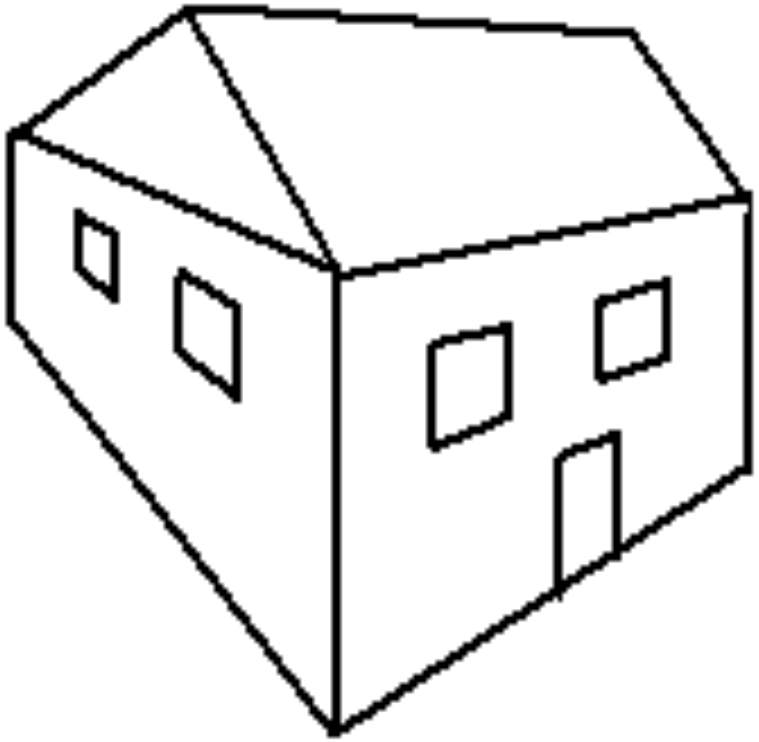
Computer Graphics

COMP H3016

Lecturer: Simon McLoughlin

Lecture 3



Review and Overview

- Last week we looked at how to apply **transformations to 2-d primitives** and get them to move around the screen the way we would like
- The transformations we looked at were **translation, rotation and scaling**
- We also looked at **homogenous coordinates** and saw how we can combine transformations (or concatenate) using only multiplication of transformation matrices.
- Today we will look at some of the **properties of the transformation matrices in this form** and look at performing these transformations using homogenous coordinates

Translation matrix

- We saw already that two transformations can be applied to a object by **multiplying the transformation matrices to get a composite transformation matrix** (one that includes more than one transformation)
- If the two transformations are translations then the matrix multiplication need not be done because they possess the following property:

$$\text{The matrix product } T(t_{x1}, t_{y1}) \cdot T(t_{x2}, t_{y2}) \text{ is:}$$

$$\begin{bmatrix} 1 & 0 & t_{x1} \\ 0 & 1 & t_{y1} \\ 0 & 0 & 1 \end{bmatrix} . \begin{bmatrix} 1 & 0 & t_{x2} \\ 0 & 1 & t_{y2} \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & t_{x1} + t_{x2} \\ 0 & 1 & t_{y1} + t_{y2} \\ 0 & 0 & 1 \end{bmatrix}$$

- Which demonstrates that two successive translations are **additive**
- It also demonstrates that translation matrices are **commutative**

Translation Properties

1. $T(0,0) = I$
2. $T(t_{x1}, t_{y1}) \cdot T(t_{x2}, t_{y2}) = T(t_{x1} + t_{x2}, t_{y1} + t_{y2})$
3. $T(t_{x1}, t_{y1}) \cdot T(t_{x2}, t_{y2}) = T(t_{x2}, t_{y2}) \cdot T(t_{x1}, t_{y1})$
4. $\mathrm{T}^{-1}(t_x, t_y) = T(-t_x, -t_y)$

Rotation matrix

- Two successive rotations applied to a point **P** produce the transformed position

$$\mathbf{P}' = (\mathbf{R}(\theta_2).\mathbf{R}(\theta_1)) \cdot \mathbf{P}$$

- By multiplying the two rotation matrices we can verify that two successive **rotations are additive**

$$\begin{bmatrix} \cos\theta_2 & -\sin\theta_2 & 0 \\ \sin\theta_2 & \cos\theta_2 & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} \cos\theta_1 & -\sin\theta_1 & 0 \\ \sin\theta_1 & \cos\theta_1 & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} \cos(\theta_1+\theta_2) & -\sin(\theta_1+\theta_2) & 0 \\ \sin(\theta_1+\theta_2) & \cos(\theta_1+\theta_2) & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

