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 graphics applications 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 easy to implement it if we use a 'world coordinate system'.

A world coordinate system 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 that depend on the application. If the application is for visualising a house, then the units could be meters; If it is to draw accurate molecular models the units could be μm. The application program uses drawing primitives that work in these units, and converts their numeric values to pixels just before the image is rendered on the screen. This makes it easy to transport the application 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(Vxmin, Vymin, Vxmax, Vymax)
```

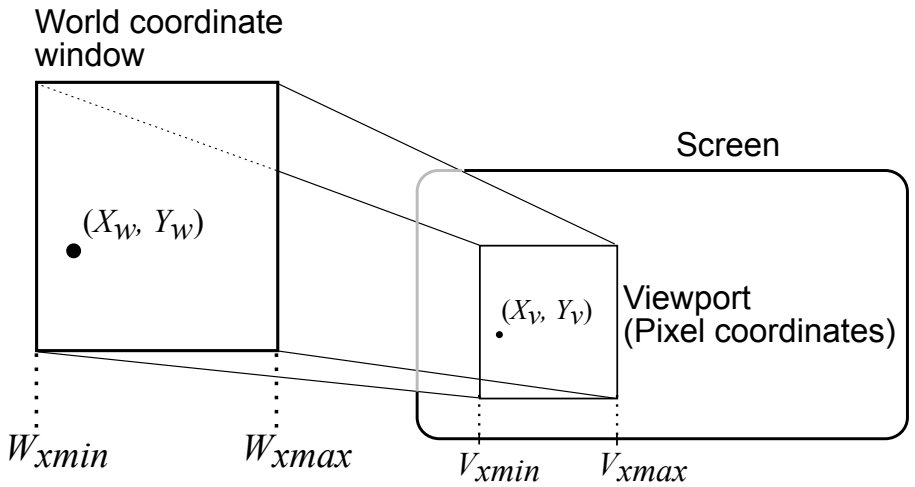


Figure 1: The normalisation transformation

In the Windows API this procedure is called `GetClientRect()`. If we know both the world and device coordinate systems, it is possible to define a normalisation process to compute the pixel coordinates from the world coordinates (or vice versa). This can be done with simple ratios. Using the illustration in Figure 1, we can carry out the normalisation for the x direction as follows:

$$\frac{(X_w - W_{xmin})}{(W_{xmax} - W_{xmin})} = \frac{(X_v - V_{xmin})}{(V_{xmax} - V_{xmin})}$$

This equation is linear and relates X_w to X_v. It can be rearranged to the form $X_v = AX_w + B$ where A and B are constants that can be calculated from W_{xmin}, W_{xmax}, V_{xmin} and V_{xmax}. Applying the same idea to the y direction yields a pair of simple linear equations:

$$\begin{aligned} X_v &= AX_w + B \\ Y_v &= CY_w + D \end{aligned}$$

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

Graphical Input

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

The operating system must share control of the mouse with the application, since it needs to act on mouse actions that take place outside the graphics window. For instance, processing a menu bar or launching a different application. It therefore traps all mouse events (ie changes in position or buttons) and informs the program whenever an event has taken place using a "callback procedure". The application program must, after every action carried out, return to the callback procedure (or event loop) to determine whether any mouse action (or other event such as a keystroke) has occurred. The callback is the main program part of any application, and, in simplified pseudo code, looks like this:

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

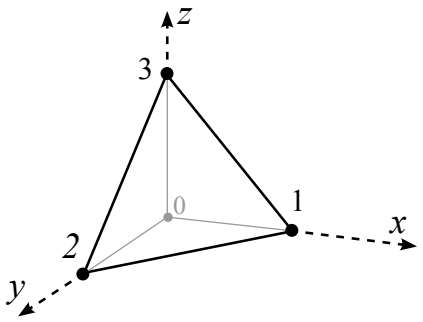
The procedure `ProcessMenuRequest` will be used to launch all the normal actions, such as save and open and quit, together with all the application specific requests. The procedures `GetMouseCoordinates` and `PerformMouseProcess` will be used by the programmer to create whatever effect is wanted, for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

Most graphical scenes and objects are made up of planar (flat) faces. Each face is an *ordered* set of 3D point vertices, lying on one plane, which form a closed polygon. Two types of data are needed to describe an object made up of polygons: These are the *locations* of the vertices and their *topology*.

