Clipping and Scan Conversion

Michael Kazhdan

(600.357 / 600.457)

HB Ch. 3.2, 3.11, 6.7, 6.8
FvDFH Ch. 3.2, 3.6, 3.12, 3.14
Announcements

The first midterm will be on October 21st.
3D Rendering Pipeline (for direct illumination)

- 3D Primitives
  - 3D Modeling Coordinates
  - Modeling Transformation
    - 3D World Coordinates
    - Camera Transformation
      - 3D Camera Coordinates
      - Lighting
        - 3D Camera Coordinates
        - Projection Transformation
          - 2D Screen Coordinates
          - Clipping
            - 2D Screen Coordinates
            - Viewport Transformation
              - 2D Screen Coordinates
              - Scan Conversion
                - 2D Image Coordinates
                - Image
                  - 2D Image Coordinates
                  - 3D Model
                    - 3D Model
                      - 2D Image
Transformations

\((x, y, z)\)

- **Modeling Transformation**
  - 3D Object Coordinates

  \((x, y, z)\)

- **Camera Transformation**
  - 3D World Coordinates

  \((x, y, z)\)

- **Projection Transformation**
  - 3D Camera Coordinates

  \((x', y', z')\)

- **Window-to-Viewport Transformation**
  - 2D Screen Coordinates

  \((x', y')\)

  - 2D Image Coordinates

  \((x', y')\)
Transformations

\[(x, y, z)\]

Modeling Transformation

3D Object Coordinates

Camera Transformation

3D World Coordinates

Projection Transformation

3D Camera Coordinates

Window-to-Viewport Transformation

2D Screen Coordinates

\[(x', y')\]

Transform = \( M \)

\( M = \) local to world transform
Transformations

\((x, y, z)\)

Modeling Transformation

3D Object Coordinates

Camera Transformation

3D World Coordinates

Projection Transformation

3D Camera Coordinates

Window-to-Viewport Transformation

2D Screen Coordinates

2D Image Coordinates

\((x', y')\)

\[\text{Transform} = M\]

\[M = \text{local to world transform}\]
Transformations

3D Object Coordinates

Modeling Transformation

3D World Coordinates

Camera Transformation

3D Camera Coordinates

Projection Transformation

2D Screen Coordinates

Window-to-Viewport Transformation

2D Image Coordinates

\[(x, y, z)\]

Transform = \(C^{-1}M\)

\[
C = \begin{pmatrix}
R_x & U_x & B_x & E_x \\
R_y & U_y & B_y & E_y \\
R_z & U_z & B_z & E_z \\
0 & 0 & 0 & 1
\end{pmatrix}
\]
Transformations

\[ (x, y, z) \]

Modeling Transformation

3D Object Coordinates

Camera Transformation

3D World Coordinates

Camera Transformation

3D Camera Coordinates

Projection Transformation

3D Screen Coordinates

Window-to-Viewport Transformation

2D Screen Coordinates

2D Image Coordinates

\[ (x', y') \]

\[ \text{Transform} = PC^{-1}M \]

\[ P = \text{projection transform} \]

\[ P_o = \begin{bmatrix} 1 & 0 & L \cos \phi & 0 \\ 0 & 1 & L \sin \phi & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \]

\[ P_p = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \]
Transformations

\[(x, y, z)\]

- **Modeling Transformation**
  - 3D Object Coordinates
  - 3D World Coordinates

- **Camera Transformation**
  - 3D Camera Coordinates

- **Projection Transformation**
  - 2D Screen Coordinates

- **Window-to-Viewport Transformation**
  - 2D Image Coordinates

\[(x', y')\]

\[\text{Transform} = V P C^{-1} M\]

\[V = \text{viewport transform}\]
Transformations

\[ (x, y, z) \]

3D Object Coordinates

Modeling Transformation

\[ (x, y, z) \]

3D World Coordinates

Camera Transformation

\[ (w_x, w_y) \]

3D Camera Coordinates

Projection Transformation

\[ (v_x, v_y) \]

Screen Coordinates

Window-to-Viewport Transformation

\[ (x', y') \]

