An Introduction to 3D Computer Graphics, Stereoscopic Image, and Animation in OpenGL and C/C++

Fore June

Chapter 4 Projection Transformations

4.1 Projections

We create a 3D graphics scene using certain models. At the end, all the 3D objects will be projected on a 2D screen. In this chapter, we examine the principles of projection and how to define the desired projection matrix to transform the vertices in our scene.

In the projection process, we define a **viewing volume**, which is bounded by a **near plane**, a **far plane**, and four side-planes as shown in Figure 4-1. All the scenes inside the viewing volume are projected into the near plane, which is also called the **view plane**. All the objects or portions of objects that lie outside the viewing volume are clipped and will not appear in the final projected image. The position of the eye (viewpoint) is called the **center of projection** (point O of Figure 4-1).

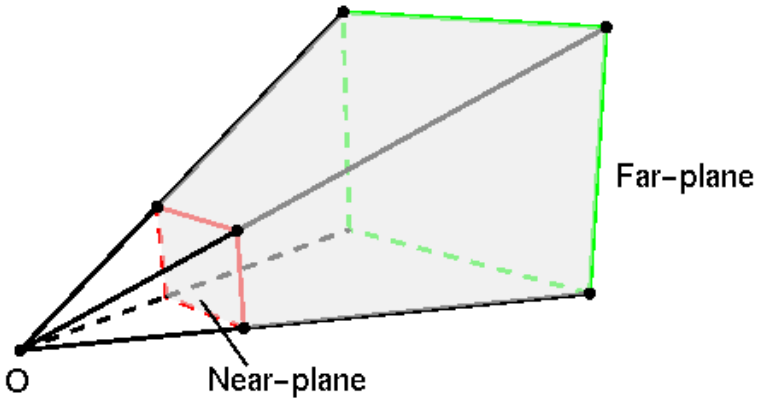
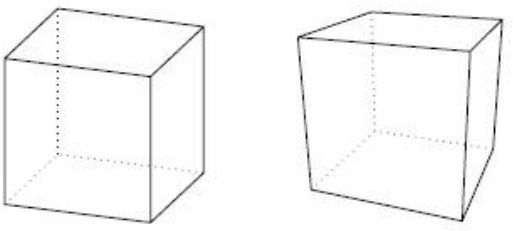


Figure 4-1 3D to 2D Projection

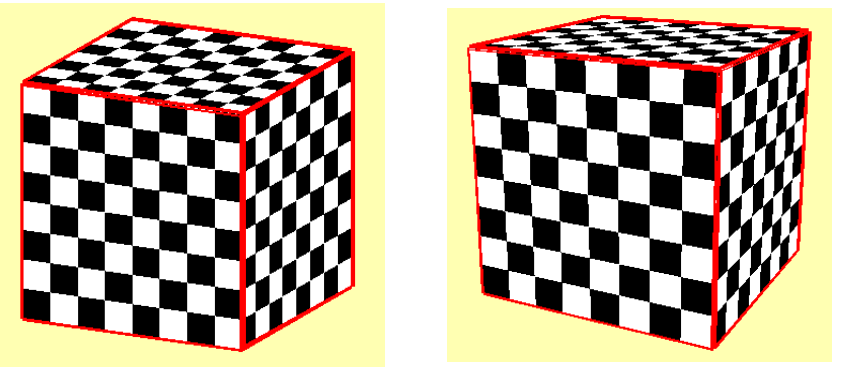
In most cases, people use one of the two techniques, **perspective projection** or **orthographic projection** to make 3D-to-2D projection in graphics. In perspective projection, the farther an object is from the viewpoint, the smaller it appears in the projected image. In contrast, orthographic projection projects objects in the same way whether they are near or far away from the viewpoint. Figure 4-2 shows a cube that is rendered with (a) orthographic projection, and (b) perspective projection.



(a) (b)

Figure 4-2 Cubes Rendered with (a) Orthographic Projection, and (b) Perspective Projection

Figure 4-3 shows a scene that is rendered with (a) orthographic projection, and (b) perspective projection. Note in the scene that in orthographic projection, the diamond-shaped texture pattern appears the same whether it is near or far away from the viewpoint; however, in perspective projection, the far-away pattern appears smaller.



(a) (b)

Figure 4-3 Scenes Rendered with (a) Orthographic Projection, and (b) Perspective Projection

4.2 Perspective Projections

4.2.1 Vanishing Points

A perspective projection is characterized by the fact that a distant object appears smaller as compared to a near object with the same size. Typically, such a projection is associated with one, two or three vanishing points. A **vanishing point** is a point in a perspective drawing or an image obtained from perspective projection to which parallel lines appear to converge.

A **one-point perspective** image consists of one vanishing point. Any objects that are made up of lines either directly parallel with the viewer's line of sight or directly perpendicular to it. Figure 4-4(a) below shows a cube that is rendered as a one-point perspective image. Figure 4-5 shows two one-point perspective photos.

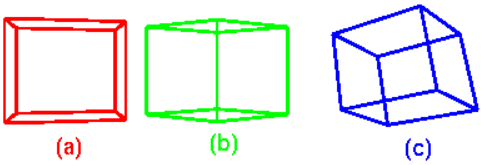


Figure 4-4 Cubes Rendered with (a)One-point, (b)Two-point, and (c) Three-point Perspectives

Figure 4-5 One-point Perspective Photos

A **two-point perspective** image consists of two vanishing points. One point represents one set of parallel lines, and the other point represents the other set. For example, when we look at a house from a corner, one wall would recede towards one vanishing point, and the other wall

would recede towards the opposite vanishing point. Figure 4-4(b) shows a cube rendered with the two-point perspective technique. Figure 4-6 shows two-point perspective drawings.

A **three-point perspective** image consists of three vanishing points meaning that we have three sets of parallel lines that converge to three different points. For example, when we look up at a tall building from a corner, the third vanishing point is high in space. Figure 4-4(c) shows a cube rendered with three-point perspective. Figure 4-7 shows a three-point perspective image.

