Announcements

• Assignment 2 is due today
• Programming Assignment 2 will be out today
  • Demo/review in class on Thursday
  • Due Thursday after fall break (10/25)
• Late drop deadline (for undergrads) is next Monday (10/15)
• Talk to me if you have any questions or concerns

Last Time

• Talked about the purpose of the rasterization step
• Discussed line drawing
  • Presented several algorithms
  • Finished up with Bresenham’s algorithm
• Started on line anti-aliasing
  • Included a brief aside on filtering

Today

• Finish up line anti-aliasing
  • Ratio method
• Present several methods for polygon drawing
• Discuss hidden surface removal algorithms

Rasterization

• In the rasterization step, geometry in device coordinates is converted into fragments in screen coordinates
• After this step, there are no longer any “polygons”
Rasterization

- All geometry that makes it to rasterization is within the normalized viewing region
- All the rasterizer cares about is \((x, y)\)
- \(z\) is only used for z-buffering later on
- Need to convert continuous (floating point) geometry to discrete (integer) pixels

Line Drawing

- A classic part of the computer graphics curriculum
- Input:
  - Line segment definition
  - \((x_1, y_1), (x_2, y_2)\)
- Output:
  - List of pixels

How Do They Look?

- So now we know how to draw lines
- But they don’t look very good:
  ```
  \[
  \begin{array}{c}
  \vdots \\
  \vdots \\
  \vdots \\
  \vdots \\
  \vdots \\
  \vdots \\
  \end{array}
  \]
  ```
- Why not?
- Aliasing

Antialiasing

- Essentially 2 techniques:
  1. Supersample then filter
     - We discussed a simple averaging filter
  2. Compute the fraction of a line that should be applied to a pixel
     - Ratio method

Antialiasing #1: Supersampling

- Technique:
  1. Create an image 2x (or 4x, or 8x) bigger than the real image
  2. Scale the line endpoints accordingly
  3. Draw the line as before
     - No change to line drawing algorithm
  4. Average each 2x2 (or 4x4, or 8x8) block into a single pixel
Antialiasing #1: Supersampling

- No antialiasing
- 2x2 Supersampled
- Downsampling to original size

Antialiasing #2: Ratios

- Ratios: 100% 0% 0% 75% 25% 0% 50% 50% 0%

Polygon Drawing
- After clipping, we know that the entire polygon is inside the viewing region
- Makes the problem easier
- Need to determine which pixels are inside the polygon, and color those
- Find edges, and fill in between them
- Edges - Connected line segments
- How to fill?

Scan-Line Polygons
- Algorithm:
  1. Mark local minima and maxima
  2. Mark all distinct y values on edges
  3. For each scan line:
     1. Create pairs of edge pixels (going from left to right)
     2. Fill in between pairs

Scan-Line Polygons
- Difficulties:
  1. Need to handle local maxima/minima correctly
  2. Appear double in the edge pixel list
  3. Need to handle overlapping pixels correctly
  4. What to do if a pair of edge pixels map to the same pixel?
  5. Need to handle horizontal lines correctly
Scan-Line Polygon Example

Flood Fill

- 4-fill
  - Neighbor pixels are only up, down, left, or right from the current pixel
- 8-fill
  - Neighbor pixels are up, down, left, right, or diagonal

Flood Fill Algorithm

```c
void boundaryFill4(int x, int y, int fill, int boundary)
{
    int curr;
    curr = getPixel(x, y);
    if ((current != ... fill))
    {
        setColor(curr);
        boundaryFill4(x, y+1, fill, boundary);
        boundaryFill4(x, y-1, fill, boundary);
        boundaryFill4(x+1, y, fill, boundary);
        boundaryFill4(x-1, y, fill, boundary);
    }
}
```

Flood Fill Example

Difficulties with Flood-Fill

- Have to worry about stack depth
- How deep can you go?
- How do you choose the start point?
- Which buffer is used?
Difficulties with Flood Fill

- What happens in this case?

Which to Use?

