Unit 3 – Transformations and Clipping

Transformation

Changes in orientation, size and shape of an object by changing the coordinate description, is known as **Geometric Transformation**.

Translation

To reposition an object from one coordinate position to another is known as **Translation**. We translate a two-dimensional point by adding **Translation Vector (Distance)** to the original coordinate position. The translation vector **(t_x, t_y)** is the distance by which the x-coordinate and the y-coordinate are to be translated.

If (x, y) is the original coordinate position, we add translation vector (t_x, t_y) to move the point to a new position (x', y').

$$x' = x + t_x \qquad y' = y + t_y$$

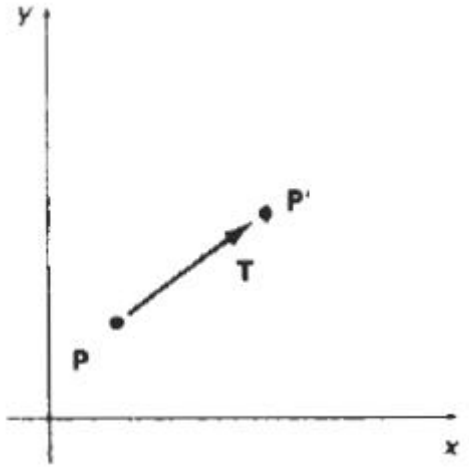
We can express the above translation equations as a single matrix equation by using column to represent coordinate positions and the translation vectors.

$$P = \begin{bmatrix} x \\ y \end{bmatrix} \quad P' = \begin{bmatrix} x' \\ y' \end{bmatrix} \quad T = \begin{bmatrix} t_x \\ t_y \end{bmatrix}$$

So, the two-dimensional translation equation in matrix form can be written as:

$$P' = P + T$$

Translation is a rigid body transformation that moves objects without changing the shape. Every point on the body is translated by the same amount.

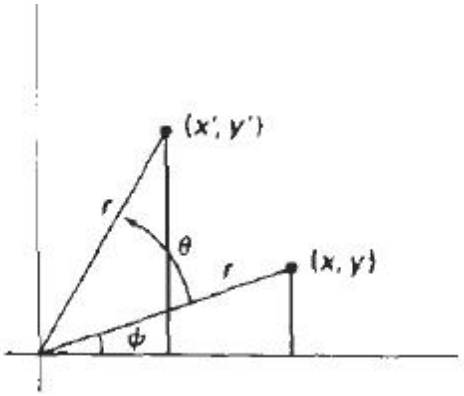


A straight line is translated by applying the transformation equation to each end point and redrawing the line again at the new endpoint coordinate position. Similarly, a polygon can be translated by adding the translation vector to each vertex (end point) and regenerating the polygon at the new set of vertex position. In case of curved objects, such as circle, we translate the centre coordinates and redraw the object at new position.

Rotation

To reposition an object along a circular path in xy plane is known as **Rotation**. To generate a rotation, we need a **Rotation Angle** and the position of the **Rotation Point**. Rotation Point **(x_r, y_r)** is the point about which the object has to be rotated and Rotation Angle θ is the angle by which the object has to be rotated about the rotation point. Positive values of rotation angle rotate the object in counter clockwise direction about the rotation point and negative values rotate the abject in clockwise direction.

We first determine the transformation equations for rotation of a point P when the rotation point (x_r, y_r) is at the origin.



In the figure, r, is the constant distance of the point from the origin, angle ϕ is the original angular position of the point from the horizontal and θ is the rotation angle.

The circle equation for polar coordinates gives:

$$x = x_r + r.\cos\phi$$

$$y = y_r + r.\sin\phi$$

Since (X_r, Y_r) is (0, 0)

$$x = r.\cos\phi$$

$$y = r.\sin\phi$$

Similarly, for the final angular position equation are:

$$x' = x_r + r.\cos(\phi + \theta)$$

$$y' = y_r + r.\sin(\phi + \theta)$$

Since (X_r, Y_r) is (0,0),

$$x' = r.\cos(\phi + \theta)$$

$$y' = r.\sin(\phi + \theta)$$

or,

$$x' = r.\cos\phi.\cos\theta - r.\sin\phi.\sin\theta$$

$$y' = r.\cos\phi.\sin\theta + r.\sin\phi.\cos\theta$$

or, $x' = x.\cos\theta - y.\sin\theta \quad y' = x.\sin\theta + y.\cos\theta$

This is the transformation equation to rotate a point at (x, y) through an angle θ about origin. The matrix form of rotation equation:

$$P = \begin{bmatrix} x \\ y \end{bmatrix} \quad P' = \begin{bmatrix} x' \\ y' \end{bmatrix} \quad R = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}$$

So, the two-dimensional rotation equation in matrix form can be written as:

$$P' = R \cdot P$$

The above rotation equation can be generalized for rotation of a point about any specified rotation point (x_r, y_r).

A straight line is rotated by applying the transformation equation to each end point and redrawing the line again at the new endpoint coordinate position. Similarly, a polygon can be rotated by applying the specified rotation angle and regenerating the polygon at the new set of vertex position. In case of curved objects, such as circle, we rotate the centre coordinates about the rotation point and redraw the object at new position.

