Clipping and Scan Conversion

Michael Kazhdan

(601.457/657)

HB Ch. 3.2, 3.11, 6.7, 6.8
FvDFH Ch. 3.2, 3.6, 3.12, 3.14
3D Rendering Pipeline (for direct illumination)

3D Primitives

Modeling Transformation

Viewing Transformation

Lighting

Projection Transformation

Clipping

Viewport Transformation

Scan Conversion

Image

3D Modeling Coordinates

3D World Coordinates

3D Camera Coordinates

3D Camera Coordinates

2D Window Coordinates

2D Window Coordinates

2D Viewport Coordinates

2D Viewport Coordinates

2D Viewport

3D Model
Transformations

\[(x, y, z)\]

**Modeling Transformation**

3D Object Coordinates

**Viewing Transformation**

3D World Coordinates

**Projection Transformation**

3D Camera Coordinates

**Window-to-Viewport Transformation**

2D Window Coordinates

\[(x', y')\]

3D Object Coordinates

3D World Coordinates
Transformations

\[(x, y, z)\]

Modeling Transformation

3D Object Coordinates

Viewing Transformation

3D World Coordinates

Projection Transformation

3D Camera Coordinates

Window-to-Viewport Transformation

2D Window Coordinates

Transform = \(T_{W \leftarrow o}\)

\(T_{W \leftarrow o} = \text{object to world transform}\)
Transformations

$$(x, y, z)$$

3D Object Coordinates

Modeling Transformation

3D World Coordinates

Viewing Transformation

3D Camera Coordinates

Projection Transformation

2D Window Coordinates

Window-to-Viewport Transformation

2D Viewport Coordinates

$$(x', y')$$

Camera Up

Camera Right

Camera Back

$T_{C←W} = W_{W←C}^{-1} = \text{camera to world transform}$

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

\((x, y, z)\)

Modeling Transformation

3D Object Coordinates

Viewing Transformation

3D World Coordinates

Projection Transformation

3D Camera Coordinates

Window-to-Viewport Transformation

2D Window Coordinates

\((x', y')\)

Transform = \(T_{C \leftarrow W} T_{W \leftarrow O}\)

Modelview transform
Transformations

\[(x, y, z)\]

\[\text{Modeling Transformation}\]

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

\[\text{Viewing Transformation}\]

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

\[\text{Projection Transformation}\]

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

\[\text{Projection Transform} = P_{S \leftarrow C} T_{C \leftarrow W} T_{W \leftarrow O}\]

\[P_{S \leftarrow C} = \text{projection transform}\]

\[
P_{S \leftarrow C}^O = \begin{bmatrix}
1 & 0 & L \cos \phi & 0 \\
0 & 1 & L \sin \phi & 0 \\
0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1
\end{bmatrix}
\]

\[
P_{S \leftarrow C}^P = \begin{bmatrix}
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & 1 & 0
\end{bmatrix}
\]

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

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

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

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

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

\[(x, y, z)\]

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

2. **Viewing Transformation**
   - 3D World Coordinates
   - 3D Camera Coordinates

3. **Projection Transformation**
   - 2D Window Coordinates

4. **Window-to-Viewport Transformation**
   - 2D Viewport Coordinates

\[(x', y')\]

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

\[(x, y, z)\]

1. **Modeling Transformation**
   - 3D Object Coordinates
2. **Viewing Transformation**
   - 3D World Coordinates
3. **Projection Transformation**
   - 3D Camera Coordinates
4. **Window-to-Viewport Transformation**
   - 2D Screen Coordinates

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

3D Primitives
- Modeling Transformation
- Viewing Transformation
- Lighting
- Projection Transformation
- Clipping
- Viewport Transformation
- Scan Conversion

Image

3D Model

2D Screen

$I = I_E + \sum_L \left[ K_A \cdot I_L^A + \left( K_D \cdot \langle \vec{N}, \vec{L} \rangle + K_S \cdot \langle \vec{V}, \vec{R} \rangle^n \right) \cdot I_L \right]$
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 Window Coordinates