Figure 4-6 Two-point Perspective Drawing

Figure 4-7 Three-point Perspective Image

4.2.2 Perspective Projection Transformations

Perspective projection is accomplished through the use of a frustum-shaped viewing volume as shown in Figure 4-8 below. In the figure, **O** is the center of projection (viewpoint or camera position). The view-frustum is the volume bounded by a near plane, a far plane and the field of view (fov). Objects that fall within the viewing volume are projected towards the center of projection where the camera or viewpoint is at. Objects or portions of objects outside the frustum are clipped.

In Figure 4-8, the eye or the camera is at the origin O, looking towards the negative z direction; the near-plane is at $z = -n$ and the far-plane is at $z = -f$; more detailly,

$$\begin{array}{llll} n = & \text{near distance} & f = & \text{far distance} \\ l = & \text{left} & r = & \text{right} \\ b = & \text{bottom} & t = & \text{top} \\ O = & (0,0,0) & & \end{array} \tag{4.1}$$

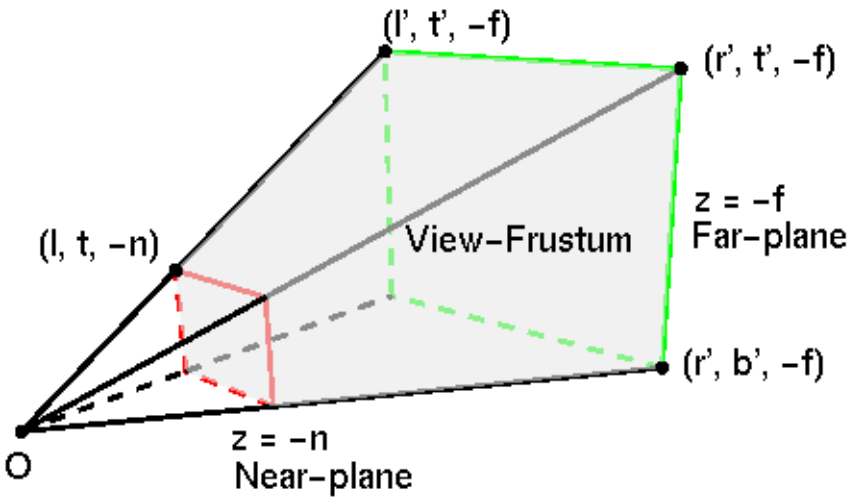


Figure 4-8 Frustum-shaped Viewing Volume

Any point (x, y, z) inside the frustum is projected into a point (x', y', z') on the screen at $z = -n$ (near-plane). Obviously, z' is always equal to $-n$. The values of x' and y' can be easily calculated using similar triangles shown in Figure 4-9. For example, $\frac{x}{x'} = \frac{z}{z'} = \frac{z}{-n}$. Thus, $x' = \frac{-nx}{z}$. Equation (4.2) shows the calculated values.

$$\begin{pmatrix} x' \\ y' \\ z' \end{pmatrix} = \begin{pmatrix} \frac{-nx}{z} \\ \frac{-ny}{z} \\ -n \end{pmatrix} \tag{4.2}$$

Any point lying outside the frustum is **not** projected. Therefore, $l \leq x' \leq r$ and $b \leq y' \leq t$. That is,

$$\begin{aligned} l &\leq \frac{nx}{-z} \leq r \\ b &\leq \frac{ny}{-z} \leq t \end{aligned} \tag{4.3}$$

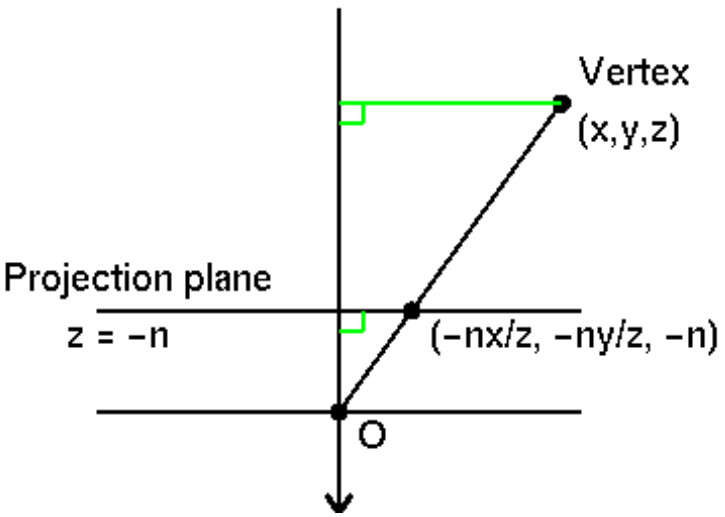


Figure 4-9 Perspective Projection Values

Obviously, the mapping given by Equation (4.2) is a **many-to-one** mapping. Many different (x, y, z) points in the frustum may be projected into the same (x', y', z') at the near-plane. We want to find a transformation matrix that will map (x, y, z) to (x', y', z') of (4.2) in a form similar to that of an affine transformation (translation, rotation, and scaling) that we discussed in Chapter 3. However, each mapping in Equation (4.2) involves a **division** by the z coordinate! We know that matrix multiplications of affine transformations only have additions and multiplications! *How can we combine this perspective transformation with the affine transformations*?

The trick is to make use of the characteristics of homogeneous coordinates that we have discussed in the previous chapter. Recall that in the homogeneous coordinate representation, the point (x, y, z, w) represents the **same** point as does $(\alpha x, \alpha y, \alpha z, \alpha w)$ with $\alpha \neq 0$. Thus $(x, y, z, 1)$ and $(\frac{x}{w}, \frac{y}{w}, \frac{z}{w}, \frac{1}{w})$ are equivalent provided $w \neq 0$. For example, the point $(1, 2, 3)$ has the representations $(1, 2, 3, 1), (2, 4, 6, 2), (0.03, 0.06, 0.09, 0.03), (-1, -2, -3, -1)$, and so on in homogeneous coordinate system. Therefore, to convert from homogeneous coordinates to ordinary coordinates, we need to

1. divide all components by the fourth component, and
2. discard the fourth component.

