CS 548: COMPUTER GRAPHICS

CLIPPING

SPRING 2015

DR. MICHAEL J. REALE
CLIPPING DEFINED

• We now have our primitives (points, lines, polygons, etc.) in **normalized device coordinates**
  • In OpenGL → everything in unit cube from (-1,-1,-1) to (1,1,1) → clipping volume

• In terms of x and y → everything in rectangle from (-1,-1) to (1,1) → **clipping window**

• **Clipping** = removing part or all of the primitives outside of the clipping volume/window
  • May require intersection primitives with clipping volume/window boundaries
  • Converted to normalized device coordinates → more efficient to do clipping
NON-RECTANGULAR CLIPPING REGIONS

- You can have a non-rectangular clipping region
- However, it’s not very common, and most algorithms assume a rectangular clipping window/volume with coordinate values going from (-1, 1)
CLIPPING WINDOW VS. VIEWPORT

- **Viewport** = determines where on display window the clipping window will be displayed
  - Defined with starting point and width/height (in pixels)
  - May be (and most probably is) a different aspect ratio than clipping window
  - Can have multiple viewports
VIEWPORT IN OPENGL

To set the viewport in OpenGL:

void glViewport(
    GLint x,  // Starting x in pixels
    GLint y,  // Starting y in pixels
    GLsizei width,  // Width in pixels
    GLsizei height);  // Height in pixels
CLIPPING ALGORITHMS

• Clipping algorithms can be 2D or 3D
• The clipping algorithm also may clip:
  • Points
  • Lines
  • Fill-Areas (Polygons)
  • Curves
  • Text
• We will start with 2D clipping algorithms first.
2D CLIPPING ALGORITHMS: INTRODUCTION
POINT CLIPPING IN 2D

• Not much to say here; as long as the point coordinates are within the window, draw the point:

\[ x_{\text{min}} \leq x \leq x_{\text{max}} \]
\[ y_{\text{min}} \leq y \leq y_{\text{max}} \]

• Otherwise, do not draw the point

• Useful for particle systems
LINE CLIPPING: BRUTE FORCE

- Test endpoints p1 and p2 of line:
  - If both INSIDE all boundaries → line completely inside → keep line
  - If both OUTSIDE ONE of the boundaries (completely on one side) → line completely outside → discard line

- If the above tests fail, line MAY intersect one or more of the boundaries
  - Convert the line to parametric form:
    \[
    \begin{align*}
    x &= x_0 + t(x_{\text{end}} - x_0) \\
    y &= y_0 + t(y_{\text{end}} - y_0)
    \end{align*}
    \quad 0 \leq t \leq 1
    \]
  - Intersect with each boundary (x = -1, x = 1, y = -1, and y = 1)
    - If 0 <= t <= 1 for a given intersection → line segment intersects boundary
    - Check inside portion of line with rest of boundaries
  - No intersections with any boundaries → discard line
  - One to two intersections → form new line
  - Not very efficient → lots of line intersections
COHEN-SUTHERLAND LINE CLIPPING
INTRODUCTION

- Cohen-Sutherland line clipping
  - One of the earliest algorithms → variations still widely used
  - Does more tests before doing intersection calculations
  - Assumes rectangular clipping region

“Without the fun, none of us would go on.” – Ivan Sutherland
BASIC IDEA

• For each line endpoint → assign 4-bit **region code**
  • Each bit → refers to different window boundary/border
    • Most-significant to least-significant bits → Top, Bottom, Right, and Left boundaries
  • Bit = 1 → OUTSIDE of that window border
    • Example: Left = 1 → point is to the left (outside) of the left border
  • Bit = 0 → INSIDE or ON window border
    • Example: Left = 0 → point is to the right or on the left border
  • Region code → sometimes called “out code”
• Four window boundaries → creates nine regions
Let’s say we have a line endpoint \((x,y)\); we need to compute its region code.

For each bit of the region code, one can use inequalities \(\rightarrow\) e.g., if \(x < x_{\text{left}}\) then left border bit = 1.