Scaling

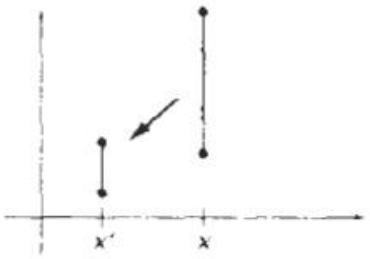
To change the size of an object is known as **Scaling**. Scaling can be performed by multiplying coordinate value (x, y) of each vertex by **Scaling Factor (S_x, S_y)** to produce transformed coordinate (x', y'). Scaling Factor S_x scales objects in the x direction and S_y in y direction.

$$x' = x.S_x$$
$$y' = y.S_y$$

The transformation equations in matrix form are:

$$P = \begin{bmatrix} x \\ y \end{bmatrix} \quad P' = \begin{bmatrix} x' \\ y' \end{bmatrix} \quad S = \begin{bmatrix} S_x & 0 \\ 0 & S_y \end{bmatrix}$$

Or, $P' = S.\ P$



Numbers more than 1 for scaling factors increase the size of the object whereas values less than 1 reduce the size of object. Specifying value of 1 for both S_x and S_y leaves the size of object unchanged. When S_x and S_y are same value, then it is **Uniform Scaling** and unequal values of S_x and S_y results in **Differential Scaling**.

Scaling Factors: $S_x = 2, S_y = 2$ Uniform Scaling

Scaling Factors: $S_x = 2, S_y = 1$ Differential Scaling

Scaling of a Square

"Objects transformed with the above equations are both scaled and repositioned, that is, translated. Scaling factors of less than 1 moves objects closer to the coordinate origin, whereas values greater than 1 move coordinate position away from origin."

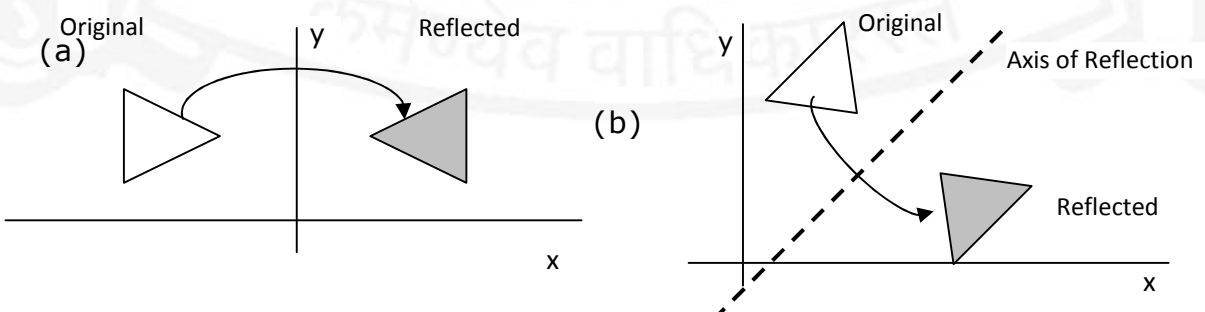
Figure shows a line scaled with scaling factors Sx = Sy = 0.5. It is reduced in size and moves closer to the origin. Both the line length and the distance from the origin are reduced by 1/2.

Polygons are scaled by applying transformation to each point and then, redrawing the polygon using transformed points. Uniform scaling of a circle is done by simply changing the radius and redrawing using new radius.

Reflection

A **Reflection** is a transformation that produces a mirror image of an object. The mirror image for a two-dimensional reflection is generated relative to an **Axis of Reflection** by rotating the object 180° about the reflection axis.

We can choose an axis of reflection in the xy plane or perpendicular to the xy plane. When the reflection axis is a line in the xy plane, the rotation path about this axis is in a plane perpendicular to the xy plane. For reflection axes that are perpendicular to the xy plane, the rotation path is in the *xy* plane. Following are examples of some common reflections.



Reflection of an object (a)about the y-axis(b)about the axis of reflection y=x

Shear

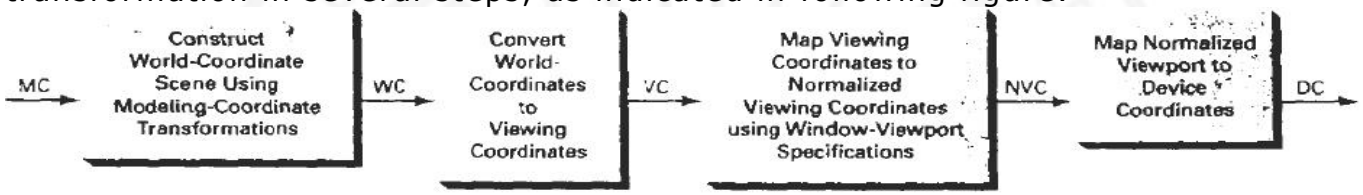
A transformation that changes the shape of an object such that the transformed shape appears as if the object was made of layers that had been caused to slide over each other, is called a **Shear**. Two common shearing transformations are those that shift coordinate x value and those that shift y values.

