**Scan Conversion**

**CPSC 314**

---

**Line Clipping**

- Window clipping:
  - $x = x_{\text{min}}$ to $x = x_{\text{max}}$
  - $y = y_{\text{min}}$ to $y = y_{\text{max}}$

---

**The Rendering Pipeline**

1. **Geometry Processing**
   - Geometry Database
   - Model/View Transform.
   - Lighting
   - Perspective Transform.
   - Clipping

2. **Scan Conversion**
3. **Texturing**
4. **Depth Test**
5. **Blending**
6. **Frame-buffer**

---

**Line Clipping**

### Outcodes (Cohen, Sutherland '74)

- 4 flags encoding position of a point relative to top, bottom, left, and right boundary
- E.g.:
  - $OC(p_1) = 0010$
  - $OC(p_2) = 0000$
  - $OC(p_3) = 1001$

---

**Line Clipping**

### Triangle Clipping

**Line segment:**

- $(p_1, p_2)$

**Trivial cases:**

- $OC(p_1) = 0$ & $OC(p_2) = 0$
  - Both points inside window, thus line segment completely visible (trivial accept)
- $(OC(p_1) & OC(p_2)) = 0$
  - There is (at least) one boundary for which both points are outside (same flag set in both outcodes)
  - Thus line segment completely outside window (trivial reject)

---

**Line Clipping**

### $\alpha$-Clipping

- Handling of all the non-trivial cases
- Improvement of earlier algorithms (Cohen/Sutherland, Cyrus/Berk, Liang/Barsky)
- Define window-edge-coordinates of a point $p = (x, y)^T$

  - $WEC_x(p) = x - x_{\text{min}}$
  - $WEC_y(p) = y - y_{\text{min}}$
  - $WEC_1(p) = x_{\text{max}} - x$
  - $WEC_2(p) = y_{\text{max}} - y$

  **Negative if outside!**
Line Clipping

\( \alpha \)-Clipping: example for clipping \( p_1 \)

- Start configuration
- After clipping to left
- After clipping to top

Polygon Clipping

Example

Scan Conversion - Rasterization

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

The Rendering Pipeline

Scan Conversion - Lines

Scan Conversion - Lines
Scan Conversion - Lines

First Attempt:
- Line (s,0) 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: \( x_s < x_e \)
- Slope of the line between 0 and 1 (i.e. elevation between 0 and 45 degrees):
  \[
  0 \leq \frac{y_e - y_s}{x_e - x_s} \leq 1
  \]

Scan Conversion of Lines

DDA:

Scan Conversion of Lines

Idea: decision variable

\[
\text{ddt, float xs, ys, xe, ye) }
\]
float d= 0.0;
float m= (ye-ys)/(xe-xs);
int y= round( y/s );
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 - Digital Differential Analyzer

First Attempt:

\[
\text{ddt(float xs, ys, xe, ye) }
\]
// assume xs < xe, and slope m between 0 and 1
float m= (ye-ys)/(xe-xs);
float y= round( y/s );
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 pixel centers above or below line

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

Midpoint: \( x+1, y+5.5 \)

Check if midpoint above or below line
- Below: top pixel
- Above: bottom pixel

Key idea behind Bresenham Alg.

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
  \[
  L(x,y) > 0 \text{ if } (x,y) \text{ below the line, and}
  L(x,y) < 0 \text{ if } (x,y) \text{ above the line} 
  \]
**Scan Conversion of Lines**

**Bresenham Algorithm**
- Decision variable: after drawing point \((x, y)\) decide whether to draw
  - \((x+1, y)\): case E (for “east”)
  - \((x, y+1)\): 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**
- This is still floating point
- But: only sign of \(j\) matters
- Thus: can multiply everything by \(2(x - x_j)\)

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

---

**Scan Conversion of Lines**

**Bresenham Algorithm**
- Problem: how to update \(d\)?
- Case E (point above line, \(d < 0\))
  - \(x = x + 1\);
  - \(d = L(x + 2, y + 1/2) - d + (y - y_e)(x - x_e)\)
- Case NE (point below line, \(d > 0\))
  - \(x = x + 1; y = y + 1\);
  - \(d = L(x + 2, y + 3/2) - d + (y - y_e)(x - x_e) - 1\)
- Initialization:
  - \(d = L(x + 1, y + 1/2) = (y - y_e)(x - x_e) - 1/2\)

