Chapter 8

Viewing

> Where am I? What am I looking at?
>
> James F. Blinn

8.1 Viewing Coordinates

World coordinates provide a single coordinate system to place all of the objects in our scene. For example, if the scene is a room, then its origin might be in a corner of the room, and its units might be measured in feet or some other natural measurement. Our goal now is to create a view in that room, by placing the observer in this scene, determining the direction the observer is facing, and figuring our what scene geometry falls within the observer's field of view.

In order to figure our what geometry lies in front of an observer, we construct a new convenient coordinate system called *viewing coordinates.* There are many different conventions and constructions for viewing coordinates, but we will focus on a popular one used e.g. in OpenGL, that we will use as reference for the remainder of the book.

Following this convention, in viewing coordinates the observer is placed at the origin looking down the negative z axis. The negative z axis is chosen because we want the positive x axis to extend to the right of the viewer and the positive y axis to extend up. In order to maintain a right handed coordinate system, the negative z axis extends in the direction of viewing.

In this coordinate system, the observer "sees" geometry that lies somewhere near the negative z axis, and that geometry will eventually get projected onto a "window" parallel to the $x - y$ plane such that its x, y coordinates will correspond to its 2-D window coordinates. We will cover the specifics of that projection later, but for now that reasoning serves to justify the choice of viewing coordinate axes.

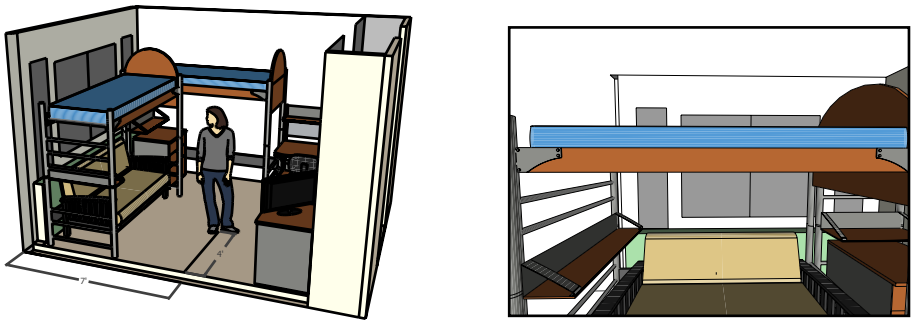

Figure 8.1: A room displayed in world coordinates (left) and viewing coordinates (right). Scene modeled by user "mobo" (Alexander Policht) rendered by Google Sketchup. Model used without permission.

Graphics systems, such as OpenGL, default to these viewing coordinates, displaying only geometry lying somewhere near the negative z axis in front of the observer at the origin. In the global coordinates of our room, the origin is at the corner and the negative z axis is where the floor meets the wall. Our default view would place the obsever at this corner looking down this floor-wall edge.

Hence we set up a view in world coordinates by placing the observer at a world coordinate position, and determining the direction the observer is looking. Then a viewing transformation can be implemented as a stage of the vertex pipeline that transforms scene vertices from their world coordinates positions to their viewing coordinates positions. Any vertices near the observer's line of sight in world coordinates would end up near the negative z axis in viewing coordinates.

For example in Figure 8.1, our observer is located at the point $(7, 4, 5)$ in world coordinates (seven feet East, four feet North and five feet up from the corner of the room) and is looking West (in the negative x direction). Note that in this Google Sketchup example, x and y are horizontal and z is the vertical axis, colored by red, green and blue respectively. We need a viewing transformation that moves the eye to the origin, and rotates the viewing direction from negative x (West) to negative z, and also rotated positive z (the world coordinates up direction) to positive y (the viewing coordinates vertical axis). This same transformation will reposition scene vertices just West of the observer in world coordinates to positions near the negative z axis in viewing coordinates so they appear in the viewing window.

In this example, the viewing transformation first translates the oberserver to the origin, using a translation by $(-7, -4, -5)$. Then the viewing transformation would need to rotate the view direction from the $-x$ direction, e.g. indicated by the direction vector $(-1, 0, 0)$ to the $-z$ direction, e.g.

$(0, 0, -1)$. This would be a rotation about the y axis by $-90°$. We then need to rotate 90° about the view direction -z (note this is a $-90°$ rotation about $+z$) so that the world coordinate up direction (originally $+z$, but after the y-rotation it becomes $-x$) to the view coordinate vertical direction $+y$.