2D Image Coordinates

Window

Viewport

\[ \text{Transform} = VPC^{-1}M \]

\[ V = \begin{bmatrix}
1 & 0 & v_x^1 \\
0 & 1 & v_x^2 \\
0 & 0 & 1
\end{bmatrix} \begin{bmatrix}
w_x^2 - w_x^1 \\
0 \\
0
\end{bmatrix} \begin{bmatrix}
0 & 0 & 0 \\
0 & 0 & 0 \\
1 & 0 & -w_x^1
\end{bmatrix} \begin{bmatrix}
v_x^2 \\
v_y^2 \\
1
\end{bmatrix} \begin{bmatrix}
w_x^2 - w_x^1 \\
v_y^2 - w_y^1 \\
0
\end{bmatrix} \begin{bmatrix}
0 & 1 & -w_y^1 \\
0 & 0 & 1
\end{bmatrix} \begin{bmatrix}
\]

\[ (x', y') \]
3D Rendering Pipeline (for direct illumination)

\[(x, y, z)\]  
\[\rightarrow\] 3D Object Coordinates  
Modeling Transformation  
\[\rightarrow\] 3D World Coordinates  
Camera Transformation  
\[\rightarrow\] 3D Camera Coordinates  
Projection Transformation  
\[\rightarrow\] 2D Screen Coordinates  
Window-to-Viewport Transformation  
\[\rightarrow\] 2D Image Coordinates  
\[(x', y')\]  
3D Model  
2D Viewport  
2D Screen
3D Rendering Pipeline (for direct illumination)

3D Primitives

3D Modeling Coordinates

Modeling Transformation

3D World Coordinates

Camera Transformation

3D Camera Coordinates

Lighting

3D Camera Coordinates

Projection Transformation

2D Screen Coordinates

Clipping

2D Screen Coordinates

Viewport Transformation

2D Image Coordinates

Scan Conversion

2D Image Coordinates

Image
Clipping

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

• Avoid drawing parts of primitives outside window
  ◦ Points
  ◦ Line Segments
  ◦ Polygons
Point Clipping

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

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

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

\((x, y)\)
Clipping

- Avoid drawing parts of primitives outside window
  - Points
  - **Line Segments**
  - Polygons
Line Segment Clipping

• Find the part of a line inside the clip window
Line Segment Clipping

- Find the part of a line inside the clip window

After Clipping
Cohen-Sutherland Line Clipping

- Use simple tests to classify easy cases first
Cohen-Sutherland Line Clipping

- If both outcodes are 0, line segment is inside
- If AND of outcodes not 0, line segment is outside
- Otherwise clip and test
Cohen-Sutherland Line Clipping

- If both outcodes are 0, line segment is inside
- If AND of outcodes not 0, line segment is outside
- Otherwise clip and test
Cohen-Sutherland Line Clipping

- If both outcodes are 0, line segment is inside
- If AND of outcodes not 0, line segment is outside
- Otherwise clip and test
Cohen-Sutherland Line Clipping

- If both outcodes are 0, line segment is inside
- If AND of outcodes not 0, line segment is outside
- Otherwise clip and test

<table>
<thead>
<tr>
<th>Bit 1</th>
<th>Bit 2</th>
<th>Bit 3</th>
<th>Bit 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>0001</td>
<td>0010</td>
<td>0011</td>
</tr>
<tr>
<td>0001</td>
<td>0010</td>
<td>0011</td>
<td>0100</td>
</tr>
<tr>
<td>0010</td>
<td>0011</td>
<td>0100</td>
<td>0101</td>
</tr>
<tr>
<td>0011</td>
<td>0100</td>
<td>0101</td>
<td></td>
</tr>
</tbody>
</table>

- $P_3$: 0000
- $P_4$: 0010
- $P_5$: 0010
- $P_6$: 0010
- $P_7$: 0000
- $P_8$: 0100
- $P_9$: 0110
- $P_{10}$: 0111
Cohen-Sutherland Line Clipping

- If both outcodes are 0, line segment is inside
- If AND of outcodes not 0, line segment is outside
- Otherwise clip and test
Cohen-Sutherland Line Clipping

- If both outcodes are 0, line segment is inside
- If AND of outcodes not 0, line segment is outside
- Otherwise clip and test

```
Bit 1  Bit 2  Bit 3  Bit 4

1001  1000  1010  0101

P3   P7

1010  0010  0100  0110

P5   P6   P9   P10

0000  0001  0100  0101

P4   P7
```
Cohen-Sutherland Line Clipping

