Rasterization, Line Drawing, Polygon Filling & Hidden Surface Removal

Anders Hast
Rasterization

- Pixel colour is set in a scanline fashion
Line Drawing

- Digital Differential Analyser, DDA, (float)
- Implicit Lines
- Bresenham (integer)
  - Steps on pixel centers using integer operations
  - 8 connected
- Parametric Lines
  - 8 connected
  - 4 connected
Digital Differential Analyser

DDA

- A line in 2D is defined as: \( y = kx + m \) where: \( x \) and \( y \) are variables (screen coordinates)
- Starts at \( m=(x_0, y_0) \) and ends at \( (x_1, y_1) \)
- slope: \( k = \frac{\Delta y}{\Delta x} = (y_1 - y_0)/(x_1-x_0) \)
- Algorithm:
  - Start at \( (x_0, y_0) \); \( x=x_0, y=y_0 \)
  - Increase \( x \) by 1 and \( y \) by \( k \)
  - repeat until \( x=x_1 \)
DDA

- Works for $k=\leq 1$
- For $k>1$:
  Reverse the roles for $x$ and $y$.

Algorithm:

$$y = y + 1$$
$$x = x + 1/k$$
Implicit Lines

- Line
  \[ y = kx + m \]
- Implicit line
  \[ 0 = kx - y + m \]
- or
  \[ 0 = ax + by + c \]
### Bresenham

- 1965
- Integer Arithmetic
- Start at $f(x_0, y_0) = 0$
- Next pixel?
  - Choose $(x+1, y)$ or $(x+1, y+1)$
- Check the sign of $f(x+1, y+1/2)$
- Depending on the sign: set pixel above or below
Derivation

- Line equation
  \[ y = \left( \frac{\Delta y}{\Delta x} \right) x + m \]

- Rewrite
  \[ \Delta xy = \Delta y \cdot x + \Delta x m \]

- and the implicit line is
  \[ 0 = \Delta y \cdot x - \Delta xy + \Delta x m = f(x,y) \]

- Check the sign of
  \[ f(x+1,y+1/2) = \Delta y \cdot (x+1) - \Delta x \cdot (y+1/2) + \Delta x m \]
Derivation

- Make it integer!
  \[ 2f(x+1,y+1/2) = 2\Delta y \ (x+1) - 2\Delta x (y+1/2) + 2\Delta x m \]

- and
  \[ 2f(x+1,y+1/2) = 2\Delta y \ x + 2\Delta y - 2\Delta x y - \Delta x + 2\Delta x m \]

- The Line itself
  \[ 2f(x,y) = 2\Delta y \ x - 2\Delta x y + 2\Delta x m \]

- The difference (initial value for decision)
  \[ 2f(x+1,y+1/2) - 2f(x,y) = 2\Delta y - \Delta x \]
Derivation

- The line was
  \[ 2f(x,y) = 2\Delta y \cdot x - 2\Delta x \cdot y + 2\Delta x \cdot m \]
- Hence Increment
  \[ 2\Delta y - 2\Delta x \]
- or
  \[ 2\Delta y \]
Bresenham C code

```c
void brenenham1(int x1, int y1, int x2, int y2){
    int slope;
    int dx, dy, incE, incNE, d, x, y;

    dx = x2 - x1;
    dy = y2 - y1;
    // Adjust y-increment for negatively sloped lines
    if (dy < 0) { slope = -1; dy = -dy;} else{ slope = 1;}

    incE = 2 * dy;
    incNE = 2 * dy - 2 * dx;
    d = 2 * dy - dx;

    y = y1;
    for (x = x1; x <= x2; x++)
    {
        putpixel(x, y);
        if (d <= 0) { d += incE; }
        else { d += incNE;
            y += slope; }
    }
}
```
Parametric Lines

- Linear Interpolation
  \[ p(u) = p_0(1-u) + p_1(u) \]