Hence our viewing transformation V is composed of a translation by $(-7, -4, -5)$ following by a $-90°$ rotation about y, then a $-90°$ rotation about z, yielding

$$V = \begin{bmatrix} & 1 & & \\ -1 & & & \\ & & 1 & \\ & & & 1 \end{bmatrix} \begin{bmatrix} & & -1 & \\ & 1 & & \\ 1 & & & \\ & & & 1 \end{bmatrix} \begin{bmatrix} 1 & & & -7 \\ & 1 & & -4 \\ & & 1 & -5 \\ & & & 1 \end{bmatrix} \quad (8.1)$$

$$= \begin{bmatrix} & 1 & & -4 \\ & & 1 & -5 \\ 1 & & & -7 \\ & & & 1 \end{bmatrix}. \quad (8.2)$$

We can check this result by applying it to various points in world coordinates to see if it transforms them to their expected locations in viewing coordinates. Indeed V transforms the (homogeneous) world coordinate eye point $[7, 4, 5, 1]^T$ to the viewing coordinates origin $[0, 0, 0, 1]^T$. It also transforms a point one unit west along the line of sight $[6, 4, 5, 1]^T$ to a corresponding distance along the negative z axis $[0, 0, -1, 1]$ in viewing coordinates. Finally, it transforms a point one unit above the viewer $[7, 4, 6, 1]^T$ to the viewing coordinates point $[0, 1, 0, 1]$, which ensures that we have oriented our view such that the world coordinate up direction $(0, 0, 1)$ is oriented in our viewing coordinates to be the vertical direction $(0, 1, 0)$.

Figure 8.1 right demonstrates a (perspective) rendering of the view, obtained by plotted the x and y coordinates after the viewing transformation. (This particular image results after a perspective distortion we study in the next chapter.) Notice that the viewing transformation takes 3-D global coordinate vertices to 3-D viewing coordinate vertices. Even though we want the x and y coordinates to eventually indicate screen position for our vertices, we still need to keep track of the (negative) z coordinates of what we can see. We use this in the next chapter for perspective, and later for determining which portions of screen polygons are occluded or visible.

8.2 Lookat

Most graphics libraries create the change of coordinates from world coordinates to viewing coordinates by specifying a "lookat" transformation. The specification of a lookat transformation consists of two 3-D points and a direction vector.

The first parameter of this specification is the position of the eye in world coordinates. We refer to this 3-D world coordinates position as the eye point **eye**. The lookat transformation will map this world coordinate position to the origin in viewing coordinates.

The second parameter of the lookat specification is a "lookat" point in world coordinates that the viewer is focused on. We refr to this 3-D world coordinate position as the lookat point **at**. The lookat transformation will map this world coordinate position **at** to some location along the $-\mathbf{z}$ axis in viewing coordinates.

We will define a unit length view vector[1]

$$\mathbf{v} = (\mathbf{at} - \mathbf{eye})/||\mathbf{at} - \mathbf{eye}|| \tag{8.3}$$

which will map to the $-z$ axis in viewing coordinates.

Notice that the specification of an eye point and a lookat point is not enough to specify a unique view, because this view can be rotated by an arbitrary orientation around the view vector.

The third parameter is a direction vector indicating the "up" direction in world coordinates. This "up" vector **up** will be used to define which orientation about the view vector should be used. The lookat transformation will choose the orientation that minimizes the difference between the viewing coordinate vertical direction (the $+\mathbf{y}$ axis) and the direction that the up vector **up** is mapped to by the lookat transformation.

Hence the lookat transformation takes the parameters $\mathbf{eye}, \mathbf{at}$ and **up** and produces a viewing transformation L that specifies a view from the eyepoint **eye** looking at **at** oriented so that the vector **up** would be pointing up vertically in the view.

8.3 The Flight Simulator

For a flight simulator, we set up a view from the pilot's position looking forward through the windshield in the direction the plane is flying. In world coordinates, we define an orthonormal coordinate system centered at the plane position P, with a wing axis W extending down the left wing, a plane vertical axis U extending through the top of the plane, and a heading vector H through the nose of the plane.

We want to create a correspondence between these plane axes W, U, H and the viewing coordinate axes $\mathbf{x}, \mathbf{y}, \text{-}z$. The viewing transform V should translate P to the origin and rotate W to $\mathbf{x}, U$ to $\mathbf{y}$ and H to $\text{-}z$. Hence in world coordinates we are at point P looking in the H direction, but in viewing coordinates we are at the origin looking in the $-\mathbf{z}$ direction.