- Last week we looked at **general pivot points rotations** and their homogenous form - these properties apply to these also when the pivot point is the same for the two rotations

Rotation matrices - other properties

For rotation matrices,

$$R^{-1}(\theta) = R(-\theta).$$

$$R(0) = I$$

and

$$R(\theta) \cdot R(\phi) = R(\theta + \phi)$$

$$R(\theta) \cdot R(\phi) = R(\phi) \cdot R(\theta)$$

Scaling matrix (fixed point is coordinate origin)

- **Concatenating transformation matrices for two successive scaling transformations** produces the following composite scaling matrix

The matrix product $S(s_{x1}, s_{y1}) \cdot S(s_{x2}, s_{y2})$ is:

$$\begin{bmatrix} s_{x1} & 0 & 0 \\ 0 & s_{y1} & 0 \\ 0 & 0 & 1 \end{bmatrix} . \begin{bmatrix} s_{x2} & 0 & 0 \\ 0 & s_{y2} & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} s_{x1} \cdot s_{x2} & 0 & 0 \\ 0 & s_{y1} \cdot s_{y2} & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Only diagonal elements in the matrix - easy to multiply!

- The resulting matrix indicates that **successive scaling operations are multiplicative**. That is, if we were to triple the size of an object twice in succession, the final size would be 9 times the original

General scaling directions

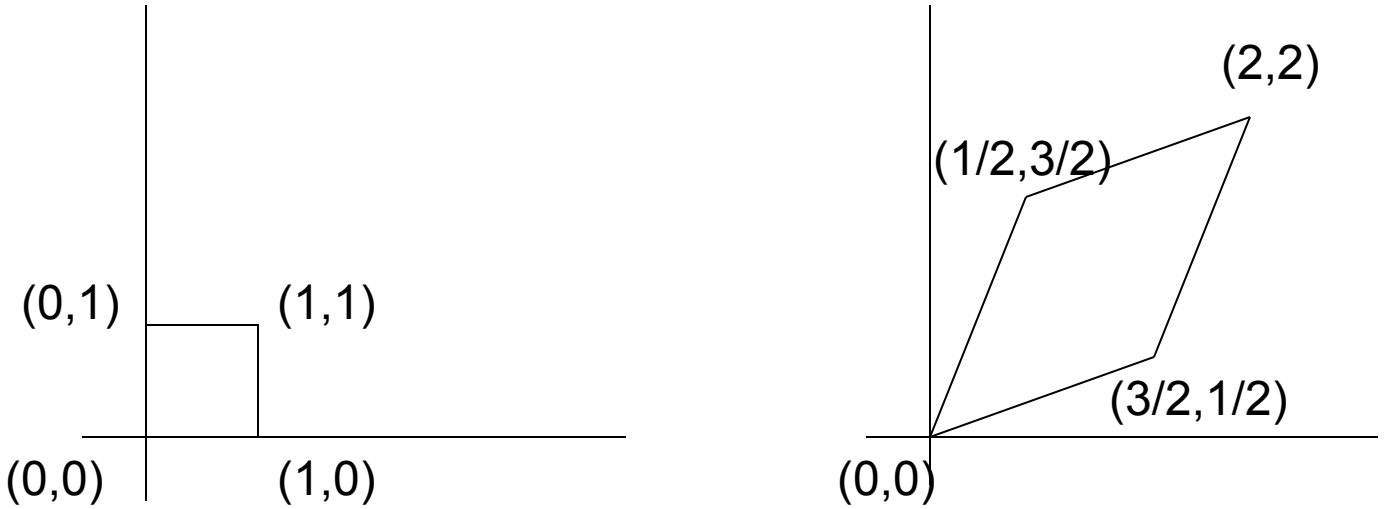
- Parameters for s_x and s_y scale the object along the *x* and *y* directions
- If we want to **scale the object along an arbitrary orthogonal axis by amounts specified by s_x and s_y we can first perform a rotation of the object so the directions for s_x and s_y coincide with the *x* and *y* axis.** Then the **scaling transformation is applied**, followed by an **opposite rotation to return the points to their original positions**

$$\mathbf{R}^{-1}(\theta)\cdot\mathbf{S}(s_1,s_2)\cdot\mathbf{R}(\theta)$$

$$=\begin{bmatrix} s_1\cos^2\theta + s_2\sin^2\theta & (s_2-s_1)\cos\theta\sin\theta & 0 \\ (s_2-s_1)\cos\theta\sin\theta & s_1\sin^2\theta + s_2\cos^2\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

