

# *Interactive Computer Graphics*

Lecturers: Duncan Gillies (dfg@doc.ic.ac.uk)  
Daniel Rueckert (dr@doc.ic.ac.uk)

Tutors: Paul Aljabar (pa100@doc.ic.ac.uk)  
Vu Luong (vl05@doc.ic.ac.uk)  
Peng He (ph206@doc.ic.ac.uk)  
Robin Wolz (rw1008@doc.ic.ac.uk)

Webpage:

<http://www.doc.ic.ac.uk/~dfg/graphics/graphics.html>

## *Non-DOC Students*

In order to do this course for credit you need register with the department of computing. Information on enrolment can be found on the department of computing web page:

[www.doc.ic.ac.uk](http://www.doc.ic.ac.uk)

Follow: Internal-> Student Centered Teaching  
-> External Student Registration

# *Interactive Computer Graphics*

## Lecture 1:

### Three Dimensional Graphical Scenes, Projection and Transformation

# *Two Dimensional Graphics*

The lowest level of graphics processing operates directly on the pixels in a window provided by the operating system.

Typical Primitives are:

`SetPixel(int x, int y, int colour);`

`DrawLine(int xs, int ys, int xf, int yf);`

etc.

# *World Coordinate Systems*

To achieve device independence when drawing objects we can define a world coordinate system.

This will define our drawing area in units that are suited to the application:

meters

light years

microns

etc

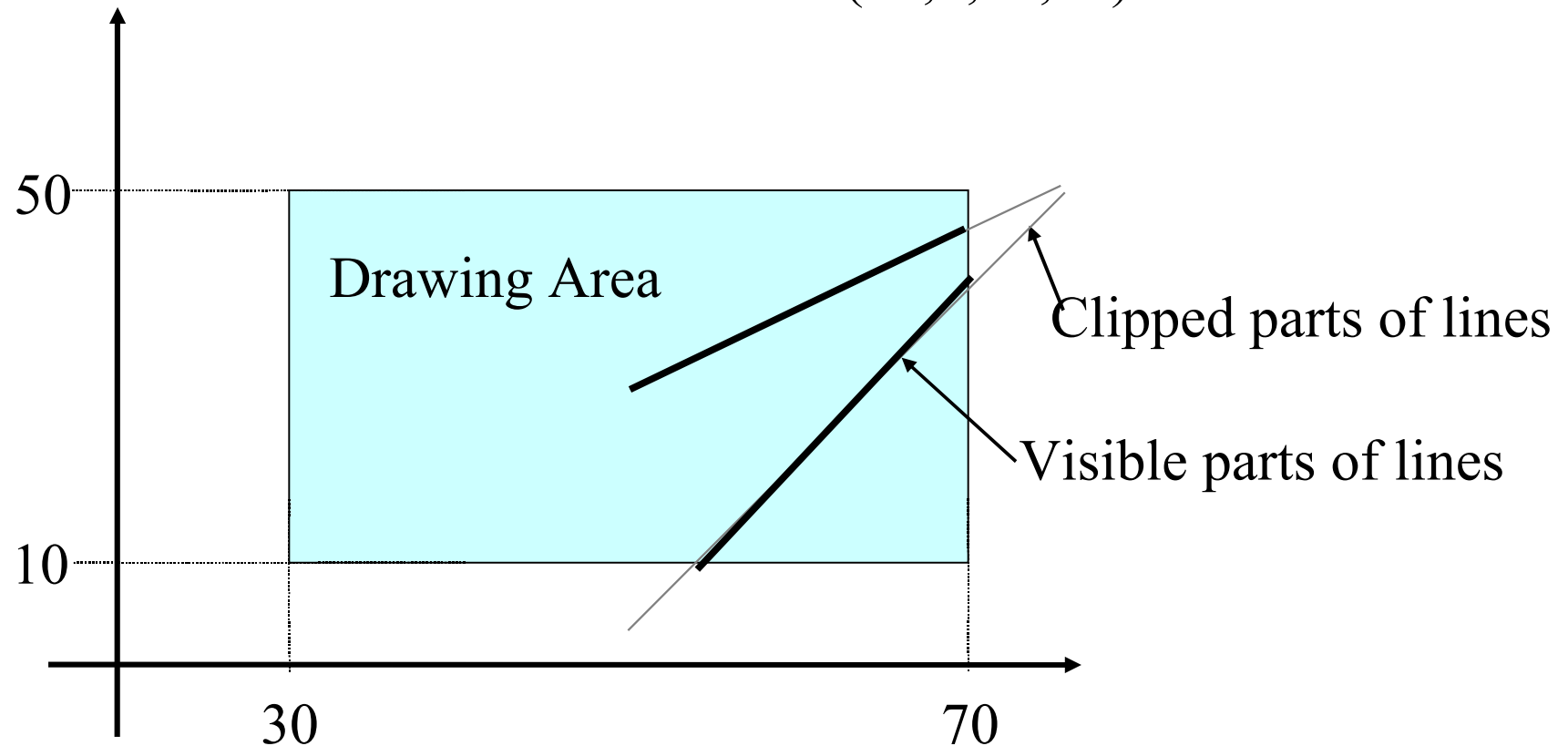
# Example

```
SetWindow(30,10,70,50)
```

```
DrawLine (50,30,80,50)
```

```
DrawLine (50,5,80,50)
```

World Coordinates



## *Normalisation*

To map device independent graphics commands to the drawing commands using the screen pixels we need a process of normalisation.

