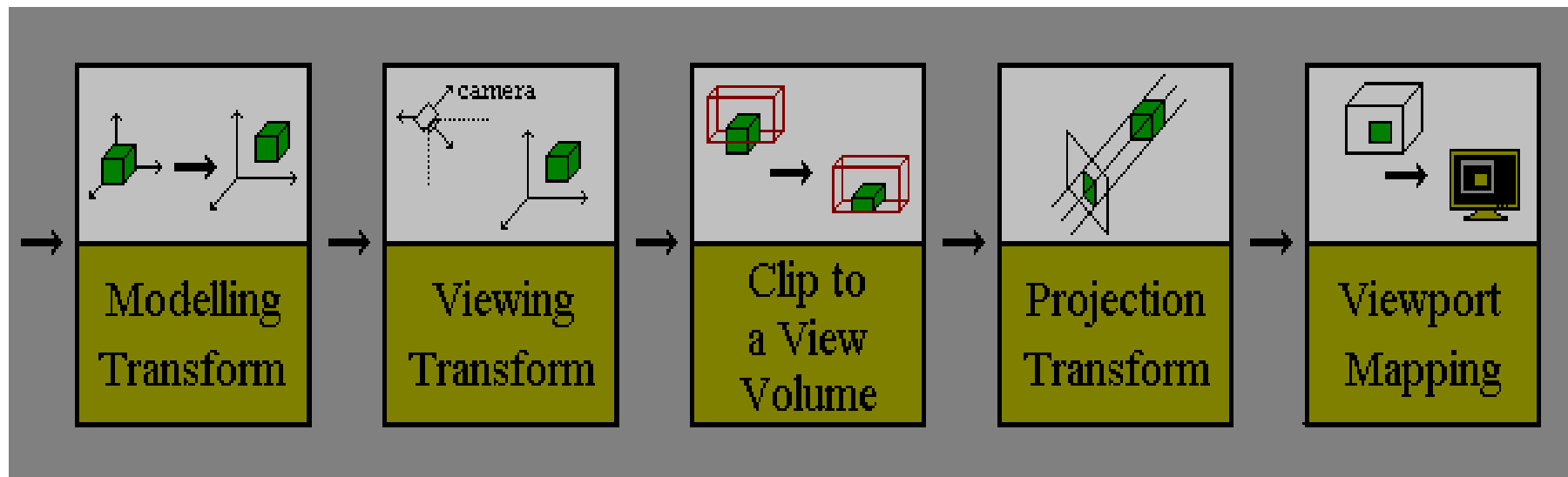


**2D Viewing:
Windows & Viewports
Clipping Operations**

A standard viewing pipeline

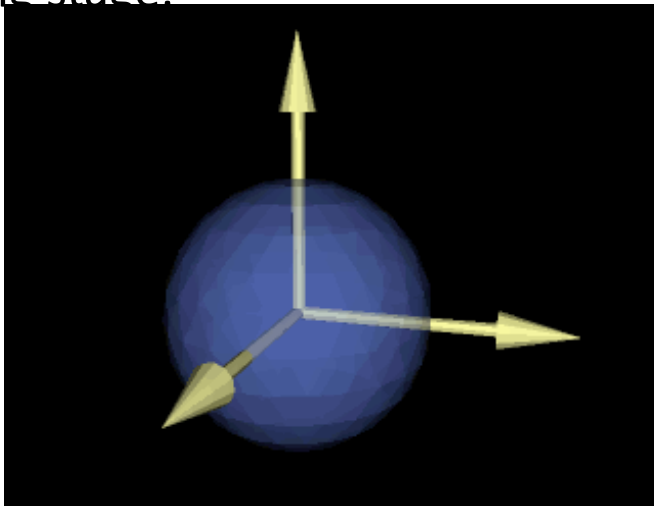


The viewing pipeline is a group of processes common from wireframe display through to near photo-realistic image generation, and is basically concerned with transforming objects to be displayed from specific viewpoint and removing surfaces that cannot be seen from this viewpoint.

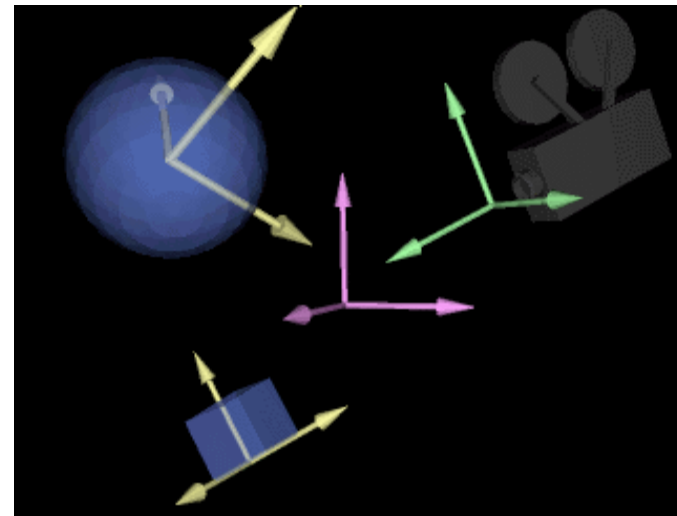
The input to the viewing pipeline is a list of objects with their points defined in their own local coordinate system, and the position and orientation of the viewpoint. The output is the same list of objects with their points defined in a two dimensional screen coordinate system.

Modeling Transformation

A typical rendering system will start out with each object being defined in its own **local coordinate system**. The origin may be some convenient point in the object, for example the center of a sphere. Using a local coordinate system gives an object independence from any other objects in a scene and aids the object modeling stage.



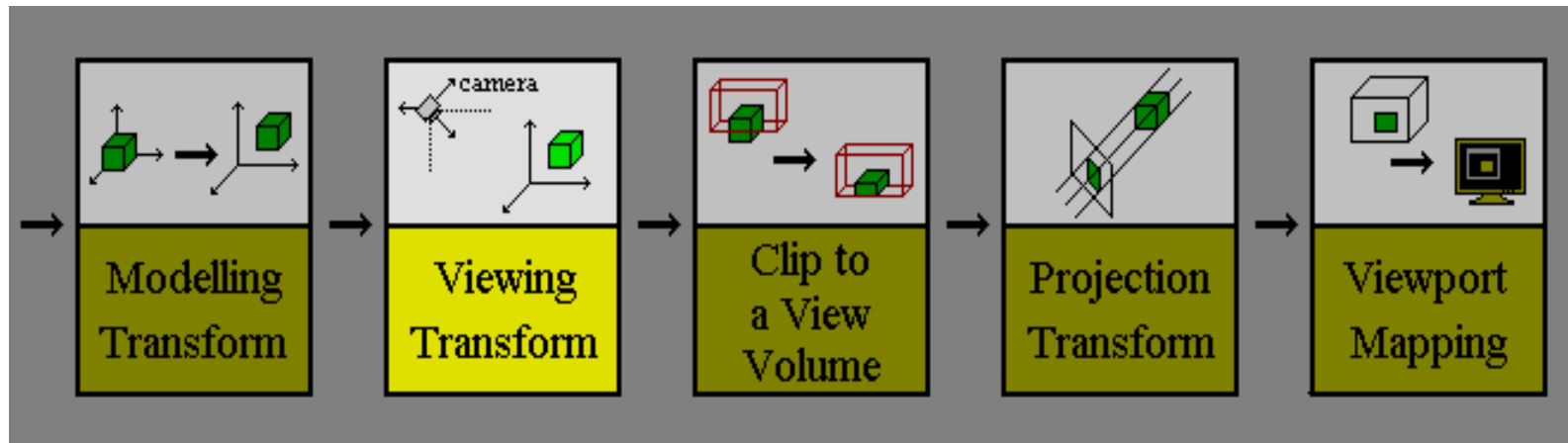
A sphere displayed with its own local coordinate system axes.



