Scan Conversion
CPSC 314

The Rendering Pipeline

Geometry Processing

Geometry Database → Model/View Transform. → Lighting → Perspective Transform. → Clipping

Scan Conversion → Texturing → Depth Test → Blending → Frame-buffer

Rasterization → Fragment Processing
Line Clipping

**Outcodes (Cohen, Sutherland '74)**

- 4 flags encoding position of a point relative to top, bottom, left, and right boundary
- E.g.:
  - OC(p1)=0010
  - OC(p2)=0000
  - OC(p3)=1001

```
<table>
<thead>
<tr>
<th>x=x_{min}</th>
<th>y=y_{min}</th>
<th>x=x_{max}</th>
<th>y=y_{max}</th>
</tr>
</thead>
<tbody>
<tr>
<td>0010</td>
<td>0000</td>
<td>0001</td>
<td>1001</td>
</tr>
<tr>
<td>0110</td>
<td>0100</td>
<td>0101</td>
<td></td>
</tr>
</tbody>
</table>
```

© Wolfgang Heidrich
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)

**α-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 \(p=(x, y)^T\)
  - \(\text{WEC}_L(p)= x-x_{\text{min}}\)
  - \(\text{WEC}_R(p)= x_{\text{max}} - x\)
  - \(\text{WEC}_B(p)= y-y_{\text{min}}\)  \textbf{Negative if outside!}
  - \(\text{WEC}_T(p)= y_{\text{min}} - y\)
**Line Clipping**

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

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

**Polygon Clipping**

*Example*

- Inside
- Outside
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!
BSP Trees: Viewpoint B

© Wolfgang Heidrich

BSP Trees: Viewpoint B

© Wolfgang Heidrich
**Hidden Surface Removal: Z-Buffer / Depth Buffer**

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

- typically 8+8+8+24 bits, can be more

  ```
  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**

- Therefore, depth-buffer essentially stores \(1/z\), rather than \(z\)!
- Issue with integer depth buffers
  - *High precision for near objects*
  - *Low precision for far objects*
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**

[Diagram of the rendering pipeline with steps: Geometry Database → Model/View Transform → Lighting → Perspective Transform → Clipping → Scan Conversion → Texturing → Depth Test → Blending → Frame-buffer]
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
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:**

def 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;
    }
}

© Wolfgang Heidrich
Scan Conversion of Lines

**DDA:**

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**
Scan Conversion of Lines

**Idea: decision variable**

```c
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, \( d \leq 0 \) )
  - \( x = x+1; \)
  - \( d = L(x+2, y+1/2) = d + (y_e - y_s) / (x_e - x_s) \)
- Case NE (point below line, \( d > 0 \) )
  - \( x = x+1; \ y = y+1; \)
  - \( d = L(x+2, y+3/2) = d + (y_e - y_s) / (x_e - x_s) - 1 \)
- Initialization:
  - \( d = L(x_s+1, y_s+1/2) = (y_e - y_s) / (x_e - x_s) - 1/2 \)
Scan Conversion of Lines

Bresenham Algorithm

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

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++; }
    }
}
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

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

```cpp
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

\[
\text{scanTrapezoid}(x_3, x_m, y_3, y_b, \frac{1}{m_{13}}, \frac{1}{m_{12}})
\]
\[
\text{scanTrapezoid}(x_2, x_2, y_2, y_3, \frac{1}{m_{23}}, \frac{1}{m_{12}})
\]

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 (x₁, y₁), (x₂, y₂)**

\[ Ax_1 + By_1 + C = 0 \]
\[ Ax_2 + By_2 + C = 0 \]

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

---

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

\[
\begin{bmatrix}
  A \\
  B
\end{bmatrix} = \frac{C}{x_0y_1 - x_1y_0}
\begin{bmatrix}
  y_1 - y_0 \\
  x_1 - x_0
\end{bmatrix}
\]

- Choose \( C = x_0y_1 - x_1y_0 \) for convenience
- Then \( A = y_0 - y_1 \) and \( B = x_1 - x_0 \)
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

```
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

```
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

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

Sliver

Moving Slivers
Triangle Rasterization Issues

**Shared Edge Ordering**

![Diagram of triangle rasterization issues]

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
  - $N_x, N_y, N_z$  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

![Diagram of bilinear interpolation]

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”

![Diagram of barycentric coordinates]
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