The Graphics Pipeline, Revisited

- Must eliminate objects that are outside of viewing frustum
- Clipping: object space (eye coordinates)
- Scissoring: image space (pixels in frame buffer)
  - most often less efficient than clipping
- We will first discuss 2D clipping (for simplicity)
  - OpenGL uses 3D clipping

Clipping Against a Frustum

- General case of frustum (truncated pyramid)
  
  - Clipping is tricky because of frustum shape

Perspective Normalization

- Solution:
  - Implement perspective projection by perspective normalization and orthographic projection
  - Perspective normalization is a homogeneous transformation

The Normalized Frustum

- OpenGL uses \(-1 \leq x,y,z \leq 1\) (others possible)
- Clip against resulting cube
- Clipping against arbitrary (programmer-specified) planes requires more general algorithms and is more expensive

The Viewport Transformation

- Transformation sequence again:
  1. Camera: From object coordinates to eye coords
  2. Perspective division: to normalized device coords
  3. Clipping
  4. Orthographic projection (setting \(z_p = 0\))
  5. Viewport transformation: to screen coordinates
- Viewport transformation can distort
  - Solution: pass the correct window aspect ratio to `gluPerspective`
Clipping

- General: 3D object against cube
- Simpler case:
  - In 2D: line against square or rectangle
  - Later: polygon clipping

Clipping Against Rectangle in 2D

- Line-segment clipping: modify endpoints of lines to lie within clipping rectangle

Several practical algorithms for clipping

- Main motivation:
  Avoid expensive line-rectangle intersections (which require floating point divisions)
- Cohen-Sutherland Clipping
- Liang-Barsky Clipping
- There are many more (but many only work in 2D)

Cohen-Sutherland Clipping

- Clipping rectangle is an intersection of 4 half-planes
- Encode results of four half-plane tests
- Generalizes to 3 dimensions (6 half-spaces)
Outcodes (Cohen-Sutherland)

- Divide space into 9 regions
- 4-bit outcode determined by comparisons

\[
\begin{array}{c|c|c|c}
\text{ymax} & 1001 & 1010 & 0100 \\
\hline
0001 & 0000 & 0110 & 0010 \\
\hline
\text{ymin} & 0101 & 0110 & 0110 \\
\hline
\end{array}
\]

- \( b_0: y > y_{\text{max}} \)
- \( b_1: y < y_{\text{min}} \)
- \( b_2: x > x_{\text{max}} \)
- \( b_3: x < x_{\text{min}} \)

\[ o_1 = \text{outcode}(x_1, y_1) \]
\[ o_2 = \text{outcode}(x_2, y_2) \]

Cases for Outcodes

- Outcomes: accept, reject, subdivide

\[
\begin{array}{c|c|c|c}
1001 & 1010 & 0000 & 0100 \\
\hline
0001 & 0000 & 0110 & 0010 \\
\hline
\end{array}
\]

- \( o_1 = o_2 = 0000: \) accept entire segment
- \( o_1 \& o_2 \neq 0000: \) reject entire segment
- \( o_1 = 0000, o_2 \neq 0000: \) subdivide
- \( o_1 \neq 0000, o_2 = 0000: \) subdivide

Cohen-Sutherland Subdivision

- Pick outside endpoint (\( o = 0000 \))
- Pick a crossed edge (\( o = b_0 b_1 b_2 b_3 \) and \( b_k = 0 \))
- Compute intersection of this line and this edge
- Replace endpoint with intersection point
- Restart with new line segment
  - Outcodes of second point are unchanged
- This algorithms converges

Liang-Barsky Clipping

- Start with parametric form for a line
  \[
  p(\alpha) = (1 - \alpha)p_1 + \alpha p_2, \quad 0 \leq \alpha \leq 1
  \]
  \[
  x(\alpha) = (1 - \alpha)x_1 + \alpha x_2
  \]
  \[
  y(\alpha) = (1 - \alpha)y_1 + \alpha y_2
  \]

Ordering of intersection points

- Order the intersection points
  - Figure (a): \( 1 > \alpha_4 > \alpha_3 > \alpha_2 > \alpha_1 > 0 \)
  - Figure (b): \( 1 > \alpha_4 > \alpha_2 > \alpha_3 > \alpha_1 > 0 \)
Liang-Barsky Idea

