APPENDIX D
Building a Simple Camera Controller

Throughout this book, we built our graphics scenes near the origin, and placed the camera at a fixed location somewhere on the positive world Z axis, looking in the direction of the world negative Z axis towards the origin. Because the built-in OpenGL camera already points down the negative Z axis, and we never *turned* the camera, we were able (as described in Chapters 3 and 4) to build a very simple *view matrix* that took into account only the camera's *location*, without worrying about the camera's *orientation*. The command that we used to build this simple view matrix was typically something like:

```
vMat.translation(-cameraX, -cameraY, -cameraZ);
```

As described in Chapter 4, this creates a translation matrix that translates object vertices to the negative of the desired camera position, creating the illusion that we have moved the camera to position (cameraX, cameraY, cameraZ). This works fine as long as there is no change in the orientation of the camera; that is, if there is no need to *turn* the camera.

However, many common 3D graphics applications allow the user to move and turn the camera. The obvious example is in video games, where the human playing the game moves freely through a virtual 3D world. Video games are typically built using *game engines*, which provide a framework and suite of game-building tools. One such tool is a *camera controller*, which handles the positioning, moving, and turning of the camera.

At the heart of a camera controller is its ability to generate a view matrix that supports both the position and the orientation of the camera. Actually, we already described a formula for building such a view matrix back in Chapter 3, specifically in Figure 3.13. We just didn't use it in the code examples. It is reproduced here:

$$\underset{\textit{point } P_C \textit{ in eye space}}{\begin{pmatrix} X_C \\ Y_C \\ Z_C \\ 1 \end{pmatrix}} = \underbrace{\underset{\textbf{\textit{R}}\textit{ (rotation)}}{\overset{\textit{negative of camera rotation angles}}{\begin{bmatrix} \hat{U}_X & \hat{U}_Y & \hat{U}_Z & 0 \\ \hat{V}_X & \hat{V}_Y & \hat{V}_Z & 0 \\ -\hat{N}_X & -\hat{N}_Y & -\hat{N}_Z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}}} * \underset{\textbf{\textit{T}}\textit{ (translation)}}{\overset{\textit{negative of camera location}}{\begin{bmatrix} 1 & 0 & 0 & -C_X \\ 0 & 1 & 0 & -C_Y \\ 0 & 0 & 1 & -C_Z \\ 0 & 0 & 0 & 1 \end{bmatrix}}}}_{\textbf{\textit{V}}\textit{ (viewing transform)}} * \underset{\textit{world point } P_W}{\begin{pmatrix} P_X \\ P_Y \\ P_Z \\ 1 \end{pmatrix}}$$

The *viewing transform* matrix shown here – instead of the simple code described at the beginning of this appendix – requires us to maintain not only the camera's position C, but also its orientation. As described previously in Chapter 3 (and reflected in the formula from Figure 3.13), the camera's orientation is represented with three mutually orthogonal vectors U, V, and N. This enables us to move the camera (by changing C), and turning the camera (by changing U, V, and N).

It may not be immediately obvious, just by looking at the formula, how to go about moving and turning the camera. This appendix describes how to do this, and thus how to build a simple camera controller. It can be implemented in its own class or file as desired.

D.1 Elements of a Camera Controller

A camera controller should include the following:

- Data structures for storing location C and orientation vectors U, V, and N
- Functions for moving the camera by modifying C
- Functions for turning the camera by modifying U, V, and N
- A function that builds a view matrix based on C, U, V, and N

Let's take a close look at each of these elements.

