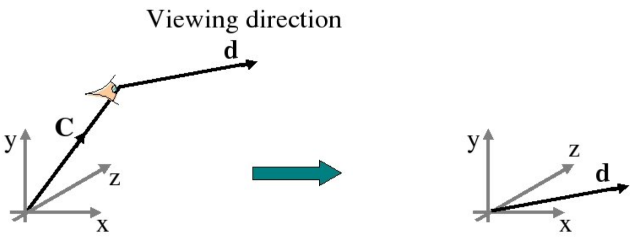
Lecture 2: Scene Transformation and Animation

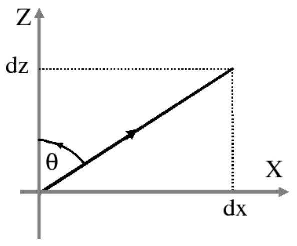
Flying Sequences

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 **C** = [Cx,Cy,Cz], looking along the direction **d** = [dx,dy,dz]. The first step is to move the origin to **C** for which we use the transformation matrix ***A***.



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

Following this, we wish to rotate about the y-axis so that **d** lies in the plane x=0. Using the fact that **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 ***B***.



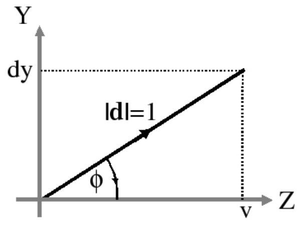
$$v = \sqrt{d_x^2 + d_z^2}$$

$$Cos\theta = d_z/v$$

$$Sin\theta = d_x/v$$

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

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 ***C***.



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

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

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

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

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

T = ***A*** * ***B*** * ***C***

and so for all the points

P = **P** * ***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,-1] 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.

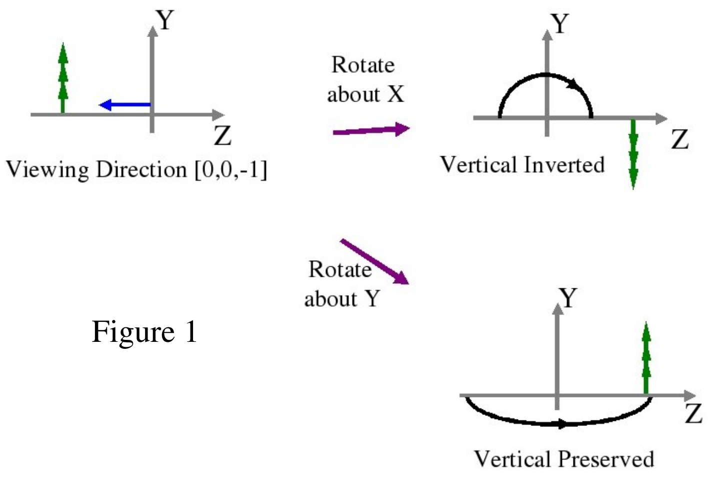


Figure 1

Rotation about a general line
A very similar problem 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: **L** + μ **d** where **L** = $[L_x,L_y,L_z]$ is the position vector of a point on the line and **d** is a unit direction vector along the line. In essence we follow exactly the process. We use a translation to move the origin so that it is on the line (matrix ***A*** but transforming the origin to L rather than C) followed by two rotations to make the z axis coincident with the direction vector **d** (matrices ***B*** and ***C***). Now we can perform a rotation of the object about the z-axis using the standard rotation matrix ***Rz*** defined previously. Now 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 A B and C. As before we multiply all the individual transformation matrices together to make one matrix which is then applied to all the points.

$\boldsymbol{T} = \boldsymbol{A} * \boldsymbol{B} * \boldsymbol{C} * \boldsymbol{R} * \boldsymbol{C}^{-1} * \boldsymbol{B}^{-1} * \boldsymbol{A}^{-1}$ and so for all the points $\mathbf{P} := \mathbf{P} * \boldsymbol{T}$

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 express projection by the multiplication of a projection matrix. Placing the centre of projection at the origin and using z=f as the projection plane gives us matrix $\boldsymbol{M}_p$ for perspective projection. Matrix $\boldsymbol{M}_o$ is for orthographic projection:

$$M_p = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1/f \\ 0 & 0 & 0 & 0 \end{bmatrix} \qquad M_o = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

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

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

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:

$\mathbf{P}_H = [x*f/z, y*f/z, f, 1]$

which is the correct answer. It is interesting to note that the projection matrix is obviously a singular matrix (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. Projections 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 capital letters. Hence, we can say that a normalised homogenous coordinate is the same as a position vector.

By contrast, if we consider the case where the last ordinate is zero - [x,y,z,0] - we find that we cannot normalise this coordinate 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 contains information in the sizes of x y and z, and hence we can consider it to be a direction vector. It has magnitude and direction, but not position. Thus homogenous coordinates fall into two classes, those with the final ordinate non-zero, which can be normalised into position vectors, and those with zero in the final ordinate which are direction vectors, and which also have direction magnitude.

