Recall: Liang-Barsky 3D Clipping

- **Goal:** Clip object edge-by-edge against Canonical View volume (CVV)
- **Problem:**
  - 2 end-points of edge: \( A = (Ax, Ay, Az, Aw) \) and \( C = (Cx, Cy, Cz, Cw) \)
  - If edge intersects with CVV, compute intersection point \( I = (lx, ly, lz, lw) \)
Recall: Determining if point is inside CVV

Problem: Determine if point \((x,y,z)\) is inside or outside CVV?

Point \((x,y,z)\) is **inside CVV** if
\(-1 \leq x \leq 1\)
\(-1 \leq y \leq 1\)
\(-1 \leq z \leq 1\)
else point **is outside CVV**

CVV == 6 infinite planes \((x=\pm1; y=\pm1; z=\pm1)\)
Recall: Determining if point is inside CVV

- If point specified as \((x,y,z,w)\)
- Test \((x/w, y/w, z/w)\)!

Point \((x/w, y/w, z/w)\) is inside CVV

\[
\begin{align*}
\text{if} \quad & (-1 \leq x/w \leq 1) \\
\text{and} \quad & (-1 \leq y/w \leq 1) \\
\text{and} \quad & (-1 \leq z/w \leq 1)
\end{align*}
\]

else point is outside CVV
Recall: Modify Inside/Outside Tests Slightly

Our test: \((-1 < \frac{x}{w} < 1)\)

Point \((x,y,z,w)\) inside plane \(x = 1\) if

\[
\frac{x}{w} < 1 \\
\Rightarrow w - x > 0
\]

Point \((x,y,z,w)\) inside plane \(x = -1\) if

\[
-1 < \frac{x}{w} \\
\Rightarrow w + x > 0
\]
Recall: Numerical Example: Inside/Outside CVV Test

- Point \((x,y,z,w)\) is
  - inside plane \(x=-1\) if \(w+x > 0\)
  - inside plane \(x=1\) if \(w - x > 0\)

- Example Point \((0.5, 0.2, 0.7)\) inside planes \((x = -1,1)\) because -1 <= 0.5 <= 1

- If \(w = 10\), \((0.5, 0.2, 0.7) = (5, 2, 7, 10)\)
- Can either divide by \(w\) then test: \(-1 <= 5/10 <= 1\) OR
  - To test if inside \(x = -1\), \(w + x = 10 + 5 = 15 > 0\)
  - To test if inside \(x = 1\), \(w - x = 10 - 5 = 5 > 0\)
Recall: 3D Clipping

- Do same for y, z to form boundary coordinates for 6 planes as:

<table>
<thead>
<tr>
<th>Boundary coordinate (BC)</th>
<th>Homogenous coordinate</th>
<th>Clip plane</th>
<th>Example (5,2,7,10)</th>
</tr>
</thead>
<tbody>
<tr>
<td>BC0</td>
<td>w+x</td>
<td>x=-1</td>
<td>15</td>
</tr>
<tr>
<td>BC1</td>
<td>w-x</td>
<td>x=1</td>
<td>5</td>
</tr>
<tr>
<td>BC2</td>
<td>w+y</td>
<td>y=-1</td>
<td>12</td>
</tr>
<tr>
<td>BC3</td>
<td>w-y</td>
<td>y=1</td>
<td>8</td>
</tr>
<tr>
<td>BC4</td>
<td>w+z</td>
<td>z=-1</td>
<td>17</td>
</tr>
<tr>
<td>BC5</td>
<td>w-z</td>
<td>z=1</td>
<td>3</td>
</tr>
</tbody>
</table>

- Consider line that goes from point A to C
  - **Trivial accept:** 12 BCs (6 for pt. A, 6 for pt. C) > 0
  - **Trivial reject:** Both endpoints outside (-ve) for same plane
Edges as Parametric Equations

- Implicit form \( F(x, y) = 0 \)

- Parametric forms:
  - points specified based on single parameter value
  - Typical parameter: time \( t \)
    \[
    P(t) = P_0 + (P_1 - P_0) \times t \quad 0 \leq t \leq 1
    \]

- Some algorithms work in parametric form
  - Clipping: exclude line segment ranges
  - Animation: Interpolate between endpoints by varying \( t \)

- Represent each edge parametrically as \( A + (C - A)t \)
  - at time \( t=0 \), point at \( A \)
  - at time \( t=1 \), point at \( C \)
Inside/outside?

