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

Fore June

Chapter 3 Modeling and Viewing Transformations

3.1 Viewing

Any graphical object that we create in 3D space will be transformed and projected on a two-dimensional screen. OpenGL makes use of three computer operations, namely, transformations , clipping, and viewport mapping to convert an object's three-dimensional coordinates to pixel positions on the screen. All the operations are carried out by matrix operations.

A **transformation** is represented by a matrix multiplication; it includes modeling, viewing, and projection operations. Rotation, translation, reflection, scaling, orthographic projection, and perspective projection are examples of transformation operations. A **modelview transformation** is a transformation which consists of modeling and positioning of the view point (camera or eye position). This includes rotation, translation, and scaling. A **projection transformation** is a transformation that involves the projection of 3D scene on a 2D plane.

Since the scene is rendered on a rectangular window, lines or objects (or parts of objects) that lie outside the window will not be rendered. In practice, we are only interested to see objects within a certain space that we define. The defined space is referred to as **viewing volume**. **Clipping** is the process of removing the parts of the objects that lie outside the viewing volume. All the points in space are transformed accordingly in a viewing transformation. Therefore, points that lie outside a viewing volume will be still outside after such a transformation. Consequently, the operation is more efficient if we remove those points before the transformation. We can carry out clipping either in three-dimensional space, or in projected 2D space.

We discussed in Chapter 2 that a viewport is a rectangular region on a display device that objects are drawn or rendered and is described in pixels. On the other hand, objects are drawn in world coordinates and the units or scales can be anything in the real world. To display the objects properly in the viewports, a correspondence must be established between the transformed world coordinates of the objects and the screen pixels of the viewport; this process is referred to as **viewport mapping**.

Figure 3-1 summarizes these operations. The figure shows that graphics rendering is like a manufacturing assembly line with each stage adding something to the previous one. This rendering architecture is referred to as **pipeline** architecture, which is supported by OpenGL.

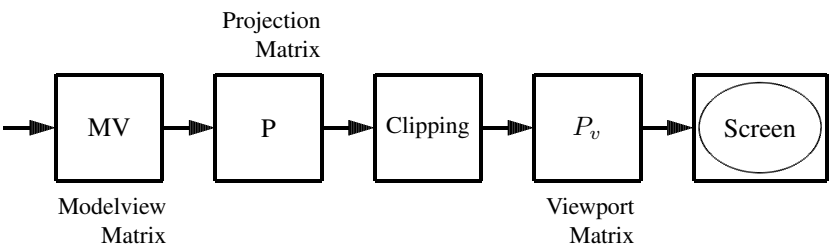

Figure 3-1 Graphics Pipeline

In traditional OpenGL programming, a fixed-function pipeline architecture is used. That is, the operations of the functions in each stage of Figure 3-1 are fixed. There is a trend to

replace this fixed-function architecture with a programmable one. The OpenGL Shading Language (glsl) is introduced for this purpose. OpenGL for Embedded Systems (OpenGL ES), a subset of the OpenGL API designed for embedded systems and maintained by the nonprofit technology consortium, the Khronos Group, Inc., has eliminated most of the fixed-function rendering pipeline in favor of a programmable one in version 2.0 released in March 2007.

3.2 Points and Vectors

We discuss in this section points and geometric vectors intuitively. Later in this chapter, we will have a more formal, rigid and mathematical discussion of this topic. We know that both points and geometric vectors can be represented by (x, y, z). Given a 3-tuple (x, y, z), we really cannot tell whether it represents a point or a vector. At the surface, points and vectors appear to be the same but in reality, they are very different. A **point** denotes a position or a location; it does not have any direction or magnitude. On the other hand, a **vector** specifies a direction rather than a location; it has a magnitude and directional components. It makes sense to add two vectors but it does **not** make any sense to add two points. In some situations, a vector may be considered as a special point located at infinity. A common way to distinguish between a point and a vector denoted by (x, y, z) is to introduce an additional component, usually expressed as 'w', with '$w = 1$' denoting a point and '$w = 0$' denoting a vector. Therefore, in two-dimensional space, $(x, y, 1)$ represents a point and $(x, y, 0)$ represents a vector. If we consider three-dimensional situations, $(x, y, z, 1)$ represents a point and $(x, y, z, 0)$ represents a vector. With this representation, the operations on points and vectors are consistent with our intuition or understanding of points and vectors. For example, a point $(x_1, y_1, z_1, 1)$ plus a vector $(x_2, y_2, z_2, 0)$ is a point $(x_1 + x_2, y_1 + y_2, z_1 + z_2, 1)$, and a point $(x_1, y_1, z_1, 1)$ minus a point $(x_2, y_2, z_2, 1)$ is a vector $(x_1 - x_2, y_1 - y_2, z_1 - z_2, 0)$. A point $(x_1, y_1, z_1, 1)$ plus another point $(x_2, y_2, z_2, 1)$ gives $(x_1 + x_2, y_1 + y_2, z_1 + z_2, 2)$, which is an invalid representation and therefore, adding two points is an illegal operation. The operations may be summarized in the following table:

Table 3-1

Operation			Result
Vector	+	Vector	Vector
Vector	-	Vector	Vector
Vector	+	Point	Point
Vector	-	Point	Illegal
Point	+	Point	Illegal
Point	-	Point	Vector
Point	+	Vector	Point
Point	-	Vector	Point

Though it is illegal to add two points, we may form linear combinations of points:

$$P = c_0P_0 + c_1P_1 + ... + c_{n-1}P_{n-1} \tag{3.1}$$

where $P_i = (x_i, y_i, z_i, 1)$ is a point and c_i's are constant coefficients. The combination is legitimate if the summing coefficients are summed up to 1, and the combination gives a valid new point. i.e.,

$$c_0 + c_1 + ... + c_{n-1} = 1 \tag{3.2}$$

Linear combination of points satisfying (3.2) is in general referred to as affine combination of points. (The word "affine" has the Latin root "affinis" meaning "connected with"; "finis" means border or end, and "af" means sharing a common boundary.) This is the principle behind point interpolation and extrapolation. To construct a valid interpolation or extrapolation of points, the combination must be affine.

3.3 Vector, Affine, and Projective Spaces

Graphical scenes and objects are defined by points and vectors. Therefore, when we manipulate a graphical object, we basically process points and vectors. We have introduced the concepts of points and vectors in the previous section. It is important for us to have a deeper understanding of vectors and points in order to process our objects properly.

3.3.1 Vector Space

A **vector space** V is a set of objects called vectors that is closed under finite vector addition and scalar multiplication. That is, addition of two vectors yields another vector in the set, and multiplication of a vector by a scalar also produces a vector in the set. For a vector space to be valid, the following axioms must be satisfied for every vector $\mathbf{u}, \mathbf{v}, \mathbf{w}$ in V and every scalar a and b.

1. $(\mathbf{u} + \mathbf{v}) \in V$ (V is closed under addition)
2. $\mathbf{u} + \mathbf{v} = \mathbf{v} + \mathbf{u}$ (commutative property)
3. $(\mathbf{u} + \mathbf{v}) + \mathbf{w} = \mathbf{u} + (\mathbf{v} + \mathbf{w})$ (associative property)
4. There exists a vector in V, called the zero vector and denoted $\mathbf{0}$ such that $\mathbf{u} + \mathbf{0} = \mathbf{u}$ (additive identity)
5. For every vector $\mathbf{u}$ in V, there exists a vector $-\mathbf{u}$ also in V such that $\mathbf{u} + (-\mathbf{u}) = 0$ (additive inverse)
6. $a\mathbf{u} \in V$ (V is closed under scalar multiplication)
7. $a(\mathbf{u} + \mathbf{v}) = a\mathbf{u} + a\mathbf{v}$
8. $(a + b)\mathbf{u} = a\mathbf{u} + b\mathbf{u}$
9. $a(b\mathbf{u}) = (ab)\mathbf{u}$
10. $1\mathbf{u} = \mathbf{u}$

We specify both a magnitude and a direction for a geometric vector quantity. On the other hand, we specify a scalar quantity with just a number. Any number of vector quantities of the same type (i.e., same units) can be combined by basic vector operations. Figure 3-2 shows the vector addition and scalar multiplication properties.