For example, $(3, 6, 2, 3) \rightarrow (1, 2, \frac{2}{3}, 1) \rightarrow (1, 2, \frac{2}{3})$. On the other hand, to convert from ordinary coordinates to homogeneous coordinates, we simply append a 1 as the fourth component. For example, $(2, 3, 4) \rightarrow (2, 3, 4, 1)$.

So, both representations

$$\begin{pmatrix} \frac{-nx}{z} \\ \frac{-ny}{z} \\ -n \\ 1 \end{pmatrix} \quad \text{and} \quad \begin{pmatrix} x \\ y \\ z \\ \frac{-z}{n} \end{pmatrix}$$

represent the **same** 3D point

$$\begin{pmatrix} \frac{-nx}{z} \\ \frac{-ny}{z} \\ -n \end{pmatrix}$$

Therefore, finding the matrix that maps

$$\begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} \rightarrow \begin{pmatrix} \frac{-nx}{z} \\ \frac{-ny}{z} \\ -n \\ 1 \end{pmatrix}$$

is the same as finding the matrix that maps

$$\begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} \rightarrow \begin{pmatrix} x \\ y \\ z \\ \frac{-z}{n} \end{pmatrix}$$

Such a matrix is given by

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & \frac{-1}{n} & 0 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x \\ y \\ z \\ \frac{-z}{n} \end{pmatrix} \tag{4.4}$$

Therefore, the perspective matrix A_0 that projects a point in the frustum into a point in the near-plane is given by

$$A_0 = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & \frac{-1}{n} & 0 \end{pmatrix} \tag{4.5}$$

Matrix A_0 no longer involves any division by the z coordinate. It seems that we can now seamlessly combine this matrix with the matrix of an affine transformation. However, there is a catch here. The matrix given in (4.5) is **singular**, meaning that its inverse does not exist. That is, it is noninvertible. This is because we have many points in the 3D view-frustum projected into the same point at the 2D view-plane. The solution to this problem is to treat the z coordinate separately. We actually don't care about the z coordinate after the transformation; all we care about are the x and y coordinates where the point is rendered on the screen. We therefore modify the transformation matrix A_0 to A that operates on a point P as follows:

$$P' = AP = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & \frac{-1}{n} \\ 0 & 0 & \frac{-1}{n} & 0 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x \\ y \\ z - \frac{1}{n} \\ \frac{-z}{n} \end{pmatrix} \tag{4.6}$$

Of course, we know that in homogeneous coordinate system, we can always divide each coordinate of the projected point given on the right side of (4.6) by the fourth coordinate, which is $\frac{-z}{n}$. Thus, the projected point is equal to the following:

$$P' = \begin{pmatrix} x \\ y \\ z - \frac{1}{n} \\ \frac{-z}{n} \end{pmatrix} = \begin{pmatrix} \frac{-nx}{z} \\ \frac{-ny}{z} \\ -n + \frac{1}{z} \\ 1 \end{pmatrix} \tag{4.7}$$

Converting back to ordinary 3D space, the point (x, y, z) now maps to $(\frac{-nx}{z}, \frac{-ny}{z}, -n + \frac{1}{z})$, which will be projected onto the screen at $(\frac{-nx}{z}, \frac{-ny}{z}, 0)$; the third component has been peeled off for depth testing.

The transformation matrix A is nonsingular. Its inverse is given by

$$A^{-1} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & \frac{-1}{n} \\ 0 & 0 & \frac{-1}{n} & 0 \end{pmatrix}^{-1} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & -n \\ 0 & 0 & -n & -n^2 \end{pmatrix} \tag{4.8}$$

We have derived the perspective transformation matrix A given by (4.6) and its inverse A^{-1} given by (4.8). However, in the derivation process we have assumed that the camera (viewpoint) is at the origin looking at the negative z direction; the projection plane is perpendicular to the z-axis and is at a distance n from the origin. Any point (x, y, z) inside the viewing frustum is projected into (x', y') at the view-plane. One may refer to this coordinate system as the **eye coordinate**

system because the origin is at the eye position (viewpoint). Some authors may write (x, y, z) as (x_e, y_e, z_e) with the subscript e signifying "eye"; they may express the projected point (x', y', z') as (x_p, y_p, z_p) with the subscript p signifying "projection". For simplicity, we shall stick to our notations (x, y, z) and (x', y', z') to denote these quantities. *What if the camera is not at the origin and it is not aiming at the negative z direction?* In this case, we have to first transform the camera to the origin and align its orientation to be the same as that shown in Figure 4-8. This is what the function **gluLookAt**() does. In other words, in the OpenGL pipeline, after the modelview transformation, coordinates are expressed in the eye coordinate system. Note that at this point, the coordinate values are still expressed in the world coordinate system (**WCS**). For example, we may limit the range of the x coordinates from $-\pi$ to $+\pi$. The parameters l, r, b, and r are all specified in world coordinates. The next step is to transform the world coordinates to device coordinates. For example, a screen may be specified as 800×640 pixels and we need to define the position of a point in terms of pixels. To ensure that graphics output is independent of the display device, OpenGL will first map the projected values to normalized device coordinates (NDC) that we are going to discuss in the next section.

4.2.3 Normalized Device Coordinates

After we have projected the desired scenes into the world window bounded by l, r, b, t, we need to determine where to display the scene on the screen or a display device. That is, we need to specify our viewport and we specify it in device coordinates such as pixel column and pixel row. However, if we want to run our programs on several hardware platforms or graphic devices, we will run into the difficulties of obtaining identical scenes in various platforms or devices. There are two common conventions for device coordinate systems (DCS):

1. The origin is at the lower left corner, with x to the right and y upward.
2. The origin is at the top left corner, with x to the right and y downward.