(a) (b)

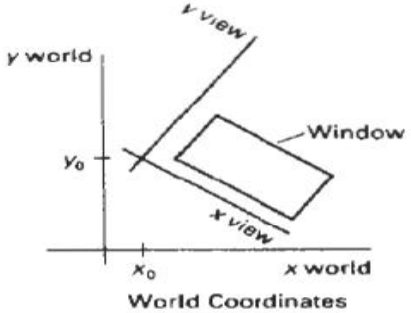
An object after (a) x-direction shear and (b) y-direction shear

Viewing Pipeline

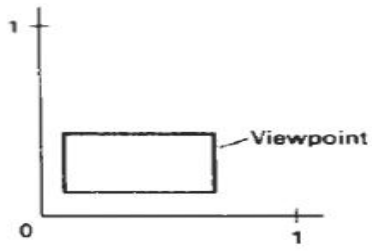
Some graphics packages that provide window and viewport operations allow only standard rectangles, but a more general approach is to allow the rectangular window to have any orientation. In this case, we carry out the viewing transformation in several steps, as indicated in following figure.



First, we construct the scene in world coordinates using the output primitives and attributes. Next, to obtain a particular orientation for the window, we can set up a two-dimensional viewing-coordinate system in the world-coordinate plane, and define a window in the viewing-coordinate system. The viewing coordinate reference frame is used to provide a method for setting up arbitrary orientations for rectangular windows. Once the viewing reference frame is established, we can transform descriptions in world coordinates to viewing coordinates. We then define a viewport in normalized coordinates (in the range from O to 1) and map the viewing-coordinate description of the scene to normalized coordinates. At the final step, all parts of the picture that he outside the viewport are clipped, and the contents of the viewport are transferred to device coordinates. Following Figure illustrates a rotated viewing-coordinate reference frame and the mapping to normalized coordinates.



World Coordinates



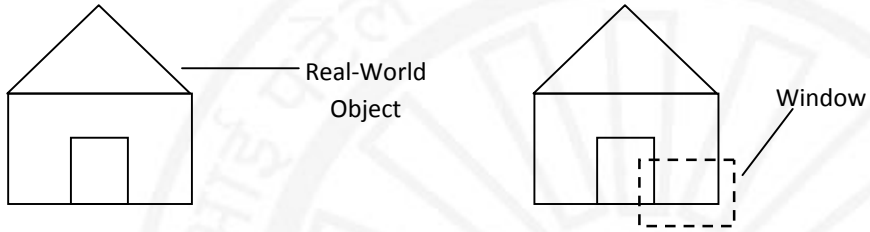
Normalized Device Coordinates

By changing the position of the viewport, we can view objects at different positions on the display area of an output device. Also, by varying the size of viewports, we can change the size and proportions of displayed objects. We achieve zooming effects by successively mapping different-sized windows on a fixed-size viewport. As the windows are made smaller, we zoom in on some part of a scene to view details that are not shown with larger windows. Similarly, more overview is obtained by zooming out from a section of a scene with successively larger windows. Panning effects are produced by moving a fixed-size window across the various objects in a scene.

Window

A rectangular box to view only a part of a real-world object we to draw on the screen, is called a ***Window***.

In other words, a world-coordinate area selected for display on a display device is called a ***Window***.

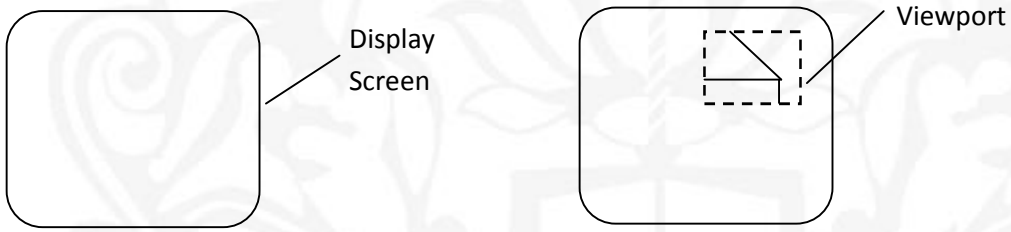


Window to view only part of an object

Viewport

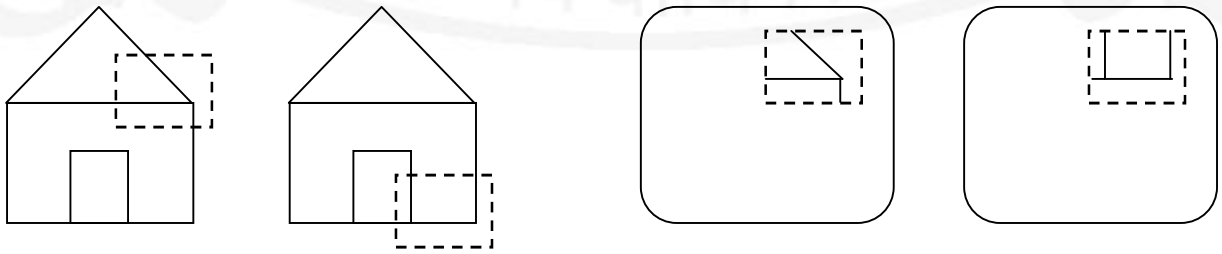
A rectangular region of screen selected for displaying an object or part of an object, which has been specified in a window, is called a ***Viewport***.

