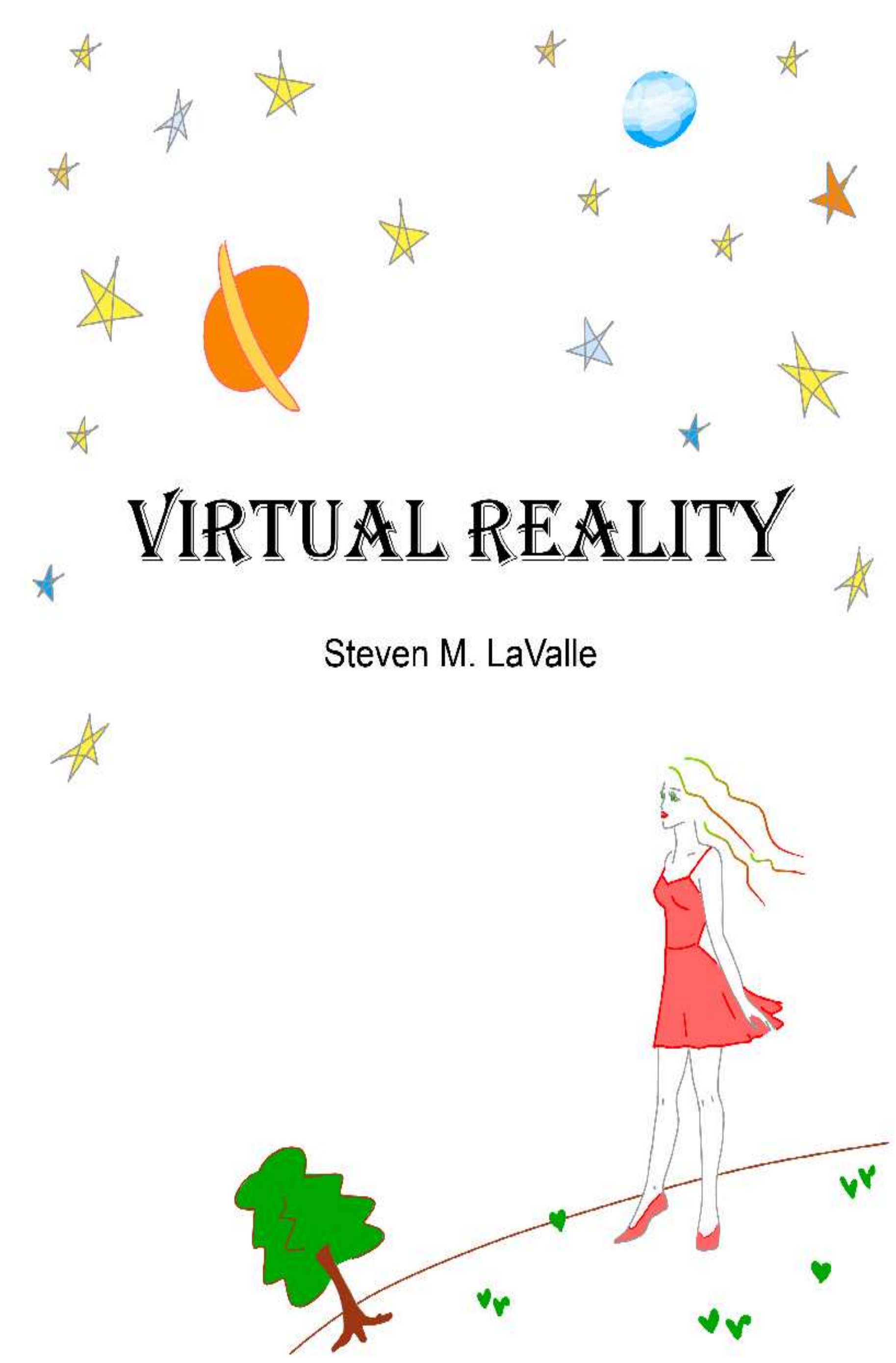


Chapter 3

The Geometry of Virtual Worlds

Steven M. LaValle

University of Oulu

Chapter 3

The Geometry of Virtual Worlds

Section 2.2 introduced the Virtual World Generator (VWG), which maintains the geometry and physics of the virtual world. This chapter covers the *geometry* part, which is needed to make models and move them around. The models could include the walls of a building, furniture, clouds in the sky, the user's avatar, and so on. Section 3.1 covers the basics of how to define consistent, useful models. Section 3.2 explains how to apply mathematical transforms that move them around in the virtual world. This involves two components: Translation (changing position) and rotation (changing orientation). Section 3.3 presents the best ways to express and manipulate 3D rotations, which are the most complicated part of moving models. Section 3.4 then covers how the virtual world appears if we try to "look" at it from a particular perspective. This is the geometric component of visual rendering, which is covered in Chapter 7. Finally, Section 3.5 puts all of the transformations together, so that you can see how to go from defining a model to having it appear in the right place on the display.

If you work with high-level engines to build a VR experience, then most of the concepts from this chapter might not seem necessary. You might need only to select options from menus and write simple scripts. However, an understanding of the basic transformations, such as how to express 3D rotations or move a camera viewpoint, is essential to making the software do what you want. Furthermore, if you want to build virtual worlds from scratch, or at least want to *understand* what is going on under the hood of a software engine, then this chapter is critical.