Clipping → 2D Window Coordinates

Viewport Transformation → 2D Viewport Coordinates

Scan Conversion → Image
Clipping

- Avoid drawing parts of primitives outside window
  - Window defines the subset of the 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?
Is point \((x, y)\) inside the clip window?

\[
\text{inside} = (x \geq x_{\text{min}}) \land (x < x_{\text{max}}) \land (y \geq y_{\text{min}}) \land (y < y_{\text{max}});
\]
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 a 4-bit outcode \( b_0 b_1 b_2 b_3 \) to each vertex
  - \( b_0 = 1 \) if the vertex is left of the window
  - \( b_1 = 1 \) if the vertex is right of the window
  - \( b_2 = 1 \) if the vertex is below the window
  - \( b_3 = 1 \) if the vertex is above the window
Cohen-Sutherland Line Clipping

- Associate a 4-bit outcode $b_0b_1b_2b_3$ to each vertex
- If both outcodes are 0, line segment is inside
- If AND of outcodes is not 0, line segment is outside
- Otherwise clip and test
Cohen-Sutherland Line Clipping

- Associate a 4-bit outcode $b_0b_1b_2b_3$ to each vertex
- If both outcodes are 0, line segment is inside
- If AND of outcodes is not 0, line segment is outside
- Otherwise clip and test
Cohen-Sutherland Line Clipping

- Associate a 4-bit outcode $b_0 b_1 b_2 b_3$ to each vertex.
- If both outcodes are 0, line segment is inside.
- If AND of outcodes is not 0, line segment is outside.
- Otherwise clip and test.

```
0000 01001000
0001 01011001
0010 01101010
```

$P_{10}$ $P_{9}$ $P_{8}$ $P_{7}$ $P_{6}$ $P_{5}$ $P_{4}$ $P_{3}$
Cohen-Sutherland Line Clipping

- Associate a 4-bit outcode $b_0b_1b_2b_3$ to each vertex
- If both outcodes are 0, line segment is inside
- If AND of outcodes is not 0, line segment is outside
- Otherwise clip and test
**Cohen-Sutherland Line Clipping**

- Associate a 4-bit outcode $b_0b_1b_2b_3$ to each vertex.
- If both outcodes are 0, line segment is inside.
- If AND of outcodes is not 0, line segment is outside.
- **Otherwise clip and test**
Cohen-Sutherland Line Clipping

- Associate a 4-bit outcode $b_0b_1b_2b_3$ to each vertex
- If both outcodes are 0, line segment is inside
- If AND of outcodes is not 0, line segment is outside
- Otherwise clip and test
Cohen-Sutherland Line Clipping

- Associate a 4-bit outcode $b_0b_1b_2b_3$ to each vertex
- If both outcodes are 0, line segment is inside
- If AND of outcodes is not 0, line segment is outside
- Otherwise clip and test
Cohen-Sutherland Line Clipping

- Associate a 4-bit outcode $b_0b_1b_2b_3$ to each vertex.
- If both outcodes are 0, line segment is inside.
- If AND of outcodes is not 0, line segment is outside.
- Otherwise clip and test.

```
0000 01001000
0001 01011001
0010 01101010
P_5'
P_7
P_9
P_10
```

Diagram with vertices labeled and outcodes for each region.
Cohen-Sutherland Line Clipping

- Associate a 4-bit outcode $b_0b_1b_2b_3$ to each vertex
- If both outcodes are 0, line segment is inside
- If AND of outcodes is not 0, line segment is outside
- Otherwise clip and test
Cohen-Sutherland Line Clipping

• Associate a 4-bit outcode $b_0b_1b_2b_3$ to each vertex
• If both outcodes are 0, line segment is inside
• If AND of outcodes is not 0, line segment is outside
• Otherwise clip and test
Cohen-Sutherland Line Clipping

- Associate a 4-bit outcode $b_0 b_1 b_2 b_3$ to each vertex
- If both outcodes are 0, line segment is inside
- If AND of outcodes is not 0, line segment is outside
- Otherwise clip and test

\[
\begin{align*}
&P_3' &\quad P_4 \\
&P_6 &\quad P_7 \\
&P_5 &\quad P_9 \\
&P_8' &\quad P_{10}
\end{align*}
\]
Cohen-Sutherland Line Clipping