**Faster alternative**: use bit-processing operations

1. **Step 1**: Get differences between \((x,y)\) and each window/clipping boundary
   - Left = \((x - x_{\text{min}})\)
   - Right = \((x_{\text{max}} - x)\)
   - Bottom = \((y - y_{\text{min}})\)
   - Top = \((y_{\text{max}} - y)\)

2. **Step 2**: Use sign bit of result to get bit value:
   - \(\text{int bit} = ((\text{unsigned int})\text{result}) >> (\text{sizeof(int)}*\text{CHAR_BIT} - 1))\)
     - \(\text{CHAR_BIT} = \) numbers of bits in a character (byte)
     - 0 = positive, 1 = negative
CHECKING ENDPOINTS

• If both endpoints have code 0000 → both inside clipping window → keep line

• If both endpoints have region code values with the same bit set to 1 → both points in the same “outside” region → outside of clipping window → discard line
  • Example: 1000 and 1010 → top boundary bit set → both points in top region (outside of window)

• Use logical OR and AND with region codes R₁ and R₂
  • If R₁ | R₂ == false → both must be 0000 → completely inside
  • If R₁ & R₂ == true → one bit must be shared → completely outside

• If neither of these cases are true, then we need to intersect the line with the boundaries
  • If corresponding bit value flips → must intersect boundary
    • Example: 1000 and 0010 → “top” and “right” bits flip → must intersect top and right boundaries
**GETTING INTERSECTION WITH BOUNDARIES**

- Let’s say we have a line with two endpoints \((x_0, y_0)\) and \((x_{\text{end}}, y_{\text{end}})\)
  - Slope \(m\) is given by:
    \[ m = \frac{y_{\text{end}} - y_0}{x_{\text{end}} - x_0} \]
  - We intersecting our line from \(p_1\) to \(p_2\) with one of the following:
    - \(x = x_{\text{min}}\)
    - \(x = x_{\text{max}}\)
    - \(y = y_{\text{min}}\)
    - \(y = y_{\text{max}}\)
  - For intersecting with **vertical** boundaries:
    \[ y = y_0 + m(x - x_0) \]
  - For intersecting with **horizontal** boundaries:
    \[ x = x_0 + \frac{y - y_0}{m} \]
FULL ALGORITHM

- Given a line with two endpoints p1 and p2
- While not done:
  - (Re)compute region codes for p1 and p2 → r1 and r2
  - *If both are in clipping window* → break out of loop and KEEP line from p1 to p2
  - *If both region codes share a bit that equals 1* → outside of window → break out of loop and DISCARD line
  - Otherwise:
    - If p1 is inside clipping window → swap p1 and p2 (also swap r1 and r2)
    - Compute slope m
    - Check p1 against each boundary: top, bottom, left, right → check if corresponding bit equals 1
      - If it is, compute intersection point → set it to be the new p1
One downside of Cohen-Sutherland line clipping is that it only works for rectangular clipping regions

- VERY common, but there might be cases where you want a non-rectangular clipping region
LIANG-BARSKY LINE CLIPPING
INTRODUCTION

- Liang-Barsky Line Clipping
  - Does even MORE testing before intersection calculations → faster
  - Uses parametric line equations
  - Can be used with non-rectangular clipping regions
PARAMETRIC LINES AND CLIPPING

• We can define a line in parametric form using a parameter $u$ as follows:

\[
\begin{align*}
  x &= x_0 + u \Delta x \\
  y &= y_0 + u \Delta y \quad 0 \leq u \leq 1
\end{align*}
\]

• Remember the point clipping conditions:

\[
\begin{align*}
  x_{\text{min}} &\leq x \leq x_{\text{max}} \\
  y_{\text{min}} &\leq y \leq y_{\text{max}}
\end{align*}
\]

• We combine these with the parametric line equations:

\[
\begin{align*}
  x_{\text{min}} &\leq x_0 + u \Delta x \leq x_{\text{max}} \\
  y_{\text{min}} &\leq y_0 + u \Delta y \leq y_{\text{max}}
\end{align*}
\]
TESTING CONDITIONS

• All our testing conditions have the form:

\[(u)p_k \leq q_k \quad k = 1, 2, 3, 4\]

• Where p and q are defined as follows:

\[
\begin{align*}
    p_1 &= -\Delta x \quad q_1 = x_0 - x_{\text{min}} \\
    p_2 &= \Delta x \quad q_2 = x_{\text{max}} - x_0 \\
    p_3 &= -\Delta y \quad q_3 = y_0 - y_{\text{min}} \\
    p_4 &= \Delta y \quad q_4 = y_{\text{max}} - y_0
\end{align*}
\]

\[
\begin{align*}
    -u\Delta x &\leq x_0 - x_{\text{min}} \implies x_{\text{min}} \leq x_0 + u\Delta x \\
    u\Delta x &\leq x_{\text{max}} - x_0 \implies x_0 + u\Delta x \leq x_{\text{max}} \\
    -u\Delta y &\leq y_0 - y_{\text{min}} \implies y_{\text{min}} \leq y_0 + u\Delta y \\
    u\Delta y &\leq y_{\text{max}} - y_0 \implies y_0 + u\Delta y \leq y_{\text{max}}
\end{align*}
\]
COMPUTING THE INTERSECTION POINT

• The line intersects a given boundary when the following is true:

\[(u)p_k = q_k\]

• Therefore, to get \(u\):

\[u = r_k = \frac{q_k}{p_k}\]

• This gives us our intersection point.