- If both outcodes are 0, line segment is inside
- If AND of outcodes not 0, line segment is outside
- Otherwise clip and test

<table>
<thead>
<tr>
<th>Outcode</th>
<th>Points</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>P₃, P₆</td>
</tr>
<tr>
<td>0001</td>
<td>P₇, P₆</td>
</tr>
<tr>
<td>0010</td>
<td>P₅, P₆, P₉</td>
</tr>
<tr>
<td>0011</td>
<td>P₅, P₆, P₉</td>
</tr>
<tr>
<td>0100</td>
<td>P₈, P₁₀</td>
</tr>
<tr>
<td>0101</td>
<td>P₈, P₁₀</td>
</tr>
<tr>
<td>0110</td>
<td>P₈, P₁₀</td>
</tr>
<tr>
<td>1000</td>
<td>P₈, P₁₀</td>
</tr>
<tr>
<td>1001</td>
<td>P₈, P₁₀</td>
</tr>
<tr>
<td>1010</td>
<td>P₈, P₁₀</td>
</tr>
<tr>
<td>1100</td>
<td>P₈, P₁₀</td>
</tr>
<tr>
<td>1110</td>
<td>P₈, P₁₀</td>
</tr>
</tbody>
</table>
Cohen-Sutherland Line Clipping

- If both outcodes are 0, line segment is inside
- If AND of outcodes not 0, line segment is outside
- Otherwise clip and test
Cohen-Sutherland Line Clipping

- If both outcodes are 0, line segment is inside
- If AND of outcodes not 0, line segment is outside
- Otherwise clip and test
Cohen-Sutherland Line Clipping

- If both outcodes are 0, line segment is inside
- If AND of outcodes not 0, line segment is outside
- Otherwise clip and test
Cohen-Sutherland Line Clipping

- If both outcodes are 0, line segment is inside
- If AND of outcodes not 0, line segment is outside
- Otherwise clip and test
Cohen-Sutherland Line Clipping

- If both outcodes are 0, line segment is inside
- If AND of outcodes not 0, line segment is outside
- Otherwise clip and test
Cohen-Sutherland Line Clipping

- If both outcodes are 0, line segment is inside
- If AND of outcodes not 0, line segment is outside
- Otherwise clip and test
Cohen-Sutherland Line Clipping

- If both outcodes are 0, line segment is inside
- If AND of outcodes not 0, line segment is outside
- Otherwise clip and test
Cohen-Sutherland Line Clipping

- If both outcodes are 0, line segment is inside
- If AND of outcodes not 0, line segment is outside
- Otherwise clip and test
Cohen-Sutherland Line Clipping

- If both outcodes are 0, line segment is inside
- If AND of outcodes not 0, line segment is outside
- Otherwise clip and test
Clipping

• Avoid drawing parts of primitives outside window
  ◦ Points
  ◦ Line Segments
  ◦ Polygons
Polygon Clipping

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

- Find the part of a polygon inside the clip window

After Clipping
Sutherland-Hodgeman Clipping

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

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

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

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

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

- How do we clip a polygon with respect to a line?
Sutherland-Hodgeman Clipping

- Do inside test for each point in sequence,
- Insert new points when cross window boundary,
- Remove points outside window boundary
Sutherland-Hodgeman Clipping

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

Window Boundary

Outside

Inside

$P_1$ $P_2$ $P_3$ $P_4$ $P_5$
Sutherland-Hodgeman Clipping

- Do inside test for each point in sequence,
  Insert new points when cross window boundary,
  Remove points outside window boundary
Sutherland-Hodgeman Clipping

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

![Diagram showing clipping process with points P1, P2, P3, P4, P5. The window boundary is depicted with a red line, and the inside and outside areas are indicated.]
Sutherland-Hodgeman Clipping

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

- Diagram showing points $P_1, P_2, P_3, P_4, P_5$ with $P'$ on the boundary.
  The shaded area represents the clipped polygon.
  Points $P_1$ and $P_5$ are inside the window, while $P_3$ is outside.
  New points are inserted at $P'$ to maintain the boundary.
Sutherland-Hodgeman Clipping