3.1 Geometric Models

We first need a virtual world to contain the geometric models. For our purposes, it is enough to have a 3D Euclidean space with Cartesian coordinates. Therefore, let $\mathbb{R}^3$ denote the virtual world, in which every point is represented as a triple of real-valued coordinates: (x, y, z). The coordinate axes of our virtual world are shown in Figure 3.1. We will consistently use right-handed coordinate systems in this book because they represent the predominant choice throughout physics and

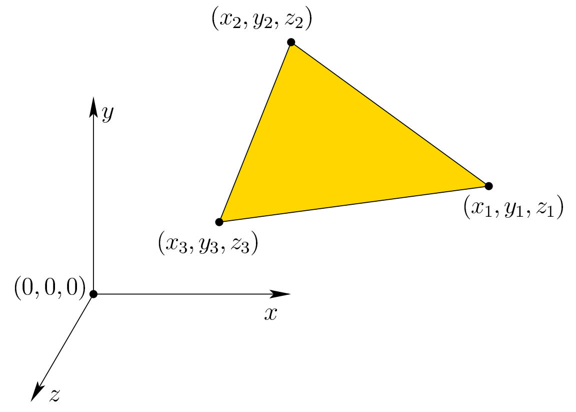


Figure 3.1: Points in the virtual world are given coordinates in a right-handed coordinate system in which the y axis is pointing upward. The origin $(0, 0, 0)$ lies at the point where axes intersect. Also shown is a 3D triangle is defined by its three vertices, each of which is a point in $\mathbb{R}^3$.

engineering; however, left-handed systems appear in some places, with the most notable being Microsoft's DirectX graphical rendering library. In these cases, one of the three axes points in the opposite direction in comparison to its direction in a right-handed system. This inconsistency can lead to hours of madness when writing software; therefore, be aware of the differences and their required conversions if you mix software or models that use both. If possible, avoid mixing right-handed and left-handed systems altogether.

Geometric models are made of surfaces or solid regions in $\mathbb{R}^3$ and contain an infinite number of points. Because representations in a computer must be finite, models are defined in terms of *primitives* in which each represents an infinite set of points. The simplest and most useful primitive is a *3D triangle*, as shown in Figure 3.1. A planar surface patch that corresponds to all points "inside" and on the boundary of the triangle is fully specified by the coordinates of the triangle *vertices*:

$$((x_1, y_1, z_1), (x_2, y_2, z_2), (x_3, y_3, z_3)). \tag{3.1}$$

To model a complicated object or body in the virtual world, numerous triangles can be arranged into a *mesh*, as shown in Figure 3.2. This provokes many important questions:

1. How do we specify how each triangle "looks" whenever viewed by a user in VR?

2. How do we make the object "move"?

3. If the object surface is sharply curved, then should we use curved primitives,

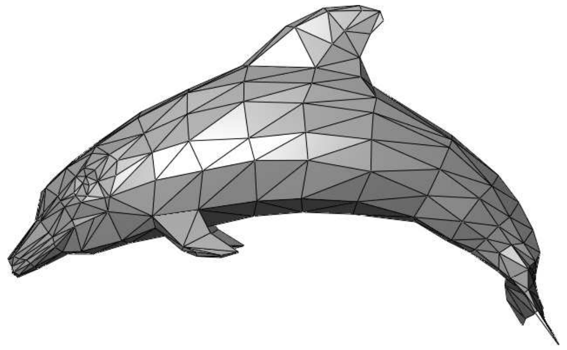


Figure 3.2: A geometric model of a dolphin, formed from a mesh of 3D triangles (from Wikipedia user Chrschn).

rather than trying to approximate the curved object with tiny triangular patches?

4. Is the interior of the object part of the model, or does the object consist only of its surface?

5. Is there an efficient algorithm for determining which triangles are adjacent to a given triangle along the surface?

6. Should we avoid duplicating vertex coordinates that are common to many neighboring triangles?

We address these questions in reverse order.

Data structures Consider listing all of the triangles in a file or memory array. If the triangles form a mesh, then most or all vertices will be shared among multiple triangles. This is clearly a waste of space. Another issue is that we will frequently want to perform operations on the model. For example, after moving an object, can we determine whether it is in collision with another object (covered in Section 8.3)? A typical low-level task might be to determine which triangles share a common vertex or edge with a given triangle. This might require linearly searching through the triangle list to determine whether they share a vertex or two. If there are millions of triangles, which is not uncommon, then it would cost too much to perform this operation repeatedly.

For these reasons and more, geometric models are usually encoded in clever data structures. The choice of the data structure should depend on which operations will be performed on the model. One of the most useful and common is the *doubly connected edge list*, also known as *half-edge data structure* [2, 10]. See Figure 3.3. In this and similar data structures, there are three kinds of data elements: *faces*, *edges*, and *vertices*. These represent two, one, and zero-dimensional