Objects are positioned against the world coordinate system (in pink).

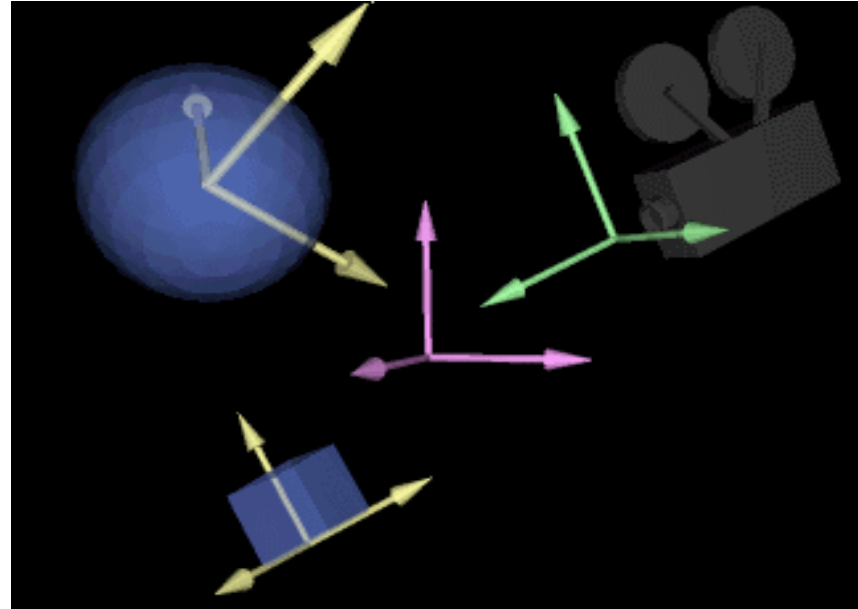
The first stage of the viewing pipeline is to convert each object from its local coordinate system into the **world coordinate system** (*modelling transformation*)

Viewing Transformation



This is the second stage of the pipeline. At this stage the objects are transformed from the world coordinate system to the view reference coordinate system (coordinate system of the camera). However, before we explain how to perform this transformation, we need to understand how a **camera is modeled** in computer graphics.

Camera Model



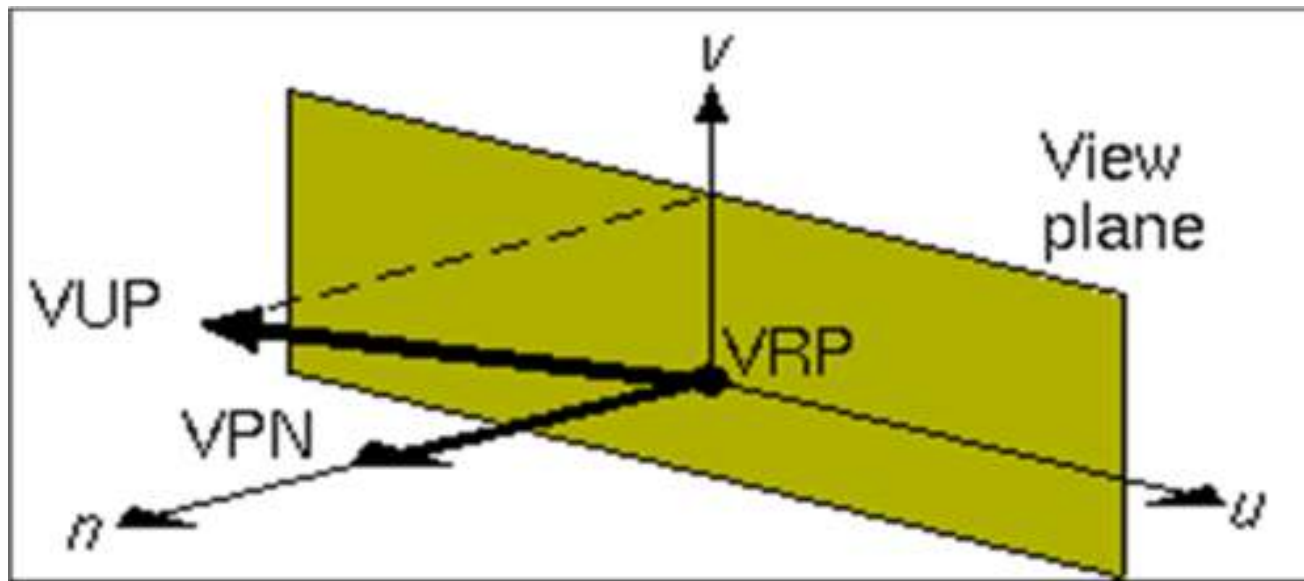
The green coordinate system represent the camera).

Once the scene is constructed, the position of the camera is defined in order to generate a picture of the scene. The definition of the camera is an important component with in a rendering package. Several parameters such as its position, orientation, position and size of the film etc need to be defined in order to define a complete **camera model**.

Camera Model

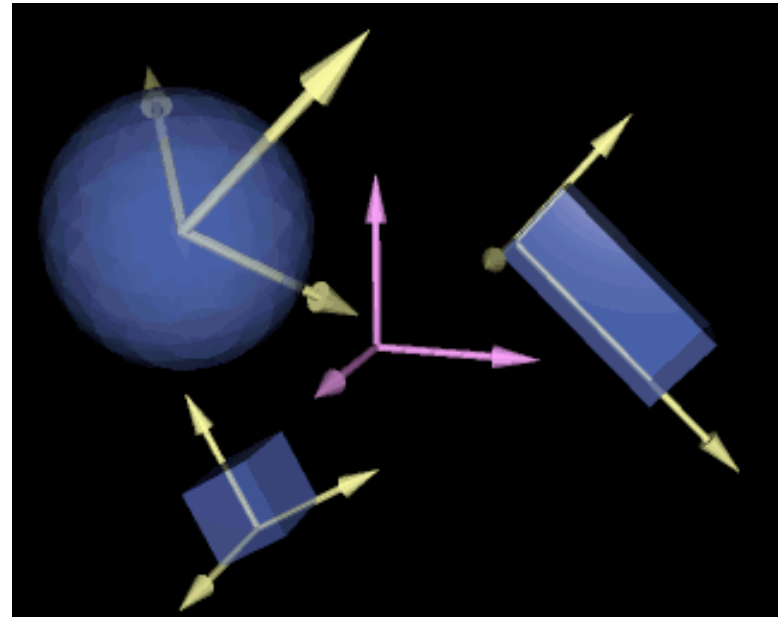
We need to define the following basic features to model a virtual camera :

- (i) **Viewing Plane** : This defines the position of the film. Objects are finally projected onto this plane to generate a 2D image.
- (ii) **View Reference Coordinate System** : This enable the user to define the size and the orientation of the viewing window (film).



