**Painter’s HSR Algorithm**

- Render polygons farthest to nearest
- Similar to painter layers oil paint

Viewer sees B behind A

Render B then A
Depth Sort

- Requires sorting polygons (based on depth)
  - $O(n \log n)$ complexity to sort $n$ polygon depths
  - Not every polygon is clearly in front or behind other polygons

Polygons sorted by distance from COP
Easy Cases

- Case a: A lies behind all polygons

- Case b: Polygons overlap in z but **not** in x or y
Hard Cases

Overlap in \((x,y)\) and \(z\) ranges

cyclic overlap

penetration
Back Face Culling

- **Back faces**: faces of opaque object that are “pointing away” from viewer
- **Back face culling**: do not draw back faces (saves resources)

How to detect back faces?
Back Face Culling

- **Goal:** Test if a face F is a backface
- **How?** Form vectors
  - View vector, V
  - Normal N to face F

![Diagram showing view vector V and normal N to a face F]

**Backface test:** F is backface if $N \cdot V < 0$  
why??
void drawFrontFaces()
{
    for(int f = 0; f < numFaces; f++)
    {
        if(isBackFace(f, ....) continue; \textcolor{red}{\textbf{if } N.V < 0}
        glDrawArrays(GL_POLYGON, 0, N);
    }
}
View-Frustum Culling

- **Goal**: Remove objects outside view frustum
- Done by 3D clipping algorithm (e.g. Liang-Barsky)
Ray Tracing

- **Ray tracing** is another image space method
- Ray tracing: Cast a ray from eye through each pixel into world.
- Ray tracing algorithm figures out: what object seen in direction through given pixel?

Topic of grad class
Combined z-buffer and Gouraud Shading (Hill)

- Can combine shading and hsr through scan line algorithm

```java
for(int y = ybott; y <= ytop; y++) // for each scan line
{
    for(each polygon)
    {
        find xleft and xright
        find dleft, dright, and dinc
        find colorleft and colorright, and colorinc
        for(int x = xleft, c = colorleft, d = dleft; x <= xright;
            x++, c+= colorinc, d+= dinc)
        if(d < d[x][y])
            { put c into the pixel at (x, y)
                d[x][y] = d; // update closest depth
            }
    }
}
```
Computer Graphics (CS 4731)  
Lecture 22: Rasterization: Line Drawing  

Prof Emmanuel Agu  

Computer Science Dept.  
Worcester Polytechnic Institute (WPI)
Rasterization

- Rasterization generates set of **fragments**
- Implemented by graphics hardware
- Rasterization algorithms for primitives (e.g., lines, circles, triangles, polygons)

**Rasterization: Determine Pixels**

(fragments) each primitive covers
Line drawing algorithm

- Programmer specifies \((x,y)\) of end pixels
- Need algorithm to determine pixels on line path

Line: \((3,2)\) -> \((9,6)\)

Which intermediate pixels to turn on?
Line drawing algorithm

- Pixel (x,y) values constrained to integer values
- Computed intermediate values may be floats
- Rounding may be required. E.g. (10.48, 20.51) rounded to (10, 21)
- Rounded pixel value is off actual line path (jaggy!!)
- Sloped lines end up having jaggies
- Vertical, horizontal lines, no jaggies
Line Drawing Algorithm

- **Slope-intercept line equation**
  - \( y = mx + b \)
  - Given 2 end points \((x_0, y_0), (x_1, y_1)\), how to compute \(m\) and \(b\)?

  \[
  m = \frac{dy}{dx} = \frac{y_1 - y_0}{x_1 - x_0} \quad y_0 = m \cdot x_0 + b \\
  \Rightarrow b = y_0 - m \cdot x_0
  \]
Line Drawing Algorithm

● Numerical example of finding slope $m$:
  ● $(Ax, Ay) = (23, 41), (Bx, By) = (125, 96)$

$$m = \frac{By - Ay}{Bx - Ax} = \frac{96 - 41}{125 - 23} = \frac{55}{102} = 0.5392$$
Digital Differential Analyzer (DDA): Line Drawing Algorithm

Consider slope of line, \( m \):

- \( m < 1 \) \( x \) incrementing faster
  - Step in \( x=1 \) increments, compute \( y \) (a fraction) and round
- \( m > 1 \) \( y \) incrementing faster
  - Step in \( y=1 \) increments, compute \( x \) (a fraction) and round
- \( m = 1 \)
- \( m > 1 \)
- \( m < 1 \)

- Step through line, starting at \((x_0, y_0)\)
- **Case a: \((m < 1)\)** \( x \) incrementing faster
  - Step in \( x=1 \) increments, compute \( y \) (a fraction) and round
- **Case b: \((m > 1)\)** \( y \) incrementing faster
  - Step in \( y=1 \) increments, compute \( x \) (a fraction) and round
DDA Line Drawing Algorithm (Case a: m < 1)

\[ m = \frac{\Delta y}{\Delta x} = \frac{y_{k+1} - y_k}{x_{k+1} - x_k} = \frac{y_{k+1} - y_k}{1} \]

\[ \Rightarrow y_{k+1} = y_k + m \]

\[ x = x_0 \quad y = y_0 \]

Illuminate pixel \((x, \text{round}(y))\)

\[ x = x + 1 \quad y = y + m \]

Illuminate pixel \((x, \text{round}(y))\)

\[ x = x + 1 \quad y = y + m \]

Illuminate pixel \((x, \text{round}(y))\)

... etc

Example, if first end point is (0,0)

Example, if \(m = 0.2\)

Step 1: \(x = 1, y = 0.2\) => shade (1,0)

Step 2: \(x = 2, y = 0.4\) => shade (2, 0)

Step 3: \(x = 3, y = 0.6\) => shade (3, 1)

... etc
DDA Line Drawing Algorithm (Case b: m > 1)

\[ m = \frac{\Delta y}{\Delta x} = \frac{y_{k+1} - y_k}{x_{k+1} - x_k} = \frac{1}{x_{k+1} - x_k} \]

\[ \Rightarrow x_{k+1} = x_k + \frac{1}{m} \]

Example, if first end point is (0,0) if \(1/m = 0.2\)

Step 1: \( y = 1, x = 0.2 \) => shade (0,1)
Step 2: \( y = 2, x = 0.4 \) => shade (0, 2)
Step 3: \( y= 3, x = 0.6 \) => shade (1, 3)

... etc

x = x0  \( \quad \) y = y0

Illuminate pixel (round(x), y)

y = y + 1  \( \quad \) x = x + 1 /m

Illuminate pixel (round(x), y)

\[ y = y + 1 \quad \quad x = x + 1 /m \]

Illuminate pixel (round(x), y)

Until \( y == y1 \)
DDA Line Drawing Algorithm Pseudocode

compute m;
if m < 1:
{
    float y = y0;       // initial value
    for(int x = x0;  x <= x1;  x++, y += m)
        setPixel(x, round(y));
}
else   // m > 1
{
    float x = x0;       // initial value
    for(int y = y0;  y <= y1;  y++, x += 1/m)
        setPixel(round(x), y);
}

- **Note:** `setPixel(x, y)` writes current color into pixel in column x and row y in frame buffer
Line Drawing Algorithm Drawbacks

- DDA is the simplest line drawing algorithm
  - Not very efficient
  - Round operation is expensive
- Optimized algorithms typically used.
  - Integer DDA
  - E.g. Bresenham algorithm
- Bresenham algorithm
  - Incremental algorithm: current value uses previous value
  - Integers only: avoid floating point arithmetic
  - Several versions of algorithm: we’ll describe midpoint version of algorithm
References