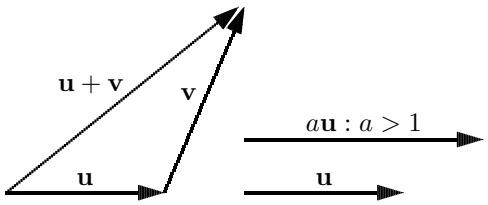

Figure 3-2 Vectors

Actually, there are many sets that satisfy the above properties. For example, polynomials of degree n form an n-dimensional vector space. Here, we are only concerned with geometric vectors which obviously satisfy the definition.

3.3.2 Affine Space

We define an n-dimensional **affine space** as a set of points, which associates with an n-dimensional vector space, and two operations, subtraction of two points in the set and addition of a point in the set to a vector in the associated vector space. In an affine space, we can subtract one point from another to get a vector, or add a vector to a point to get another point, but we cannot add points together directly. Also, there is no particular point which serves as the 'origin' of all the points. The solution set of an inhomogeneous linear equation is either empty or an affine space. Figure 3-3 shows the subtraction and addition operations; in the figure, P and Q are points and **u** is a vector.

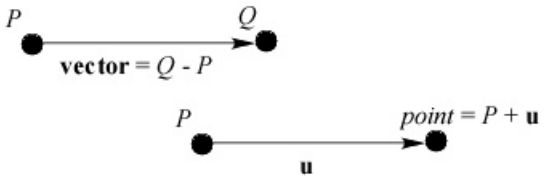

Figure 3-3 Affine Space Operations

3.3.3 Projective Space and Projective Equivalence

From an n-dimensional affine space, we can define an $(n+1)$-dimensional projective space, which embeds the points of the n-dimensional affine space. We denote the extra coordinate dimension as w and say that the affine points lie in the $w = 1$ plane of the projective space.

Figure 3-4 illustrates the $n = 2$ case. The xy plane is the affine 2D plane which is the $w = 1$ plane of a 3D projective space. (The image is copied from *http://www.ccs.neu.edu*.)

All projective space points on the line from the projective space origin through an affine point on the $w = 1$ plane are said to be projectively equivalent to the affine space point. Therefore, in a 4D projective space, the following points are projectively equivalent,

$$\begin{pmatrix} 1 \\ 2 \\ 3 \\ 1 \end{pmatrix}, \quad \begin{pmatrix} 2 \\ 4 \\ 6 \\ 2 \end{pmatrix}, \quad \begin{pmatrix} 0.5 \\ 1.0 \\ 1.5 \\ 0.5 \end{pmatrix}, \quad \begin{pmatrix} -1 \\ -2 \\ -3 \\ -1 \end{pmatrix}$$

and are "projected" into the same affine space point,

$$\begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix}$$

In Figure 3-4, all points on a line that is not parallel to the xy plane are projectively equivalent to the point symbolized by the dot on the affine (i.e. $w = 1$) plane. Basically, for any nonzero α, all the 4D projective space points in the form

$$\begin{pmatrix} \alpha x \\ \alpha y \\ \alpha z \\ \alpha \end{pmatrix}$$

are projectively equivalent to the 3D affine point

$$\begin{pmatrix} x \\ y \\ z \end{pmatrix}$$

We may use the symbol $\sim$ to denote the "projectively equivalent" relation:

$$\begin{pmatrix} \alpha x \\ \alpha y \\ \alpha z \\ \alpha \end{pmatrix} \sim \begin{pmatrix} x \\ y \\ z \end{pmatrix} \tag{3.3}$$

Therefore, to find the affine point that is equivalent to a given projective space point, we can simply divide the first three coordinates by the fourth one. *Then what would happen if the fourth component is zero*? *What is the significance of the projective space point*

$$\begin{pmatrix} x \\ y \\ z \\ 0 \end{pmatrix}?$$

It is obvious that a line from the projective space origin through this point is parallel to the $w = 1$ plane implying that it does not intersect the $w = 1$ plane, or we can imagine that they meet at infinity. It turns out that we can interpret this point as the 3D vector

$$\begin{pmatrix} x \\ y \\ z \end{pmatrix}$$

as we have discussed in the previous section. We can interpret vectors as special points at infinity of the $w = 1$ plane of a projective space.

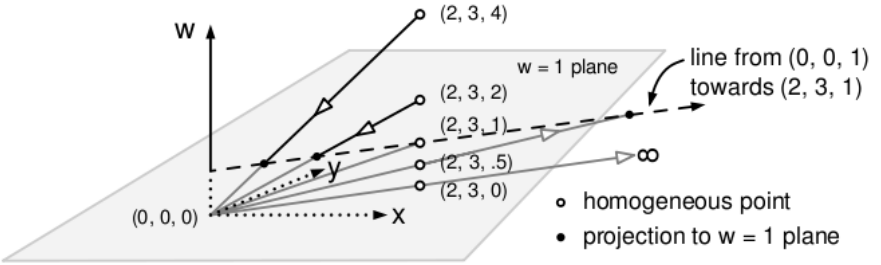

Figure 3-4 Projective Space

The idea of a projective space relates to perspective projection. It is the way that an eye or a camera projects a 3D scene into a 2D image. All points lying on a projection line (i.e., a "line-of-sight"), intersecting with the focal point of the camera, are projected onto a common image point. We can conveniently represent perspective projections in projective space. This means that we can represent them using the same formalism as ordinary transformations such as rotations, translations, and scales. This greatly eases the task of integrating perspective transformations with other transformations, which would not otherwise be true if we restrict ourselves to affine space and their associated operations.

We can study projective spaces as an independent field in mathematics, but we can also apply them to various fields, in particular geometry. We can represent geometric objects, such as points, lines, or planes, as elements in projective spaces using **homogeneous coordinates** discussed below. Consequently, we can describe various relations among these objects in a simpler way as compared to the case without using homogeneous coordinates. Moreover, we can make more consistent statements in geometry without making exceptions. For example, in standard geometry two lines always intersect at a point except when the lines are parallel. However, when we represent lines and points in a projective space, such an intersection point exists even for parallel lines, and we can compute it in the same way as other intersection points.

3.4 Homogeneous coordinates

Homogeneous coordinates are a system of coordinates used in projective geometry in a way that Cartesian coordinates are used in Euclidean geometry. In 3D graphics, a point P in homogeneous coordinates is a point in the corresponding projective space and is represented by four coordinates in the form

$$P = \begin{pmatrix} x \\ y \\ z \\ w \end{pmatrix}$$

We may sometimes simply express it as $P = (x, y, z, w)$. We have learned that a point (x, y, z) in an affine space is mapped to many points in a projective space in the form $(\alpha x, \alpha y, \alpha z)$ with nonzero α. Conversely, a projective space embeds an affine space at the

$w = 1$ plane. Therefore, a 3D affine point is given by

$$P = \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} \tag{3.4}$$

and a 3D vector $\mathbf{v}$ is represented by

$$\mathbf{v} = \begin{pmatrix} x \\ y \\ z \\ 0 \end{pmatrix} \tag{3.5}$$

For a "point" to be a valid affine point, the fourth component needs to have a value of 1 and for it to be a valid vector, the fourth component's value is 0.

Because of the restriction on the value of the fourth component of 3D points and vectors, we can always form linear combinations of 3D vectors, but we can form legitimate linear combinations of 3D points only if it satisfies the condition that the sum of the combination coefficients is equal to 0 or 1. That is,

$$\sum_{i=0}^{n-1} c_i P_i = \begin{cases} \text{point} & \text{if} \quad \sum_{i=0}^{n-1} c_i = 1 \\ \textbf{vector} & \text{if} \quad \sum_{i=0}^{n-1} c_i = 0 \\ \text{invalid} & \text{otherwise} \end{cases} \tag{3.6}$$

The advantage of using homogeneous coordinates is that we can use finite coordinates to represent all the coordinates of points, including points at infinity. Formulas involving homogeneous coordinates are often simpler and more symmetric than their Cartesian counterparts. Homogeneous coordinates are particularly convenient when we use them to represent points and vectors in 3D graphics as all the affine transformations, and projective transformations can be represented by matrices that have the same form. In general, such a transformation is represented by a 4×4 matrix M that has the following form:

$$M = \begin{pmatrix} m_{00} & m_{01} & m_{02} & m_{03} \\ m_{10} & m_{11} & m_{12} & m_{13} \\ m_{20} & m_{21} & m_{22} & m_{23} \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{3.7}$$

In projective space, an affine point is embedded in the $w = 1$ plane, and we describe an affine transformation in that projective space. The property that the bottom row of the transformation matrix M is $(0, 0, 0, 1)$ guarantees that transformed points are in the same affine space. That is, the fourth coordinate of the transformed point will be always 1. This can be easily verified by carrying out the matrix multiplication directly:

$$P' = MP = \begin{pmatrix} m_{00} & m_{01} & m_{02} & m_{03} \\ m_{10} & m_{11} & m_{12} & m_{13} \\ m_{20} & m_{21} & m_{22} & m_{23} \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix} \tag{3.8}$$

The 4×4 matrix M of (3.7) can be used to represent both affine transformations and projection transformations. This is the main reason that virtually all graphics APIs, including OpenGL, employ 4×4 matrices to represent all transformations and why most graphics hardware includes support for 4×4 matrix multiplication.

