Today

- Clipping
  - Application: scan-line conversion
  - Clipping to view volume
  - Clipping arbitrary points, lines, polygons, volumes
  - Hidden surface removal

Clipping

- Want to avoid drawing primitives outside area of interest
  - Too close: obscures view
  - Behind camera: obscures view
Clipping

• Stuff outside view volume should not be drawn

- Too close: obscures view
- Too far:
  - Complexity
  - Z-buffer problems
- Too high/low/right/left:
  - Memory errors
  - Broken algorithms
  - Complexity

3D Rendering Pipeline
3D Rendering Pipeline

- 3D Primitives
  - Modeling Transformation
  - Lighting
  - Viewing Transformation
  - Clipping
  - 2D Screen Coordinates
- Image
  - 2D Image Coordinates

2D Rendering Pipeline

- Clipping
  - Clip portions of geometric primitives residing outside the window
- Viewport Transformation
  - Transform the clipped primitives from screen to image coordinates
- Scan Conversion
  - Fill pixels representing primitives in screen coordinates
- Image
Clipping

• Avoid drawing parts of primitives outside window
  • Window defines part of scene being viewed
  • Must draw geometric primitives only inside window
Point Clipping

- Is point \((x,y)\) inside the clip window?

\[
\text{inside} = (x \geq wx1) \land (x \leq wx2) \land (y \geq wy1) \land (y \leq wy2);
\]

Line Clipping

- Find the part of a line inside the clip window

Before Clipping

After Clipping
Cohen Sutherland Line Clipping

• Use simple tests to classify easy cases first

Cohen Sutherland Line Clipping

• Classify some lines quickly by AND of bit codes representing regions of two endpoints (must be 0)

Cohen Sutherland Line Clipping

• Classify some lines quickly by AND of bit codes representing regions of two endpoints (must be 0)
Cohen Sutherland Line Clipping

- Classify some lines quickly by AND of bit codes representing regions of two endpoints (must be 0)

<table>
<thead>
<tr>
<th>Bit 1</th>
<th>Bit 2</th>
<th>Bit 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1001</td>
<td>0001</td>
<td>0101</td>
</tr>
<tr>
<td>1000</td>
<td>0000</td>
<td>0100</td>
</tr>
<tr>
<td>1010</td>
<td>0010</td>
<td>0110</td>
</tr>
</tbody>
</table>

- Compute intersections with window boundary for lines that can’t be classified quickly

<table>
<thead>
<tr>
<th>Bit 1</th>
<th>Bit 2</th>
<th>Bit 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1001</td>
<td>0001</td>
<td>0101</td>
</tr>
<tr>
<td>1000</td>
<td>0000</td>
<td>0100</td>
</tr>
<tr>
<td>1010</td>
<td>0010</td>
<td>0110</td>
</tr>
</tbody>
</table>
Cohen-Sutherland Line Clipping

- Compute intersections with window boundary for lines that can't be classified quickly

```
Bit 1  | 0000  | 0010  | 0110  | 1010
---    |-------|-------|-------|-------
Bit 2  | 0000  | 0010  | 0110  | 1010
---    |-------|-------|-------|-------
```

```
P_5   P_6   P_8   P_10
---   ---   ---   ---
0001  0100  0110  1001
---   ---   ---   ---
```
Cohen-Sutherland Line Clipping

- Compute intersections with window boundary for lines that can't be classified quickly

<table>
<thead>
<tr>
<th>Bit 1</th>
<th>Bit 2</th>
<th>Bit 3</th>
<th>Bit 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0110</td>
<td>0010</td>
<td>P9</td>
<td>P3</td>
</tr>
<tr>
<td>0101</td>
<td>0000</td>
<td>P5</td>
<td>P6</td>
</tr>
<tr>
<td>1001</td>
<td>0110</td>
<td>P8</td>
<td>P4</td>
</tr>
<tr>
<td>1010</td>
<td>0001</td>
<td>P10</td>
<td>P0</td>
</tr>
</tbody>
</table>