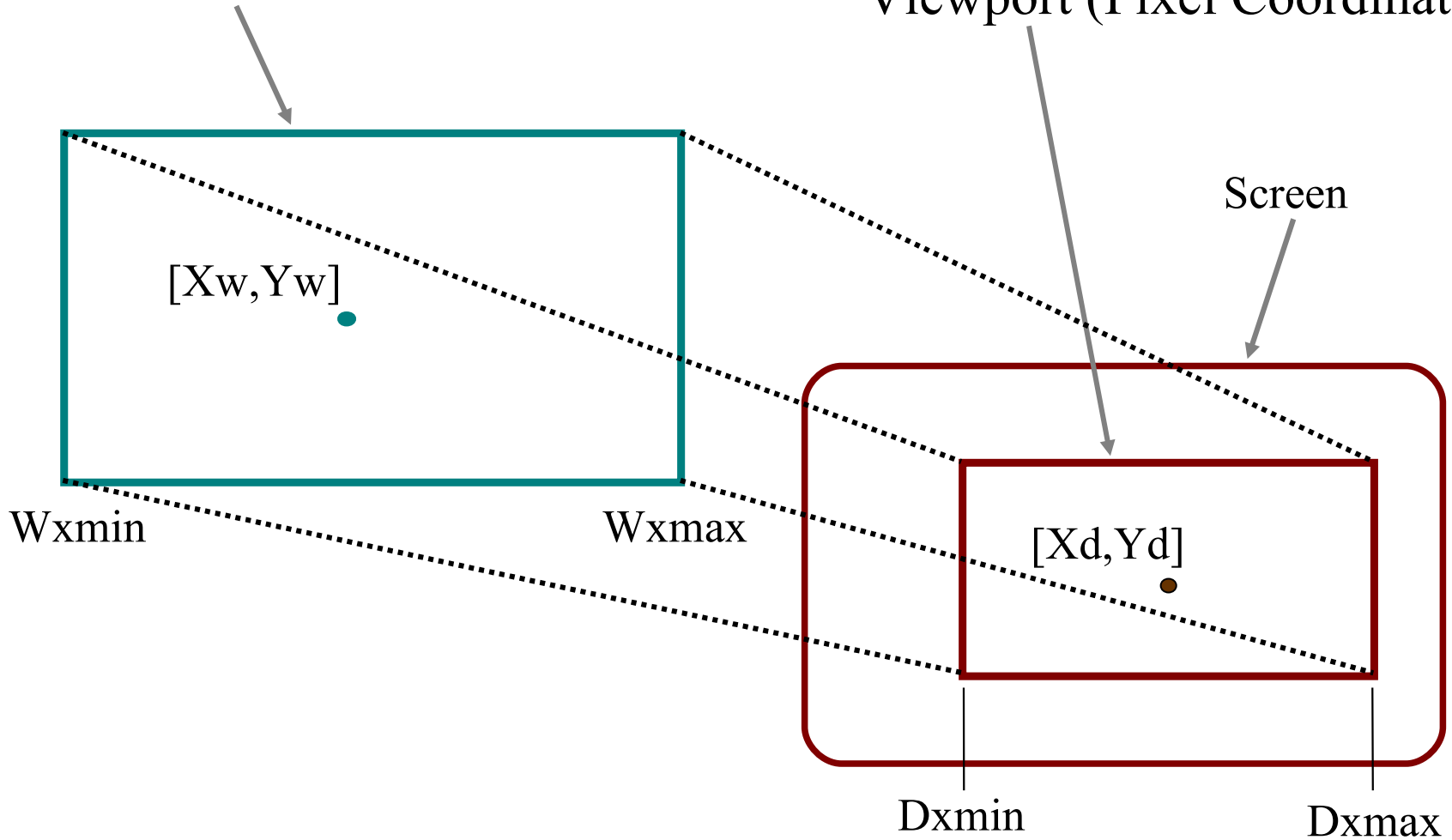
First we must call the API to find out from the operating system the pixel addresses of the corners of the area we are using.

Then we translate the world coordinates to pixel coordinates.