View Transformation

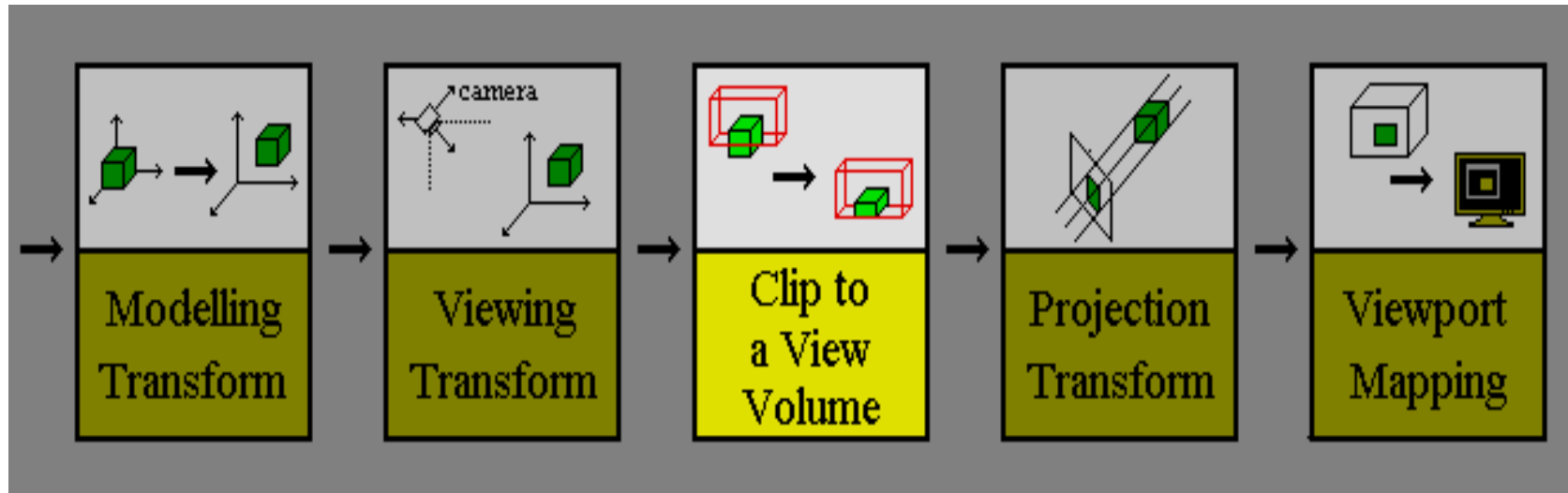
Objects seen from the camera.



Once the camera model is defined, the scene is converted into the coordinate system of the camera, which is referred to as the **View Reference Coordinate (VRC) System**. This conversion is performed using a (**view transformation**) matrix. After this conversion, the objects are positioned as if they are being looked at from the virtual camera.

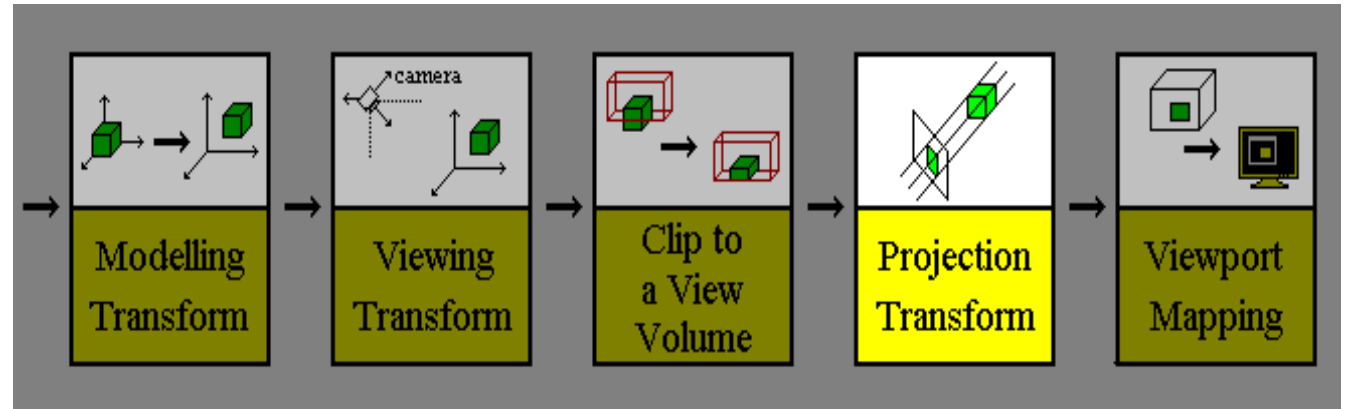
From this view, the polygons that are outside the view of the camera are removed. This is done by creating a view volume which defines the area of interest and then **clipping** the polygon against this view volume. The polygons which can not be seen from the camera, such as the polygons at the back of an object, is also not considered for further processing (**culling**).

Clipping to View Volumes

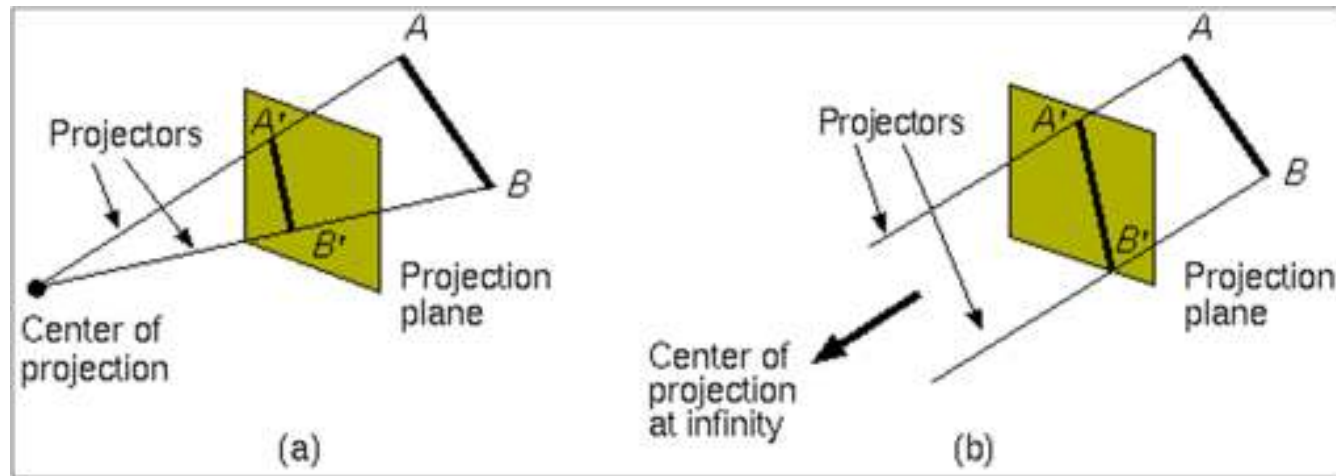


This is the third stage of the viewing pipeline. At this stage, the objects are clipped against the viewing volume.

Projections



The projection is defined by a set of straight projections rays, emanating from a *centre of projection* (PRP), which pass through the vertices of the objects in a 3D scene and then intercept a projection plane to form a projection.

