Lecture 1: Three Dimensional graphics: Projections and Transformations

Device Independence

We will start with a brief discussion of two dimensional drawing primitives. At the lowest level of an operating system we have device dependent graphics methods such as:

SetPixel(XCoord,YCoord,Colour);
DrawLine(xs,ys,xf,yf);

which draw objects using pixel coordinates. However it is clearly desirable that we create any graphics application in a device independent way. If we can do this then we can re-size a picture, or transport it to a different operating system and it will fit exactly in the window where we place it. Many graphics APIs provide this facility, and it is a straightforward matter to implement it using a world coordinate system. This defines the coordinate values to be applied to the window on the screen where the graphics image will be drawn. Typically it will use a method of the kind:

SetWindowCoords(WXMin,WYMin,WXMax,WYMax);

WXMin etc are real numbers whose units are application dependent. If the application is to produce a visualisation of a house then the units could be meters, and if it is to draw accurate molecular models the units will be µm. The application program uses drawing primitives that work in these units, and converts the numeric values to pixels just before the image is rendered on the screen. This makes it easy to transport it to other systems or to upgrade it when new graphics hardware becomes available. There may be other device characteristics that need to be accounted for to achieve complete device independence, for example aspect ratios.

In order to implement a world coordinate system we need to be able to translate between world coordinates and the device or pixel coordinates. However, we do not necessarily know what the pixel coordinates of a window are, since the user can move and resize it without the program knowing. The first stage is therefore to find out what the pixel coordinates of a window are, which is done using an enquiry procedure of the kind:

GetWindowPixelCoords(DXmin, DYmin, DXmax, DY max)

In the Windows API this procedure is called GetClientRect. Having established both the world and device coordinate systems, it is possible to define a normalisation process to compute the pixel coordinates from the world coordinates. This is done by simple ratios as shown in the figure 1. For the X direction:

(Xw-WXmin)/(WXMax-WXMin) = (Xd - DXMin)/(DXMax-DXMin)

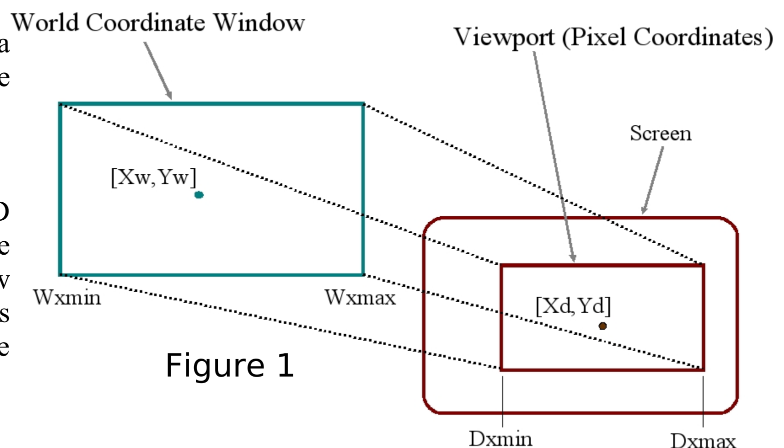


Figure 1

Rearranging, and applying the same idea to the Y direction yields a pair of simple linear equations equations:

Xd = Xw * A + B;
Yd = Yw * C + D;

where the four constants A, B C and D define the normalisation between the world coordinate system and the window pixel coordinates. Whenever a window is re-sized it is necessary to re-calculate the constants A,B,C and D.

Graphical Input

The most important input device is the mouse, which records the distance moved in the X and Y directions. In the simplest form it provides at least three pieces of information: the x distance moved, the y distance moved and the button status. The mouse causes an interrupt every time it is moved, and it is up to the system software to keep track of the changes. Note that the mouse is not connected with the screen in any way. Either the operating system or the application program must achieve the connection by drawing the visible marker.

The operating system must share control of the mouse with the application, since it needs to act on mouse actions that take place outside the graphics window. For instance, processing a menu bar or launching a different application. It therefore traps all mouse events (ie changes in position or buttons) and informs the

program whenever an event has taken place using a "callback procedure". The application program must, after every action carried out, return to the callback procedure (or event loop) to determine whether any mouse action (or other event such as a keystroke) has occurred. The callback is the main program part of any application, and, in simplified pseudo code, looks like this:

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

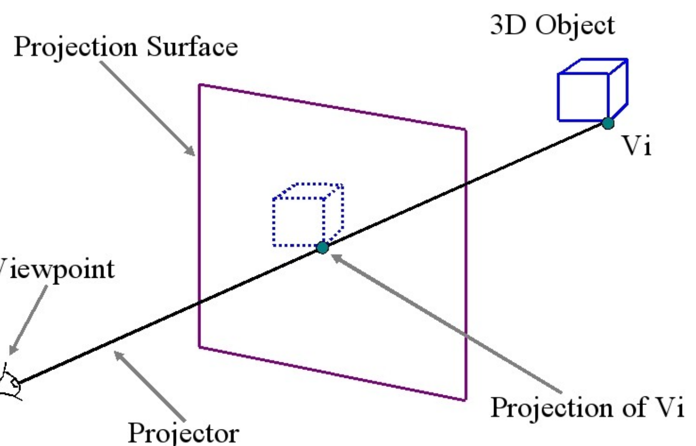
The procedure *ProcessMenuRequest* will be used to launch all the normal actions, such as save and open and quit, together with all the application specific requests. The procedures *GetMouseCoordinates* and *PerformMouseProcess* will be used by the application writer to create whatever effect is wanted, for example, moving an object with the mouse. This may well involve re-drawing the graphics. If the window is re-sized then the whole picture will be re-drawn.

3-Dimensional Objects Bounded by Planar Polygons (Facets)

Most graphical scenes are made up of planar facets. Each facet is an ordered set of 3D vertices, lying on one plane, which form a closed polygon. The data describing a facet are of two types. First, there is the numerical data which is a list of 3D points, (3*N numbers for N points), and secondly, there is the topological data which describes how points are connected to form edges and facets.

Projections of Wire-Frame Models

Since the display device is only 2D, we have to define a transformation from the 3D space to the 2D surface of the display device. This transformation is called a *projection*. In general, projections transform an n-dimensional space into an m-dimensional space where $m<n$. Projection of an object onto a surface is done by selecting a viewpoint and then defining projectors or lines which join each vertex of the object to the viewpoint. The projected vertices are placed where the projectors intersect the projection surface.



The most common (and simplest) projections used for viewing 3D scenes use planes for the projection surface and straight lines for projectors. These are called planar geometric projections. A rectangular window can be defined defined on the plane of projection which can be mapped into the device window as described above. Once all the vertices of an object have been projected it can be rendered. An easy way to do this is drawing all the projected edges. This is called a wire-frame representation. Note that for such rendering the topological information defining the facets is not required.

There are two common classes of planar geometric projections. Parallel projections use parallel projectors, perspective projections use projectors which pass through one single point called the viewpoint. In order to minimise confusion in dealing with a general projection problem, we can standardise the plane of projection by making it always parallel to the z=0 plane, (the plane which contains the x and y axis). This does not limit the generality of our discussion because if the required projection plane is not parallel to the z=0 plane then we can use a coordinate transformations in 3D and make so. We will see shortly how to do this. We shall restrict the viewed objects to be in the positive half space (z>0), therefore the projectors starting at the vertices will always run in the negative z direction.

Parallel Projections

If the direction of a projector is given by vector **d**=[d_x, d_y, d_z], and it passes through the vertex **V**=[V_x, V_y, V_z] it may be expressed by the parametric line equation:

$$\mathbf{P} = \mathbf{V} + \mu \mathbf{d}$$

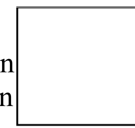
In *orthographic* projection the projectors are perpendicular to the projection plane, which we define as z=0. In this case the projectors are in the direction of the z axis and:

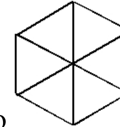
$$\mathbf{d} = [0,0,-1]$$

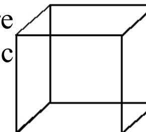
and so $P_x = V_x$

and $P_y = V_y$

which means that the x and y co-ordinates of the projected vertex is equal to the x and y co-ordinates of the vertex itself and no calculations are necessary. Some examples of a wireframe cube drawn in orthographic projection are shown opposite.







If the projectors are not perpendicular to the plane of projection then the projection is called *oblique*. The projected vertex intersects the z=0 plane where the z component of the **P** vector is equal to zero, therefore:

$$P_z = 0 = V_z + \mu\, d_z$$

so $\mu = - V_z / d_z$

and we can use this value of μ to compute:

$$P_x = V_x + \mu\, d_x = V_x - d_x V_z / dz$$

and $P_y = V_y + \mu\, d_y = V_y - d_y V_z / d_z$

These projections are similar to the orthographic projection with one or other of the dimensions scaled. They are not often used in practice.