Cohen-Sutherland Line Clipping

- Compute intersections with window boundary for lines that can't be classified quickly

<table>
<thead>
<tr>
<th>Bit 1</th>
<th>Bit 2</th>
<th>Bit 3</th>
<th>Bit 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0110</td>
<td>0010</td>
<td>P9</td>
<td>P3</td>
</tr>
<tr>
<td>0101</td>
<td>0000</td>
<td>P5</td>
<td>P6</td>
</tr>
<tr>
<td>1001</td>
<td>0110</td>
<td>P8</td>
<td>P4</td>
</tr>
<tr>
<td>1010</td>
<td>0001</td>
<td>P10</td>
<td>P0</td>
</tr>
</tbody>
</table>
Cohen-Sutherland Line Clipping

- Compute intersections with window boundary for lines that can't be classified quickly

Bit 1  Bit 2  Bit 3  Bit 4

1001  0001  P'7  0101

1000  0000  P3  0100  P8

1010  0010  P6  0110

1010  0100  P5  0110

Cohen-Sutherland Line Clipping

- Compute intersections with window boundary for lines that can't be classified quickly

Bit 1  Bit 2  Bit 3  Bit 4

1001  0001  P'7  0101

1000  0000  P3  0100  P8

1010  0010  P6  0110

1010  0100  P5  0110

Cohen-Sutherland Line Clipping

- Compute intersections with window boundary for lines that can't be classified quickly

Bit 1  Bit 2  Bit 3  Bit 4

1001  0001  P'7  0101

1000  0000  P3  0100  P8

1010  0010  P6  0110

1010  0100  P5  0110
Cohen-Sutherland Line Clipping

• Compute intersections with window boundary for lines that can’t be classified quickly

Bit 1 | Bit 2
--- | ---
1001 | 0001
1000 | 0010
1010 | 0100

P'8

P7

P4

P3

P9

P5

P6

P10

Bit 3

Bit 4
Cohen-Sutherland Line Clipping

- Compute intersections with window boundary for lines that can't be classified quickly

```
1. Define constants for each side of clipping rectangle:
   - const int LEFT = 1; // 0001
   - const int RIGHT = 2; // 0010
   - const int BOTTOM = 4; // 0100
   - const int TOP = 8; // 1000

2. Use a simple function to test a single point's location
   - if (x < xmin)    // to the left of clip window
     code = LEFT;
   else if (x > xmax) // to the right of clip window
     code = RIGHT;
   if (y < ymin)    // below the clip window
     code |= BOTTOM;
   else if (y > ymax) // above the clip window
     code |= TOP;
   return code;
```

Cohen-Sutherland Line Clipping

3. For a line with endpoints \((x_0, y_0)\) and \((x_1, y_1)\):
   a) Get position codes for each endpoint, compare them using | (OR)
      - If neither code contains any 1's, draw the entire line
   b) Compare using & (AND)
      - If true, then both points are outside the box. Reject entire line
   c) Otherwise, one point is outside the rectangle, so find the intersection
      i) Choose a point outside the viewing rectangle \((x_t, y_t)\)
      ii) Find the closest intersection with the rectangle, relative to that pt.
         Use \(y = y_t + \text{slope} \times (x - x_t)\), \(x = x_t + 1/\text{slope} \times (y - y_t)\)
         Set \(x\) or \(y\) in the above equation depending on location of point:
         Top: \(y = \text{ymax}\)   Bottom: \(y = \text{ymin}\)
         Left: \(x = \text{xmin}\)   Right: \(x = \text{xmax}\)
   iv. Reset old point to new location
   v. Recursively pass new line to algorithm
Cohen-Sutherland Line Clipping

iv. Reset old point to new location
v. Recursively pass new line to algorithm

\[(x_0, y_0)\]
\[(x_1, y_1)\]
\[y_{\text{max}}\]
\[y_{\text{min}}\]
\[x_{\text{min}}\]
\[x_{\text{max}}\]

Liang-Barsky Clipping

• Save time by using parametric equation of line from the start to find its intersections with the window
• Calculations used to determine in/out state of points also useful for finding intersections

\[P_0\]
\[P_1\]

