Chapter 2

Surface Models

2.1 Surface Modelling

To draw a picture, such as a frame from an animation, or a architectural plan of a building, we must first start with some *model* within the computer. This model will contain all the information we require to make the picture, such as the shape of objects (i.e. their geometry); what colours they are; what textures, what reflectivity, and so on.

We need to consider how to store this kind of data in such a way that it is convenient for interactive manipulation. Later, it will also serve as the input to the process which draws the picture: this is process is called *rendering*. Modelling and rendering are separate, though the latter is often chosen to suit the former.

We shall look in detail at two popular ways of modelling, surface modelling and solid modelling. The kind of model strongly affects the rendering process in that certain kinds of models tend to be rendered in certain ways, but this connection is not absolute. Later, we will look at appropriate rendering techniques.

2.1.1 Surface Models

An object is stored in terms of its *surface*. That is, a sphere is represented by its surface; a chair is represented by a collection of planes, cylinders etc. This is the longest established modelling process. Surfaces are what we see, so they are a natural choice for graphical applications.

Usually we use a *mesh* representation: a mesh consists of a (large) number of flat polygons, often triangles, that map out the surface in question. Note that each polygon is flat, so this will be an approximation to a curved surface, but it is easy to make an approximation as good as we require simply by making the polygons smaller. This form dominates the PC games card market, making it a cost-effective form to render quickly. "Triangles per second" is often a performance indicator for such cards.

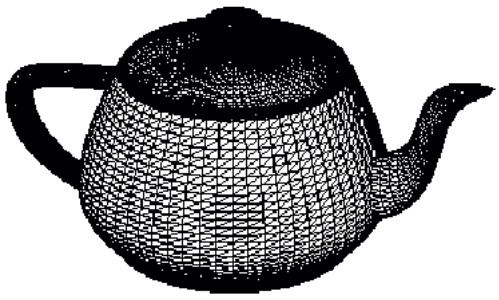


Figure 2.1: Newell's Teapot (the Utah Teapot)

2.2 Data Structures for Meshes

To write sophisticated application such as a modeller, we have to spend a lot of time making sure we have appropriate data structures. Particular problems occur when we are at the interactive modelling stage: making sure we can find a 3D vertex (or edge or triangle or complete object) quickly from a nearby 2D mouse click is a basic requirement for interaction. The data structure needs to support this: it isn't enough to just do a search of the data because models can be hugely complex and this would be very slow. A second key requirement is updating the data structure. If we wish to move a vertex or delete part of an object we again want to do this quickly. However we also need to ensure consistency. For example, if we delete the top of a table we still expect to leave the legs. We also expect the legs to be complete, not to be missing vertices which disappeared with the table top.

A mesh is composed of many facets, each a polygon. In this Unit we shall restrict ourselves to triangular facets. The advantages of triangles are

1. any three (non collinear) points define a plane, even in the presence of rounding error: with four or more points the facets we describe using floating point numbers may not be quite flat, and edges may not line up properly

2. triangles are convex: non convex polygons introduce extra complication later in hidden surface removal

In professional modelling applications various other forms are used, including facets which are curved in various ways. However all of these are used to generate polygonal meshes to display the object: graphics cards can handle polygons really fast.

We would like a data structure that is easily modified, as typically we want to manipulate dynamically a line drawing of the mesh, grabbing and moving edges and vertices interactively. We must allow for such alterations. So what are the problems? Let's try some simple solutions. These are not very practical at first but are designed to show you why it is not straightforward.

2.2.1 Simple Vertex List

Define all vertices v_i by a list of their 3D coordinates. Define all edges by a list of (v_i, v_j) pairs.

This is enough to draw our image as a line drawing because we can run along the edge list drawing a line between the vertex pairs.

It is useless for interaction: we cannot relate edges to polygons, or vertices to edges. Thus, if a user picks a vertex and wants to move it, we would have to search the edge list to find the

edges the vertex belongs to. In a model containing thousands or even millions of edges this is woefully slow.

Being unable to determine polygons also means that we are unable to produce a solid shaded picture, or determine which parts of the object are obscured by other parts.

2.2.2 Polygons as a Vertex List

Identify each polygon in the model as a list of vertices.

$$P = [(x_1, y_1, z_1), (x_2, y_2, z_2), \ldots (x_n, y_n, z_n)].$$

Edges are implied by the order of the vertices in the list. So now we can determine polygons easily. However there are problems.