Perspective Projections

In perspective projection, all the rays pass through one point in space, the centre of projection as shown in the figure 2. If the centre of projection is behind the plane of projection then the orientation of the image is the same as the image. By contrast, in a pin hole camera it is inverted. To calculate perspective projections we adopt a canonical form in which the centre of projection is at the origin, and the projection plane is placed at a constant z value, z=f. The projection of a 3D point onto the z=f plane is calculated as follows. If we are projecting the point **V** then the projector has equation:

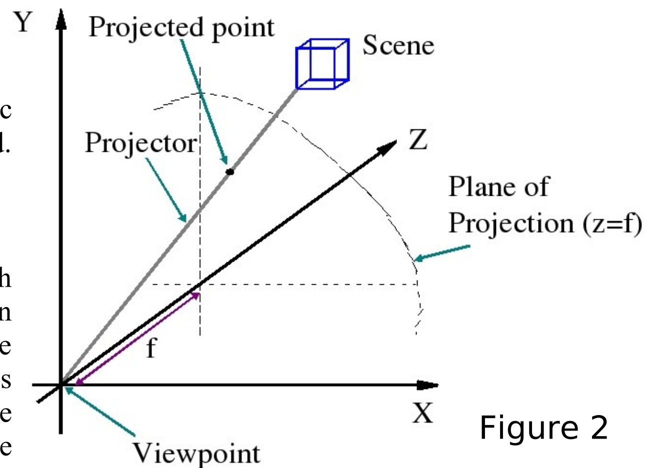


Figure 2

$$P = \mu \mathbf{V}$$

Since the projection plane has equation z=f, it follows that, at the point of intersection:

$$f = \mu\, V_z$$

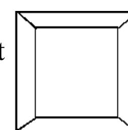
If we write $\mu_p = f / V_z$ for the intersection point on the plane of projection then:

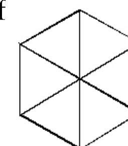
thus

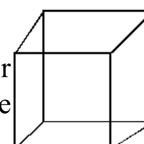
$$P_x = \mu_p V_x = f^* V_x / V_z$$

and $P_y = \mu_p V_y = f^* V_y / V_z$

The factor μ_p is called the foreshortening factor, because the further away an object is, the larger V_z and the smaller is its image. Some examples of the perspective projection of a cube are shown opposite.



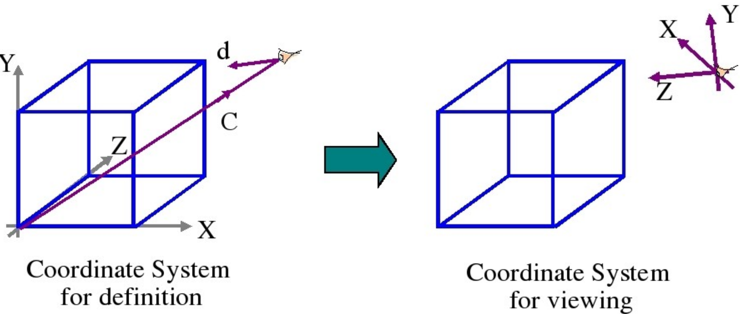




The introduction of canonical forms for perspective and orthographic projection simplifies their computation. However it means that we must be able to transform a scene, which could be defined in any

3D coordinate system, such that the view direction is along the z axis and (for perspective projection) the viewpoint is at the origin. In general we would like to change the coordinates of every point in the scene, such that some chosen viewpoint **C** = [Cx,Cy,Cz] is the origin and some view direction **d** = [dx,dy,dz] is the Z axis.

Frequently, we may also want to transform the points of a graphical scene for other purposes such as generation of special effects like rotating or shrinking objects. Transformations of this kind are achieved by multiplying every point of the scene by a transformation matrix. Unfortunately however, we cannot perform a general translation using normal Cartesian coordinates, and for that reason we now introduce a system called *homogeneous coordinates*. Three dimensional points expressed in homogeneous form have a fourth ordinate:



P = $[p_x, p_y, p_z, s]$

The fourth ordinate is a scale factor, and conversion to Cartesian form is achieved by dividing it into the other ordinates, so

$[p_x, p_y, p_z, s]$ has Cartesian coordinate equivalent $[p_x/s, p_y/s, p_z/s]$.

