

COMP3201 – Computer Graphics

Module 2: Transformations and Scene Creation

2.4 Projection Transformations

2.4.1 Introduction

The purpose of the projection transformations is to convert the eye coordinates into clip coordinates. You may ask why go through the process of clip coordinates and normalized device coordinates (NDC) before ending up at screen coordinates when we could project directly from eye coordinates to screen coordinates. There are many reasons for this multi-step pipeline. One of these reasons is that it is possible to implement more efficient clipping algorithms in NDC than in eye coordinates. A second reason is to cater for projections that require non-linear transformations. Non-linear transformations (like a projection transformation) usually require a division. This can be implemented by making use of the homogeneous coordinate, which later divides the other three coordinates. This means that we can stay with the four homogeneous coordinates for as long as possible and thus make use of the same hardware or software that has been optimized for doing the 4x4 matrix multiplications that the other stages of the rendering pipeline require.

Although the OpenGL specification does not impose left or right handedness on coordinate systems, in all current systems it is true that the world coordinates are a right handed coordinate system while the screen coordinates are a left handed coordinate system. Somewhere along the rendering pipeline this switch has to be made. The most natural place for the switch to occur is in the projection transformation.

2.4.2 Derivation of the Orthographic Projection

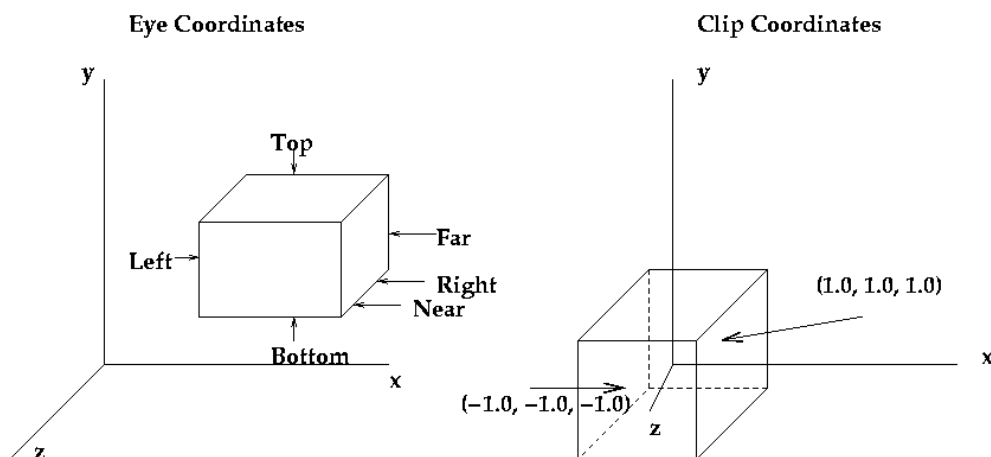


Figure 2.15

It is quite easy to see how to move the viewing volume from the eye coordinates to the volume required in clip coordinates. Firstly translate the centre of the viewing volume to the origin. We will denote this operation by T . Next scale the x, y, z directions so that they form the $2 \times 2 \times 2$ cube. Denote this operation by S_c . So the final projection transformation P is given by

$$P = S_c T$$

The centre of the viewing volume in eye coordinates is

$$Centre = \left(\frac{left + right}{2}, \frac{top + bottom}{2}, \frac{-(near + far)}{2}, 1 \right)$$

Therefore the translation matrix to move the centre of the viewing volume to the origin is

$$T = \begin{bmatrix} 1 & 0 & 0 & -\left(\frac{right+left}{2}\right) \\ 0 & 1 & 0 & -\left(\frac{top+bottom}{2}\right) \\ 0 & 0 & 1 & \left(\frac{far+near}{2}\right) \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The length along the x -axis of the viewing volume is given by $right - left$. Thus to reduce this length to 2 units (i.e. from -1 to $+1$) in the clip coordinates we must scale by a factor of $2/(right - left)$. Similarly we will scale the y -axis by $2/(top - bottom)$. The scale factor for the z -axis is given by $-2/(far - near)$. The negative sign is due to the change from the right- to left-handed coordinate systems. Note that this means that $z = -near$ will become -1 and $z = -far$ will become $+1$.

