**Department of Computing**

## 3rd Year/MSc Option in Computer Graphics

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

Timetable Spring Term 2009:
Lectures Monday 16.00-17.00 room 145 and Wednesday 10.00-11.00 room 311
Tutorials Wednesday 11.00-12.00 room 311

Course webpage: http://www.doc.ic.ac.uk/~dfg/graphics/graphics.html

This course is intended to cover the fundamental aspects of computer graphics that would be utilised in the most prominent of its applications. The course is self contained but since it is offered to all third year and MSc students, there will be some duplication of the material covered in the first year computer course. This will be presented at a level more appropriate to third year teaching.

### Lectures

1 Device Independent Graphics, Projection and Transformation: 3D Planar objects, projection to 2D, Homogenous coordinates, scene transformation (dfg)
2 Transformations for Scene Animation: Flying Sequences, object transformations (dfg)
3 Analysis of 3D scenes: Clipping and containment in 3D convex objects, splitting concave objects. (dr)
4 The graphics pipeline (dr)
5 Texture mapping and anti-aliasing. (dr)
6 Shading planar polygons: Gouraud Shading, Phong Shading. (dr)
7 Using Colours: Tri-stimulus model, RGB model, YCM model, Perceptual colour spaces. (dr)
8 Introduction to Spline curves, cubic spline patches and Bezier Curves (dfg)
9 Introduction to Surface Construction, Bezier Surfaces, the Coon's patch (dfg)
10 Ray Tracing 1: Ray/object intersection calculations Secondary rays, shadows, reflection and refraction. (dr)
11 Ray Tracing 2: Computational efficiency, object space coherence, ray space coherence (dr)
12 Radiosity 1: Modeling ambient light, form factors (dfg)
13 Radiosity 2: Specular effects, shooting patches, computational efficiency (dfg)
14 Geometric Warping (dr)
15 Morphing Objects (dr)
16 Modelling Fire (dfg)
17 Non-photorealistic Rendering (dfg)
18 Kinematics and Animation

### Tutorials

1 3D geometry Transformations and Projections
2 Coordinate space transformations
3 Texture Mapping and Differential Algorithms
4 Shading
5 Colour
6 Curves and Surfaces
7 Ray Tracing
8 Radiosity
9 Warping and Morphing

**Coursework**

The coursework will be a series of practical programming exercises using OpenGL, and may be done in groups of up to three. It will involve rendering a 3D data set with different lighting and texture from different viewpoints, and some warping and morphing.

**Books**

There are many good books on graphics. Our advice is don't bother to buy one unless you really feel you want one. We will hand out comprehensive notes on each subject covered. Some possibilities for reference are:

Interactive Computer Graphics – A top down approach with OpenGL, E. Angel, 2nd edition, Addison Wesley, 2000, well suited for this course, many examples in OpenGL

Computer Graphics: Principles and practice. J. D. Foley et al, 2nd edition, Addison Wesley, 1996
very comprehensive – "The Bible of Graphics" - but rather verbose.

3D Computer Graphics, Alan Watt Addison Wesley 2000 Very clearly written, but doesn't cover all the course material.

**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)

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.