The locations of the vertices needs to be specified by a set of coordinates. So, in 3-D, each vertex location will be described by three numbers and if the object has N points, we need $3 \times N$ numbers to describe all their locations. The topology of the object is a description of how the points are joined together to form the polygon faces.

A simple example is given by a tetrahedron which has four vertices and four triangular faces. Figure 2 illustrates how data might be written for a tetrahedron with vertices placed at the origin and one unit along each of the coordinate axes. The first two columns in the table give the vertex data. Each vertex has coordinates to specify its position and an index to specify its order in the list. Each face is specified by listing the vertices it uses in order. For example the second face uses the vertices at index 0, 2 and 1.



Vertex Data		Face Data
Index	Location	Vertices for face
0	$(0, 0, 0)$	0 1 3
1	$(1, 0, 0)$	0 2 1
2	$(0, 1, 0)$	0 3 2
3	$(0, 0, 1)$	1 2 3

Figure 2: An example of location and topology data that can be given to describe a tetrahedron. Each vertex has a location and an index (starting from zero). Each face is defined by the indices of the vertices it uses.

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 $m < n$. So, for example, an ordinary camera projects a 3-dimensional space into a two dimensional space. 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. This process is illustrated in Figure 3. The point in the surface where a projector intersects is defined as the projection of the corresponding vertex in the object.

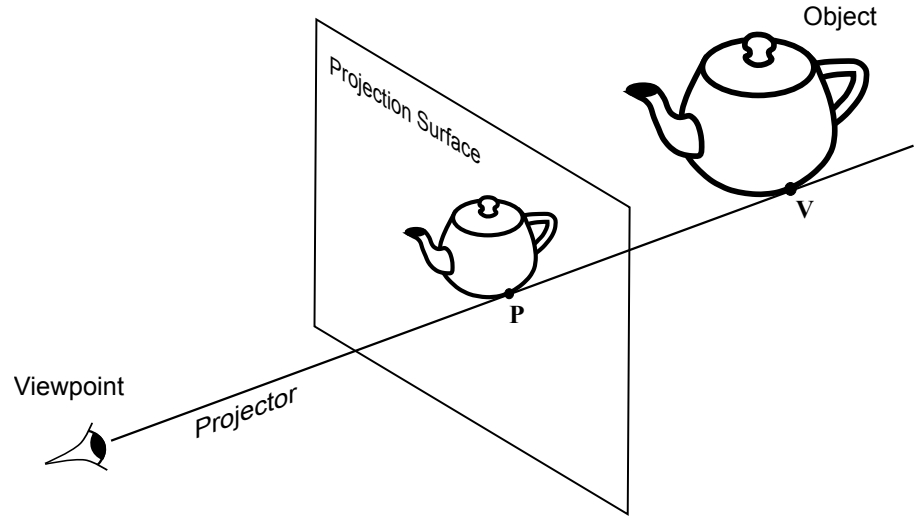


Figure 3: Planar Projection: The vertex $\mathbf{V}$ in the object is projected to the point $\mathbf{P}$ in the image.

The most common (and simplest) projections used for viewing 3D scenes use a plane for the projection surface and straight lines for the projectors. These are called planar geometric projections. A rectangular window can be 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 by drawing only the projected edges. This is called a 'wire-frame' representation. Note that for such rendering the topological information only needs to specify which edges join which points. For other forms of rendering we also need to define the object faces. For an example of a wire-frame representation, see the old arcade game 'Battlezone'.