• Associate a 4-bit outcode $b_0b_1b_2b_3$ to each vertex
• If both outcodes are 0, line segment is inside
• If AND of outcodes is not 0, line segment is outside
• Otherwise clip and test

$$
\begin{align*}
\begin{array}{c}
\text{0000} \\
\text{0001} \\
\text{0010} \\
\text{0011} \\
\text{0100} \\
\text{0101} \\
\text{0110} \\
\text{1000} \\
\text{1001} \\
\end{array}
\end{align*}
$$

$$
\begin{align*}
\begin{array}{c}
P_1, P_2, P_3, P_4, P_5, P_6, P_7, P_8, P_9, P_{10}
\end{array}
\end{align*}
$$
Cohen-Sutherland Line Clipping

- Associate a 4-bit outcode $b_0b_1b_2b_3$ to each vertex
- If both outcodes are 0, line segment is inside
- If AND of outcodes is not 0, line segment is outside
- Otherwise clip and test

```
<table>
<thead>
<tr>
<th>1001</th>
<th>0001</th>
<th>0101</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>0000</td>
<td>0100</td>
</tr>
<tr>
<td>1010</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>0110</td>
</tr>
</tbody>
</table>
```

$P_0, P_1, P_2, P_3, P_4, P_5, P_6, P_7, P_8, P_9, P_{10}$
Cohen-Sutherland Line Clipping

- Associate a 4-bit outcode $b_0b_1b_2b_3$ to each vertex
- If both outcodes are 0, line segment is inside
- If AND of outcodes is not 0, line segment is outside
- Otherwise clip and test
Cohen-Sutherland Line Clipping

- Associate a 4-bit outcode $b_0b_1b_2b_3$ to each vertex
- If both outcodes are 0, line segment is inside
- If AND of outcodes is not 0, line segment is outside
- Otherwise clip and test
Cohen-Sutherland Line Clipping

- Associate a 4-bit outcode $b_0b_1b_2b_3$ to each vertex
- If both outcodes are 0, line segment is inside
- If AND of outcodes is 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 **convex** polygon with respect to a (window boundary) line?
Sutherland-Hodgeman Clipping

- Do interior test for each point in sequence.
- Insert a new point when crossing the line.
- Remove points outside the line.
Sutherland-Hodgeman Clipping

- Do interior test for each point in sequence.
- Insert a new point when crossing the line.
- Remove points outside the line.

![Diagram of Sutherland-Hodgeman Clipping](image.png)
Sutherland-Hodgeman Clipping

- Do interior test for each point in sequence.
- Insert a new point when crossing the line.
- Remove points outside the line.

*Diagram showing points on a line and inside the polygon.*
Sutherland-Hodgeman Clipping

- Do interior test for each point in sequence.
- Insert a new point when crossing the line.
- Remove points outside the line.
Sutherland-Hodgeman Clipping

- Do interior test for each point in sequence.
- Insert a new point when crossing the line.
- Remove points outside the line.
Sutherland-Hodgeman Clipping

• Do interior test for each point in sequence.
• Insert a new point when crossing the line.
• Remove points outside the line.
Sutherland-Hodgeman Clipping

- Do interior test for each point in sequence.
- Insert a new point when crossing the line.
- Remove points outside the line.
Sutherland-Hodgeman Clipping

- Do interior test for each point in sequence.
- Insert a new point when crossing the line.
- Remove points outside the line.
Sutherland-Hodgeman Clipping

- Do interior test for each point in sequence.
- Insert a new point when crossing the line.
- Remove points outside the line.
Sutherland-Hodgeman Clipping

- Do interior test for each point in sequence.
- Insert a new point when crossing the line.
- Remove points outside the line.

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

- Do interior test for each point in sequence.
- Insert a new point when crossing the line.
- Remove points outside the line.