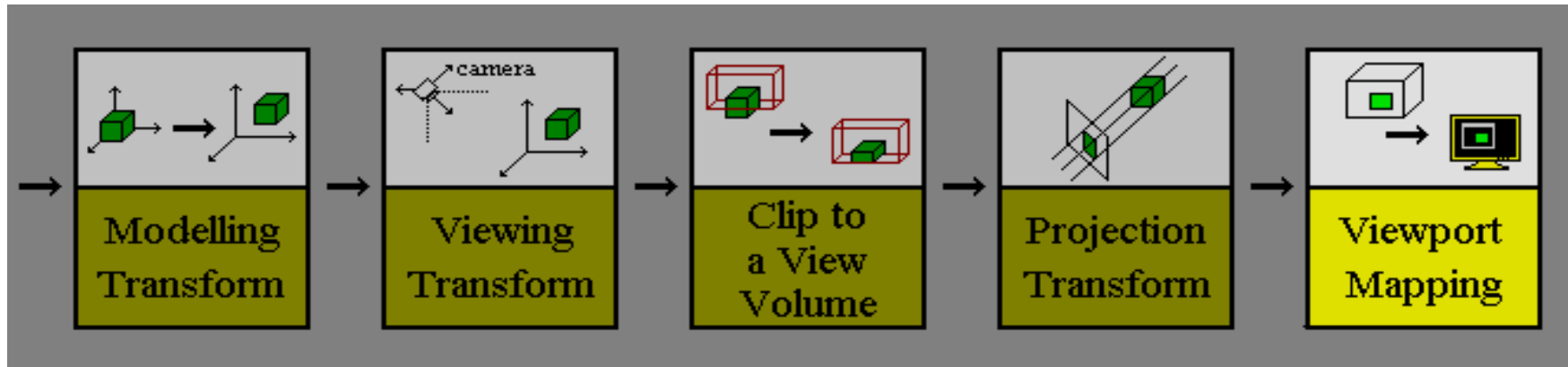


Perspective projection

Parallel projection

For a perspective projection we specify its *center of projection*, and for a parallel projection we specify its *direction of projection* (DOP).

Mapping to a Viewport



Mapping to a viewport occurs after the Projection transformation. It maps from a canonical view-plane to the actual screen coordinates of the display device. The transformation used for this depends on the canonical view volume used. Here the canonical parallel-projection cube is used.

The transformation is a three step process:

1. Translate the view volume such that its corner $(-1, -1, -1)$ becomes the origin
2. Scale the volume to the size of the viewport
3. Translate to the lower-left corner of the 3D viewport

2D Rendering Pipeline

3D Primitives



2D Primitives



Clipping



Viewport
Transformation



Scan
Conversion



Image

Clip portions of geometric primitives residing outside the window

Transform the clipped primitives from screen to image coordinates

Fill pixels representing primitives in screen coordinates

Clipping

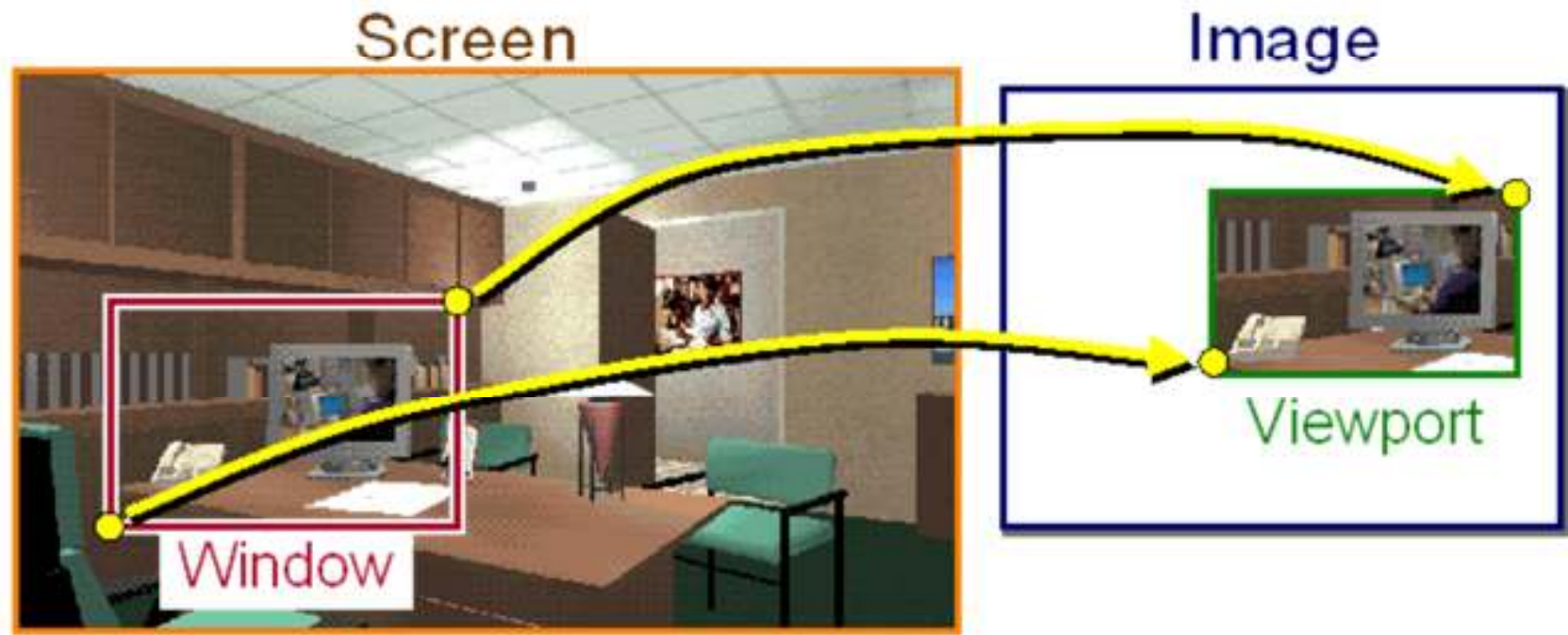
- Avoid drawing parts of primitives outside window
 - Window defines part of scene being viewed
 - Must draw geometric primitives only inside window



Screen Coordinates

Viewport Transformation

- Transform 2D geometric primitives from screen coordinate system (normalized device coordinates) to image coordinate system (pixels)

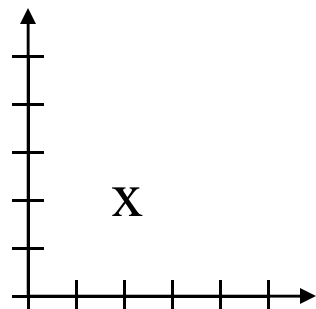


Windows & Viewports

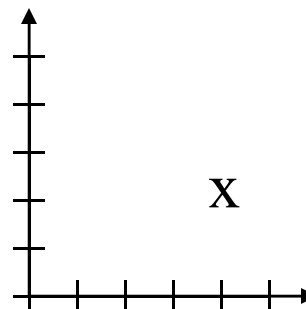
- A world-coordinate area selected for display is called a *window*.
- An area on a display device to which a window is mapped is called a *viewport*.
- A *viewing transformation* maps a window onto a viewport.

Moving Data vs. Moving Coordinates

(A)



(B)



Question: In going from (A) to (B), did the 'x' move two to the right, or did the y-axis move two to the left?