- Note: **Assumes scaling fixed point is the origin**. We could take this one step further and concatenate translation matrices for non-origin fixed points

General scaling directions



A square is converted to a parallelogram using the general direction scaling matrix with s1 = 1, s2 = 2, θ=45 degrees

Types of Transformations

- Rotation and translation
  - Angles and distances are preserved
  - Unit cube is always unit cube
  - *Rigid-Body* transformations
- Rotation, translation and scale
  - Angles & distances not preserved.
  - But parallel lines are.
  - *Affine* transformations.

Homogenous coordinates – Transformation Example

```
#define DEG_TO_RAD 0.0174532925199
#include <GL/glut.h>
#include <math.h>

//declare global variables
typedef double Matrix3x3[3][3];
Matrix3x3 theMatrix;
double square[4][2];
double cx;
double cy;

//declare method prototypes
void matrix3x3SetIdentity (Matrix3x3);
void matrix3x3Multiply(Matrix3x3, Matrix3x3);
void translate(double,double);
void rotate(double,double,double);
void scale(double,double,double,double);
void update_center();
void transformPoints();
void initSquare();
void display();
void init();

//set a 3x3 matrix to the identity matrix [1 0 0;0 1 0;0 0 1]
void matrix3x3SetIdentity (Matrix3x3 m) {
          int i,j;
          for (i=0;i<3;i++)
                    for (j=0;j<3;j++){
                              m[i][j] = (i==j);
          }
}
```

Homogenous coordinates – Transformation Example

```
// multiplies matrix a times b putting the result in b
void matrix3x3Multiply(Matrix3x3 a, Matrix3x3 b) {
          int r,c;
          Matrix3x3 tmp;

          for (r=0;r<3;r++) {
                    for (c=0;c<3;c++)
                              tmp[r][c] = a[r][0]*b[0][c] + a[r][1]*b[1][c] + a[r][2]*b[2][c];
          }

          for (r=0;r<3;r++) {
                    for(c=0;c<3;c++)
                              b[r][c] = tmp[r][c];
          }
}

//translate a point by tx and ty using homogenous coordinates
void translate(double tx, double ty) {
          Matrix3x3 m;

          matrix3x3SetIdentity(m);
          m[0][2] = tx;
          m[1][2] = ty;
          matrix3x3Multiply(m,theMatrix);
}
```

Homogenous coordinates – Transformation Example

```
//Write a function to rotate points by an angle theta about a fixed pivot point //(rx,ry)
using homogenous coordinates – see function translate

//rotate a point by theta degrees about pivot point(rx,ry)
void rotate(double theta,double rx,double ry)
{
}

//Write a function to scale points by sx and sy relative to fixed point (fx,fy) //using
homogenous coordinates – see function translate

// scale points by sx and sy relative to fixed point (fx,fy)
void scale(double sx, double sy, double fx, double fy)
{
}

//update the center of the square so as to do the rotation
void update_center(){
          cx = abs(square[3][0]+square[1][0])/2;
          cy = abs(square[3][1]+square[1][1])/2;
}
```

Homogenous coordinates – Transformation Example

```
//multiply the composite or concatenated transformation matrix by our coordinates
void transformPoints()
{
          int k;
          float tmp;
          for (k=0;k<4;k++) {
                    tmp = theMatrix[0][0] * square[k][0] + theMatrix[0][1] * square[k][1]
+ theMatrix[0][2];
                    square[k][1] = theMatrix[1][0] * square[k][0] + theMatrix[1][1] *
square[k][1] + theMatrix[1][2];
                    square[k][0] = tmp;
          }
}

//initialise square vertices
void initSquare(){
          square[0][0] = 100.0;
          square[0][1] = 100.0;
          square[1][0] = 100.0;
          square[1][1] = 200.0;
          square[2][0] = 200.0;
          square[2][1] = 200.0;
          square[3][0] = 200.0;
          square[3][1] = 100.0;
}
```