D.1.1 Storing the Location and Orientation Vectors

The camera location C is a designated point in 3D world space, and therefore simply requires three float values C_x, C_y, and C_z. These can be stored in separate variables, or in a single JOML Vector3f. We have already been storing these as float variables, and can continue to do so:

```
private float camera, cameraY, cameraZ;
// default camera location on the Z axis
cameraX = 0.0f; cameraY = 0.0f; cameraZ = 8.0f;
```

The orientation vectors U, V, and N can also be stored in float variables (three each), or in JOML Vector3fs. Here we opt for the latter, and utilize the set command to initialize them:

```
private Vector3f cameraU = new Vector3f();
private Vector3f cameraV = new Vector3f();
private Vector3f cameraN = new Vector3f();
cameraU.set(1.0f, 0.0f, 0.0f);
cameraV.set(0.0f, 1.0f, 0.0f);
cameraN.set(0.0f, 0.0f, -1.0f);
```

Note that we have taken care to initialize the orientation vectors so as to satisfy the following three requirements (the view matrix computations in Figure 3.13 depend on them):

- They must be of unit length
- They must be mutually orthogonal
- They must be arranged as a *left-handed* system

Specifically in the above example, the right-facing U vector is initialized so that it points down the positive X axis, the up-facing V vector points up the positive Y axis, and the forward-facing N axis points down the *negative* Z axis. This is consistent with the default OpenGL camera orientation as shown back in Chapter 3 (Figure 3.11). Other initial orientations are possible, so long as they obey the three requirements listed above.

It is important that vectors U, V, and N maintain their unit length and mutual orthogonality as the camera moves and turns. We will describe how this is accomplished shortly.

D.1.2 Functions for Moving the Camera

Moving the camera refers to changing its position C, without altering its orientation vectors U, V, and N. The most common camera movement is *forward* and *backward*, but other movements are possible. Moving "forward" means that the camera moves along its forward-facing orientation vector N. That is, whichever direction the camera is currently facing, moving "forward" means to change the position C to a different location along that N axis. Note that in so doing, C changes, but N does not change.

A common mistake made by beginners is to assume that moving the camera forward means to move it along the world Z axis. While this is true for the initial camera orientation in the example shown earlier, after turning the camera it would no longer be correct. Instead, we need to move the camera along its own N orientation vector.

Moving the camera forward requires simple vector arithmetic. If we treat the camera position C as a vector (recalling that a location in space can be treated as a vector from the origin to that location), then simply adding C to N will move the camera forward, as illustrated in Figure D.1. Of course, we typically would want finer control over how far we move the camera forward, so the complete formula is:

$$C' = C + (a*N)$$

where C is the current camera position, N is the forward-facing camera orientation vector, a is a floating-point scalar indicating how much movement is desired, and C' is the resulting new camera position. Moving the camera backwards is nearly identical; we simply add -N to C, and the formula becomes $C' = C - (a*N)$. Functions for adding two vectors, and for multiplying a vector by a scalar, are available in the **JOML** math library.

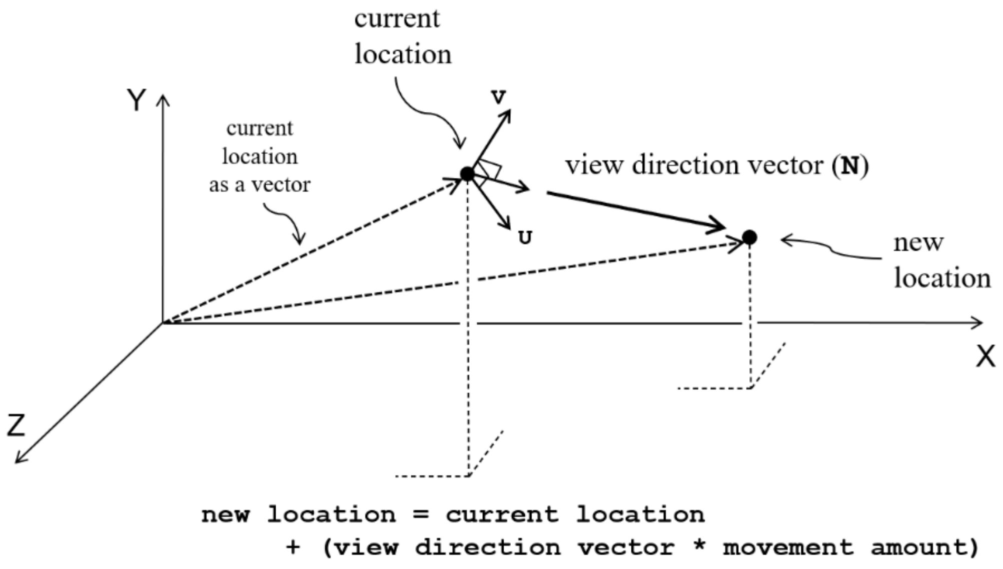
current location
current location as a vector
V
U
view direction vector (N)
new location
Y
X
Z
new location = current location
+ (view direction vector * movement amount)

