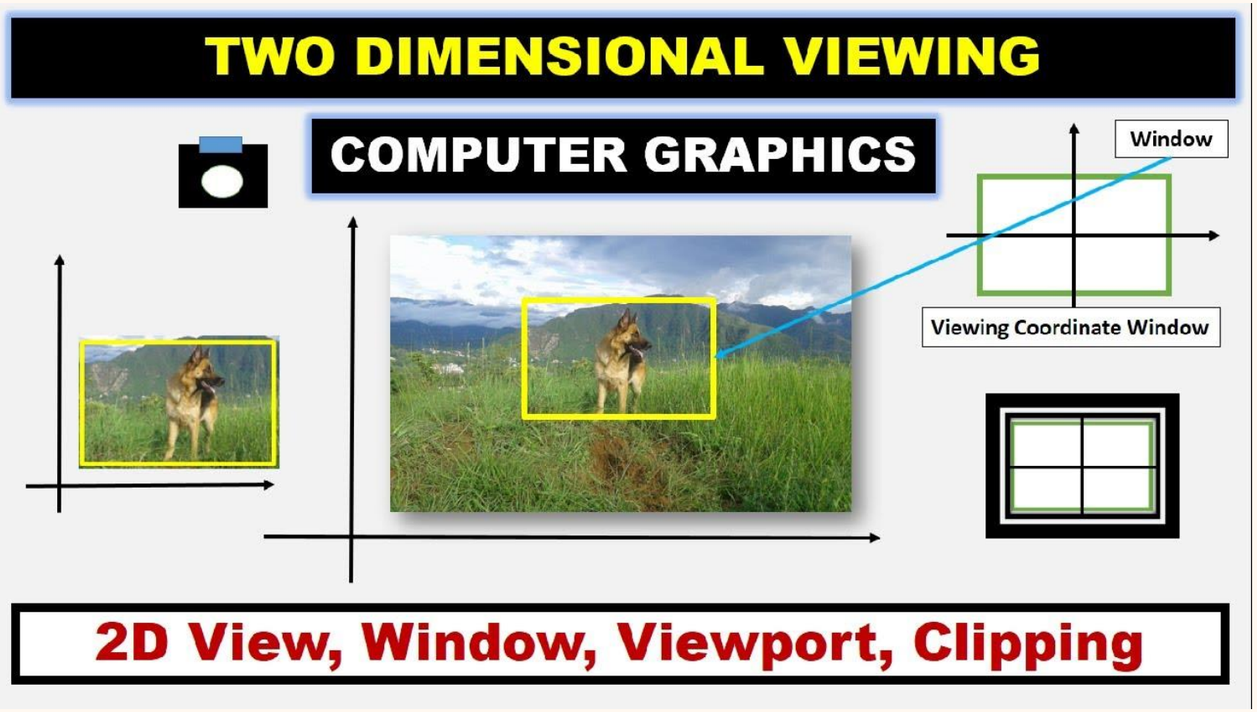
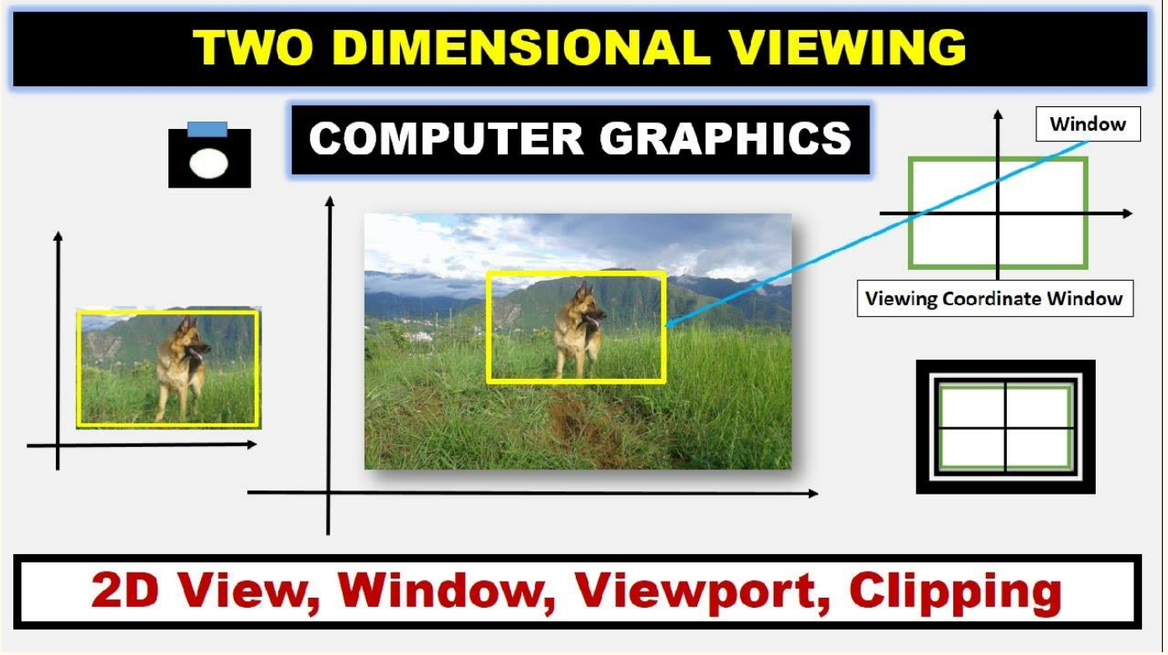
2D Viewing and Clipping

Introduction to 2D Viewing and Clipping



2D viewing and clipping are essential techniques in computer graphics for rendering a portion of a scene based on the viewer's perspective. The **2D viewing** process involves transforming real-world objects into a viewing coordinate system that maps them to a displayable area. This transformation is crucial for rendering only the visible parts of the scene.