OpenGL uses 4-tuples to represent homogeneous coordinates for 3D space points extensively. For example, the function call **glVectex4f**(x, y, z, w) specifies a point (x, y, z, w) in homogeneous coordinates. If we specify a point with only three (nonhomogeneous) coordinates, such as a function call to **glVectex3f**(x, y, z), OpenGL translates it to $(x, y, z, 1)$.

3.5 General Transformations

In general, an invertible $n \times n$ matrix M represents a transformation from one coordinate system to another. The columns of M give the images to which the principal axes of the original system are mapped in the new coordinate system. For example, when the 4×4 matrix M of (3.7) operates on the vectors $(1, 0, 0, 0)$, $(0, 1, 0, 0)$, and $(0, 0, 1, 0)$, we have

$$\begin{pmatrix} m_{00} & m_{01} & m_{02} & m_{03} \\ m_{10} & m_{11} & m_{12} & m_{13} \\ m_{20} & m_{21} & m_{22} & m_{23} \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} m_{00} \\ m_{10} \\ m_{20} \\ 0 \end{pmatrix} \tag{3.9a}$$

$$\begin{pmatrix} m_{00} & m_{01} & m_{02} & m_{03} \\ m_{10} & m_{11} & m_{12} & m_{13} \\ m_{20} & m_{21} & m_{22} & m_{23} \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} m_{01} \\ m_{11} \\ m_{21} \\ 0 \end{pmatrix} \tag{3.9b}$$

$$\begin{pmatrix} m_{00} & m_{01} & m_{02} & m_{03} \\ m_{10} & m_{11} & m_{12} & m_{13} \\ m_{20} & m_{21} & m_{22} & m_{23} \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \end{pmatrix} = \begin{pmatrix} m_{02} \\ m_{12} \\ m_{22} \\ 0 \end{pmatrix} \tag{3.9c}$$

Conversely, the columns of M^{-1} give the images to which the principal axes of the new coordinate system are mapped in the original coordinate system. Therefore, given any arbitrary independent vectors **u, v**, and **w**, we can construct a transformation matrix which maps these vectors to the vectors $(1, 0, 0, 0)$, $(0, 1, 0, 0)$, and $(0, 0, 1, 0)$. From (3.9), if **u, v**, and **w** form the columns of the inverse of the transformation matrix, we have

$$\begin{pmatrix} u_x & v_x & w_x & 0 \\ u_y & v_y & w_y & 0 \\ u_z & v_z & w_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}^{-1} \begin{pmatrix} u_x \\ u_y \\ u_z \\ 0 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix} \tag{3.10a}$$

$$\begin{pmatrix} u_x & v_x & w_x & 0 \\ u_y & v_y & w_y & 0 \\ u_z & v_z & w_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}^{-1} \begin{pmatrix} v_x \\ v_y \\ v_z \\ 0 \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \end{pmatrix} \tag{3.10b}$$

$$\begin{pmatrix} u_x & v_x & w_x & 0 \\ u_y & v_y & w_y & 0 \\ u_z & v_z & w_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}^{-1} \begin{pmatrix} w_x \\ w_y \\ w_z \\ 0 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \end{pmatrix} \tag{3.10c}$$

In representing transformations using matrices, we need to keep in mind a few matrix multiplication properties. Suppose A, B, and C are matrices that we can carry out multiplications. The following are some basic but important properties of matrix operations.

Noncommutative multiplication:	$AB \neq BA$
Associative:	$(AB)C = A(BC)$
Transpose Property:	$(AB)^T = B^T A^T$
Inverse Property:	$(AB)^{-1} = B^{-1}A^{-1}$

We can concatenate multiple transformations and represent the resultant transformation by a single matrix. For example,

$$M\mathbf{V} = M_1 M_2 M_3 \mathbf{V} \tag{3.11}$$

So the composite matrix is

$$M = M_1 M_2 M_3 \tag{3.12}$$

3.5.1 Orthogonal Matrix

We define an invertible $n \times n$ matrix M to be **orthogonal** if and only if its inverse is equal to its transpose (i.e. $M^{-1} = M^T$). Orthogonal matrices have a few interesting properties that we want to study. First, with this definition, we can prove the following theorem.

Theorem 3.1
If the vectors $\mathbf{v_0}, \mathbf{v_1}, ..., \mathbf{v_{n-1}}$ form an orthonormal set, then the $n \times n$ matrix M constructed by setting the j-th column equal to v_j for all $0 \leq j < n$ is orthogonal.

Proof
Since v_j's are orthonormal, $(M^T M)_{ij} = \mathbf{v_i} \cdot \mathbf{v_j} = \delta_{ij}$ where δ_{ij} is the Kronecker delta symbol which is defined as

$$\delta_{ij} = \begin{cases} 0 & \text{if } i \neq j \\ 1 & \text{if } i = j \end{cases}$$

So $M^T M = I$. Therefore, $M^T = M^{-1}$ and according to our definition, M is orthogonal.

Another property of orthogonal matrix relates to length and angle preservation. We say that a matrix M **preserves length** if for any vector $\mathbf{V}$ we have

$$|M\mathbf{V}| = |\mathbf{V}| \tag{3.13}$$

A matrix M that preserves lengths also **preserves angles** if for any two vectors $\mathbf{U}$ and $\mathbf{V}$, we have

$$(M\mathbf{U}) \cdot (M\mathbf{V}) = \mathbf{U} \cdot \mathbf{V} \tag{3.14}$$

Now we prove the following theorem.

Theorem 3.2
If an $n \times n$ matrix M is orthogonal, then M preserves lengths and angles.

Proof
Let M be orthogonal and $\mathbf{U}, \mathbf{V}$ are two vectors. We use the conventional notation that a vector $\mathbf{U}$ is a column matrix and its transpose $\mathbf{U}^T$ is a row matrix. The dot product of any two vectors is equal to the product of the corresponding row matrix and column matrix. For example, the dot product of two vectors **u** and **v** is given by:

$$\mathbf{u} \cdot \mathbf{v} = \mathbf{u}^T\mathbf{v} = (u_x, u_y, u_z, 0) \begin{pmatrix} v_x \\ v_y \\ v_z \\ 0 \end{pmatrix} = u_x v_x + u_y v_y + u_z v_z$$

Therefore,

$$\begin{aligned} (M\mathbf{U}) \cdot (M\mathbf{V}) &= (M\mathbf{U})^T (M\mathbf{V}) \\ &= (\mathbf{U}^T M^T)(M\mathbf{V}) \\ &= \mathbf{U}^T M^T M\mathbf{V} \\ &= \mathbf{U}^T\mathbf{V} \\ &= \mathbf{U} \cdot \mathbf{V} \end{aligned}$$

where we have made use of the transpose property of matrix (i.e. $(AB)^T = B^T A^T$), and the orthogonal property of M (i.e. $M^T M = M^{-1} M = I$). Therefore,

$$(M\mathbf{U}) \cdot (M\mathbf{V}) = \mathbf{U} \cdot \mathbf{V} \tag{3.15}$$

implying that the matrix M preserves angles.