Homogenous coordinates – Transformation Example

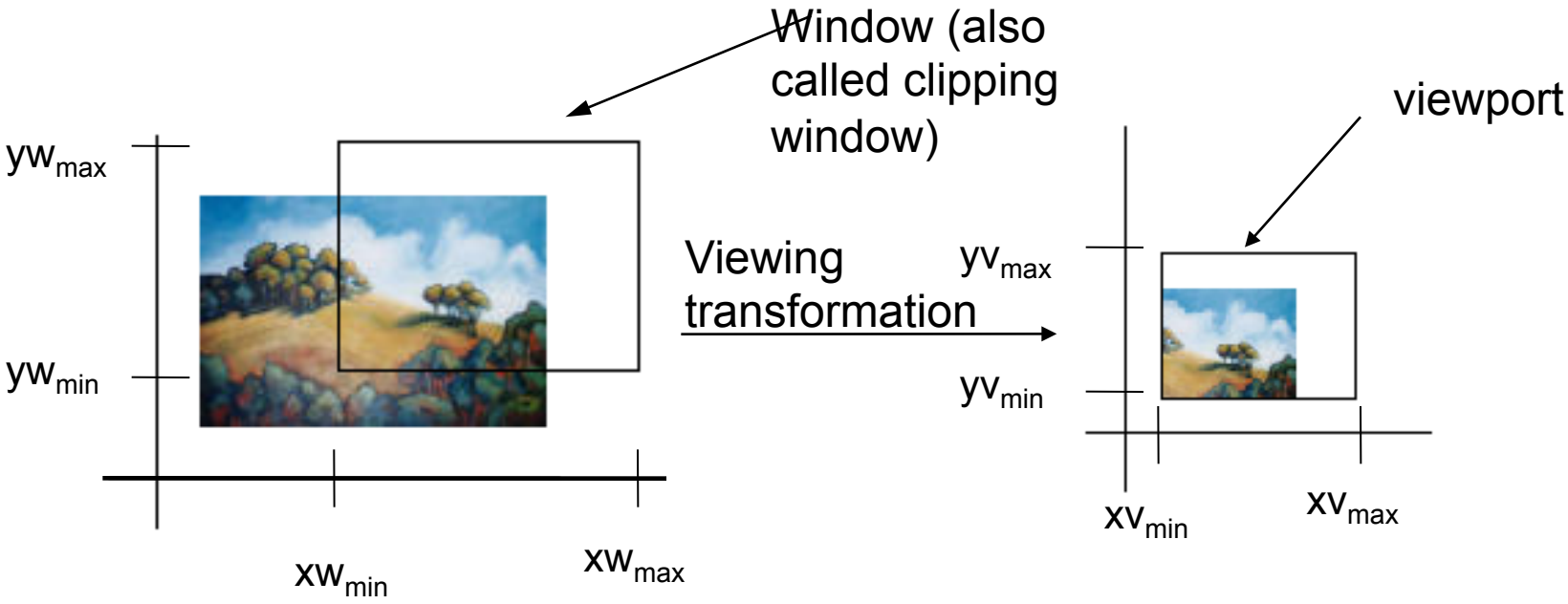
```
//draw the image
void display()
{
          int i,j;
          glClear(GL_COLOR_BUFFER_BIT);
          glBegin(GL_LINE_LOOP);
                    glVertex2d(square[0][0],square[0][1]);
                    glVertex2d(square[1][0],square[1][1]);
                    glVertex2d(square[2][0],square[2][1]);
                    glVertex2d(square[3][0],square[3][1]);
          glEnd();

          matrix3x3SetIdentity(theMatrix);
          translate(1.0,1.0);
          update_center();
          // add in each of the lines below only after you've written the corresponding
          // function
          //rotate(2.0,cx,cy);
          //scale(0.99,0.99,cx,cy);
          transformPoints();
          glutSwapBuffers();
}
```

Review and Overview

- We have looked at the **properties of the transformation matrices (in homogenous form)** and we looked at performing these transformations using homogenous coordinates in the lab
- You may (or may not) have noticed that sometimes the **object would leave the window on the screen and then return sometime later** (if you were making mistakes)
- It is almost like the window on the screen is like a window (in your house) that you can look through and see a portion of the world outside.
- This is in fact what happens in your openGL programs
- You specify object positions and transformations within a world coordinate system and the window (called a viewport when on screen) can see a subspace of this world coordinate system.
- When the object enters this subspace it can be seen on the viewport
- You may have also noticed that as part of an object leaves the viewport the other part remains in the viewport
- This is called clipping whereby we must clip part of the object as it leaves the viewport.