In other words, an area on a display device to which a window is mapped is called a ***Viewport***.

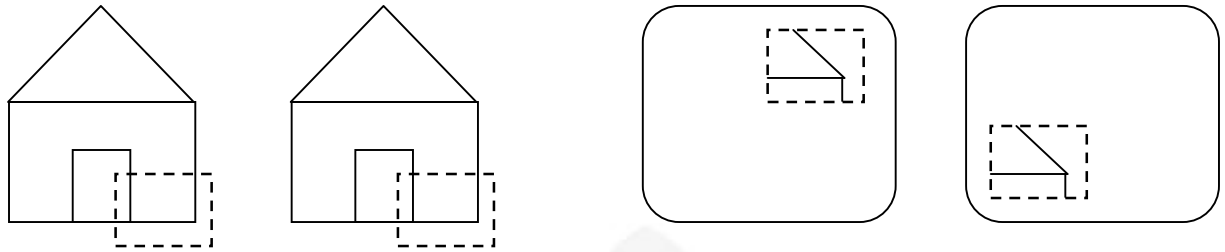


Viewport to define the part of the screen to be

A window defines what is to be viewed, whereas a viewport defines where it is to be displayed. When window is changed, we see a different part of object at same position of display device. When viewport is changed, we see the same object on different place on the display device.



Different Windows, same Viewport



Same Windows, different Viewports

Viewing Transformation/Window-to-Viewport Transformation

The mapping of a part of an object from world coordinate system to device or screen coordinate system is known as ***Viewing Transformation*** or ***Window-to-Viewport Transformation***.

Window-to Viewport Coordinate Transformation

Window-to-viewport transformation is done by maintaining the same relative placement of objects in device coordinates as they had been in world coordinates. For example, if a coordinate position is at the center of the viewing window, then it will be displayed at the center of the viewport.

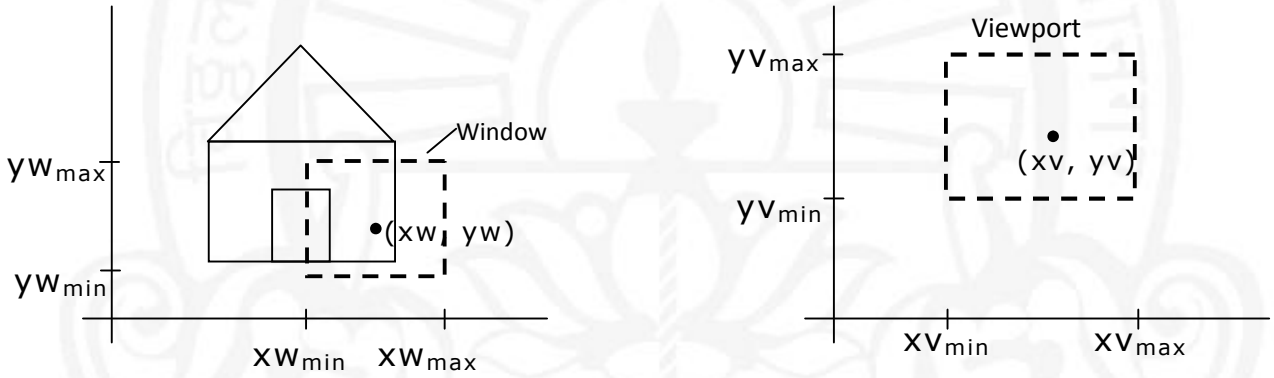


Figure shows a point at position (xw, yw) in a window mapped to viewport coordinates (xv, yv) so that the relative positions in the two areas are the same.

Figure shows the window-to-viewport mapping. A point at position (xw, yw) in window is mapped to position (xv, yv) in the viewport. To maintain the same relative placement in viewport, we require:

$$\frac{xv - xv_{min}}{xv_{max} - xv_{min}} = \frac{xw - xw_{min}}{xw_{max} - xw_{min}} \text{ and,}$$

$$\frac{yv - yv_{min}}{yv_{max} - yv_{min}} = \frac{yw - yw_{min}}{yw_{max} - yw_{min}}$$

Solving these expressions for the viewport position (xv, yv), we have

$$xv = xv_{min} + (xw - xw_{min}).S_x$$

and, $yv = yv_{min} + (yw - yw_{min}).S_y$

where the scaling factors (S_x, S_y) are:

$$S_x = \frac{xv_{max} - xv_{min}}{xw_{max} - xw_{min}}$$

$$S_y = \frac{yv_{max} - yv_{min}}{yw_{max} - yw_{min}}$$

If the scaling factors, S_x and S_y are same, relative proportions are maintained. Otherwise, world objects will be stretched or contracted in either x or y direction, that is, its shape will be distorted, when displayed on the output device.

Clipping

Generally, any procedure that identifies those portions of a picture that are either inside or outside of a specified region of space is referred to as a **clipping algorithm**, or simply **clipping**. The region against which an object is to clip is called a clip window.

Applications of clipping

- Extracting part of a defined scene for viewing
- Identifying visible surfaces in three-dimensional views
- Antialiasing line segments or object boundaries
- Creating objects using solid-modeling procedures
- Displaying a multiwindow environment
- Drawing and painting operations that allow parts of a picture to be selected for copying, moving, erasing, or duplicating.