Moreover, when $\mathbf{U} = \mathbf{V}$, (3.15) is reduced to

$$|M\mathbf{U}|^2 = |\mathbf{U}|^2 \tag{3.16}$$

which implies that $|M\mathbf{U}| = |\mathbf{U}|$ and thus the length is preserved.

For example, the following scaling matrix represents a reflection about the yz plane (i.e. $x \to -x$).

$$S = \begin{pmatrix} -1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{3.16}$$

It is obvious that in (3.16), $S^T = S^{-1} = S$ and thus the reflection matrix S is orthogonal, preserving both lengths and angles, which is consistent with our knowledge about reflection. In general, orthogonal matrices preserve the overall structure of a coordination system as they preserve lengths and angles. Orthogonal matrices can thus represent only combinations of rotations and reflections.

3.6 3D Affine Transformations

In geometry, an **affine transformation** (or affine map) is a linear transformation (rotation, scaling or shear) followed by a translation. Under such a transformation, a point P in 3D graphics is transformed to another point Q by the equation,

$$Q = AP + \mathbf{v} \tag{3.17}$$

where A is a matrix and **v** is the translational vector. Several linear transformations can be combined into a single one, so that the general formula of (3.17) is still applicable. In the one-dimensional case, A and **v** are called **slope** and **intercept** respectively. Geometrically, one can show that an affine transformation in Euclidean space satisfies the following properties:

1. It preserves the collinearity relation between points. That is, the points which lie on a line continue to be on the same line after the transformation.

2. It preserves the ratios of distances along a line implying that for distinct collinear points p_1, p_2, p_3, the ratio $\frac{|p_2 - p_1|}{|p_3 - p_2|}$ remains the same after transformation.

If we use homogeneous coordinates that we have discussed in section 3.4, the translational vector can be "absorbed" in the transformation matrix M; in this case, (3.17) can be expressed as:

$$Q = MP \tag{3.18}$$

Equations (3.18) and (3.8) above represent affine transformations in homogeneous coordinate systems. In 3D graphics, affine transformation includes translation, scaling, and rotation. The affine transformation matrix M is always in the form given in (3.7). In the special case that $m_{ij} = \delta_{ij}$ (i.e. $m_{ii} = 1$, and $m_{ij} = 0$ for $i \neq j$), M is reduced to the identity matrix I:

$$I = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

In OpenGL, we can set the transformation matrix to the identity matrix using the command **glLoadIdentity**(). OpenGL provides a few functions to change or examine the values of the transformation matrix M:

1. void **glLoadIdentity**(void);
   Sets the transformation matrix to the 4×4 identity matrix

2. void **glLoadMatrix**{fd}(const TYPE *m);
   Sets the sixteen values of the transformation matrix to those specified by *m*.

3. void **glMultMatrix**{fd}(const TYPE *A);
   Multiplies the transformation matrix by the sixteen values pointed to by *A*. The 16 values pointed by *A* are not changed. i.e. $M \leftarrow MA$

4. void **glPushMatrix**(void);
   Pushes the transformation matrix onto stack. The current stack is determined by **glMatrixMode**().

5. void **glPopMatrix**(void);
   Pops matrix at top of stack and sets the transformation matrix to the 16 values of the popped matrix. The current stack is determined by **glMatrixMode**().

We can use the function **glGetFloatv**() or **glGetDoublev**() to retrieve and examine the values of the transformation matrix. Note that OpenGL operates in a column-major data format. That is, it first reads the first column of the transformation matrix M into memory,

followed by the second column and so on. In the same way, when it loads data from a memory array to the transformation matrix, the first four elements in the memory are loaded into the first column of M, and the next four elements into the second column and so on. Therefore, if we use the above commands to read the transformation matrix into a 2D 4×4 array (e.g. double a[4][4];), we will end up storing the transpose of the transformation matrix in the array.

3.6.1 Translation

If we let **t** to be the translational vector, the functional form for **translation** can be expressed as

$$\begin{aligned} x' &= x + \mathbf{t}_x \\ y' &= y + \mathbf{t}_y \\ z' &= z + \mathbf{t}_z \end{aligned} \tag{3.19}$$

When represented in the matrix form of (3.18) or (3.8), this becomes:

$$\begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & \mathbf{t}_x \\ 0 & 1 & 0 & \mathbf{t}_y \\ 0 & 0 & 1 & \mathbf{t}_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} \tag{3.20}$$

We can use the OpenGL command **glTranslate**{fd}() to perform a translation operation:

> void **glTranslate**{fd}(TYPE *tx*, TYPE *ty*, TYPE *tz*);
> Multiplies the current transformation matrix by a matrix that translates an object by the given *tx, ty*, and *tz* values.

This command sets $M = MT$, where T is the translation matrix:

$$T = \begin{pmatrix} 1 & 0 & 0 & \mathbf{t}_x \\ 0 & 1 & 0 & \mathbf{t}_y \\ 0 & 0 & 1 & \mathbf{t}_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{3.21}$$

3.6.2 Scaling

Scaling is the process of resizing a graphical object. The functional form for scaling is given by

$$\begin{aligned} x' &= s_x x \\ y' &= s_y y \\ z' &= s_z z \end{aligned} \tag{3.22}$$

Equation (3.22) scales the x, y, and z coordinates independently. In matrix form, this becomes

$$\begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix} = \begin{pmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} \tag{3.23}$$

In this case, the transform matrix is the scaling matrix S:

$$S = \begin{pmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{3.24}$$

OpenGL provides the functions "**glScale**{fd}(TYPE *sx*, TYPE *sy*, TYPE *sz*)" to do scaling transformation. That is, it sets $M = MS$.

OpenGL does not have any special commands for reflections or shearing transformations. A **reflection** transforms points across some plane to generate mirror-image points. Reflections across the coordinate planes are special cases of scaling transformations. For example, a reflection across the yz-plane can be done by setting s_x of S shown in (3.24) to -1, and s_y, and s_z to 1; such a reflection transformation matrix is

$$S = \begin{pmatrix} -1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{3.25}$$

This corresponds to executing the command,

```
glScalef( -1.0, 1.0, 1.0 );
```

The code below shows an example of scaling and reflection; the output of the code is shown in Figure 3-5. The teapot in the upper half of the figure is the original wired teapot. The one in the second half is a reflection of the upper one about the zx-plane (.i.e. $y \rightarrow -y$); it has also been scaled in the x-direction by a factor of 1.5. Note that we have used **glLoadIdentity**() to isolate the effects of modeling transformations. This function initializes the transformation matrix values to δ_{ij} to prevent successive transformations from having a cumulative effect. Even though using **glLoadIdentity**() repeatedly has the desired effect, it may be inefficient, because we may have to re-specify viewing or modeling transformations repeatedly.

```
glMatrixMode (GL_MODELVIEW);
glLoadIdentity();
glTranslatef ( 0, 0.8, 0 );     //move teapot upward
glutWireTeapot( 1.0 );

glLoadIdentity();
glColor3f (0.0, 1.0, 0.0);      //green color
glTranslatef ( 0, -0.8, 0 );    //move reflected teapot downward
glScalef ( 1.5, -1.0, 1.0 );    //scale and reflect
glutWireTeapot( 1.0 );
```

A **shearing** transformation is a more complicated kind of transformation. There is no direct OpenGL command that does a shearing transformation but we can always produce the effect by combining scaling, rotation and translation transformations. However, it may be easier to perform a desired transformation by setting the transformation matrix M to predefined values using the commands **glLoadMatrixf**() or **glMultMatrix**() discussed above.

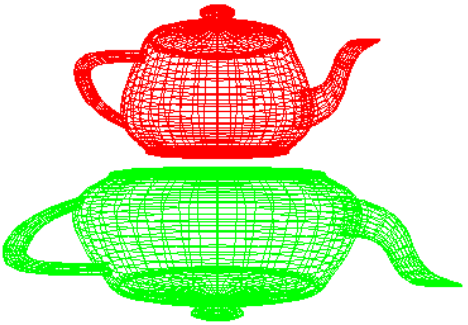

Figure 3-5 Scaling and Reflection

3.6.3 Rotations

Rotations about the coordinate axes are the simplest kinds of rotations and they form the basis of an arbitrary rotation. Rotations about an arbitrary axis can be done by appropriately combining translations and coordinate-axis rotations. We refer to coordinate-axis rotations as **elementary rotations**.