1. Each edge is doubly represented, being in the lists for two polygons. If we want to move an edge, we have to make sure we update both copies consistently.
2. Vertices are similarly duplicated.
3. We can only find edges and vertices by searching all the data. So "find the vertex closest to these coordinates" will be very slow.

2.2.3 Polygons are pointers to a Vertex List

List all vertices once. Define polygons as pointers (or indices) into this list. e.g., polygon P_2 might be vertices $[2, 5, 7]$.

Now vertices can be moved easily, but we still have difficulties with edges.

2.2.4 Polygons, Vertex and Edge Lists

We put all the above together:

Polygon list	Edge list		Vertex list
edges	polygons	vertices	coordinates
p1: e1 e2 e3	e1: p1	v1 v4	v1: x_1 y_1 z_1
p2: e3 e4 e5	e2: p1	1 2	v2: x_2 y_2 z_2
	e3: p1 p2	v2 v4	v3: x_3 y_3 z_3
	e4: p2	v2 v3	v4: x_4 y_4 z_4
	e5: p2	v3 v4	

Polygons are pointers into the edge list. Pointers are held in (say) clockwise order as seen from the front, so we can easily determine the front and back of a facet.

Edges are pointers to the containing polygons, and to their vertices.

Vertices are kept just once.

This allows us to add, delete and change vertices, edges and polygons quite flexibly.

In a "real" modelling program, other issues may have to be addressed. For example, in engineering Computer-Aided Design (CAD) we will want to calculate the volume of objects, which needs some representation of closed volumes, allowing for arbitrary holes etc. We will also want to calculate mass, which means each component will be associated with a material of known properties.

For many straightforward graphical applications, we can side-step the detailed data structures by using the Open Source OpenGL API. This is essentially a readily-available library with many built-in routines to help you define and view 3D objects, set up colours, textures, lights etc. If you want to do anything more sophisticated, you can often still use OpenGL for the basic graphics.

2.2.5 Wavefront OBJ format

The above practical considerations have led to a de facto standard for mesh objects. Wavefront is a commercial package but its format is published on the web and widely used. It incorporates sophisticated features such as free-form curves and surfaces but at heart it allows you to specify vertices (v) and faces (f) in a simple, compact way, within an ASCII file. For example, a single triangle is:
v 0.0 0.0 0.0
v 0.0 1.0 0.0
v 1.0 0.0 0.0
f 1 2 3

The index of each vertex is implicit in the order in which they appear. In this example, vertex 1 is at the origin, vertex 2 is at y=1, vertex 3 is at x=1. Texture and material can also be spefified. You don't have to specify all the vertices before you specify a face. If the face's vertices are negative, then it means "number the vertex from the previous block of vertices", rather than number in absolute order, which permits you to specify the vertices for a face, then the face itself.

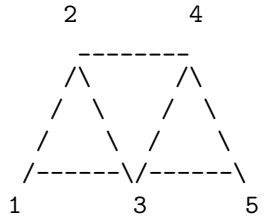
2.3 Data Structures for Games

Computer games have driven real-time graphics very strongly in recent years. The key requirement is speed. Most of the scenery in a computer game is fixed, in the sense that users - game players - do not interactively modify it the way an engineer might interactively modify a CAD model. There is therefore no need for a data structure which permits interactive modification (though it is highly likely that the *designer* of the game used such a structure).

As a result, games rely on simple data structures which, as far as possible, can be pumped onto the graphics card at high speed.

We clearly have to send all the vertex data, so any method which sends each vertex once is efficient. What about the edge and triangle information? We can reduce the need to send this information if the edges are implicit in the vertex order.

For example, suppose we send vertices 1,2,3,4 and 5. By convention, we assume that this defines three triangles, arranged as follows.



The point about this interpretation is that we simply take the vertices in the order they arrive and, at all times, the most recent three define the next triangle. So, when we see 1,2,3,4,5, we

interpret it to mean the triangles

$$(1,2,3),\ (2,3,4),\ (3,4,5)$$

In other words, each new vertex makes a triangle with the previous two received. This makes it very easy to build the full data structure from the triangle strip: the program only needs to remember the last two vertices.

This method extends to make an indefinitely long *tri-strip*, at a cost of sending only the vertices in the correct sequence. We can thus send n triangles by sending only $n + 2$ vertices. The sequence can be established at the game design stage. The correct vertices are stored on the game disk in the correct sequence. When you enter a new level, the triangles are loaded onto the graphics card.

