# 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 $\mu m$. 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.

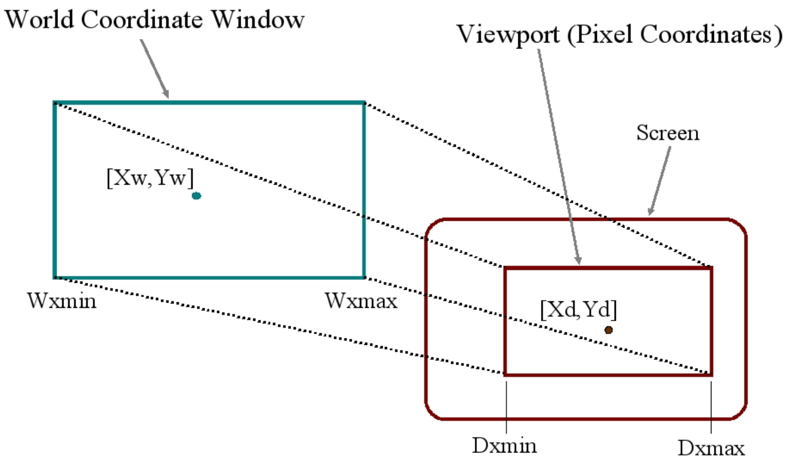


Figure 1: Normalisation transformation

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, Dymax)

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 Figure 1 . For the X direction:

$$\frac{(X_w - W_{xmin})}{(W_{xmax} - W_{xmin})} = \frac{(X_d - D_{xmin})}{(D_{xmax} - D_{xmin})}$$

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

$$X_d = AX_w + B$$
$$Y_d = CY_w + D$$

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 Perform-MouseProcess 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, ($3 \times N$ 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 as shown in Figure 2.

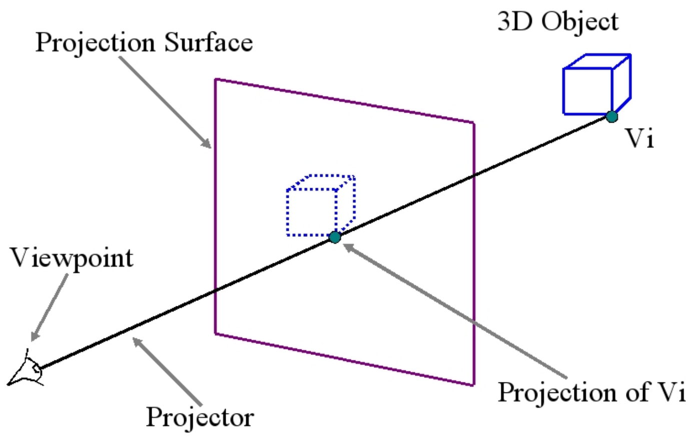


Figure 2: Planar Projection

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 only specifies which points are joined by edges. For other forms of rendering we also need to define the object faces.

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

In a parallel projection all the projectors have the same direction $\mathbf{d}$, and the viewpoint can be considered to be at infinity. For a vertex $\mathbf{V} = [V_x, V_y, V_z]$ the projector is defined 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 usually 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 $P_y = V_y$

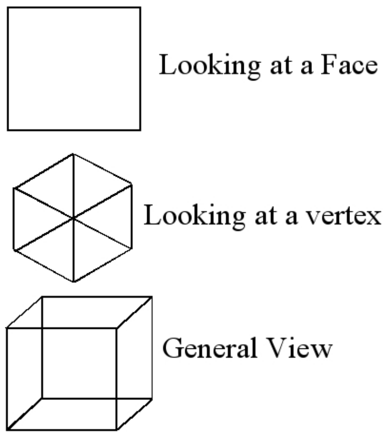


Figure 3: Orthographic Projections

which means that the $x$ and $y$ co-ordinates of the projected vertex are 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 in Figure 3.

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 $\mathbf{P}$ 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 $\mu$ to compute:

$$P_x = V_x + \mu d_x = V_x - d_x V_z/d_z$$

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 figure 2. If the centre of projection is behind the plane of projection then the orientation of the image is the same as the 3D object. 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, $z = f$. This canonical form is illustrated in Figure 4. The projection of a 3D point onto the $z = f$ plane is calculated as follows. If we are projecting the point $\mathbf{V}$ then the projector has equation:

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

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:

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

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

The factor $\mu_p$ 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 in figure 5.