Conventionally, a counterclockwise rotation about an axis is given by a positive rotation angle; the direction is determined by the right-hand thumb rule. That is, if our right-hand fingers are wrapping in the counterclockwise direction our thumb points in the positive direction of the axis as shown in Figure 3-6.

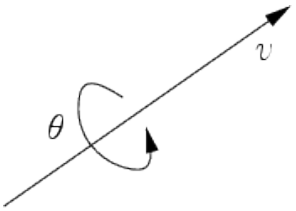

Figure 3-6 Counterclockwise rotation

Borrowing aviation terminology, rotations about z-axis, y-axis, and x-axis in counterclockwise direction are referred to as **yaw, pitch**, and **roll** respectively. Sometimes people also refer to these rotations as z-roll, y-roll, and x-roll. **roll** or x-roll. as shown in Figure 3-7.

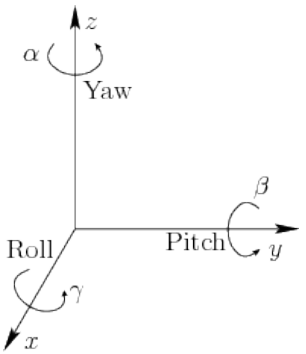

Figure 3-7 Yaw, Pitch, and Roll

z-axis rotation (yaw or z-roll)

The functional form for a rotation about the z-axis for an angle θ is given by

$$\begin{aligned} x' &= x\cos\theta - y\sin\theta \\ y' &= x\sin\theta + y\cos\theta \\ z' &= z \end{aligned} \tag{3.26}$$

In homogeneous coordinate matrix form, this becomes

$$\begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix} = \begin{pmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} \tag{3.27}$$

A positive rotation about the z-axis rotates the x-axis towards the y-axis. This rotation is usually denoted by $R_z(\theta)$, which is the rotation matrix

$$R_z(\theta) = \begin{pmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{3.28}$$

Figure 3-8 shows an example of such a rotation. In the figure, a teapot is rotated by 135^o about the z-axis.

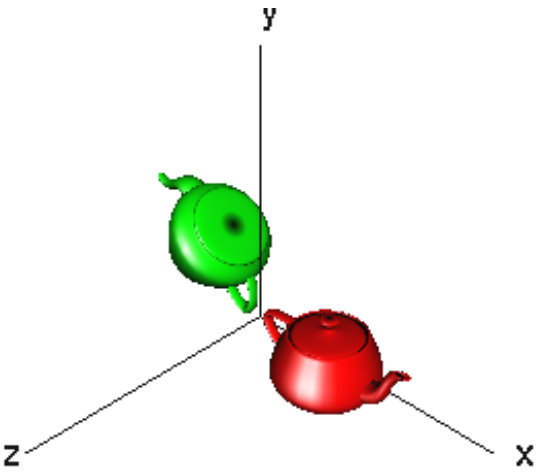

Figure 3-8 Rotation about z-axis for 135^o

y-axis rotation (pitch or y-roll)

A rotation about the y-axis for an angle θ can be specified by the equations

$$\begin{aligned} x' &= x\cos\theta + z\sin\theta \\ y' &= y \\ z' &= -x\sin\theta + z\cos\theta \end{aligned} \tag{3.29}$$

A positive rotation about the y-axis rotates the z-axis towards the x-axis. This rotation is usually denoted by $R_y(\theta)$, which is the rotation matrix

$$R_y(\theta) = \begin{pmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{3.30}$$

x-axis rotation (roll or x-roll)

A rotation about the x-axis for an angle θ can be specified by the equations

$$\begin{aligned} x' &= x \\ y' &= y\cos\theta - z\sin\theta \\ z' &= y\sin\theta + z\cos\theta \end{aligned} \tag{3.31}$$

A positive rotation about the x-axis rotates the y-axis towards the z-axis. This rotation is usually denoted by $R_x(\theta)$, which is the rotation matrix

$$R_x(\theta) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{3.32}$$

Composite Rotations

We can place a 3D object in any orientation using yaw, pitch, and roll rotations. We can form a single rotation matrix R by multiplying the yaw, pitch, and roll rotation matrices as follows.

$$R(\alpha,\beta,\gamma) = R_z(\alpha)R_y(\beta)R_x(\gamma) =$$

$$\begin{pmatrix} \cos\alpha\cos\beta & \cos\alpha\sin\beta\sin\gamma - \sin\alpha\cos\gamma & \cos\alpha\sin\beta\cos\gamma + \sin\alpha\sin\gamma & 0 \\ \sin\alpha\cos\beta & \sin\alpha\sin\beta\sin\gamma + \cos\alpha\cos\gamma & \sin\alpha\sin\beta\cos\gamma - \cos\alpha\sin\gamma & 0 \\ -\sin\beta & \cos\beta\sin\gamma & \cos\beta\cos\gamma & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{3.33}$$

Note that matrix operations on points start from the right side first. In (3.33), we perform the rotation $R_x(\gamma)$ first, $R_y(\beta)$ second, and $R_z(\alpha)$ last. That is, we obtain the final point, say P_3, from an initial point P_0 that undergoes matrix multiplications in the following order:

$$\begin{aligned} P_3 &= R(\alpha,\beta,\gamma)P_0 \\ &= R_z(\alpha)R_y(\beta)R_x(\gamma)P_0 \\ &= R_z(\alpha)R_y(\beta)P_1 \\ &= R_z(\alpha)P_2 \\ &= P_3 \end{aligned} \tag{3.34}$$

This means that $R(\alpha,\beta,\gamma)$ performs the roll first, then the pitch, and finally the yaw. If the order of these operations is changed, a different rotation matrix would result as matrix multiplications are not commutative. Note also that 3D rotations depend on three parameters, α, β, and γ, whereas 2D rotations depend only on a single parameter, θ. The angles α, β, and γ, are often called **Euler angles**.

Euler's Rotation Theorem

We have discussed rotations about coordinate axes and their rotation matrices. In practice, very often we need to rotate an object about an axis that points in an arbitrary direction. It turns out that every rotation can be represented in this way as stated by Euler's rotation theorem.

> **Euler's rotation theorem**
> Any rotation (or sequence of rotations) about a point is equivalent to a single rotation about some axis through that point.

We skip the derivation steps and proof of the theorem here. We just present the matrix that represents an arbitrary rotation about a vector pointing in a specific direction. Suppose $\mathbf{u}$ is a unit vector. That is, $\mathbf{u} = (u_x, u_y, u_z)$, and $|\mathbf{u}| = 1$. One can show that a rotation $R_{\mathbf{u}}(\theta)$, through an angle θ around axis $\mathbf{u}$ is given by

$$R_{\mathbf{u}}(\theta) = \begin{pmatrix} (1-c)u_x^2 + c & (1-c)u_xu_y - su_z & (1-c)u_xu_z + su_y & 0 \\ (1-c)u_xu_y + su_z & (1-c)u_y^2 + c & (1-c)u_yu_z - su_x & 0 \\ (1-c)u_xu_z - su_y & (1-c)u_yu_z + su_x & (1-c)u_z^2 + c & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{3.35}$$

where $c = \cos\theta$ and $s = \sin\theta$. OpenGL provides a command to perform such an operation:

> **glRotatef**(float $theta$, float u_x, float u_y, float u_z);

This command rotates an object for an angle $theta$ around an axis through the origin in the direction of the vector $\mathbf{u}$. That is, it sets the transformation matrix $M = MR_{\mathbf{u}}(\theta)$. The vector $\mathbf{u}$ must not be $\mathbf{0}$. If the vector provided in the argument is not a unit vector, OpenGL will normalize it to a unit vector. The rotation angle $theta$ is measured in degrees, and the direction of rotation follows the right-hand rule discussed above. The functions, glRotatef($theta$, 1, 0, 0); glRotatef($theta$, 0, 1, 0); and glRotatef($theta$, 0, 0, 1); reduce to the elementary rotations, x-roll, y-roll, and z-roll respectively.

If we need to rotate an object about an axis that does not pass through the origin of the coordinate system, all we need to do is to translate the origin of the coordinate system to a point on the rotating axis and perform the rotation around the axis. After rotation, we translate the system back.