Introduction to 2D Viewing and Clipping



Clipping is used to remove parts of objects that fall outside the visible area, known as the viewport. There are several types of clipping, such as line clipping and polygon clipping, which help in determining whether an object or part of an object should be displayed. The **window-to-viewport transformation** plays a significant role in mapping the visible portion from the world coordinates to screen coordinates

Two Dimensional Viewing: The Viewing Pipeline

The **2D Viewing Pipeline** is a fundamental concept in computer graphics used to define how a scene is mapped from the world coordinate system to the display device. It involves a series of transformations that allow objects in a scene to be viewed from different perspectives and displayed within a specific window. Here’s an outline of the steps involved in the 2D viewing pipeline:

Two Dimensional Viewing: The Viewing Pipeline

World Coordinates

The **World Coordinate System (WCS)** is a global reference frame used in computer graphics to define the positions of objects in a scene. It acts as a universal coordinate system where all objects are initially defined and modeled. The objects in the scene are first defined in a world coordinate system (WCS).

Window Definition

A **window** is defined in the WCS. It represents the rectangular portion of the scene that the user wants to display. The window is defined by the minimum and maximum x and y coordinates (e.g., (x_{min},y_{min}) and (x_{max},y_{max}).

Two Dimensional Viewing: The Viewing Pipeline

Window – to – Viewport transformation

A **transformation** is applied to map the coordinates from the world window to the viewport. This involves scaling and translating the window coordinates so that they fit into the viewport.

The window-to-viewport transformation can be described mathematically as:

$$x_v = x_{vmin} + ((x_w - x_{wmin}) / x_{wmax} - x_{wmin})(x_{vmax} - x_{vmin})$$

$$y_v = y_{vmin} + ((y_w - y_{wmin}) / y_{wmax} - y_{wmin})(y_{vmax} - y_{vmin})$$

Two Dimensional Viewing: The Viewing Pipeline

Window – to – Viewport transformation

The window-to-viewport transformation can be described mathematically as:

$x_v = x_{vmin} + ((x_w - x_{wmin}) / x_{wmax} - x_{wmin})(x_{vmax} - x_{vmin})$

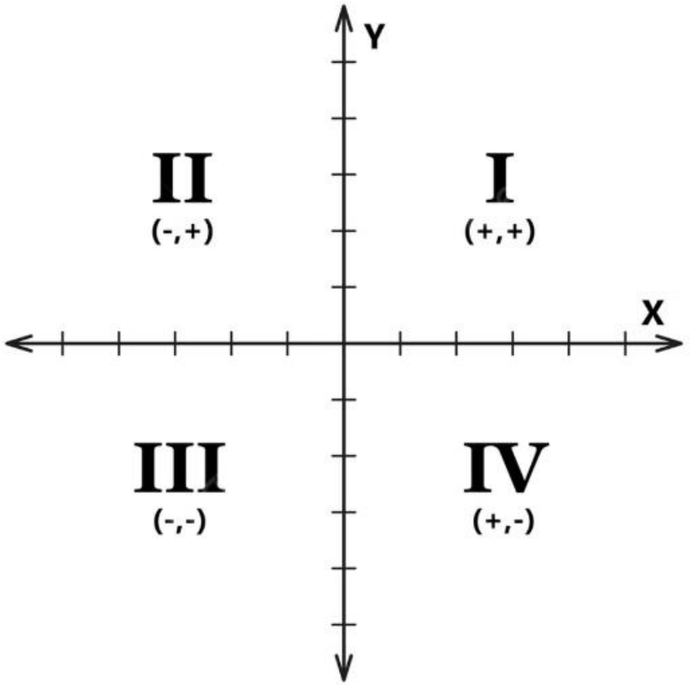
$y_v = y_{vmin} + ((y_w - y_{wmin}) / y_{wmax} - y_{wmin})(y_{vmax} - y_{vmin})$

Where:

- (x_w, y_w) are the window coordinates,
- (x_v, y_v) are the viewport coordinates,
- (x_{wmin}, y_{wmin}) and (x_{wmax}, y_{wmax}) are the window boundaries,
- (x_{vmin}, y_{vmin}) and (x_{vmax}, y_{vmax}) are the viewport boundaries.