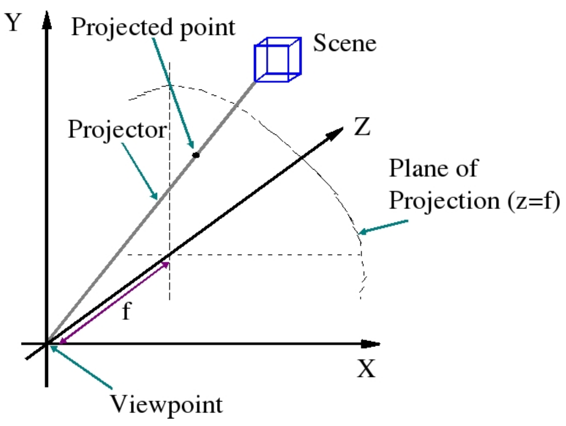


Figure 4: Canonical form for perspective projection

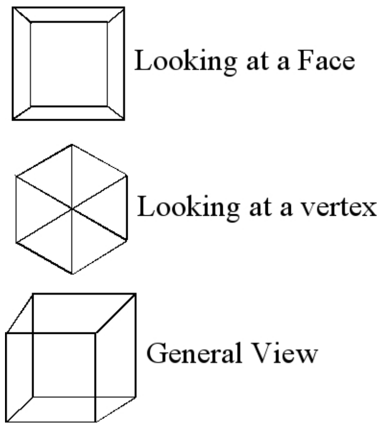


Figure 5: Perspective projection of a cube

## Space Transformations

The introduction of canonical forms for perspective and orthographic projection simplifies their computation. However, in cases where we wish to move around a graphical scene and view if from any particular point, we must be able to transform the coordinates of the scene, 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 $\mathbf{C} = [C_x, C_y, C_z]$ is the origin and some view direction $\mathbf{d} = [d_x, d_y, d_z]$ is the $Z$ axis. This new coordinate system in which the scene is to be defined is sometimes called the "view centered" coordinate system and is shown in Figure 6.

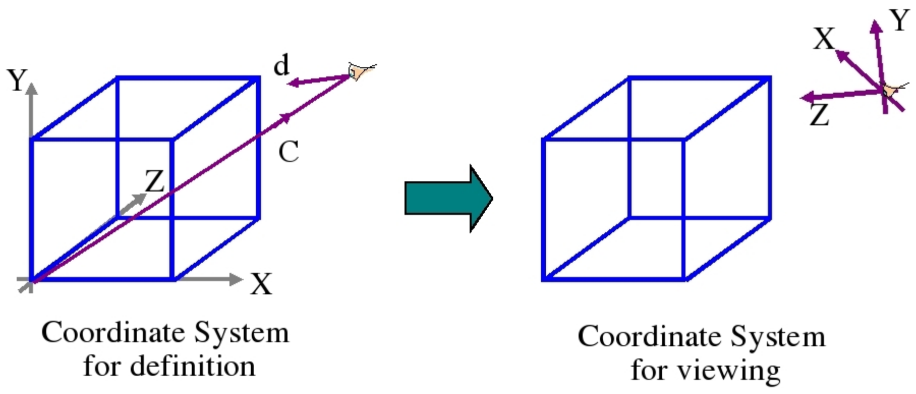


Figure 6: View centered coordinate transformation

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:

$$\mathbf{P} = [p_x, p_y, p_z, s]$$

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. Figure 7 illustrates the problem for a simple picture.

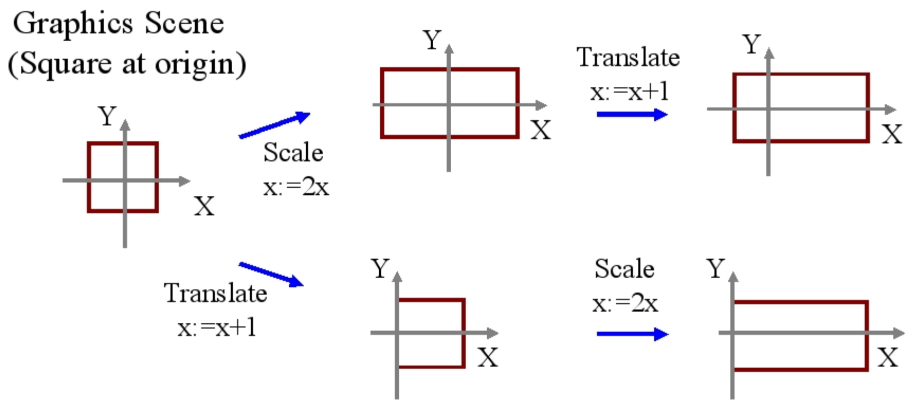


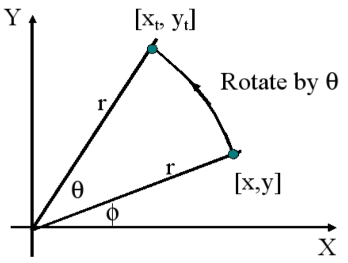
Figure 7: The importance of the order in which transfrmations are done