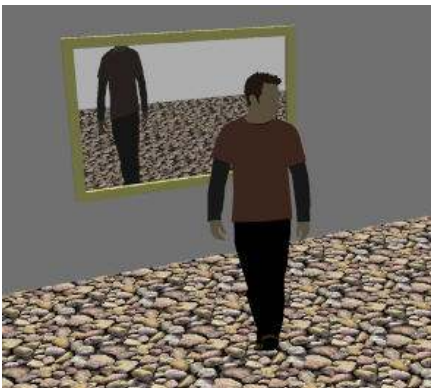
Answer: No Difference Between the Two

Translating points by (DX, DY) is equivalent to translating the coordinate system by $(-DX, -DY)$

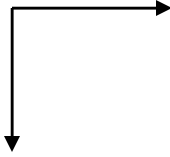
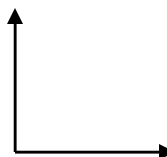
Similarly, rotating points by F is equivalent to rotating the coordinate system by $-F$

Scaling points by S is equivalent to scaling the coordinate system by $1/S$

Q: What about reflection?



2D Coordinate System Shifts

Task: convert from  to 

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & -1 & h \\ 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} \text{translate} \\ 1 & 0 & 0 \\ 0 & 1 & h \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} \text{reflect} \\ 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & -1 & h \\ 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} \text{reflect} \\ 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} \text{translate} \\ 1 & 0 & 0 \\ 0 & 1 & -h \\ 0 & 0 & 1 \end{pmatrix}$$

*note
sign*

Window to Viewport Transformations

The mapping from world coordinates to screen coordinates is another example of a linear transformation. How can this linear transformation be specified? One way is for the programmer to provide a transformation matrix explicitly. A simpler way (from the point of view of the programmer, anyway) is for the programmer to specify matching rectangular windows - a world-coordinate window and a screen coordinate viewport.

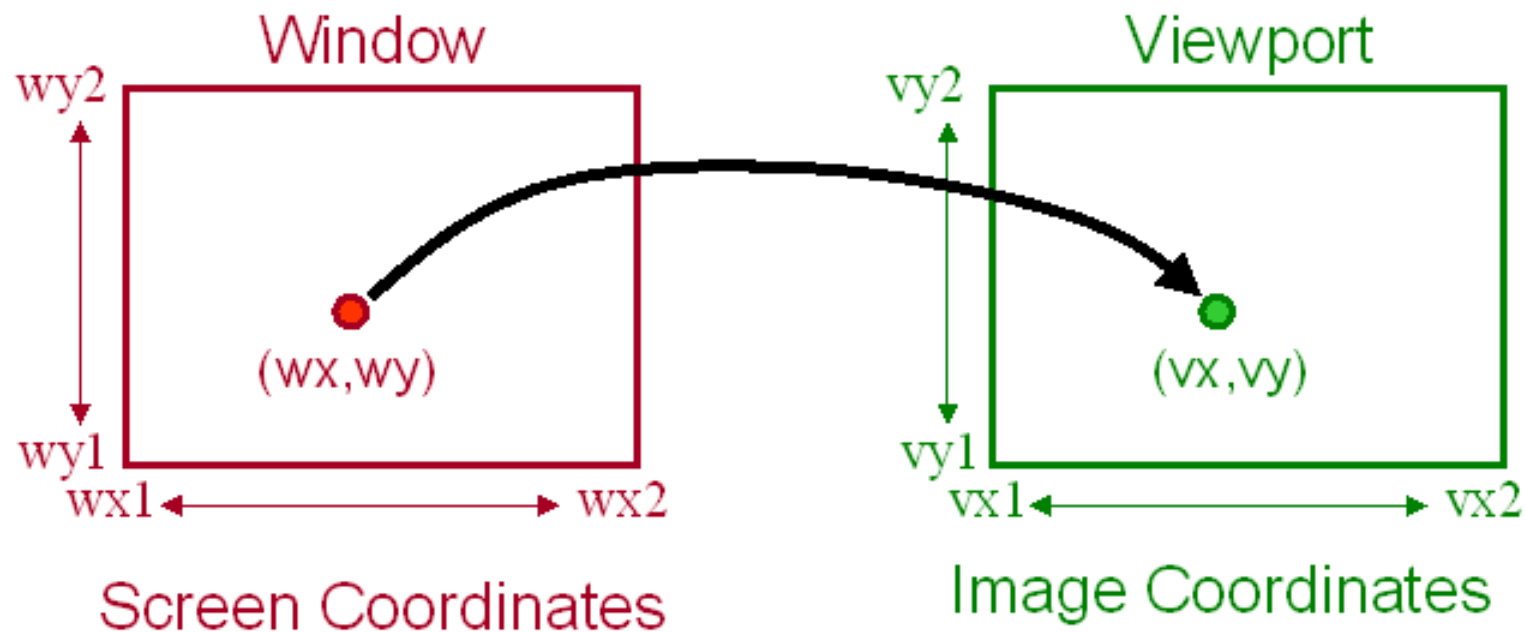
The final transformation matrix can be determined by composing the matrices for the following transformations:

1. Translate the world-coordinate window to the origin.
2. Rotate the window to match the orientation of the screen window (if necessary).
3. Scale the window to match the size of the viewport.
4. Translate the window to the screen coordinates.

Viewport Transformation

1

- Window-to-viewport mapping



$$\begin{aligned} vx &= vx1 + (wx - wx1) * (vx2 - vx1) / (wx2 - wx1); \\ vy &= vy1 + (wy - wy1) * (vy2 - vy1) / (wy2 - wy1); \end{aligned}$$

The Viewing Transformation

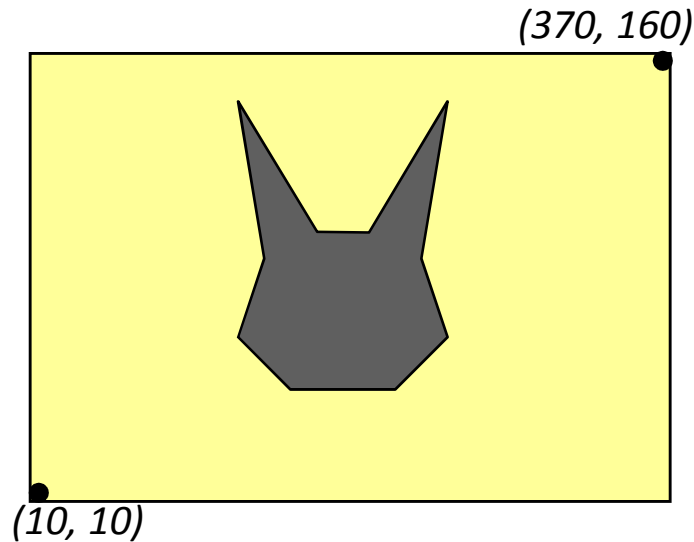
the complete transformation is therefore

$$\begin{pmatrix} S_x & 0 & 0 \\ 0 & S_y & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} \cos F & \sin F & 0 \\ -\sin F & \cos F & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & -D_x \\ 0 & 1 & -D_y \\ 0 & 0 & 1 \end{pmatrix}$$

