Clipping and Scan Conversion

Michael Kazhdan

(601.357/456)

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

• Assignment 2 posted!
3D Rendering Pipeline (for direct illumination)
Transformations

\[(x, y, z)\] → 3D Object Coordinates

Modeling Transformation

3D Object Coordinates → 3D World Coordinates

Viewing Transformation

3D World Coordinates → 3D Camera Coordinates

Projection Transformation

3D Camera Coordinates → 2D Screen Coordinates

Window-to-Viewport Transformation

2D Screen Coordinates → 2D Image Coordinates

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

\[(x, y, z)\]

- Modeling Transformation
  - 3D Object Coordinates
- Viewing Transformation
  - 3D World Coordinates
- Projection Transformation
  - 3D Camera Coordinates
- Window-to-Viewport Transformation
  - 2D Screen Coordinates
  - 2D Image Coordinates
  \[(x', y')\]

Transform \( = M \)

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

\[(x, y, z)\]

Modeling Transformation

3D Object Coordinates

Viewing Transformation

3D World Coordinates

Projection Transformation

3D Camera Coordinates

Window-to-Viewport Transformation

2D Screen Coordinates

2D Image Coordinates

\[(x', y')\]

Transform = \(M\)

\(M = \) local to world transform
Transformations

\((x, y, z)\)  
3D Object Coordinates

Modeling Transformation

3D World Coordinates

Viewing Transformation

3D Camera Coordinates

Projection Transformation

2D Screen Coordinates

Window-to-Viewport Transformation

2D Image Coordinates

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

\[ \text{Transform} = T_{C\rightarrow W}^{-1} M \]

\(T_{C\rightarrow W}\) is the camera to world transform

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

\[ \begin{align*}
(x, y, z) & \\
\text{Modeling Transformation} & \rightarrow 3D \text{ Object Coordinates} \\
\text{Viewing Transformation} & \rightarrow 3D \text{ World Coordinates} \\
\text{Projection Transformation} & \rightarrow 3D \text{ Camera Coordinates} \\
\text{Window-to-Viewport Transformation} & \rightarrow 2D \text{ Screen Coordinates} \\
(x', y') & 
\end{align*} \]

\[ \text{Transform} = PT^{-1}_{C \rightarrow W} M \]

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

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

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

\[(x, y, z)\]

3D Object Coordinates

Modeling Transformation

3D World Coordinates

Viewing Transformation

3D Camera Coordinates

Projection Transformation

2D Screen Coordinates

Window-to-Viewport Transformation

2D Image Coordinates

\[\begin{align*}
\mathbf{V} &= \begin{bmatrix}
1 & 0 & v_x^1 \\
0 & 1 & v_x^2 \\
0 & 0 & 1
\end{bmatrix}
\begin{bmatrix}
\frac{v_x^2 - v_x^1}{w_x^2 - w_x^1} & 0 & 0 \\
0 & \frac{v_y^2 - v_y^1}{w_y^2 - w_y^1} & 0 \\
0 & 0 & 1
\end{bmatrix}
\begin{bmatrix}
1 & 0 & -w_x^1 \\
0 & 1 & -w_y^1 \\
0 & 0 & 1
\end{bmatrix}
\end{align*}\]
3D Rendering Pipeline (for direct illumination)

\[(x, y, z)\]

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

- **Viewing Transformation**
  - 3D World Coordinates

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

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

\[(x', y')\]

- 2D Image Coordinates

3D Model

2D Screen
Transformations

3D Geometric Primitives

- Modeling Transformation
- Viewing Transformation
- Lighting
- Projection Transformation
- Clipping
- Scan Conversion

Image

\[ I = I_E + \sum_L [K_A \cdot I_L^A + (K_D \cdot \langle \vec{N}, \vec{L} \rangle + K_S \cdot \langle \vec{V}, \vec{R} \rangle^n) \cdot I_L] \]
3D Rendering Pipeline (for direct illumination)