Euler's rotation theorem also implies that the composition of two rotations is also a rotation. Consequently, the set of rotations has a structure known as a *rotation group*.

3.7 Composite Transformations

3.7.1 Modelview Transformation

As we have mentioned before, affine transformations can be combined to give a resultant affine transformation. This is done by multiplying matrices in the form of (3.7). We often use affine transformations to do modelling transformation but we can also use them for viewing transformation. OpenGL combines the modelling and viewing steps into one stage called **modelview transformation** (Figure 3-1). It is important to note that matrix multiplication is not commutative. That is, for two matrices of (3.7), M_1 and M_2, we have

$$M_2M_1 \neq M_1M_2 \tag{3.36}$$

Therefore, the process of *translate-then-rotate* is different from the process of *rotate-then-translate*. The operation *translate-then-rotate* is given by

$$M = RT \tag{3.37}$$

Figure 3-8 above shows the situation of translating a teapot from the origin to another position along the x-axis and then rotating it by 135^o around the z-axis. On the other hand, *rotate-then-translate* is

$$M = TR \tag{3.38}$$

Figure 3-9 below shows the corresponding *rotate-then-translate* situation.

Note again that matrix operations on a point start from the right. For example,

$$MP_0 = R(TP_0) = RP_1 = P_2$$

The following code shows an example of composite transformation:

```
glLoadIdentity();      //M = I
glTranslatef(..);      //M = I.T = T
glRotate(..);          //M = M.R = T.R
glScale(..);           //M = M.S = T.R.S
draw_a_point( P );     //Q = M.P = T.R.S.P
```

The equivalent order of operation on P is : scale, rotate, translate. Therefore, the code for *translate-then-rotate* that generates the image of Figure 3-8 is:

```
glLoadIdentity();              //M = I
glRotatef (135, 0, 0, 1 );     //M = MR = IR = R
glTranslatef ( 1.0, 0.0, 0 ); //M = MT = RT
glutSolidTeapot( 0.6 );        //P' = MP = (RT)P = R(TP)
```

As usual, the statements of the code are executed sequentially, from top to bottom. In the graphics pipeline, a vertex is always multiplied by the current modelview transformation matrix M before rendering (Figure 3-1). Thus, as shown in the comments of the code, any point P is transformed to $P' = MP = RTP = R(TP)$, showing that the translation effect takes place first, though in reality the vertex is transformed by multiplying its coordinates by the transformation matrix M.

Similarly, the code for *rotate-then-translate* (Figure 3-9) is:

```
glLoadIdentity();              //M = I
glTranslatef ( 1.0, 0.0, 0 ); //M = MT = IT = T
glRotatef (135, 0, 0, 1 );     //M = MR = TR
glutSolidTeapot( 0.6 );        //P' = MP = (TR)P = T(RP)
```

Beginners need to pay special attention in writing this kind of transformation code as its statements appear to be in "reverse order".

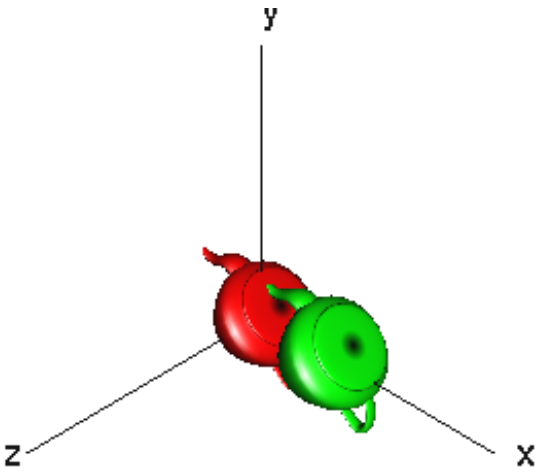

Figure 3-9 Rotation around z-axis for 135^o then Translate along x-axis

3.7.2 Affine Transformations Properties

Here are some useful and pleasing properties of affine transformations.

1. **Affine transformations preserve affine combination of points**.
   Suppose C is an affine combination of two points A and B:

   $$C = aA + bC$$

   where $a + b = 1$. Under an affine transformation with matrix M,

   $$MC = M(aA + bC) = a(MA) + b(MC)$$

   So the affine transformation of a combination of points is equal to the combination of the affine transformation of each of the points.

2. **Lines and planes are preserved.**
   A straight line passing through points A, B is given by

   $$L(t) = (1 - t)A + tB$$

   where $-\infty < t < \infty$. Under an affine transformation M, the image of $L(t)$ is the same affine combination of the images of A and B:

   $$L'(t) = ML(t) = (1 - t)MA + tMB$$

   Therefore, a line is still a line after the transformation.

   A plane can be written as an affine combination of 3 points, A, B, and C:

   $$P(s, t) = sA + tB + (1 - s - t)C$$

   When each point is transformed, this becomes:

   $$MP(s, t) = sMA + tMB + (1 - s - t)MC$$

   Therefore, a plane maps to a plane under the transformation.

3. **Parallelism of Lines and Planes is Preserved.**
   Consider two lines with the same direction $\mathbf{b}$:

$$L_1(t) = A + \mathbf{b}t$$

$$L_2(t) = B + \mathbf{b}t$$

   Under an affine transformation M, they become

$$L_1'(t) = MA + (M\mathbf{b})t$$

$$L_2'(t) = MB + (M\mathbf{b})t$$

   Both of the transformed lines have the same direction $(M\mathbf{b})$.

   Two planes with same directional normal can be described by

$$P_1(s,t) = A + \mathbf{a}s + \mathbf{b}t$$

$$P_2(s,t) = B + \mathbf{a}s + \mathbf{b}t$$

   Under affine transformation M, the planes become

$$P_1'(t) = MA + (M\mathbf{a})s + (M\mathbf{b})t$$

$$P_2'(t) = MB + (M\mathbf{a})s + (M\mathbf{b})t$$

   Again, both of the transformed planes have the same directional normal, characterized by $M\mathbf{a}$ and $M\mathbf{b}$.

3.8 Viewing Transformations

In OpenGL, viewing and modeling transformations are inextricably related. They are considered in the same stage of the graphics pipeline as shown in Figure 3-1. The modeling and viewing transformation matrices are combined into a single **modelview** matrix.

In general, a viewing transformation changes the position and orientation of the viewpoint. A viewing transformation on a graphics scene is generally composed of translations and rotations. It is like moving the camera to some position and rotating it to a desired direction when taking a photo. To achieve a certain scene composition in the final image, we can either move the camera to capture the image view or move all the objects in the opposite direction into the field of view of the camera. Therefore, a modeling transformation that rotates an object counterclockwise achieves the same effect as one that rotates the camera clockwise for the same number of degrees. Because of the matrix characteristics that matrix operations on a point starts from the right side, we must first call the viewing transformation functions before performing any modeling transformations, so that the modeling transformations take effect on the objects first. The following code shows an example of the structure with V denoting a viewing transformation matrix and D denoting a modeling matrix (as usual, M is the current transformation matrix):

```
glLoadIdentity();          //M = I
ViewingTransform();        //M = MV = V
ModelTransform();          //M = MD = VD
glutSolidTeapot( 0.6 );    //P' = MP = (VD)P = V(DP)
```

In an OpenGL program, if we do not do anything to set the viewpoint (camera location), the default location and orientation of the viewpoint are used. The default viewpoint is at the origin, looking down the negative z-axis. We may also use one of the following two methods to set the viewpoint.

1. We may use the modeling transformation functions such as **glTranslatef**() and **glRotatef**() to move the objects relative to the default viewpoint. Actually, we are **not** setting the viewpoint directly. We simply transform the objects relative to a fixed point.

2. We can use the Utility Library routine **gluLookAt**() to define a line of sight and a camera position. This routine makes use of a series of rotations and translations to set the viewpoint.

Of course, we can always create our own utility routine that encapsulates rotations and translations to set the camera position and orientation. For some applications, it might be more convenient to use custom routines to specify the viewing transformation. For example, we might want to render a scene at various viewpoint positions of a roller-coaster; in this case it is more convenient to specify a transformation in terms of the coordinates of a **Frenet frame**, which defines a local coordinate system at any point of a given curve.