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

$$\mathcal{R}_x = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & Cos\theta & Sin\theta & 0 \\ 0 & -Sin\theta & Cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \mathcal{R}_y = \begin{bmatrix} Cos\theta & 0 & -Sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ Sin\theta & 0 & Cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \mathcal{R}_z = \begin{bmatrix} Cos\theta & Sin\theta & 0 & 0 \\ -Sin\theta & Cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

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 $\mathcal{R}_z$ 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:

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

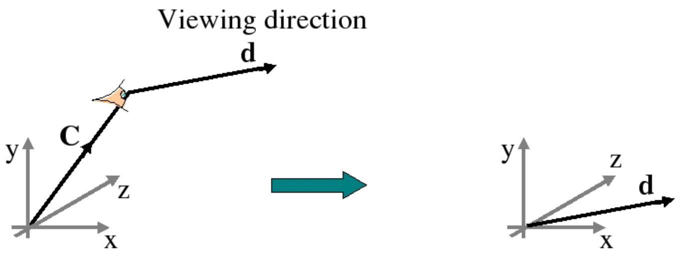
Hence to invert the matrix we simply change the sign of the $Sin$ terms, for example:

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

# Lecture 2: Scene Transformation and Animation

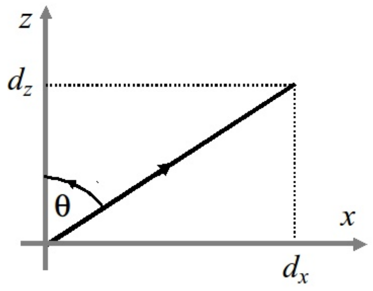
## Flying Sequences

We will now consider a most important subject, namely, scene transformation. In any viewer centered application, such as a flight simulator or a computer game, we need to view the scene from a moving position. As the viewpoint changes we transform all the coordinates of the scene such that the viewpoint is the origin and the view direction is the $z$ axis, before projecting and drawing the scene. Let us suppose that, in the coordinate system in which the scene is defined we wish to view it from the point $\mathbf{C} = [C_x, C_y, C_z]$, looking along the direction $\boldsymbol{d} = [d_x, d_y, d_z]$. The first step is to move the origin to $\mathbf{C}$ for which we use the transformation matrix $\mathcal{A}$.



$$\mathcal{A} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -C_x & -C_y & -C_z & 1 \end{bmatrix}$$

Following this, we wish to rotate about the $y$-axis so that $\mathbf{d}$ lies in the plane $x = 0$. Using the fact that $\mathbf{d}$ is defined by the co-ordinates $[d_x d_y d_z]$ and using the notation $v^2 = d_x^2 + d_z^2$ this is done by matrix $\mathcal{B}$.



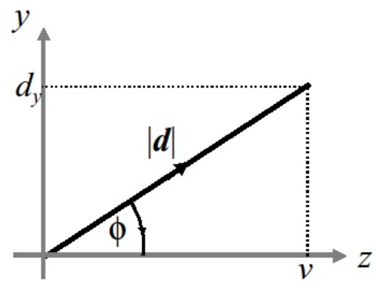
$$v = \sqrt{d_x^2 + d_z^2}$$

$$Cos\theta = d_z/v$$

$$Sin\theta = d_x/v$$

$$\mathcal{B} = \begin{bmatrix} d_z/v & 0 & d_x/v & 0 \\ 0 & 1 & 0 & 0 \\ -d_x/v & 0 & d_z/v & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Notice that we have avoided computing the $Cos$ and $Sin$ functions for this rotation by use of the direction cosine. To get the direction vector lying along the $z$ axis a further rotation is needed. This time it is about the $x$ axis using matrix $\mathcal{C}$.



$$v = \sqrt{d_x^2 + d_z^2}$$

$$Cos\phi = v/|d|$$

$$Sin\phi = d_y/|d|$$

$$\mathcal{C} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & v/|d| & d_y/|d| & 0 \\ 0 & -d_y/|d| & v/|d| & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Finally the transformation matrices are combined into one, and each point of the scene is transformed:

$$\mathcal{T} = \mathcal{A} \cdot \mathcal{B} \cdot \mathcal{C}$$

and so for all the points

$$\mathbf{P_t} = \mathbf{P} \cdot \mathcal{T}$$

## Problems with verticals

The concept of "vertical" is missing from the above analysis, and needs attention since it is easy to invert the vertical. This will be easily observed in the trivial example of figure 1, where an arrow whose base is at [0,0,-l] is being observed from the origin. A transformation based on rotating about the $y$ axis first yields the correct solution. However, a transformation involving rotation about the $x$ axis first inverts the image.