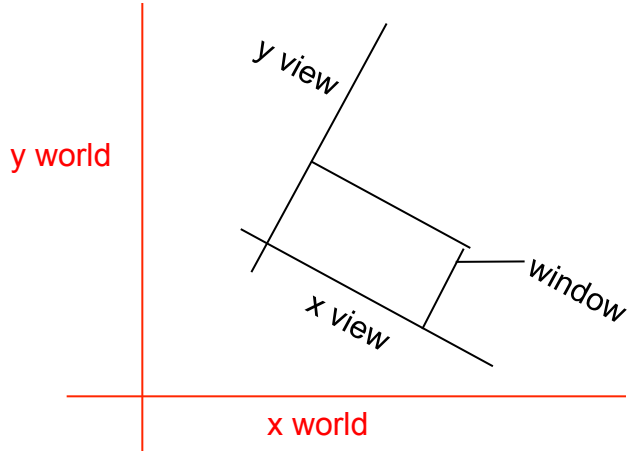
Some terminology

- A world coordinate area selected for display is called a **window**
- An area on the display device to which the window is to be mapped is called a **viewport** (not necessarily rectangular but obviously the same shape as the window)
- The mapping of the window to the viewport is called a **viewing transformation**



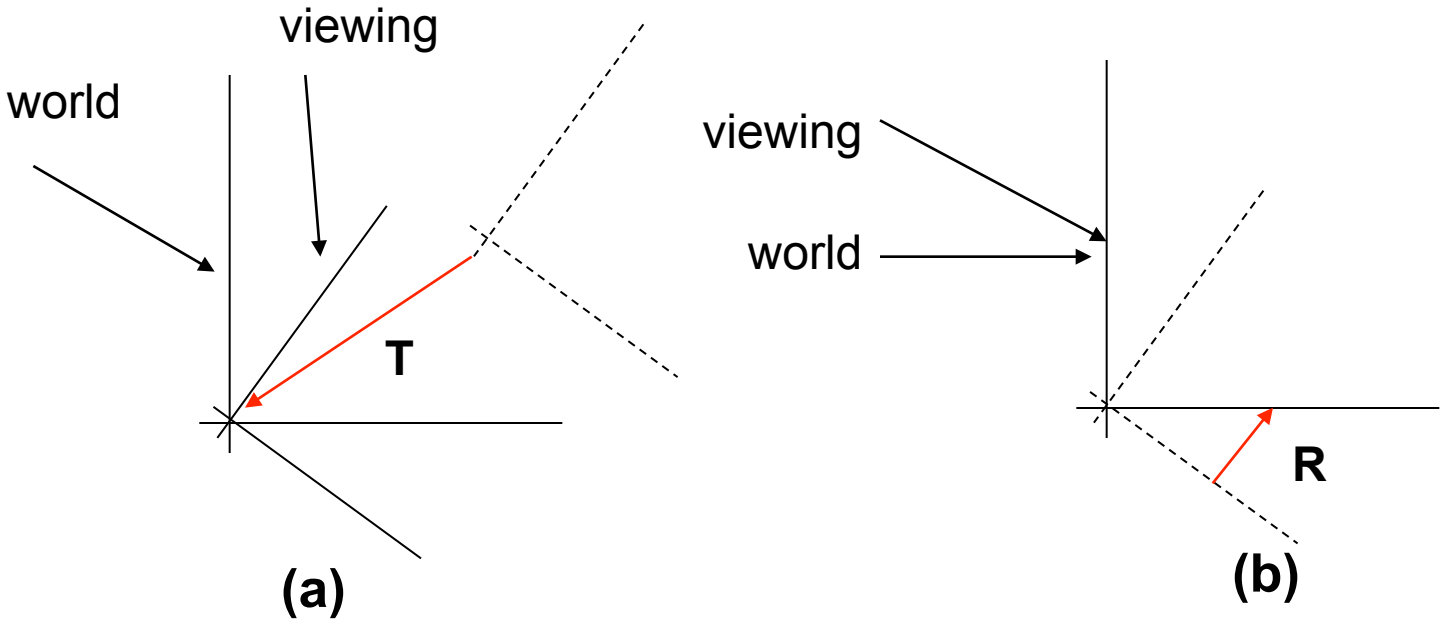
Viewing coordinate system

- It is usually the case (especially with 3-d computer graphics) that the view of the world coordinate system should be possible from different angles and positions.
- We introduce the idea of a **viewing coordinate system** to make this possible
- This coordinate system can have a different orientation and origin to the world coordinate system
- Points are transformed from the world coordinate system to the viewing coordinate system (by knowing rotation and translation parameters)
- The viewing coordinate system is merely a reference frame for setting up arbitrary orientations for the viewing window