[WARNING] If the polygon is not convex, we may end up with more than one polygon!
At this point we have the:
- Positions of the mesh vertices (including new vertices obtained through clipping)
- Color information at each vertex.
- A list of (possibly clipped) polygons describing the intersection of the projected 3D polygons with the window.
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 viewport 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

    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
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

```c
void ScanTriangle( Triangle T, Color rgba )
{
    for both edge pairs
    {
        initialize $x_L$, $x_R$, $y$;
        compute $dx_L/dy_L$ and $dx_R/dy_R$;
        until $y$ reaches the first end-point
        for( int $x=x_L$; $x<=x_R$; $x++$ ) SetPixel( $x$, $y$, rgba );
        $x_L += dx_L/dy_L$;
        $x_R += dx_R/dy_R$;
        $y++$;
    }
}
```
void ScanTriangle(Triangle T, Color rgba)
{
    for both edge pairs
    {
        initialize \( x_L, x_R, y \);
        compute \( \frac{dx_L}{dy_L} \) and \( \frac{dx_R}{dy_R} \);
        until \( y \) reaches the first end-point
        for( int x=x_L; x<=x_R; x++ ) SetPixel( x, y, rgba );
        \( x_L += \frac{dx_L}{dy_L} \);
        \( x_R += \frac{dx_R}{dy_R} \);
        y++;
    }
}
Triangle Sweep-Line Algorithm

```c
void ScanTriangle( Triangle T , Color rgba )
{
    for both edge pairs
    {
        initialize $x_L$, $x_R$, $y$;
        compute $dx_L/dy_L$ and $dx_R/dy_R$;
        until $y$ reaches the first end-point
        for( int $x=x_L$ ; $x<=x_R$ ; $x++$ ) SetPixel( $x$ , $y$ , rgba );
        $x_L$ += $dx_L/dy_L$;
        $x_R$ += $dx_R/dy_R$;
        $y++$;
    }
}
```

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
  - 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 (horizontal) parity rule

• Takes advantage of scan line coherence
Polygon Sweep-Line Algorithm

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 Sweep-Line Algorithm

```cpp
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 );
    }
}
```

Observation:
Don’t have to explicitly sort since ordering will not change until an intersection event occurs.
And intersections only happen between adjacent edges.
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” (similar to using nearest interpolation)
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

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

- Ideally: 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

- In practice: Supersampling (aka postfiltering)
  - Sample as if screen were higher resolution
  - Average multiple samples to get final intensity
    » This is done by rendering the scene multiple times with different (fractional) offsets and averaging

![Diagram showing supersampling process with points P1, P2, and P3, and the resulting intensity averaging.]
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 Window Coordinates
  - Clipping
  - 2D Window Coordinates
  - Viewport Transformation
  - 2D Viewport Coordinates
  - Scan Conversion
  - 2D Viewport Coordinates
  - Image
    - 3D Model
    - 2D Viewport
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.

\[
I = I_E + \sum_L \left[ K_A \cdot I_L^A + \left( K_D \cdot \langle \vec{N}, \vec{L} \rangle + K_S \cdot \langle \vec{V}, \vec{R} \rangle^n \right) \cdot I_L \right]
\]
Recall:
In the **Lighting** phase, we calculated the color at each vertex.

- **Scan Convert:**
  Linearly interpolate across and between edges
  \[(I_1, I_2, I_3) \rightarrow (I_{12}, I_{23}) \rightarrow I_{123}\]
Gouraud Shading

Note: The values of $I_{12}$ and $I_{23}$ only need to be updated as we move to the next scan-line. The value of $I_{123}$ needs to be updated as we advance along the scan-line.

- **Scan Convert:**
  Linearly interpolate across and between edges
  
  $$(I_1, I_2, I_3) \rightarrow (I_{12}, I_{23}) \rightarrow I_{123}$$
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 Window Coordinates
  - Clipping
  - 2D Window Coordinates
  - Viewport Transformation
  - 2D Viewport Coordinates
  - Scan Conversion
  - 3D Model
  - Image
  - 2D Viewport Coordinates
  - 2D Viewport
  - 3D Model