Putting this into a scaling matrix

$$S_c = \begin{bmatrix} \frac{2}{right-left} & 0 & 0 & 0 \\ 0 & \frac{2}{top-bottom} & 0 & 0 \\ 0 & 0 & \frac{-2}{far-near} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Thus the total projection matrix is

$$P = \begin{bmatrix} \frac{2}{right-left} & 0 & 0 & -\left(\frac{right+left}{right-left}\right) \\ 0 & \frac{2}{top-bottom} & 0 & -\left(\frac{top+bottom}{top-bottom}\right) \\ 0 & 0 & \frac{-2}{far-near} & -\left(\frac{far+near}{far-near}\right) \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

2.4.3 Derivation of the Perspective Projection

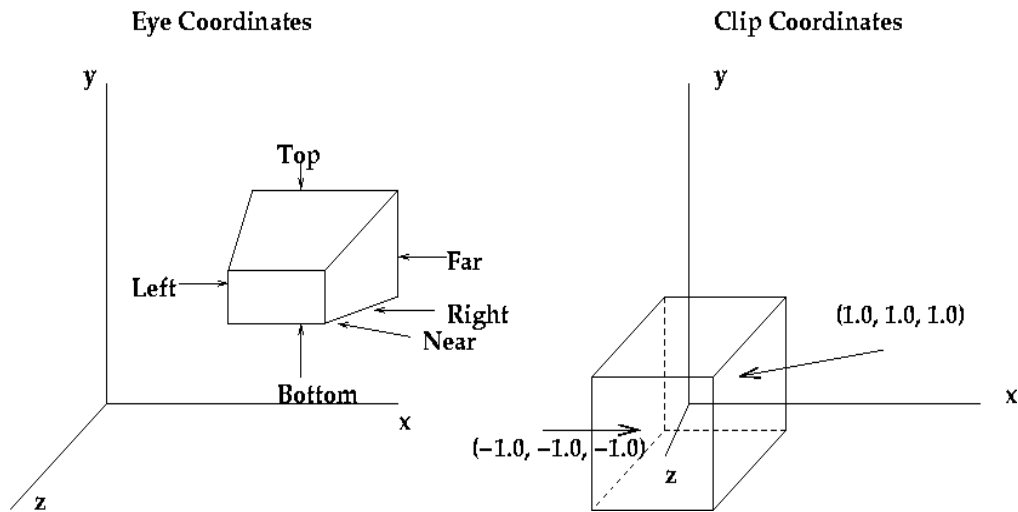


Figure 2.16

Deriving the perspective projection uses the same principles as the orthographic case, but it is slightly more involved. In general the frustum specified for the viewing volume is not symmetrical (it is symmetrical if you use `gluPerspective`). Thus the first step is to correct the asymmetry. We will denote this operation by S_h . The second step will be to convert the now symmetrical frustum into a rectangular prism centred on the origin. This operation will be denoted by V (for volume correction). The last step is to scale the x , y , z directions so that they form the $2 \times 2 \times 2$ cube. Denote this operation by S_c . So the final projection transformation P is given by

$$P = S_c V S_h$$

2.4.3.1 Shear Correction

In the x - z plane the viewing frustum will in general look like

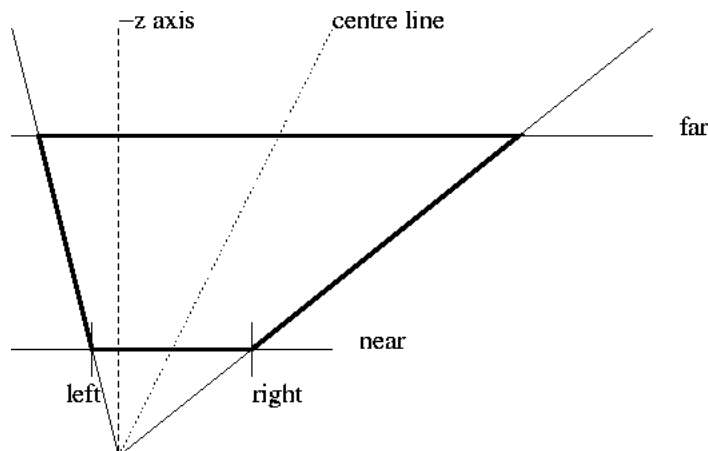


Figure 2.17

We need to align the line denoting the centre of the frustum (the dotted line) with the z -axis. This is done by shearing in the x -direction proportionally to the z -value. Similarly we also need to do the same operation in the y - z plane. The shear matrix to achieve this is given by

$$S_h = \begin{bmatrix} 1 & 0 & \frac{1}{near} \frac{right+left}{2} & 0 \\ 0 & 1 & \frac{1}{near} \frac{top+bottom}{2} & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