Also, many different resolutions for graphics display devices exist. For example,

1. Workstations commonly have 1280×1024 frame buffers.
2. A PostScript page has 612×792 points, but it has 2550×3300 pixels at 300 dpi resolution.

Moreover, different devices may have different aspect ratios.

If we map directly from a world coordinate system to a device coordinate system, then changing our device requires rewriting the program to change many of the display parameters in order to obtain the same look-and-feel display as before. This is certainly a tedious task. A better approach is to use **normalized device coordinates** (NDC) as an intermediate coordinate system that gets mapped to the device layer as shown in Figure 4-10. In the figure, the world window bounded by l, r, b, t is mapped to the unit square bounded by $-1, 1, -1, 1$ in the NDC space. Normalized device coordinates (NDC) are also referred to as **normalized coordinates** because this representation makes a graphics package independent of the coordinate range for any specific output device. The coordinate systems for display devices are generally called **device coordinates**, or **screen coordinates** for the case of a video monitor. Often, the normalized coordinates are specified in left-handed coordinate system; that is, when we use our left hand to roll our fingers from the x axis to the y axis, our thumb points to the z-axis. In this representation, increasing positive distances from the xy plane (the screen or view-plane) can be interpreted as farther from the viewing position.

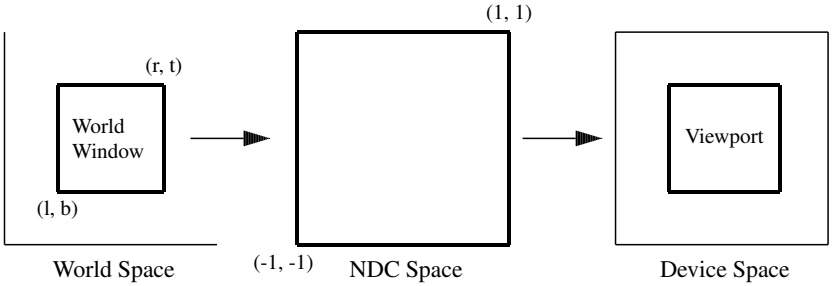


Figure 4-10 NDC as Intermediate Platform

OpenGL also uses this approach, mapping world coordinates to normalized device coordinates before transforming them to device coordinates. The truncated pyramid frustum shown in Figure 4-8 above is mapped to a cube with length 2 in normalized coordinates. The x-coordinate is from $[l, r]$ to $[-1, 1]$, the y-coordinate from $[b, t]$ to $[-1, 1]$ and the z-coordinate from $[-n, -f]$ to $[-1, 1]$. These are shown in Figure 4-11 and Figure 4-12. Figure 4-11 shows the view-frustum of perspective projection in the eye coordinate system as discussed in the above section; Figure 4-12 shows that the view-frustum is mapped to a $2 \times 2 \times 2$ cube in normalized device coordinate.

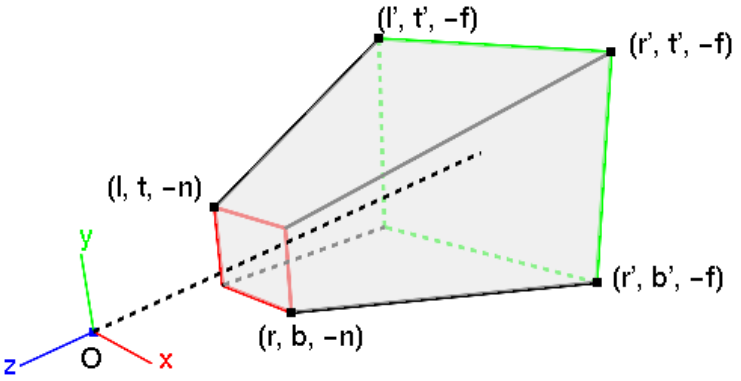


Figure 4-11 View-frustum of Perspective Projection in Eye Coordinates

Note that the eye coordinates are specified in right-handed coordinate system, but normalized coordinates are defined using left-handed coordinate system. As shown in Figure 4-11, the camera at the origin is looking along the $-z$ axis in eye space, but as shown in Figure 4-12, it is looking along $+z$ axis in normalized coordinate.

Now we want to find the transformation that projects a point (x, y, z) into $(x', y'z')$ of (4.2) at the near-plane (i.e. $z' = -n$) and then maps the projected point $(x', y', -n)$ to a point (x^n, y^n, z^n) in the NDC space. We assume a linear relationship in mapping the range $[l, r]$ to $[-1, 1]$ and $[b, t]$ to $[-1, 1]$. Thus

$$x^n = \frac{1-(-1)}{r-l}x' + c = \frac{2x'}{r-l} + c \tag{4.9}$$

where c is a constant which can be found by boundary conditions. When $x' = r$, we have $x^n = 1$. Substituting this into (4.9), we obtain

$$1 = \frac{2r}{r-l} + c \tag{4.10}$$

From (4.10), we can then solve for c:

$$c = 1 - \frac{2r}{r-l} = -\frac{r+l}{r-l} \tag{4.11}$$

Substituting c into (4.9), we obtain the following:

$$x^n = \frac{2x'}{r-l} - \frac{r+l}{r-l} \tag{4.12}$$

From (4.2), we know that $x' = \frac{-nx}{z}$. Substituting this into (4.12), we obtain

$$x^n = \frac{2nx}{r-l}(\frac{1}{-z}) - \frac{r+l}{r-l} \tag{4.13}$$

We can similarly obtain the equation for y^n:

$$y^n = \frac{2ny}{t-b}(\frac{1}{-z}) - \frac{t+b}{t-b} \tag{4.14}$$

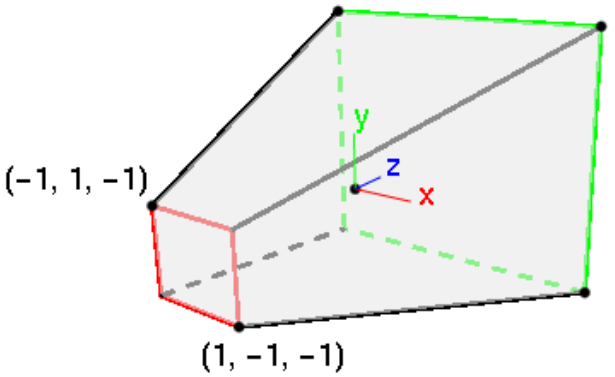


Figure 4-12 View-frustum of Perspective Projection in Normalized Coordinates

Mapping the z-coordinate of a point in the view-frustum to the range $[-1, 1]$ is more complicated. We wish to find a function that maps $-n \rightarrow -1$ and $-f \rightarrow 1$. (Note again that such a mapping reflects the z-axis, resulting in a left-handed coordinate system.) As we have mentioned in the above section, z-coordinates are used for depth information and we have to interpolate their reciprocals in the transformation. Therefore, we construct this mapping as a function of $\frac{1}{z}$, consequently allowing projected depth values to be interpolated linearly. Our mapping function thus has the form

$$z^n = \frac{c}{z} + d \tag{4.15}$$

where c and d are constants to be determined by boundary conditions. Since $z^n = -1$, when $z = -n$, and $z^n = 1$, when $z = -f$, we have

$$\begin{aligned} -1 &= -\frac{c}{n} + d \\ 1 &= -\frac{c}{f} + d \end{aligned} \tag{4.16}$$

Solving for c and d, we obtain

$$\begin{aligned} c &= \frac{2nf}{f-n} \\ d &= \frac{f+n}{f-n} \end{aligned} \tag{4.17}$$

Substituting this back to (4.15), we obtain the mapping function for the z-coordinate:

$$z^n = \frac{2nf}{f-n}(\frac{1}{-z}) + \frac{f+n}{f-n} \tag{4.18}$$

We can rewrite (4.13), (4.14), and (4.18) by multiplying $-z$ to both sides of the equations:

$$\begin{aligned} -zx^n &= \frac{2nx}{r-l} + \frac{r+l}{r-l}z \\ -zy^n &= \frac{2ny}{t-b} + \frac{t+b}{t-b}z \\ -zz^n &= -\frac{2nf}{f-n} - \frac{f+n}{f-n}z = -\frac{f+n}{f-n}z - \frac{2nf}{f-n} \end{aligned} \tag{4.19}$$

We know that a 3D point (x^n, y^n, z^n) when represented in homogeneous coordinates is equivalent to the point $P^n = (-zx^n, -zy^n, -zz^n, -z) = (x^n, y^n, z^n, 1)$. The equations of (4.19) are linear functions of the coordinates x, y, and z. We can therefore represent them using a 4×4 matrix N to calculate the point P^n as follows:

$$P^n = \begin{pmatrix} x^n \\ y^n \\ z^n \\ 1 \end{pmatrix} = \begin{pmatrix} -zx^n \\ -zy^n \\ -zz^n \\ -z \end{pmatrix} = \begin{pmatrix} \frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2ny}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & -\frac{f+n}{f-n} & -\frac{2nf}{f-n} \\ 0 & 0 & -1 & 0 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} \equiv NP \tag{4.20}$$

Therefore, the transformation matrix that transforms a point inside the frustum in the eye coordinate system (i.e. the eye is at the origin looking down the $-z$ axis) to a point in the normalized coordinate system is given by

$$N = \begin{pmatrix} \frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & -\frac{f+n}{f-n} & -\frac{2nf}{f-n} \\ 0 & 0 & -1 & 0 \end{pmatrix} \tag{4.21}$$

Matrix N of (4.21) is nonsingular. Its inverse is given by

$$N^{-1} = \begin{pmatrix} \frac{r-l}{2n} & 0 & \frac{r+l}{2n} & 0 \\ 0 & \frac{t-b}{2n} & \frac{t+b}{2n} & 0 \\ 0 & 0 & 0 & -1 \\ 0 & 0 & -\frac{f-n}{2fn} & -\frac{f+n}{2fn} \end{pmatrix} \tag{4.22}$$

The transformation matrix N given by (4.21) is the OpenGL perspective projection matrix generated by the function **glFrustum**(l, r, b, t, n, f) to do the projection. Points in the eye-coordinate system are transformed by the matrix N into homogeneous clip space in such a way that the w-coordinate holds the negation of the original z-coordinate (i.e. $-z$). We can use this value for depth-test.

The projection matrix N of (4.21) is for a general frustum. If the viewing volume is symmetric, (i.e. $r = -l$, and $t = -b$), then it can be simplified to N_s as follows:

$$N_s = \begin{pmatrix} \frac{n}{r} & 0 & 0 & 0 \\ 0 & \frac{n}{t} & 0 & 0 \\ 0 & 0 & -\frac{f+n}{f-n} & -\frac{2nf}{f-n} \\ 0 & 0 & -1 & 0 \end{pmatrix} \tag{4.23}$$

We can also construct a view frustum that is not bounded by depth by allowing the far-plane distance f to go to infinity. The resulted projection matrix N_{inf} is given by

$$N_{inf} = \lim_{f \to \infty} N = \begin{pmatrix} \frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & -1 & -2n \\ 0 & 0 & -1 & 0 \end{pmatrix} \tag{4.24}$$

The matrix of (4.24) is a valid projection matrix that renders objects at any depth greater than or equal to n. Moreover, a vertex $(x, y, z, 0)$ with w-coordinate value 0 can be rendered correctly; such a vertex lies infinitely far from the viewpoint in the direction of (x, y, z). This is consistent with our earlier discussion that a vector is a special point lying at infinity.

glFrustum and gluPerspective

In OpenGL, there are two popular ways to define a view-volume of perspective projection. One way is to use the function **glFrustum**():

> void **glFrustum**(double l, double r, double b, double t, double n, double f);
> Creates a matrix for a perspective-view frustum and multiplies the current matrix by it. The frustum's viewing volume is defined by the parameters $(l, b, -n)$ and $(r, t, -n)$; they specify the (x, y, z) coordinates of the lower-left and upper-right corners of the near clipping plane; n and f give the distances from the viewpoint to the near and far clipping planes. They should always be positive.