2.4 Transformations

Suppose we have a collection of primitive objects (each in their own object coordinates), such as a cylinder and a torus. How can we combine these to model a bicycle wheel? We can think of a wheel as a large thin torus, with many long thin cylinders within it for the spokes and maybe a cylinder for the hub.

First we *scale* the each object to the correct size, *rotate* to the correct (3D) orientation, then *translate* to the final destination. This allows us to use *instancing*. That is, we only need to describe a cylinder once but we then make several instances of it using different transformations, thus producing (for example) all the spokes.

The order in which we do these operations matters. Transformations are relative to a particular origin and a particular coordinate system. Applying a transformation changes the object, so the effect of the next transformation in a sequence depends on what went before. It is far simpler to use the above order because we can use a standard set of transformations.

For example, suppose we scale a car model along the x-axis (its length, say), then rotate it by 90 degrees around the vertical. This gives a different effect to rotating first because the later x-axis scale would then affect the width of the car.

2.4.1 Representing transformations

Scaling is a multiplicative operation. Translation is an additive operations. Rotation requires multiplication and addition. It would be nice to have a single transformation mechanism that combines all three in a uniform manner, so that we can compose any of the three in arbitrary combinations – while still ensuring they are independent.

We can obviously represent a 3D coordinate as a vector (a 3-tuple if you prefer to call it that) and vectors can be manipulated with matrices. Matrix multiplication has the properties we need: it involves both scalar multiplication and scalar addition, while still allowing each item to be independently changed. However, we are working in 3D, so 3×3 matrices are not going to solve all our problems: we need a 3×3 matrix just to rotate a vector: there are no elements in the matrix which will allow us to translate it as well.

The solution is to use *homogeneous coordinates*. Instead of using the simple triple (x, y, z) we use (wx, wy, wz, w), where w is any non-zero number. From the homogeneous form (x, y, z, w) we can retrieve the original as $(x/w, y/w, z/w)$ since $w \neq 0$. Often $w = 1$ can be chosen, so we have an easy, direct correspondence

$$(x, y, z) \leftrightarrow (x, y, z, 1).$$

The advantage of this is that we can use 4D matrices to represent 3D translation, scaling and rotation in a uniform way, as we will now see. (N.B. all are relative to the origin).

[Most graphics textbooks introduce homogeneous coordinates this way, as a convenient fix to combine translation and rotation. What they don't tell you is that there is a really fundamental reason why this is the correct thing to do. We will explain this in a later chapter, when we look at projective geometry.]

A quick reminder that matrices can be represented in row-order or column-order. We can premultiply the vector or post-multiply the vector. Both are valid: different texts use different conventions. Just don't mix the two in the same calculation!

2.4.2 Translation in 3D

Suppose we want to translate by (t_x, t_y, t_z). We can represent this in matrix form as

$$T = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ t_x & t_y & t_z & 1 \end{pmatrix}$$

so that a point (x, y, z) transforms as

$$(x\ y\ z\ 1)T = (x + t_x\ y + t_y\ z + t_z\ 1).$$

The inverse of T is simple

$$T^{-1} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -t_x & -t_y & -t_z & 1 \end{pmatrix}.$$

Note that we have not had to make use of the upper-left 3×3, so we can put our multiplicative rotate/scale in there, as we will now see.

2.4.3 Rotation in 3D

Rotation in 3D is a lot more complicated than rotation in 2D. Before we start we have to determine some conventions.

In 2D, a positive rotation of an angle θ is given by a matrix

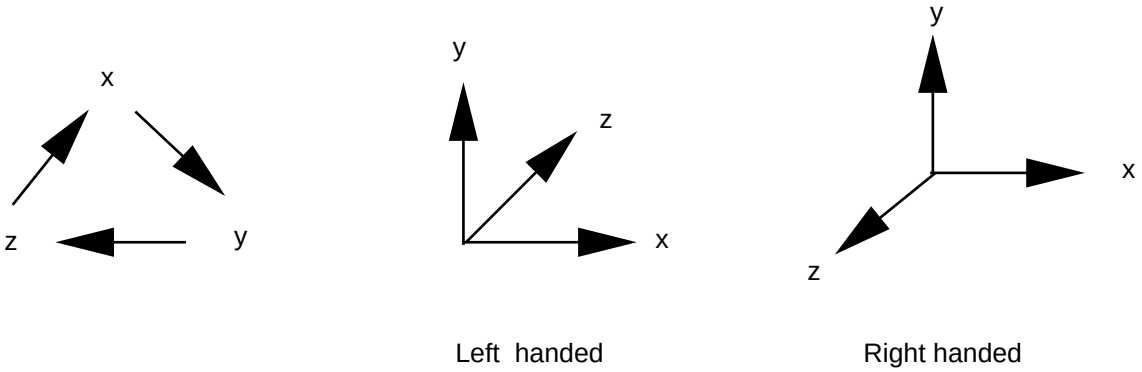
$$\begin{pmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{pmatrix}$$