# Normalisation

World Coordinate Window

Viewport (Pixel Coordinates)





## *Normalisation*

Having defined our world coordinates, and obtained our device coordinates we relate the two by simple ratios:

$$\frac{X_w - W X_{min}}{W X_{max} - W X_{min}} = \frac{X_d - D X_{min}}{D X_{max} - D X_{min}}$$

rearranging we get:

$$X_d = \frac{(X_w - W X_{min})(D X_{max} - D X_{min})}{W X_{max} - W X_{min}} + D X_{min}$$

## *Normalisation*

A similar equation allows us to calculate the Y pixel coordinate. The two form a simple pair of linear equations:

$$X_d := X_w * A + B;$$

$$Y_d := Y_w * C + D;$$

Where A, B, C and D are constants defining the normalisation

# *Input for Graphics Systems*

An input event occurs when something changes, ie a mouse is moved or a button is pressed. The operating system informs the application program of events that are relevant to it.

The application program must receive this information in what is sometimes called a callback procedure (or event loop).

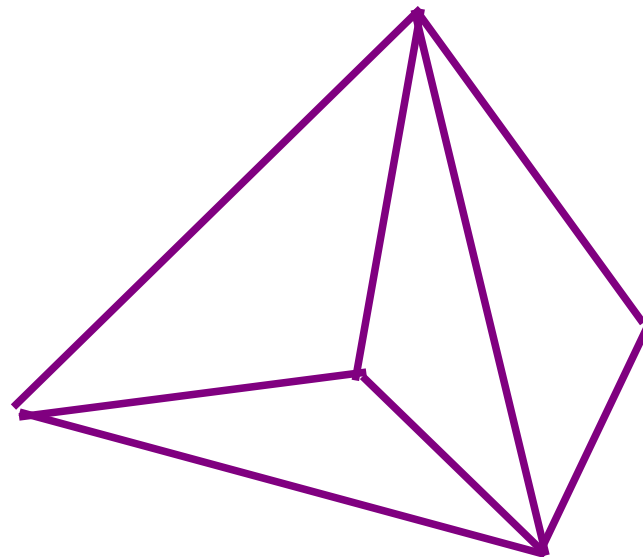
## *Simple Callback procedure*

```
while (executing) do  
{ if (menu event) ProcessMenuRequest();  
  if (mouse event)  
  { GetMouseCoordinates();  
    GetMouseButtons();  
    PerformMouseProcess();  
  }  
  if (window resize event) RedrawGraphics();  
}
```

# *Polygon Rendering*

Many graphics applications use scenes built out of planar polyhedra.

These are three dimensional objects whose faces are all *planar polygons* often called *facets*.



# *Representing Planar Polygons*

In order to represent planar polygons in the computer we will require a mixture of numerical and topological data.

## Numerical Data

Actual 3D coordinates of vertices, etc.

