Clipping

Wolfgang Heidrich

Course News

Assignment 2
- Due March 2

Homework 4
- Out today

Reading (this week)
- Chapter 8

Reading (reading week)
- Rehearsal: Ch. 2, 3 (except 3.8), 6, 7, 8
- New: remainder of Ch. 2, Ch. 12
The Rendering Pipeline

Geometry Processing

Geometry Database → Model/View Transform. → Lighting → Perspective Transform. → Clipping

Scan Conversion → Texturing → Depth Test → Blending → Frame-buffer

Rasterization → Fragment Processing

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
Line Clipping

Outcodes (Cohen, Sutherland '74)
- 4 flags encoding position of a point relative to top, bottom, left, and right boundary
- E.g.:
  - OC(p1)=0010
  - OC(p2)=0000
  - OC(p3)=1001

<table>
<thead>
<tr>
<th>p1</th>
<th>1010</th>
<th>1000</th>
<th>1001</th>
</tr>
</thead>
<tbody>
<tr>
<td>p2</td>
<td>0010</td>
<td>0000</td>
<td>0001</td>
</tr>
<tr>
<td>p3</td>
<td>0110</td>
<td>0100</td>
<td>0101</td>
</tr>
</tbody>
</table>

Wolfgang Heidrich
Line Clipping

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

**Trivial cases:**
- \(OC(p1) == 0 \&\& OC(p2) == 0\)
  - Both points inside window, thus line segment completely visible (trivial accept)
- \((OC(p1) \&\& OC(p2)) !== 0\) (i.e. bitwise “and”)
  - 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**
- Handling of all the non-trivial cases
- Improvement of earlier algorithms (Cohen/Sutherland, Cyrus/Beck, Liang/Barsky)
- Define window-edge-coordinates of a point \( p = (x, y) \)
  - \( WEC_L(p) = x - x_{\min} \)
  - \( WEC_R(p) = x_{\max} - x \) **Negative if outside!**
  - \( WEC_B(p) = y - y_{\min} \)
  - \( WEC_T(p) = y_{\max} - y \)

**Line Clipping**

**α-Clipping**
- Line segment defined as: \( p_1 + \alpha(p_2 - p_1) \)
- Intersection point with one of the borders (say, left):
  \[
  x_1 + \alpha(x_2 - x_1) = x_{\min} \iff \\
  \alpha = \frac{x_{\min} - x_1}{x_2 - x_1} \\
  = \frac{x_{\min} - x_1}{WEC_L(x_1)} \\
  = \frac{WEC_L(x_2) - WEC_L(x_1)}{WEC_L(x_1) - WEC_L(x_2)}
  \]
Line Clipping

*α-Clipping: algorithm*

alphaClip( p1, p2, window ) {
    Determine window-edge-coordinates of p1, p2
    Determine outcodes OC(p1), OC(p2)

    Handle trivial accept and reject

    $\alpha_1 = 0; // line parameter for first point$
    $\alpha_2 = 1; // line parameter for second point$

    ...}

Line Clipping

*α-Clipping: algorithm (cont.)*

...  
// now clip point p1 against all edges
if( OC(p1) & LEFT_FLAG ) {
    $\alpha = \text{WEC}_L(p1) / (\text{WEC}_L(p1) - \text{WEC}_L(p2));$
    $\alpha_1 = \max(\alpha_1, \alpha);$
}

Similarly clip p1 against other edges

...
Line Clipping

**α-Clipping: example for clipping p1**

Start configuration | After clipping to left | After clipping to top

---

Line Clipping

**α-Clipping: algorithm (cont.)**

```cpp
... // now clip point p2 against all edges
if( OC(p2) & LEFT_FLAG ) {
    \[ \alpha = \frac{WEC_L(p2)}{WEC_L(p1) - WEC_L(p2)}; \]
    \[ \alpha_2 = \min(\alpha_2, \alpha); \]
}
```

Similarly clip p1 against other edges

...
\textbf{Line Clipping}

\textbf{\textalpha-Clipping: algorithm (cont.)}

...  
// wrap-up
if(\alpha_1 > \alpha_2 )
  no output;
else
  output line from $p_1 + \alpha_1 (p_2-p_1)$ to $p_1 + \alpha_2 (p_2-p_1)$
} // end of algorithm

\textbf{Line Clipping}

\textbf{Example}

\begin{itemize}
  \item \textbf{top}
  \item \textbf{left}
\end{itemize}

\begin{itemize}
  \item Start configuration
  \item After clipping $p_1$
  \item After clipping $p_2$
\end{itemize}

Wolfgang Heidrich
Line Clipping

Another Example

Start configuration  After clipping p1  After clipping p2

Line Clipping in 3D

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

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

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

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
  – Left, right, top, bottom, near, far planes
Polygon Clipping

Triangles Scan-Converted with Edge Equations:
- Go over each pixel in bounding rectangle
- Check if pixel is inside/outside of triangle

Triangle Clipping (w/ Edge Equation Scan Conversion)