• If \(p_k = 0\), then the line is parallel to the boundary \(\rightarrow\) check value of \(q_k\):
  • If \(q_k < 0\) \(\rightarrow\) completely OUTSIDE boundary
  • Otherwise \(\rightarrow\) INSIDE or ON boundary

\[x_{\min} = x_0 + u\Delta x\]
\[x_0 + u\Delta x = x_{\max}\]
\[y_{\min} = y_0 + u\Delta y\]
\[y_0 + u\Delta y = y_{\max}\]
VALUE OF $P_k$

• If $p_k = 0$ → line is parallel to the boundary
• If $p_k < 0$ → line goes from OUTSIDE to INSIDE the boundary
• If $p_k > 0$ → line goes from INSIDE to OUTSIDE the boundary
FULL ALGORITHM

• Start with u₁ and u₂ (starting and ending parameter values of line)
  • u₁ = 0, u₂ = 1
  • For each clipping boundary (can implement as nested if statements):
    • Compute pₖ and qₖ values
    • If (pₖ != 0)
      • Compute rₖ
      • If (pₖ < 0) → OUTSIDE to INSIDE → u₁ = rₖ ONLY if line will be shorter
      • If (pₖ > 0) → INSIDE to OUTSIDE → u₂ = rₖ ONLY if line will be shorter
      • If at any point (u₁ > u₂) → REJECT LINE
    • Otherwise (pₖ = 0)
      • If (qₖ < 0) → line is parallel to AND outside boundary → REJECT LINE
      • If after all boundary checks line is NOT rejected → use values of u₁ and u₂ to compute line endpoints
LIANG-BARSKY VS. COHEN-SUTHERLAND

• Liang-Barsky $\rightarrow$ generally more efficient than Cohen-Sutherland
  • Only one divide per boundary check
  • Window intersections only computed once when final values of $u_1$ and $u_2$ are computed
    • Cohen-Sutherland $\rightarrow$ may repeatedly calculate intersections, even if line is completely outside clip window

• To extend to non-rectangular clipping regions $\rightarrow$ use parametric lines for boundaries
POLYGON FILL-AREA CLIPPING: INTRODUCTION
LINE CLIPPING VS. POLYGON CLIPPING

• Polygon clipping → cannot just use line clipping on the edges! → in general, does not produce closed polyline!
CLIPPING POLYGONS

• We can look at the vertices of the polygon:
  • If ALL inside clipping boundary → KEEP polygon
  • If ALL outside any ONE of the boundaries → DISCARD polygon

• Otherwise → locate polygon intersection positions with clipping boundaries
  • Convex → check each boundary → output new vertex list to next boundary check
  • Concave → must be able to output MULTIPLE vertex lists (polygon may break up into multiple polygons)
SUTHERLAND-HODGMAN POLYGON CLIPPING
INTRODUCTION

- Sutherland-Hodgman Polygon Clipping
  - By default, only handles convex polygons → only produces one list of vertices
    - Can be modified to do concave as well
BASIC IDEA