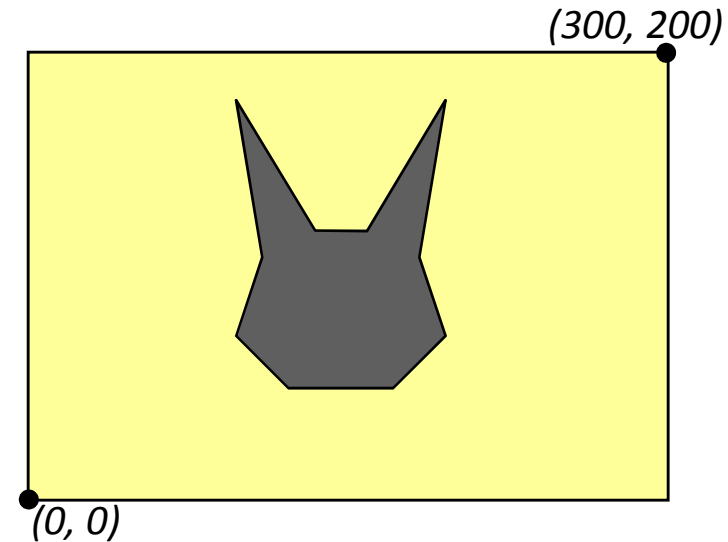
*Note: this leaves you in unit coordinates.
It is easiest to clip in unit coordinates,
and then scale to match your window.*

Window to Viewport Example

- Consider the following change of coordinates:



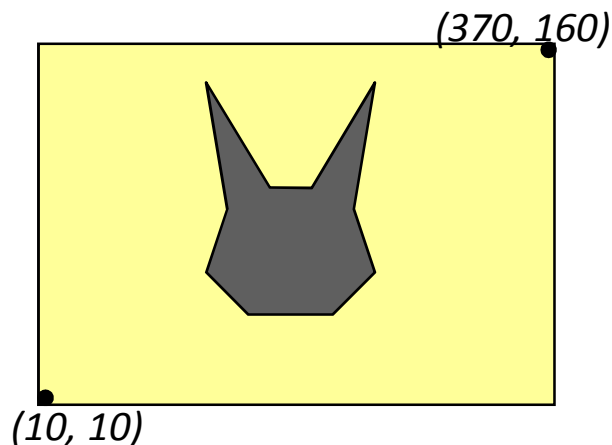
*World coordinates
("window")*



*Display coordinates
("viewport")*

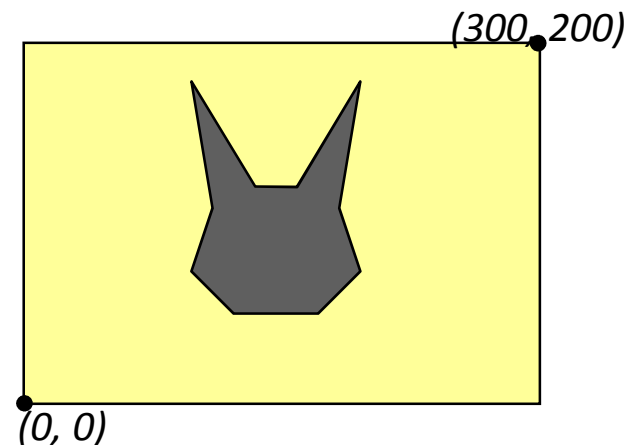
What's the VCRF?

- We need an origin and a view-up vector:



*World coordinates
("window")*

Origin: (10, 10)



*Display coordinates
("viewport")*

VUP: (0,1)



Now we need scale

General case: all VCRFs are axis-aligned:

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

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

$$s_x = \frac{300 - 0}{370 - 10} = \frac{300}{360} = \frac{5}{6}$$

$$s_y = \frac{200 - 0}{160 - 10} = \frac{200}{150} = \frac{4}{3}$$

So Window to Viewport Transformation is....

$$\begin{bmatrix} \frac{5}{6} & 0 & \frac{-25}{3} \\ 0 & \frac{4}{3} & \frac{-40}{3} \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} \frac{5}{6} & 0 & 0 \\ 0 & \frac{4}{3} & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & -10 \\ 0 & 1 & -10 \\ 0 & 0 & 1 \end{bmatrix}$$

*Axis-aligned coordinates
might still be reflected or
coordinate-swapped*

Clipping

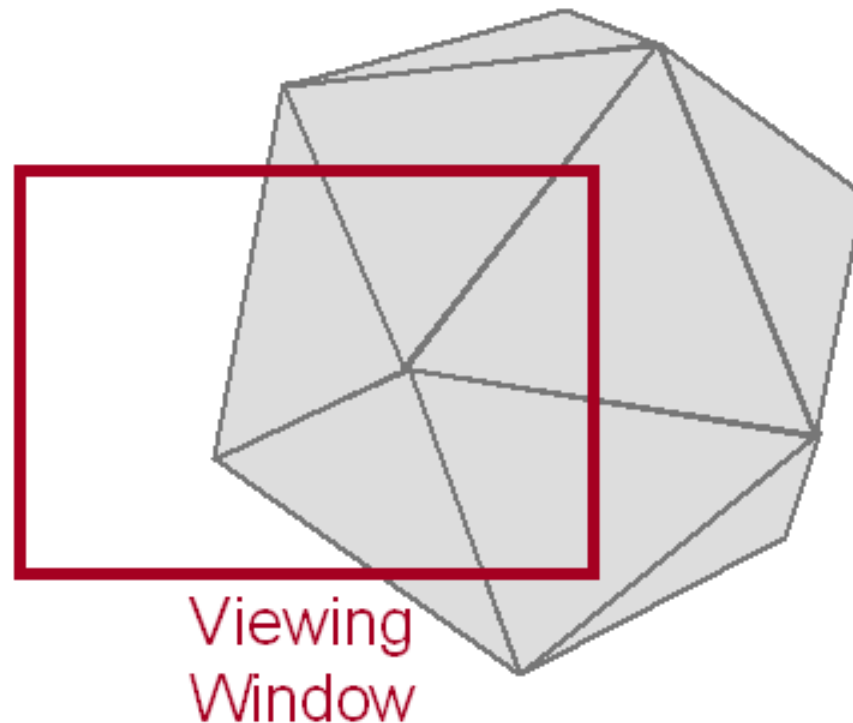
- Avoid drawing parts of primitives outside window
 - Window defines part of scene being viewed
 - Must draw geometric primitives only inside window



Screen Coordinates

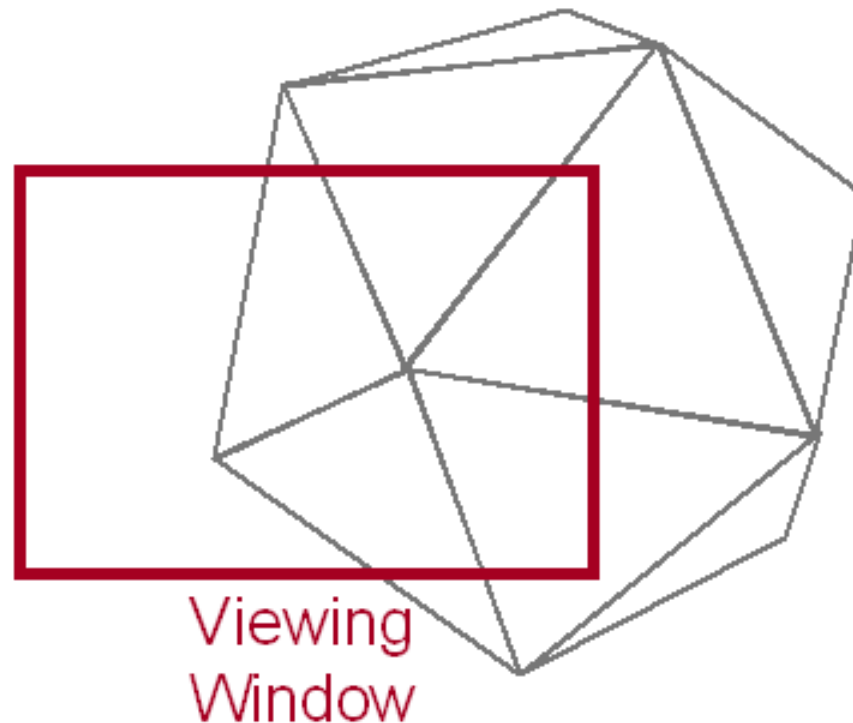
Clipping

- Avoid drawing parts of primitives outside window
 - Window defines part of scene being viewed
 - Must draw geometric primitives only inside window



