Scan Conversion

Wolfgang Heidrich

Course News

Assignment 2
- Due March 2

Homework 3
- Discussed in labs next week

Reading
- Chapter 3

The Rendering Pipeline

Scan Conversion - Rasterization

Convert continuous rendering primitives into discrete fragments/pixels
- Lines
  - Midpoint/Bresenham
- Triangles
  - Flood fill
  - Scanline
  - Implicit formulation
- Interpolation

Scan Conversion - Lines
Scan Conversion of Lines - Digital Differential Analyzer

First Attempt:
- dda(float xs, ys, xe, ye)
  // assume xs < xe, and slope m between 0 and 1
  float m = (ye-ys)/(xe-xe);
  for(int x = round(xs); x <= xe; x++)
  {
    drawPixel(x, round(y));
    y = y + m;
  }

Scan Conversion of Lines - Midpoint Algorithm

Moving horizontally along x direction
- Draw at current y value, or move up vertically to y+1?
  - Check if midpoint between two possible pixels above or below line

Candidates
- Top pixel (x+1, y+1)
- Bottom pixel (x+1, y)

Midpoint: (x+1, y+0.5)
- Check if midpoint above or below line
  - Below top pixel
  - Above: bottom pixel

Key idea behind Bresenham Alg.

Scan Conversion of Lines

Idea: decision variable
- ddb(float xs, ys, xe, ye)
  float d = 0.0;
  float m = (ys-ys)/(xe-xe);
  int y = round(ys);
  for(int x = round(xs); x <= xe; x++)
  {
    drawPixel(x, y);
    d = d + m;
    if(d >= 0.5) { d = d-1.0, y++; }
  }

Scan Conversion of Lines

Bresenham Algorithm (*63)
- Use decision variable to generate purely integer algorithm
- Explicit line equation:
  \[ y = \frac{(y_f-y_i)(x-x_i) + y_i}{x_f-x_i} \]
- Implicit version:
  \[ L(x, y) = \frac{(y_f-y_i)(x-x_i) - (y-y_i)}{(x_f-x_i)} = 0 \]
- In particular for specific x, y, we have
  - \( L(x, y) > 0 \) if \( (x, y) \) below the line, and
  - \( L(x, y) < 0 \) if \( (x, y) \) above the line

Scan Conversion of Lines
Scan Conversion of Lines

Bresenham Algorithm

- Decision variable: after drawing point (x,y) decide whether to draw
  - (x+1,y): case E (for 'east')
  - (x+1,y+1): case NE (for 'north-east')
- Check whether \((x+1,y+1/2)\) is above or below line
  \[ d = L(x+1,y+1/2) \]
- Point above line if and only if \(d < 0\)

Scan Conversion of Lines

Bresenham Algorithm

- Problem: how to update \(d\)?
- Case E (point above line, \(d < 0\))
  - \(x = x + 1;\)
  - \(d = L(x+1,y+1/2) + d = (y+y')(x_e-x_e)\)
- Case NE (point below line, \(d > 0\))
  - \(x = x + 1; y = y + 1;\)
  - \(d = L(x+1,y+1/2) - d = (y+y')(x_e-x_e) - 1\)
- Initialization:
  - \(d = L(x+1,y+1/2) = (y+y')(x_e-x_e) - 1/2\)

Scan Conversion of Lines

Bresenham Algorithm

```c
Bresenham( int xx, ys, xe, ye ) {
    int y = ys;
    incE = 2(ye - ys);
    incNE = 2((ye - ys) - (xe-xx));
    for int x = xx; x <= xe; x++ ) {
        drawPixel( x, y );
        if( d < 0 ) d += incE;
        else { d += incNE; y++; }
    }
}
```

Scan Conversion of Lines

**Discussion**

- Bresenham sets same pixels as DDA
- Intensity of line varies with its angle!

Scan Conversion of Lines

**Discussion**

- Bresenham
  - Good for hardware implementations (integer!)
- DDA
  - May be faster for software (depends on system!)
  - Floating point ops higher parallelized (pipelined)
    - E.g. RISC CPUs from MIPS, SUN
  - No if statements in inner loop
    - More efficient use of processor pipelining
Scan Conversion of Polygons

A General Algorithm
- Intersect each scanline with all edges
- Sort intersections in x
- Calculate parity to determine in/out
- Fill the ‘in’ pixels

Scan Conversion of Polygons

One possible scan conversion

Scan Conversion of Polygons

- Works for arbitrary polygons
- Efficiency improvement:
  - Exploit row-to-row coherence using “edge table”

Edge Walking

Past graphics hardware
- Exploit continuous L and R edges on trapezoid

Edge Walking

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

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
  \[ \text{scanTrapezoid}(X_0, Y_0, X_1, Y_1, \frac{X_2 - X_0}{X_1 - X_0}, \frac{Y_2 - Y_0}{Y_1 - Y_0}) \]
  \[ \text{scanTrapezoid}(X_2, Y_2, X_3, Y_3, \frac{X_0 - X_2}{X_1 - X_0}, \frac{Y_0 - Y_2}{Y_1 - Y_0}) \]

Issues
- Many applications have small triangles
  - Setup cost is non-trivial
- Clipping triangles produces non-triangles
  - This can be avoided through re-triangulation, as discussed

Modern Rasterization: Edge Equations

Define a triangle as follows:

Using Edge Equations

Usage:
- Go over each pixel in bounding rectangle
- Check if pixel is inside/outside of triangle
  - Using sign of edge equations

Computing Edge Equations

Implicit equation of a triangle edge:
\[ L(x, y) = \frac{(y_1 - y_2)}{(x_1 - x_2)}(x - x_1) - (y - y_1) = 0 \]

(see Bresenham algorithm)
- \( L(x, y) \) positive on one side of edge, negative on the other

Question:
- What happens for vertical lines?