where, by convention, positive is anticlockwise. Notice that rotation is about a *point* in 2D, about an *axis* in 3D but is within a plane in both.

Similarly, in 3D, we need to determine which way is positive, and we need to do it for all three axes.

rotation axis	direction of positive rotation
x	from y towards z
y	from z towards x
z	from x towards y

This is easy to remember as x, y, and z are cyclic.



This definition gives us "positive" regardless of handedness of the coordinates: thus matrices are independent of handedness.

However, "clockwise" is not independent of handedness (and so is a less helpful concept, except possibly when all your coordinate systems have the same handedness and do not change). Whether a rotation is clock- or anticlockwise is determined by looking along the axis, from a positive value towards the origin. The above positive rotations are anticlockwise rotations in a right-handed system but are clockwise in a left-handed one. Better to stick with positive and negative.

Here are the matrices for rotations about the three major axes.

$$R_x = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & \sin\theta & 0 \\ 0 & -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$R_y = \begin{pmatrix} \cos\theta & 0 & -\sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ \sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

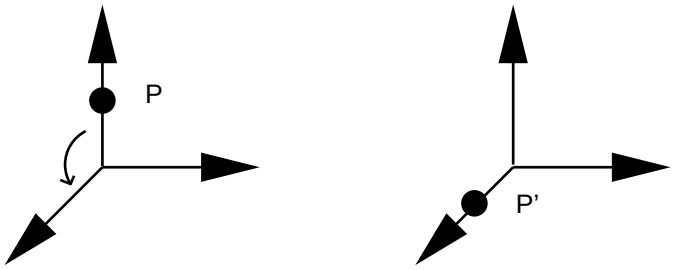
$$R_z = \begin{pmatrix} \cos\theta & \sin\theta & 0 & 0 \\ -\sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

The inverses of these matrices are their transposes – swap the rows and columns. This is vital information if you only have the matrix numerically because you cannot in general invert such matrices. If you have it algebraically, as above, then you can instead replace θ by $-\theta$ everywhere. The result is the same as the inverse of course.

Why does the transpose produce the inverse? If you think that the matrix is transforming from coordinate system A to coordinate system B, then the columns in the matrix say how each axis of A is transformed. If we now wish to construct a matrix which goes in the reverse direction (from coordinate system B to coordinate system A), then the roles of the two sets of axes are reversed; so we take the rows of the original matrix and make them the columns.

Notice the similarity between 2D rotation and the 3D z-rotation: this is because the familiar 2D rotation in the xy plane can be thought of as a rotation about the z axis.

Example. Let $P = (0, 1, 0)$ in a right-handed system. Apply a rotation about the x-axis of $+90°$.



The matrix is

$$M = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

Thus, in homogeneous form

$$PM = (0\ 1\ 0\ 1) \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} = (0\ 0\ 1\ 1),$$

as expected.

2.4.4 Scaling in 3D

This one is easy. Here is the homogeneous matrix for scaling s_x in direction x, etc.

$$S = \begin{pmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

This has inverse

$$S^{-1} = \begin{pmatrix} 1/s_x & 0 & 0 & 0 \\ 0 & 1/s_y & 0 & 0 \\ 0 & 0 & 1/s_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix},$$

as long as all of the ss are non-zero. A zero s corresponds to flattening an object into a lower dimension. The inverse is not generally the same as the transpose in that case. So, unless all scale factors are known to be 1, transposition will not invert a general transformation matrix.

2.4.5 Combining Transformations

We can compose translations, scalings and rotations as we please, by matrix multiplication. We will end up with a matrix of the form

$$\begin{pmatrix} r_{11} & r_{12} & r_{13} & 0 \\ r_{21} & r_{22} & r_{23} & 0 \\ r_{31} & r_{32} & r_{33} & 0 \\ t_x & t_y & t_z & 1 \end{pmatrix}.$$

The rs represent scaling and rotation, while the ts represent a translation. We often contrive that scaling is 1, so then the r values are pure rotation (and we can invert any such transformation by transposition of the matrix).