3D Primitives
- 3D Modeling Coordinates
  - Modeling Transformation
  - 3D World Coordinates
    - Viewing 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
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
Is point \((x, y)\) inside the clip window?
Point Clipping

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

\[
\text{inside} = (x \geq wx1) \land (x \leq wx2) \land (y \geq wy1) \land (y \leq wy2);
\]
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
  ◦ Do this as efficiently as possible by identifying the easiest cases first
Cohen-Sutherland Line Clipping

- Associate an **outcode** to each vertex
- 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

- Associate an outcode to each vertex
- 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

- Associate an **outcode** to each vertex
- 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

- Associate an outcode to each vertex
- 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

- Associate an **outcode** to each vertex
- 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

- Associate an **outcode** to each vertex
- 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>1001</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1000</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1010</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0010</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0000</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Points:
- $P_3$, $P_4$, $P_5$, $P_6$, $P_7$, $P_8$, $P_9$, $P_{10}$
Cohen-Sutherland Line Clipping

- Associate an outcode to each vertex
- 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

- Associate an outcode to each vertex
- 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

- Associate an **outcode** to each vertex
- If both outcodes are 0, line segment is inside
- If AND of outcodes not 0, line segment is outside
- **Otherwise clip and test**

---

**Diagram:**

- Vertices:
  - \( P_3 \)
  - \( P_4 \)
  - \( P_6 \)
  - \( P_5' \)
  - \( P_7 \)
  - \( P_8 \)
  - \( P_9 \)
  - \( P_{10} \)

- Bit Assignment:
  - Bit 1: 1010
  - Bit 2: 0010
  - Bit 3: 1000
  - Bit 4: 0101

---
Cohen-Sutherland Line Clipping

- Associate an outcode to each vertex
- 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

- Associate an outcode to each vertex
- 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

- Associate an **outcode** to each vertex
- 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

- Associate an **outcode** to each vertex
- 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

- Associate an outcode to each vertex
- 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

- Associate an **outcode** to each vertex
- 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

- Associate an **outcode** to each vertex
- 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

- Associate an outcode to each vertex
- If both outcodes are 0, line segment is inside
- If AND of outcodes not 0, line segment is outside
- Otherwise clip and test

How many bits would you need in 3D?
Clipping

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

- Find the part of a polygon inside the clip window
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
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
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

- $P_1$
- $P_2$
- $P_3$
- $P_4$
- $P_5$

Outside

Inside
Sutherland-Hodgeman Clipping

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

\[ \overline{P_1 P_2 P_3 P_4 P_5} \]

Window Boundary

- \( P' \) inside
- \( P'' \) outside

Inside
Outside
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

When polygons are clipped, per-vertex properties (e.g. lighting) is interpolated to the new vertices.
Sutherland-Hodgeman Clipping

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

What happens if the polygon is not convex?
3D Rendering Pipeline (for direct illumination)

1. Image
2. Scan Conversion
3. Viewport Transformation
4. Clipping
5. Projection Transformation
6. Lighting
7. Viewing Transformation
8. Modeling Transformation
9. 3D Primitives

- 2D Image Coordinates
- 2D Screen Coordinates
- 3D Camera Coordinates
- 3D World Coordinates
- 3D Modeling Coordinates
2D Rendering Pipeline

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

- Clipping: Clip portions of geometric primitives residing outside the window.
- Scan Conversion: 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

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

• Example: Filling the inside of a triangle

![Diagram of a triangle with vertices P1, P2, and P3, and filled region]
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 center (x,y)
        if( PointInsideTriangle( (x,y), T ) )
            SetPixel( x, y, rgba );
}
```
Line defines two halfspaces

- Test: use implicit equation for a line
  - On line: \( ax + by + c = 0 \)
  - To the right: \( ax + by + c < 0 \)
  - To the 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 PointInsideTriangle(Point P, Triangle T)
{
    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 center (x,y)
        if( PointInsideTriangle( (x,y) , T ) )
            SetPixel( x , y , rgba );
}
```
Triangle Sweep-Line Algorithm