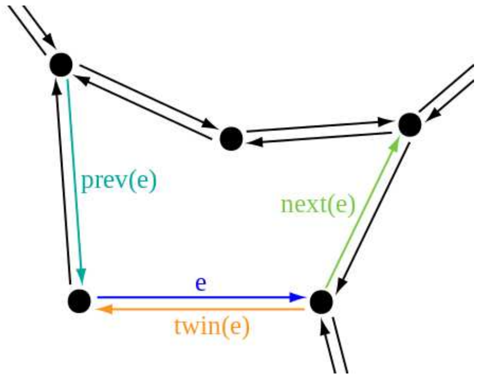


Figure 3.3: Part of a doubly connected edge list is shown here for a face that has five edges on its boundary. Each half-edge structure e stores pointers to the next and previous edges along the face boundary. It also stores a pointer to its twin half-edge, which is part of the boundary of the adjacent face. (Figure from Wikipedia author Nico Korn.)

parts, respectively, of the model. In our case, every face element represents a triangle. Each edge represents the border of one or two triangles, without duplication. Each vertex is shared between one or more triangles, again without duplication. The data structure contains pointers between adjacent faces, edges, and vertices so that algorithms can quickly traverse the model components in a way that corresponds to how they are connected together.

Inside vs. outside Now consider the question of whether the object interior is part of the model (recall Figure 3.2). Suppose the mesh triangles fit together perfectly so that every edge borders exactly two triangles and no triangles intersect unless they are adjacent along the surface. In this case, the model forms a complete barrier between the *inside* and *outside* of the object. If we were to hypothetically fill the inside with a gas, then it could not leak to the outside. This is an example of a *coherent model*. Such models are required if the notion of inside or outside is critical to the VWG. For example, a penny could be inside of the dolphin, but not intersecting with any of its boundary triangles. Would this ever need to be detected? If we remove a single triangle, then the hypothetical gas would leak out. There would no longer be a clear distinction between the inside and outside of the object, making it difficult to answer the question about the penny and the dolphin. In the extreme case, we could have a single triangle in space. There is clearly no natural inside or outside. At an extreme, the model could be as bad as *polygon soup*, which is a jumble of triangles that do not fit together nicely and could even have intersecting interiors. In conclusion, be careful when constructing models so that the operations you want to perform later will be logically clear.

If you are using a high-level design tool, such as Blender or Maya, to make your models, then coherent models will be automatically built.

Why triangles? Continuing upward through the questions above, triangles are used because they are the simplest for algorithms to handle, especially if implemented in hardware. GPU implementations tend to be biased toward smaller representations so that a compact list of instructions can be applied to numerous model parts in parallel. It is certainly possible to use more complicated primitives, such as quadrilaterals, splines, and semi-algebraic surfaces [3, 4, 9]. This could lead to smaller model sizes, but often comes at the expense of greater computational cost for handling each primitive. For example, it is much harder to determine whether two spline surfaces are colliding, in comparison to two 3D triangles.

Stationary vs. movable models There will be two kinds of models in the virtual world, which is embedded in $\mathbb{R}^3$:

- *Stationary models*, which keep the same coordinates forever. Typical examples are streets, floors, and buildings.
- *Movable models*, which can be *transformed* into various positions and orientations. Examples include vehicles, avatars, and small furniture.

Motion can be caused in a number of ways. Using a tracking system (Chapter 9), the model might move to match the user's motions. Alternatively, the user might operate a controller to move objects in the virtual world, including a representation of himself. Finally, objects might move on their own according to the laws of physics in the virtual world. Section 3.2 will cover the mathematical operations that move models to the their desired places, and Chapter 8 will describe velocities, accelerations, and other physical aspects of motion.

Choosing coordinate axes One often neglected point is the choice of coordinates for the models, in terms of their placement and scale. If these are defined cleverly at the outset, then many tedious complications can be avoided. If the virtual world is supposed to correspond to familiar environments from the real world, then the axis scaling should match common units. For example, $(1, 0, 0)$ should mean *one meter* to the right of $(0, 0, 0)$. It is also wise to put the origin $(0, 0, 0)$ in a convenient location. Commonly, $y = 0$ corresponds to the floor of a building or sea level of a terrain. The location of $x = 0$ and $z = 0$ could be in the center of the virtual world so that it nicely divides into quadrants based on sign. Another common choice is to place it in the upper left when viewing the world from above so that all x and z coordinates are nonnegative. For movable models, the location of the origin and the axis directions become extremely important because they affect how the model is rotated. This should become clear in Sections 3.2 and 3.3 as we present rotations.

Viewing the models Of course, one of the most important aspects of VR is how the models are going to "look" when viewed on a display. This problem is divided into two parts. The first part involves determining where the points in the virtual world should appear on the display. This is accomplished by viewing transformations in Section 3.4, which are combined with other transformations in Section 3.5 to produce the final result. The second part involves how each part of the model should appear after taking into account lighting sources and surface properties that are defined in the virtual world. This is the rendering problem, which is covered in Chapter 7.

3.2 Changing Position and Orientation

Suppose that a movable model has been defined as a mesh of triangles. To move it, we apply a single transformation to every vertex of every triangle. This section first considers the simple case of *translation*, followed by the considerably complicated case of *rotations*. By combining translation and rotation, the model can be placed anywhere, and at any orientation in the virtual world $\mathbb{R}^3$.