Converting world coordinates to viewing coordinates

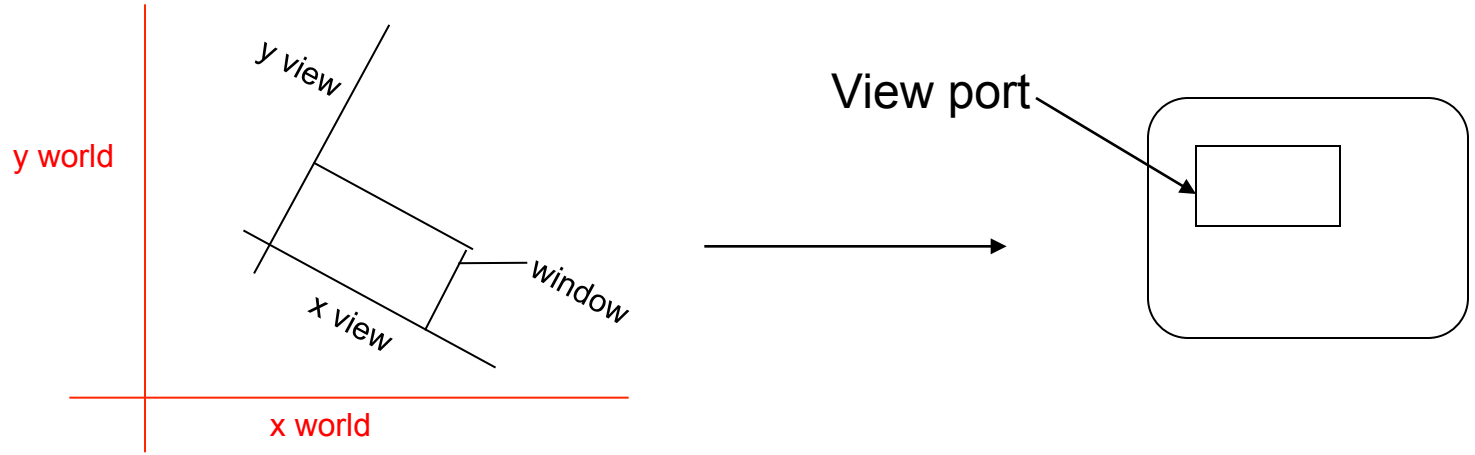
- The matrix for converting from world coordinate positions to viewing coordinate positions is a composite transformation (a translation and a rotation)
- Firstly the viewing origin is translated to the world origin(a), then a rotation is performed to align the viewing coordinate system with the world coordinate system(b)



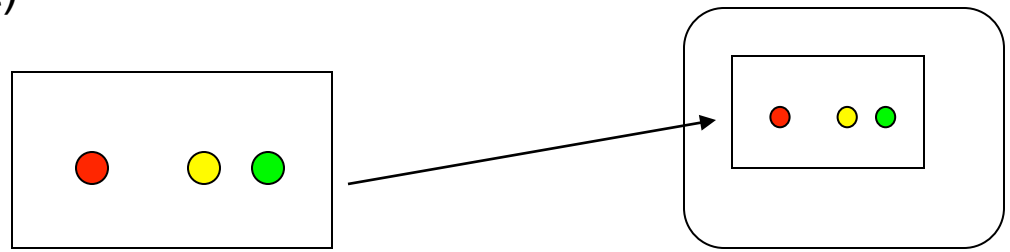
$$\mathbf{M}_{vc} = (\mathbf{R} \cdot \mathbf{T}) \cdot \mathbf{M}_{wc}$$

From viewing coordinates to screen coordinates

- Once we have converted from world coordinates to viewing coordinates we must convert to viewport coordinates (on the screen)



- Object placing must maintain the same relative placing in the viewport as they did in the window (in effect the window is scaled to the size of the viewport)



From viewing coordinates to screen coordinates

- To maintain the same relative placing in the viewport as in the window we require that