Point Clipping

Assuming that the clip window is a rectangle in standard position, we save a point P = (x, y) for display if the following inequalities are satisfied:

$$XW_{min} \le X \le XW_{max} \quad \text{and} \quad YW_{min} \le Y \le YW_{max}$$

Where the edges of the clip window (XW_{min},XW_{max},YW_{min},YW_{max}) can be either the world-coordinate window boundaries or viewport boundaries. If any one of these four inequalities is not satisfied, the point is clipped (not saved for display).

Although point clipping is applied less often than line or polygon clipping, some applications may require a point clipping procedure. For example, point clipping can be applied to scenes involving explosions or sea foam that is modeled with particles (points) distributed in some region of the scene.

Line Clipping

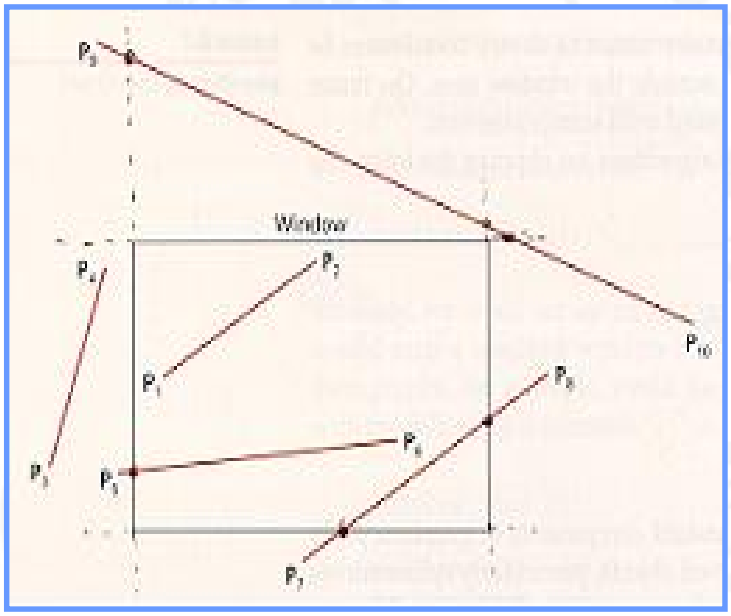
A line clipping procedure involves several parts. First, we can test a given line segment to determine whether it lies completely inside the clipping window. If it does not, we try to determine whether it lies completely outside the window. Finally, if we cannot identify a line as completely inside or completely outside, we must perform intersection calculations with one or more clipping boundaries. We process lines through the "inside-outside'' tests by checking the line endpoints.

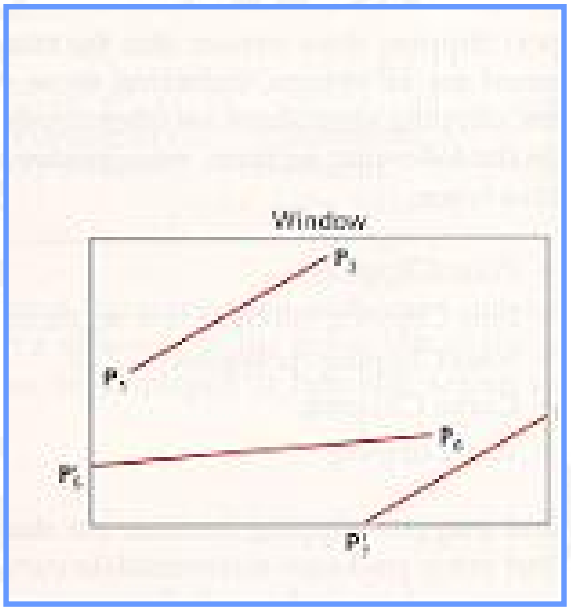
A line with both endpoints inside all clipping boundaries, such as the line from P_1 to P_2 is saved. A line with both endpoints outside any one of the clip boundaries (line P_3P_4 in Figure) is outside the window. All other lines cross one or more clipping boundaries, and may require calculation of multiple intersection points. To minimize calculations, we try to devise clipping algorithms that can efficiently identify outside lines and reduce intersection calculations.

For a line segment with endpoints (x1, y1) and (x2, y2) and one or both endpoints outside the clipping rectangle, the parametric representation

$$x = x_1 + u(x_2-x_1)$$
$$y = y_1 + u(y_2-y_1), \qquad 0 \le u \le 1$$

could be used to determine values of parameter u for intersections with the clipping boundary coordinates. If the value of u for an intersection with a rectangle boundary edge is outside the range 0 to 1, the line does not enter the interior of the window at that boundary. If the value of u is within the range from 0 to 1, the line segment does indeed cross into the clipping area. This method can be applied to each clipping boundary edge in turn to determine whether any part of the line segment is to be displayed. Line segments that are parallel to window edges can be handled as special cases.