There are two common types of planar geometric projection: *Parallel projection* uses parallel projectors and *perspective projection* uses 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 assuming that it is 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

h

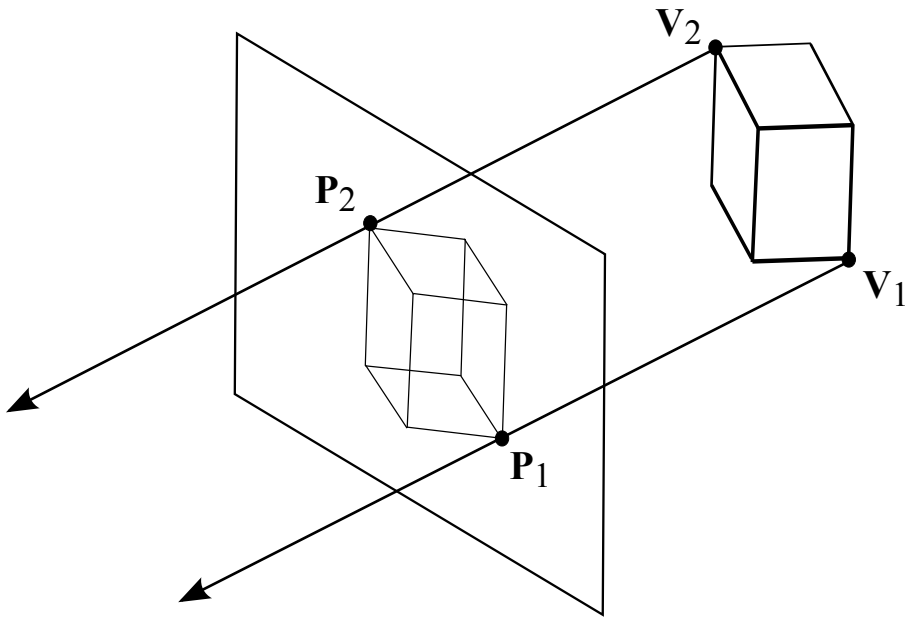


Figure 4: Parallel Projection: The projection lines (projectors) are parallel. Two are illustrated for object vertices $\mathbf{V}_1$ and $\mathbf{V}_2$ which project onto the points $\mathbf{P}_1$ and $\mathbf{P}_2$ in the projection surface.

to the $z = 0$ plane then we can use a coordinate transformations in 3D to make it so and we will see shortly how to do this. We shall also assume that the viewed objects are in the positive half space ($z > 0$), therefore the projectors, starting at the objects' 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. The process of parallel projection is illustrated in Figure 4. For a vertex $\mathbf{V} = (V_x, V_y, V_z)^T$, the projector is defined by the parametric line equation

$$\mathbf{P} = \mathbf{V} + \mu\mathbf{d} \tag{1}$$

A special case of parallel projection is *orthographic* projection in which the projectors are perpendicular to the projection plane, which we usually define as $z = 0$. In this case the projectors are in the opposite direction to the positive z axis, so:

$$\mathbf{d} = \begin{pmatrix} 0 \\ 0 \\ -1 \end{pmatrix}$$

So we can easily obtain the x and y coordinates of the projected point:

$$\begin{aligned} P_x &= V_x \\ P_y &= V_y \end{aligned}$$

This 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. The z-coordinate of $\mathbf{P}$ after projection is P_z and is always zero by definition because that is where we assume the projection plane to be. In any case, we ignore the z-coordinate because we are projecting on to a 2-D plane so we only need P_x and P_y to specify the location of a projected point. Some examples of a wireframe cube after applying orthographic projection are shown in Figure 5.

If the projectors are not perpendicular to the plane of projection then the projection is called *oblique*. We can represent this by keeping the projection plane as $z = 0$ but representing the direction $\mathbf{d}$ of the projectors as

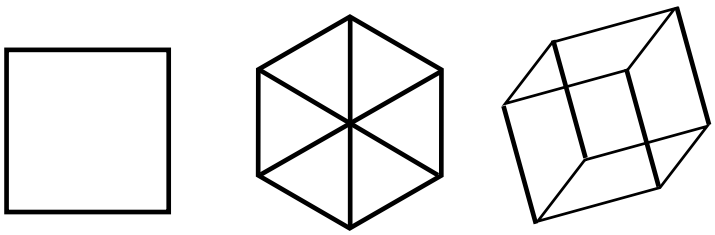


Figure 5: Different orthographic Projections of a cube. From left to right: Looking at a face; Looking at a vertex; A more general view.