• Sends pairs of endpoints through a series of clippers: left, right, bottom, top
• For a given pair of endpoints v1 and v2, there are 4 possibilities with respect to a boundary:
  • Both v1 and v2 are INSIDE $\rightarrow$ output v2 only
  • v1 is INSIDE and v2 is OUTSIDE $\rightarrow$ intersect v1-v2 with boundary to get v2$'$ $\rightarrow$ output v2$'$
  • Both v1 and v2 are OUTSIDE $\rightarrow$ output NOTHING
  • v1 is OUTSIDE and v2 is INSIDE $\rightarrow$ intersect v1-v2 with boundary to get v1$'$ $\rightarrow$ output v1$'$ AND v2
<table>
<thead>
<tr>
<th>Input Edge:</th>
<th>Left Clipper</th>
<th>Right Clipper</th>
<th>Bottom Clipper</th>
<th>Top Clipper</th>
</tr>
</thead>
<tbody>
<tr>
<td>{1,2}</td>
<td>(in-in) → {2}</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>{2,3}</td>
<td>(in-out) → {2′}</td>
<td>{2,2′}: (in-in) → {2′}</td>
<td></td>
<td></td>
</tr>
<tr>
<td>{3,1}</td>
<td>(out-in) → {3′,1}</td>
<td>{2′,3′}: (in-in) → {3′}</td>
<td>{2′,3′}: (in-out) → {2″′}</td>
<td></td>
</tr>
<tr>
<td></td>
<td>{3′,1}: (in-in) → {1}</td>
<td>{3′,1}: (out-out) → {}</td>
<td></td>
<td></td>
</tr>
<tr>
<td>{1,2}: (in-in) → {2}</td>
<td>{1,2}: (out-in) → {1′,2}</td>
<td>{2″′,1′}: (in-in) → {1′}</td>
<td></td>
<td></td>
</tr>
<tr>
<td>{2,2′}: (in-in) → {2′}</td>
<td>{1′,2}: (in-in) → {2}</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>{2,2′}: (in-in) → {2′}</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>{2′,2″′}: (in-in) → {2″′}</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
PARALLEL EXECUTION

- As soon as one clipper outputs a pair of endpoints, it can pass it to the next clipper → allows parallel execution of clippers 😊
CONCAVE POLYGONS?

- Only one vertex list sent along → last vertex linked with last vertex
  - Can create extraneous lines if we try to process concave polygons

- **Alternatives:**
  - Split polygon into convex polygons
  - Check for multiple intersection points on each boundary → split lists (non-trivial to do this, however)
  - Use another algorithm
WEILER-ATHERTON POLYGON CLIPPING
INTRODUCTION

- **Weiler-Atherton Polygon Clipping**
  - Works with both convex and concave polygons
  - Also works with any shape clipping region
BASIC IDEA

• Trace around perimeter of polygon → look for borders that enclose clipped fill region
  • Follow (clockwise or counterclockwise) path around polygon
  • If we hit the clipping boundary → follow boundary until we hit polygon again
• Can get multiple fill regions → separate, unconnected polygons
WEILER-ATHERTON ALGORITHM

- Start at first polygon vertex and follow POLYGON boundary; output vertex list $P = {}$
- While any polygon vertices are “unprocessed”
  - If current vertex unprocessed AND (current vertex inside OR on clipping boundary) → add to $P$
  - Go to next vertex
  - If following POLYGON boundary:
    - If next vertex = previously processed vertex
      - Output $P$ as new polygon, and clear $P$ to start over → current vertex = last INSIDE-OUTSIDE intersection point
    - If next vertex = intersection point (INSIDE → OUTSIDE)
      - Start following CLIPPING boundary → current vertex = next vertex
    - Otherwise → current vertex = next vertex
  - If following CLIPPING boundary:
    - If next vertex = intersection point (OUTSIDE → INSIDE)
      - Start following POLYGON boundary
    - Current vertex = next vertex
Start at V1
Follow polygon to intersection point V1' -->
Switch to boundary
Remember to come back to V1' later
Follow boundary to B2
Follow boundary to intersection point V3' -->
Switch to polygon
Follow polygon to V1 -->
Already processed; output polygon

Output: \{V1, V1', B2, V3'\}

Start at V1'
Follow polygon to V2 --> outside
Follow polygon to V3 --> outside
Follow polygon to intersection point V3' -->
Already processed; nothing to output
Start at $V_1$
$V_1$ to $V_1'$ --> add $V_1$; follow boundary
$V_1'$ to $B_2$ --> add $V_1'$
$B_2$ to $V_6'$ --> add $B_2$; follow polygon
$V_6'$ to $V_1$ --> add $V_6'$; $V_1$ already processed

Polygon 1 = \{ $V_1$, $V_1'$, $B_2$, $V_6'$ \}

Resume at $V_1'$
$V_1'$ to $V_2$ --> $V_1'$ already processed
$V_2$ to $V_3$ --> $V_2$ outside
$V_3$ to $V_4$ --> $V_3$ outside
$V_4$ to $V_4'$ --> $V_4$ outside
$V_4'$ to $V_5$ --> add $V_4'$
$V_5$ to $V_5'$ --> add $V_5$; follow boundary
$V_5'$ to $V_4'$ --> add $V_5'$; $V_4'$ already processed