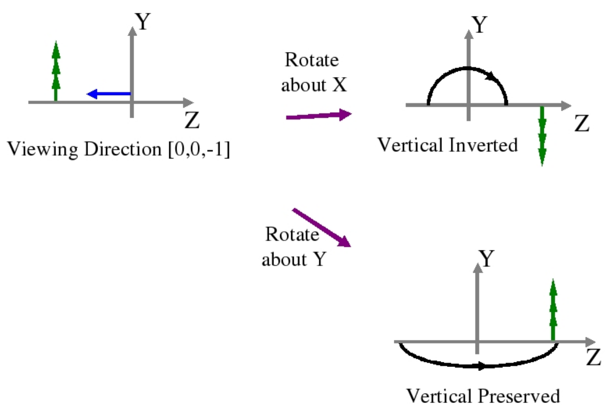


Figure 1: Inversion of the Vertical

## Rotation about a general line

A very similar problem to the viewer centered transformation concerns animation of objects in a fixed scene. Let us suppose that we want to rotate one sub-object about some line in the Cartesian space where the scene is defined. Let the line be: $\mathbf{L} + \mu\mathbf{d}$ where $\mathbf{L} = [L_x, L_y, L_z]$ is the position vector of any point on the line and $\mathbf{d}$ is a unit direction vector along the line. We use a translation to move the origin so that it is on the line (matrix $\mathcal{A}$ but transforming the origin to $\mathbf{L}$ rather than $\mathbf{C}$) followed by two rotations to make the $z$ axis coincident with the direction vector $\mathbf{d}$ (matrices $\mathcal{B}$ and $\mathcal{C}$). Now we can perform a rotation of the object about the $z$-axis using the standard rotation matrix $\mathcal{R}_z$ defined previously. Finally we need to restore the coordinate system as it was, such that the viewpoint is the same as before. To do this we simply invert the transformation matrices $\mathcal{A}$ $\mathcal{B}$ and $\mathcal{C}$. As before we multiply all the individual transformation matrices together to make one matrix which is then applied to all the points.

$\mathcal{T} = \mathcal{A} \cdot \mathcal{B} \cdot \mathcal{C} \cdot \mathcal{R}_z \cdot \mathcal{C}^{-1} \cdot \mathcal{B}^{-1} \cdot \mathcal{A}^{-1}$

and for all the points $\mathbf{P_t} = \mathbf{P} \cdot \mathcal{T}$

Other object transformations for graphical animation are performed similarly. For example to make an object shrink we move the origin to its centre, perform a scaling and then restore the origin to its original position.

## Projection by Matrix Multiplication

If we use homogeneous co-ordinates then it is also possible to compute projection by multiplication by a projection matrix. Placing the centre of projection at the origin and using $z = f$ as the projection plane gives us matrix $\mathcal{M}_p$ for perspective projection. Matrix $\mathcal{M}_o$ is for orthographic projection:

$$\mathcal{M}_p = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1/f \\ 0 & 0 & 0 & 0 \end{bmatrix} \qquad \mathcal{M}_o = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

It is not immediately obvious that matrix $\mathcal{M}_p$ produces the correct perspective projection. Let us transform an arbitrary point $\mathbf{V}$ with homogeneous co-ordinates $[x, y, z, 1]$ by using matrix multiplication. For the projected point $\mathbf{P}$ we get:

$$\mathbf{P} = \mathbf{V} \cdot \mathcal{M}_p = [x, y, z, z/f]$$

This point must be normalised into a Cartesian coordinate. To do this we divide the first three co-ordinates by the value of the fourth and we get:

$$\mathbf{P_c} = [xf/z, yf/z, f, 1]$$

which is the projected point. It is interesting to note that the projection matrix is obviously singular (it has a row of zeros) and, therefore, it has no inverse. This must be so because it is impossible to reconstruct a 3D object from its 2D projection without other information. Projection matrices can of course be combined with the other matrices. Indeed the popularity of the orthographic projection is that it simplifies the amount of calculations since the $z$ row and column fall to zero. Although a simplification applies when the perspective projection is used, there is also the need to normalise the resulting homogenous coordinates, and this adds to the computation time.

## Homogenous coordinates

We now take a second look at homogeneous coordinates, and their relation to vectors. Previously, we described the fourth ordinate as a scale factor, and ensured that, with the exception of the projection transformation, it was always normalised to 1. As an alternative, we can consider the fourth ordinate as indicating a type as follows. Informally we acknowledge that a normal Cartesian coordinate is a special form of vector, which we call a position vector, and usually denote using bold face capital letters. Hence, we can say that a normalised homogenous coordinate is the same as a position vector.