- Take advantage of spatial coherence
- Take advantage of edge linearity
Triangle Sweep-Line Algorithm

```cpp
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 inside $P$ to infinity must cross an odd number of edges
Polygon Sweep-Line Algorithm

- Use incremental algorithm to find spans
- 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 successive pair of edge-points (left-to-right)
            SetPixels( x_i , x_{i+1} , y , rgba );
    }
}
Polygon Scan Conversion

- Convert everything into triangles
  - Scan convert the triangles
Polygon Scan Conversion

• Convert everything into triangles
  ◦ Scan convert the triangles

Note:
OpenGL will render polygons, but it assumes that:
• The polygon is planar
• The polygon is convex

Note:
Even if you only pass in triangles for rendering, OpenGL may still have to render (convex planar) polygons after the triangle is clipped.
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 is like 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 boundaries for antialiasing
Scan Conversion

- What about pixels on edges?
  - Setting them either “on” or “off” we get aliasing/“jaggies”
  - Antialias by varying pixel intensities along boundaries
Antialiasing

- Method 1: Area sampling
  - Calculate percent of pixel covered by primitive
  - Multiply this percentage by desired intensity/color

2%  25%  60%  100%
Antialiasing

- Method 1: Area sampling
  - Calculate percent of pixel covered by primitive
  - Multiply this percentage by desired intensity/color

This is like 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 can be done by rendering the scene multiple times with different (fractional) offsets
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.
Scan Conversion

• Example:

No Anti-Aliasing

4 x Anti-Aliasing

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

- **3D Primitives**
  - 3D Modeling Coordinates
- **Modeling Transformation**
  - 3D World Coordinates
- **Viewing 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

3D Model

2D Window

2D Screen
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 comes from the front-most primitive
• **Recall:**
  In the “lighting” phase, we calculated the color at each vertex.
Recall:
In the “lighting” phase, we calculated the color at each vertex.

- In the “scan conversion” phase, linearly interpolate colors at vertices

\[
A = (1 - \alpha) \cdot I_1 + \alpha \cdot I_2
\]

\[
B = (1 - \beta) \cdot I_2 + \beta \cdot I_3
\]

\[
I = (1 - \gamma) \cdot A + \gamma \cdot B
\]
Recall:
In the “lighting” phase, we calculated the color at each vertex.

In the “scan conversion” phase, linearly interpolate colors at vertices

\[ A = (1 - \alpha) \cdot I_1 + \alpha \cdot I_2 \]

\[ B = (1 - \beta) \cdot I_2 + \beta \cdot I_3 \]

\[ I = (1 - \gamma) \cdot A + \gamma \cdot B \]

Note: The values of \( \alpha \) and \( \beta \) only need to be updated as we move to the next scan-line. The value of \( \gamma \) needs to be updated as we advance along the scan-line.

In the “scan conversion” phase, linearly interpolate colors at vertices.
3D Rendering Pipeline (for direct illumination)

3D Primitives
  \[\xrightarrow{\text{3D Modeling Coordinates}}\]
  Modeling Transformation
  \[\xrightarrow{\text{3D World Coordinates}}\]
  Viewing Transformation
  \[\xrightarrow{\text{3D Camera Coordinates}}\]
  Lighting
  \[\xrightarrow{\text{3D Camera Coordinates}}\]
  Projection Transformation
  \[\xrightarrow{\text{2D Screen Coordinates}}\]
  Clipping
  \[\xrightarrow{\text{2D Screen Coordinates}}\]
  Viewport Transformation
  \[\xrightarrow{\text{2D Image Coordinates}}\]
  Scan Conversion
  \[\xrightarrow{\text{2D Image Coordinates}}\]
  Image

3D Model

2D Screen