Clipping

**Purpose**
- Originally: 2D
  - Determine portion of line inside an axis-aligned rectangle (screen or window)
- 3D
  - Determine portion of line inside axis-aligned parallelepiped (viewing frustum in NDC)
  - Simple extension to the 2D algorithms

**Outcodes (Cohen, Sutherland '74)**
- 4 flags encoding position of a point relative to top, bottom, left, and right boundary
- E.g.,
  - \( \text{OC}(p_1) = 0010 \)
  - \( \text{OC}(p_2) = 0000 \)
  - \( \text{OC}(p_3) = 1001 \)

<table>
<thead>
<tr>
<th>Flag</th>
<th>0000</th>
<th>0001</th>
<th>0010</th>
<th>0100</th>
<th>0101</th>
</tr>
</thead>
<tbody>
<tr>
<td>( y )</td>
<td>( y_{min} )</td>
<td>( y_{max} )</td>
<td>( y_{min} )</td>
<td>( y_{max} )</td>
<td></td>
</tr>
<tr>
<td>( x )</td>
<td>( x_{min} )</td>
<td>( x_{max} )</td>
<td>( x_{min} )</td>
<td>( x_{max} )</td>
<td></td>
</tr>
</tbody>
</table>

**Line segment:**
- \( (p_1, p_2) \)

**Trivial cases:**
- \( \text{OC}(p_1) == 0 \&\& \text{OC}(p_2) == 0 \)
  - Both points inside window, thus line segment completely visible (trivial accept)
- \( \text{OC}(p_1) \&\& \text{OC}(p_2) != 0 \)
  - There is (at least) one boundary for which both points are outside (same flag set in both outcodes)
  - Thus line segment completely outside window (trivial reject)
Line Clipping

\[ x + \alpha(x_2 - x_1) = x_{\text{min}} \]

\[ \alpha = \frac{x_{\text{max}} - x_1}{(x_2 - x_{\text{max}}) - (x_1 - x_{\text{min}})} \]

\[ \text{window} \]

\[ y = y_{\text{min}} \]

\[ x = x_{\text{min}} \]

\[ x = x_{\text{min}} \]

\[ y = y_{\text{max}} \]

\[ x = x_{\text{max}} \]

\[ y = y_{\text{max}} \]

\[ \text{Intersection point with one of the borders (say, left):} \]

\[ x + \alpha(x_2 - x_1) = x_{\text{min}} \]

\[ \alpha = \frac{x_{\text{max}} - x_1}{(x_2 - x_{\text{max}}) - (x_1 - x_{\text{min}})} \]

\[ \text{Line segment defined as:} \ p1 + \alpha(p2-p1) \]

\[ \text{Window-edge-coordinates of a point} \ p = (x, y) \]

\[ WEC_y(p) = y - y_{\text{min}} \]

\[ WEC_y(p) = y - y_{\text{max}} \]

\[ WEC_y(p) = y - y_{\text{min}} \]

\[ WEC_y(p) = y - y_{\text{max}} \]

\[ \text{Negative if outside!} \]

\[ \text{Line Clipping: algorithm} \]

\[ \text{alphaClip}(p1, p2, \text{window}) \{ \]

\[ \text{Determine window-edge-coordinates of p1, p2}\]

\[ \text{Determine outcodes OC(p1), OC(p2)}\]

\[ \text{Handle trivial accept and reject} \]

\[ \alpha_1 = 0; \ \text{// line parameter for first point} \]

\[ \alpha_2 = 1; \ \text{// line parameter for second point} \]

\[ \ldots \]

\[ \text{Similarly clip p1 against other edges} \]

\[ \ldots \]

\[ \text{Line Clipping: example for clipping p1} \]

\[ \text{Start configuration} \]

\[ \text{After clipping to left} \]

\[ \text{After clipping to top} \]
Line Clipping

α-Clipping: Algorithm (cont.)

```c
// now clip point p2 against all edges
if (OC(p2) & LEFT_FLAG) {
    α = WEC_t(p2)/(WEC_t(p1) - WEC_t(p2));
    α2 = min(α2, α);
}
Similarly clip p1 against other edges
```

Line Clipping

Example

![Example Diagram](image1)

Line Clipping

Another Example

![Another Example Diagram](image2)

Line Clipping in 3D

Approach:
- Clip against parallelepiped in NDC (after perspective transform)
- Means that the clipping volume is always the same!
  - OpenGL: \( x_{\min} \leq x \leq x_{\max}, y_{\min} \leq y \leq y_{\max}, z = 1 \)
  - Boundary lines become boundary planes
- But outcodes and WECs still work the same way
- Additional front and back clipping plane
  - \( z_{\min} = 0, z_{\max} = 1 \) in OpenGL

Line Clipping

Extensions
- Algorithm can be extended to clipping lines against
  - Arbitrary convex polygons (2D)
  - Arbitrary convex polytopes (3D)
**Line Clipping**

**Non-convex clipping regions**
- E.g.: windows in a window system!

**Polygon Clipping**

**Objective**
- 2D: clip polygon against rectangular window
  - Or general convex polygons
  - Extensions for non-convex or general polygons
- 3D: clip polygon against parallelepiped

**Polygon Clipping**

**Classes of Polygons**
- Triangles
- Convex
- Concave
- Holes and self-intersection

**Sutherland/Hodgeman Algorithm (‘74)**
- Arbitrary convex or concave object polygon
  - Restriction to triangles does not simplify things
- Convex subject polygon (window)
**Polygon Clipping**

*Sutherland/Hodgemann Algorithm ('74)*

- Approach: clip object polygon independently against all edges of subject polygon

---

**Polygon Clipping**

*Clipping against one edge:*

```cpp
clipPolygonToEdge( p[n-], edge ) { 
  for( i = 0 ; i < n ; i++ ) { 
    if( p[i] inside edge ) { 
      if( p[i-1] inside edge ) // p[1] = p[n-1] 
        output p[i]; 
      else { 
        p = intersect( p[i-1], p[i], edge ); 
        output p, p[i]; 
      } 
    } else... 
  } 
} // end of algorithm
```

---

**Polygon Clipping**

*Clipping against one edge (cont)*

- p[i] inside: 2 cases
- p[i] outside: 2 cases

---

**Example**

![Example Diagram](image)
**Polygon Clipping**

*Sutherland/Hodgeman Algorithm*
- Inside/outside tests: outcodes
- Intersection of line segment with edge: window-edge coordinates
- Similar to Cohen/Sutherland algorithm for line clipping

**Polygon Clipping**

*Sutherland/Hodgeman Algorithm*
- Discussion:
  - Clipping against individual edges independent
    - Great for hardware (pipelining)
  - All vertices required in memory at the same time
    - Not so good, but unavoidable
    - Another reason for using triangles only in hardware rendering

**Polygon Clipping**

*Other Polygon Clipping Algorithms*
- Weller/Auston ’77:
  - Arbitrary concave polygons with holes both as subject and as object polygon
- Vatti ’92:
  - Self intersection allowed as well

  ... many more
  - Improved handling of degenerate cases
  - But not often used in practice due to high complexity

**The Rendering Pipeline**
Coming Up:

**Tuesday, Oct 9:**
- Hidden surface removal / visibility

**Thursday, Oct 11:**
- Scan Conversion