## Topological Data

Details of what is connected to what

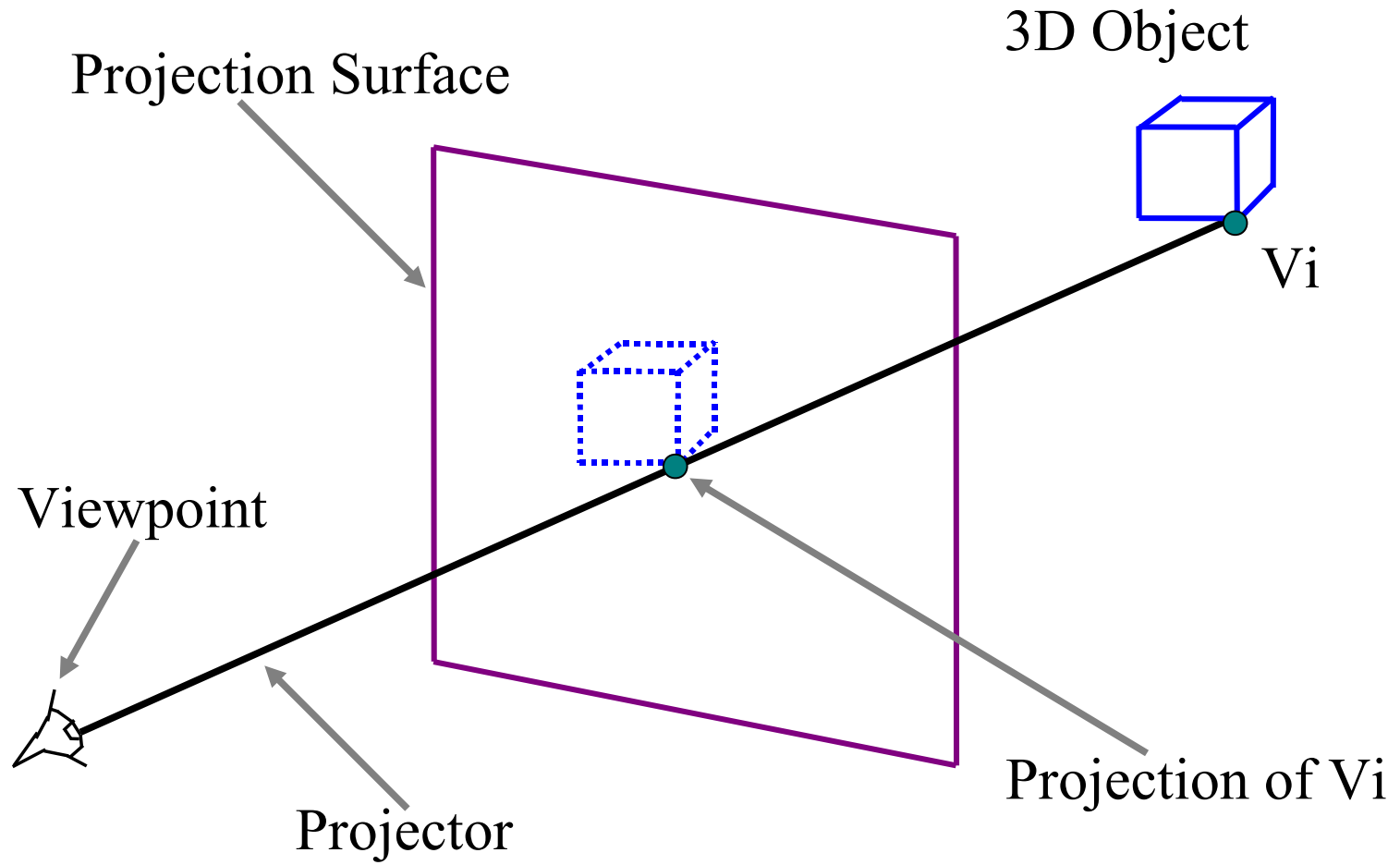
## *Projections of Wire Frame Models*

Wire frame models simply include points and lines.

In order to draw a 3D wire frame model we must first convert the points to a 2D representation. Then we can use simple drawing primitives to draw them.

The conversion from 3D into 2D is a projection.

# Projection





## *Non Linear Projections*

In general it is possible to project onto any surface:

Sphere

Cone

etc

or to use curved projectors, for example to produce lens effects.

However we will only consider planar linear projections.

# *Normal Orthographic Projection*

This is the simplest form of projection, and effective in many cases.