Liang-Barsky Clipping

• Parametric equation \( P(t) = P_0 + t(P_1 - P_0) \)
• Want to find the values of \( t \) on the edges of the rectangle
• Initial \( t_0 = 0 \) and \( t_1 = 1 \). We'll change them as we find for intersections
• Check one side at a time
Liang-Barsky Clipping

- Example: Start with left side
  \[ p = -\text{deltaX} \]
  \[ q = -(\text{xmin} - x_0) \]
  \[ r = q/p \]
  Intuition: Value of \( t \) when the line crosses left side

- Example: Check left side
  \[ p = -\text{deltaX} \]
  \[ q = -(\text{xmin} - x_0) \]
  \[ r = q/p \]
  If \( p = 0 \) && \( q < 0 \): Line is vertical and to the left of the rectangle. Break and don’t draw.
  If \( p < 0 \): If \( r > t_1 \) Line is outside 1
  else if \( r > t_0 \) \( t_0 = r \) 2
  If \( p > 0 \): If \( r < t_0 \) Line is outside 2
  else if \( r < t_1 \) \( t_1 = r \) 4
Liang-Barsky Clipping

- Example: Check left side
  \[ p = -\delta X \]
  \[ q = -(x_{min} - x_0) \]
  \[ r = q/p \]

  If \( p < 0 \):
  If \( r > t_1 \) Line is outside
  \[ t_0 = r \]
  else if \( (r > t_0) \) \( t_0 = r \)

  If \( p > 0 \):
  If \( r < t_0 \) Line is outside
  \[ t_0 = r \]
  else if \( (r < t_0) \) \( t_0 = r \)

- Redefine \( P_0 \) and \( P_1 \) using new values for \( t_0 \) and \( t_1 \)
  \[ P_0 = t_0(P_1 - P_0) \]
  \[ P_0 = t_1(P_1 - P_0) \]

- Proceed to next side (bottom, then right, then top)

Liang-Barsky Clipping

- Variables for each side:
  - Left: \( p = -\delta X \), \( q = -(x_{min} - x_0) \)
  - Right: \( p = \delta X \), \( q = (x_{max} - x_0) \)
  - Top: \( p = \delta Y \), \( q = (y_{max} - y_0) \)
  - Bottom: \( p = -\delta Y \), \( q = -(y_{min} - y_0) \)
Clipping Line to Arbitrary Line/Plane

Line segment to be clipped
\[ x(t) = a + t(b - a) \]

Line/plane that clips it
\[ \hat{n} \cdot x - \hat{n} \cdot r = 0 \]

Clipping Line to Line/Plane

Line segment to be clipped
\[ x(t) = a + t(b - a) \]

Line/plane that clips it
\[ \hat{n} \cdot x - f = 0 \]
\[ \hat{n} \cdot (a + t(b - a)) - f = 0 \]
\[ \hat{n} \cdot a + t(\hat{n} \cdot (b - a)) - f = 0 \]

Clipping Line to Line/Plane

• Segment may be on one side
  \[ t \notin [0 \ldots 1] \]
• Lines may be parallel
  \[ \hat{n} \cdot d = 0 \]
  \[ |\hat{n} \cdot d| \leq \epsilon \] (Recall comments about numerical issues)
Recap: Line Clipping

- Sutherland-Hodgman algorithm
  - Basically edge walking
- Clipping done often... should be efficient
  - Liang-Barsky parametric space algorithm
  - Use dot-product trick for finding intersections with arbitrary lines/planes
  - See text for clipping in 4D homogenized coordinates

Polygon Clipping

- Avoid drawing parts of primitives outside window
  - Points
  - Lines
  - Polygons
  - etc.

Polygon Clipping

- Find the part of a polygon inside the clip window?
Polygon Clipping

- Find the part of a polygon inside the clip window?