This command specifies a view-frustum like that shown in Figure 4-8 above and creates the transformation matrix given by (4.21).

Another way to define a viewing volume is to use the function **gluPerspective**(). As shown in Figure 4-13. We need to provide the following information to this function:

1. Specify the angle of the field of view (fov) in the y-direction. **Field of view** or vision (fov) is the extent of the observable world that the viewer can see; it determines how much of the world is taken into the picture. A larger field of view implies a smaller object projection size.
2. Specify the aspect ratio of width to height (w/h) of the projection plane.
3. Specify the distance n from the viewpoint to the near-plane (projection plane) and the distance f from the viewpoint to the far-plane.

In Figure 4-13, the viewpoint is the origin O, which is the point where the camera is located. The angle a is the angle of the field of view in the y-direction. w is the width of the near-plane and h is its height. The function **gluPerspective**() provides an alternative way to define the view-volume and works as follows.

void **gluPerspective**(GLdouble $fovy$, GLdouble $aspect$, GLdouble $near$, GLdouble far); Creates a matrix for a symmetric perspective-view frustum and multiplies the current matrix by it. $fovy$ is the angle of the field of view in the $y-z$ plane; its value must be in the range $[0.0, 180.0]$. $aspect$ is the aspect ratio of the frustum, its width divided by its height. $near$ and far values are the distances between the viewpoint and the clipping planes, along the negative z-axis. They should be always positive.

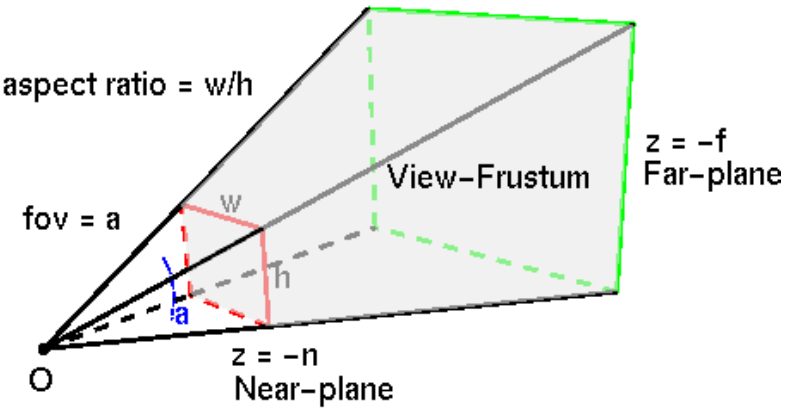


Figure 4-13 Setup for gluPerspective()

Example 4-1

Suppose $t = r = 1$, $b = l = -1$, $n = 2$, and $f = 10$, corresponding to the command " glFrustum (-1.0, 1.0, -1.0, 1.0, 2.0, 10.0);". Then the NDC perspective projection matrix N can be calculated from (4.23) and is equal to:

$$N = \begin{pmatrix} \frac{2}{1} & 0 & 0 & 0 \\ 0 & \frac{2}{1} & 0 & 0 \\ 0 & 0 & -\frac{10+2}{10-2} & -\frac{2\times 2\times 10}{10-2} \\ 0 & 0 & -1 & 0 \end{pmatrix} = \begin{pmatrix} 2 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & -1.5 & -5 \\ 0 & 0 & -1 & 0 \end{pmatrix} \tag{4.25}$$

If the viewpoint is located at $(0, 0, 5)$ and the **up** vector is pointing at the y direction corresponding to the command "gluLookAt (0.0, 0.0, 5.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0);", the composite projection P matrix can be found by multiplying N by the appropriate affine transformation matrix:

$$P = \begin{pmatrix} 2 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & -1.5 & -5 \\ 0 & 0 & -1 & 0 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & -5 \\ 0 & 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 2 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & -1.5 & 2.5 \\ 0 & 0 & -1 & 5 \end{pmatrix} \tag{4.26}$$

We can verify this using the following piece of OpenGL code.

```
//Use template so that we can print double, float or int
template<class T>
void print_mat ( T m[][4] )      //print 4 x 4 matrix
{
  cout.precision ( 2 );
  cout << fixed;                 //fixed-point format
```

```
  for ( int i = 0; i < 4; ++i ) {
    cout << "\t";
    for ( int j = 0; j < 4; ++j )
      cout <<  m[j][i] << "\t";  //OpenGL is column-major
    cout << endl;
  }
  cout << endl;
}

void display(void)
{
   float p[4][4];                //4 x 4 matrix

   glMatrixMode (GL_PROJECTION); //Current matrix is projection
   glLoadIdentity ();            //set matrix to identity
   glFrustum (-1.0,1.0,-1.0,1.0,2.0,10.0); //set view-volume
   glGetFloatv(GL_PROJECTION_MATRIX,&p[0][0]);
   cout << "NDC Perspective Projection Matrix:" << endl;
   print_mat ( p );              //print matrix
   gluLookAt (0.0, 0.0, 5.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0);
   glGetFloatv(GL_PROJECTION_MATRIX,&p[0][0]);
   cout << "Perspective Projection Matrix:" << endl;
   print_mat ( p );
}
```

