Interactive Computer Graphics

A Top-Down Approach with WebGL

SEVENTH EDITION

Edward Angel • Dave Shreiner

ALWAYS LEARNING

PEARSON

INTERACTIVE COMPUTER GRAPHICS

A Top-Down Approach with **WebGL** | 7TH EDITION

GLOBAL EDITION

in the same direction simplifies the subsequent steps. We say that **v** is the result of **normalizing u**. We have already seen that moving the fixed point to the origin is a helpful technique. Thus, our first transformation is the translation $\mathbf{T}(-\mathbf{p}_0)$, and the final one is $\mathbf{T}(\mathbf{p}_0)$. After the initial translation, the required rotation problem is as shown in Figure 4.53. Our previous example (see Section 4.10.2) showed that we could get an arbitrary rotation from three rotations about the individual axes. This problem is more difficult because we do not know what angles to use for the individual rotations. Our strategy is to carry out two rotations to align the axis of rotation, **v**, with the z-axis. Then we can rotate by θ about the z-axis, after which we can undo the two rotations that did the aligning. Our final rotation matrix will be of the form

$$\mathbf{R} = \mathbf{R}_x(-\theta_x)\mathbf{R}_y(-\theta_y)\mathbf{R}_z(\theta)\mathbf{R}_y(\theta_y)\mathbf{R}_x(\theta_x).$$

This sequence of rotations is shown in Figure 4.54. The difficult part of the process is determining θ_x and θ_y.

We proceed by looking at the components of **v**. Because **v** is a unit-length vector,

$$\alpha_x^2 + \alpha_y^2 + \alpha_z^2 = 1.$$

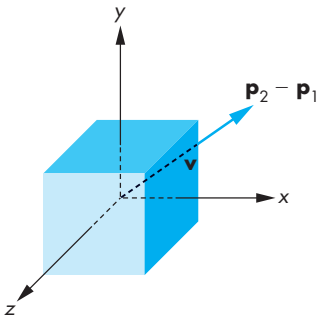

FIGURE 4.53 Movement of the fixed point to the origin.

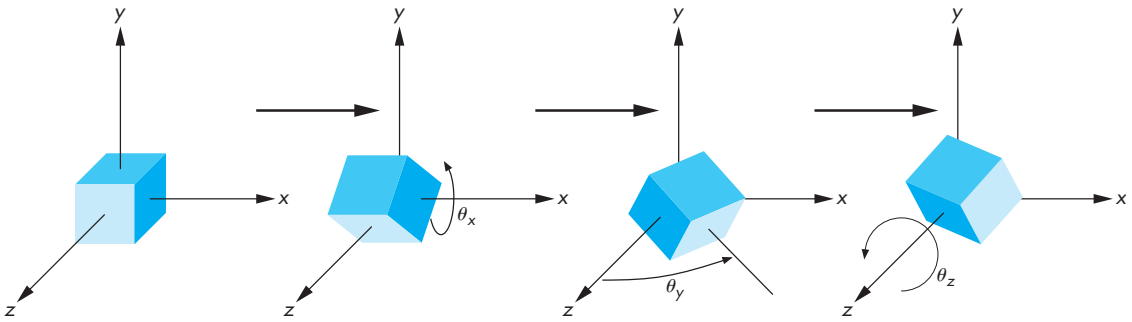

FIGURE 4.54 Sequence of rotations.

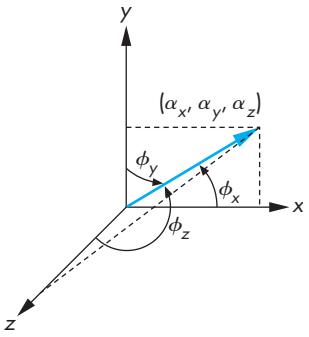

FIGURE 4.55 Direction angles.