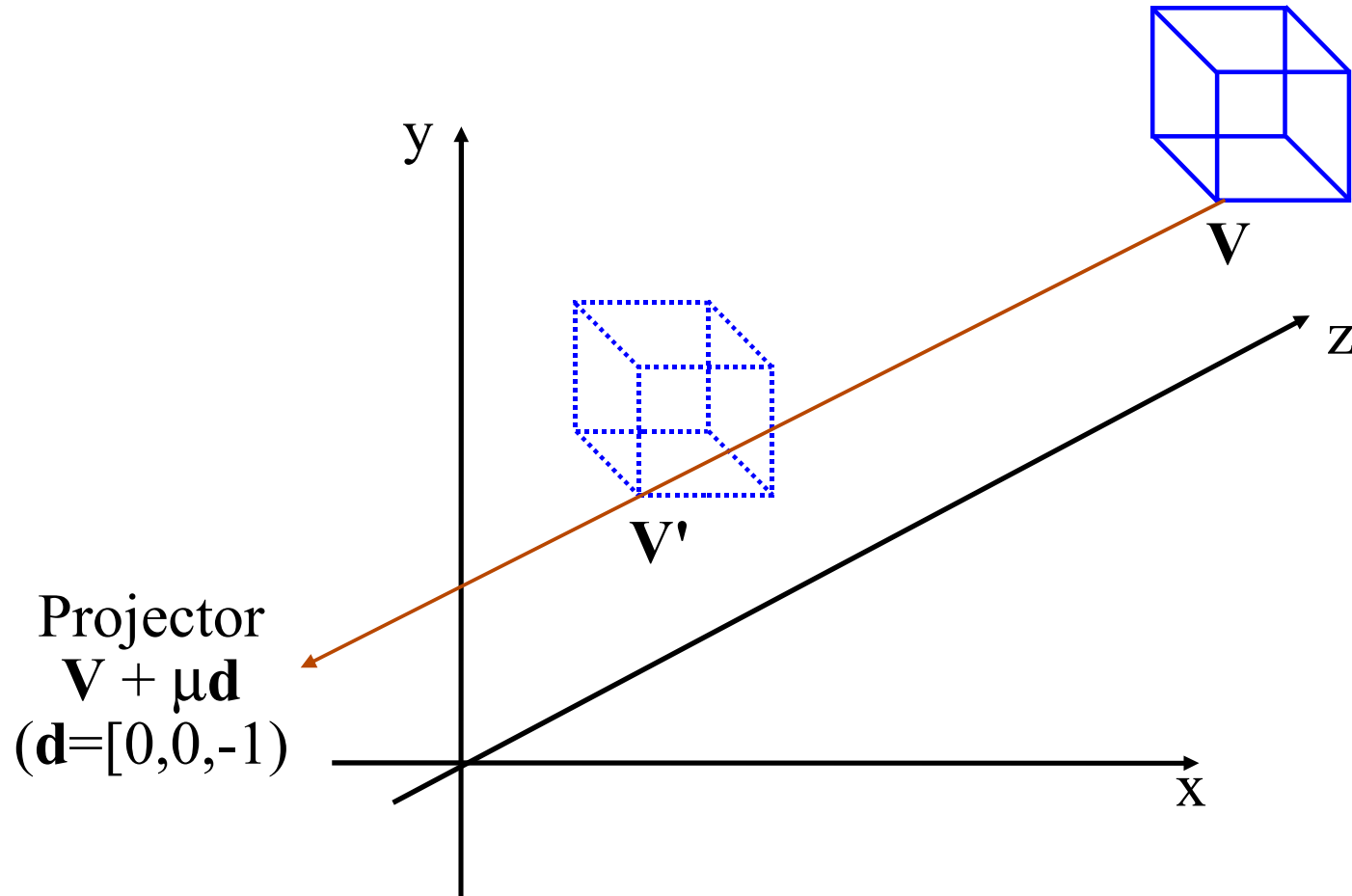
The viewpoint is at  $z = -\infty$

The plane of projection is  $z=0$

so

All projectors have direction  $d = [0,0,-1]$

# Orthographic Projection onto $z=0$



# *Calculating an Orthographic Projection*

Projector Equation:

$$\mathbf{P} = \mathbf{V} + \mu \mathbf{d} \quad (\text{from vertex } \mathbf{V})$$