By contrast, if we consider a homogenous coordinate where the last ordinate is zero - $[x, y, z, 0]$ - we find that we cannot normalise it in the usual way because of the divide by zero. So clearly a homogenous coordinate of this form cannot be directly associated with a point in Cartesian space. However, it still has a magnitude and direction, and hence we can consider it to be a direction vector. Thus homogenous coordinates fall into two classes, those with the final ordinate non-zero, which can be normalised into position vectors or points, and those with zero in the final ordinate which are direction vectors, or vectors in the pure sense of the word.

Consider now how vector addition works with these definitions. If we add two direction vectors, we add the ordinates as before and we obtain a direction vector. ie:

$$[x_i, y_i, z_i, 0] + [x_j, y_j, z_j, 0] = [x_i + x_j, y_i + y_j, z_i + z_j, 0]$$

This is the normal vector addition rule which operates independently of Cartesian space. However, if we add a direction vector to a position vector we obtain a position vector or point:

$$[X_i, Y_i, Z_i, 1] + [x_j, y_j, z_j, 0] = [X_i + x_j, Y_i + y_j, Z_i + z_j, 1]$$

This is a nice result, because it ties in with our definition of a straight line in Cartesian space, which is the sum of a point and a scaled direction as shown in figure 2.

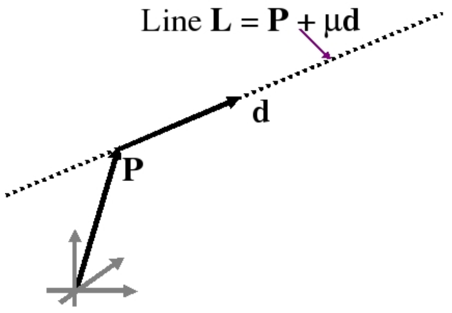


Figure 2: Definition of a straight line

Now, consider a general affine transformation matrix. We ignore the possibility of doing perspective projection or shear, so that the last column will always be $[0, 0, 0, 1]^T$, and the matrix will be of the form shown in figure 3, with the rows viewed as three direction vectors and a position vector.

$$\begin{array}{l} \left[\begin{array}{cccc} q_x & q_y & q_z & 0 \\ r_x & r_y & r_z & 0 \\ s_x & s_y & s_z & 0 \\ C_x & C_y & C_z & 1 \end{array}\right] \end{array} \begin{array}{l} \text{Direction vector} \\ \text{Direction vector} \\ \text{Direction vector} \\ \text{Position vector} \end{array}$$

Figure 3: The structure of a space transformation matrix

To see what the individual rows mean we consider the effect of the transformation in simple cases. For example take the unit vectors along the Cartesian axes eg $\boldsymbol{i} = [1, 0, 0, 0]$:

$$[1, 0, 0, 0] \left[\begin{array}{cccc} q_x & q_y & q_z & 0 \\ r_x & r_y & r_z & 0 \\ s_x & s_y & s_z & 0 \\ C_x & C_y & C_z & 1 \end{array}\right] = [q_x, q_y, q_z, 0]$$

In other words the direction vector of the top row represents the direction in which the $x$ axis points after transformation, and similarly we find that $\boldsymbol{j} = [0, 1, 0, 0]$ will be transformed to direction $[r_x, r_y, r_z, 0]$ and $\boldsymbol{k} = [0, 0, 1, 0]$ will be transformed to $[s_x, s_y, s_z, 0]$. Similarly, we can see the effect of the bottom row by considering the transformation of the origin which has homogeneous coordinate $[0, 0, 0, 1]$. This will be transformed to $[Cx, Cy, Cz, 1]$.

$$[0, 0, 0, 1] \left[\begin{array}{cccc} q_x & q_y & q_z & 0 \\ r_x & r_y & r_z & 0 \\ s_x & s_y & s_z & 0 \\ C_x & C_y & C_z & 1 \end{array}\right] = [C_x, C_y, C_z, 1]$$

Notice also that the zero in the last ordinate ensures that direction vectors will not be affected by the translation, whereas all position vectors will be moved by the same factor. Notice also that if we do not shear the object the three vectors $\mathbf{q}$ $\mathbf{r}$ and $\mathbf{s}$ will remain orthogonal so that $\mathbf{q} \cdot \mathbf{r} = \mathbf{r} \cdot \mathbf{s} = \mathbf{q} \cdot \mathbf{s} = 0$. Unfortunately however, this analysis does not help us to determine the transformation matrix, though we will need it later in the course when we look at annimation. In general it would be more natural to assume that we know the vectors ***u***,***v***, and ***w*** which form the view centered coordinate system. To see how to write down a transformation matrix from thses vectors we need to introduce the notion of the dot product as a projection onto a line. This is most readily seen in two dimensions as shown in Figure 4. By dropping perpendiculars from the point $\mathbf{P}$ to the line defined by vector ***u*** and vector ***v*** we see that the distances from the origin are respectively $\mathbf{P} \cdot \boldsymbol{u}$ and $\mathbf{P} \cdot \boldsymbol{v}$.