We draw a line segment from the origin to the point $(\alpha_x, \alpha_y, \alpha_z)$. This line segment has unit length and the orientation of **v**. Next, we draw the perpendiculars from the point $(\alpha_x, \alpha_y, \alpha_z)$ to the coordinate axes, as shown in Figure 4.55. The three **direction angles**—ϕ_x, ϕ_y, ϕ_z—are the angles between the line segment (or **v**) and the axes. The **direction cosines** are given by

$$\cos \phi_x = \alpha_x,$$

$$\cos \phi_y = \alpha_y,$$

$$\cos \phi_z = \alpha_z.$$

Only two of the direction angles are independent, because

$$\cos^2 \phi_x + \cos^2 \phi_y + \cos^2 \phi_z = 1.$$

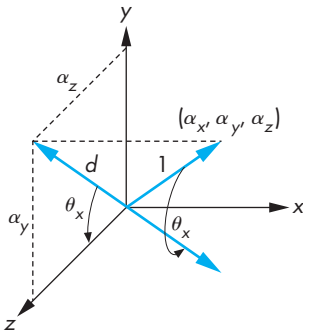

FIGURE 4.56 Computation of the x rotation.

We can now compute θ_x and θ_y using these angles. Consider Figure 4.56. It shows that the effect of the desired rotation on the point $(\alpha_x, \alpha_y, \alpha_z)$ is to rotate the line segment into the plane $y = 0$. If we look at the projection of the line segment (before the rotation) on the plane $x = 0$, we see a line segment of length d on this plane. Another way to envision this figure is to think of the plane $x = 0$ as a wall and consider a distant light source located far down the positive x-axis. The line that we see on the wall is the shadow of the line segment from the origin to $(\alpha_x, \alpha_y, \alpha_z)$. Note that the length of the shadow is less than the length of the line segment. We can say the line segment has been **foreshortened** to $d = \sqrt{\alpha_y^2 + \alpha_z^2}$. The desired angle of rotation is determined by the angle that this shadow makes with the z-axis. However, the rotation matrix is determined by the sine and cosine of θ_x. Thus, we never need to compute θ_x; rather, we need only compute

$$\mathbf{R}_x(\theta_x) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \alpha_z/d & -\alpha_y/d & 0 \\ 0 & \alpha_y/d & \alpha_z/d & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

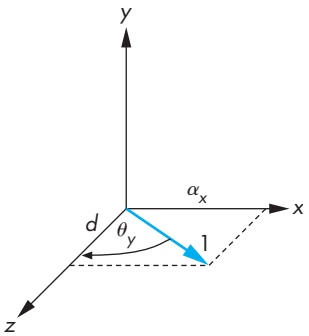

FIGURE 4.57 Computation of the y rotation.

We compute $\mathbf{R}_y$ in a similar manner. Figure 4.57 shows the rotation. This angle is clockwise about the y-axis; therefore, we have to be careful about the sign of the sine terms in the matrix, which is

$$\mathbf{R}_y(\theta_y) = \begin{bmatrix} d & 0 & -\alpha_x & 0 \\ 0 & 1 & 0 & 0 \\ \alpha_x & 0 & d & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

Finally, we concatenate all the matrices to find

$$\mathbf{M} = \mathbf{T}(\mathbf{p}_0)\mathbf{R}_x(-\theta_x)\mathbf{R}_y(-\theta_y)\mathbf{R}_z(\theta)\mathbf{R}_y(\theta_y)\mathbf{R}_x(\theta_x)\mathbf{T}(-\mathbf{p}_0).$$

Let's look at a specific example. Suppose that we wish to rotate an object by 45 degrees about the line passing through the origin and the point (1, 2, 3). We leave the fixed point at the origin. The first step is to find the point along the line that is a unit distance from the origin. We obtain it by normalizing (1, 2, 3) to $(1/\sqrt{14}, 2/\sqrt{14}, 3/\sqrt{14})$, or $(1/\sqrt{14}, 2/\sqrt{14}, 3/\sqrt{14}, 1)$ in homogeneous coordinates. The first part of the rotation takes this point to (0, 0, 1, 1). We first rotate about the x-axis by the angle $\cos^{-1}\frac{3}{\sqrt{14}}$. This matrix carries $(1/\sqrt{14}, 2/\sqrt{14}, 3/\sqrt{14}, 1)$ to $(1/\sqrt{14}, 0, \sqrt{13/14}, 1)$, which is in the plane $y = 0$. The y rotation must be by the angle $-\cos^{-1}(\sqrt{13/14})$. This rotation aligns the object with the z-axis, and now we can rotate about the z-axis by the desired 45 degrees. Finally, we undo the first two rotations. If we concatenate these five transformations into a single rotation matrix $\mathbf{R}$, we find that

