Line 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.:
  - \( \text{OC}(p1)=0010 \)
  - \( \text{OC}(p2)=0000 \)
  - \( \text{OC}(p3)=1001 \)

<table>
<thead>
<tr>
<th></th>
<th>0110</th>
<th>0100</th>
<th>0101</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x = x_{\text{min}} )</td>
<td>0001</td>
<td>0000</td>
<td>0001</td>
</tr>
<tr>
<td>( x = x_{\text{max}} )</td>
<td>0010</td>
<td>0100</td>
<td>0101</td>
</tr>
</tbody>
</table>

**Line Clipping**

**Line segment:**
- \((p1, p2)\)

**Trivial cases:**
- \( \text{OC}(p1)==0 \&\& \text{OC}(p2)==0 \)
  - Both points inside window, thus line segment completely visible (trivial accept)
- \( \text{OC}(p1) \& \text{OC}(p2))=0 \)
  - 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**

**α-Clipping**
- Line segment defined as: \( p_1 + \alpha(p_2 - p_1) \)
- Intersection point with one of the borders (say, left):
  \[
  x_i + \alpha(x_e - x_i) = x_{\text{min}} \\
  \alpha = \frac{x_{\text{min}} - x_i}{x_e - x_i} = \frac{x_{\text{min}} - y_{\text{min}}}{y_e - y_{\text{min}}} \\
  \text{i.e.} \quad WEC_i(x_i) - WEC_i(x_e) \\
  \text{if} (OC(p_1) \& \& \text{LEFT\_FLAG}) \{
  \alpha = \frac{WEC_i(p_1)}{WEC_i(p_1) - WEC_i(p_2)}; \\
  \alpha_1 = \max(\alpha_1, \alpha); \\
  \}
- Similarly clip \( p_1 \) against other edges

**Line Clipping**

**Line Clipping**

**α-Clipping**
- Handling all the non-trivial cases
- Improvement of earlier algorithms (Cohen/Sutherland, Cyrus/Beck, Liang/Barsky)
- Define window-edge-coordinates of a point \( p = (x, y)^T \)
  - \( WEC_p = x_{\text{xmin}} \)
  - \( WEC_p = y_{\text{xmax}} \)
  - \( WEC_p = y_{\text{ymax}} - y \)  \( \text{Negative if outside!} \)

**Line Clipping**

**α-Clipping: algorithm**

```c
alphaClip(p1, p2, 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; // line parameter for first point \\
    \alpha_2 = 1; // line parameter for second point \\
    \ldots
}
```

**Line Clipping**

**α-Clipping: example for clipping \( p_1 \)**

- Start configuration
- After clipping to left
- After clipping to top
Line Clipping

\( \alpha \)-Clipping: algorithm (cont.)

...  
// now clip point \( p_2 \) against all edges  
if( OCl(\( p_2 \)) & LEFT_FLAG ) {  
  \( \alpha = WEC_0(p_2) / WEC_0(p_1) - WEC_0(p_2) \);  
  \( \alpha_2 = \min(\alpha, \alpha) \);  
}

Similarly clip \( p_1 \) against other edges  
...

Line Clipping

Example

Start configuration  
After clipping \( p_1 \)  
After clipping \( p_2 \)

Line Clipping

Another Example

Start configuration  
After clipping \( p_1 \)  
After clipping \( p_2 \)

Line Clipping

in 3D

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

**Non-convex clipping regions**
- Problem: arbitrary number of visible line segments
- Different approaches:
  - Break down polygon into convex parts
  - Scan convert for full window, and discard hidden pixels

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

Sutherland/Hodgeman Algorithm (’74)
- Approach: clip object polygon independently against all edges of subject polygon

Clipping against one edge:
clipPolygonToEdge( p[n], edge ) {
  for( i=0; i<n; i++ ) {
    if( p[i] inside edge ) { // 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…
  }
}

Clipping against one edge (cont)
- p[i] inside: 2 cases
  - inside
  - outside

p[0]

Output: p[i]
Output: p, p[i]

Clipping against one edge (cont)
- p[i] outside: 2 cases
  - inside
  - outside

Example
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.