Translations Consider the following 3D triangle,

$$((x_1, y_1, z_1), (x_2, y_2, z_2), (x_3, y_3, z_3)), \tag{3.2}$$

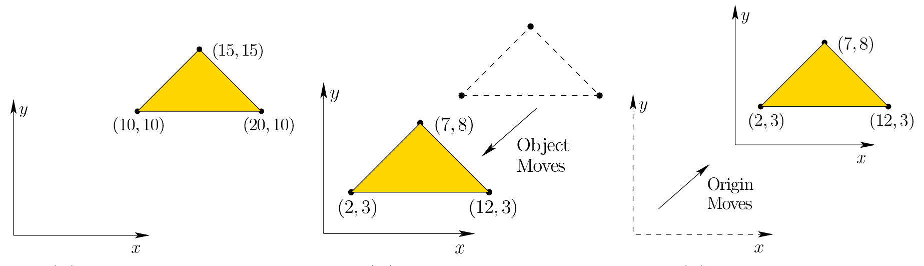
in which its vertex coordinates are expressed as generic constants.

Let x_t, y_t, and z_t be the amount we would like to change the triangle's position, along the x, y, and z axes, respectively. The operation of changing position is called *translation*, and it is given by

$$\begin{aligned} (x_1, y_1, z_1) &\mapsto (x_1 + x_t, y_1 + y_t, z_1 + z_t) \\ (x_2, y_2, z_2) &\mapsto (x_2 + x_t, y_2 + y_t, z_2 + z_t) \\ (x_3, y_3, z_3) &\mapsto (x_3 + x_t, y_3 + y_t, z_3 + z_t), \end{aligned} \tag{3.3}$$

in which $a \mapsto b$ denotes that a becomes replaced by b after the transformation is applied. Applying (3.3) to every triangle in a model will translate all of it to the desired location. If the triangles are arranged in a mesh, then it is sufficient to apply the transformation to the vertices alone. All of the triangles will retain their size and shape.

Relativity Before the transformations become too complicated, we want to caution you about interpreting them correctly. Figures 3.4(a) and 3.4(b) show an example in which a triangle is translated by $x_t = -8$ and $y_t = -7$. The vertex coordinates are the same in Figures 3.4(b) and 3.4(c). Figure 3.4(b) shows the case we are intended to cover so far: The triangle is interpreted as having moved in the virtual world. However, Figure 3.4(c) shows another possibility: The coordinates of the virtual world have been reassigned so that the triangle is



(a) Original object (b) Object moves (c) Origin moves

Figure 3.4: Every transformation has two possible interpretations, even though the math is the same. Here is a 2D example, in which a triangle is defined in (a). We could translate the triangle by $x_t = -8$ and $y_t = -7$ to obtain the result in (b). If we instead wanted to hold the triangle fixed but move the origin up by 8 in the x direction and 7 in the y direction, then the coordinates of the triangle vertices change the exact same way, as shown in (c).

closer to the origin. This is equivalent to having moved the entire world, with the triangle being the only part that does not move. In this case, the translation is applied to the coordinate axes, but they are negated. When we apply more general transformations, this extends so that transforming the coordinate axes results in an *inverse* of the transformation that would correspondingly move the model. Negation is simply the inverse in the case of translation.

Thus, we have a kind of "relativity": Did the object move, or did the whole world move around it? This idea will become important in Section 3.4 when we want to change viewpoints. If we were standing at the origin, looking at the triangle, then the result would appear the same in either case; however, if the origin moves, then we would move with it. A deep perceptual problem lies here as well. If we perceive ourselves as having moved, then VR sickness might increase, even though it was the object that moved. In other words, our brains make their best guess as to which type of motion occurred, and sometimes get it wrong.

Getting ready for rotations How do we make the wheels roll on a car? Or turn a table over onto its side? To accomplish these, we need to change the model's *orientation* in the virtual world. The operation that changes the orientation is called *rotation*. Unfortunately, rotations in three dimensions are much more complicated than translations, leading to countless frustrations for engineers and developers. To improve the clarity of 3D rotation concepts, we first start with a simpler problem: 2D linear transformations.

Consider a 2D virtual world, in which points have coordinates (x, y). You can imagine this as a vertical plane in our original, 3D virtual world. Now consider a

generic two-by-two matrix

$$M = \begin{bmatrix} m_{11} & m_{12} \\ m_{21} & m_{22} \end{bmatrix} \tag{3.4}$$

in which each of the four entries could be any real number. We will look at what happens when this matrix is multiplied by the point (x, y), when it is written as a column vector.

Performing the multiplication, we obtain

$$\begin{bmatrix} m_{11} & m_{12} \\ m_{21} & m_{22} \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} x' \\ y' \end{bmatrix}, \tag{3.5}$$

in which (x', y') is the transformed point. Using simple algebra, the matrix multiplication yields

$$\begin{aligned} x' &= m_{11}x + m_{12}y \\ y' &= m_{21}x + m_{22}y. \end{aligned} \tag{3.6}$$

Using notation as in (3.3), M is a transformation for which $(x, y) \mapsto (x', y')$.

Applying the 2D matrix to points Suppose we place two points $(1, 0)$ and $(0, 1)$ in the plane. They lie on the x and y axes, respectively, at one unit of distance from the origin $(0, 0)$. Using vector spaces, these two points would be the standard unit basis vectors (sometimes written as $\hat{\imath}$ and $\hat{\jmath}$). Watch what happens if we substitute them into (3.5):

$$\begin{bmatrix} m_{11} & m_{12} \\ m_{21} & m_{22} \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} m_{11} \\ m_{21} \end{bmatrix} \tag{3.7}$$

and

$$\begin{bmatrix} m_{11} & m_{12} \\ m_{21} & m_{22} \end{bmatrix} \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \begin{bmatrix} m_{12} \\ m_{22} \end{bmatrix}. \tag{3.8}$$

These special points simply select the column vectors on M. What does this mean? If M is applied to transform a model, then each column of M indicates precisely how each coordinate axis is changed.