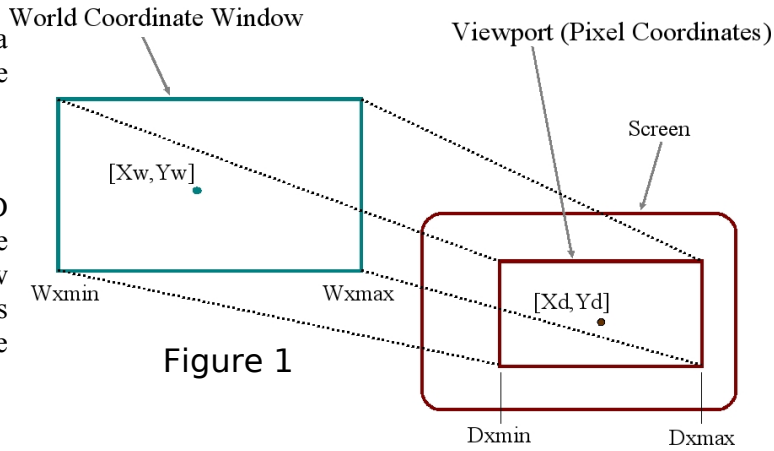


Figure 1

**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();
}
```

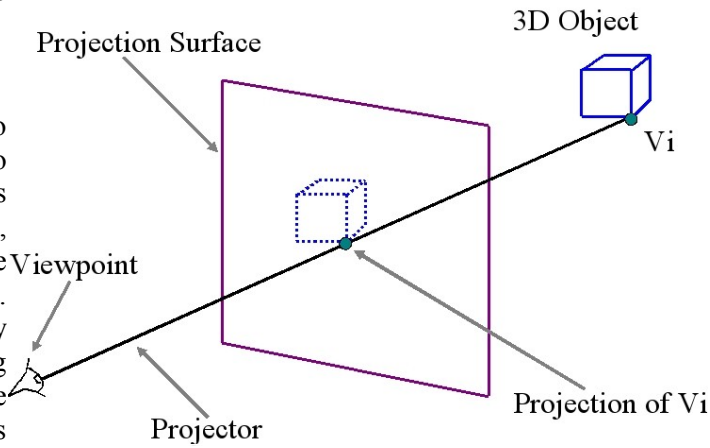
The procedure *ProcessMenuRequest* will be used to launch all the normal actions, such as save and open and quit, together with all the application specific requests. The procedures *GetMouseCoordinates* and *PerformMouseProcess* will be used by the application writer to create whatever effect is wanted, 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:

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

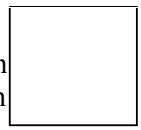
In *orthographic* projection the projectors are perpendicular to the projection plane, which we define as z=0. In this case the projectors are in the direction of the z axis and:

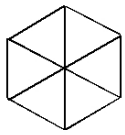
$$\mathbf{d} = [0,0,-1]$$

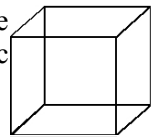
and so $P_x = V_x$

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.


Looking at a Face


Looking at a vertex


General View

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.

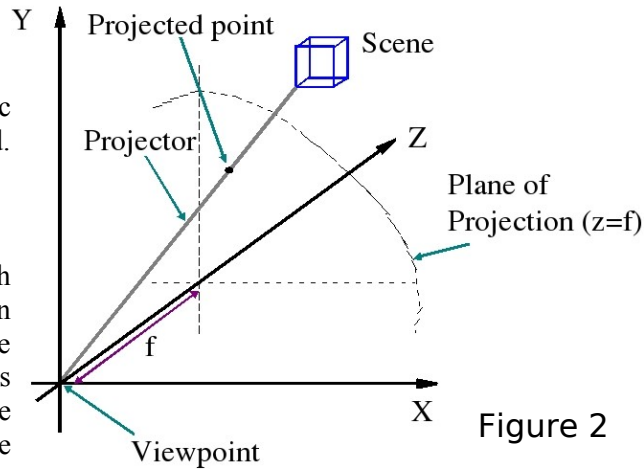


Figure 2

**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:

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

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

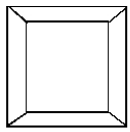
$$f = \mu V_z$$

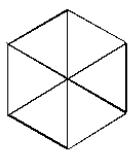
If we write $\mu_p = f / V_z$ for the intersection point on the plane of projection then:

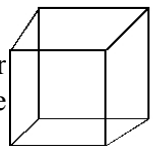
thus

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

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


Looking at a Face


Looking at a vertex

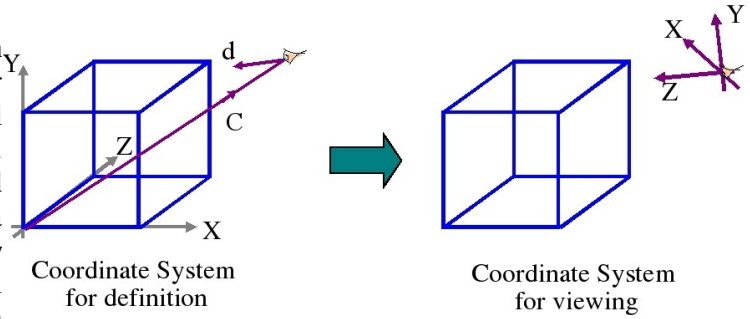

General View

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 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].

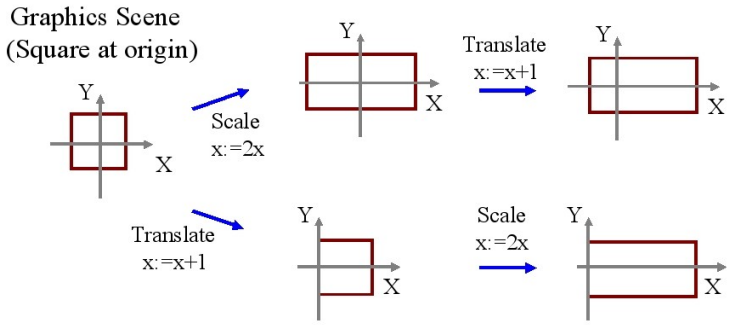
In most cases *s* will be 1. The point of introducing homogenous coordinates is to allow us to translate the points of a scene by using matrix multiplication.

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

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.

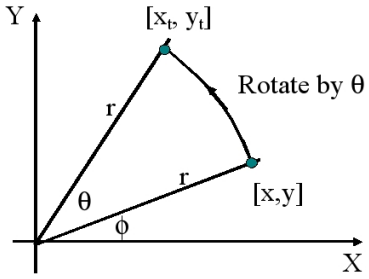



Rotation has to be treated differently since we need to specify an axis. The matrices for rotation about the three Cartesian axes are:

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

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

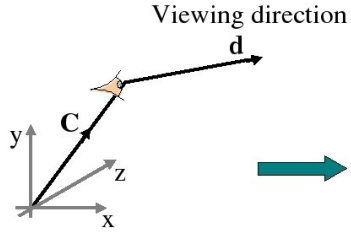
Hence to invert the matrix we simply change the sign of the Sin terms.

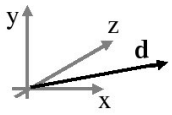
$$