Two Dimensional Viewing: Viewing Coordinate Reference Frame

The **Viewing Coordinate Reference Frame** (also known as the **Viewing Coordinate System**, or VCS) is a coordinate system used in computer graphics to define how a scene or object is viewed. In two-dimensional (2D) viewing, the VCS helps in transforming objects from the **World Coordinate System (WCS)** to the **Device Coordinate System (DCS)** through intermediate steps, allowing for transformations like zooming, panning, or rotating the scene for viewing.



Two Dimensional Viewing: Viewing Coordinate Reference Frame

Key Concepts of the Viewing Coordinate Reference Frame:

1. **Viewing Transformation:** The **viewing transformation** is the process of mapping or transforming the world coordinates of objects (WCS) into a **Viewing Coordinate System** (VCS).

2. **View Plane (Projection Plane):** The **view plane** or **projection plane** is where the objects in the viewing coordinate system are projected to form the final image.

3. **Window in the Viewing Coordinate System:** In the VCS, a **viewing window** is defined, which corresponds to a rectangular portion of the scene in world coordinates that the viewer wants to see.

Two Dimensional Viewing: Viewing Coordinate Reference Frame

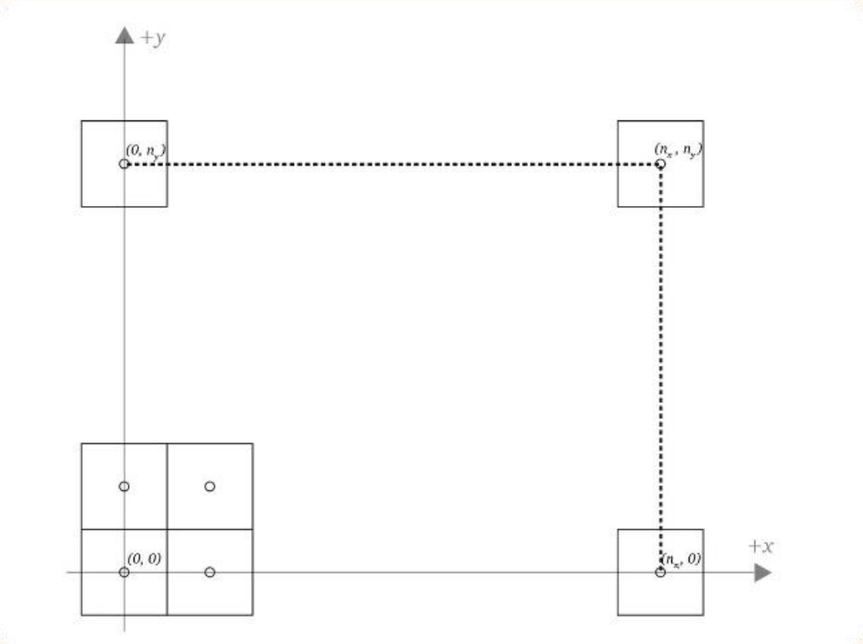
Key Concepts of the Viewing Coordinate Reference Frame:

4. **Alignment with the World Coordinate System (WCS):** The viewing coordinate system is often aligned with the world coordinate system. However, transformations can occur to alter the view—such as translating, scaling, or rotating the window—in order to display different perspectives of the scene.

5. **Clipping in VCS:** Clipping in the viewing coordinate system ensures that only objects within the defined window are displayed, while objects outside the window are excluded from the final image.

Window-to-Viewport Coordinate Transformation

Window-to-Viewport Coordinate Transformation is a key process in computer graphics used to map a portion of a 2D scene (defined by the **window**) onto a designated area of the display screen (defined by the **viewport**). This transformation ensures that the content within the window is scaled and positioned correctly within the viewport on the output device (such as a monitor or screen).



Window-to-Viewport Coordinate Transformation

Overview of Terms:

1 Window

- The window represents a rectangular region in the **world coordinate system (WCS)** that defines the portion of the scene that we want to display.
- It is defined by the boundaries (x_{wmin}, y_{wmin}) and (x_{wmax}, y_{wmax}) which represent the minimum and maximum x and y coordinates of the window.

Window-to-Viewport Coordinate Transformation

Overview of Terms:

2 **Viewport**

- The viewport represents a rectangular area on the **display coordinate system (DCS)**, or normalized device coordinates, where the window contents will be displayed.
- It is defined by the boundaries (x_{vmin}, y_{vmin}) and (x_{vmax}, y_{vmax}), which represent the minimum and maximum x and y coordinates of the viewport.

Window-to-Viewport Coordinate Transformation

Purpose of Window-to-Viewport Transformation

The primary purpose is to ensure that the portion of the scene visible in the window is scaled and mapped to fit within the viewport, allowing different views of the scene to be displayed in different sizes or positions on the screen.