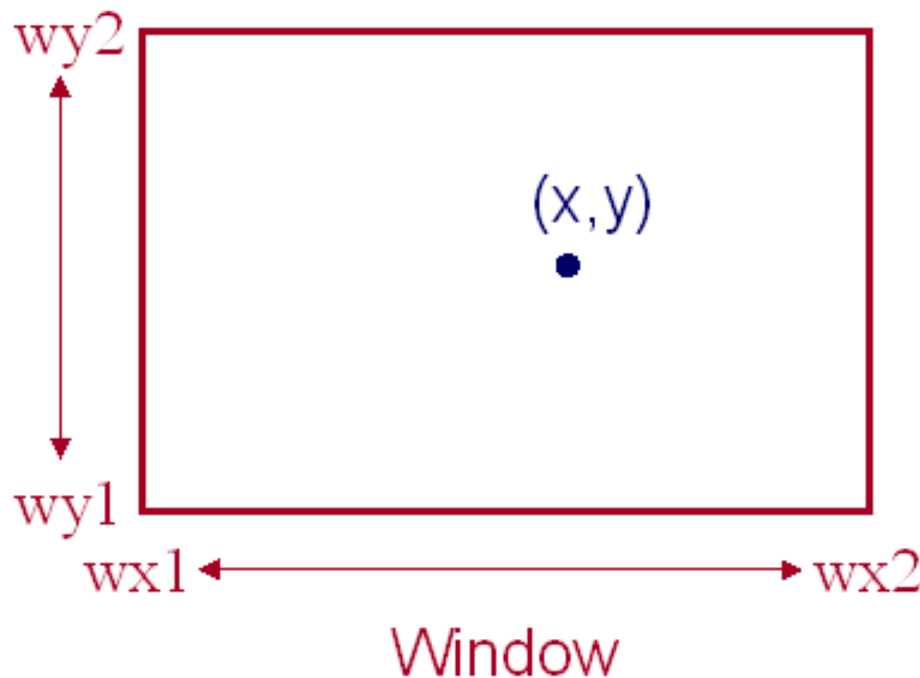
Clipping

- Avoid drawing parts of primitives outside window
 - Points
 - Lines
 - Polygons
 - Circles
 - etc.



Point Clipping

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



```
inside =  
    (x >= wx1) &&  
    (x <= wx2) &&  
    (y >= wy1) &&  
    (y <= wy2) ;
```

The Cohen-Sutherland algorithm (2D Version)

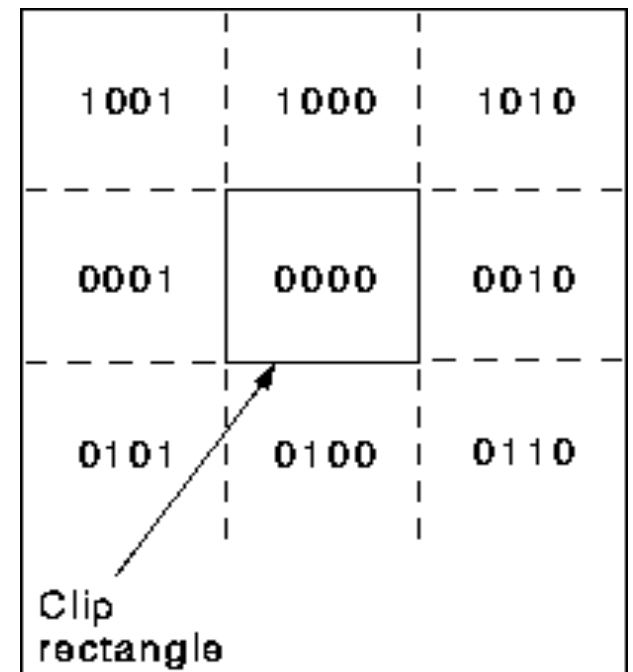
The Cohen-Sutherland is a commonly used method of line-clipping. Here the 2D version is illustrated, to demonstrate the algorithm in principle. Each endpoint of the line is assigned a 4-bit 'outcode', based on these criteria:

bit 1 - point above top edge ($Y > Y_{\max}$)

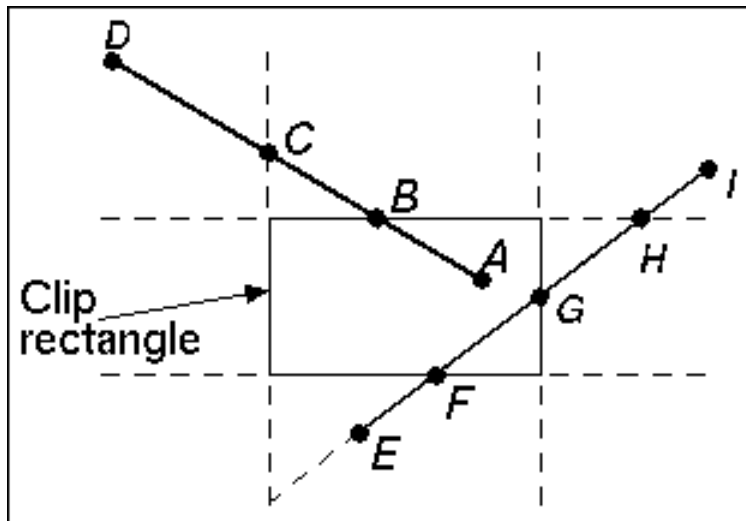
bit 2 - point below bottom edge ($Y < Y_{\min}$)

bit 3 - point to right of right edge ($X > X_{\max}$)

bit 4 - point to left of left edge ($X < X_{\min}$)