Figure 3.5 illustrates the effect of applying various matrices M to a model. Starting with the upper right, the identity matrix does not cause the coordinates to change: $(x, y) \mapsto (x, y)$. The second example causes a flip as if a mirror were placed at the y axis. In this case, $(x, y) \mapsto (-x, y)$. The second row shows examples of scaling. The matrix on the left produces $(x, y) \mapsto (2x, 2y)$, which doubles the size. The matrix on the right only stretches the model in the y direction, causing an *aspect ratio* distortion. In the third row, it might seem that the matrix on the left produces a mirror image with respect to both x and y axes. This is true, except that the mirror image of a mirror image restores the original. Thus, this corresponds to the case of a 180-degree (π radians) rotation, rather than a mirror image. The matrix on the right produces a shear along the x direction: $(x, y) \mapsto (x + y, y)$. The amount of displacement is proportional to y.

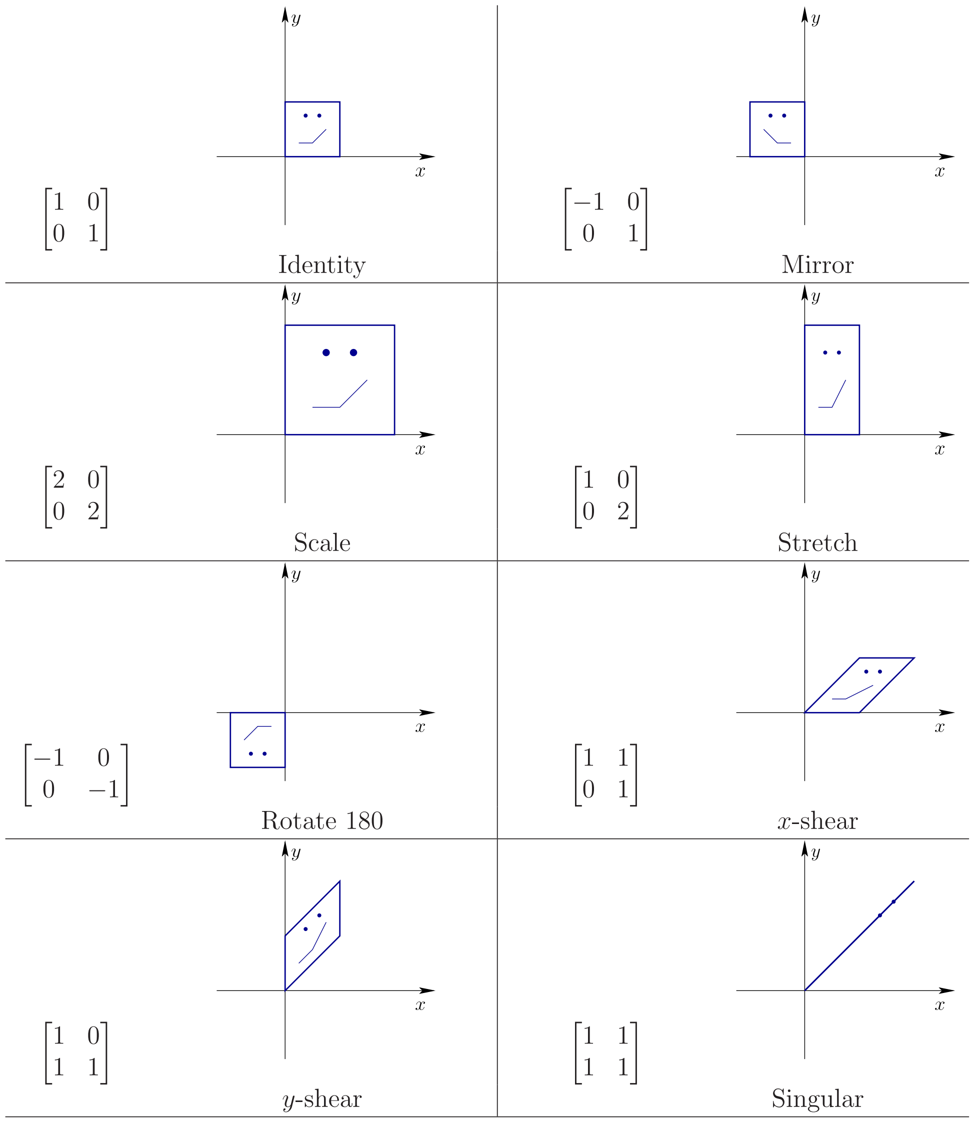

$\begin{bmatrix}1 & 0\\0 & 1\end{bmatrix}$
Identity
$\begin{bmatrix}-1 & 0\\0 & 1\end{bmatrix}$
Mirror
$\begin{bmatrix}2 & 0\\0 & 2\end{bmatrix}$
Scale
$\begin{bmatrix}1 & 0\\0 & 2\end{bmatrix}$
Stretch
$\begin{bmatrix}-1 & 0\\0 & -1\end{bmatrix}$
Rotate 180
$\begin{bmatrix}1 & 1\\0 & 1\end{bmatrix}$
x-shear
$\begin{bmatrix}1 & 0\\1 & 1\end{bmatrix}$
y-shear
$\begin{bmatrix}1 & 1\\1 & 1\end{bmatrix}$
Singular
x
y

Figure 3.5: Eight different matrices applied to transform a square face. These examples nicely cover all of the possible cases, in a qualitative sense.