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

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

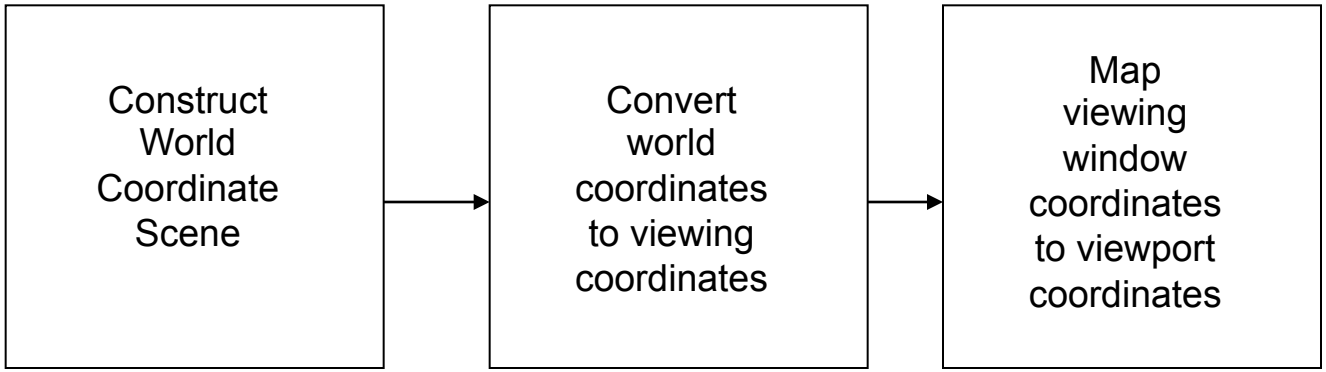
- Solving these expressions for the viewport position (xv, yv) gives:

$$xv = xv_{\min} + (xw - xw_{\min})sx \qquad yv = yv_{\min} + (yw - yw_{\min})sy$$

- where

$$sx = \frac{xv_{\max} - xv_{\min}}{xw_{\max} - xw_{\min}} \qquad sy = \frac{yv_{\max} - yv_{\min}}{yw_{\max} - yw_{\min}}$$

Two dimensional viewing-transformation pipeline



Zooming and Panning effects

- Zooming effects can be obtained by varying the size of the window while the size of the viewport remains fixed. As windows are made smaller we zoom in on some part of a scene to show details not visible with larger windows
- Panning effects can be obtained by moving a fixed size window across the various objects in a scene

Clipping

- A procedure that identifies the **portions of a picture that are inside or outside a specific region** is called a **clipping algorithm**.

- Many applications of Clipping exist:
  - extracting part of a defined scene for viewing
  - identifying visible surfaces in three dimensional views
  - displaying a multi-window environment
  - drawing applications that allow part of a picture to be selected for cutting copying or moving

- For the viewing transformation we want to display only those picture parts that are within the window area
- Clipping routines can be applied to the scene in either world coordinates or device coordinates although the latter requires that the whole world coordinate system is mapped to device coordinates

Clipping

- Clipping can be applied to the following output primitives
  - Points
  - Lines
  - Polygons
  - Curves
  - Text

- Line and polygon clipping routines are standard components in graphics packages
- Most graphics packages accommodate curved objects also
- Another way to handle curved objects is to approximate them with straight line segments and apply line or polygon clipping procedures

Point Clipping

- Point clipping is trivial
- Consider a point P = (x,y) and a rectangular clipping window defined by the edges (xw_{min}, xw_{max}, yw_{min}, yw_{max})
- The point P will be displayed in the window if the following inequalities are satisfied

$$xw_{min} <= x <= xw_{max}$$

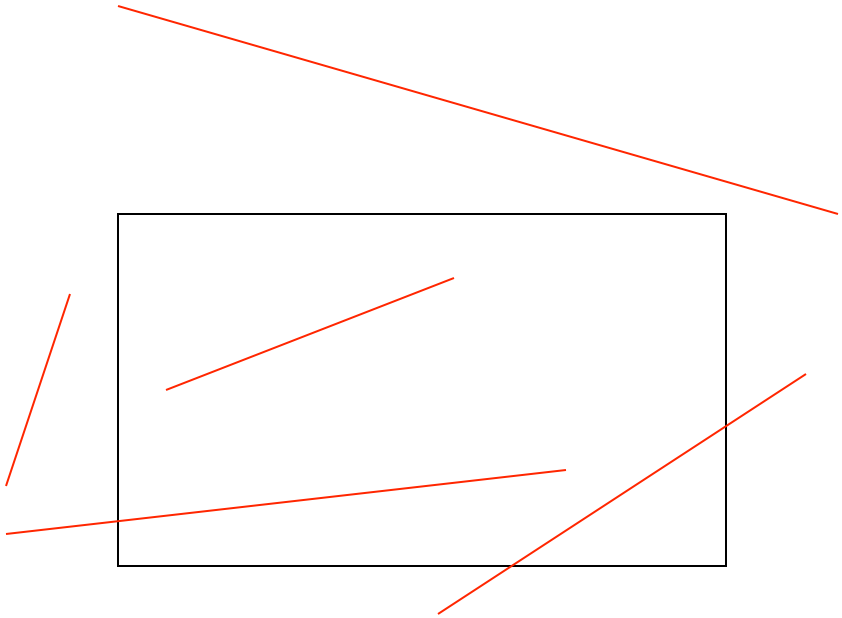
$$yw_{min} <= y <= yw_{max}$$

- Applications involving explosions etc. that are modeled with point particles in some region of the scene