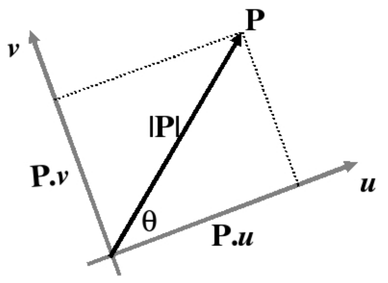


Figure 4: The dot product as projection

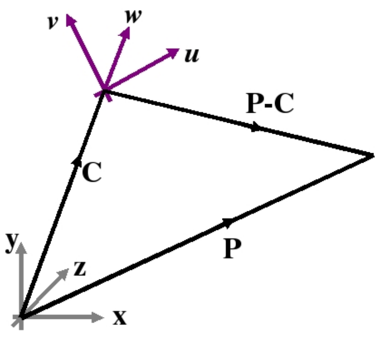


Figure 5: Three dimensional point projection

$\mathbf{P} \cdot \boldsymbol{u} = |\mathbf{P}||\boldsymbol{u}|Cos\theta = |\mathbf{P}|Cos\theta$ since $\boldsymbol{u}$ is a unit vector. Now, suppose that we wish to rotate the scene so that the new $x$ and $y$ axes were the $\boldsymbol{u}$ and $\boldsymbol{v}$ vectors, then the $x$ ordinate would be defined by $\mathbf{P} \cdot \boldsymbol{u}$ and the $y$ by $\mathbf{P} \cdot \boldsymbol{v}$.

The generalisation to three dimensions is shown in figure 5, in which a point $\mathbf{P}$ is transformed into the $(u, v, w)$ axis system, translated by vector $\mathbf{C}$, is given by:

$$\begin{aligned} P^t_x &= (\mathbf{P} - \mathbf{C}) \cdot \boldsymbol{u} \\ P^t_y &= (\mathbf{P} - \mathbf{C}) \cdot \boldsymbol{v} \\ P^t_z &= (\mathbf{P} - \mathbf{C}) \cdot \boldsymbol{w} \end{aligned}$$

Expressing this as a transformation matrix we get:

$$[P^t_x, P^t_y, P^t_z, 1] = [P_x, P_y, P_z, 1] \begin{bmatrix} u_x & v_x & w_x & 0 \\ u_y & v_y & w_y & 0 \\ u_z & v_z & w_z & 0 \\ -\mathbf{C} \cdot \boldsymbol{u} & -\mathbf{C} \cdot \boldsymbol{v} & -\mathbf{C} \cdot \boldsymbol{w} & 1 \end{bmatrix}$$

So now by maintaining values of $\boldsymbol{u}$, $\boldsymbol{v}$, $\boldsymbol{w}$ and $\boldsymbol{w}$ throughout an animation sequence, for example by adjusting their values in response to mouse or joystick commands, we can simply write down the correct scene transformation matrix for viewing the virtual world from the correct position.

Finally we will derive the matrix used for a view centered transformation with the viewer at point $\mathbf{C}$ and looking in direction $\mathbf{d}$ by finding the new axis system $\mathbf{u}$, $\boldsymbol{v}$, $\boldsymbol{w}$, and then deducing the transformation matrix. The direction vector for viewing is $\mathbf{d} = [d_x, d_y, d_z]$ and this must lie along the $\boldsymbol{w}$ direction, so $\boldsymbol{w} = \mathbf{d}/|\mathbf{d}|$. We first find any two vectors, $\mathbf{p}$ and $\mathbf{q}$,which have the same directions as $\mathbf{u}$ and $\mathbf{v}$, but do not necessarily have unit length. We can constrain the system to preserve horizontals and verticals in two ways: (i) Vector $\mathbf{p}$ will be in the direction of the new $x$-axis, so to preserve the horizontal, we choose: $p_y = 0$, and (ii) Vector $\mathbf{q}$ will be the new $y$ axis (vertical) and should have a positive $y$ component (so that the picture is not upside down) so we choose (arbitrarily): $q_y = 1$. Substituting our constraints in the Cartesian components we get:

$\mathbf{p} = [p_x, 0, p_z]$ and $\mathbf{q} = [q_x, 1, q_z]$

Now we know that, in a left hand axis system:

$\boldsymbol{k} = \boldsymbol{i} \times \boldsymbol{j}$ so therefore we can write: $\mathbf{d} = \mathbf{p} \times \mathbf{q}$

since we have not as yet constrained the magnitude of $\mathbf{p}$. Evaluating the cross product we get three Cartesian equations:

$d_x = p_y q_z - p_z q_y \qquad d_y = p_z q_x - p_x q_z \qquad d_z = p_x q_y - p_y q_x$

Substituting the values $p_y = 0$ and $q_y = 1$ these equations simplify to

$d_x = -p_z \qquad d_y = p_z q_x - p_x q_z \qquad d_z = p_x$

Thus we have solved for vector $\mathbf{p} = [d_z, 0, -d_x]$. To solve for $\mathbf{q}$ we need to express the condition that $\mathbf{p}$ and $\mathbf{q}$ are orthogonal and so have zero dot product, whence:

$\mathbf{p} \cdot \mathbf{q} = d_z q_x + 0 - d_x q_z = 0$ thus $q_z = d_z q_x / d_x$

and from the cross product already evaluated:

$d_y = p_z q_x - p_x q_z = -d_x q_x - d_z q_z = -d_x q_x - d_z^2 q_x / d_x$

so $q_x = -d_y d_x / (d_x^2 + d_z^2)$

and $\mathbf{q} = (-d_y d_x / (d_x^2 + d_z^2), 1, d_y d_z / (d_x^2 + d_z^2))$

finally we have that: $\boldsymbol{u} = \mathbf{p}/|\mathbf{p}|$ and $\boldsymbol{v} = \mathbf{q}/|\mathbf{q}|$

Thus we can deduce the whole of the transformation matrix.

## Tutorial 1: Analysis of three dimensional space.

This tutorial is about the use of vector algebra in the analysis of three dimensional scenes used in computer graphics system. The following notation is used:

- Position vectors are denoted by boldface capital letters: **P**, **Q**, **V** etc. Position vectors are the same as cartesian coordinates, and represent position relative to the origin.

- Direction vectors are indicated by boldface lowercase letters **d**, **n** etc. Direction vectors are independent of any origin.

- Scalars are represented by italics: *a*, *b*, etc.

A plane is an object that is only defined in Cartesian space, however, each plane has a normal vector, whose size is non zero, and whose direction is at right angles to that plane. We can find a normal vector by taking the cross product of any two direction vectors which are parallel to the plane.

1. Given three points:

$$\mathbf{P}_1 = (10, 20, 5)$$
$$\mathbf{P}_2 = (15, 10, 10)$$
$$\mathbf{P}_3 = (25, 20, 10)$$

find two direction vectors which are parallel to the plane defined by $\mathbf{P}_1$, $\mathbf{P}_2$ and $\mathbf{P}_3$. Hence find a normal vector to the plane.

2. A plane is defined in vector terms by the equation:

$$\mathbf{n} \cdot (\mathbf{P} - \mathbf{P}_1) = 0$$

where $\mathbf{P} = (x, y, z)$ is the locus of a point on the plane, and $\mathbf{P}_1$ is any point known to be in the plane.

For the points given in part 1, expand the vector plane equation to find the Cartesian form of the plane equation, ( i.e. $ax + by + cz + d = 0$).

Verify that you get the same result using either $\mathbf{P}_1$ or $\mathbf{P}_2$.

3. Write a procedure in any programming language you like which takes as input three points and returns the coefficients of the Cartesian plane equation (*a*,*b*,*c* and *d*).

4. Starting from any point on a face of a polyhedron, an inner surface normal is a normal vector to the plane of the face whose direction points into the polyhedron.

A tetrahedron is defined by the three points of part 1, and a fourth point $\mathbf{P}_4 = (30, 20, 10)$. Determine whether the normal vector that you calculated in part 1 is an inner surface normal, and if not find the inner surface normal.

5. Two lines intersect at a point $\mathbf{P}_1$, and are in the directions defined by $\mathbf{d}_1$ and $\mathbf{d}_2$. Provided that $\mathbf{d}_1$ and $\mathbf{d}_2$ represent different directions, the two lines define a plane.

Any point on the plane can be reached by travelling from $\mathbf{P}_1$ in direction $\mathbf{d}_1$ by some distance $\mu$ and then in direction $\mathbf{d}_2$ by a distance $\nu$.

Using this fact construct the parametric equation of any point on the plane of part 1 in terms of $\mu, \nu$, $\mathbf{P}_1$, $\mathbf{P}_2$ and $\mathbf{P}_3$. By taking the dot product with a normal vector to the plane $\mathbf{n}$, show that the parametric plane equation is equivalent to the vector plane equation of part 2.

# Vector Algebra Revision Notes

A vector is a one dimensional array of numbers: [1,55,79.3, -25] is a vector. For analysis of three dimensional geometry we will consider only vectors of dimension two or three.