There are two interesting points to note about this matrix. The first is why don't the shearing terms have a negative sign since we want to achieve a shear to the left? The answer is because a positive shear (i.e. to the right) along the positive z -axis is simultaneously a shear to the left along the negative z -axis. The second point of interest is the $1/near$ factor. The reason for this becomes clear by multiplying out the matrix against a point.

$$x_{new} = x_{old} + \frac{1}{near} \frac{right+left}{2} z_{old}$$

This means that when the z value is $-near$, then the shear factor is $-(right+left)/2$ and as z becomes larger the shearing becomes larger, as required. Refer to Figure 2.16 to see what is happening. You should also note that this procedure has had the pleasant side effect of centering the volume on the x and y axes. This means that we won't have to do a translation for these axes as we did in the orthographic case.

Now that the viewing volume is symmetric we need to convert it to a rectangular prism. We will do this by shrinking the far clipping plane to be the same length and breadth as the near clipping plane. It is this conversion that gives the perspective to the objects in the volume. To understand this think about what happens to a cube inside the volume while the back clipping plane is being shrunk.

In order to build up the transformation matrix to achieve this, we first need to understand how traditional perspective transformations are done.

2.4.3.2 Volume Correction

Traditional Perspective Projection

Let's have a look at the x - z plane again.

To project a point P from inside the volume onto the near clipping plane, draw a line from the point to the missing peak of the truncated pyramid (see Figure 2.18). Using similar triangles we can see that

$$\frac{x_p}{z_p} = \frac{x_n}{near}.$$

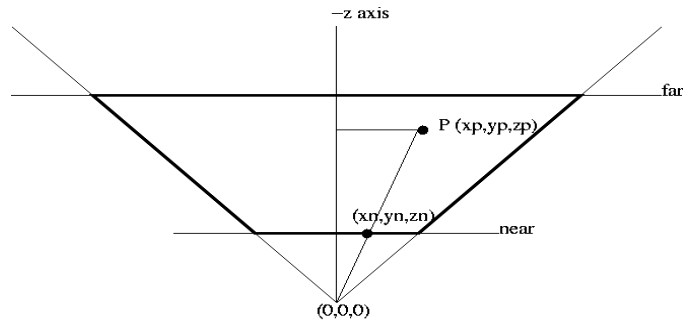


Figure 2.18

Thus the projected x coordinate is given by

$$x_n = near \frac{x_p}{z_p}.$$

Similarly the projected y coordinate is given by

$$y_n = near \frac{y_p}{z_p}.$$

These equations mean that as the z values increase, the x and y coordinates are projected closer to the origin. That is, objects get smaller (as expected).

Perspective retaining the z coordinate

For us to achieve a perspective projection we will have to use x and y coordinates as given above. However we still need to keep the z coordinate so that we have depth information for clipping and other purposes.

Let us examine the perspective transformation equations more closely. In both equations we need to multiply the original values by $near$ and divide by z_p . The multiplication is easy to achieve. The diagonal elements of a matrix do this. Division by a coordinate value can't be done in linear algebra under the usual coordinate system, however the homogeneous coordinate system gives us the flexibility required. The trick is to use the bottom row of the matrix.

$$\begin{bmatrix} near & 0 & 0 & 0 \\ 0 & near & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} near \cdot x \\ near \cdot y \\ z \\ z \end{bmatrix} \Rightarrow \begin{bmatrix} \frac{near \cdot x}{z} \\ \frac{near \cdot y}{z} \\ z \\ 1 \end{bmatrix}$$

While this trick gives us the perspective transformation of the x and y coordinates that we needed, the z value is now mapped to a constant, which is not what we want. What we do want is for z to range from -1 to $+1$. We also want to move to a left-handed coordinate system. In this coordinate system we want $z = -near$ to become -1 and $z = -far$ to become $+1$.