\begin{bmatrix} Cos\theta & Sin\theta & 0 & 0 \\ -Sin\theta & Cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \text{has inversion} \begin{bmatrix} Cos\theta & -Sin\theta & 0 & 0 \\ Sin\theta & Cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}
$$

## Lecture 2: Scene Transformation and Animation

### Flying Sequences

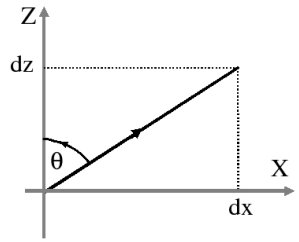
We will now consider a most important subject, namely, scene transformation. In any viewer centered application, such as a flight simulator or a computer game, we need to view the scene from a moving position. As the viewpoint changes we transform all the coordinates of the scene such that the viewpoint 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***.



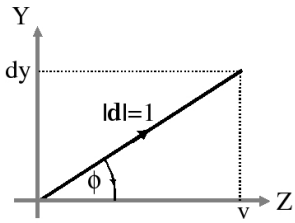

$$v = \sqrt{d_x^2 + d_z^2}$$

$$Cos\theta = d_z/v$$

$$Sin\theta = d_x/v$$

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

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,-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.

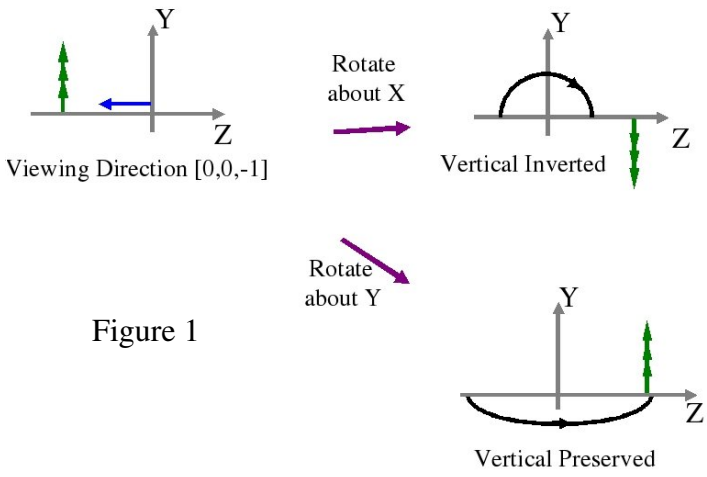



Figure 1

**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: $\mathbf{L} + \mu\,\mathbf{d}$ where $\mathbf{L} = [L_x, L_y, L_z]$ 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 $\boldsymbol{M}_\mathbf{p}$ for perspective projection. Matrix $\boldsymbol{M}_\mathbf{o}$ 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 $\boldsymbol{M}_\mathbf{p}$ 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

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

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.

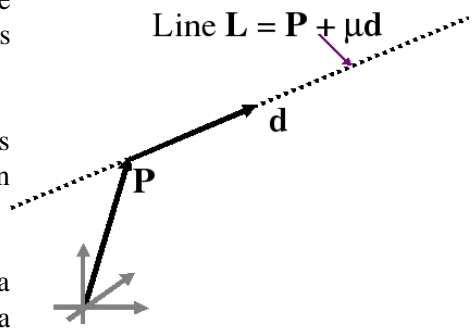
Consider now how vector addition works with these definitions. If we add two direction vectors, we add the ordinates as before and we obtain a direction vector. ie:

[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.

$$\begin{bmatrix} q_x & q_y & q_z & 0 \\ r_x & r_y & r_z & 0 \\ s_x & s_y & s_z & 0 \\ C_x & C_y & C_z & 1 \end{bmatrix}$$

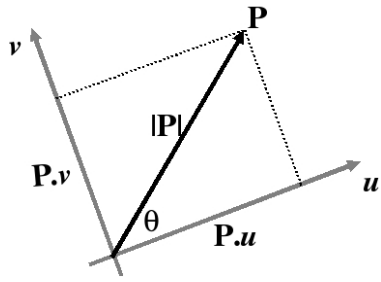
Direction vector
Direction vector
Direction vector
Position vector

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:

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

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:

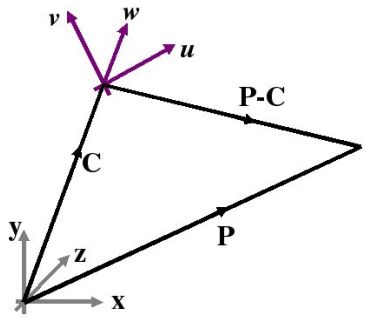
P'x = (**P**-**C**)•***u***

P'y = (**P**-**C**)•***v***

P'z = (**P**-**C**)•w

Expressing this as a transformation matrix we get:

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




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 **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.

## 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.