Sutherland-Hodgman Clipping

- Clip to each window boundary one at a time
Sutherland-Hodgman Clipping

• Clip to each window boundary one at a time
Triangle Clip/Split

- One clipping plane at a time
- Want to remove triangle, keep triangle, or clip and produce smaller triangles
- Depends on number of vertices outside clipping plane:
  - 0: Keep triangle
  - 3: Remove triangle
  - 2: Replace with ONE triangle
  - 1: Replace with TWO triangles

Double vertices if you want separation...

Polygon Clipping

<table>
<thead>
<tr>
<th>Inside</th>
<th>Outside</th>
<th>Inside</th>
<th>Outside</th>
<th>Inside</th>
<th>Outside</th>
<th>Inside</th>
<th>Outside</th>
</tr>
</thead>
<tbody>
<tr>
<td>s</td>
<td>p</td>
<td>s</td>
<td>p</td>
<td>p</td>
<td>i</td>
<td>s</td>
<td>p</td>
</tr>
</tbody>
</table>

Output p | Output i | No output | Output i and p
Clipping to a Boundary

• Do inside test for each point in sequence, Insert new points when cross window boundary, Remove points outside window boundary

Clipping to a Boundary

• Do inside test for each point in sequence, Insert new points when cross window boundary, Remove points outside window boundary

Clipping to a Boundary

• Do inside test for each point in sequence, Insert new points when cross window boundary, Remove points outside window boundary
Clipping to a Boundary

- Do inside test for each point in sequence, insert new points when cross window boundary, remove points outside window boundary.

Outside

Inside

Window Boundary

$P_1$ $P_2$ $P_3$ $P_4$ $P_5$
Clipping to a Boundary

- Do inside test for each point in sequence, insert new points when cross window boundary, remove points outside window boundary.
Polygon Clip to Convex Domain

• Convex domain defined by collection of planes (or lines or hyper-planes)
• Planes have outward pointing normals
• Clip against each plane in turn
• Check for early/trivial rejection
General Polygon Clipping

\( A - B \)
\( B - A \)
\( A \cup B \)
\( A \cap B \)

- Weiler-Atherton Algorithm
  - Double edges

- Solid = inside
- Dashed = outside
General Polygon Clipping

- Weiler-Atherton Algorithm

- New borders determine ownership
  - Replace one polygon with a clipping plane to separates the other one into inside and outside portions

Pros: Allows for shadow calculations with off-screen polygons

Cons: Slower than Sutherland-Hodgman

Hidden Surface Removal

- True 3D to 2D projection would put every thing overlapping into the view plane.
- We need to determine what’s in front and display only that.
Z-Buffers

• Add extra depth channel to image
• Write Z values when writing pixels
• Test Z values before writing

Images from Chandelle

Z-Buffers

• Benefits
  • Easy to implement
  • Works for most any geometric primitive
  • Parallel operation in hardware

• Limitations
  • Quantization and aliasing artifacts
  • Overfill
  • Transparency does not work well

Z-Buffers

• Transparency requires partial sorting:

<table>
<thead>
<tr>
<th>1st</th>
<th>2nd</th>
<th>3rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>Opaque</td>
<td>Opaque</td>
<td>Partially transparent</td>
</tr>
</tbody>
</table>

Front

Good

<table>
<thead>
<tr>
<th>1st</th>
<th>3rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>Partially transparent</td>
<td>Opaque</td>
</tr>
</tbody>
</table>

Not Good
A-Buffers

- Store sorted list of “fragments” at each pixel
- Draw all opaque stuff first then transparent
- Stuff behind full opacity gets ignored
- Nice for antialiasing...

Scan-line Algorithm

- Assume polygons don’t intersect
- Each time an edge is crossed determine who’s on top

Painters Algorithm

- Sort Polygons Front-to-Back
  - Draw in order
  - Back-to-Front works also, but wasteful
- How to sort quickly?
- Intersecting polygons?
- Cycles?
### BSP-Trees

- **Binary Space Partition Trees**
  - Split space along planes
  - Allows fast queries of some spatial relations

- **Draw Front-to-Back**
  - Draw same-side polygons first
  - Draw root node polygon (if any)
  - Draw other-side polygons last