Window-to-Viewport Coordinate Transformation

Mathematical Representation of the Translation

Given a point (x_w,y_w) in the window coordinate system, the corresponding point (x_v,y_v) in the viewport coordinate system is calculated as follows:

1. Transformation Formula for X-coordinate:

$$x_v = x_{\text{vmin}} + \left(\frac{x_w - x_{\text{wmin}}}{x_{\text{wmax}} - x_{\text{wmin}}} \right) \times (x_{\text{vmax}} - x_{\text{vmin}})$$

Window-to-Viewport Coordinate Transformation

Mathematical Representation of the Translation

2. Transformation Formula for Y-coordinate:

$$y_v = y_{\text{vmin}} + \left(\frac{y_w - y_{\text{wmin}}}{y_{\text{wmax}} - y_{\text{wmin}}} \right) \times (y_{\text{vmax}} - y_{\text{vmin}})$$

Explanation of Formula:

- (x_w, y_w) are the coordinates of a point in the window.
- (x_v, y_v) are the coordinates of the same point in the viewport.
- $(x_{\text{wmin}}, y_{\text{wmin}})$ and $(x_{\text{wmax}}, y_{\text{wmax}})$ are the window boundaries.
- $(x_{\text{vmin}}, y_{\text{vmin}})$ and $(x_{\text{vmax}}, y_{\text{vmax}})$ are the viewport boundaries.

Window-to-Viewport Coordinate Transformation

Steps in Window-to-Viewport Transformation:

1. **Clipping:** First, the objects in the scene are clipped to the boundaries of the window, so only the portion of the scene inside the window is considered.

2. **Scaling:** The coordinates of the objects inside the window are then scaled proportionally to fit inside the viewport. The scaling factors for the x and y dimensions are derived from the ratio of the window size to the viewport size.

3. **Translation:** After scaling, the coordinates are translated so that the window's position aligns with the viewport's position on the display.

Window-to-Viewport Coordinate Transformation

Example

Suppose we have the following:

- Window coordinates: $(x_{\text{wmin}}, y_{\text{wmin}}) = (1, 2)$, $(x_{\text{wmax}}, y_{\text{wmax}}) = (5, 6)$
- Viewport coordinates: $(x_{\text{vmin}}, y_{\text{vmin}}) = (100, 200)$, $(x_{\text{vmax}}, y_{\text{vmax}}) = (300, 400)$

Now, to transform a point $(x_w, y_w) = (3, 4)$ from the window to the viewport:

For the x-coordinate:

$$x_v = 100 + \left(\frac{3-1}{5-1}\right) \times (300 - 100) = 100 + \frac{2}{4} \times 200 = 100 + 100 = 200$$

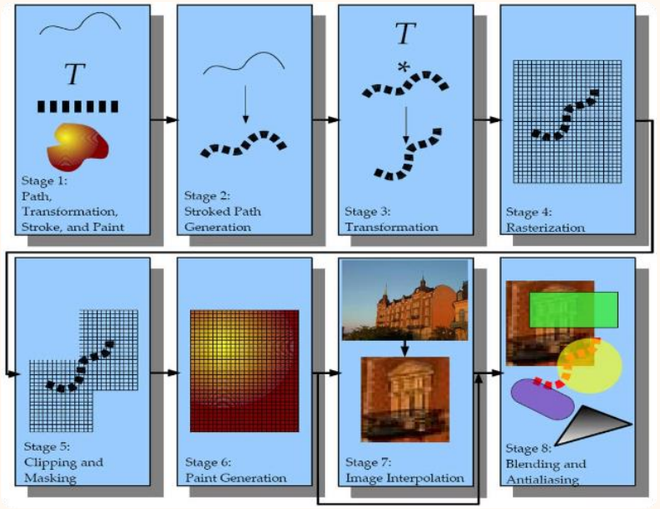
For the y-coordinate:

$$y_v = 200 + \left(\frac{4-2}{6-2}\right) \times (400 - 200) = 200 + \frac{2}{4} \times 200 = 200 + 100 = 300$$

So, the point $(3, 4)$ in the window would be mapped to $(200, 300)$ in the viewport.

Transformations in OpenGL Clipping Operations

In OpenGL, **2D transformations** are essential for manipulating the positions, orientations, and sizes of objects within the scene, and they play a key role in the **clipping operations** that ensure objects are properly displayed within the viewing area. Clipping is the process of determining which parts of a scene are visible in the viewport and discarding the rest.



Transformations in OpenGL Clipping Operations

2D Transformations in OpenGL:

OpenGL supports several types of 2D transformations that are used in various stages of rendering. These transformations are applied before the clipping process to position objects within the viewing volume. The common transformations include:

Transformations in OpenGL Clipping Operations

2D Transformations in OpenGL:

- **Translation:**
  - Moves an object from one position to another in the 2D plane.
  - Represented by a translation matrix:

$$T(x, y) = \begin{bmatrix} 1 & 0 & x \\ 0 & 1 & y \\ 0 & 0 & 1 \end{bmatrix}$$

- **Rotation**
  - Rotates an object around a point (usually the origin) in the 2D plane.
  - Represented by a rotation matrix:

$$R(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

  - The angle of rotation, θ, determines how much the object rotates.

Transformations in OpenGL Clipping Operations

2D Transformations in OpenGL:

- **Scaling:**
  - Changes the size of an object in the x and y directions.
  - Represented by a scaling matrix:

$$S(s_x, s_y) = \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

  - The scaling factors s_x and s_y define how much the object is stretched or shrunk.

