CSCI 420 Computer Graphics
Lecture 13

Clipping

Line Clipping
Polygon Clipping
Clipping in Three Dimensions
[Angel Ch. 6]

Jernej Barbic
University of Southern California
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

See [Angel Ch. 5.9]
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 normalization**: to clip coordinates
  3. **Clipping**
  4. **Perspective division**: to normalized device coords.
  5. **Orthographic projection** (setting $z_p = 0$)
  6. **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
Clipping Against Rectangle in 2D

• The result (in red)
Clipping Against Rectangle in 2D

- Could calculate intersections of line segments with clipping rectangle
  - expensive, due to floating point multiplications and divisions
- Want to minimize the number of multiplications and divisions

\[ y = kx + n \]

\[ x = x_0 \]

\[ y = y_0 \]

\[ x = x_1 \]

\[ y = y_1 \]
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

\[
\text{interior} = \bigcap_{y < y_{\max}} y > y_{\min} \cap x > x_{\min} \cap x < x_{\max}
\]

• Encode results of four half-plane tests
• Generalizes to 3 dimensions (6 half-planes)
Outcodes (Cohen-Sutherland)

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

\[
\begin{align*}
\text{b}_0 &: y > y_{\text{max}} \\
\text{b}_1 &: y < y_{\text{min}} \\
\text{b}_2 &: x > x_{\text{max}} \\
\text{b}_3 &: x < x_{\text{min}} \\
o_1 &= \text{outcode}(x_1, y_1) \\
o_2 &= \text{outcode}(x_2, y_2)
\end{align*}
\]
Cases for Outcodes

- Outcomes: accept, reject, subdivide

- $o_1 = o_2 = 0000$: 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
- $o_1 \& o_2 = 0000$: subdivide
Cohen-Sutherland Subdivision

- Pick outside endpoint \((o \neq 0000)\)
- Pick a crossed edge \((o = b_0b_1b_2b_3 \text{ and } b_k \neq 0)\)
- 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 algorithm converges
Liang-Barsky Clipping

- Start with parametric form for a line

\[
p(\alpha) = (1 - \alpha)p_1 + \alpha p_2, \quad 0 \leq \alpha \leq 1
\]

\[
x(\alpha) = (1 - \alpha)x_1 + \alpha x_2
\]

\[
y(\alpha) = (1 - \alpha)y_1 + \alpha y_2
\]
Liang-Barsky Clipping

- Compute all four intersections $1, 2, 3, 4$ with extended clipping rectangle
- Often, no need to compute all four intersections
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 $\alpha_3$, $\alpha_2$
    
    \[
    y_{\text{max}} = (1 - \alpha_3)y_1 + \alpha_3y_2 \\
    x_{\text{min}} = (1 - \alpha_2)x_1 + \alpha_2x_2 \\
    \alpha_3 = \frac{y_{\text{max}} - y_1}{y_2 - y_1} \quad \alpha_2 = \frac{x_{\text{min}} - x_1}{x_2 - x_1}
    \]
  
  - 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 tesselate concave polygons (OpenGL supported)
Concave Polygons

• Approach 1: clip, and then join pieces to a single polygon
  – often difficult to manage

- {
  (a)
  (b)

• Approach 2: tesselate and clip triangles
  – this is the common solution

- {tessellation}
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!