---

**Scan Conversion of Lines**

**Bresenham Algorithm**
\[
\text{Bresenham}( \text{int } x_s, y_s, x_e, y_e) \{ \\
\text{int } y = y_s; \\
\text{incrE} = 2(y_e - y_s); \\
\text{incrNE} = 2((y_e - y_s) - (x_e - x_s)); \\
\text{for } (\text{int } x = x_s; x <= x_e; x++) \{ \\
\text{drawPixel} (x, y); \\
\text{if } (d < 0) \text{ d} += \text{incrE}; \\
\text{else } \text{ d} += \text{incrNE}; y++; \} \\
\}
\]

**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

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

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

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

Edge Walking
- Works for arbitrary polygons
- Efficiency improvement:
  - Exploit row-to-row coherence using "edge table"
**Edge Walking Triangles**

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

\[
\text{scanTrapezoid}(x_3, x_2, y_3, y_2, m_1, m_2, m_3) = \frac{1}{2} (x_2 - x_3) (y_3 - y_2)
\]

**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:**

\[
L(x, y) = \frac{(x_2 - x_1)(y - y_1) - (x - x_1)(y_2 - y_1)}{(x_2 - x_1)} = 0
\]

- \((x_2 - x_1)\) positive on one side of edge, negative on the other

**Using Edge Equations**

- 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:**

\[
L(x, y) = \frac{(x_2 - x_1)(y - y_1) - (x - x_1)(y_2 - y_1)}{(x_2 - x_1)} = 0
\]

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

**Assumption:**
- Triangle vertices given in counter-clockwise order

**Then:**
- If \(x_1 < x_s\), then
  - Use \(-L(x, y)\) as edge equation
- Else
  - Use \(+L(x, y)\) as edge equation

**Question:**
- How do we know which side is in, and which side out?
  - And how do we make the \(L(x, y)\) positive for points inside?
Computing Edge Equations

**Inside/Outside depends on vertex order:**
- Implicit equation of a triangle interior:
  \[ L(x, y) > 0 \]
  
  with
  \[
  L(x, y) = \left\{ \begin{array}{ll}
  (y_2 - y_1)(x - x_1) + (y_1 - y_2)(x - x_2) & \text{if } x < x_1 \\
  (y_2 - y_1)(x - x_1) - (y_1 - y_2)(x - x_2) & \text{if } x > x_1
  \end{array} \right.
  
  \]

**What about vertical lines?**
- \( x = x_s \) → division by zero

Computing Edge Equations

**Summary:**
- Now we have only ONE equation
  \[ L(x, y) = -(y_2 - y_1)(x - x_1) + (y_1 - y_2)(x - x_2) \]
  
  • Works for both cases
  • Also works for vertical lines!

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
    - We’ll discuss a better way for these next lecture
- 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 \( g, b, N_x, N_y, N_z \)...
- From intersections we know:
  \[ r_1 = A x_1 + B y_1 + C \]
  \[ r_2 = A x_2 + B y_2 + C \]
  \[ r_3 = A x_3 + B y_3 + C \]
- Solve for \( A, B, C \)
- One-time set-up cost per triangle and interpolated quantity

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

Egde Equation Rasterization and Clipping

Note:
- Once we use edge equations, we no longer really have to clip the geometry against window boundary!
- Instead: clip bounding rectangle against window
  - Only evaluate edge equations for pixels inside the window!
  - Near/far clipping: when interpolating depth values, detect whether point is closer than near or farther than far
  - If so, don’t draw it

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 draws 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

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

Triangle Rasterization Issues

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...

Coming Up...

Thursday (Oct 11):
- Visibility

Tuesday (Oct 16):
- Quiz 1 solutions (Brad)

Thursday (Oct 18):
- Double Buffering, Picking, Alpha Blending (Brad)