Announcements

- Assignment 2 is out today
  - Due next Tuesday by the end of class

Last Time

- Reviewed the OpenGL pipeline
- Discussed classical viewing and presented a taxonomy of different views
- Talked about how projections and viewport transforms are used in OpenGL

Today

- Discuss clipping
  - Points
  - Lines
  - Polygons

Rendering Pipeline

- OpenGL rendering works like an assembly line
  - Each stage of the pipeline performs a distinct function on the data flowing by
  - Each stage is applied to every vertex to determine its contribution to output pixels
Determining What’s in the Viewport

- Not all primitives map to inside the viewport
  - Some are entirely outside
  - Need to **cull**
  - Some are partially inside and partially outside
  - Need to **clip**
  - There must be **NO DIFFERENCE** to the final rendered image

Why Clip?

- Rasterization is very expensive
  - Approximately linear with number of fragments created
  - Math and logic per pixel
  - If we only rasterize what is actually viewable, we can save a lot
  - A few operations now can save many later

Clipping Primitives

- Different primitives can be handled in different ways
  - Points
  - Lines
  - Polygons

Point Clipping

- This one is easy
  - How to determine if a point \((x, y, z)\) is in the viewing volume \((x_{near}, y_{near}, z_{near}), (x_{far}, y_{far}, z_{far})\)?
  - Who wants to tell me how?
  - if \(((x > x_{far} \lor x < x_{near}) \land
  (y > y_{far} \lor y < y_{near}) \land
  (z > z_{far} \lor z < z_{near}))\)
    cull the point
  else
    keep it

Line Clipping

- What happens when a line passes out of the viewing volume/plane?
  - Part is visible, part is not
  - Need to find the entry/exit points, and shorten the line
  - The shortened line is what gets passed to rasterization
Line Clipping Example

- Let’s do 2D.
- Clip a line against 1 edge of the viewport.
- What do we know?
  - Similar triangles:
    - \( A / B = C / D \)
    - \( B = (x_2 - x_1) \)
    - \( A = (y_2 - y_1) \)
    - \( C = (y_1 - y_{\text{max}}) \)
    - \( D = BC / A \)
  - \((x', y') = (x_1 - D, y_1 - D)\)

Cohen-Sutherland Line Clipping

- Split plane into 9 regions:
- Assign each a 4-bit tag:
  - (above, below, right, left)
- Assign each endpoint a tag:

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>1001</td>
<td>1000</td>
<td>1010</td>
<td></td>
</tr>
<tr>
<td>0001</td>
<td>0000 Viewport</td>
<td>0010</td>
<td></td>
</tr>
<tr>
<td>0101</td>
<td>0100</td>
<td>0100</td>
<td></td>
</tr>
</tbody>
</table>

Cohen-Sutherland Example

- What are the vertex codes for these lines?

<table>
<thead>
<tr>
<th>G</th>
<th>H</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>J</td>
</tr>
<tr>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>C</td>
<td>D</td>
</tr>
<tr>
<td>E</td>
<td>F</td>
</tr>
</tbody>
</table>

Cohen-Sutherland Line Clipping

- Algorithm:
  1. if \((\text{tag}_{x} \& \text{tag}_{y} == 0000)\) accept the line
  2. if \((\text{tag}_{x} \& \text{tag}_{y} != 0)\) reject the line
  3. Clip the line against an edge (where both bits are nonzero)
  4. Assign the new vertex a 4-bit value
  5. Return to 1

N.B.: \& is the bitwise AND operator

Line Clipping

- The other cases are handled similarly.
- The algorithm extends easily to 3D.
- The problem?
  - Too expensive! (these numbers are for 2D)
  - 3 floating point subtracts
  - 2 floating point multiplies
  - 1 floating point divide
  - 4 times! (once for each edge)
- We need to do better

Cohen-Sutherland Line Clipping

- Lets us eliminate many edge clips early.
- Extends easily to 3D.
  - 27 regions
  - 6 bits
- Similar triangles still works in 3D.
  - Just have to do it for 2 sets of similar triangles.
Consider the parametric definition of a line:

\[ x = x_1 + u \Delta x \]
\[ y = y_1 + u \Delta y \]
\[ \Delta x = (x_2 - x_1), \quad \Delta y = (y_2 - y_1), \quad 0 \leq (u, v) \leq 1 \]

What if we could find the range for \( u \) and \( v \) in which both \( x \) and \( y \) are inside the viewport?