In the bottom row, the matrix on the left shows a skew in the y direction. The final matrix might at first appear to cause more skewing, but it is degenerate. The two-dimensional shape collapses into a single dimension when M is applied: $(x, y) \mapsto (x + y, x + y)$. This corresponds to the case of a *singular* matrix, which means that its columns are not linearly independent (they are in fact identical). A matrix is singular if and only if its determinant is zero.

Only some matrices produce rotations The examples in Figure 3.5 span the main qualitative differences between various two-by-two matrices M. Two of them were rotation matrices: the identity matrix, which is 0 degrees of rotation, and the 180-degree rotation matrix. Among the set of all possible M, which ones are valid rotations? We must ensure that the model does not become distorted. This is achieved by ensuring that M satisfies the following rules:

1. No stretching of axes.
2. No shearing.
3. No mirror images.

If none of these rules is violated, then the result is a rotation.

To satisfy the first rule, the columns of M must have unit length:

$$m_{11}^2 + m_{21}^2 = 1 \text{ and } m_{12}^2 + m_{22}^2 = 1. \tag{3.9}$$

The scaling and shearing transformations in Figure 3.5 violated this.

To satisfy the second rule, the coordinate axes must remain perpendicular. Otherwise, shearing occurs. Since the columns of M indicate how axes are transformed, the rule implies that their inner (dot) product is zero:

$$m_{11}m_{12} + m_{21}m_{22} = 0. \tag{3.10}$$

The shearing transformations in Figure 3.5 violate this rule, which clearly causes right angles in the model to be destroyed.

Satisfying the third rule requires that the determinant of M is positive. After satisfying the first two rules, the only possible remaining determinants are 1 (the normal case) and -1 (the mirror-image case). Thus, the rule implies that:

$$\det \begin{bmatrix} m_{11} & m_{12} \\ m_{21} & m_{22} \end{bmatrix} = m_{11}m_{22} - m_{12}m_{21} = 1. \tag{3.11}$$

The mirror image example in Figure 3.5 results in $\det M = -1$.

The first constraint (3.9) indicates that each column must be chosen so that its components lie on a unit circle, centered at the origin. In standard planar coordinates, we commonly write the equation of this circle as $x^2 + y^2 = 1$. Recall

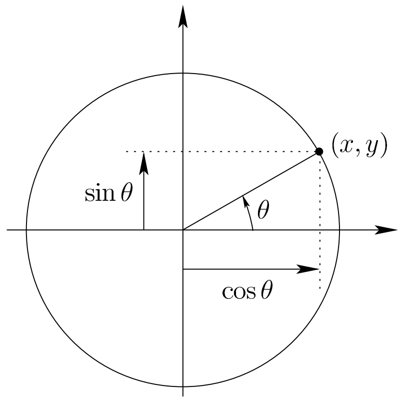


Figure 3.6: For a circle with unit radius, centered at the origin, a single parameter θ reaches all xy points along the circle as it ranges from $\theta = 0$ to $\theta = 2\pi$.

the common parameterization of the unit circle in terms of an angle θ that ranges from 0 to 2π radians (see Figure 3.6):

$$x = \cos\theta \text{ and } y = \sin\theta. \tag{3.12}$$

Instead of x and y, we use the notation of the matrix components. Let $m_{11} = \cos\theta$ and $m_{21} = \sin\theta$. Substituting this into M from (3.4) yields

$$\begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}, \tag{3.13}$$

in which m_{12} and m_{22} were uniquely determined by applying (3.10) and (3.11). By allowing θ to range from 0 to 2π, the full range of all allowable rotations is generated.

Think about degrees of freedom. Originally, we could chose all four components of M independently, resulting in 4 DOFs. The constraints in (3.9) each removed a DOF. Another DOF was removed by (3.10). Note that (3.11) does not reduce the DOFs; it instead eliminates exactly half of the possible transformations: The ones that are mirror flips and rotations together. The result is one DOF, which was nicely parameterized by the angle θ. Furthermore, we were lucky that set of all possible 2D rotations can be nicely interpreted as points along a unit circle.

The 3D case Now we try to describe the set of all 3D rotations by following the same general template as the 2D case. The matrix from (3.4) is extended from 2D to 3D, resulting in 9 components:

$$M = \begin{bmatrix} m_{11} & m_{12} & m_{13} \\ m_{21} & m_{22} & m_{23} \\ m_{31} & m_{32} & m_{33} \end{bmatrix}. \tag{3.14}$$

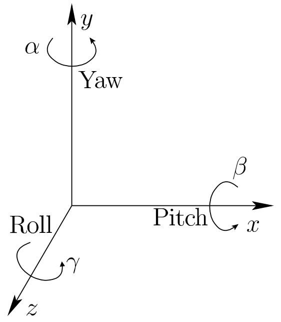


Figure 3.7: Any three-dimensional rotation can be described as a sequence of yaw, pitch, and roll rotations.