Figure D.1 – Moving the camera forwards

Moving the camera right and left uses the same process, except that the right-facing camera orientation vector U is used. The formulas for moving right and left are C' = C + (a*U), and C' = C - (a*U), respectively. It is important to understand the difference between *moving* right, versus *turning* right. Moving right means sliding the camera to its right, in the direction of its right-facing U vector, *without changing the direction it is facing*. Turning right means to rotate the camera, changing its orientation so that it faces a different direction (we will learn how to do this shortly).

Similarly, moving the camera up and down would be via the formulas C' = C + (a*V), and C' = C - (a*V), respectively.

Enabling the end user to move the camera, such as in a video game, requires capturing user inputs and then calling the desired function. For example, it is common for video games to allow the player to "move forward" by pressing the "W" key. We can use Java key bindings to capture appropriate keystrokes, and call the corresponding functions described above to move the camera as desired.

D.1.3 Functions for Turning the Camera

Turning the camera refers to changing its orientation vectors U, V, and N, without altering its position C. The most common camera turning is *right* and *left*, but other turnings are possible. Turning "right" (or "left") means that the camera rotates around its upward-facing orientation vector V. That is, whichever direction the camera is currently facing, turning "right" means to change its right-facing U and forward-facing N orientation vectors so that the camera is pointing in a different direction. Note that in so doing, U and N change, but V (and C) do not. Also note that this operation must ensure that the orientation vectors U, V, and N remain orthogonal to each other, and that each remain unit length.

A common mistake made by beginners is to assume that turning the camera to the right means to rotate it around the world Y axis. While this is true for the initial camera orientation in the example shown earlier, for different orientations it would not be correct. Instead, we need to rotate the camera around its V orientation vector.

Turning the camera requires rotating two of its orientation axes around the third axis. For example, turning the camera right and left (also called "yaw") require rotating U and N around V. This is illustrated in Figure D.2. Similarly, turning the camera up and down (also called "pitch") requires rotating U and V around N, and "roll" is achieved by rotating V and N around U. In each case, we would want to specify the amount of rotation desired (in degrees or radians).