Polygon 2 = \{ $V_4'$, $V_5$, $V_5'$ \}

Resume at $V_5'$
$V_5'$ to $V_6$ --> $V_5'$ already processed
$V_6$ to $V_6'$ --> $V_6$ outside; $V_6'$ already processed

No output polygon
3D CLIPPING ALGORITHMS
INTRODUCTION

• With 2D clipping, we had 2D boundaries → i.e., lines

• With 3D clipping, we have 3D boundaries → i.e., planes
  • Assuming we have everything in normalized device coordinates, our boundary planes are:

\[
\begin{align*}
x_{\text{min}} &= -1 & x_{\text{max}} &= 1 \\
y_{\text{min}} &= -1 & y_{\text{max}} &= 1 \\
z_{\text{min}} &= -1 & z_{\text{max}} &= 1
\end{align*}
\]

• Moreover, a lot of our 2D clipping algorithms are extendable to 3D
3D REGION CODES
(EXTENDING COHEN-SUTHERLAND)

• In 3D, we use a **6-digit region code**: Far, Near, Top, Bottom, Right, Left
• Bit values:

<table>
<thead>
<tr>
<th>Bit 6</th>
<th>Bit 5</th>
<th>Bit 4</th>
<th>Bit 3</th>
<th>Bit 2</th>
<th>Bit 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Far</td>
<td>Near</td>
<td>Top</td>
<td>Bottom</td>
<td>Right</td>
<td>Left</td>
</tr>
</tbody>
</table>

- bit 1 = 1 if \( (1 + x) < 0 \) (Left)
- bit 2 = 1 if \( (1 - x) < 0 \) (Right)
- bit 3 = 1 if \( (1 + y) < 0 \) (Bottom)
- bit 4 = 1 if \( (1 - y) < 0 \) (Top)
- bit 5 = 1 if \( (1 + z) < 0 \) (Near)
- bit 6 = 1 if \( (1 - z) < 0 \) (Far)
3D POINT AND LINE CLIPPING

- Point clipping $\rightarrow$ keep only if region code = 000000

- Line clipping $\rightarrow$ similar to Cohen-Sutherland
  - Both 000000 $\rightarrow$ keep whole line
  - Both share a 1 bit $\rightarrow$ reject whole line
  - Otherwise $\rightarrow$ need to compute intersection points
3D LINE SEGMENTS

- Given two line endpoints \( P_1 \) and \( P_2 \), can use parametric form of line:

\[
P = P_1 + (P_2 - P_1)u \quad 0 \leq u \leq 1
\]

- \( u = 0 \rightarrow P_1 \)
- \( u = 1 \rightarrow P_2 \)

- Explicitly, this forms three equations:

\[
\begin{align*}
x &= x_1 + (x_2 - x_1)u \\
y &= y_1 + (y_2 - y_1)u \quad 0 \leq u \leq 1 \\
z &= z_1 + (z_2 - z_1)u
\end{align*}
\]
3D INTERSECTION POINTS

• To get the intersection with each plane, we solve for $u$ given the line equation and the plane equation.

• Example: intersect line with $x_{\text{max}} = 1$:

  
  \begin{align*}
  1 &= x_1 + (x_2 - x_1)u \\
  u &= \frac{1}{x_1 + (x_2 - x_1)}
  \end{align*}

• If $0 \leq u \leq 1$, then line segment intersects boundary plane.

• If corresponding $y$ and $z$ coordinates equal $\pm 1$ $\rightarrow$ intersection point inside view volume.

• Can progressively cut off parts of line outside boundaries and recompute region codes.
DEALING WITH 3D OBJECTS

• Can test polyhedron for trivial acceptance or rejection:
  • Look at vertices
  • Look at bounding sphere
  • Etc.
• Otherwise, more complicated
  • One approach:
    • Divide surface into triangle strip
    • Clip triangles using Sutherland-Hodgman on each of the six clipping planes
    • Get output vertices of final strip
ARBITRARY CLIPPING PLANES

• Might want to:
  • Isolate / clip off irregularly shaped object
  • Eliminate part of scene (special effect)
  • Slice off section of object (show interior)

• Specify clipping boundary using full plane equation: \(Ax + By + Cz + D < 0\)
  • Anything satisfying that equation \(\rightarrow\) clipped from scene

• Testing line segment:
  • Test endpoints (make sure not completely in front of or behind plane)
  • Otherwise, calculate intersection point:

\[
P = P_1 + (P_2 - P_1)u \quad 0 \leq u \leq 1
\]

• Testing a solid object \(\rightarrow\) intersect each polygon face