- **Shear:**
  - Skews the object by shifting one axis relative to the other.
  - Represented by a shear matrix:

$$Sh_x(sh_x) = \begin{bmatrix} 1 & sh_x & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad Sh_y(sh_y) = \begin{bmatrix} 1 & 0 & 0 \\ sh_y & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Transformations in OpenGL Clipping Operations

2D Transformations in OpenGL:

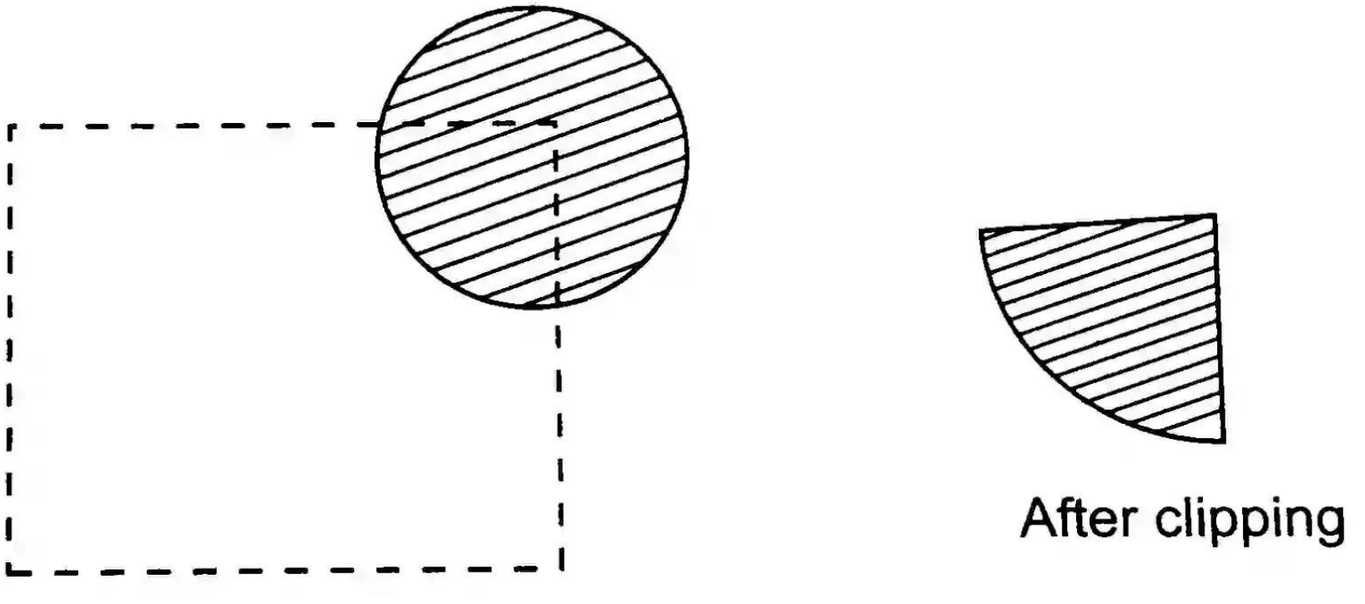
- **Reflection:**
  - Flips an object across an axis.
  - For reflection across the x-axis or y-axis, the matrix becomes:

$$R_x = \begin{bmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad R_y = \begin{bmatrix} -1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Transformations in OpenGL Clipping Operations

Clipping Operations

After applying the appropriate 2D transformations, the next step is clipping. OpenGL performs clipping in **Normalized Device Coordinates (NDC)**, meaning it clips the geometry to a standardized viewing volume, ensuring that only visible portions of objects are rendered.



Transformations in OpenGL Clipping Operations

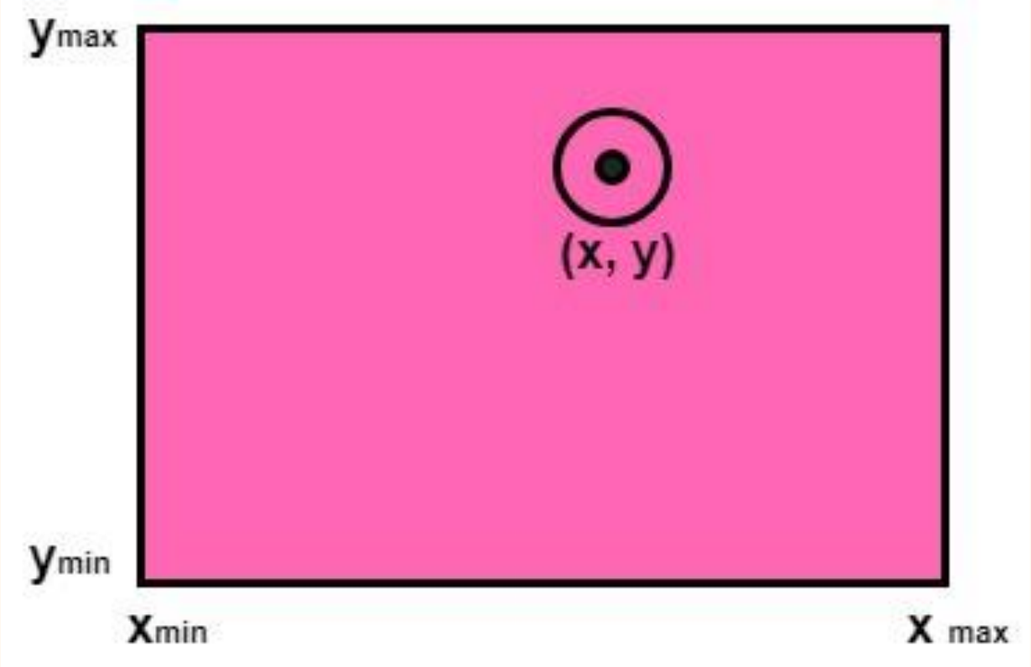
Viewport and Clipping Window:

- The **clipping window** is a rectangle in the **viewing coordinates** that defines the visible region of the scene. Everything outside this window is clipped, or removed, from the scene.
- OpenGL defines the clipping window in NDC as a 2D box with coordinates (−1,−1) to (1,1). This box defines the visible area of the scene after transformations are applied.

Transformations in OpenGL Clipping Operations

Point Clipping

Point clipping is the simplest form of clipping in computer graphics. It involves determining whether a given point is inside or outside a defined region, typically referred to as the **clipping window** or **viewing window**. If a point lies inside the clipping window, it is retained and rendered; if it lies outside, it is discarded.



Transformations in OpenGL Clipping Operations

Point Clipping: Clipping Window

The clipping window is usually defined as a rectangular area in 2D space. It is determined by specifying its **minimum** and **maximum** x and y coordinates:

- x_{min}: The minimum x-coordinate (left boundary)
- x_{max}: The maximum x-coordinate (right boundary)
- y_{min}: The minimum y-coordinate (bottom boundary)
- y_{max}: The maximum y-coordinate (top boundary)

This rectangular area forms the region where all visible points should be displayed.

Transformations in OpenGL Clipping Operations

Point Clipping: Conditions for Point Clipping

For a point P(x,y) to be inside the clipping window, the following conditions must be satisfied:

$$x_{min} \leq x \leq x_{max}$$

$$y_{min} \leq y \leq y_{max}$$

If all the above conditions are met, the point is **inside** the clipping window, and it will be displayed. Otherwise, the point is **outside** the window and is clipped (i.e., not displayed).

Transformations in OpenGL Clipping Operations

Point Clipping: Example

Let’s assume a clipping window is defined with the following coordinates:

$x_{min} = 10$, $x_{max} = 100$, $y_{min} = 20$, $y_{max} = 80$

Now, given a point P(30,50), we can check if it satisfies the conditions:

$x_{min} \leq x \leq x_{max}$ becomes $10 \leq 30 \leq 100$, which is true.

$y_{min} \leq y \leq y_{max}$ becomes $20 \leq 50 \leq 80$, which is also true.

Since both conditions are true, the point P(30,50) is inside the clipping window and will be displayed.

Transformations in OpenGL Clipping Operations

Point Clipping: Example

$x_{min} = 10$, $x_{max} = 100$, $y_{min} = 20$, $y_{max} = 80$

Now, given a point Q(150,50), we can check if it satisfies the conditions:

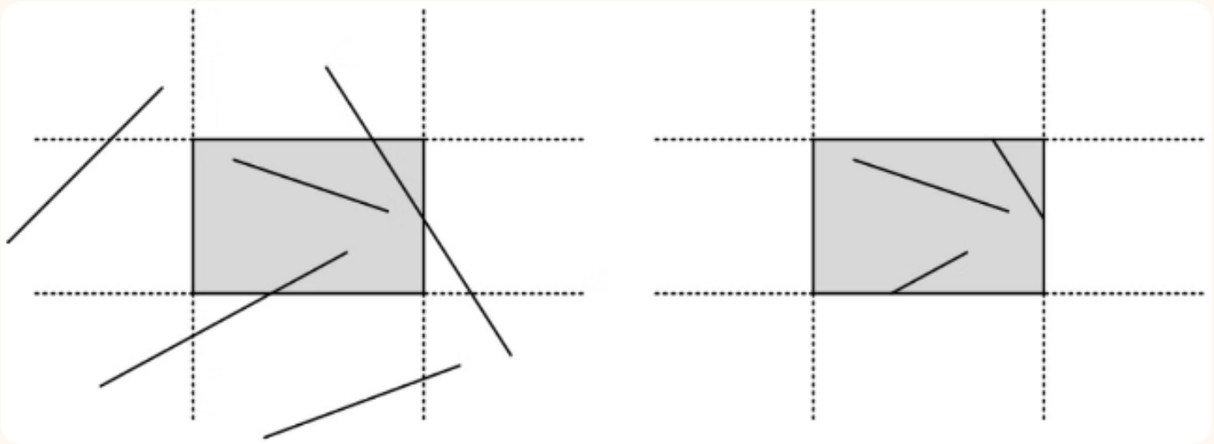
$x_{min} \leq x \leq x_{max}$ becomes $10 \leq 150 \leq 100$, which is false because 150 is greater than x_{max}=100.

