Scan Conversion

**Line Clipping**

**Line segment:**
- \((p_1, p_2)\)

**Trivial cases:**
- \(\text{OC}(p_1) = 0 \) \& \& \(\text{OC}(p_2) = 0\)
  - Both points inside window, thus line segment completely visible (trivial accept)
- \((\text{OC}(p_1) \& \& \text{OC}(p_2)) = 0\)
  - 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**

**Outcodes (Cohen, Sutherland '74)**
- 4 flags encoding position of a point relative to top, bottom, left, and right boundary
- E.g.:
  - \(\text{OC}(p_1) = 0010\)
  - \(\text{OC}(p_2) = 0000\)
  - \(\text{OC}(p_3) = 1001\)

**Line Clipping**

**\(\alpha\)-Clipping**
- Handling of all the non-trivial cases
- Improvement of earlier algorithms (Cohen/Sutherland, Cyrus/Beck, Liang/Barsky)
- Define window-edge-coordinates of a point \(p = (x, y)^T\)
  - \(\text{WEC}_1(p) = x - x_{\text{min}}\)
  - \(\text{WEC}_2(p) = x_{\text{max}} - x\)
  - \(\text{WEC}_3(p) = y - y_{\text{min}}\)
  - \(\text{WEC}_4(p) = y_{\text{max}} - y\)
  - Negative if outside!
Line Clipping

\[ \alpha \text{-Clipping: example for clipping } p_1 \]

\[ \text{Start configuration} \quad \text{After clipping to left} \quad \text{After clipping to top} \]

Polygon Clipping

\[ \text{Example} \]

Hidden Surface Removal:
Binary Space Partitioning Trees

BSP Trees: Viewpoint A

BSP Trees: Viewpoint A

- decide independently at each tree vertex
- not just left or right child!
Hidden Surface Removal: 
Z-Buffer / Depth Buffer

Store \( r,g,b,z \) for each pixel

- typically 8+8+8+24 bits, can be more
- Typically 32 bits, but can be larger

```plaintext
for all i,j {
    Depth[i,j] = MAX_DEPTH
    Image[i,j] = BACKGROUND_COLOUR
}
for all polygons P {
    for all pixels in P {
        if (Z_pixel < Depth[i,j]) {
            Image[i,j] = C_pixel
            Depth[i,j] = Z_pixel
        }
    }
}
```

Depth Test Precision

- Low precision can lead to depth-fighting for far objects
  - Two different depths in eye space get mapped to same depth in framebuffer
  - Which object "wins" depends on drawing order and scan-conversion
- Gets worse for larger ratios \( f/n \)
  - Rule of thumb: \( f/n < 1000 \) for 24 bit depth buffer
- With 16 bits cannot discern millimeter differences in objects at 1 km distance

Z-Buffer Pros

- Simple!!!
- Easy to implement in hardware
  - Hardware support in all graphics cards today
- Polygons can be processed in arbitrary order
- Easily handles polygon interpenetration

Z-Buffer Cons

- Requires lots of memory
  - (e.g., 1280x1024x32 bits)
- Requires fast memory
  - Read-Modify-Write in inner loop
- Hard to simulate transparent polygons
  - We throw away color of polygons behind closest one
  - Works if polygons ordered back-to-front
  - Extra work throws away much of the speed advantage

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

Objective
- Convert continuous rendering primitives to discrete fragments/pixels

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: \( x_s < x_e \)
- 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 - 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:**

- Draw horizontally along x direction
- Check if midpoint between two possible pixel centers above or below line

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

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

**Key idea behind Bresenham**

### Scan Conversion of Lines

**Bresenham Algorithm (’63)**

- Use decision variable to generate purely integer algorithm
- Explicit line equation:
  
  \[
  y = \frac{(y_2 - y_1)}{(x_2 - x_1)} (x - x_1) + y_1
  \]
- Implicit version:
  
  \[
  L(x, y) = \frac{(y_2 - y_1)}{(x_2 - x_1)} (x - x_1) - (y - y_1) = 0
  \]
- 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**

- Problem: how to update \(d\)?
- Case E (point above line, \(d < 0\))
  
  \[
  \begin{align*}
  \text{x} &= x+1; \\
  \text{d} &= \text{L}(x+2, y+1/2) = d + (y_2 - y_1)(x_1 - x_2)
  \end{align*}
  \]
- Case NE (point below line, \(d > 0\))
  
  \[
  \begin{align*}
  \text{x} &= x+1; \\
  \text{y} &= y+1; \\
  \text{d} &= \text{L}(x+2, y+1/2) = d + (y_2 - y_1)(x_2 - x_1) - 1
  \end{align*}
  \]
- Initialization:
  
  \[
  \text{d} = \text{L}(x+1, y+1/2) = (y_2 - y_1)(x_1 - x_2) - 1/2
  \]
**Scan Conversion of Lines**

**Bresenham Algorithm**
- This is still floating point
- But: only sign of \( j \) matters
- Thus: can multiply everything by \( 2(x_j-x_i) \)

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

---

**Scan Conversion of Lines**

**Bresenham Algorithm**

```
Bresenham( int xs, ys, xe, ye ) {
  int y = ys;
  incrE= 2( ye - ys );
  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++; }
  }
}
```

**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

Edge Walking

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

Scan Conversion of Polygons

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

Edge Walking

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

Define a triangle as follows:

Using Edge Equations

Computing Edge Equations

Computing \( A, B, C \) from \((x_0, y_0), (x_2, y_2)\)

\[
\begin{align*}
Ax_1 + By_1 + C &= 0 \\
Ax_2 + By_2 + C &= 0
\end{align*}
\]

- Two equations, three unknowns
- Solve for \( A \) & \( B \) in terms of \( C \)

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
// more efficient inner loop
for (int y = ymin; y <= ymax; y++) {
    for (int x = xmin; x <= xmax; x++) {
        float a0 = a0*xMin + b0*y + c0;
        float a1 = a1*xMin + b1*y + c1;
        float a2 = a2*xMin + b2*y + c2;
        for (int x = xmin; x <= xmax; x++) {
            if (x0 > 0 && a0 > 0 && a1 > 0 && a2 > 0)
                Image[x][y] = TriangleColor;
        }
    }
}
```

### 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

### Edge 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

**Silver**

**Moving Slivers**
### Triangle Rasterization Issues

#### Shared Edge Ordering

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

### Interpolation During Scan Conversion

- Interpolate between vertices: (demo)
  - $z$
  - $r, g, b$ color components
  - $u, v$ texture coordinates
- We’ll discuss a better way for these next lecture
  - $N^e, N^s, N^t$ surface normals
- Three equivalent ways of viewing this (for triangles)
  1. **Bilinear interpolation**
  2. **Barycentric coordinates**
  3. **Plane Equation**

### 1. Bilinear Interpolation

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

### 2. Barycentric Coordinates

- **Weighted combination of vertices**
  \[ P = \alpha \cdot P_1 + \beta \cdot P_2 + \gamma \cdot P_3 \]
  \[ \alpha + \beta + \gamma = 1 \]
  \[ 0 \leq \alpha, \beta, \gamma \leq 1 \]
  "convex combination of points"

### 3. Plane Equation

- **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

### Discussion

- **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

**Modern GPUs:**
- Use edge equations
  - And plane equations for 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. space-filling curve rather than scanlines

Coming Up...

**Monday**
- Texture Mapping

**Wednesday**
- Color