Substitute  $\mathbf{d} = [0,0,-1]$

Yields cartesian form

$$P_x = V_x + 0 \quad P_y = V_y + 0 \quad P_z = V_z - \mu$$

The projection plane is  $z=0$  so the projected coordinate is

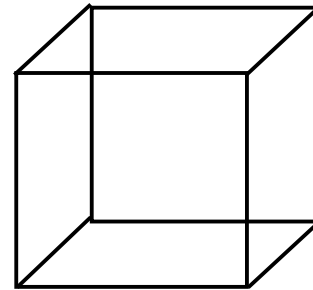
$$[V_x, V_y, 0]$$

ie we simply take the 3D x and y components of the vertex

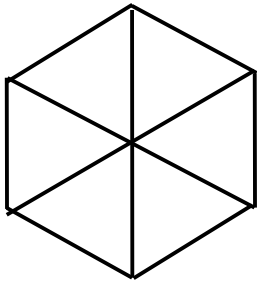
# *Orthographic Projection of a Cube*



Looking at a Face



General View



Looking at a vertex

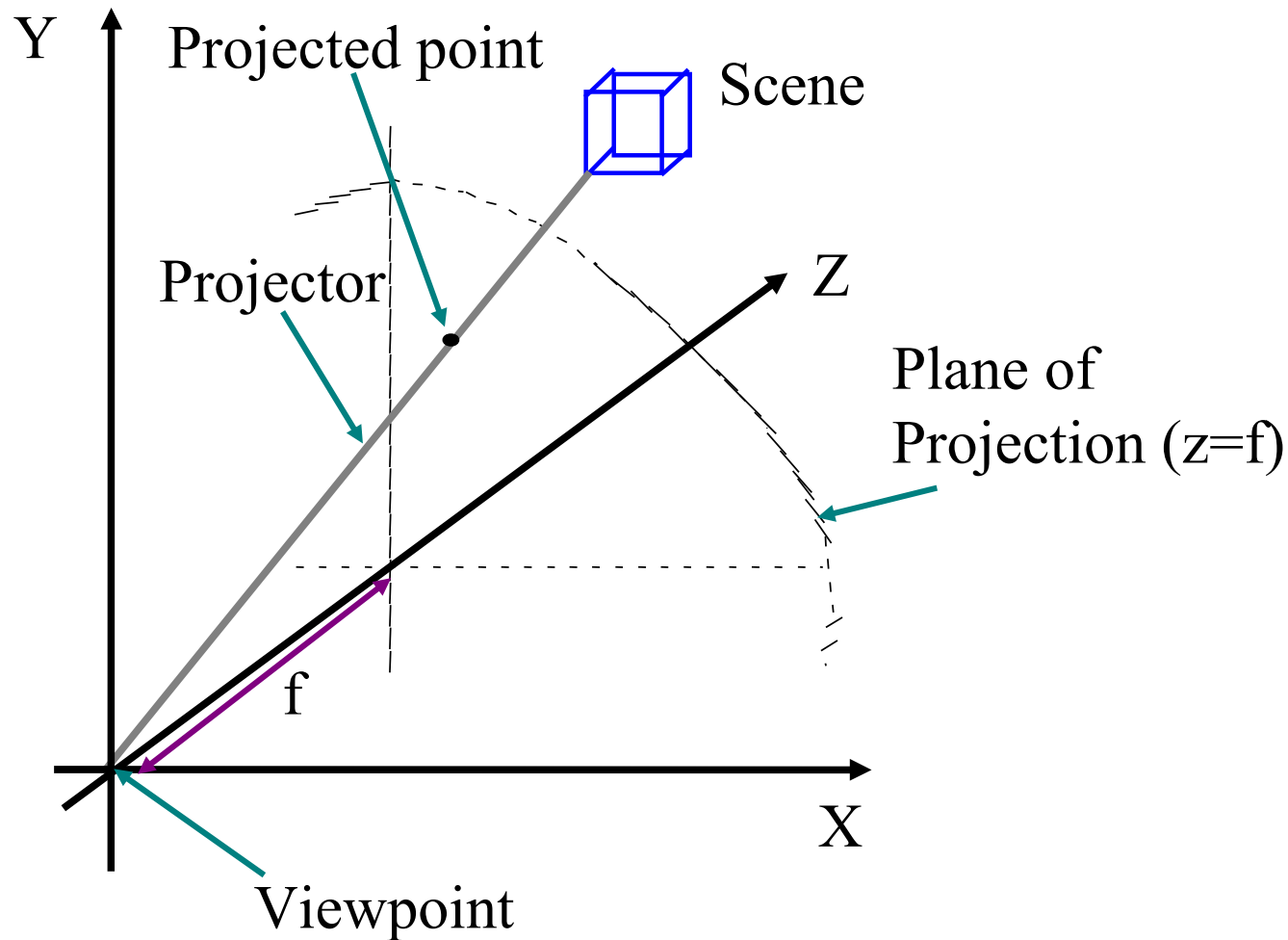
# *Perspective Projection*

Orthographic projection is fine in cases where we are not worried about depth (ie most objects are at the same distance from the viewer).

However for close work (particularly computer games) it will not do.

Instead we use perspective projection

# *Canonical Form for Perspective Projection*



# *Calculating Perspective Projection*

Projector Equation (from vertex  $\mathbf{V}$ ):

$$\mathbf{P} = \mu \mathbf{V} \quad (\text{all projectors go through the origin})$$

At the projected point  $P_z=f$

$$\mu_p = P_z/V_z = f/V_z$$

$$P_x = \mu_p V_x \text{ and } P_y = \mu_p V_y$$

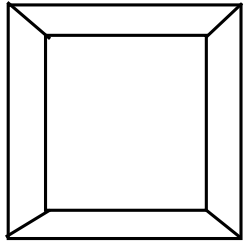
Thus

$$P_x = f V_x/V_z \text{ and } P_y = f V_y/V_z$$

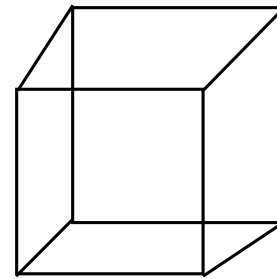
The constant  $\mu_p$  is sometimes called the foreshortening factor



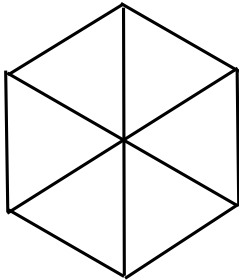
# *Perspective Projection of a Cube*



Looking at a Face



General View



Looking at a vertex

## *Problem Break*

Given that the viewpoint is at the origin, and the viewing plane is at  $z=5$ : What point on the viewplane corresponds to the 3D vertex  $\{10,10,10\}$  in

- a. Perspective projection
- b. Orthographic projection

## *Problem Break*

Given that the viewpoint is at the origin, and the viewing plane is at  $z=5$ : What point on the viewplane corresponds to the 3D vertex  $\{10,10,10\}$  in

- a. Perspective projection
- b. Orthographic projection

Perspective  $x' = f x/z = 5$  and  $y' = f y/z = 5$

Orthographic  $x' = 10$  and  $y' = 10$

## *The Need for Transformations*

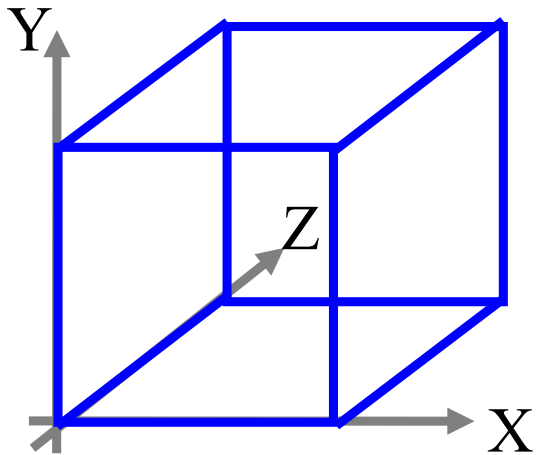
Graphics scenes are defined in a particular coordinate system, however we want to be able to draw a graphics scene from any angle

To draw a graphics scene we need the viewpoint to be the origin and the z axis to be the direction of view.

