ is above line} \]

**circle**

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

Plane:

\[ F(x, y, z) = Ax + By + Cz + D = 0 \]
\[ \mathbf{N} \cdot \mathbf{P} + D = 0 \]
\[ \mathbf{N} \text{ is normal to the plane} \]

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

\[
\text{scanTrapezoid}(x_L, x_R, y_B, y_T, \Delta x_L, \Delta x_R)
\]

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

\[
\text{scanTrapezoid}(x_3, x_{m}, y_3, y_{m}, \frac{1}{m_3}, \frac{1}{m_2})
\]

\[
\text{scanTrapezoid}(x_2, x_2, y_2, y_3, \frac{1}{m_3}, \frac{1}{m_2})
\]

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

- $Ax_1 + By_1 + C = 0$
- $Ax_2 + By_2 + C = 0$

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

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

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

- choose $C = x_0 y_1 - x_1 y_0$ for convenience
- Then $A = y_0 - y_1$ and $B = x_0 - x_1$

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: $[P_0, P_1], [P_1, P_2], [P_2, P_0]$)

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

Shared Edge Ordering

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

[ remainder of lecture done on board -- get this from a friend if you could not make this class ]
1. Bilinear Interpolation

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

\[ P(x,y) = \frac{y}{h} \left( \frac{v_1}{v_2} \right) + \frac{y}{h} \left( \frac{v_3}{v_2} \right) \]

2. Plane Equation

- \( \mathbf{v} = Ax + By + C \)

\[ A(x-x_1) + B(y-y_1) + C(z-z_1) = 0 \]

3. Barycentric Coordinates

- Weighted combination of vertices

\[ P = \alpha \cdot P_1 + \beta \cdot P_2 + \gamma \cdot P_3 \]

- Convex combination of points

\[ \alpha + \beta + \gamma = 1 \]

\[ 0 \leq \alpha, \beta, \gamma \leq 1 \]
Barycentric Coordinates

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

\[ V = \alpha V_1 + \beta V_2 + \gamma V_3 \]

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

etc.

Computing Barycentric Coords

\[ P = \alpha P_1 + \beta P_2 + \gamma P_3 \]

Begin with implicit line \( g \) for \( P_1P_3 \):

\[ F(x, y) = Ax + By + C \]

\[ F(x_0, y_0) > 0 \]

\[ F(x_0, y_0) = k \]

Rescale so that

\[ \alpha(P) = \frac{F'(x_1, y_1)}{k} = 1 \]

Summary:

\[ \alpha = \frac{Ax + By + C}{k} \]

where

\[ k = Ax_1 + By_1 + C \]
\[ A = y_2 - y_3 \]
\[ B = x_2 - x_3 \]
\[ C = x_2 y_3 - x_3 y_2 \]

Similar equations can be computed for \( \beta \) and \( \gamma \).

Can also use \( Y = 1 - \alpha - \beta \).