- It is possible to clip already if one knows the order of the four intersection points!
- Even if the actual intersections were not computed!
- Can enumerate all ordering cases

Liang-Barsky efficiency improvements

- Efficiency improvement 1:
  - Compute intersections one by one
  - Often can reject before all four are computed
- Efficiency improvement 2:
  - Equations for $\alpha_3$, $\alpha_2$
  
  $$y_{\text{max}} = (1 - \alpha_3)y_1 + \alpha_3y_2$$
  $$x_{\text{min}} = (1 - \alpha_2)x_1 + \alpha_2x_2$$
  $$\alpha_3 = \frac{y_{\text{max}} - y_1}{y_2 - y_1}$$
  $$\alpha_2 = \frac{x_{\text{min}} - x_1}{x_2 - x_1}$$
  - Compare $\alpha_3$, $\alpha_2$ without floating-point division

Line-Segment Clipping Assessment

- Cohen-Sutherland
  - Works well if many lines can be rejected early
  - Recursive structure (multiple subdivisions) is a drawback
- Liang-Barsky
  - Avoids recursive calls
  - Many cases to consider (tedious, but not expensive)

Outline

- Line-Segment Clipping
  - Cohen-Sutherland
  - Liang-Barsky
- Polygon Clipping
  - Sutherland-Hodgeman
- Clipping in Three Dimensions

Polygon Clipping

- Polygon is clipped into one or more polygons
- Their union is intersection with clip window
- Alternatively, we can first tessellate concave polygons (OpenGL supported)

Concave Polygons

- Approach 1: clip, and then join pieces to a single polygon
  - Often difficult to manage
- Approach 2: tessellate and clip triangles
  - This is the common solution
Sutherland-Hodgeman (part 1)

- Subproblem:
  - Input: polygon (vertex list) and single clip plane
  - Output: new (clipped) polygon (vertex list)
- Apply once for each clip plane
  - 4 in two dimensions
  - 6 in three dimensions
  - Can arrange in pipeline

Sutherland-Hodgeman (part 2)

- To clip vertex list (polygon) against a half-plane:
  - Test first vertex. Output if inside, otherwise skip.
  - Then loop through list, testing transitions
    - In-to-in: output vertex
    - In-to-out: output intersection
    - Out-to-in: output intersection and vertex
    - Out-to-out: no output
  - Will output clipped polygon as vertex list
- May need some cleanup in concave case
- Can combine with Liang-Barsky idea

Other Cases and Optimizations

- Curves and surfaces
  - Do it analytically if possible
  - Otherwise, approximate curves / surfaces by lines and polygons
- Bounding boxes
  - Easy to calculate and maintain
  - Sometimes big savings

Outline

- Line-Segment Clipping
  - Cohen-Sutherland
  - Liang-Barsky
- Polygon Clipping
  - Sutherland-Hodgeman
- Clipping in Three Dimensions

Clipping Against Cube

- Derived from earlier algorithms
- Can allow right parallelepiped

Cohen-Sutherland in 3D

- Use 6 bits in outcode
  - \( b_4 : z > z_{\text{max}} \)
  - \( b_5 : z < z_{\text{min}} \)
- Other calculations as before
Liang-Barsky in 3D

- Add equation \( z(\alpha) = (1 - \alpha) z_1 + \alpha z_2 \)
- Solve, for \( p_0 \) in plane and normal \( n \):
  \[
p(\alpha) = (1 - \alpha)p_1 + \alpha p_2 \\
  n \cdot (p(\alpha) - p_0) = 0
\]

- Yields
  \[
  \alpha = \frac{n \cdot (p_0 - p_1)}{n \cdot (p_2 - p_1)}
  \]
- Optimizations as for Liang-Barsky in 2D

Summary: Clipping

- Clipping line segments to rectangle or cube
  - Avoid expensive multiplications and divisions
  - Cohen-Sutherland or Liang-Barsky
- Polygon clipping
  - Sutherland-Hodgeman pipeline
- Clipping in 3D
  - Essentially extensions of 2D algorithms

Preview and Announcements

- Scan conversion
- Anti-aliasing
- Other pixel-level operations
- Assignment 2 due a week from today!