For modelling, we scale, then rotate, then translate. For *viewing* we use the same transformations, but in the opposite order, namely translate, then rotate, then scale. This is because we are moving the *viewer*, not the object: moving the viewer in one direction is as if we are moving the object/world in the opposite direction.

For example, suppose we have defined an object, and we wish to look at it from different directions, as it were by tumbling it in our hands. Suppose we have some axis that we wish to rotate about: recall that transformations are relative to the origin. So we have to translate some point P on the axis to the origin, rotate, then translate P back:

1. translate by $-P$
2. rotate about desired axis
3. translate by P

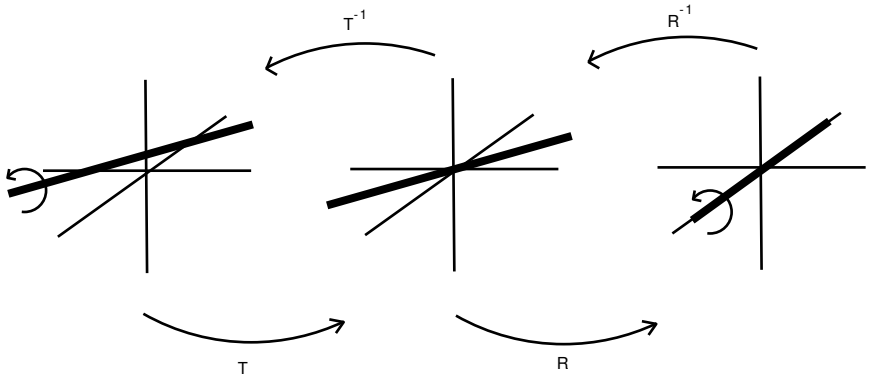
Thus far, so good. The above will work for any axis parallel to the three major axes, which is what we try to do wherever possible. (When we are designing a system, it is always simpler in the long run to try to stick with these major axis rotations). Once in a while however, we do not have the problem formulated that way: we have to rotate about an arbitrary axis. So how do we do that?

2.4.6 Rotation about Arbitrary Axes

This is not easy and we usually try to avoid doing it. However, it's general solution gives insight into how we can combine transformations.

The solution is to reduce the problem to one we have solved already, viz., transform the model until our desired axis of rotation lies along (say) the z axis, rotate about z, then transform back. (This additional computation takes time, which is one reason why we try to avoid it.)

1. translate so the arbitrary axis passes through the origin
2. rotate the model about one or two axes until the arbitrary axis lies along the z axis
3. rotate about z the desired amount
4. invert step 2
5. invert step 1



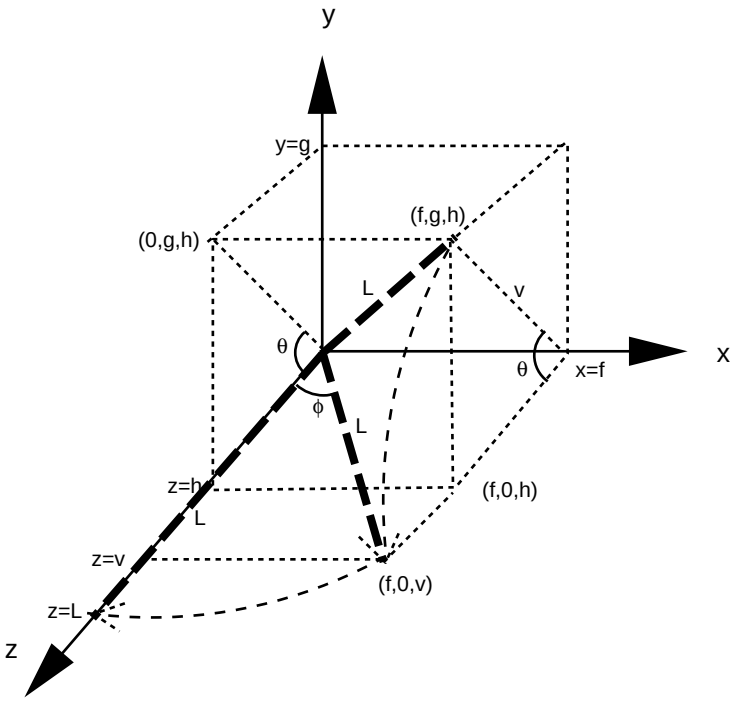
So we need to determine the matrices to do this. Suppose the axis we want to rotate about lies on (x_1, y_1, z_1) in the direction of (f, g, h). So the line has parametric equation