Thus, we start with 9 DOFs and want to determine what matrices remain as valid rotations. Follow the same three rules from the 2D case. The columns must have unit length. For example, $m_{11}^2 + m_{21}^2 + m_{31}^2 = 1$. This means that the components of each column must lie on a unit sphere. Thus, the unit-length rule reduces the DOFs from 9 to 6. By following the second rule to ensure perpendicular axes result, the pairwise inner products of the columns must be zero. For example, by choosing the first two columns, the constraint is

$$m_{11}m_{12} + m_{21}m_{22} + m_{31}m_{32} = 0. \tag{3.15}$$

We must also apply the rule to the remaining pairs: The second and third columns, and then the first and third columns. Each of these cases eliminates a DOF, resulting in only 3 remaining DOFs. To avoid mirror images, the constraint $\det M = 1$ is applied, which does not reduce the DOFs.

Finally, we arrive at a set of matrices that must satisfy the algebraic constraints; however, they unfortunately do not fall onto a nice circle or sphere. We only know that there are 3 degrees of rotational freedom, which implies that it should be possible to pick three independent parameters for a 3D rotation, and then derive all 9 elements of (3.14) from them.

Yaw, pitch, and roll One of the simplest ways to parameterize 3D rotations is to construct them from "2D-like" transformations, as shown in Figure 3.7. First consider a rotation about the z-axis. Let *roll* be a counterclockwise rotation of γ about the z-axis. The rotation matrix is given by

$$R_z(\gamma) = \begin{bmatrix} \cos\gamma & -\sin\gamma & 0 \\ \sin\gamma & \cos\gamma & 0 \\ 0 & 0 & 1 \end{bmatrix}. \tag{3.16}$$

The upper left of the matrix looks exactly like the 2D rotation matrix (3.13), except that θ is replaced by γ. This causes yaw to behave exactly like 2D rotation in the xy plane. The remainder of $R_z(\gamma)$ looks like the identity matrix, which causes z to remain unchanged after a roll.

Similarly, let *pitch* be a counterclockwise rotation of β about the x-axis:

$$R_x(\beta) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\beta & -\sin\beta \\ 0 & \sin\beta & \cos\beta \end{bmatrix}. \tag{3.17}$$

In this case, points are rotated with respect to y and z while the x coordinate is left unchanged.

Finally, let *yaw* be a counterclockwise rotation of α about the y-axis:

$$R_y(\alpha) = \begin{bmatrix} \cos\alpha & 0 & \sin\alpha \\ 0 & 1 & 0 \\ -\sin\alpha & 0 & \cos\alpha \end{bmatrix}. \tag{3.18}$$

In this case, rotation occurs with respect to x and z while leaving y unchanged.

Combining rotations Each of (3.16), (3.17), and (3.18) provides a single DOF of rotations. The yaw, pitch, and roll rotations can be combined sequentially to attain any possible 3D rotation:

$$R(\alpha, \beta, \gamma) = R_y(\alpha)R_x(\beta)R_z(\gamma). \tag{3.19}$$

In this case, the ranges of α and γ are from 0 to 2π; however, the pitch β need only range from $-\pi/2$ to $\pi/2$ while nevertheless reaching all possible 3D rotations.

Be extra careful when combining rotations in a sequence because the operations are not commutative. For example, a yaw by $\pi/2$ followed by a pitch by $\pi/2$ does not produce the same result as the pitch followed by the yaw. You can easily check this by substituting $\pi/2$ into (3.17) and (3.18), and observing how the result depends on the order of matrix multiplication. The 2D case is commutative because the rotation axis is always the same, allowing the rotation angles to additively combine. Having the wrong matrix ordering is one of the most frustrating problems when writing software for VR.

Matrix multiplications are "backwards" Which operation is getting applied to the model first when we apply a product of matrices? Consider rotating a point $p = (x, y, z)$. We have two rotation matrices R and Q. If we rotate p using R, we obtain $p' = Rp$. If we then apply Q, we get $p'' = Qp'$. Now suppose that we instead want to first combine the two rotations and then apply them to p to get p''. Programmers are often temped to combine them as RQ because we read from left to right and also write sequences in this way. However, it is backwards for linear algebra because Rp is already acting from the left side. Thus, it "reads" from right to left.[1] We therefore must combine the rotations as QR to obtain $p'' = QRp$. Later in this chapter, we will be chaining together several matrix transforms. Read them from right to left to understand what they are doing!

[1]Perhaps coders who speak Arabic or Hebrew are not confused about this.

Translation and rotation in one matrix It would be convenient to apply both rotation and translation together in a single operation. Suppose we want to apply a rotation matrix R, and follow it with a translation by (x_t, y_t, z_t). Algebraically, this is

$$\begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = R \begin{bmatrix} x \\ y \\ z \end{bmatrix} + \begin{bmatrix} x_t \\ y_t \\ z_t \end{bmatrix}. \tag{3.20}$$

Although there is no way to form a single 3 by 3 matrix to accomplish both operations, it can be done by increasing the matrix dimensions by one. Consider the following 4 by 4 *homogeneous transformation matrix*:

$$T_{rb} = \left[\begin{array}{ccc|c} & & & x_t \\ & R & & y_t \\ & & & z_t \\ \hline 0 & 0 & 0 & 1 \end{array}\right], \tag{3.21}$$

in which R fills the upper left three rows and columns. The notation T_{rb} is used to denote that the matrix is a *rigid body transform*, meaning that it does not distort objects. A homogeneous transform matrix could include other kinds of transforms, which will appear in Section 3.5.