The trick to mapping the z coordinate is to scale and translate. The trick to moving to a left-handed coordinate system is to turn the 1 in the bottom row to a -1 .

$$V = \begin{bmatrix} \textit{near} & 0 & 0 & 0 \\ 0 & \textit{near} & 0 & 0 \\ 0 & 0 & B & C \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

The z coordinate is thus mapped to (after homogeneous scaling)

$$z_n = -B - \frac{C}{z_p}$$

Using the information that $z_n = -1$ when $z_p = -\textit{near}$ and that $z_n = +1$ when $z_p = -\textit{far}$ it is easy to solve for B and C (this is an exercise for the reader).

$$B = -\frac{\textit{far} + \textit{near}}{\textit{far} - \textit{near}} \quad C = -2\frac{\textit{near} \cdot \textit{far}}{\textit{far} - \textit{near}}$$

2.4.3.3 Scaling Correction

While the z axis has been scaled properly during the volume change, the x and y axes still need to be rescaled. Along the x -axis the length of the volume is given by the quantity *right* – *left*. Along the y -axis the length of the volume is given by the quantity *top* – *bottom*. The scaling matrix to reduce each of these lengths to be 2 units long (i.e., from -1 to $+1$ since we are already centred on the origin) is

$$S_c = \begin{bmatrix} 2\frac{1}{\textit{right} - \textit{left}} & 0 & 0 & 0 \\ 0 & 2\frac{1}{\textit{top} - \textit{bottom}} & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

2.4.3.4 Complete Transformation

The complete perspective volume transformation is given by

$$P = S_c V S_h.$$

Thus

$$P = \begin{bmatrix} 2\frac{1}{\textit{right} - \textit{left}} & 0 & 0 & 0 \\ 0 & 2\frac{1}{\textit{top} - \textit{bottom}} & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \textit{near} & 0 & 0 & 0 \\ 0 & \textit{near} & 0 & 0 \\ 0 & 0 & -\frac{\textit{far} + \textit{near}}{\textit{far} - \textit{near}} & -2\frac{\textit{far} \cdot \textit{near}}{\textit{far} - \textit{near}} \\ 0 & 0 & -1 & 0 \end{bmatrix} \begin{bmatrix} 1 & 0 & \frac{1}{n}\frac{\textit{right} + \textit{left}}{2} & 0 \\ 0 & 1 & \frac{1}{n}\frac{\textit{top} + \textit{bottom}}{2} & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$= \begin{bmatrix} 2 \frac{\text{near}}{\text{right-left}} & 0 & \frac{\text{right+left}}{\text{right-left}} & 0 \\ 0 & 2 \frac{\text{near}}{\text{top-bottom}} & \frac{\text{top+bottom}}{\text{top-bottom}} & 0 \\ 0 & 0 & -\frac{\text{far+near}}{\text{far-near}} & -2 \frac{\text{far-near}}{\text{far-near}} \\ 0 & 0 & -1 & 0 \end{bmatrix}.$$

2.4.4 Projection Transformations in OpenGL

The modelview transformations described in sections 2.2 and 2.3 are the modelling transformations that are applied to objects in the scene. The other type of transformation is the projection transformation, as described in the beginning of this section. It defines how the scene is to be mapped onto the screen. A projection transformation can be thought of as defining the lens of the camera.

The projection matrix stack is used to save information about the projection and viewing transformations; a particular viewing volume state can then be reloaded at a later stage in the program.

This subsection describes how to define the two possible types of projection (orthographic, which is also called parallel, and perspective, which makes distant objects appear smaller) in OpenGL.