Edge Equations

Multiply with denominator
\[ L(x, y) = (y - y_1)(x - x_1) - (y - y_2)(x - x_2) = 0 \]

- Avoids singularity
- Works with vertical lines

What about the sign?
- Which side is in, which is out?
Edge Equations

**Determining the sign**
- Which side is “in” and which is “out” depends on order of start/end vertices...
- Convention: specify vertices in counter-clockwise order

\[ L(x,y) = (y_2 - y_3)(x - x_3) + (y_3 - y_1)(x - x_1) = 0 \]

**Counter-Clockwise Triangles**
- The equation \( L(x,y) \) as specified above is negative inside, positive outside
  - **Flip sign:**
    \[ L(x,y) = -(y_2 - y_3)(x - x_3) + (y_3 - y_1)(x - x_1) = 0 \]

**Clockwise triangles**
- Use original formula
  \[ L(x,y) = (y_2 - y_3)(x - x_3) + (y_3 - y_1)(x - x_1) = 0 \]

Discussion of Polygon Scan Conversion Algorithms

**On old hardware:**
- Use first scan-connection algorithm
  - Scan-convert edges, then fill in scanlines
  - Compute interpolated values by interpolating along edges, then scanlines
- Requires clipping of polygons against viewing volume
- Faster if you have a few, large polygons
- Possibly faster in software

**Modern GPUs:**
- Use edge equations
  - And plane equations for attribute interpolation
- No clipping of primitives required
- Faster with many small triangles

**Additional advantage:**
- Can control the order in which pixels are processed
- Allows for more memory-coherent traversal orders
  - E.g. tiles or space-filling curve rather than scanlines

Triangle Rasterization Issues (Independent of Algorithm)

**Exactly which pixels should be lit?**
- A: Those pixels inside the triangle edge (of course)

**But 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

Shared Edge Ordering
Triangle Rasterization Issues

These are ALIASING Problems
- Problems associated with representing continuous functions (triangles) with finite resolution (pixels)
- More on this problem when we talk about sampling.

Shading

Interpolation During Scan Conversion

Need to propagate vertex attributes to pixels
- Interpolate between vertices:
  - $z$ (depth)
  - $r, g, b$ color components
  - $N_x, N_y, N_z$ surface normals
  - $u, v$ texture coordinates (talk about these later)
- Three equivalent ways of viewing this (for triangles)
  1. Bilinear interpolation
  2. Barycentric coordinates
  3. Plane Equation

1. Bilinear Interpolation

We’ve seen this before:
- Interpolate quantity along LH and RH edges, as a function of $y$
  - Then interpolate quantity as a function of $x$
2. Barycentric Coordinates

This too:
- Barycentric Coordinates: weighted combination of vertices
  \[ P = \alpha \cdot P_1 + \beta \cdot P_2 + \gamma \cdot P_3 \]
  \[ \alpha + \beta + \gamma = 1 \]
  \[ 0 \leq \alpha, \beta, \gamma \leq 1 \]

\[ (0,0,1) \quad \beta = 0 \]
\[ (0,1,0) \quad \beta = 0.5 \]
\[ (1,0,0) \quad \beta = 1 \]

3. Plane Equation

Observation: Quantities vary linearly across image plane
- E.g.: \( r = Ax + By + C \)
  - \( r \) = red channel of the color
  - Same for \( g, b, N_x, N_y, N_z, z \)...
- From info at vertices we know:
  \[ r_1 = Ax_1 + By_1 + C \]
  \[ r_2 = Ax_2 + By_2 + C \]
  \[ r_3 = Ax_3 + By_3 + C \]
- Solve for \( A, B, C \)
- One-time set-up cost per triangle and interpolated

Flat Shading
- Simplest approach calculates illumination at a single point for each polygon
- Obviously inaccurate for smooth surfaces

Flat Shading Approximations

If an object really is faceted, is this accurate?
- No!
  - For point sources, the direction to light varies across the facet
  - For specular reflectance, direction to eye varies across the facet

Improving Flat Shading

What if evaluate Phong lighting model at each pixel of the polygon?
- Better, but result still clearly faceted
- For smoother-looking surfaces we introduce vertex normals at each vertex
  - Usually different from facet normal
  - Used only for shading
  - Think of as a better approximation of the real surface that the polygons approximate
**Vertex Normals**

**Vertex normals may be**
- Provided with the model
- Computed from first principles
- Approximated by averaging the normals of the facets that share the vertex

**Barycentric Coordinates**

- Convex combination of 3 points
  \[ x = \alpha x_1 + \beta x_2 + \gamma x_3 \]
  \[ \text{with } \alpha + \beta + \gamma = 1, \ 0 \leq \alpha, \beta, \gamma \leq 1 \]
- \( \alpha, \beta, \gamma \) are called barycentric coordinates

**Gouraud Shading**

**Mach bands**
- Eye enhances discontinuity in first derivative
- Very disturbing, especially for highlights
Phong Shading

- Linearly interpolating surface normal across the facet, applying Phong lighting model at every pixel
- Same input as Gouraud shading
- Pros: much smoother results
- Cons: considerably more expensive

Not the same as Phong lighting
- Common confusion
- Phong lighting: empirical model to calculate illumination at a point on a surface

Phong Shading Difficulties

- Computationally expensive
  - Per-pixel vector normalization and lighting computation
  - Floating point operations required
- Lighting after perspective projection
  - Messes up the angles between vectors
  - Have to keep eye-space vectors around
- No direct support in hardware
  - But can be simulated with texture mapping

Shading Artifacts: Silhouettes

Polygonal silhouettes remain

Gouraud Phong

Coming Up:

Friday
- Scan conversion / shading

Next week
- Clipping, hidden surface removal