Thus, the point Q(150,50) lies outside the clipping window and will be clipped (not displayed).

Transformations in OpenGL Clipping Operations

Line Clipping

Cohen-Sutherland line clipping is a widely-used algorithm for clipping lines in 2D space. It determines whether a line segment is visible, partially visible, or completely outside the clipping window and clips the portion that lies outside the window. The algorithm uses a divide-and-conquer approach to decide the visibility of a line segment by assigning **region codes** to endpoints and performing logical operations.



Transformations in OpenGL Clipping Operations

Line Clipping: Region Codes

Each endpoint of a line is assigned a **4-bit region code** based on its position relative to the clipping window. The bits represent the position of the point in relation to the top, bottom, right, and left edges of the clipping window.

Transformations in OpenGL Clipping Operations

Line Clipping: Region Codes

Each bit in the 4-bit region code corresponds to a position:

Bit 1 (Top): The point is above the window ($y > y_{max}$).

Bit 2 (Bottom): The point is below the window ($y < y_{min}$).

Bit 3 (Right): The point is to the right of the window ($x > x_{max}$).

Bit 4 (Left): The point is to the left of the window ($x<x_{min}$).

For each endpoint of the line segment, a region code is generated. The region code helps to identify whether the point is inside or outside the window and, if outside, which side of the window it lies.

Transformations in OpenGL Clipping Operations

Line Clipping: Steps of the Cohen-Sutherland Algorithm

1. **Assign region codes**:
   - For each endpoint of the line, assign a 4-bit region code based on its position relative to the clipping window (top, bottom, left, right).
2. **Check for trivial acceptance or rejection**:
   - If **both endpoints** are inside the window (region codes are 0000), accept the line without clipping.
   - If both endpoints are completely outside in the same region (logical AND of the region codes is not 0), reject the line.

Transformations in OpenGL Clipping Operations

Line Clipping: Steps of the Cohen-Sutherland Algorithm

3. Clip the line:

- If part of the line is inside and some of the part is outside, calculate where the line crosses the clipping window edges.
- Replace the endpoints with the intersection points and repeat the process until the line is fully inside or rejected.

Transformations in OpenGL Clipping Operations

Line Clipping: Example

Let's go through an example using the Cohen-Sutherland algorithm.

Clipping Window:

$x_{min} = 10$

$x_{max} = 100$

$y_{min} = 10$

$y_{max} = 100$

For a point P(x,y), the region code is determined as follows:

- (0,0,0,0): The point is inside the clipping window.
- (0,0,1,0): The point is to the right of the window.
- (0,1,0,0): The point is below the window.

Transformations in OpenGL Clipping Operations

Line Clipping: Example

Line Segment:

Let the line segment have endpoints P1(5,5) and P2(80,120).

1. Assign region codes:

- P1(5,5): This point is to the left of the window ($x < x_{min}$) and below the window ($y < y_{min}$).
  - Region code: 0101 (left, bottom).
- P2(80,120): This point is inside the horizontal boundaries but above the window ($y > y_{max}$).
  - Region code: 1000 (top).

Transformations in OpenGL Clipping Operations

Line Clipping: Example

Line Segment:

Let the line segment have endpoints P1(5,5) and P2(80,120).

2. Check for trivial acceptance/rejection:

- The region codes of P1 and P2 are 0101 and 1000, respectively.
- The **logical AND** of these codes is 0000, meaning there is no overlap in the region codes that indicate both points are in the same non-visible region, so the line is not trivially rejected.

Transformations in OpenGL Clipping Operations

Line Clipping: Example

Line Segment:

Let the line segment have endpoints P1(5,5) and P2(80,120).

3. Clip the line:

We need to find the intersection points where the line crosses the clipping window boundaries.

Transformations in OpenGL Clipping Operations

Line Clipping: Example

Line Segment:

Let the line segment have endpoints P1(5,5) and P2(80,120).

4. Calculate intersection with the bottom boundary:

- The bottom boundary is $y = y_{min} = 10$.
- The equation of the line between P1(5,5) and P2(80,120) is:

$$y - y_1 = \frac{y_2 - y_1}{x_2 - x_1}(x - x_1)$$

Plugging in the values:

$$y - 5 = \frac{120 - 5}{80 - 5}(x - 5)$$

Transformations in OpenGL Clipping Operations

Line Clipping: Example

Line Segment:

Let the line segment have endpoints P1(5,5) and P2(80,120).

Simplifying:

$$y - 5 = \frac{115}{75}(x - 5)$$

To find the intersection with y=10:

$$10 - 5 = \frac{115}{75}(x - 5) \quad \Rightarrow \quad x = 8.26$$

So, the new intersection point is P1′(8.26,10).

Transformations in OpenGL Clipping Operations

Line Clipping: Example

Line Segment:

Let the line segment have endpoints P1(5,5) and P2(80,120).

5. **Calculate intersection with the top boundary:**

   - The top boundary is $y = y_{\max} = 100$.
   - Using the same equation:

$$y - 5 = \frac{115}{75}(x - 5)$$

   For $y = 100$:

$$100 - 5 = \frac{115}{75}(x - 5) \quad \Rightarrow \quad x = 71.52$$

   So, the new intersection point is $P2'(71.52, 100)$.