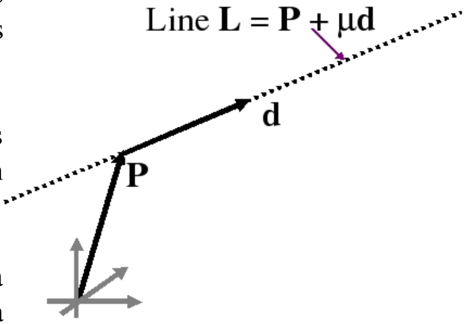
Consider now how vector addition works with these definitions. If we add two direction vectors, we add the ordinates as before and we obtain a direction vector. ie:

[xi,yi,zi,0] + [xj,yj,zj,0] = [xi+xj, yi+yj, zi+zj,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:

[Xi,Yi,Zi,1] + [xj,yj,zj,0] = [Xi+xj,Yi+yj,Zi+zj,1]

This is a nice result, because it ties in with our definition of a straight line in Cartesian space being defined by a one point and a direction.



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 below, with the rows viewed as three direction vectors and a position vector.

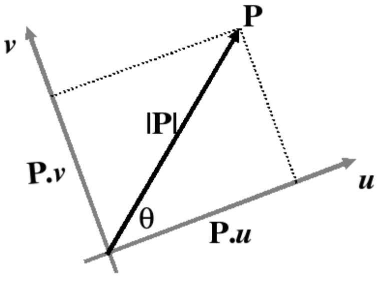
$$\begin{array}{cl} \left[\begin{array}{cccc} q_x & q_y & q_z & 0 \\ r_x & r_y & r_z & 0 \\ s_x & s_y & s_z & 0 \\ C_x & C_y & C_z & 1 \end{array}\right] & \begin{array}{l} \text{Direction vector} \\ \text{Direction vector} \\ \text{Direction vector} \\ \text{Position vector} \end{array} \end{array}$$

We ask what the individual rows mean, and to see this we consider the effect of the transformation in simple cases. For example take the unit vectors along the Cartesian axes eg:

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

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 ***j***= [0,1,0,0] will be transformed to direction [rx,ry,rz,0] and ***k*** = [0,0,1,0] will be transformed to [sx,sy,sz,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]. 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 **q r** and **s** will remain orthogonal so that **q•r** = **r•s** = **q•s** = 0. Unfortunately however, this analysis does not help us to determine the transformation matrix. In general it would be more natural to assume that we know the vectors ***u***,***v***, and ***w*** which we would like to transform into the Cartesian axes ***i***,***j***,***k***. This is the case when for example we are transforming a scene before viewing it in the normal position for computing a projection, that is with the viewpoint at the origin and the viewing direction along the z axis. In this case we need to use the notion of the dot product as a projection onto a line.

This is most readily seen in two dimensions as shown opposite. By dropping perpendiculars from the point **P** to the line defined by vector ***u*** and vector ***v*** we see that the distances from the origin are respectively **P•*u*** and **P•*v*. P•*u*** = |**P**||***u***|Cosθ = |**P**|Cosθ since u is a unit vector. Now, suppose that we wish to rotate the scene so that the new x and y axes were the u and v vectors, then the x ordinate would be defined by **P•*u*** and the y by **P•*v***.



The generalisation to three dimensions, in which a point P is transformed into the {***u,v,w***} axis system, translated by vector **C**, is given by:

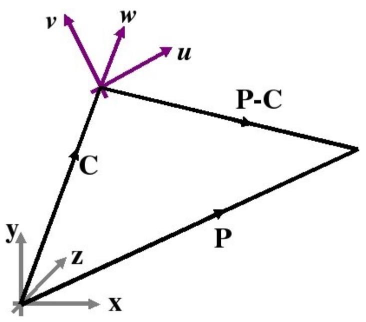
P'x = (**P**-**C**)•***u***

P'y = (**P**-**C**)•***v***

P'z = (**P**-**C**)•w

Expressing this as a transformation matrix we get:

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



So now by maintaining values of C, u, v and 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 solve the problem by flying problem (to view a scene from point **C** in direction **d**) by finding the new axis system ***u***,***v***,***w***, and then deducing the transformation matrix. The direction vector for viewing is **d** = [dx,dy,dz] and this must lie along the ***w*** direction, so ***w*** = **d**/|**d**|. We first find any two vectors, **p** and **q**,which have the same directions as ***u***,***v***, but not necessarily have unit length. We can constrain the system to preserve horizontals and verticals in two ways:

1. **p** will be in the direction of the new x-axis, so to preserve the horizontal, we therefore choose: py = 0.
2. **q** will be the new y axis and should have a positive y component (so that the picture is not upside down) so we choose (arbitrarily): qy = 1.

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

k = ***i*** x ***j*** so therefore we can write: **d** = **p** x **q**

since we have not constrained the magnitude of **p**. Substituting our constraints in the Cartesian components we get:

p = [px,0,pz] and **q** = [qx,1,qz]

and evaluating the cross product we get three Cartesian equations:

dx = -pz dy = pz qx - px qz dz = px

Thus we have solved for vector **p** = [dz,0,-dx].

To solve for **q** we need to express the condition that **p** and **q** are orthogonal and so have zero dot product, whence: dz qx + 0 - dx qz = 0 or qz = dz qx / dx

and from the cross product already evaluated

$dy = pz\,qx - px\,qz = -dx\,qx - dz\,qz = -dx\,qx - dz^2\,qx/dx$

so $qx = -dy\,dx/(dx^2 + dz^2)$

and $\mathbf{q} = (-dydx/(dx^2 + dz^2), 1, dydz/(dx^2 + dz^2))$

finally we have that: ***u*** = **p**/|**p**| ***v*** = **q**/|**q**|

Thus we can deduce the whole of the transformation matrix.