- Do inside test for each point in sequence,
  Insert new points when cross window boundary,
  Remove points outside window boundary
Sutherland-Hodgeman Clipping

- Do inside test for each point in sequence,
- Insert new points when cross window boundary,
- Remove points outside window boundary
Sutherland-Hodgeman Clipping

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

Window Boundary

Inside

Outside

$P'$

$P_1$

$P''$

$P_2$

$P_3$

$P_4$

$P_5$
Sutherland-Hodgeman Clipping

- Do inside test for each point in sequence,
- Insert new points when cross window boundary,
- Remove points outside window boundary
3D Rendering Pipeline (for direct illumination)

- 3D Primitives
  - Modeling Transformation
    - 3D Modeling Coordinates
    - 3D World Coordinates
  - Camera Transformation
    - 3D Camera Coordinates
  - Lighting
    - 3D Camera Coordinates
  - Projection Transformation
    - 2D Screen Coordinates
  - Clipping
    - 2D Screen Coordinates
    - 2D Screen Coordinates
  - Viewport Transformation
    - 2D Image Coordinates
  - Scan Conversion
    - 2D Image Coordinates
    - Image

3D Model

2D Window

2D Screen
2D Rendering Pipeline

3D Primitives → 2D Primitives → Clipping → Scan Conversion → Image

- Clip portions of geometric primitives residing outside the window
- Fill pixels representing primitives in screen coordinates
Overview

• Scan conversion
  ◦ Figure out which pixels to fill

• Shading
  ◦ Determine a color for each filled pixel

• Depth test
  ◦ Determine when the color of a pixel should be overwritten
Scan Conversion

• Render an image of a geometric primitive by setting pixel colors

\[
\text{void SetPixel( int } x, \text{ int } y, \text{ Color rgba })
\]

• Example: Filling the inside of a triangle

\[
P_1 \quad P_2 \quad P_3
\]
Scan Conversion

• Render an image of a geometric primitive by setting pixel colors

```c
void SetPixel( int x, int y, Color rgba )
```

• Example: Filling the inside of a triangle
Triangle Scan Conversion

- Properties of a good algorithm
  - Must be fast
  - No cracks between adjacent primitives
Triangle Scan Conversion

- Properties of a good algorithm
  - Must be fast
  - No cracks between adjacent primitives
Simple Algorithm

• Color all pixels inside triangle

```c
void ScanTriangle( Triangle T , Color rgba )
{
    for each pixel P at (x,y)
        if( Inside( T , P ) )
            SetPixel( x , y , rgba );
}
```
Line defines two halfspaces

• Test: use implicit equation for a line
  - On line: \( ax + by + c = 0 \)
  - On right: \( ax + by + c < 0 \)
  - On left: \( ax + by + c > 0 \)
Inside Triangle Test

- A point is inside a triangle if it is in the positive half-space of all three boundary lines
  - Triangle vertices are ordered counter-clockwise
  - Point must be on the left side of every boundary line
Inside Triangle Test

Boolean Inside( Triangle T , Point P )
{
    for each boundary line L of T
    {
        Scalar d = L.a*P.x + L.b*P.y + L.c;
        if( d<0.0 ) return FALSE;
    }
    return TRUE;
}
Simple Algorithm

• What is bad about this algorithm?

```c
void ScanTriangle( Triangle T , Color rgba )
{
    for each pixel P at (x,y)
        if( Inside( T , P ) )
            SetPixel( x , y , rgba );
}
```
Triangle Sweep-Line Algorithm

• Take advantage of spatial coherence
  ◦ Compute which pixels are inside using horizontal spans
  ◦ Process horizontal spans in scan-line order

• Take advantage of edge linearity
  ◦ Use edge slopes to update coordinates incrementally
void ScanTriangle(Triangle T, Color rgba)
{
    for both edge pairs
    {
        initialize $x_L$, $x_R$;
        compute $dx_L/dy_L$ and $dx_R/dy_R$;
        for each scanline at $y$
            for(int $x=x_L$; $x<=x_R$; $x++$) SetPixel($x$, $y$, rgba);
                $x_L += dx_L/dy_L$;
                $x_R += dx_R/dy_R$;
    }
}