a more general vector:

$$\mathbf{d} = \begin{pmatrix} d_x \\ d_y \\ d_z \end{pmatrix}$$

The projected vertex intersects the $z = 0$ plane where the z component of the $\mathbf{P}$ vector in Equation 1 is equal to zero. Therefore:

$$\begin{aligned} 0 = P_z &= V_z + \mu d_z \\ \text{so} \quad \mu &= \frac{-V_z}{d_z} \end{aligned}$$

We can use this value of μ to compute P_x and P_y:

$$P_x = V_x + \mu d_x = V_x - \frac{d_x V_z}{d_z} \quad \text{and} \quad P_y = V_y + \mu d_y = V_y - \frac{d_y V_z}{d_z}$$

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

Perspective Projections

For a perspective projection, all the projectors or rays pass through one point in space, the viewpoint which can also be described as the *centre of projection*. A schematic illustration of persepective projection is shown in Figure 6. If the centre of projection is on the opposite side of the plane of projection compared to the 3D ('real world') object, then the orientation of the image is the same as the object (as is the case in Figure 6). By contrast, in a pin hole camera the centre of projection is between the projection plane and the object which means that the image is inverted.

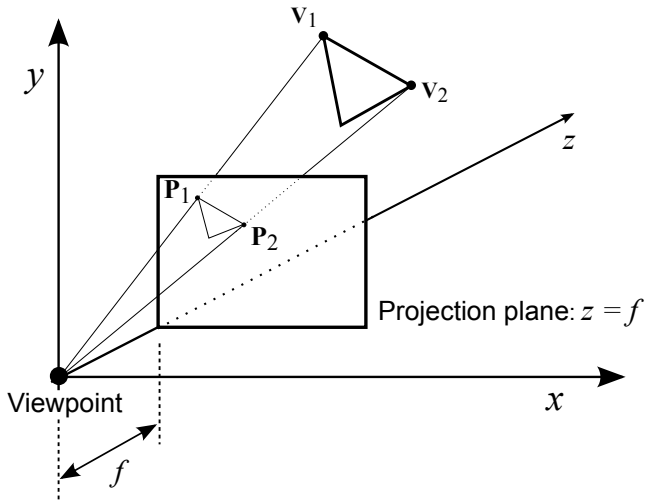


Figure 6: The canonical form for perspective projection (viewpoint at origin; projection plane at $z = f$). Two rays are shown projecting points $\mathbf{V}_1$ and $\mathbf{V}_2$ onto $\mathbf{P}_1$ and $\mathbf{P}_2$

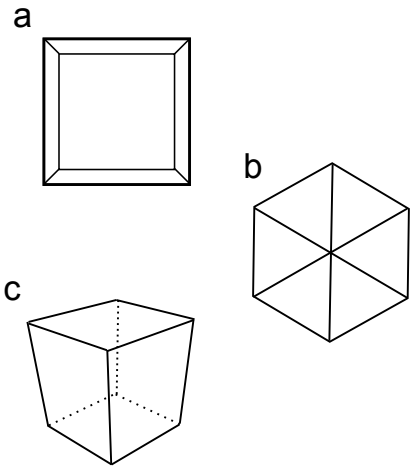


Figure 7: Perspective projections of a cube. Three views are shown: a. Viewing through a face. b. Through a vertex. c. A more general view.

To calculate how points are affected by a perspective projection, we make two simplifying assumptions: that the centre of projection is at the origin, and that the projection plane is placed at a constant z value, $z = f$.

This canonical form is illustrated in Figure 6. In order to calculate the projection of a 3D world point $\mathbf{V}$ onto the $z = f$ plane, we note that the projecting ray through the origin 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 the parameter value for the intersection point on the projection plane as $\mu_p = f/V_z$ then

$$P_x = \mu_p V_x = \frac{fV_x}{V_z} \quad \text{and} \quad P_y = \mu_p V_y = \frac{fV_y}{V_z}$$

The factor μ_p is called the 'foreshortening' factor, because, if the object moves further away, V_z becomes larger and the image becomes smaller (for a parallel projection this would not happen) . Some examples of the perspective projection of a cube are shown in Figure 7.

Space Transformations

The use of canonical forms for perspective and orthographic projection makes them easier to compute. However, sometimes we wish to move around a graphical scene and view it from *any* point of our choosing. Therefore, in order to still be able to use a canonical projection, we need to transform the coordinates of the scene so 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, $\mathbf{C} = (C_x, C_y, C_z)$, becomes the origin and the chosen view direction, expressed as a vector $\mathbf{d} = (d_x, d_y, d_z)^T$, becomes 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 8.