The Cohen-Sutherland algorithm

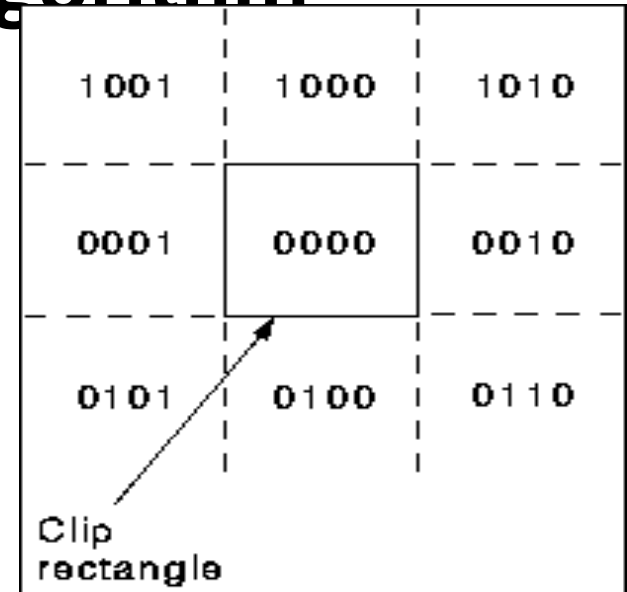


1001	1000	1010
0001	0000	0010
0101	0100	0110

Clip rectangle

Consider line AD (above). Point A has outcode 0000 and point D has outcode 1001. The line AD cannot be trivially accepted or rejected. D is chosen as the outside point. Its outcode shows that the line cuts the top and left edges (The bits for these edges are different in the two outcodes). Let the order in which the algorithm tests edges be top-to-bottom, left-to-right. The top edge is tested first, and line AD is split into lines DB and BA. Line BA is trivially accepted (both A and B have outcodes of 0000). Line DB is in the outside halfspace of the top edge, and gets rejected.

The Cohen-Sutherland algorithm



If both ends have outcodes of 0000, then the line is trivially accepted, as it is entirely within the clipping rectangle. If the logical AND of the two outcodes is not zero then the line can be trivially rejected. This is because if the two endpoints each have the same bit set, then both points are on the same side of one of the clipping lines (ie. both to the right of the clipping rectangle), and so the line cannot cross the clipping rectangle.

If a line cannot be trivially accepted or trivially rejected then it is subdivided into two segments such that one or both can be rejected. Lines are divided at the point at which they cross the clipping rectangle edges. The segment(s) lying outside the clipping edge is rejected.

```
gluLookAt(GLdouble eyeX,  
          GLdouble eyeY,  
          GLdouble eyeZ,  
          GLdouble centerX,  
          GLdouble centerY,  
          GLdouble centerZ,  
          GLdouble upX,  
          GLdouble upY,  
          GLdouble upZ);
```

creates a viewing matrix derived from an eye point, a reference point indicating the center of the scene, and an UP vector.

eyeX, eyeY, eyeZ

Specifies the position of the eye point.

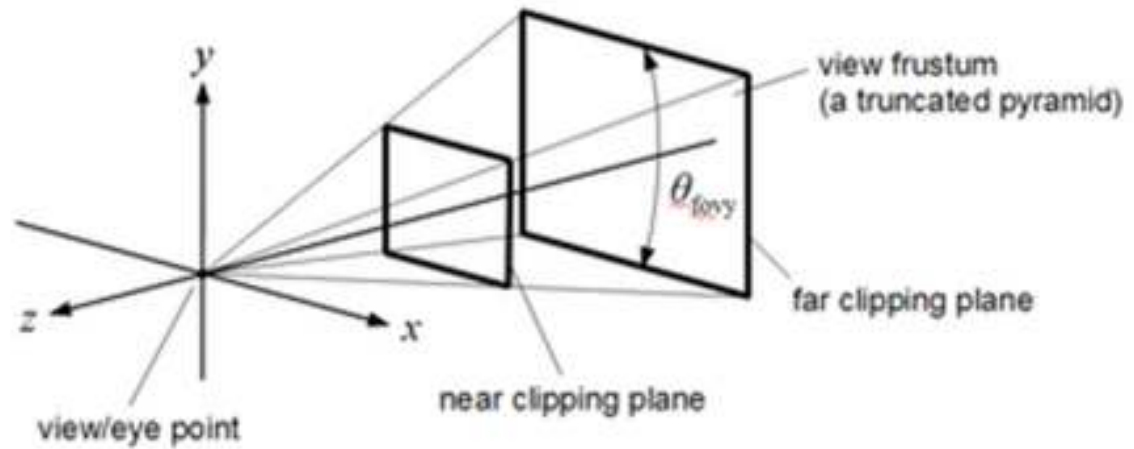
centerX, centerY, centerZ

Specifies the position of the reference point.

upX, upY, upZ

Specifies the direction of the up vector.


```
void gluPerspective(  
    GLdouble fovy,  
    GLdouble aspect,  
    GLdouble zNear,  
    GLdouble zFar);
```



set up a perspective projection matrix

fovy

Specifies the field of view angle, in degrees, in the y direction.

aspect

Specifies the aspect ratio that determines the field of view in the x direction. The aspect ratio is the ratio of x (width) to y (height).

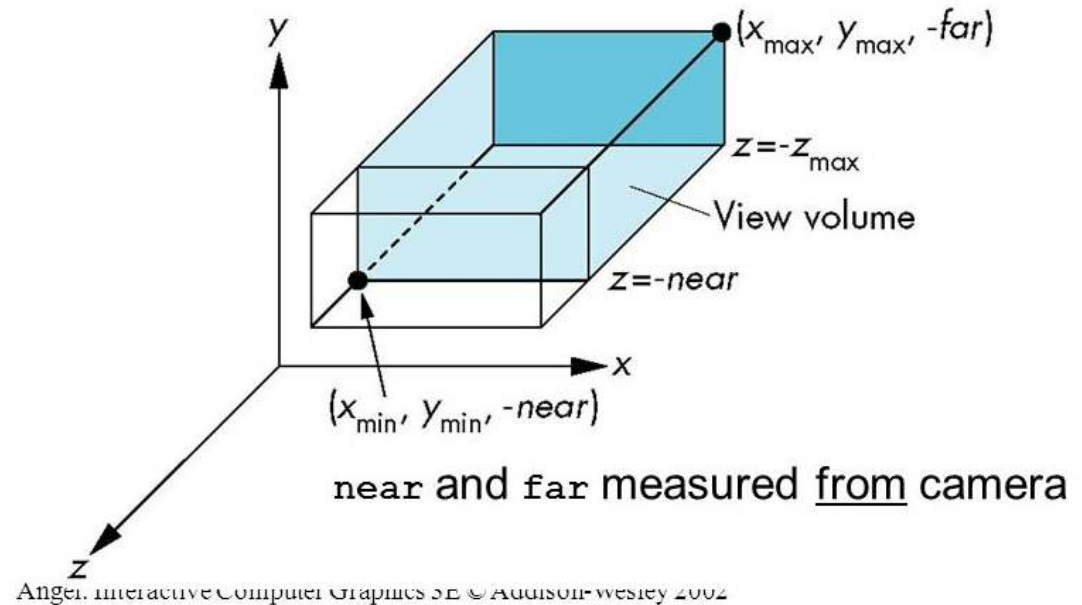
zNear

Specifies the distance from the viewer to the near clipping plane (always positive).

zFar

Specifies the distance from the viewer to the far clipping plane (always positive).

```
void glOrtho(
    GLdouble left,
    GLdouble right,
    GLdouble bottom,
    GLdouble top,
    GLdouble nearVal,
    GLdouble farVal);
```



multiply the current matrix with an orthographic matrix

left, right

Specify the coordinates for the left and right vertical clipping planes.

bottom, top

Specify the coordinates for the bottom and top horizontal clipping planes.

nearVal, farVal

Specify the distances to the nearer and farther depth clipping planes. These values are negative if the plane is to be behind the viewer.