**Note:**
- Once we use edge equations, we no longer really have to clip the geometry against window boundary!
- Instead: clip bounding rectangle against window
  - Only evaluate edge equations for pixels inside the window!
- Near/far clipping: when interpolating depth values, detect whether point is closer than near or farther than far
  - If so, don't draw it
General Polygon Clipping

**Task:**
- Clipping of general polygons
- Convex and concave
- Works with other scan conversion algorithms
  - *Independent of edge equations*

Polygon Clipping

*Not just clipping all boundary lines*
- May have to introduce new line segments
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/Hodgeman Algorithm (’74)**
- Approach: clip object polygon independently against all edges of subject polygon

```c
void 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...
    }
}
```
\textbf{Polygon Clipping}

\textit{Clipping against one edge (cont)}

- \(p[i]\) inside: 2 cases

\begin{tabular}{c|c|c|c}
inside & outside & inside & outside \\
\hline
\[ p[i-1] \] & & \[ p[i] \] & \[ p[i-1] \] \\
\hline
\[ p[i] \] & & \[ p[i] \] & \\
\hline
\end{tabular}

Output: \(p[i]\)  
Output: \(p, p[i]\)

Wolfgang Heidrich

\textbf{Polygon Clipping}

\textit{Clipping against one edge (cont)}

... 
else {  // \(p[i]\) is outside edge 
    if( \(p[i-1]\) inside edge ) { 
        \(p = \text{intersect}(p[i-1], p[i], \text{edge})\); 
        output \(p\); 
    } 
}  // end of algorithm

Wolfgang Heidrich
Polygon Clipping

Clipping against one edge (cont)

- p[i] outside: 2 cases

  inside  outside  inside  outside

  p[i-1]  p       p[i]  p[i-1]

  Output: p  Output: nothing

Polygon Clipping

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:
- Works for concave polygons
- But generates degenerate cases
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

**Sutherland/Hodgeman Algorithm**

- For Rendering Pipeline:
  - Re-triangulate resulting polygon (can be done for every individual clipping edge)
Polygon Clipping

Other Polygon Clipping Algorithms

- Weiler/Aetherton ’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

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 cyan, then green, then red

_**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.
Hidden Surface Removal

**Object Space Methods:**
- Work in 3D before scan conversion
  - *E.g. Painter’s algorithm*
- Usually independent of resolution
  - *Important to maintain independence of output device (screen/printer etc.)*

**Image Space Methods:**
- Work on per-pixel/per fragment basis after scan conversion
- Z-Buffer/Depth Buffer
- Much faster, but resolution dependent

---

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 scan converting: 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
Z-Buffer

**Store \((r,g,b,z)\) for each pixel**

- typically 8+8+8+24 bits, can be more
  ```
  for all i,j {
    Depth[i,j] = MAX_DEPTH
    Image[i,j] = BACKGROUND_COLOUR
  }
  for all polygons P {
    for all pixels in P {
      if (Z_pixel < Depth[i,j]) {
        Image[i,j] = C_pixel
        Depth[i,j] = Z_pixel
      }
    }
  }
  ```

Interpolating Z

**Edge walking**
- Just interpolate Z along edges and across spans

**Barycentric coordinates**
- Interpolate z like other parameters
- E.g. color
The Z-Buffer Algorithm (mid-70’s)

History:
- Object space algorithms were proposed when memory was expensive
- First 512x512 framebuffer was >$50,000!

**Radical new approach at the time**
- The big idea:
  - Resolve visibility independently at each pixel

Depth Test Precision

- Reminder: projective transformation maps eye-space z to generic z-range (NDC)
- Simple example:
  \[
  T \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & a & b \\ 0 & 0 & -1 & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}
  \]
- Thus:
  \[
  z_{NDC} = \frac{a \cdot z_{eye} + b}{z_{eye}} = a + \frac{b}{z_{eye}}
  \]
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

![Graph showing depth test precision](image)

Depth Test Precision

- Low precision can lead to depth fighting for far objects
  - Two different depths in eye space get mapped to same depth in framebuffer
  - Which object “wins” depends on drawing order and scan-conversion
- Gets worse for larger ratios $f:n$
  - Rule of thumb: $f:n < 1000$ for 24 bit depth buffer
- With 16 bits cannot discern millimeter differences in objects at 1 km distance
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

**Poor for scenes with high depth complexity**
- Need to render all polygons, even if most are invisible

**Shared edges are handled inconsistently**
- Ordering dependent

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
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

Object Space Visibility Algorithms

- Early visibility algorithms computed the set of visible *polygon fragments* directly, then rendered the fragments to a display.
Object Space Visibility Algorithms

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

Answer: $O(n^2)$

Object Space 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
  - Still in use today for ray-tracing, and in combination with z-buffer
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
Creating BSP Trees: Objects

Creating BSP Trees: Objects
Creating BSP Trees: Objects

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
BSP Trees: Viewpoint A

- decide independently at each tree vertex
- not just left or right child!
BSP Trees: Viewpoint A
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
- Still very popular for video games (but getting less so)

Cons:
- Slow(ish) 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

Coming Up:

After Reading Week
- More hidden surface removal
- Blending
- Texture mapping