$$x = x_1 + ft,\ y = y_1 + gt,\ z = z_1 + ht.$$

Thus the translation we need is $(-x_1, -y_1, -z_1)$ with matrix

$$T = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -x_1 & -y_1 & -z_1 & 0 \end{pmatrix}.$$

The picture now looks like:



We now rotate about the x axis by an angle θ until the axis lies in the xz plane. Now we do not actually have to determine θ, as the rotation matrix only requires $\sin\theta$ and $\cos\theta$, and these are easy. The diagonal $v = \sqrt{g^2 + h^2}$, so

$$\sin\theta = g/v,\ \cos\theta = h/v.$$

This gives us the rotation matrix (rotating about x)

$$R_x = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & h/v & g/v & 0 \\ 0 & -g/v & h/v & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

This is particularly nice as we don't have to compute any expensive trigonometric functions.

So the axis now lies in the xz plane. Next we are going to rotate about the y axis to bring it in line with the z axis. The diagonal of the triangle is $L = \sqrt{f^2 + g^2 + h^2}$. Rotate about y by ϕ (negative)

$$\sin\phi = f/L,\ \cos\phi = v/L,$$

giving the matrix

$$R_y = \begin{pmatrix} v/L & 0 & f/L & 0 \\ 0 & 1 & 0 & 0 \\ -f/L & 0 & v/L & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

We can now rotate about z by the requested amount (say ψ)

$$R_z = \begin{pmatrix} \cos\psi & \sin\psi & 0 & 0 \\ -\sin\psi & \cos\psi & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

Now we restore everything back to its starting place: apply R_y^{-1}, then R_x^{-1}, and then T^{-1}.

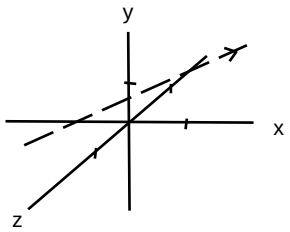
Combining all these together we get

$$R = TR_xR_yR_zR_y^{-1}R_x^{-1}T^{-1}.$$

Example

[This is NOT examined]

Rotate about the axis through $P = (1, 1, 1)$ in direction $(1, 1, -1)$.



The line has parametric form $(1 + t, 1 + t, 1 - t)$, and passes though $(0, 0, 2)$, $(1, 1, 1)$, $(2, 2, 0)$, etc. In homogeneous form this is $(1 + t, 1 + t, 1 - t, 1)$ First translate the point P to the origin:

$$T = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -1 & -1 & -1 & 1 \end{pmatrix}.$$

Check: the line transforms to $(1 + t, 1 + t, 1 - t, 1)T = (t, t, -t, 1)$, which passes through the (3-space) origin when $t = 0$. Next rotate about x. Now $v = \sqrt{g^2 + h^2}$, so we get $v = \sqrt{2}$. Further $(f, g, h) = (1, 1, -1)$; in particular we note that h is negative so, putting these values in to the x-rotation matrix gives us:

$$R_x = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & -1/\sqrt{2} & 1/\sqrt{2} & 0 \\ 0 & -1/\sqrt{2} & -1/\sqrt{2} & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

The line becomes $(t, t, -t, 1)R_x = (t, 0, \sqrt{2}t, 1)$, which line in the xz plane. Rotate about y (negative angle). We get $L = \sqrt{3}$, then

$$R_y = \begin{pmatrix} \sqrt{2/3} & 0 & 1/\sqrt{3} & 0 \\ 0 & 1 & 0 & 0 \\ -1/\sqrt{3} & 0 & \sqrt{2/3} & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

The line becomes $(t, 0, \sqrt{2}t, 1)R_y = (0, 0, \sqrt{3}t, 1)$, which runs along the z axis.