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

**Other Polygon Clipping Algorithms**
- Weller/Aehlert `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

Visibility / Hidden Surface Removal
(Depth Test)

*CSC 314*
The Rendering Pipeline

- Geometry Database
- Model/View Transform
- Lighting
- Perspective Transform
- Clipping
- Scan Conversion
- Texturing
- Depth Test
- Blending
- Framebuffer

Occlusion

- For most interesting scenes, some polygons overlap

- To render the correct image, we need to determine which polygons occlude which

Painter's Algorithm

- Simple: render the polygons from back to front, “painting over” previous polygons

- Draw blue, then green, then orange

  will this work in the general case?

Painter's Algorithm: Problems

- Intersecting polygons present a problem
- Even non-intersecting polygons can form a cycle with no valid visibility order:

Analytic Visibility Algorithms

- Early visibility algorithms computed the set of visible polygon fragments directly, then rendered the fragments to a display:

Analytic Visibility Algorithms

- What is the minimum worst-case cost of computing the fragments for a scene composed of $n$ polygons?

  Answer: $O(n^2)$
Analytic Visibility Algorithms

- So, for about a decade (late 60s to late 70s) there was intense interest in finding efficient algorithms for hidden surface removal
- We’ll talk about one:
  - Binary Space Partition (BSP) Trees

Binary Space Partition Trees (1979)

**BSP Tree: partition space with binary tree of planes**

- Idea: divide space recursively into half-spaces by choosing splitting planes that separate objects in scene
- Preprocessing: create binary tree of planes
- Runtime: correctly traversing this tree enumerates objects from back to front

Creating BSP Trees: Objects

1. **Preprocessing**: Create a binary tree of planes.
2. **Runtime**: Correctly traversing this tree enumerates objects from back to front.
Creating BSP Trees: Objects

Splitting Objects
- No bunnies were harmed in previous example
- But what if a splitting plane passes through an object?
  - Split the object; give half to each node

Traversing BSP Trees
- Tree creation independent of viewpoint
  - Preprocessing step
- Tree traversal uses viewpoint
  - Runtime, happens for many different viewpoints
- Each plane divides world into near and far
  - For given viewpoint, decide which side is near and which is far
    - Check which side of plane viewpoint is on independently for each tree vertex
    - Tree traversal differs depending on viewpoint!
- Recursive algorithm
  - Recurse on far side
  - Draw object
  - Recurse on near side

Traversing BSP Trees
- renderBSP(BSPtree *T)
  - BSPtree *near, *far;
  - if (eye on left side of T->plane)
    - near = T->left; far = T->right;
  - else
    - near = T->right; far = T->left;
  - renderBSP(far);
  - if (T is a leaf node)
    - renderObject(T)
  - renderBSP(near);
BSP Trees: Viewpoint A

- decide independently at each tree vertex
- not just left or right child!
BSP Trees: Viewpoint B

BSP Tree Traversal: Polygons
- Split along the plane defined by any polygon from scene
- Classify all polygons into positive or negative half-space of the plane
  - If a polygon intersects plane, split polygon into two and classify them both
- Recurse down the negative half-space
- Recurse down the positive half-space

BSP Demo

Useful demo:
http://symbolcraft.com/graphics/bsp

Summary: BSP Trees
Pros:
- Simple, elegant scheme
- Correct version of painter’s algorithm back-to-front rendering approach
- Was very popular for video games (but getting less so)
Cons:
- Slow to construct tree: \( O(n \log n) \) to split, sort
- Splitting increases polygon count: \( O(n^2) \) worst-case
- Computationally intense preprocessing stage restricts algorithm to static scenes

The Z-Buffer Algorithm (mid-70’s)

History:
- BSP trees and Warnock’s algorithm were proposed when memory was expensive
- First 512x512 framebuffer was >$50,000!

Radical new approach: z-buffering
- The big idea:
  - Resolve visibility independently at each pixel

The Z-Buffer Algorithm
- We know how to rasterize polygons into an image discretized into pixels:
The Z-Buffer Algorithm

- What happens if multiple primitives occupy the same pixel on the screen?
- Which is allowed to paint the pixel?

