Clipping and Scan Conversion

Jason Lawrence

CS 4810: Graphics

Acknowledgment: slides by Misha Kazhdan, Allison Klein, Tom Funkhouser, Adam Finkelstein and David Dobkin
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

3D Model

2D Image
Transformations

\[ p(x,y,z) \]

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

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

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

- **Window-to-Viewport Transformation**
  - 2D Image Coordinates
  
  \[ p'(x',y') \]

- 3D Object Coordinates
Transformations

\[ p(x,y,z) \]

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

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

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

\[ p(x, y, z) \]

**Modeling Transformation**

\[ 3D \text{ Object Coordinates} \]

**Camera Transformation**

\[ 3D \text{ World Coordinates} \]

**Projection Transformation**

\[ 3D \text{ Camera Coordinates} \]

**Window-to-Viewport Transformation**

\[ 2D \text{ Screen Coordinates} \]

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

Transform = \( M \)

\( M := \text{ local to world transform} \)
Transformations

\[ p(x, y, z) \]

Modeling Transformation

\[ 3D \text{ Object Coordinates} \]

Camera Transformation

\[ 3D \text{ World Coordinates} \]

Projection Transformation

\[ 3D \text{ Camera Coordinates} \]

Window-to-Viewport Transformation

\[ 2D \text{ Screen Coordinates} \]

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

Camera Up

\[ y \]

Camera Right

\[ x \]

Camera Back

\[ z \]

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

\[ C := \text{camera transform} \]

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

\[ \mathbf{p}(x, y, z) \]

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

\[ \mathbf{p}'(x', y') \]

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

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

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

\[ \mathbf{P}_p = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \]

D
Transformations

\[ p(x,y,z) \]

Modeling Transformation

3D Object Coordinates

Camera Transformation

3D World Coordinates

Projection Transformation

3D Camera Coordinates

Window-to-Viewport Transformation

2D Screen Coordinates

Normalized Coords

Viewport

Window

Image

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

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

\[ p(x, y, z) \]

- 3D Object Coordinates
- 3D World Coordinates
- 3D Camera Coordinates

\[ \text{Modeling Transformation} \]

\[ \text{Camera Transformation} \]

\[ \text{Projection Transformation} \]

- 2D Screen Coordinates
- 2D Image Coordinates

\[ \text{Window-to-Viewport Transformation} \]

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

\[ \text{Window} \]

\[ (w_x^1, w_y^1) \]

\[ \text{Viewport} \]

\[ (v_x^2, v_y^2) \]

\[ \text{Screen Coordinates} \]

\[ (v_x^2, v_y^2) \]

\[ \text{Image Coordinates} \]

\[ (v_x^2, v_y^2) \]

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

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

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

\[ p(x,y,z) \]

- **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** → **\( p'(x',y') \)**
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 → 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?

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

\[\text{Bit 1} \quad \text{Bit 2} \quad \text{Bit 3} \quad \text{Bit 4}\]

- \(P_3\)
- \(P_4\)
- \(P_6\)
- \(P_7\)
- \(P_8\)
- \(P_{10}\)
- \(P'_5\)
- \(P_9\)
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

Note: both lines have the same bit codes!
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

• Before clipping
Cohen-Sutherland Line Clipping

- After clipping
Clipping

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

- Find the part of a polygon inside the clip window

Before Clipping
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

\begin{itemize}
  \item \text{Window Boundary}
  \item \text{Inside}
  \item \text{Outside}
\end{itemize}
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

- \( P_1 \)
- \( P_2 \)
- \( P_3 \)
- \( P_4 \)
- \( P_5 \)
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

\[ P' \rightarrow P_2 \rightarrow P_1 \rightarrow P'' \]
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
                                    - 2D Window
                                      - 2D Screen
2D Rendering Pipeline

3D Primitives

↓

2D Primitives

↓

Clipping

Clip portions of geometric primitives residing outside the window

↓

Scan Conversion

Fill pixels representing primitives in screen coordinates

↓

Image
Overview

• Scan conversion
  o Figure out which pixels to fill

• Shading
  o Determine a color for each filled pixel

• Depth test
  o 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
Scan Conversion

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

```cpp
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);
                $x_L$ += $dx_L/dy_L$;
                $x_R$ += $dx_R/dy_R$;
    }
}
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

Convex Polygon

Concave Polygon
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 max y
    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
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?
  o 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?
  o If we set them either “on” or “off” we get aliasing or “jaggies”
  o Vary pixel intensities along primitive boundaries for antialiasing
Antialiasing

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

![Diagram showing antialiasing process with points P1, P2, and P3 and percentages 2%, 25%, 60%, and 100%]
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)
  o Sample as if screen were higher resolution
  o Average multiple samples to get final intensity

This amounts 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 amounts to using a “bilinear” interpolation filter!

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

- Example:

No Anti-Aliasing

4 x Anti-Aliasing

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

1. **3D Primitives** → **3D Modeling Coordinates**
2. **Modeling Transformation** → **3D World Coordinates**
3. **Camera Transformation** → **3D Camera Coordinates**
4. **Lighting** → **3D Camera Coordinates**
5. **Projection Transformation** → **2D Screen Coordinates**
6. **Clipping** → **2D Screen Coordinates**
7. **Viewport Transformation** → **2D Image Coordinates**
8. **Scan Conversion** → **2D Image Coordinates**
9. **Image**

- **3D Model**
- **2D Window**
- **2D Screen**