$$\mathbf{R} = \mathbf{R}_x\left(-\cos^{-1}\frac{3}{\sqrt{13}}\right)\mathbf{R}_y\left(\cos^{-1}\sqrt{\frac{13}{14}}\right)\mathbf{R}_z(45)\,\mathbf{R}_y\left(-\cos^{-1}\sqrt{\frac{13}{14}}\right)$$
$$\times\,\mathbf{R}_x\left(\cos^{-1}\frac{3}{\sqrt{13}}\right)$$
$$= \begin{bmatrix} \frac{2+13\sqrt{2}}{28} & \frac{2-\sqrt{2}-3\sqrt{7}}{14} & \frac{6-3\sqrt{2}+4\sqrt{7}}{28} & 0 \\ \frac{2-\sqrt{2}+3\sqrt{7}}{14} & \frac{4+5\sqrt{2}}{14} & \frac{6-3\sqrt{2}-\sqrt{7}}{14} & 0 \\ \frac{6-3\sqrt{2}-4\sqrt{7}}{28} & \frac{6-3\sqrt{2}+\sqrt{7}}{14} & \frac{18+5\sqrt{2}}{28} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

This matrix does not change any point on the line passing through the origin and the point (1, 2, 3). If we want a fixed point other than the origin, we form the matrix

$$\mathbf{M} = \mathbf{T}(\mathbf{p_f})\mathbf{R}\mathbf{T}(-\mathbf{p_f}).$$

This example is not trivial. It illustrates the powerful technique of applying many simple transformations to get a complex one. The problem of rotation about an arbitrary point or axis arises in many applications. The major variations lie in the manner in which the axis of rotation is specified. However, we can usually employ techniques similar to the ones that we have used here to determine direction angles or direction cosines.

4.11 TRANSFORMATION MATRICES IN WEBGL

We can now focus on the implementation of a homogeneous-coordinate transformation package and that package's interface to the user. We have introduced a set of frames, including the world frame and the camera frame, that should be important for developing applications. In a shader-based implementation of OpenGL, the existence or nonexistence of these frames is entirely dependent on what the application

FIGURE 4.58 Current transformation matrix (CTM).

programmer decides to do.[9] In a modern implementation of OpenGL, the application programmer can choose not only which frames to use but also where to carry out the transformations between frames. Some will best be carried out in the application, others in a shader.

As we develop a method for specifying and carrying out transformations, we should emphasize the importance of *state*. Although very few state variables are predefined in WebGL, once we specify various attributes and matrices, they effectively define the state of the system. Thus, when a vertex is processed, how it is processed is determined by the values of these state variables.

The two transformations that we will use most often are the model-view transformation and the projection transformation. The model-view transformation brings representations of geometric objects from the application or object frame to the camera frame. The projection matrix will both carry out the desired projection and change the representation to clip coordinates. We will use only the model-view matrix in this chapter. The model-view matrix normally is an affine transformation matrix and has only 12 degrees of freedom, as discussed in Section 4.7. The projection matrix, as we will see in Chapter 5, is also a 4×4 matrix but is not affine.

4.11.1 Current Transformation Matrices

The generalization common to most graphics systems is of a **current transformation matrix** (**CTM**). The CTM is part of the pipeline (Figure 4.58); thus, if $\mathbf{p}$ is a vertex specified in the application, then the pipeline produces $\mathbf{Cp}$. Note that Figure 4.58 does not indicate where in the pipeline the current transformation matrix is applied. If we use a CTM, we can regard it as part of the state of the system.