Transformations in OpenGL Clipping Operations

Line Clipping: Example

Line Segment:

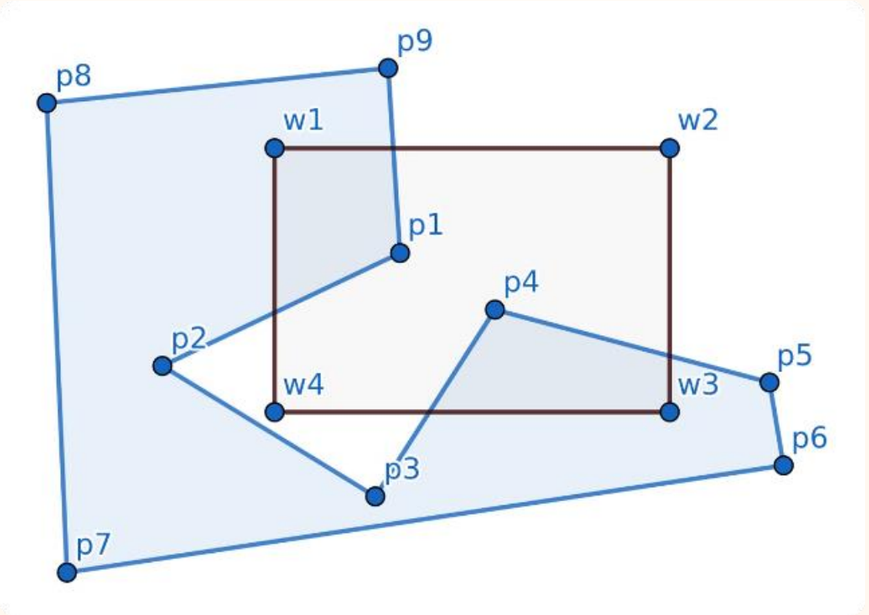
Let the line segment have endpoints P1(5,5) and P2(80,120).

6. **Final line segment**: The line segment is now clipped to lie within the clipping window, with new endpoints $P1'(8.26, 10)$ and $P2'(71.52, 100)$. This clipped line is visible and will be rendered within the clipping window.

Transformations in OpenGL Clipping Operations

Polygon Clipping – Sutherland – Hodgeman Polygon Clipping

The **Sutherland – Hodgman Polygon Clipping** algorithm is used to clip polygons (convex or concave) against a rectangular clipping window. It works by processing each edge of the polygon against each edge of the clipping window one at a time.



Transformations in OpenGL Clipping Operations

Polygon Clipping – Sutherland – Hodgeman Polygon Clipping

Steps of the Algorithm:

1. **Start with the polygon** and the clipping window.
2. **Clip the polygon edge by edge** against each boundary of the clipping window (left, right, top, bottom).
3. For each edge of the clipping window, process the vertices of the polygon:
   - **Four cases** arise for each vertex:
     1. **Inside** → **Inside**: Add the second vertex to the output list.
     2. **Inside** → **Outside**: Add the intersection point to the output list.
     3. **Outside** → **Inside**: Add the intersection point and the second vertex to the output list.
     4. **Outside** → **Outside**: No vertex is added to the output list.
4. After processing all edges, the resulting output is the clipped polygon.

Transformations in OpenGL Clipping Operations

Polygon Clipping – Sutherland – Hodgeman Polygon Clipping

Example

Let’s take a simple polygon with 4 vertices (forming a square) and a rectangular clipping window.

Initial Polygon Vertices:

- P1(20,30)
- P2(70,30)
- P3(70,60)
- P4(20,60)

Transformations in OpenGL Clipping Operations

Polygon Clipping – Sutherland – Hodgeman Polygon Clipping

Example: Clipping Window:

- Left boundary: x=30
- Right boundary: x=60
- Bottom boundary: y=40
- Top boundary: y=50

Transformations in OpenGL Clipping Operations

Polygon Clipping – Sutherland – Hodgeman Polygon Clipping

Example: Clipping Window:

1. **Clip against the left boundary x=30**:
    - P1(20,30) is outside, and P2(70,30) is inside. So, we add the intersection point (30,30) and P2(70,30).
    - P3(70,60) and P4(20,60): Similar clipping is done, adding necessary points and intersections. Resulting points: (30,60),(60,60),(60,50),(30,50).

Transformations in OpenGL Clipping Operations

Polygon Clipping – Sutherland – Hodgeman Polygon Clipping

Example: Clipping Window:

2. Clip against the right boundary x=60x = 60x=60:

- The new vertices are clipped again, and unnecessary portions outside the boundary are discarded.

3. Clip against the bottom and top boundaries:

- Follow the same logic to get the final clipped polygon.

Transformations in OpenGL Clipping Operations

Polygon Clipping – Sutherland – Hodgeman Polygon Clipping

Example: Clipping Window:

Final Clipped Polygon:

After all the clipping stages, the vertices of the resulting polygon will be those inside the clipping window.