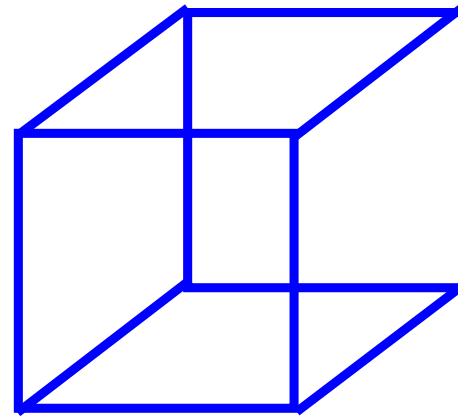
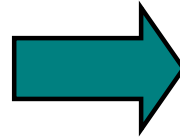
Hence we need to be able to transform the coordinates of a graphics scene.

# Transformation of viewpoint

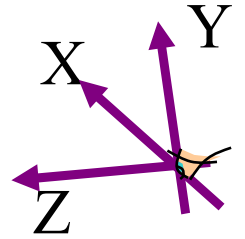
Required Viewpoint



Coordinate System  
for definition



Coordinate System  
for viewing



# *Other Transformations*

We also need transformations for other purposes:

## Animating Objects

eg flying titles rotating shrinking etc.

## Multiple Instances

the same object may appear at different places or different sizes

## Reflections and other special effects

## *Matrix transformations of points*

To transform points we use matrix multiplications, for example to make an object at the origin twice as big we could use:

$$[x', y', z'] = \begin{bmatrix} 2 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 2 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

which multiplied out gives:

$$x' = 2x, \quad y' = 2y, \quad z' = 2z$$

## *Translation by Matrix multiplication*

Many of our transformations will require translation of the points.

For example if we want to move all the points two units along the x axis we would require:

$$x' = x + 2$$

$$y' = y$$

$$z' = z$$

But how can we do this with a matrix?



# *Homogenous Coordinates*

The answer is to use 4D homogenous coordinates. The use of the fourth ordinate allows us to place a translation in the bottom row of the matrix.

$$[x', y', z', 1] = [x, y, z, 1] \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 2 & 0 & 0 & 1 \end{bmatrix}$$

multiplying out gives:

$$x' = x + 2, \quad y' = y, \quad z' = z$$

# *General Homogenous Coordinates*

In most cases the last ordinate will be 1, but in general it is a scale factor.

Thus, in the projection from 4D to 3D:

$[x, y, z, s]$  is equivalent to  $[x/s, y/s, z/s]$   
Homogenous Cartesian

# *Affine Transformations*

Affine transformations are those that preserve parallel lines.

Most transformations we require are affine, the most important being:

Scaling

Translating

Rotating

Other more complex transforms will be built from these three.

## *Translation*

We can apply a general translation by  $(t_x, t_y, t_z)$  to the points of a scene by using the following matrix multiplication.

$$[x, y, z, 1] \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ t_x & t_y & t_z & 1 \end{bmatrix} = [x + t_x, y + t_y, z + t_z, 1]$$

## *Inverting a translation*

Since we know what transformation matrices do, we can write down their inversions directly

For example:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ t_x & t_y & t_z & 1 \end{bmatrix} \text{ has inversion } \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -t_x & -t_y & -t_z & 1 \end{bmatrix}$$

# Scaling

Scaling simply multiplies each ordinate by a scaling factor. It can be done with the following homogenous matrix:

$$[x, y, z, 1] \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = [s_x x, s_y y, s_z z, 1]$$

## *Inverting scaling*

To invert a scaling we simply divide the individual ordinates by the scale factor.

$$\begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \text{ has inversion } \begin{bmatrix} 1/s_x & 0 & 0 & 0 \\ 0 & 1/s_y & 0 & 0 \\ 0 & 0 & 1/s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

## *Combining transformations*

Suppose we want to make an object at the origin twice as big and then move it to a point [5, 5, 20].

The transformation is a scaling followed by a translation.

$$[x', y', z', 1] = [x, y, z, 1] \begin{bmatrix} 2 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 5 & 5 & 20 & 1 \end{bmatrix}$$



## *Combined transformations*

We multiply out the transformation matrices first, then transform the points

$$[x', y', z', 1] = [x, y, z, 1] \begin{bmatrix} 2 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & 2 & 0 \\ 5 & 5 & 20 & 1 \end{bmatrix}$$

## *Transformations are not commutative*

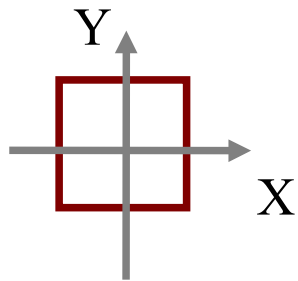
The order in which transformations are applied matters:

In general

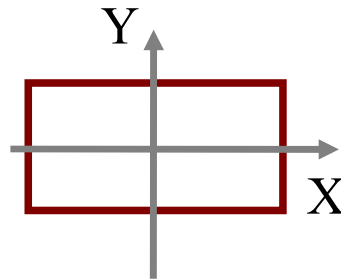
$T * S$  is not the same as  $S * T$

# The order of transformations is significant

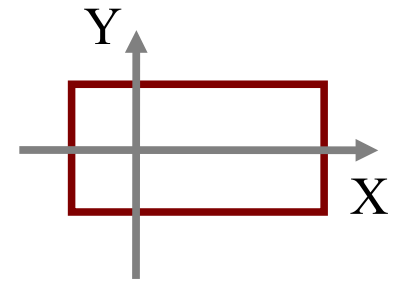
Graphics Scene  
(Square at origin)



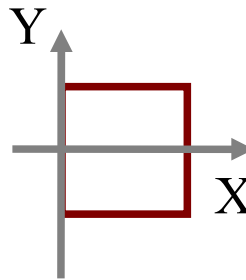
Scale  
 $x:=2x$



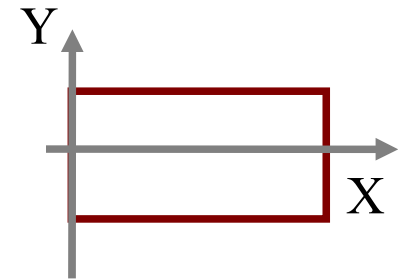
Translate  
 $x:=x+1$



Translate  
 $x:=x+1$



Scale  
 $x:=2x$



# *Rotation*

To define a rotation we need an axis.

The simplest rotations are about the Cartesian axes

eg

**$R_x$**  - Rotate about the X axis

**$R_y$**  - Rotate about the Y axis

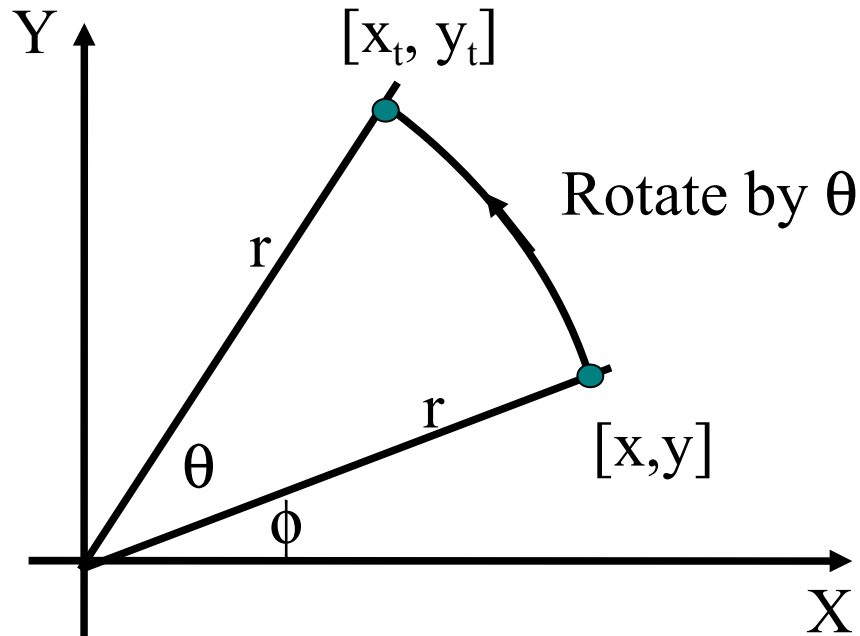
**$R_z$**  - Rotate about the Z axis

# Rotation Matrices

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

$$R_z = \begin{bmatrix} \cos\theta & \sin\theta & 0 & 0 \\ -\sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

# Deriving $R_z$



$$\begin{aligned} [x, y] &= [r \cos \phi, r \sin \phi] \\ [x_t, y_t] &= [r \cos(\phi + \theta), r \sin(\phi + \theta)] \\ &= [r \cos \phi \cos \theta - r \sin \phi \sin \theta, r \cos \phi \sin \theta + r \sin \phi \cos \theta] \\ &= [x \cos \theta - y \sin \theta, x \sin \theta + y \cos \theta] \\ &= [x, y] \begin{bmatrix} \cos \theta & \sin \theta \\ -\sin \theta & \cos \theta \end{bmatrix} \end{aligned}$$

# *Signs of Rotations*

Rotations have a direction.

The following rule applies to the matrix formulations given in the notes:

Rotation is clockwise when viewed from the positive side of the axis

## *Inverting Rotation*

Inverting a rotation by an angle  $\theta$  is equivalent to rotating through an angle of  $-\theta$ , now

$$\text{Cos}(-\theta) = \text{Cos}(\theta)$$

and

$$\text{Sin}(-\theta) = -\text{Sin}(\theta)$$



## *Inverting Rz*

To invert a rotation matrix simply change the sign of the sin terms.

$$\begin{bmatrix} \cos\theta & \sin\theta & 0 & 0 \\ -\sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \text{ has inversion } \begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$