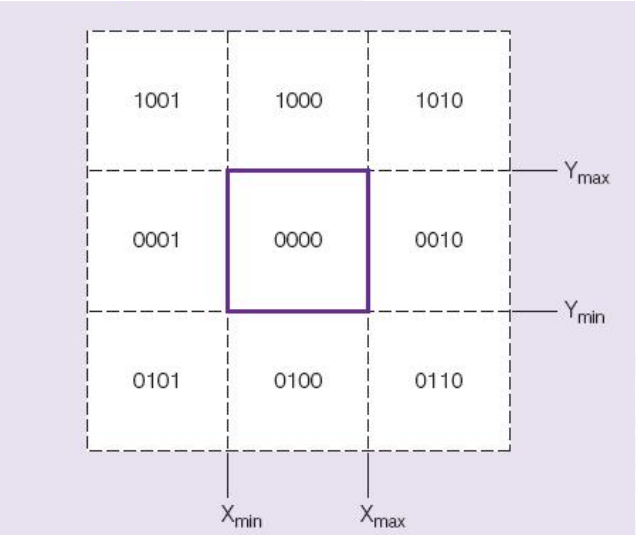




(Before Clipping) **(After Clipping)**

Cohen-Sutherland Line Clipping

This is one of the oldest and most popular line-clipping procedures. Generally, the method speeds up the processing of line segments by performing initial tests that reduce the number of intersections that must be calculated. Every line endpoint in a picture is assigned a four-digit binary code, called a region code that identifies the location of the point relative to the boundaries of the clipping rectangle. Regions are set up in reference to the boundaries as shown in Figure.



Each bit position in the region code is used to indicate one of the four relative coordinate positions of the point with respect to the clip window: to the left, right, top, or bottom. By numbering the bit positions in the region code as 1 through 4 from right to left, the coordinate regions can be correlated with the bit positions as

bit 1: left
bit 2: right
bit 3: below
bit 4: above

A value of 1 in any bit position indicates that the point is in that relative position; otherwise, the bit position is set to 0. If a point is within the clipping rectangle, the region code is 0000. A point that is below and to the left of the rectangle has a region code of 0101.

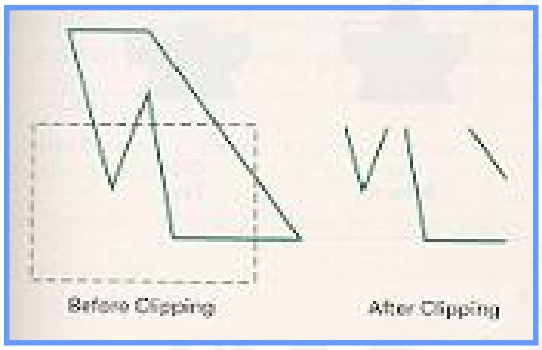
Bit values in the region code are determined by comparing endpoint coordinate values (x, y) to the clip boundaries. Bit 1 is set to 1 if $x < XW_{min}$. The other three bit values can be determined using similar comparisons. For languages in which bit manipulation is possible, region-code bit values can be determined with the following two steps: (1) Calculate differences between endpoint coordinates and clipping boundaries. (2) Use the resultant sign bit of each difference calculation to set the corresponding value in the region code. Bit 1 is the sign bit of $X-XW_{min}$ bit 2 is the sign bit of $XW_{max}-X$; bit 3 is the sign bit of $Y-YW_{min}$; and bit 4 is the sign bit of $YW_{max}-Y$.

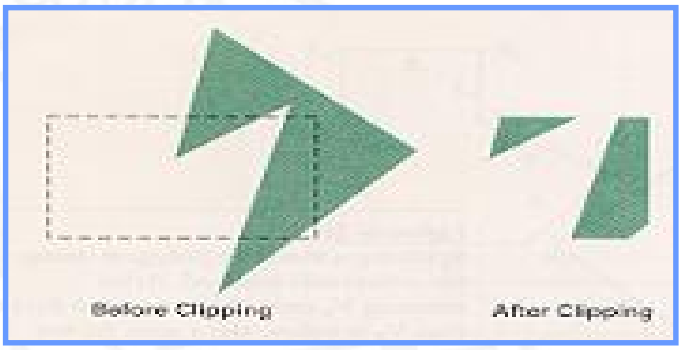
Once we have established region codes for all line endpoints, we can quickly determine which lines are completely inside the clip window and which are clearly outside. Any lines that are completely contained within the window boundaries have a region code of 0000 for both endpoints, and we trivially accept these lines.

Any lines that have a 1 in the same bit position in the region codes for each endpoint are completely outside the clipping rectangle, and we trivially reject these lines. We would discard the line that has a region code of 1001 for one endpoint and a code of 0101 for the other endpoint. Both endpoints of this line are left of the clipping rectangle, as indicated by the 1 in the first bit position of each region code. A method that can be used to test lines for total clipping is to perform the logical *and* operation with both region codes. If the result is not 0000, the line is completely outside the clipping region.

Polygon Clipping

A polygon boundary processed with a line clipper may be displayed as a series of unconnected line segments, depending on the orientation of the polygon to the clipping window. What we really want to display is a bounded area after clipping, as in Figure. For polygon clipping, we require an algorithm that will generate one or more closed areas that are then scan converted for the appropriate area fill. The output of a polygon clipper should be a sequence of vertices that defines the clipped polygon boundaries.