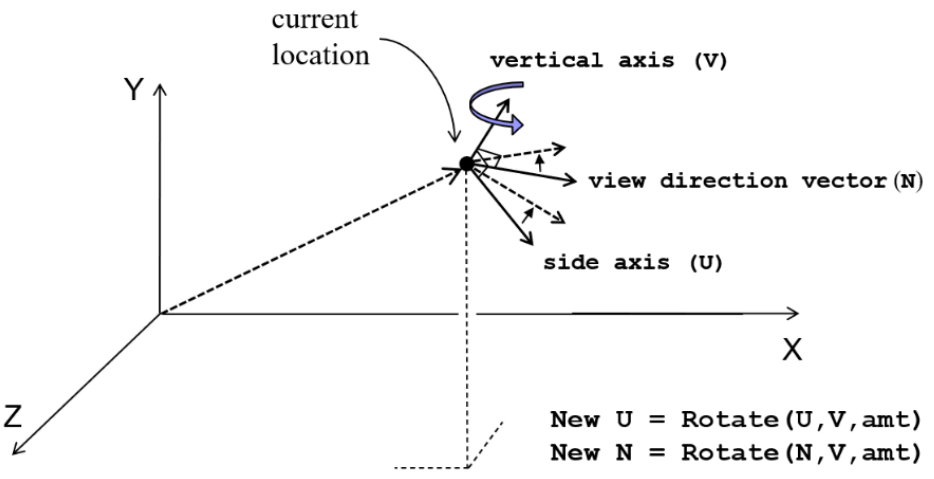

Figure D.2 – Turning the camera to the left

Functions for rotating a vector a specified amount around an axis (i.e., around another vector) are available in the **JOML** math library. Note that as long as we rotate the two vectors (U and N) around the third (V) *by the same amount*, the three camera orientation vectors will remain orthogonal to each other, and will retain their left-handed alignment. Also note that simply rotating a vector doesn't change its length, so they also retain their unit length.

As described earlier for moving the camera, it is common for video games to allow the player to "turn left" or "turn right", such as by pressing the arrow keys. So as described earlier, we can use Java to capture the desired keystrokes, and then call the appropriate turning functions just described to look around and view the scene from different directions.

D.1.4 Building the View Matrix

Now that we have seen the role of the C, U, V, and N vectors in controlling the position and orientation of the camera, building an associated view matrix is simply a matter of applying the formula in Figure 3.13 (and reproduced at the start of this appendix). The steps to do this are as follows:

1. Build the T matrix by inserting the negatives of C_x, C_y, and C_z into an identity matrix, in the rightmost column (thus forming a translation matrix for -C).
2. Build the R matrix by inserting U_x, U_y, U_z, V_x, V_y, V_z, and the negatives of N_x, N_y, and N_z, into an identity matrix, in the positions shown in Figure 3.13.
3. Concatenate (multiply) matrices R and T, forming the view matrix V.

Functions for each of these steps are available in the **JOML** math library.

A common error that beginners often make, is transposing the R matrix when building it. The T matrix can be built most easily by using one of the translation matrix constructors in **JOML**, which ensures that it will be built correctly. However, it is very easy to confuse rows and columns when building the R matrix. Another very common error is to concatenate R and T in the wrong order. They must be multiplied left-to-right as R*T (recall that matrix multiplication is not commutative).

Whereas moving and turning the camera are commonly associated with user controls, building the view matrix should happen automatically, typically at every frame. The function that builds the view matrix should therefore be called from display().[1]

[1] If camera movement is very infrequent, it could instead be called from the functions that perform the movement.

D.2 More About Camera Controllers

The controller described in Section D.1 may be adequate for the task of viewing a 3D scene from a variety of vantage points, and it is sometimes called a “flying camera.” For example, a CAD tool that enables the user to design a part for an automobile would need to enable the designer to view the part from various angles to ensure that it meets the desired specification. It also may be adequate for offline rendering, such as for a movie, where human interaction is not necessary.

However, video games (and video game engines) typically offer more sophisticated camera controllers. One important reason is that video games often involve a player “avatar”, complicating the desired point of view for the camera. For example, it is often useful to include the avatar in the player’s view, such as looking over the avatar’s shoulder. Or, they may wish to view the world from the avatar’s point of view. Two types of camera controllers commonly used in games are:

- *Orbit* or *Targeted* Camera Controller – Here, the camera always looks at the position of the avatar, and as the avatar moves and turns, the camera moves and turns accordingly. The end user is given controls for positioning the camera relative to the avatar, such as its distance from the avatar, and the camera’s elevation and angle of view. Implementation often makes use of spherical (polar) coordinates.
- *Chase* or *Tracking* Camera Controller – Here, the camera is always positioned behind and slightly above the avatar, appropriate for example in racing games. As the avatar moves, the camera “chases” it, allowing the end user to watch the avatar as well as the area immediately in front and beside the avatar, as well as other nearby competitors. Chase cameras typically also employ a “spring” system, so that sharp, sudden turns made by the avatar result in a more gradual change to the camera position, so as to avoid motion sickness in the player.

There are numerous online tutorials on how to implement various camera controllers. We also build a simple orbit camera controller in CSc-165.