First we will look at the flight and controls of the plane in world coordinates. We want our plane to fly in its heading direction, so each step in

[1]This "view vector" $mathbfv$ is in the opposite direction of the "view vector" $\mathbf{v}$ used in Chapter **??** for lighting calculations.

Figure 8.2: The view from eyepoint $(3, 2, 2)$ looking at $(2, 1, 3)$ with up direction $(0, 1, 0)$ all in world coordinates, transformed to viewing coordinates by the lookat transformation.

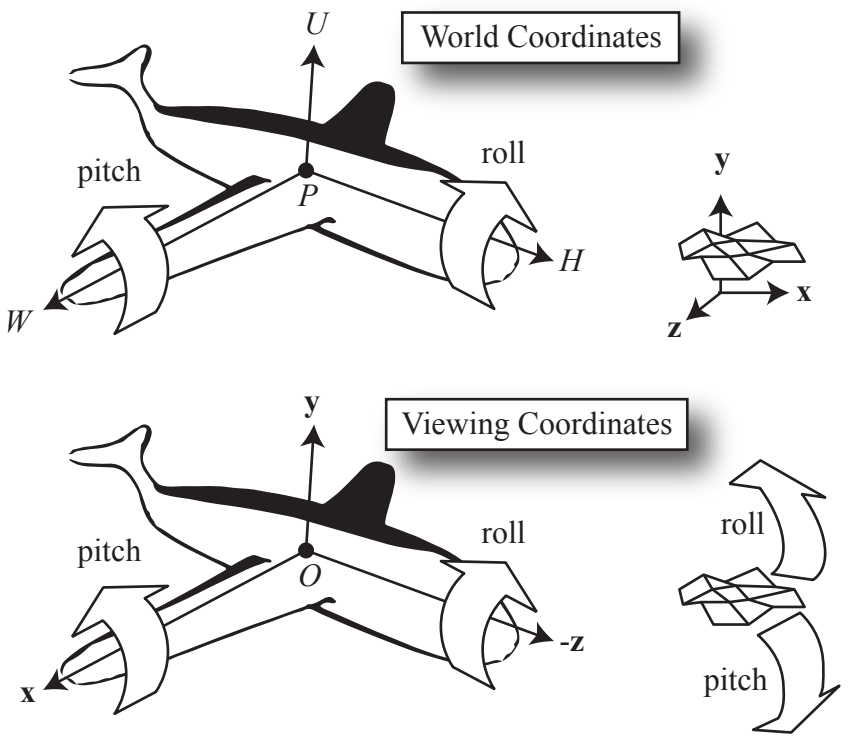

Figure 8.3: World (top) and viewing (bottom) coordinate systems for a flight simulator.

our simulation adds a small amount of the heading direction to the plane's position

$$P = P + sH, \tag{8.4}$$

where s is the speed of the plane, measured in world spatial units per frame time. For example, if we update the screen 60 times per second, and our spatial units are meters, then if we want out plane to fly 225 meters per second (around 500 mph, typical for a passenger jet) then this update rate should be $s = 225/60 = 3.75$ meters per screen update.

We control the plane by changing its heading direction. If we move the control stick left we want the plane to roll left, which is a positive rotation about the plane's heading axis H in world coordinates. Likewise moving the control stick right rolls the plane right, a negative rotation about its heading axis. If we pull the stick back, then we want the plane to pitch up, which is a positive rotation about the plane's wing axis W in world coordinates. Pushing the stick forward pitches the plane down, a negative rotation about the wing axis.

When we simulate these maneuvers, we do not actually rotate the plane. Instead, we apply the opposite maneuver to the vertices in our scene. A roll left by an angle θ would be specified by the matrix $R_{P,H,\theta}$ implementing a positive rotation by θ about the line parallel to vector H passing through point P. But to simulate the plane rolling left we need to rotate the vertices in the scene in the opposite direction around the plane's heading axis. Hence if V is the current viewing transformation and the scene vertices are $\begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$, then the viewing coordinates $\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix}$ of these vertices after an airplane roll of θ would be

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = V R_{P,H,-\theta} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}. \tag{8.5}$$

If we want to simulate these maneuvers in world coordinates, we need to update the plane axes W, U, H. When the plane rolls, we need to rotate the W and U axes around the H axis,

$$W = R_{P,H,\theta}(W), \tag{8.6}$$

$$U = R_{P,H,\theta}(U). \tag{8.7}$$

When the plane pitches, we need to rotate the U and H axes around the W axis,

$$U = R_{P,W,\theta}(U), \tag{8.8}$$

$$H = R_{P,W,\theta}(H). \tag{8.9}$$

However, there is an easier way using viewing coordinates.