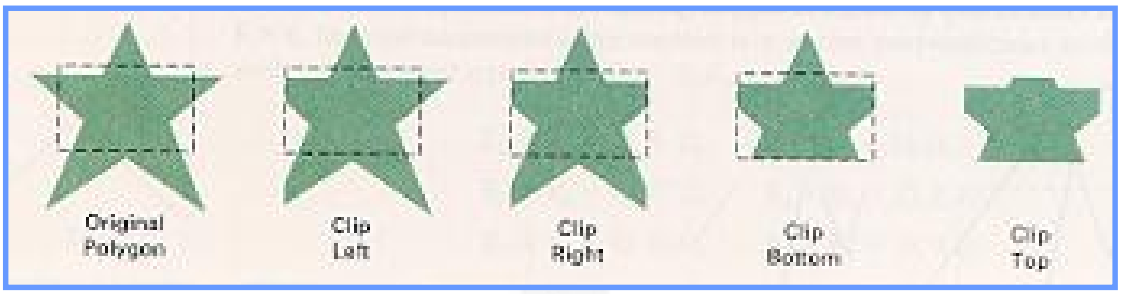




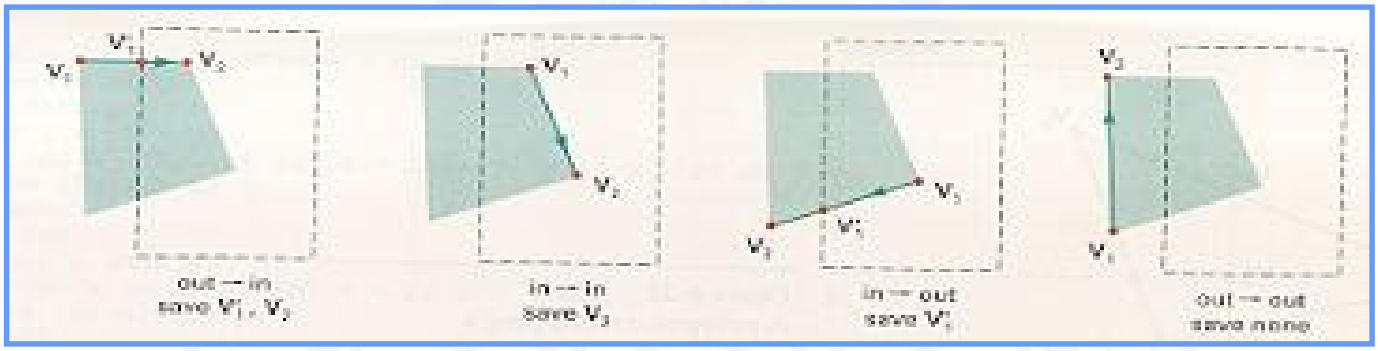
Sutherland Hodgeman Polygon Clipping Algorithm

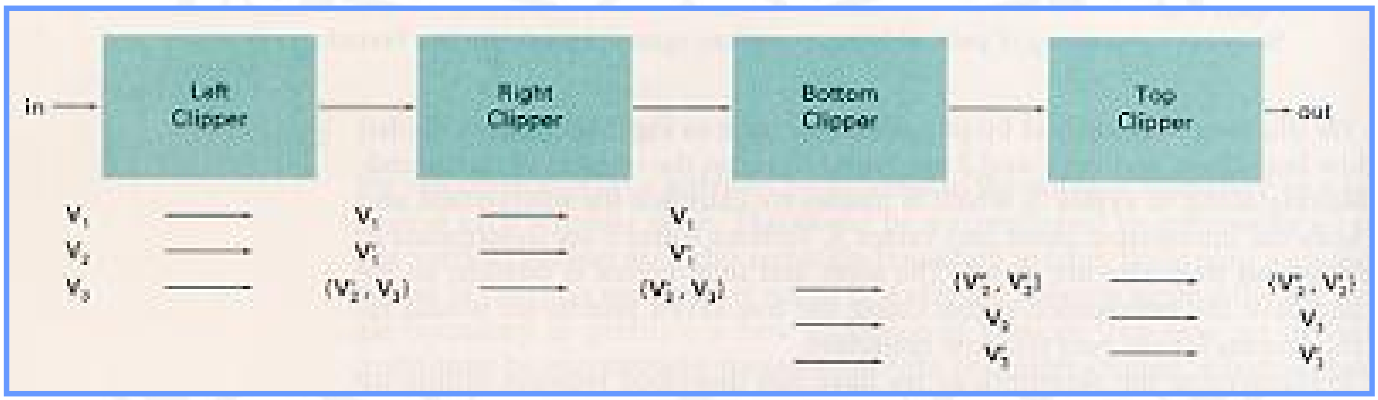
We can correctly clip a polygon by processing the polygon boundary as a whole against each window edge. This could be accomplished by processing all polygon vertices against each clip rectangle boundary in turn. Beginning with the initial set of polygon vertices, we could first clip the polygon against the left rectangle boundary to produce a new sequence of vertices. The new set of vertices could then k successively passed to a right boundary clipper, a bottom boundary clipper, and a top boundary clipper. At each step, a new sequence of output vertices is generated and passed to the next window boundary clipper. There are four possible cases when processing vertices in sequence around the perimeter of a polygon.