3.8.1 The gluLookAt() Utility Routine

The utility function **gluLookAt**() lets us set the viewpoint position and a line of sight. The function takes three sets of arguments; one set specifies the location of the viewpoint; the second set defines a reference point towards which the camera is aimed, and the third set specifies the upward direction of the camera, usually referred to as the **up-vector**. We can choose the viewpoint to yield the desired view of the scene. Typically, the reference point is somewhere in the middle of the scene. (For example, if we build a scene around the origin, it is desirable to set the reference point to be the origin.) The setting of the up-vector is not as obvious; we shall explain this below. The following is the prototype and details of **gluLookAt**().

void **gluLookAt**(GLdouble *eyex*, GLdouble *eyey*, GLdouble *eyez*,
GLdouble *centerx*, GLdouble *centery*, GLdouble *centerz*,
GLdouble *upx*, GLdouble *upy*, GLdouble *upz*);

The function defines a viewing matrix V and multiplies it to the right of the current matrix (i.e. $M = MV$). The desired viewpoint is specified by (*eyex, eyey*, and *eyez*).

The (*centerx, centery*, and *centerz*) arguments specify a point along the desired line of sight (i.e. the camera is aiming at the point (*centerx, centery*, and *centerz*)).

The (*upx, upy*, and *upz*) arguments specify the up-vector, indicating which direction is up (that is, the direction from the bottom to the top of the viewing volume).

For example, the command "**gluLookAt** (0.0, 0.0, 5.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0);" does the following:

1. Sets the camera location (viewpoint) at (0.0, 0.0, 5.0).
2. Points the camera towards the point (0.0, 0.0, 0.0).
3. Specifies the orientation of camera by the up-vector (0.0, 1.0, 0.0). (So in this example, y-axis is the up-down axis.)

Now lets take a more detailed look at what exactly **gluLookAt**() does. We have discussed the concept of world coordinates. All the object positions and most of the parameters of OpenGL functions, including those of **gluLookAt**() are specified in world coordinates. However, we can always attach a coordinate system to an object and move it

along with the associated object; this system is called the **object coordinate system**. At the end, we want to specify the object positions relative to the eye (or camera). When the coordinates are specified relative to the eye, the coordinate system is called **eye coordinate system**. For convenience of discussion, lets assume that e is a point where the eye is located, C is the point that the camera (or eye) is aiming at, and $\mathbf{U}$ is the normalized up-vector (i.e. $|\mathbf{U}| = 1$). (If the up-vector provided is not normalized, gluLookAt() will normalize it.) So we have

$$e = \begin{pmatrix} e_x \\ e_y \\ e_z \\ 1 \end{pmatrix}, \quad C = \begin{pmatrix} C_x \\ C_y \\ C_z \\ 1 \end{pmatrix}, \quad \mathbf{U} = \begin{pmatrix} U_x \\ U_y \\ U_z \\ 0 \end{pmatrix}$$

The corresponding gluLookAt() function to use these parameters is

$$\text{gluLookAt}(e_x, e_y, e_z, C_x, C_y, C_z, U_x, U_y, U_z);$$

When the function is called, it computes a forward-vector $\mathbf{F}$ given by

$$\mathbf{F} = C - e = \begin{pmatrix} C_x - e_x \\ C_y - e_y \\ C_z - e_z \\ 0 \end{pmatrix}$$

(Recall that the difference between two points is a vector.) It then normalizes $\mathbf{F}$ to

$$\mathbf{f} = \frac{\mathbf{F}}{|\mathbf{F}|}$$

We can then define a unit backward-vector $\mathbf{b} = -\mathbf{f}$ and compute a unit side-vector $\mathbf{s}$ given by

$$\mathbf{s} = \mathbf{U} \times \mathbf{b}$$

and another unit up-vector $\mathbf{u}$ given by

$$\mathbf{u} = \mathbf{b} \times \mathbf{s}$$

The vectors $\mathbf{s}$, $\mathbf{u}$, and $\mathbf{b}$ are orthogonal unit vectors; they form the axes of a new orthonormal coordinate system. We can construct a viewing transformation matrix V as follows

$$V = \begin{pmatrix} s_x & u_x & b_x & 0 \\ s_y & u_y & b_y & 0 \\ s_z & u_z & b_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{3.39}$$

The function call "gluLookAt$(e_x, e_y, e_z, C_x, C_y, C_z, U_x, U_y, U_z)$;" is equivalent to

$$\begin{array}{l} \text{glMultMatrixf}(V^T); \\ \text{glTranslatef}(-e_x, -e_y, -e_z); \end{array} \tag{3.40}$$

We need to use the transpose of V, which is V^T in (3.40) because as we mentioned earlier, OpenGL array operations are column-major. So the first row of V^T, which is $(s_x, s_y, s_z, 0)$, will become the first column of the corresponding OpenGL matrix.

What (3.40) means is that we move the origin of the coordinate system to the eye location and use the orthogonal unit vectors, **s**, **u**, and **b** as the coordinate axes. (See also discussions in Section 3.5.) That is, we align the view volume with the line-of-sight of the camera. The effect of this command is to change the coordinates of the scene vertices into the **camera's coordinate system** (which is also called eye coordinate system).

The following few code segments show some examples of the usage of **gluLookAt**(); the outputs are shown on the right side of the code segments. Each code segment uses the following parameters and **drawLine**() routine that draws a line to draw the coordinate axes:

```
float origin[3] = {0, 0, 0};
float axes[3][3] = {{2.0, 0, 0}, {0, 2.0, 0}, {0, 0, 2.0}};
void drawLine( float v0[], float v1[] ) {
   glBegin( GL_LINE );
     glVertex3fv( v0 );
     glVertex3fv( v1 );
   glEnd();
}
```

Example 3-1

Line of sight = negative z-axis, **up** = (0, 1, 0)
The z-axis in this example is pointing out of the page.

```
void display(void)
{
  .....
  glMatrixMode (GL_MODELVIEW);
  glLoadIdentity();
  gluLookAt ( 0.0, 0.0, 5.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0);
  for ( int i = 0; i ¡ 3; i++ )
     drawLine ( origin, axes[i] );
  glutSolidTeapot( 0.6 );
}
```

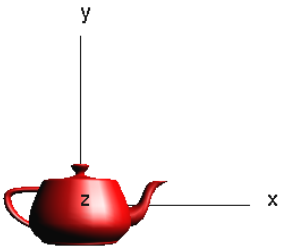

Figure 3-10

Example 3-2

Line of sight = negative z-axis, **up** = (1, 0, 0)
The z-axis in this example is pointing out of the page.

```
void display(void)
{
  .....
  glMatrixMode (GL_MODELVIEW);
  glLoadIdentity();
  gluLookAt ( 0.0, 0.0, 5.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0);
  for ( int i = 0; i ¡ 3; i++ )
     drawLine ( origin, axes[i] );
  glutSolidTeapot( 0.6 );
}
```

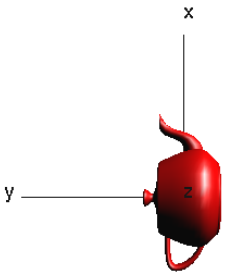

Figure 3-11

Example 3-3

Line of sight = (5, 5, 5) - (0, 0, 0), **up** = (1, 1, 0)
x, y, z axes are pointing out of the plane.

```
void display(void)
{
   .....
   glMatrixMode (GL_MODELVIEW);
   glLoadIdentity();
   gluLookAt ( 5.0, 5.0, 5.0, 0.0, 0.0, 0.0, 1.0, 1.0, 0.0);
   for ( int i = 0; i ¡ 3; i++ )
      drawLine ( origin, axes[i] );
   glutSolidTeapot( 0.6 );
}
```

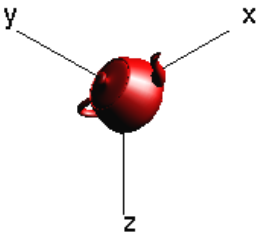

Figure 3-12

3.9 Coding Examples

3.9.1 Robot Arm

Suppose we want to rotate an object about a 'z' axis through $(-1, 0, 0)$. We can achieve this by moving everything by 1 in the x-direction, performing the rotation, and moving things back by 1. This may be a simple example of applying modeling transformation. However, if we are not careful, we may end up applying the wrong order of transformations which will lead to undesirable results. Consider the following two code segments. *Which one is the correct code to rotate an object about a 'z' axis through* $(-1, 0, 0)$ *?*

Code 1:

```
glTranslatef ( 1.0, 0.0, 0.0 );
glRotatef( degrees, 0.0, 0.0, 1.0 );
glTranslatef ( -1.0, 0.0, 0.0 );
draw_object();
```

Code 2:

```
glTranslatef ( -1.0, 0.0, 0.0 );
glRotatef( degrees, 0.0, 0.0, 1.0 );
glTranslatef ( 1.0, 0.0, 0.0 );
draw_object();
```

Before reading on, try to answer the above question and convince yourself that your answer is correct.