- Test A, C against 6 walls \((x=-1,1; \ y=-1,1; \ z=-1,1)\)
- There is an intersection if BCs have opposite signs. i.e. if either
  - A is outside \((<0)\), C is inside \((>0)\)  or
  - A inside \((>0)\), C outside \((<0)\)
- Edge intersects with plane at some \(t_{\text{hit}}\) between \([0,1]\)
Calculating hit time ($t_{hit}$)

- How to calculate $t_{hit}$?
- Represent an edge $t$ as:

$$Edge(t) = ((Ax + (Cx - Ax)t, (Ay + (Cy - Ay)t, (Az + (Cz - Az)t, (Aw + (Cw - Aw)t)$$

- E.g. If $x = 1$, 
  $$\frac{Ax + (Cx - Ax)t}{Aw + (Cw - Aw)t} = 1$$

- Solving for $t$ above, 
  $$t = \frac{Aw - Ax}{(Aw - Ax) - (Cw - Cx)}$$
Inside/outside?

- t_hit can be "entering (t_in)" or "leaving (t_out)"
- Define: "entering" if A outside, C inside
  - Why? As t goes [0-1], edge goes from outside (at A) to inside (at C)
- Define "leaving" if A inside, C outside
  - Why? As t goes [0-1], edge goes from inside (at A) to inside (at C)
Chop step by Step against 6 planes

- Initially
  
  ![Diagram showing initial state with t_in = 0, t_out = 1, Candidate Interval (CI) = [0 to 1]](image)

- Chop against each of 6 planes
  
  ![Diagram showing second step with t_in = 0, t_out = 0.74, Candidate Interval (CI) = [0 to 0.74]](image)

Why t_out?
Chop step by Step against 6 planes

- Initially
  
  t_out = 0.74

  t_in = 0,  t_out = 0.74  
  Candidate Interval (CI) = [0 to 0.74]

- Then
  
  Plane x = -1

  t_in = 0.36,  t_out = 0.74  
  Candidate Interval (CI) = [0.36 to 0.74]

Why t_in?
Candidate Interval

- Candidate Interval (CI): time interval during which edge might still be inside CVV. i.e. CI = t_in to t_out
- Initialize CI to [0,1]
- For each of 6 planes, calculate t_in or t_out, shrink CI

Conversely: values of t outside CI = edge is outside CVV
Shortening Candidate Interval

- **Algorithm:**
  - Test for trivial accept/reject (stop if either occurs)
  - Set CI to [0,1]
  - For each of 6 planes:
    - Find hit time \( t_{\text{hit}} \)
    - If \( t_{\text{in}} \), new \( t_{\text{in}} = \max(t_{\text{in}}, t_{\text{hit}}) \)
    - If \( t_{\text{out}} \), new \( t_{\text{out}} = \min(t_{\text{out}}, t_{\text{hit}}) \)
    - If \( t_{\text{in}} > t_{\text{out}} \) => exit (no valid intersections)

Note: seeking smallest valid CI without \( t_{\text{in}} \) crossing \( t_{\text{out}} \)
Calculate chopped A and C

- If valid $t_{in}$, $t_{out}$, calculate adjusted edge endpoints $A$, $C$ as

  - $A_{chop} = A + t_{in} \cdot (C - A)$ (calculate for $Ax,Ay,Az$)
  - $C_{chop} = A + t_{out} \cdot (C - A)$ (calculate for $Cx,Cy,Cz$)
3D Clipping Implementation

- Function clipEdge()
- Input: two points A and C (in homogenous coordinates)
- Output:
  - 0, if AC lies complete outside CVV
  - 1, complete inside CVV
  - Returns clipped A and C otherwise
- Calculate 6 BCs for A, 6 for C
Store BCs as Outcodes