The Z-Buffer Algorithm

Idea: retain depth after projection transform

- Each vertex maintains z coordinate
  - Relative to eye point
- Can do this with canonical viewing volumes

The Z-Buffer Algorithm

Augment color framebuffer with Z-buffer

- Also called depth buffer
- Stores z value at each pixel
- At frame beginning, initialize all pixel depths to \( \infty \)
- When rasterizing, (drawing pixels) interpolate depth (z) across polygon
- Check z-buffer before storing pixel color in framebuffer and storing depth in z-buffer
- Don’t write pixel if its z value is more distant than the z value already stored there

Interpolating Z

Edge walking

- Just interpolate Z along edges and across spans

Barycentric coordinates

- Interpolate z like other parameters
- E.g. color

Depth Test Precision

- Reminder: projective transformation maps eye-space z to generic z-range (NDC)
- Simple example:

\[
\begin{bmatrix}
\alpha \\
y \\
z \\
1
\end{bmatrix}
= \begin{bmatrix}
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & a & b \\
0 & 0 & -1 & 0
\end{bmatrix}
\begin{bmatrix}
x \\
y \\
z \\
1
\end{bmatrix}
\]

\[
z_{\text{NDC}} = a \cdot z_{\text{eye}} + b
\]

\[
z_{\text{eye}} = a + b
\]
Depth Test Precision

- Therefore, depth-buffer essentially stores 1/z, rather than z!
- Issue with integer depth buffers
  - High precision for near objects
  - Low precision for far objects

Z-Buffer Algorithm Questions

- How much memory does the Z-buffer use?
- Does the image rendered depend on the drawing order?
- Does the time to render the image depend on the drawing order?
- How does Z-buffer load scale with visible polygons with framebuffer resolution?

Z-Buffer Pros

- Simple!!!
- Easy to implement in hardware
  - Hardware support in all graphics cards today
- Polygons can be processed in arbitrary order
- Easily handles polygon interpenetration

Z-Buffer Cons

- Requires lots of memory
  - (e.g., 1280x1024x32 bits)
- Requires fast memory
  - Read-Modify-Write in inner loop
- Hard to simulate transparent polygons
  - We throw away color of polygons behind closest one
  - Works if polygons ordered back-to-front
  - Extra work throws away much of the speed advantage

Z-Buffer Cons

- Poor for scenes with high depth complexity
  - Need to render all polygons, even if most are invisible
- Shared edges are handled inconsistently
- Ordering dependent
**Hidden Surface Removal**

*Two kinds of visibility algorithms*
- Object space methods
- Image space methods

**Object Space Algorithms**

*Determine visibility on object or polygon level*
- Using camera coordinates

*Resolution independent*
- Explicitly compute visible portions of polygons

*Early in pipeline*
- After clipping

*Requires depth-sorting*
- Painter’s algorithm
- BSP trees

---

**Image Space Algorithms**

*Perform visibility test for in screen coordinates*
- Limited to resolution of display
- Z-buffer: check every pixel independently

*Performed late in rendering pipeline*

---

**Back-Face Culling**

*Not rendering back-facing polygons improves performance*
- By how much?
  - reduces by about half the number of polygons to be considered for each pixel
- Optimization when appropriate

---

**Back-Face Culling**

*Most objects in scene are typically “solid” rigorously: orientable closed manifolds*
- **Orientable**: must have two distinct sides
  - Cannot self-intersect
  - A sphere is orientable since has two sides, ‘inside’ and ‘outside’.
  - A Mobius strip or a Klein bottle is not orientable
- **Closed**: cannot “walk” from one side to the other
  - Sphere is closed manifold
  - Plane is not
**Back-Face Culling**

*Most objects in scene are typically “solid”*

Rigorously: orientable closed manifolds

- **Manifold**: local neighborhood of all points isomorphic to disc
- Boundary partitions space into interior & exterior

---

**Manifold**

*Examples of manifold objects:*

- Sphere
- Torus
- Well-formed CAD part

---

**Examples of non-manifold objects:**

- A single polygon
- A terrain or height field
- Polyhedron w/ missing face
- Anything with cracks or holes in boundary
- One-polygon thick lampshade

---

**Back-face Culling: VCS**

[first idea: cull if \( N_z < 0 \)]

sometimes misses polygons that should be culled

[better idea: cull if eye is below polygon plane]

---

**Back-face Culling: NDCS**

\[ \text{works to cull if } N_z > 0 \]

---

**The Rendering Pipeline**
Coming Up:

**Friday:**
- Scan Conversion
- A1 due (before class)

**Monday:**
- Texture Mapping