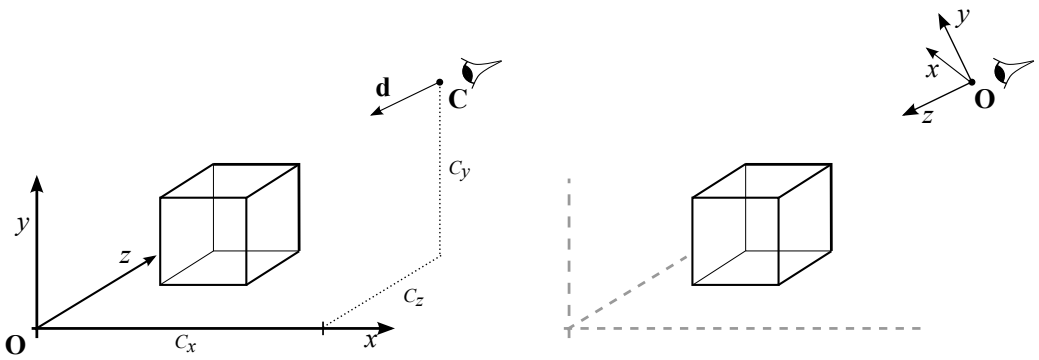


Figure 8: View centered coordinate transformation. Left: Coordinate system for definition. Right: Coordinate system for viewing.

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 can be achieved by multiplying the coordinates every point of the scene by a *transformation matrix*.

This is possible for nearly all simple transformations but, unfortunately, we cannot perform a general translation using normal Cartesian coordinates and matrix multiplication. For this reason, we introduce a new system for representing points called *homogeneous coordinates*.

Homogeneous coordinates and transformation matrices

An ordinary three dimensional point is written in terms of three coordinates

$$\mathbf{P} = (p_x, p_y, p_z)$$

In order to express a point in homogeneous coordinates, we introduce a fourth component (or '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 the homogeneous coordinates (p_x, p_y, p_z, s) are equivalent to the Cartesian coordinates $(p_x/s, p_y/s, p_z/s)$. In most cases s will be 1 which makes calculations easier.

The main point of introducing homogenous coordinates is to allow us to translate the points of a scene using matrix multiplication. In cartesian coordinates, applying a translation vector $\mathbf{t} = (t_x, t_y, t_z)^T$ to a point $\mathbf{P} = (p_x, p_y, p_z)$ results in a translated point $(p_x + t_x, p_y + t_y, p_z + t_z)$. This can be neatly expressed using matrix multiplication and homogeneous coordinates as follows:

$$\begin{pmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} p_x \\ p_y \\ p_z \\ 1 \end{pmatrix} = \begin{pmatrix} p_x + t_x \\ p_y + t_y \\ p_z + t_z \\ 1 \end{pmatrix}$$

Other simple transformations can still be easily expressed using homogeneous coordinates. For example, the matrix for scaling a graphical scene can be written

$$\begin{pmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} p_x \\ p_y \\ p_z \\ 1 \end{pmatrix} = \begin{pmatrix} s_x p_x \\ s_y p_y \\ s_z p_z \\ 1 \end{pmatrix}$$

An important point to note is that transformations like these are not necessarily commutative, i.e. carrying them out in different orders can lead to different results. So it is very important that they are carried out in the correct order and Figure 9 illustrates the problem for a simple picture.

