Recall: OpenGL - Image Space Approach

- Paint pixel with color of closest object

```plaintext
for (each pixel in image) {
    determine the object closest to the pixel
    draw the pixel using the object’s color
}
```
Recall: Z (depth) Buffer Algorithm

Depth of polygon being 
rasterized at pixel \((x, y)\) 

Largest depth seen so far 
Through pixel \((x, y)\)

For each polygon 
{
    for each pixel \((x,y)\) in polygon area 
    {
        if \((z\_polygon\_pixel(x,y) < \text{depth\_buffer}(x,y))\) 
        {
            \text{depth\_buffer}(x,y) = z\_polygon\_pixel(x,y);
            \text{color\_buffer}(x,y) = \text{polygon color at } (x,y)
        }
    }
}

Note: know depths at vertices. Interpolate for interior 
z\_polygon\_pixel(x, y) depths
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 backface
- **How?** Form vectors
  - View vector, $V$
  - Normal $N$ to 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;  \[if N.V < 0\]
        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
                }
    }
}
```
Rasterization

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

**Rasterization: Determine Pixels**

(fragments) each primitive covers

Fragments
Line drawing algorithm

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

Line: \((3, 2) \rightarrow (9, 6)\)

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
  - \( \text{y} = \text{mx} + \text{b} \)
  - Given 2 end points \((\text{x}_0, \text{y}_0), (\text{x}_1, \text{y}_1)\), how to compute \( \text{m} \) and \( \text{b} \)?

\[
m = \frac{dy}{dx} = \frac{y_1 - y_0}{x_1 - x_0}
\]

\[
y_0 = m \cdot x_0 + b
\]

\[
\Rightarrow b = y_0 - m \cdot x_0
\]
Line Drawing Algorithm

- Numerical example of finding slope \( m \):
  - \((A_x, A_y) = (23, 41), (B_x, B_y) = (125, 96)\)

\[
m = \frac{B_y - A_y}{B_x - A_x} = \frac{96 - 41}{125 - 23} = \frac{55}{102} = 0.5392
\]
Digital Differential Analyzer (DDA): Line Drawing Algorithm

Consider slope of line, m:

- Step through line, starting at (x0,y0)
- **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: \( m < 1 \))

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

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

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: \( m > 1 \))

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

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

\( x = x_0 \quad y = y_0 \)

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

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

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

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

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

... Until \( y == y_1 \)

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

if \(1/m = 0.2\)

Step 1: \( y = 1, x = 0.2 \implies \text{shade (0,1)}\)

Step 2: \( y = 2, x = 0.4 \implies \text{shade (0, 2)}\)

Step 3: \( y = 3, x = 0.6 \implies \text{shade (1, 3)}\)

... etc
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
Bresenham’s Line-Drawing Algorithm

- **Problem**: Given endpoints \((Ax, Ay)\) and \((Bx, By)\) of line, determine intervening pixels
- First make two simplifying assumptions (remove later):
  - \((Ax < Bx)\) and
  - \((0 < m < 1)\)
- Define
  - Width \(W = Bx - Ax\)
  - Height \(H = By - Ay\)
Bresenham’s Line-Drawing Algorithm

- Based on assumptions \((Ax < Bx)\) and \((0 < m < 1)\)
  - \(W, H\) are +ve
  - \(H < W\)
- Increment \(x\) by +1, \(y\) incr by +1 or stays same
- Midpoint algorithm determines which happens
Bresenham’s Line-Drawing Algorithm

What Pixels to turn on or off?
Consider pixel midpoint \( M(Mx, My) = (x + 1, y + \frac{1}{2}) \)

Build equation of actual line, compare to midpoint

Case a: If midpoint (red dot) is below line, Shade upper pixel, \((x + 1, y + 1)\)

Case b: If midpoint (red dot) is above line, Shade lower pixel, \((x + 1, y)\)
References