Mathematically, this means:

\[ x_{\text{min}} \leq x_1 + u \Delta x \leq x_{\text{max}} \]
\[ y_{\text{min}} \leq y_1 + u \Delta y \leq y_{\text{max}} \]

Rearranging, we get:

\[ -u \Delta x \leq (x_1 - x_{\text{min}}) \]
\[ u \Delta x \leq (x_{\text{max}} - x_1) \]
\[ -v \Delta y \leq (y_1 - y_{\text{min}}) \]
\[ v \Delta y \leq (y_{\text{max}} - y_1) \]

In general: \( u \cdot p_k \leq q_k \)

**Cases:**

1. \( p_k = 0 \)
   - Line is parallel to boundaries
   - If for the same \( k \), \( q_k < 0 \), reject
   - Else, accept
2. \( p_k < 0 \)
   - Line starts outside this boundary
     - \( r_k = q_k / p_k \)
     - \( u_1 = \max(0, r_k, u_1) \)
3. \( p_k > 0 \)
   - Line starts outside this boundary
     - \( r_k = q_k / p_k \)
     - \( u_2 = \min(1, r_k, u_2) \)
4. If \( u_1 > u_2 \), the line is completely outside

**Also extends to 3D**

- Just add equations for \( z = z_1 + u \Delta z \)
  - \( 2 \) more \( p \)'s and \( q \)'s

In most cases, Liang-Barsky is slightly more efficient

- According to the Hearn-Baker textbook
- Avoids multiple shortenings of line segments

However, Cohen-Sutherland is much easier to understand (I think)

- An important issue if you're actually implementing
Nicholl-Lee-Nicholl Line Clipping

- This is a theoretically optimal clipping algorithm (at least in 2D)
- However, it only works well in 2D
- More complicated than the others
- Just do an overview here

Nicholl-Lee-Nicholl Line Clipping

- Partition the region based on the first point ($p_1$):
  - Case 1: $p_1$ inside region
  - Case 2: $p_1$ across edge
  - Case 3: $p_1$ across corner

A Note on Redundancy

- Why am I presenting multiple forms of clipping?
- Why do you learn multiple sorts?
  - Fastest can be harder to understand / implement
  - Best for the general case may not be for the specific case
  - Bubble sort is really great on mostly sorted lists
  - “History repeats itself”
  - You may need to use a similar algorithm for something else; grab the closest match

Polygon Inside/Outside

- Polygons have a distinct inside and outside
- How do you tell, just from a list of vertices/edges?
  - Even/odd
  - Winding number

Polygon Inside/Outside: Even / Odd

- Count edge crossings
  - If the number is even, that area is outside
  - If odd, it is inside
Polygon Inside/Outside: Winding Number

- Each line segment is assigned a direction by walking around the edges in some pre-defined order
- OpenGL walks counter-clockwise
- Count right->left edge crossings and left->right edge crossings
- If equal, the point is outside

Polygons

Polygons are just composed of lines. Why do we need to treat them differently?
- Need to keep track of what is inside

Lines

Polygons

NOTE:

Polygon Clipping

- Many tricky bits
- Maintaining inside/outside
- Introduces variable number of vertices
- Need to handle screen corners correctly

Sutherland-Hodgeman Polygon Clipping

- Simplify via separation
- Clip the entire polygon with one edge
- Clip the output polygon against the next edge
- Repeat for all edges
- Extends easily to 3D (6 edges instead of 4)
- Can create intermediate vertices that get thrown out later

Sutherland-Hodgeman Polygon Clipping

- Example 1:
  - Out -> In: Save new clip vertex and ending vertex
  - In -> Out: Save ending vertex
  - Out -> Out: Save nothing

Sutherland-Hodgeman Polygon Clipping

- Example 2:
  - Start
  - Clip Left
  - Clip Right
  - Clip Bottom
  - Clip Top

Sutherland-Hodgeman Polygon Clipping

- Example 3

Sutherland-Hodgeman Polygon Clipping

- Example 3

NOTE:
Weiler-Atherton Polygon Clipping

- When using Sutherland-Hodgeman, concavities can end up linked

  Remember this?

- A different clipping algorithm, the Weiler-Atherton algorithm, creates separate polygons

Weiler-Atherton Polygon Clipping

- Example:

  ![Example Diagram]

  - Out -> In: Add clip vertex
  - In -> In: Add end vertex
  - In -> Out: Add clip vertex
  - Follow clip edge until (a) new crossing found (b) reach vertex already added

Weiler-Atherton Polygon Clipping

- Example (cont'd):

  ![Example Diagram (cont'd)]

  - Continue from cached vertex and direction
  - Out -> In: Add clip vertex
  - In -> Out: Add clip vertex
  - Follow clip edge until (a) new crossing found (b) reach vertex already added

  Final Result: 2 unconnected polygons

Weiler-Atherton Polygon Clipping

- Example (cont'd):

  ![Example Diagram (cont'd)]

  - Continue from cached vertex and direction
  - Nothing added
  - Finished

Done with Clipping

- Point Clipping (really just culling)
  - Easy, just do inequalities
- Line Clipping
  - Cohen-Sutherland
  - Liang-Barsky
  - Nicholl-Lee-Nicholl
- Polygon Clipping
  - Sutherland-Hodgeman
  - Weiler-Atherton

Any Questions?
Next Time

- Moving on down the pipeline
- Rasterization
- Line drawing