An orthographic projection is square, with no tapering in the distance. The default viewing volume is the 2 x 2 x 2 cube, centred at the origin; it can be specified by

```
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
glOrtho(-1.0, 1.0, -1.0, 1.0, -1.0, 1.0);
```

Parameters 5 and 6 in **glOrtho** specify the near plane ($z = -1.0$, in front of the viewpoint) and the far plane ($z = 1.0$, behind the viewpoint). In general an orthographic viewing volume (a parallelepiped) can be defined using

```
glOrtho(left, right, bottom, top, near, far);
```

A frustum is used to define the perspective viewing volume, and is set by the function **glFrustum(xmin, xmax, ymin, ymax, near, far)**. The coordinates here are eye coordinates. An alternative way of setting up the frustum is to use

```
gluPerspective(fovy, aspect, near, far);
```

where **fovy** is the field-of-view angle vertically (y -axis), **aspect** is the aspect ratio of height to width, and **near** and **far** define the distance to the near and far clipping planes (along the z -axis).

Troubleshooting Tip

A common error is to forget the settings of the near and far clipping planes, so if your objects do not appear on the screen, it is a good idea to check the viewing volume parameter values.

2.4.5 Viewing Transformations

Viewing Transformations are used to set up where we are viewing the scene from. Using a camera analogy, we look through the camera at the scene, and a viewing transformation moves the camera to a new position.

One way of specifying the camera position is to use the function **gluLookAt** which defines the eye point, the centre, and the upwards direction of the camera; the camera is pointing in the direction centre – eye.

```
gluLookAt(eyex, eyey, eyez, centrex, centrey, centrez, upx, upy, upz);
```

The viewing transformation is specified on the ModelView matrix stack. For example,

```
glMatrixMode(GL_MODELVIEW);  
glLoadIdentity();  
gluLookAt( ... );  
/* Now include your model transformations */
```

Another viewing transformation allows the model to be viewed as though it is in a glass sphere.

```
void polarView(GLfloat distance, GLfloat azimuth,  
    GLfloat incidence, GLfloat twist)  
{  
    glTranslatef(0.0, 0.0, -distance);  
    glRotatef(-twist, 0.0, 0.0, 1.0);  
    glRotatef(-incidence, 1.0, 0.0, 0.0);  
    glRotatef(-azimuth, 0.0, 0.0, 1.0);  
}
```

The parameter **distance** is measured from the view point to the origin of the world coordinates; **azimuth** is the viewing angle of the observer, measured in the x - y plane from the y -axis; the angle of **incidence** is in the y - z plane, measured from the z -axis; **twist** is the amount of rotation required around the “pointing” direction, using the righthand rule.

Yet another model-view matrix could be specified using three angles (roll, pitch and yaw) to specify the orientation of the camera. This viewing transformation is well-suited for flight simulations. Consider an aeroplane positioned at the origin, pointing down the positive z -axis. Then **roll** measures a rotation about the z -axis, **pitch** measures a rotation about the x -axis, while **yaw** is a rotation about the y -axis.

2.4.6 Viewports and Window-Resizing

The viewport is the rectangular part of the display window that you render into. By default, it is the entire display window, but can be customised as you want by calling the function

```
GLvoid glViewport(GLint x, GLint y, GLsizei width, GLsizei height);
```

The viewport is specified in pixels, with (x,y) giving the lower-left corner of the viewport and width and height completing the definition. Multiple viewports can be defined in the display window.

To avoid distortion between the scene we have constructed and its representation in the viewport, we need the rectangles to have the same aspect ratio. The aspect ratio is defined to be width/height, and this ratio should be the same for the window, the viewport and the viewing volume (which was defined via **glOrtho(left, right, bottom, top, near, far)**).

Because a user can resize the window being used, it is important for the programmer to update the viewport and viewing volume so that the correct aspect ratio is maintained. The easiest way to keep consistent aspect ratios is to register a callback for reshape events. As usual, the registration is done by **glutReshapeFunc(reshape)** which should be called in **main**. GLUT will pass the function **reshape** the window's new height and width so that it can reset the viewport and viewing volume.

For example, the **reshape** function could be

```
GLvoid reshape(GLsizei width, GLsizei height);  
{  
    GLdouble aspect;  
    aspect = (GLdouble) width / (GLdouble) height;  
    /* update the viewport */  
    glViewport(0, 0, width, height);  
    /* reset the viewing volume */  
    glMatrixMode(GL_PROJECTION);  
    glLoadIdentity();  
    gluPerspective(45.0f, aspect, near, far);  
    glMatrixMode(GL_MODELVIEW);  
}
```

Here it is assumed that the viewport is the entire window.