In viewing coordinates, the plane is at the origin pointing down the $-\mathbf{z}$ axis with its right wing along the $\mathbf{x}$ axis. In this coordinate system, pitching up is simply a positive rotation about the $\mathbf{x}$ axis and rolling left is a positive rotation about the $-\mathbf{z}$ axis (a negative rotation about the $\mathbf{z}$ axis). But how do we specify transformations in the viewing coordinate system?

Recall that in OpenGL and other tranformational geometry systems, the order that transformations are specified in a program is the order they are multiplied, from left to right. For example, OpenGL maintains a modelview matrix $MV = VM$ where V is a view transformation and M is a modeling transformation. Since we are specifying our terrain directly in world coordinates, we can ignore the modeling part of the modelview transformation and simply refer to it as the viewing transformation V.

Let's assume this modelview matrix currently holds a viewing transformation $MV = V$ from our plane position looking in its heading direction. Recall V maps the plane world-coordinate position P to the origin and its heading direction H to the $-\mathbf{z}$ axis. To fly the plane in world coordinates, we need to translate the view position P a little bit in its heading direction H. In viewing coordinates, we want to translate the terrain geometry a little bit in the $+\mathbf{z}$ direction. In viewing coordinates, the plane is always at the origin pointing down the $-\mathbf{z}$ axis, and we fly by moving the terrain past the plane (and we turn by rotating the terrain about these viewing coordinate axes). Hence in viewing coordinates, we fly by applying a translation by s in the $\mathbf{z}$ direction, which we denote $T_{s,\mathbf{z}}$.

Where does this translation go? If we apply Tsz it will be multiplied on the right of the current modelview matrix, and we get $MV = VT_{s,\mathbf{z}}$, which will move the terrain in the $\mathbf{z}$ direction of the world, regardless of which direction the plane is oriented. To translate in viewing coordinates, we need to set $MV = T_{s,\mathbf{z}}V$, but the system multiples transformation on the right, so if $MV = V$, how do we specify $T_{s,\mathbf{z}}$ so it is multiplied on the left?

In OpenGL, the command `glGetFloatv(GL_MODELVIEW_MATRIX,m)` will load `float m[4][4]` with the modelview matrix. Furthermore, the command `glMatrixf(m)` will multiple the matrix `m` on the right side of its current modelview matrix. Hence the following code fragment

```
glGetFloatv(GL_MODELVIEW_MATRIX,mv);
glLoadIdentity();
glTranslatef(0,0,s);
glMultMatrixf(mv);
```

will multiple the translation $T_{s,\mathbf{z}}$ on the *left side* of the ModelView matrix MV. The first line implements $mv = MV$. It stores a copy of OpenGL's current ModelView matrix in the array mv. The second line implements

$MV = I$, replacing OpenGL's modelview matrix with the identity. The third line implements $MV = MVT_{s,\mathbf{z}} = IT_{s,\mathbf{z}} = T_{s,\mathbf{z}}$ which loads OpenGL's ModelView matrix with the product of its ModelView matrix and the translation matrix $T_{s,\mathbf{z}}$, and since (from the previous step) the ModelView matrix MV was just the identity I, this ModelView matrix now consists of only this translation. The last line implements $MV = MVmv$ but since $MV = T_{s,\mathbf{z}}$ (from the previous step), it implements $MV = Tmv$. Hence after these steps, we have applied a transformation that is multiplied on the left side of the ModelView matrix.

Hence, flying in our heading direction is implemented by a translation in $\mathbf{z}$ in viewing coordinates. Our other plane controls such as roll and pitch are similarly implemented as small rotations about coordinate axes in viewing coordinates. (The amount of the small rotation angle controls the angular velocity of the plane's rotation.) These rotations too must be multiplied on the left side of the viewing transformation as specified above. I will leave it as an exercise to figure out which rotaton axis implements which control, and what the sign of the angle should be.