Bresenham’s algorithm works the same way, but uses only integer operations!
Polygon Scan Conversion

- Will this method work for convex polygons?
Polygon Scan Conversion

- Will this method work for convex polygons?
  - Yes, since each scan line will only intersect the polygon at two points.
Polygon Scan Conversion

- How about these polygons?
Polygon Scan Conversion

- How about these polygons?
Polygon Scan Conversion

• Fill pixels inside a polygon
  ◦ Triangle
  ◦ Quadrilateral
  ◦ Convex
  ◦ Star-shaped
  ◦ Concave
  ◦ Self-intersecting
  ◦ Holes

What problems do we encounter with arbitrary polygons?
Polygon Scan Conversion

- Need better test for points inside polygon
  - Triangle method works only for convex polygons
Inside Polygon Rule

• What is a good rule for which pixels are inside?

Concave

Self-Intersecting

With Holes
Inside Polygon Rule

- Odd-parity rule
  - Any ray from $P$ to infinity crosses odd number of edges
Polygon Sweep-Line Algorithm

- Incremental algorithm to find spans, and determine “insideness” with odd parity rule
  - Takes advantage of scan line coherence
void ScanPolygon( Polygon P , Color rgba )
{
    sort edges by maxy
    make empty "active edge list"
    for each scanline ( top-to-bottom )
    {
        insert/remove edges from "active edge list"
        update x coordinate of every active edge
        sort active edges by x coordinate
        for each pair of active edges (left-to-right)
            SetPixels( x_i , x_{i+1} , y , rgba );
    }
}
Hardware Scan Conversion

• Convert everything into triangles
  ◦ Scan convert the triangles
Scan Conversion

• What about pixels on edges?
  ◦ If we set them either “on” or “off” we get aliasing or “jaggies”
Scan Conversion

- What about pixels on edges?
  - If we set them either “on” or “off” we get aliasing or “jaggies”

This amounts to using a “nearest” interpolation filter!
Antialiasing Techniques

- Display at higher resolution
  - Corresponds to increasing sampling rate
  - Not always possible (fixed size monitors, fixed refresh rates, etc.)

- Modify pixel intensities
  - Vary pixel intensities along primitive boundaries for antialiasing
  - Must have more than bi-level display
Scan Conversion

- What about pixels on edges?
  - If we set them either “on” or “off” we get aliasing or “jaggies”
  - Vary pixel intensities along primitive boundaries for antialiasing
Antialiasing

• Method 1: Area sampling (aka prefiltering)
  ◦ Calculate percent of pixel covered by primitive
  ◦ Multiply this percentage by desired intensity/color
  ◦ Set resulting pixel to closest available display level
Antialiasing

- Method 2: Supersampling (aka postfiltering)
  - Sample as if screen were higher resolution
  - Average multiple samples to get final intensity
Antialiasing

• Method 2: Supersampling (aka postfiltering)
  ◦ Sample as if screen were higher resolution
  ◦ Average multiple samples to get final intensity

This is similar to using a “bilinear” interpolation filter!
Antialiasing

• Method 2: Supersampling (aka postfiltering)
  ◦ Sample as if screen were higher resolution
  ◦ Average multiple samples to get final intensity

This is similar to using a “bilinear” interpolation filter!

Can use other filters (e.g. Gaussian for better interpolation)
Antialiasing

Note that this makes things harder because pixels are no longer “owned” by a single triangle.

- Triangles contribute color rather than set color
- Along edges the total contribution must sum to one.
- Makes depth-testing more complicated.

\[ P_1 \]
\[ P_2 \]
\[ P_3 \]
\[ P_4 \]
Scan Conversion

• Example:

No Anti-Aliasing

4 x Anti-Aliasing

Images courtesy of NVIDIA
3D Rendering Pipeline (for direct illumination)

3D Primitives

Modeling Transformation

3D Modeling Coordinates

3D World Coordinates

Camera Transformation

3D Camera Coordinates

Lighting

3D Camera Coordinates

Projection Transformation

2D Screen Coordinates

Clipping

2D Screen Coordinates

Viewport Transformation

2D Image Coordinates

Scan Conversion

2D Image Coordinates

Image

3D Model

2D Window

2D Screen