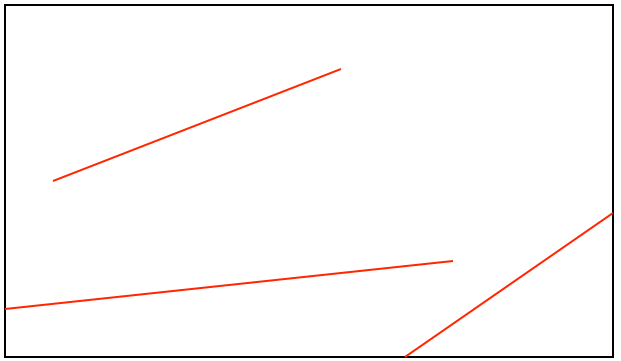
Line Clipping

- An optimised line clipping algorithm consists of several parts
- Firstly we must check to see if a **line lies completely inside or completely outside the window**
- **If a line lies completely inside a window then so will it's two endpoints**
- If a lines **endpoints both lie outside** of the **same window boundary** then that **line is outside** the window
- Lines that do not meet these two criteria will have to be checked to see **whether they intersect with the window boundaries**
- If they do intersect a boundary they have to be **clipped at that point where they leave the window**
- Just like the Bresenham line drawing and drawing algorithms there are many efficient line clipping algorithms available

Line Clipping



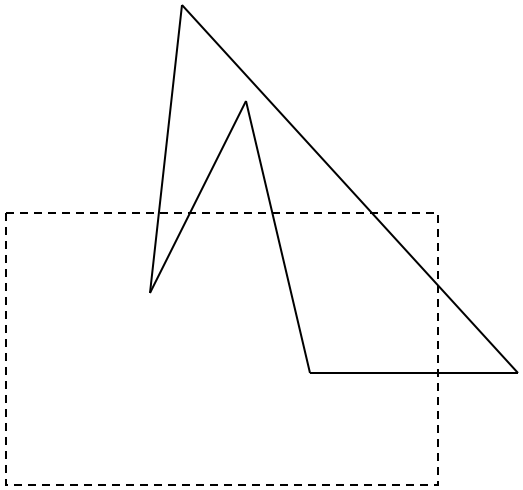
Before clipping



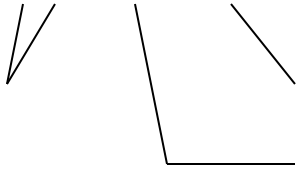
After clipping

Polygon Clipping

- You would be forgiven for thinking that a polygon is just a set of line segments so therefore a line clipping algorithm would suffice for polygons
- But…...



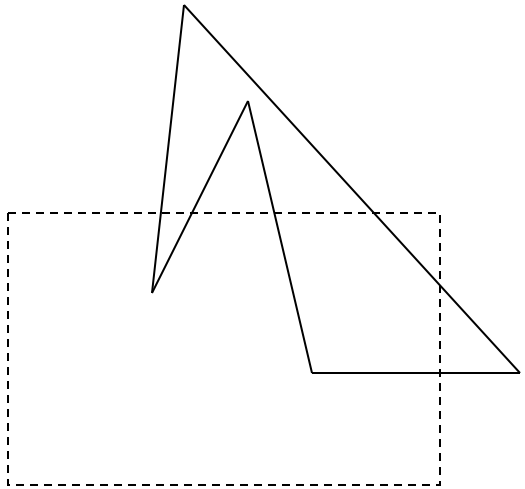
Before Clipping



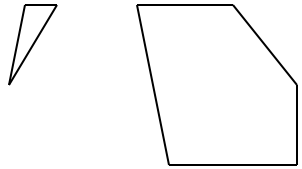
After Clipping

- Using a line clipping algorithm to clip polygons may result in a series of unconnected line segments
- What we really would like to display are bounded areas after clipping

Polygon Clipping



Before Clipping



After Clipping

- For polygon clipping we require an algorithm that will generate one or more closed areas that can be then scan converted