Finally, rotate about z by ψ, R_z is as above. Combining all of these and their inverses,

$$\begin{aligned} R &= TR_xR_yR_zR_y^{-1}R_x^{-1}T^{-1} \\ &= \tfrac{1}{3}\begin{pmatrix} 2\cos\psi & -\sqrt{3}\sin\psi - \cos\psi + 1 & -\sqrt{3}\sin\psi + \cos\psi - 1 & 0 \\ -\cos\psi + \sqrt{3}\sin\psi + 1 & 2\cos\psi + 1 & \sqrt{3}\sin\psi + \cos\psi - 1 & 0 \\ \sqrt{3}\sin\psi + \cos\psi - 1 & -\sqrt{3}\sin\psi + \cos\psi - 1 & 2\cos\psi + 1 & 0 \\ -2\cos\psi - 2\sqrt{3}\sin\psi + 2 & 2 + 2\sqrt{3}\sin\psi - 2\cos\psi & 4 - 4\cos\psi & 3 \end{pmatrix}. \end{aligned}$$

Now you can see why we try to avoid doing this!

2.5 Coordinate Systems

So far we have assumed that there is some sort of general coordinate system that we have used to fix our models. In practice it turns out to be too restrictive to have a single coordinate system: suppose we have a model containing a chair, and we want to draw two identical chairs in the picture. One way would be to incorporate each chair individually, but this is a waste of effort, duplicating all the details. Similarly we might want a big chair next to a small chair and so on.

What is useful is to have a collection of coordinate systems. Each stage of the modelling and viewing process can use a coordinate system that is convenient for that particular stage: for example, the modeller can build its world in a coordinate system of its choice, not one that is foisted upon it by the graphics package chosen to render the picture.

All systems will be Cartesian.

(a) Object Coordinates (3D). Each object is defined in is own coordinate space. E.g., a cube, a sphere, etc.

(b) World Coordinates (3D). To generate a picture we may want to use a graphics library, such as OpenGL. The library routines will work in their own preferred coordinates.

(c) Viewing Coordinates (3D). We transform and combine those objects to produce the complete model centred on the camera position and orientation. We can clip the the viewing volume at this stage.

(d) Normalised Device Coordinates (2D). The output from the package will probably be a bunch of floating point numbers in the range $[0.0, 1.0]$ for each (x, y), corresponding to some abstract output device.

(e) Device Coordinates (2D). These numbers will be converted to other numbers (often integers, e.g., pixel locations) to address a specific output device, such as a screen or a printer.

A modeller might choose a system where coordinates vary from[0, 100], while a graphics renderer might prefer a value range of [0.0, 1.0], or $[-1.0, 1.0]$, say. Each can use their preferred system, and it just takes a simple transformation (e.g. divide by 10) to move between the systems. Furthermore, a modeller can create their model even if they don't know which renderer will be used.

As part of (c) we usually define a *window* (in 3D world coordinate), which is a rectangle through which we look. This limits what we can see. We also have a *viewpoint* and a *view direction* to determine from where we see and in what direction we look.

Similarly in (d) we can define a *viewport* on the virtual screen (normalised device coordinates) to indicate where on the screen the picture will appear.

Notes.

- This is just one way of organising coordinate systems: other people might do things differently, or name the systems differently.
- (a) and (b) are part of the modelling process, while (c) to (e) are part of the viewing process. These are quite separate and are easily confused. Modelling is often application specific, and can take the larger part of the programming effort. The rest can be handled with standard graphics packages like OpenGL, GKS, and others.
- It can help to think of the window as the position of the viewer holding the "camera", and the viewport as where on the page we glue the resultant picture.

Between each coordinate system there will be a transformation. We will need the following transformations.

- Modelling transformation (from OC to WC)
- Viewing transformation (from WC to VC)
- Projection transformation (from VC to NDC)
- Viewport transformation (from NDC to DC)

A last word on cameras and coordinates. When *writing a renderer* rather than *using a modeller*, it is common to work in camera coordinates, typically with the origin at the centre of projection and the positive z-axis going away from the camera, into the scene. The transformation to these coordinates can be deduced from the position, direction and orientation of the camera in the World coordinates. The projection produced is a function of the notional lens: see the section on Perspective for how this comes about and what parameters are needed. The NDC can be thought of as the film or image plane of the camera. If we are generating a pixel image (rather than a vector drawing) we will omit the NDC replace it with the array of pixels required. In effect our pixel image is "the device" and has its own integer DC. Our only control over this will be to state the required resolution. In effect, this means we have used the window and view direction to determine the direction of the camera axes, while the viewpoint becomes the origin and a user-specified "up-vector" determines the camera orientation.