In the code, the word **template** in front of the function **print_mat**() is a C++ keyword. It allows the function to handle various data types such as float, double or int; the data type used is determined by the calling function and in our example, data type float is used. The following is the output of the above code which is consistent with our calculations of (4.25) and (4.26).

```
NDC Perspective Projection Matrix:
        2.00    0.00    0.00    -0.00
        0.00    2.00    0.00    -0.00
        0.00    0.00    -1.50   -5.00
        0.00    0.00    -1.00   -0.00

Perspective Projection Matrix:
        2.00    0.00    0.00    0.00
        0.00    2.00    0.00    0.00
        0.00    0.00    -1.50   2.50
        0.00    0.00    -1.00   5.00
```

Example 4-2

In designing our graphics scene, we need to estimate the field of view if we use **gluPerspective**() to define the view-frustum. Suppose the field of view is θ and the largest object size is about S, which is at a distance d from the viewpoint. Then we can estimate the field of view as follows.

$$tan\frac{\theta}{2} = \frac{S/2}{d} \tag{4.27}$$

This implies that

$$\theta = 2tan^{-1}\frac{S}{2d} \tag{4.28}$$

The following piece of code segment shows how to calculate this angle in degree, which can be used for the $fovy$ parameter of **gluPerspective**().

```
const double PI = 3.14159265389;

double fov(double size, double distance)
```

```
{
    double angle_rad, angle_deg;

    angle_rad = 2.0 * atan2 (size/2.0, distance);
    angle_deg = ( 180.0 * angle_rad ) / PI;
    return ( angle_deg );
}
```

4.3 Orthographic Projection

Orthographic projection, also known as **orthogonal projection**, is a form of parallel projection, where all the projection lines are orthogonal (perpendicular) to the projection plane. Most often we use orthographic projection to produce the front, side, and top views of an object as shown in Figure 4-14. Front, side, and rear orthographic projections of an object are referred to as *elevations*. A top orthographic projection is called a *plane view*.

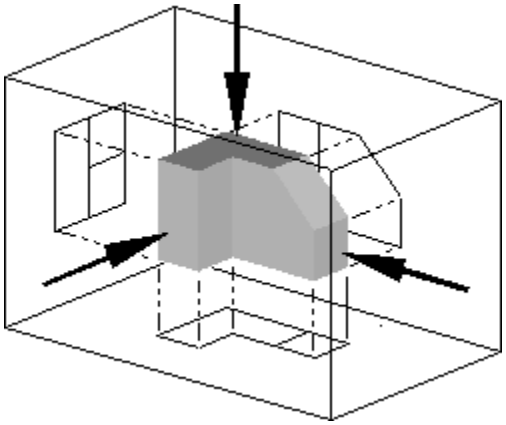


Figure 4-14 Front, Side and Top Views of an Object

With an orthographic projection, the viewing volume is a rectangular parallelepiped (box) as shown in Figure 4-15.

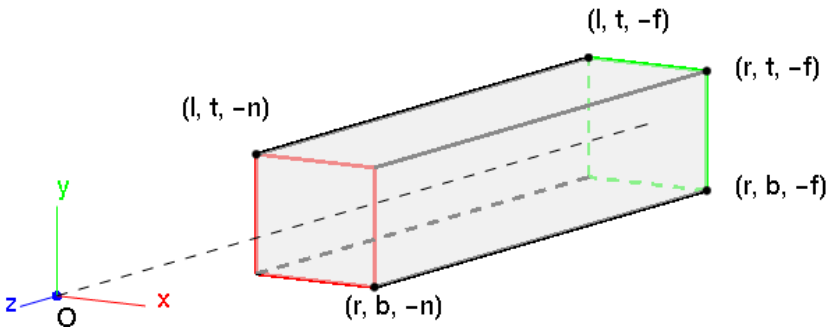


Figure 4-15 Viewing Volume of Orthographic Projection

Unlike perspective projection, the size of the planes bounding the volume does not change from one end to the other. Therefore, the distance of the viewpoint from the viewing volume does not affect how large an object appears. This type of projection is commonly used for engineering and architectural drawings because lengths and angles are accurately depicted and can be measured from the drawings.

The OpenGL function **glOrtho**() creates an orthographic parallel viewing volume. As with **glFrustum**(), we specify the corners of the near clipping plane and the distance to the far clipping plane (Figure 4-15):

> void **glOrtho**(double l, double r, double b, double t, double n, double f);
> Creates a matrix for orthographic projection and multiplies the current matrix by it. $(l, b, -n)$ and $(r, t, -n)$ are points on the near clipping plane that are mapped to the lower-left and upper-right corners of the viewport window, respectively. $(l, b, -f)$ and $(r, t, -f)$ are points on the far clipping plane that are mapped to the same respective corners of the viewport. Both n and f can be positive or negative.

Constructing the GL_PROJECTION matrix for orthographic projection is a lot simpler than for perspective projection. In this case, we map all x, y and z coordinates in eye space linearly to NDC. We just need to scale a rectangular volume to a cube, then move it to the origin. As in the case of perspective projection, we want to find the transformation that projects a point (x, y, z) into $(x', y'z')$ at the near-plane (i.e. $z' = -n$) and then maps the projected point $(x', y', -n)$ to a point (x^n, y^n, z^n) in the NDC space. Since there is no perspective distortion, we can interpolate depth values in an orthographic projection linearly. Thus, our mapping to normalized coordinates can be performed linearly in all three axes. For x and y coordinates, we have a linear relationship in mapping the range $[l, r]$ to $[-1, 1]$ and $[b, t]$ to $[-1, 1]$, which are given by

$$x^n = \frac{2x}{r-l} - \frac{r+l}{r-l} \tag{4.29}$$

and

$$y^n = \frac{2y}{t-b} - \frac{t+b}{t-b} \tag{4.30}$$

Similarly, by negating z so that $-n \rightarrow -1$ and $-f \rightarrow 1$, we can show that the function that maps the z coordinate from the range $[-f, -n]$ to the range $[-1, 1]$ is given by

$$z^n = \frac{-2z}{f-n} - \frac{f+n}{f-n} \tag{4.31}$$

Again, we can express this in matrix from. Suppose the orthograhic matrix O transforms a point $P = (x, y, z, 1)$ into normalized coordinates $P^n = (x^n, y^n, z^n, 1)$, then

$$P^n = OP \tag{4.32}$$

where

$$O = \begin{pmatrix} \frac{2}{r-l} & 0 & 0 & -\frac{r+l}{r-l} \\ 0 & \frac{2}{t-b} & 0 & -\frac{t+b}{t-b} \\ 0 & 0 & \frac{-2}{f-n} & -\frac{f+n}{f-n} \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{4.33}$$