A vector $\mathbf{p} = [p_x, p_y, p_z]$ has magnitude computed by $|\mathbf{p}| = \sqrt{(}p_x^2 + p_y^2 + p_z^2)$ and a direction which is defined by angles with the Cartesian axes: $\theta_x, \theta_y, \theta_z$ where $\cos\theta_x = p_x/|\mathbf{p}|, \cos\theta_y = p_y/|\mathbf{p}|$ and $\cos\theta_z = p_z/|\mathbf{p}|$. A vector does not have any particular position on the Cartesian axes.

A unit vector is one where $|\mathbf{p}| = 1$. By convention, the unit vectors in the directions of the Cartesian axes are labelled $\boldsymbol{i} = [1, 0, 0], \boldsymbol{j} = [0, 1, 0]$ and $\boldsymbol{k} = [0, 0, 1]$. Unit vectors are often used for specifying directions. Lower case bold italic letters are used for unit vectors in these notes.

A position vector is simply a coordinate in Cartesian space. It is a particular instance of a vector which starts from the origin. Upper case bold letters are used for position vectors in these notes.

A direction vector is the same as a vector. The name is used particularly when a vector defines a direction and its magnitude is not relevant. Lower case bold letters are used for direction vectors in these notes.

**Vector Addition:** There is only one way to add vectors, and that is to add the individual ordinates, so: $\mathbf{p} + \mathbf{q} = [p_x, p_y, p_z] + [q_x, q_y, q_z] = [p_x + q_x, p_y + q_y, p_z + q_z]$.

**Scalar multiplication:** Multiplication of a vector by a scalar means that each element of the vector is multiplied by the scalar, ie $\mu\mathbf{d} = \mu[d_x, d_y, d_z] = [\mu d_x, \mu d_y, \mu d_z]$. Multiplication by $-1$ reverses the direction of a vector. Scalar multiplication and vector addition can be combined to define a line in Cartesian space. The equation: $\mathbf{P} = \mathbf{B} + \mu\mathbf{d}$ specifies a line through point (position vector) $\mathbf{B}$ in direction $\mathbf{d}$. Choosing any value of $\mu$ will identify one point on the line.

**The scalar (or dot) product:** There are two ways of multiplying two vectors together. The scalar product returns a scalar value defined as $\mathbf{p} \cdot \mathbf{q} = p_x q_x + p_y q_y + p_z q_z$. Note that we can write the magnitude of a vector as $|\mathbf{p}| = \sqrt{(}\mathbf{p} \cdot \mathbf{p})$. The dot product can be written equivalently as $\mathbf{p} \cdot \mathbf{q} = |\mathbf{p}||\mathbf{q}| \cos(\theta)$ where $\theta$ is the angle between the directions of $\mathbf{p}$ and $\mathbf{q}$. The dot product has some useful properties:

1. If two vectors are at right angles, the dot product is zero.
2. If the angle between two vectors is acute the dot product is positive.
3. If one of the vectors is a unit vector the dot product is a projection of the other vector in that direction.

**The vector (or cross) product:** The second way of multiplying two vectors results in a vector defined by: $\mathbf{p} \times \mathbf{q} = [(p_y q_z - p_z q_y), (p_z q_x - p_x q_z), (p_x q_y - p_y q_x)]$. This formula may be easily remembered by considering the evaluation of the determinant:

$$\mathbf{p} \times \mathbf{q} = \begin{vmatrix} \boldsymbol{i} & \boldsymbol{j} & \boldsymbol{k} \\ p_x & p_y & p_z \\ q_x & q_y & q_z \end{vmatrix}$$

The vector product is a vector whose magnitude is given by $\mathbf{p} \times \mathbf{q} = |\mathbf{p}||\mathbf{q}| \sin(\theta)$ and whose direction is at right angles to both $\mathbf{p}$ and $\mathbf{q}$. Note that $\mathbf{p} \times \mathbf{p} = [0, 0, 0]$ and has no direction. The same is true for any two vectors with the same direction. To be precise about the direction we need to know its sign. The rule for determining this is best remembered by thinking of the two vectors as the hands of a clock. Measuring the angle clockwise from $\mathbf{p}$ to $\mathbf{q}$ the direction is *into* the clock for positive $|\mathbf{p}||\mathbf{q}| \sin(\theta)$. Note that $\mathbf{p} \times \mathbf{q} = -\mathbf{q} \times \mathbf{p}$ since, if the angle between $\mathbf{p}$ and $\mathbf{q}$ is $\theta$, then the angle between $\mathbf{q}$ and $\mathbf{p}$ will be $\sin(360 - \theta)$, and $\sin(360 - \theta) = \sin(-\theta) = -\sin(\theta)$. The useful property of the cross product is that it defines a direction at right angles to two vectors. Thus we can use it to find the normal vector to a plane simply by taking the cross product of two vectors on the plane. For most graphics applications we need to test which direction the normal vector goes.