The problem requires us to rotate objects about a 'z' axis passing through $(-1, 0, 0)$. Therefore, we need to shift the origin of the coordinate system to $(-1, 0, 0)$. This is equivalent to shifting the objects by $(+1, 0, 0)$. Remember that all the OpenGL transformation commands such as glTranslatef() and glRotatef() operate on objects, not the coordinate

system and that the latest command before drawing a vertex is the first one to operate on the vertex. Thus **Code 2** is the correct answer.

To build a robot arm, we can call the function **glutWireCube**() to create cubes and then scale them to be used as the segments of the robot arm. However, we need to first call the appropriate modeling transformations to align each segment. When we use **glutWireCube**() to create a cube, it is created around the origin of the coordinate system. In other words, the origin is at the center of the cube. As a rotating axis always passes through the origin $(0, 0, 0)$, if we want to rotate an object about another axis (e.g. an axis passing through a cube edge), we first move the cube so that one of its edges passes through the origin, which will be the pivot point for rotation. We can do this using glTranslatef(); after rotation, we move the objects back with glTranslatef() again. The following code builds the first arm segment (upper arm):

```
glTranslatef (-1.0, 0.0, 0.0);
glRotatef ( angle1, 0.0, 0.0, 1.0);
glTranslatef (1.0, 0.0, 0.0);
glPushMatrix();
glScalef (2.0, 0.4, 0.5);
glutWireCube (1.0);
glPopMatrix();
```

In the code, the command **glPushMatrix**() before glScalef() and the command **glPopMatrix**() are to guarantee that the scaling function only operates on the cube but does not have any effect on any other objects. We build the second segment (lower arm) by moving the local system (the cube) to the next pivot point:

```
glTranslatef (1.0, 0.0, 0.0);
glRotatef (angle2, 0.0, 0.0, 1.0);
glTranslatef (1.0, 0.0, 0.0);
glPushMatrix();
glScalef (2.0, 0.4, 0.4);
glutWireCube (1.0);
glPopMatrix();
```

Note here that we move the cube by $+1$ unit by "glTranslate(1.0, 0.0, 0.0);" so that we rotate it around an axis passing through it left-side edge. After the rotation, we move the object a further $+1$ unit (the top "glTranslatef (1.0, 0.0, 0.0);" statement) so that the left-side edge of this rotated cube is at the right-side edge of the previous cube (upper arm) as shown in Figure 3-13. The dots in the figure show the pivot points. The following is the code that creates the robot arm of Figure 3-13.

```
/*
  glutWireCube ( 1.0 )
  produces a cube: -0.5 to 0.5
  R = rotation, T = translation, S = scaling,  T(2)=T(1).T(1)
*/
void display(void)
{  ......
   glPushMatrix();                    //Save M0
   glTranslatef (-1.0, 0.0, 0.0);     //M1 = T(-1)
   drawDot( 0, 0, 0 );                //Draw dot at (0,0,0)
   glRotatef(angle1, 0.0, 0.0, 1.0); //M2=T(-1).R1
   glTranslatef (1.0, 0.0, 0.0);      //M3 = T(-1).R1.T(+1)
   glPushMatrix();                    //Save M3
   glScalef (2.0, 0.4, 0.5);          //M4 = T(-1).R1.T(+1).S
   glutWireCube (1.0);                //P' = T(-1).R1.T(+1).S.P
```

```
   glPopMatrix();                    //Restore M3 = T(-1).R1.T(+1)
   glTranslatef (1.0, 0.0, 0.0);     //M5 = T(-1).R1.T(+1).T(+1)
   drawDot( 0, 0, 0 );               //P' = T(-1).R1.T(+1).T(+1).P
   glRotatef(angle2, 0.0, 0.0, 1.0); //M6 = T(-1).R1.T(+2).R2
   glTranslatef (1.0, 0.0, 0.0);     //M7 = T(-1).R1.T(2).R2.T(+1)
   glPushMatrix();                   //Save M7
   glScalef (2.0, 0.4, 0.4);         //M8 = T(-1).R1.T(2).R2.T(+1).S
   glutWireCube (1.0);               //P' = T(-1).R1.T(2).R2.T(+1).S.P
   glPopMatrix();                    //Restore M7
   glPopMatrix();                    //Restore M0
}
```

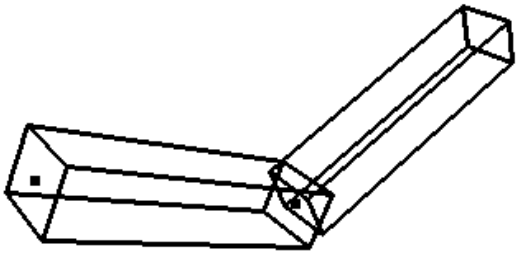

Figure 3-13 Robot Arm

3.9.2 Solar System

This simple example shows how to use affine transformations to model a solar system. In our example, we have three planets revolving around the sun about the y-axis; the most distant planet has a moon revolving around it about the z-axis of the local coordinate system of the planet; a satellite revolves around the moon about the y-axis of the local coordinate system of the moon. The following code segment shows how this is done.

```
void satellite()
{
   glColor3f ( 0.1, 1, 0.1 );
   glPushMatrix();
   glRotatef ( angleSat, 0, 1, 0 );
   glTranslatef ( 0.5, 0.0, 0 );
   glutSolidSphere( 0.1, 32, 24 );
   glPopMatrix();
}

void moon()
{
   glColor3f ( 1, 1, 0 );  //yellow
   glPushMatrix();
   glRotatef ( angleMoon, 0, 0, 1 );
   glTranslatef ( 1, 0.0, 0 );
   glutSolidSphere( 0.2, 32, 24 );
   satellite();  //satellite revolving around moon
   glPopMatrix();
}

// r is radius of planet, d is distance from the sun
void planet( float r, float d, int n )
{
   float angle = anglePlanet / d;
   glPushMatrix();
   glColor3f (0.6, 0.4, 0.5);
```

```
   glRotatef ( angle, 0, 1, 0 );
   glTranslatef ( d, 0.0, 0 );
   glutSolidSphere( r, 32, 24 );
   if ( n == 2 )
      moon();     //moon revolving around outermost planet
   glPopMatrix();
}

void display(void)
{
   glColor3f (1.0, 0.0, 0.0);     // red color
   glutSolidSphere( 1.5, 32, 24 );// the sun

   planet( 0.5, 2, 0 );  //planets revolving around sun
   planet( 0.8, 4, 1 );
   planet( 0.4, 6, 2 );
}
```

We can add animation to the scene by making use of the function **glutIdleFunc**(), which will be discussed in Chapter 11. A snapshot of such an animated solar system is shown in Figure 3-14 below. We can also add lighting, which will be discussed in Chapter 7, to the scene to make the model look more realistic.

Figure 3-14 Solar System

An Introduction to Digital Video Data Compression in Java

by *Fore June*

The book describes the the principles of digital video data compression techniques and its implementations in java. Topics covered include RBG-YCbCr conversion, macroblocks, DCT and IDCT, integer arithmetic, quantization, reorder, run-level encoding, entropy encoding, motion estimation, motion compensation and hybrid coding. See

http://www.forejune.com/

January 2011

ISBN-10: 1456570870

ISBN-13: 978-1456570873

An Introduction to Video Compression in C/C++

by *Fore June*

The book describes the the principles of digital video data compression techniques and its implementations in C/C++. Topics covered include RBG-YCbCr conversion, macroblocks, DCT and IDCT, integer arithmetic, quantization, reorder, run-level encoding, entropy encoding, motion estimation, motion compensation and hybrid coding.

January 2010
ISBN: 9781451522273