- Use outcodes to track in/out
  - Number walls $x = +1, -1; y = +1, -1,$ and $z = +1, -1$ as 0 to 5
  - Bit $i$ of A’s outcode $= 1$ if A is outside $i$th wall
  - 1 otherwise
- Example: outcode for point outside walls 1, 2, 5

<table>
<thead>
<tr>
<th>Wall no.</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>OutCode</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
Trivial Accept/Reject using Outcodes

- **Trivial accept**: inside (not outside) any walls

  ![Table](image)

  Logical bitwise test: \( A \mid C = 0 \)

- **Trivial reject**: point outside **same** wall. Example Both A and C outside wall 1

  ![Table](image)

  Logical bitwise test: \( A \& C \neq 0 \)
3D Clipping Implementation

- Compute BCs for A,C store as outcodes
- Test A, C outcodes for trivial accept, trivial reject
- If not trivial accept/reject, for each wall:
  - Compute tHit
  - Update t_in, t_out
  - If t_in > t_out, early exit
int clipEdge(Point4& A, Point4& C)
{
    double tIn = 0.0, tOut = 1.0, tHit;
    double aBC[6], cBC[6];
    int aOutcode = 0, cOutcode = 0;

    .....find BCs for A and C
    .....form outcodes for A and C

    if((aOutCode & cOutcode) != 0) // trivial reject
        return 0;
    if((aOutCode | cOutcode) == 0) // trivial accept
        return 1;
3D Clipping Pseudocode

for(i=0;i<6;i++) // clip against each plane
{
    if(cBC[i] < 0) // C is outside wall i (exit so tOut)
    {
        tHit = aBC[i]/(aBC[i] – cBC[i]); // calculate tHit
        tOut = MIN(tOut, tHit);
    }
    else if(aBC[i] < 0) // A is outside wall I (enters so tIn)
    {
        tHit = aBC[i]/(aBC[i] – cBC[i]); // calculate tHit
        tIn = MAX(tIn, tHit);
    }
    if(tIn > tOut) return 0; // C is empty: early out
}
3D Clipping Pseudocode

Point4 tmp; // stores homogeneous coordinates
If(aOutcode != 0) // A is outside: tIn has changed. Calculate A_chop
{
    tmp.x = A.x + tIn * (C.x – A.x);
    // do same for y, z, and w components
}
If(cOutcode != 0) // C is outside: tOut has changed. Calculate C_chop
{
    C.x = A.x + tOut * (C.x – A.x);
    // do same for y, z and w components
}
A = tmp;
Return 1; // some of the edges lie inside CVV
Polygon Clipping

- Not as simple as line segment clipping
  - Clipping a line segment yields at most one line segment
  - Clipping a concave polygon can yield multiple polygons

- Clipping a convex polygon can yield at most one other polygon
Clipping Polygons

- Need more sophisticated algorithms to handle polygons:
  - **Sutherland-Hodgman**: clip any given polygon against a convex clip polygon (or window)
  - **Weiler-Atherton**: Both clipped polygon and clip polygon (or window) can be concave
Tessellation and Convexity

- One strategy is to replace nonconvex (*concave*) polygons with a set of triangular polygons (a *tessellation*)
- Also makes fill easier
Computer Graphics (CS 4731)
Lecture 23: Viewport Transformation & Hidden Surface Removal

Prof Emmanuel Agu

Computer Science Dept.
Worcester Polytechnic Institute (WPI)
Viewport Transformation

- After clipping, do viewport transformation

User implements in Vertex shader
Manufacturer implements in hardware
Viewport Transformation

- Maps \( CVV (x, y) \) \( \rightarrow \) screen \( (x, y) \) coordinates

\[
\begin{align*}
&\text{viewport} = \text{glViewport}(x, y, \text{width}, \text{height}) \\
&\text{screen coordinates} (x, y)
\end{align*}
\]
Viewport Transformation: What of $z$?