The inverse of O is given by

$$O^{-1} = \begin{pmatrix} \frac{r-1}{2} & 0 & 0 & \frac{r+l}{2} \\ 0 & \frac{t-b}{2} & 0 & \frac{t+b}{2} \\ 0 & 0 & \frac{f-n}{-2} & -\frac{f+n}{2} \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{4.34}$$

The orthographic projection matrix O can be further simplified to O_s if the viewing volume is symmetric with $r = -l$, and $t = -b$.

$$O_s = \begin{pmatrix} \frac{1}{r} & 0 & 0 & 0 \\ 0 & \frac{1}{t} & 0 & 0 \\ 0 & 0 & \frac{-2}{f-n} & -\frac{f+n}{f-n} \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{4.35}$$

The matrix O of (4.33) is the orthographic projection matrix generated by the OpenGL function **glOrtho**(l, r, b, t, n, f). Note that the w coordinate of the point remains 1 after the transformation, and therefore no perspective projection has taken place.

Example 4-3

Suppose $t = r = 1$, $b = l = -1$, $n = 2$, and $f = 10$, corresponding to the command " glOrtho(-1.0, 1.0, -1.0, 1.0, 2.0, 10.0);". Then the NDC orthographic projection matrix O can be calculated from (4.33) and is equal to:

$$O = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -0.25 & -1.5 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{4.36}$$

If the viewpoint is located at $(0, 0, 5)$ and the **up** vector is pointing at the y direction corresponding to the command "gluLookAt (0.0, 0.0, 5.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0);", the composite projection P matrix can be found by multiplying N by the appropriate affine transformation matrix:

$$P = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -0.25 & -1.5 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & -5 \\ 0 & 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -0.25 & -0.25 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{4.37}$$

We can verify this with the following OpenGL code. The function **print_mat**() has been presented in Example 4-1.

```
void display(void)
{
   float p[4][4];

   glMatrixMode (GL_PROJECTION);  //Current matrix is for projection
   glLoadIdentity ();             // clear the matrix
   glOrtho (-1.0, 1.0, -1.0, 1.0, 2.0, 10.0);
   glGetFloatv(GL_PROJECTION_MATRIX,&p[0][0]);
   cout << "NDC Orthographic Projection Matrix:" << endl;
   print_mat ( p );
   gluLookAt (0.0, 0.0, 5.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0);
   glGetFloatv(GL_PROJECTION_MATRIX,&p[0][0]);
   cout << "Orthographic Projection Matrix:" << endl;
   print_mat ( p );
}
```

The following is the output of the code, which is consistent with our calculations in (4.36) and (4.37).

```
NDC Orthographic Projection Matrix:
        1.00    0.00    -0.00   0.00
        0.00    1.00    -0.00   0.00
        0.00    0.00    -0.25   -1.50
        0.00    0.00    0.00    1.00

Orthographic Projection Matrix:
        1.00    0.00    0.00    0.00
        0.00    1.00    0.00    0.00
        0.00    0.00    -0.25   -0.25
        0.00    0.00    0.00    1.00
```

In general, the 3D to 2D projection is a many-to-one mapping and is not reversible. However, if we specifiy a z value, we can map a projected 2D point (x', y') back to a 3D point (x, y, z). OpenGL provides the function **gluUnproject**() to achieve this purpose.

4.4 Rasterization

After the transformation to NDC, the graphics data will go through a process called **rasterization** or **scan conversion**, which converts the graphics objects to pixel data. The piece of code that performs the conversion is referred to as **rasterizer**. The output of a rasterizer is a set of **fragments** for each graphics primitive. Pixel values for displayed are saved in a framebuffer. We can think of a fragment as a set of data consisting of information such as color and depth value to form a potential pixel; the information will be used to update the corresponding pixel in the frame buffer. Fragments usually contain depth information that can be used to determine whether a particular fragment lies behind or in front of other previously rasterized fragments for a given pixel; typically, we discard the fragment if it lies behind otherwise we overwrite the one in the framebuffer with the new one.

The rasterizer takes vertices in normalized coordinates as inputs and outputs fragments in **window coordinates** or **screen coordinates**, units of the display device. The projection of the clipping volume must be in the assigned viewport. Suppose the viewport is bounded by x_{vmin} and x_{vmax} in the x direction, and by y_{vmin} and y_{vmax} in the y direction. Then the transformation of the normalized coordinates (x^n, y^n) to the screen coordinates (x^s, y^s) can be easily calculated. The x^s coordinate is given by

$$x^s = x_{vmin} + \frac{x^n - (-1)}{1 - (-1)}(x_{vmax} - x_{vmin})$$

which can be simplified to

$$x^s = x_{vmin} + \frac{x^n + 1}{2}(x_{vmax} - x_{vmin}) \tag{4.38}$$

Similarly,

$$y^s = y_{vmin} + \frac{y^n + 1}{2}(y_{vmax} - y_{vmin}) \tag{4.39}$$

Note that the z coordinates are scaled nonlinearly in perspective projection. However, their original depth order is preserved and thus we can use them for depth test or hidden surface removal.