- Scan-line is generally used for rasterization
- Flood-fill is generally used in applications responding to user input, like MS Paint

Done with Polygon Drawing

- Need to identify and mark the extents of the polygon, then fill in between them
- We discussed 2 algorithms:
  - Scan-Line
  - Flood Fill
- Any questions?

Continuing Down the Pipeline...

- At this point (the end of rasterization), we’ve converted all our graphics primitives to fragments
  - Basically, single pixels
- Now what we have to do is figure out which of these fragments make it to the screen
  - Backface culling
  - Depth culling

Hidden Surface Removal

- Alternatively, visible surface detection
- Need to determine which surfaces are visible to the user, and cull the rest
  - This came up briefly when we were talking about materials
- Some algorithms work on polygons, in the vertex processing stage
- Some algorithms work on fragments, in the fragment processing stage (i.e. this stage)
Backface Culling

- Where?
  - Object space
- When?
  - After transformation but before clipping
- What?
  - If \( \text{normal} \cdot \text{toViewer} < 0 \), discard face
  - That is, if the polygon face is facing away from the viewer, throw it out

- So what does this buy us?
  - Up to 50% fewer polygons to clip/rasterize
- Is this all we have to do?
  - No.
  - Can still have 2 (or more) front faces that map to the same screen pixel
  - Which actually gets drawn?

Depth Culling

- Can happen here (fragment processing)
  - z-buffering
- Can happen before rasterization
  - Painter's algorithm

- Need to maintain 1 z per fragment
- Why we project to a volume instead of a plane
- Maintain a separate depth buffer, the same size and resolution of the color buffer
  - Initialize this buffer to \( z = -1.1 \) (all \( z \) is in \([-1, 1]\))
  - As each fragment comes down the pipe, test \( z > \text{depth}[s][t] \)
  - If true, the fragment is in front of whatever was there before, so set \( \text{color}[s][t] = \text{frag.color} \) and \( \text{depth}[s][t] = \text{frag.z} \)

Depth Culling Example

- \( z = -0.7 \)
- \( z = 0 \)
- \( z = 0.5 \)
- \( z = -1.1 \)

Z-Buffering

- Where?
  - Fragment space
- When?
  - Immediately after rasterization
- How?
  - Basically, remember how far away polygons are, and only keep the ones that are in front

Z-Buffering Example

- \( z = -1.1 \)
- \( z = 2 \)
- \( z = -0.7 \)

NOTE: Can draw these shapes in any order
Z-Buffering
• Advantages:
  • Always works. The nearest object always determines the color of a pixel
  • Easy to understand / Easy to code
  • Does not require any global knowledge about the scene
• Disadvantages:
  • Expensive with memory
  • Needs a whole extra buffer
  • Not really a problem anymore

Painter’s Algorithm
• Really a class of algorithms
  • Somehow sort the objects by distance from the viewer
  • Draw objects in order from farthest to nearest
    • The entire object
    • Nearer objects will "overwrite" farther ones

Painter’s Algorithm
• Does anyone see a problem with this?
  • Objects can have a range of depth, not just a single value
  • Need to make sure they don’t overlap for this algorithm to work

Painter’s Algorithm
1. Sort all objects’ $z_{\text{min}}$ and $z_{\text{max}}$
2. If an object is uninterrupted (its $z_{\text{min}}$ and $z_{\text{max}}$ are adjacent in the sorted list), it is fine
3. If 2 objects DO overlap
   1. Check if they overlap in x
      • If not, they are fine
   2. Check if they overlap in y
      • If not, they are fine
      • If yes, need to split one

Painter’s Algorithm
• The splitting step is the tough one
  • Need to find a plane to split one polygon by so that each new polygon is entirely in front of or entirely behind the other
  • Polygons may actually intersect, so then need to split each polygon by the other
  • After splitting, you can resort the list and should be fine

Painter’s Example
Sort by depth:
Green rect
Red circle
Blue tri
Next Time

• More fragment processing
• Texture mapping
• Demo/discussion of programming assignment 2
• Written assignment 2 handed back