- or
  \[ p(u) = vu + p_0 \]
  \[ v = p_1 - p_0 \]
Polygon Filling

- Not so easy...
- Two cases $\omega = (x_1y_2 - x_2y_1)$
Slopes

- The slope of the left edge is $k_1 = x_2 / y_2$.
- The slope of the upper right side is $k_r = x_1 / y_1$
- and the slope for the lower right side is $k_r = (x_2 - x_1) / (y_2 - y_1)$
- Hence we must fill 2 triangles!
Pseudo Code

Initialize

\[ x_1 = x_b - x_a; \]
\[ y_1 = y_b - y_a; \]
\[ x_2 = x_c - x_a; \]
\[ y_2 = y_c - y_a; \]
\[ \omega = x_1*y_2 - x_2*y_1; \]

IF \( \omega > 0 \)

Upper

\[ x_s = x_e = x_a; \]
\[ y_s = y_a; \]
\[ y_e = y_b; \]
\[ k_l = x_2/y_2; \]
\[ k_r = x_1/y_1; \]
\[ \text{rasterize();} \]
\[ x_s = x_e + k_l; \]
\[ y_s = y_b; \]
\[ y_e = y_c; \]
\[ k_r = (x_2 - x_1)/(y_2 - y_1); \]
\[ \text{rasterize();} \]
ELSE IF \( \omega < 0 \)

Upper

\[ x_s = x_e = x_a; \]
\[ y_s = y_a; \]
\[ y_e = y_b; \]
\[ k_l = x_1/y_1; \]
\[ k_r = x_2/y_2; \]
\[ \text{rasterize();} \]
\[ x_s = x_b; \]
\[ y_s = y_b; \]
\[ y_e = y_c; \]
\[ \text{rasterize();} \]
END

Draw Triangle

Function rasterize()

FOR \( y = y_s \) TO \( y_e - 1 \)

FOR \( x = x_s \) TO \( x_e - 1 \)

\[ \text{frameBuffer}[x][y] = \text{colour;} \]

END

END

END
Conclusion

- Line drawing is not so easy as one might think
- Polygon filling is even harder
- Then one should also interpolate colours or normals! (Gouraud or Phong)
- What about aliasing and fragments?
Hidden Surface Removal
Problem Definition

- The 3D world is projected onto a 2D screen
- Which one of the polygons will be visible if they partly occupy the same pixels in the framebuffer?
  - Obviously the one closest to the COP!
Hidden Surface Removal

- Clipping
- Graphics Pipeline
- The Problem
- Hidden Surface Elimination
  - Painters algorithm
  - Z-Buffer
- Culling
  - Portal Culling
  - Backface Culling
Clipping

- Clip everything outside the view frustum
The Graphics Pipeline

- A simplified Graphics Pipeline

Vertices → Transformation → Projection → Clipping → Rasterization → Pixels

Interesting parts!
Hidden Surface Removal

- The same thing but different names
  - Hidden Surface Elimination
  - Hidden Surface Determination
  - Occlusion Culling
  - Visible Surface Determination
Painters Algorithm

- Do like a painter, paint the background first then objects closer and closer...
  - Sort primitives by z
    - The centre of the polygon?!
  - Render from back to front
Let’s Paint Something
Put a Sun in the Sky
Add the Sea
A bird in the skye
Some Island
We need a boat
A Jumping Fish?! 😊
A Problem

- As seen by the example the polygons usually do not have constant Z
- Furthermore....
Polygons Can Intersect

- This cannot be handled by Painters Algorithm
  - Unless we clip polygons against each other...
- And it is not as unlikely as it might seem!
  - “Bad” models
  - Bullets that penetrate something...
  - Deforming meshes
The Z-Buffer Algorithm

- Invented by Ed Catmull
  1974
- Image Precision
- Very simple to Implement
- Implemented on GPU’s
- Renders primitives in arbitrary order
  - Sorting is not necessary
- Not perfect! (Precision problems)
Graphics Pipeline

- Z-Buffering happens in the Rasterizer

Vertices → Transformation → Projection → Clipping → Rasterization

Happens Here!