As each pair of adjacent polygon vertices is passed to a window boundary clipper, we make the following tests: (1) If the first vertex is outside the window boundary and the second vertex is inside, both the intersection point of the polygon edge with the window boundary and the second vertex are added to the output vertex list. (2) If both input vertices are inside the window boundary, only the second vertex is added to the output vertex list. (3) If the first vertex is inside the window boundary and the second vertex is outside, only the edge intersection with the window boundary is added to the output vertex list. (4) If both input vertices are outside the window boundary, nothing is added to the output list. Once all vertices have been processed for one clip window boundary, the output list of vertices is clipped against the next window boundary.



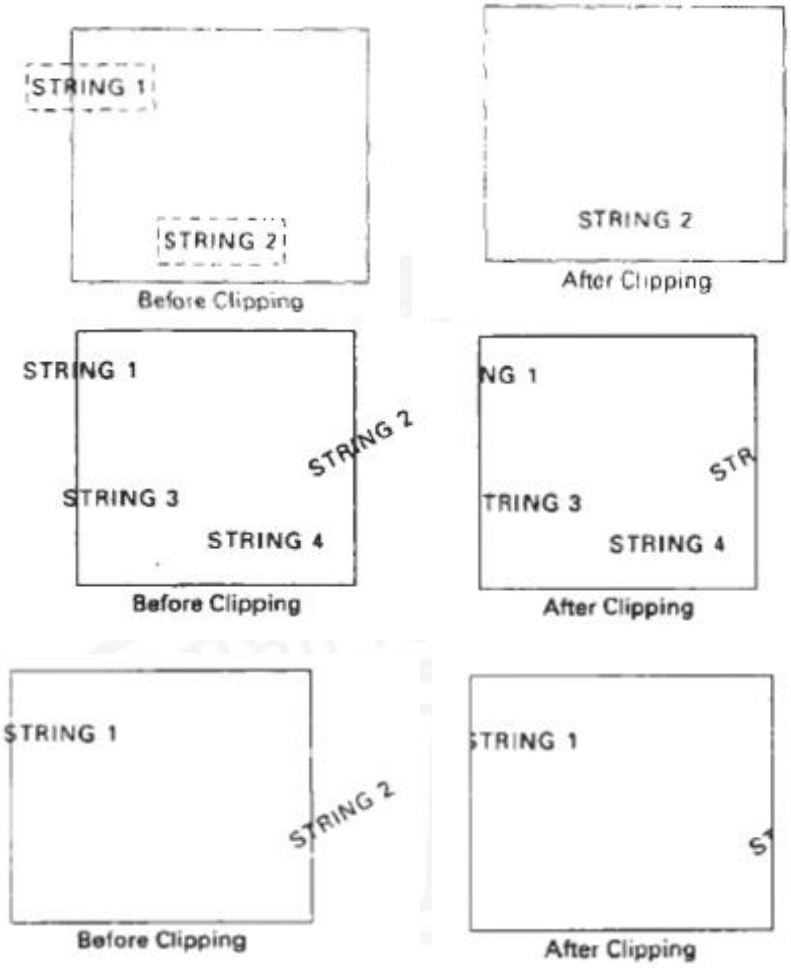
We illustrate this method by processing the area in figure against the *left* window boundary. Vertices I and 2 are found to be on the outside of the boundary. Moving along to vertex 3, which is inside, we calculate the intersection and save both the intersection point and vertex 3. Vertices 4 and 5 are determined to by inside, arid they also are saved. The sixth and final vertex is outside, so we find and save the intersection point. Using the five saved points, we would repeat the process for the next window boundary.





Text Clipping

There are several techniques that can be used to provide text clipping in a graphics package. The clipping technique used will depend on the methods used to generate characters and the requirements of a particular application. The simplest method for processing character strings relative to a window boundary is to use the all-or-none string-clipping strategy shown in Figure. If all of the string is inside a clip window, we keep it. Otherwise, the string is discarded. This procedure is implemented by considering a bounding rectangle around the text pattern. The boundary positions of the rectangle are then compared to the window boundaries, and the string is rejected if there is any overlap. This method produces the fastest text clipping. An alternative to rejecting an entire character string that overlaps a window boundary is to use the all-or-none character-clipping strategy. Here we discard only those characters that are not completely inside the window.



In this case, the boundary limits of individual characters are compared to the window. Any character that either overlaps or is outside a window boundary is clipped. A final method for handling text clipping is to clip the components of individual characters. We now treat characters in much the same way that we treated lines. If an individual character overlaps a clip window boundary, we clip off the parts of the character that are outside the window. Outline character fonts formed with line segments can be processed in this way using a line clipping algorithm. Characters defined with bit maps would be clipped by comparing the relative position of the individual pixels in the character grid patterns to the clipping boundaries.

Compiled By: Mr. Navtej Bhatt, Lecturer,

BCA Department, V.P. & R.P.T.P. Science College, VV Nagar

Class: BCA SEM V

Subject: US05CBCA02 – Computer Graphics

Declaration: This material is developed only for the reference for lectures in the classrooms. Students are required library reading for more study. This study material compiled from Book "Computer Graphics by Donald Hearn & M. Pauline Baker, PHI, 1995