- Also maps $z$ (pseudo-depth) from $[-1,1]$ to $[0,1]$
- $[0,1]$ pseudo-depth stored in depth buffer,
  - Used for Depth testing (Hidden Surface Removal)
Hidden surface Removal

- Drawing polygonal faces on screen consumes CPU cycles
- User cannot see every surface in scene
- To save time, draw only surfaces we see
- Surfaces we cannot see and elimination methods?

1. Occluded surfaces: hidden surface removal (visibility)

2. Back faces: back face culling
Hidden surface Removal

- Surfaces we cannot see and elimination methods:
  - 3. Faces outside view volume: viewing frustrum culling

Classes of HSR techniques:
- Object space techniques: applied before rasterization
- Image space techniques: applied after vertices have been rasterized
Visibility (hidden surface removal)

- Overlapping opaque polygons
- **Correct visibility?** Draw only the closest polygon
  - (remove the other hidden surfaces)
Image Space Approach

- Start from pixel, work backwards into the scene
- Through each pixel, \((nm \text{ for an } n \times m \text{ frame buffer})\) find closest of \(k\) polygons
- Complexity \(O(nmk)\)
- Examples:
  - Ray tracing
  - z-buffer: OpenGL
OpenGL - Image Space Approach

- Paint pixel with color of **closest** object

```plaintext
for (each pixel in image) {
    determine the object closest to the pixel
draw the pixel using the object’s color
}
```
Z buffer Illustration

Correct Final image

Top View

eye

Z = 0.3

Z = 0.5
Step 1: Initialize the depth buffer

Largest possible z values is 1.0
Z buffer Illustration

**Step 2:** Draw blue polygon (actually order does not affect final result)

1. Determine group of pixels corresponding to blue polygon
2. Figure out z value of blue polygon for each covered pixel (0.5)
3. For each covered pixel, z = 0.5 is less than 1.0
   1. Smallest z so far = 0.5, color = blue
Z buffer Illustration

**Step 3:** Draw the yellow polygon

1. Determine group of pixels corresponding to yellow polygon
2. Figure out z value of yellow polygon for each covered pixel (0.3)
3. For each covered pixel, z = 0.3 becomes minimum, color = yellow

**z-buffer drawback:** wastes resources drawing and redrawing faces
OpenGL HSR Commands

- 3 main commands to do HSR
- `glutInitDisplayMode(GLUT_DEPTH | GLUT_RGB)` instructs OpenGL to create depth buffer
- `glEnable(GL_DEPTH_TEST)` enables depth testing
- `glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT)` initializes depth buffer every time we draw a new picture
Z-buffer Algorithm

- Initialize every pixel’s z value to 1.0
- Rasterize every polygon
- For each pixel in polygon, find its z value (interpolate)
- Track smallest z value so far through each pixel
- As we rasterize polygon, for each pixel in polygon
  - If polygon’s z through this pixel < current min z through pixel
  - Paint pixel with polygon’s color

Find depth (z) of every polygon at each pixel
Z (depth) Buffer Algorithm

For each polygon {
    for each pixel (x, y) in polygon area {
        if \( z_{\text{polygon\_pixel}}(x, y) < \text{depth\_buffer}(x, y) \) {
            \text{depth\_buffer}(x, y) = z_{\text{polygon\_pixel}}(x, y);
            \text{color\_buffer}(x, y) = \text{polygon color at } (x, y)
        }
    }
}

Note: know depths at vertices. Interpolate for interior \( z_{\text{polygon\_pixel}}(x, y) \) depths
Perspective Transformation: Z-Buffer Depth Compression

- **Pseudodepth calculation:** Recall we chose parameters (a and b) to map z from range \([\text{near}, \text{far}]\) to **pseudodepth** range \([-1, 1]\)

These values map z values of original view volume to \([-1, 1]\) range
Z-Buffer Depth Compression

- This mapping is almost linear close to eye
- Non-linear further from eye, approaches asymptote
- Also limited number of bits
- Thus, two z values close to far plane may map to same pseudodepth: Errors!!

\[ a = -\frac{F+N}{F-N} \]
\[ b = -\frac{-2FN}{F-N} \]
References