Pixels
The Z-Buffer (Depth Buffer)

- The Z-buffer has the same resolution as the frame buffer
  - Is initialized to some value... (range: 0.0 .. 1.0)?
- The Algorithm:
  - Record the depth value (z) into the z-buffer while writing a pixel on the scanline
  - However: Only write the pixel if the z value is less than previously recorded
In the Rasterizer

If \( z < \text{read\_DepthBuffer}(x,y) \) 
{
    \text{write\_frameBuffer}(x,y,\text{colour})
    \text{write\_DepthBuffer}(x,y,z)
}
Example
Next Triangle

måndag 23 april 12
The Depth Buffer

- A simple scene and the corresponding Depth Buffer
- The depth buffer:
  - can be used for compositing effects
    - Fog
    - Atmospheric scattering
    - etc
  - but cannot be retrieved
Problems

- The precision depends on the range of the far and near clipping planes
- Today often 24 bits
Problems

- If the planes are too far away the limited precision gives bad results
  - i.e. The algorithm cannot determine if the polygon is behind or in front of the already stored polygon
  - Hence set the front and far clipping planes carefully!
- Due to the perspective projection the precision is better in the front
- Some pixels will be set more than once...
  - Fragments?
A Game Engine

- Would use the Z-buffering technique implemented on the hardware
- Would probably use some technique to speed up the rendering
  - Discard polygons that are easily detected as hidden
  - Bounding volumes
  - Sorting like in the painters algorithm
Example

- Don’t even try to draw the cars behind the building
Portal Culling

- The world is divided into cells and portals
- In the example:
  - Four Cells
  - Three Portals
- Uses Clipping
  (view frustum culling)
Some Assumptions

- Note that we can NEVER see anything in the fourth cell from cell one and vice versa!
- Hence we do not need to render anything in cell four if we are positioned in cell one and vice versa

  i.e. That cell can be completely culled!
Conclusion

- The Z-buffer technique is more reliable than the Painters Algorithm
- To speed up things we can use sorting and bounding volumes
- Even clipping is useful for Portal Culling
  - Dividing the world in cells help us to cull cells we cannot possibly see
- In games and visualizations hybrid
Backface Culling

- An easy way to discard ca 50% of the polygons
- Works for closed objects!
Doesn’t work for

- Non closed objects
  - i.e. Object for which we can see the backside of the polygons
- Impostors
  - Polygons textured on both sides
Graphics Pipeline

- Can be done before or after projection
Parallel Projection

- Check the sign of the normal
- If it points away from you then you cannot see the polygon and you can discard it!
Perspective Projection

- Now it’s a different thing!
- How can we handle it??
Check the Angle (cosine)

- Compute the cosine between the Normal and the Projector, which is just a dot product!
  - Discard if the result is positive
>90 Degrees or Not?

- “Moving” the projector to the base of the Normal makes it easier to see what happens
OpenGL

- Does Backface Culling after Clipping
- Use the signed area in Device Coordinate space
  - i.e. The polygon on the screen will have negative area if it is a backface
  - Hence it is necessary to have some consistent ordering of the vertices (clockwise or counter clockwise)
The Signed Area Test

- The ordering of the vertices, counter clockwise or clockwise gives different signs

\[ A = (x_1 - x_0)(y_2 - y_0) - (x_2 - x_0)(y_1 - y_0) \]
Conclusion

- Backface Culling generally reduces the amount of polygons by almost 50%
- There are two ways to do it
  - Computing the cosine
    - can be done before projection
    - Generally faster as it is done earlier in the pipeline
  - The signed area test
    - is done after clipping
    - Allows to use normals that are not the exact
What we have Learned

- Line Drawing
  - Several Approaches
    - DDA (float)
    - Bresenham (integer)

- Polygon Filling
  - Take care of
    - Orientation
    - Upper and lower part
What we have Learned

- Hidden surface elimination is
  - Necessary to make correct renderings
  - Z-buffering is implemented on GPU’s
  - Sorting, bounding volumes and Clipping can be used to avoid unnecessary drawing of hidden polygons

- Backface culling
  - Reduces the amount of polygons for closed objects, by almost a half