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 - Lines
Scan Conversion - Lines

First Attempt:
- Line (s,e) given in device coordinates
- Create the thinnest line that connects start point and end point without gap

Assumptions for now:
- Start point to the left of end point: xs < xe
- Slope of the line between 0 and 1 (i.e. elevation between 0 and 45 degrees):
  \[ 0 \leq \frac{ye - ys}{xe - xs} \leq 1 \]

Scan Conversion of Lines - Digital Differential Analyzer

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

**DDA:**

![Diagram of a line being scanned using DDA algorithm]

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 pixel centers above or below line

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

Midpoint: (x+1, y+.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

dda( float xs, ys, xe, ye ) {
    float d = 0.0;
    float m = (ye-ys)/(xe-xs);
    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:
  \[ y = \frac{(y_e - y_s)}{(x_e - x_s)}(x - x_s) + y_s \]
- Implicit version:
  \[ L(x, y) = \frac{(y_e - y_s)}{(x_e - x_s)}(x - x_s) - (y - y_s) = 0 \]
- 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
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
  \[ d = L(x + 1, y + \frac{1}{2}) \]
- 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, \(d <= 0\))
  - \(x = x + 1;\)
  - \(d = L(x + 2, y + 1/2) = d + \frac{(y_e - y_2)}{(x_e - x_2)}\)
- Case NE (point below line, \(d > 0\))
  - \(x = x + 1; y = y + 1;\)
  - \(d = L(x + 2, y + 3/2) = d + \frac{(y_e - y_2)}{(x_e - x_2)} - 1\)
- Initialization:
  - \(d = L(x_e + 1, y_e + 1/2) = \frac{(y_e - y_2)}{(x_e - x_2)} - 1/2\)
Scan Conversion of Lines

**Bresenham Algorithm**

- This is still floating point
- But: only sign of \( d \) matters
- Thus: can multiply everything by \( 2(x_e-x_s) \)

---

```c
Bresenham( int xs, ys, xe, ye ) {
    int y = ys;
    incrE = 2(ys - ye);
    incrNE = 2((ye - ys) - (xe-xs));
    for( int x = xs ; x <= xe ; x++ ) {
        drawPixel( x, y );
        if( d <= 0 ) d += incrE;
        else { d += incrNE; 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

One possible scan conversion
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

- 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

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

for \((y = y_B; y <= y_T; y++)\) {
for \((x = x_L; x <= x_R; x++)\)
    \[ \text{setPixel}(x, y); \]
    \[ x_L += \text{DxL}; \]
    \[ x_R += \text{DxR}; \]
}
Edge Walking Triangles

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

\[
\text{scanTrapezoid}(x_3, x_m, y_3, y_p, \frac{1}{m_1}, \frac{1}{m_2})
\]
\[
\text{scanTrapezoid}(x_p, x_2, y_2, y_3, \frac{1}{m_2}, \frac{1}{m_3})
\]

Edge walking

Edge Walking Triangles

**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

\[(x_{\text{min}}, y_{\text{min}}), (x_{\text{max}}, y_{\text{max}})\]
Computing Edge Equations

*Implicit equation of a triangle edge:*

\[ L(x, y) = \frac{(y_e - y_s)(x - x_e) - (y - y_e)(x_e - x_s)}{(x_e - x_s)} = 0 \]

(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*

\[ L(x, y) = (y_e - y_s)(x - x_e) - (y - y_e)(x_e - x_s) = 0 \]

- 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

\[ L(x, y) = \frac{(y - y_s)(x - x_s) - (y - y_e)(x - x_e)}{y_e - y_s} \]

Edge Equations

Counter-Clockwise Triangles

- The equation \( L(x, y) \) as specified above is negative inside, positive outside
  - Flip sign:
    \[ L(x, y) = -(y_e - y_s)(x - x_e) + (y - y_s)(x_e - x_s) = 0 \]

Clockwise triangles

- Use original formula
  \[ L(x, y) = (y_e - y_s)(x - x_s) - (y - y_s)(x_e - x_s) = 0 \]
Discussion of Polygon Scan Conversion Algorithms

**On old hardware:**
- Use first scan-conversion 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

---

Discussion of Polygon Scan Conversion Algorithms

**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
Triangle Rasterization Issues

Sliver

Triangle Rasterization Issues

Moving Slivers
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

Wolfgang Heidrich
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
  \[ P = \alpha \cdot P_1 + \beta \cdot P_2 + \gamma \cdot P_3 \]
  \[ \alpha + \beta + \gamma = 1 \]
  \[ 0 \leq \alpha, \beta, \gamma \leq 1 \]

3. Plane Equation

**Observation:** Quantities vary linearly across image plane
- E.g.: \( r = Ax + By + C \)
  - \( r \) = red channel of the color
  - Same for \( g, b, Nx, Ny, Nz, z \)...
- 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 quantity
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?*
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

**Gouraud Shading**

*Most common approach, and what OpenGL does*
- Perform Phong lighting at the vertices
- Linearly interpolate the resulting colors over faces
  - Along edges
  - Along scanlines

*Same as Barycentric Coordinates!

- interior: mix of $c_1$, $c_2$, $c_3$
- edge: mix of $c_1$, $c_2$
**Barycentric Coordinates**

- Convex combination of 3 points

\[ x = \alpha \cdot x_1 + \beta \cdot x_2 + \gamma \cdot x_3 \]

with \( \alpha + \beta + \gamma = 1 \), \( 0 \leq \alpha, \beta, \gamma \leq 1 \)

- \( \alpha, \beta, \) and \( \gamma \) are called *barycentric coordinates*

**One way to compute them:**

\[ x = \alpha x_1 + \beta x_2 + \gamma x_3 \]

with

\[ \alpha = A_1 / A \]
\[ \beta = A_2 / A \]
\[ \gamma = A_3 / A \]
Gouraud Shading Artifacts

often appears dull, chalky
lacks accurate specular component

- if included, will be averaged over entire polygon

Gouraud Shading Artifacts

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
- Pro: much smoother results
- Con: 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**

*Linearly interpolate the vertex normals*

- Compute lighting equations at each pixel
- Can use specular component

\[
I_{total} = k_a I_{ambient} + \sum_{i=1}^{\#lights} I_i \left( k_d \left( n \cdot I_i \right) + k_s \left( v \cdot r_i \right)^{n_{shiny}} \right)
\]

remember: normals used in diffuse and specular terms

discontinuity in normal’s rate of change harder to detect
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