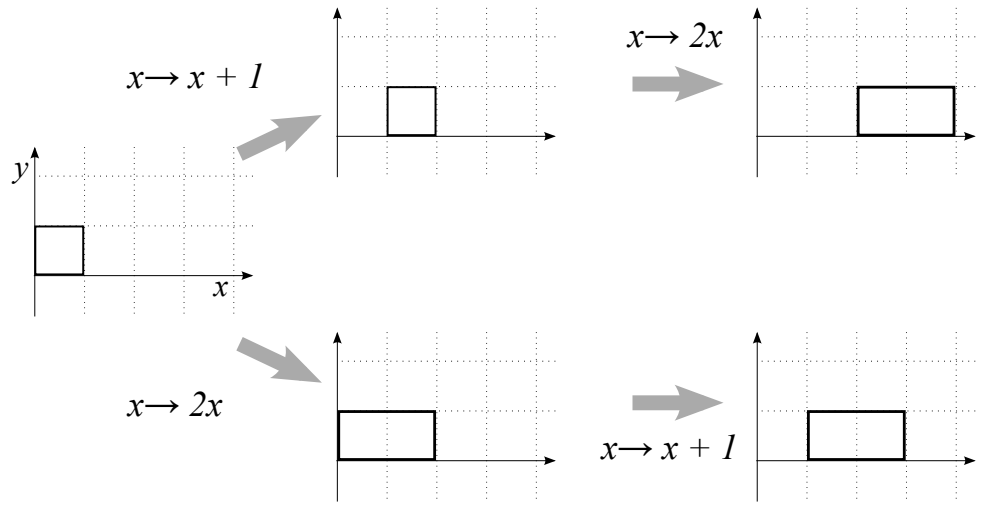


Figure 9: The order in which transformations are done is important. Applying two simple transformations to a unit square (left) in different orders gives two different results (right)

.

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

$$\mathcal{R}_x = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \mathcal{R}_y = \begin{pmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \mathcal{R}_z = \begin{pmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

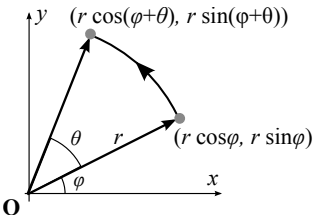
Some care is required with the signs. The above formulation assumes a left hand axis system. This means that, if the positive y-axis is taken as vertical, and the positive x-axis horizontal to the right, the positive z-axis is into the page. In this case, the matrices above correspond to an anti-clockwise rotation when looking *along* the axis in the postive direction.

As an example of how these matrices are obtained, we can derive the $\mathcal{R}_z$ matrix as follows by considering the polar form of a point (x, y) in the plane $z = 0$

$$(x, y) = (r\cos\varphi, r\sin\varphi)$$

Writing the coordinate as a vector and assuming it is rotated by an angle θ about the z-axis, we have the coordinate transformed as follows:

$$\begin{aligned}
\begin{pmatrix} x \\ y \end{pmatrix} &\rightarrow \begin{pmatrix} r\cos(\varphi+\theta) \\ r\sin(\varphi+\theta) \end{pmatrix} \\
&= \begin{pmatrix} r\cos\varphi\cos\theta - r\sin\varphi\sin\theta \\ r\cos\varphi\sin\theta + r\sin\varphi\cos\theta \end{pmatrix} \\
&= \begin{pmatrix} x\cos\theta - y\sin\theta \\ x\sin\theta + y\cos\theta \end{pmatrix} \\
&= \begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix}
\end{aligned}$$



(z-axis goes 'into' page)

The 2×2 matrix on the last line is the same as the upper left corner of the matrix $\mathcal{R}_z$ written above. The other rotation matrices, $\mathcal{R}_x$ and $\mathcal{R}_y$, may be derived similarly.

Inverting transformation matrices

Inversions of the transformation matrices can be computed easily, without needing to apply an equation solving method, such as Gaussian elimination. This can be done by considering the meaning of each transformation. For scaling, we can invert the matrix by substituting $1/s_x$, $1/s_y$ and $1/s_z$ instead of s_x, s_y and s_z.

Scaling matrix inverse

$$\begin{pmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad \begin{pmatrix} 1/s_x & 0 & 0 & 0 \\ 0 & 1/s_y & 0 & 0 \\ 0 & 0 & 1/s_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

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

Translation matrix inverse

$$\begin{pmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad \begin{pmatrix} 1 & 0 & 0 & -t_x \\ 0 & 1 & 0 & -t_y \\ 0 & 0 & 1 & -t_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

For a rotation matrix, the inverse of a rotation by θ is equivalent to a rotation by $-\theta$. Also, we note that

$$\cos(-\theta) = \cos\theta \qquad \text{and} \qquad \sin(-\theta) = -\sin\theta$$

and we can use these relations to write the inverse of a rotation matrix. For example, for a rotation about the z-axis

$\mathcal{R}_z$ inverse

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