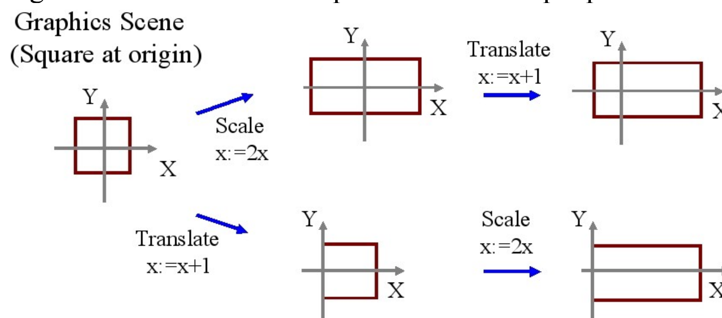
In most cases *s* will be 1. The point of introducing homogenous coordinates is to allow us to translate the points of a scene by using matrix multiplication.

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

The matrix for scaling a graphical scene is also easily expressed in homogenous form:

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

Notice that these two transformations are not commutative, and it is essential that they are carried out in the correct order. The figure below illustrates the problem for a simple picture.

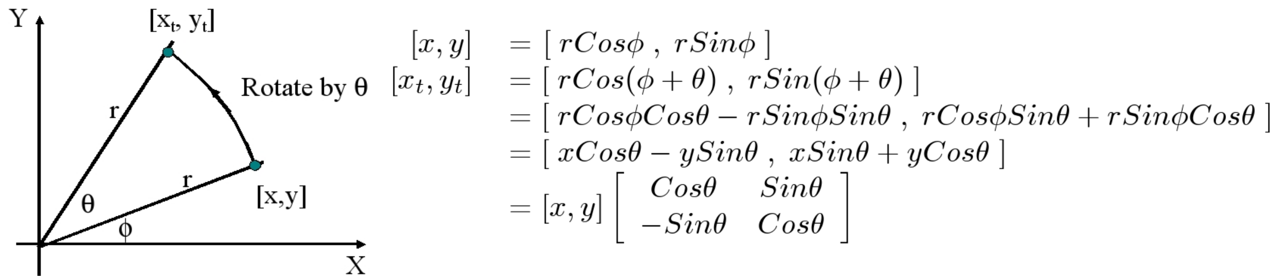


Rotation has to be treated differently since we need to specify an axis. The matrices for rotation about the three Cartesian axes are:

$$R_x = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & Cos\theta & Sin\theta & 0 \\ 0 & -Sin\theta & Cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad R_y = \begin{bmatrix} Cos\theta & 0 & -Sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ Sin\theta & 0 & Cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad R_z = \begin{bmatrix} Cos\theta & Sin\theta & 0 & 0 \\ -Sin\theta & Cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Some care is required with the signs. The above formulation obeys the conventions of a left hand axis system. That is, if the positive y-axis is taken as vertical, and the positive x-axis horizontal to the right, the

positive z-axis is into the page. In these cases, rotation is in a clockwise direction when viewed from the positive side of the axis, or vice versa, anti-clockwise when viewed from the negative side of the axis. The derivation of the ***Rz*** matrix is as follows:



$$
\begin{aligned}
[x, y] &= [\ rCos\phi\ ,\ rSin\phi\ ] \\
[x_t, y_t] &= [\ rCos(\phi+\theta)\ ,\ rSin(\phi+\theta)\ ] \\
&= [\ rCos\phi Cos\theta - rSin\phi Sin\theta\ ,\ rCos\phi Sin\theta + rSin\phi Cos\theta\ ] \\
&= [\ xCos\theta - ySin\theta\ ,\ xSin\theta + yCos\theta\ ] \\
&= [x, y]\begin{bmatrix} Cos\theta & Sin\theta \\ -Sin\theta & Cos\theta \end{bmatrix}
\end{aligned}
$$

The others may be derived similarly. Inversions of the transformation matrices can be computed easily, without recourse to Gaussean elimination, by considering the meaning of each transformation. For scaling, we substitute $1/s_x$ for s_x, $1/\ s_y$ for s_y and $1/s_z$ for s_z to invert the matrix.

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

For translation we substitute $-t_x$ for t_x, $-\ t_y$ for t_y and $-\ t_z$ for t_z.

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

For the rotation matrices we note that:

$Cos(-\theta) = Cos(\theta)$ and $Sin(-\theta) = -Sin(\theta)$

Hence to invert the matrix we simply change the sign of the Sin terms.

$$
\begin{bmatrix} Cos\theta & Sin\theta & 0 & 0 \\ -Sin\theta & Cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \text{has inversion} \begin{bmatrix} Cos\theta & -Sin\theta & 0 & 0 \\ Sin\theta & Cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}
$$