First, we will introduce a simple set of functions that we can use to form and manipulate 4×4 affine transformation matrices. Let $\mathbf{C}$ denote the CTM (or any other 4×4 affine matrix). Initially, we will set it to the 4×4 identity matrix; it can be reinitialized as needed. If we use the symbol $\leftarrow$ to denote replacement, we can write this initialization operation as

$$\mathbf{C} \leftarrow \mathbf{I}.$$

The functions that alter $\mathbf{C}$ are of three forms: those that load it with some matrix and those that modify it by premultiplication or postmultiplication by a matrix. The three transformations supported in most systems are translation, scaling with a fixed point of the origin, and rotation with a fixed point of the origin. Symbolically, we can write

9. In earlier versions of OpenGL that relied on the fixed-function pipeline, the model-view and projection matrices were part of the specification and their state was part of the environment.

these operations in postmultiplication form as

$\mathbf{C} \leftarrow \mathbf{CT}$

$\mathbf{C} \leftarrow \mathbf{CS}$

$\mathbf{C} \leftarrow \mathbf{CR}$

and in load form as

$\mathbf{C} \leftarrow \mathbf{T}$

$\mathbf{C} \leftarrow \mathbf{S}$

$\mathbf{C} \leftarrow \mathbf{R}$.

Most systems allow us to load the CTM with an arbitrary matrix **M**,

$\mathbf{C} \leftarrow \mathbf{M}$,

or to postmultiply by an arbitrary matrix **M**,

$\mathbf{C} \leftarrow \mathbf{CM}$.

Although we will occasionally use functions that set a matrix, most of the time we will alter an existing matrix; that is, the operation

$\mathbf{C} \leftarrow \mathbf{CR}$,

is more common than the operation

$\mathbf{C} \leftarrow \mathbf{R}$.

4.11.2 Basic Matrix Functions

Using our matrix types, we can create and manipulate three- and four-dimensional matrices. Because we will work mostly in three dimensions using four-dimensional homogeneous coordinates, we will illustrate only that case. We can create an identity matrix by

```
var a = mat4();
```

or fill it with components by

```
var a = mat4(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15);
```

or by vectors as in

```
var a = mat4(
  vec4(0, 1, 2, 3),
  vec4(4, 5, 6, 7),
  vec4(8, 9, 10, 11),
  vec4(12, 13, 14, 15)
);
```

We can copy into an existing matrix using

```
b = mat4(a);
```

We also can find the determinant, inverse, and transpose of a matrix a:

```
var det = determinant(a);
b = inverse(a);   // 'b' becomes inverse of 'a'
b = transpose(a); // 'b' becomes transpose of 'a'
```

We can multiply two matrices together by

```
c = mult(a b); // c = a * b
```

If d and e are vec3s, we can multiply d by a matrix a:

```
e = mult(a, d);
```

We can also reference or change individual elements using standard indexing, as in

```
a[1][2] = 0;
var d = vec4(a[2]);
```

4.11.3 Rotation, Translation, and Scaling

In our applications and shaders, the matrix that is most often applied to all vertices is the product of the model-view matrix and the projection matrix. We can think of the CTM as the product of these matrices (Figure 4.59), and we can manipulate each individually by working with the desired matrix.

Using our matrix and vector types, we can form affine matrices for rotation, translation, and scaling using the following six functions:

```
var a = rotate(angle, direction);
var b = rotateX(angle);
var c = rotateY(angle);
var d = rotateZ(angle);
var e = scale(scaleVector);
var f = translate(translateVector);
```

For rotation, the angles are specified in degrees and the rotations are around a fixed point at the origin. In the translation function, the variables are the components of the displacement vector; for scaling, the variables determine the scale factors along the coordinate axes and the fixed point is the origin.

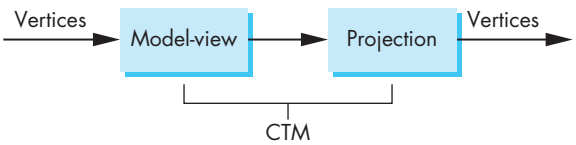

FIGURE 4.59 Model-view and projection matrices.