The same result as in (3.20) can be obtained by performing multiplication with (3.23) as follows:

$$\left[\begin{array}{ccc|c} & & & x_t \\ & R & & y_t \\ & & & z_t \\ \hline 0 & 0 & 0 & 1 \end{array}\right] \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix}. \tag{3.22}$$

Because of the extra dimension, we extended the point (x, y, z) by one dimension, to obtain $(x, y, z, 1)$. Note that (3.23) represents rotation *followed by* translation, not the other way around. Translation and rotation do not commute; therefore, this is an important point.

Inverting transforms We frequently want to invert (or undo) transformations. For a translation (x_t, y_t, z_t), we simply apply the negation $(-x_t, -y_t, -z_t)$. For a general matrix transform M, we apply the matrix inverse M^{-1} (if it exists). This is often complicated to calculate. Fortunately, inverses are much simpler for our cases of interest. In the case of a rotation matrix R, the inverse is equal to the transpose $R^{-1} = R^T$.[2] To invert the homogeneous transform matrix (3.23), it is

[2]Recall that to transpose a square matrix, we simply swap the i and j indices, which turns columns into rows.

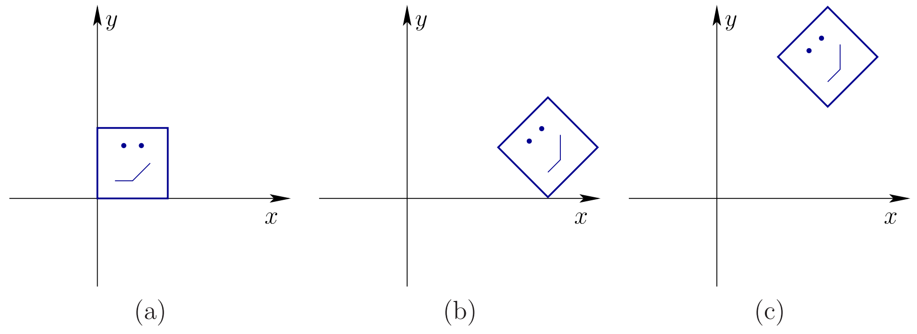


Figure 3.8: (a) A rigid model that is contained in a one-by-one square. (b) The result after rotation by $\pi/4$ (45 degrees), followed with translation by $x_t = 2$. (c) The result after reversing the order: Translation by $x_t = 2$, followed with rotation by $\pi/4$.

tempting to write

$$\left[\begin{array}{ccc|c} & R^T & & -x_t \\ & & & -y_t \\ & & & -z_t \\ \hline 0 & 0 & 0 & 1 \end{array}\right]. \tag{3.23}$$

This will undo both the translation and the rotation; however, the order is wrong. Remember that these operations are not commutative, which implies that order must be correctly handled. See Figure 3.8. The algebra for very general matrices (part of noncommutative group theory) works out so that the inverse of a product of matrices reverses their order:

$$(ABC)^{-1} = C^{-1}B^{-1}A^{-1}. \tag{3.24}$$

This can be seen by putting the inverse next to the original product:

$$ABCC^{-1}B^{-1}A^{-1}. \tag{3.25}$$

In this way, C cancels with its inverse, followed by B and its inverse, and finally A and its inverse. If the order were wrong, then these cancellations would not occur.

The matrix T_{rb} (from 3.23) applies the rotation first, followed by translation. Applying (**??**) undoes the rotation first and then translation, without reversing the order. Thus, the inverse of T_{rb} is

$$\left[\begin{array}{ccc|c} & R^T & & 0 \\ & & & 0 \\ & & & 0 \\ \hline 0 & 0 & 0 & 1 \end{array}\right] \begin{bmatrix} 1 & 0 & 0 & -x_t \\ 0 & 1 & 0 & -y_t \\ 0 & 0 & 1 & -z_t \\ 0 & 0 & 0 & 1 \end{bmatrix}. \tag{3.26}$$

The matrix on the right first undoes the translation (with no rotation). After that, the matrix on the left undoes the rotation (with no translation).

3.3 Axis-Angle Representations of Rotation

As observed in Section 3.2, 3D rotation is complicated for several reasons: 1) Nine matrix entries are specified in terms of only three independent parameters, and with no simple parameterization, 2) the axis of rotation is not the same every time, and 3) the operations are noncommutative, implying that the order of matrices is crucial. None of these problems existed for the 2D case.

Kinematic singularities An even worse problem arises when using yaw, pitch, roll angles (and related Euler-angle variants). Even though they start off being intuitively pleasing, the representation becomes degenerate, leading to *kinematic singularities* that are nearly impossible to visualize. An example will be presented shortly. To prepare for this, recall how we represent locations on the Earth. These are points in $\mathbb{R}^3$, but are represented with longitude and latitude coordinates. Just like the limits of yaw and pitch, longitude ranges from 0 to 2π and latitude only ranges from $-\pi/2$ to $\pi/2$. (Longitude is usually expressed as 0 to 180 degrees west or east, which is equivalent.) As we travel anywhere on the Earth, the latitude and longitude coordinates behave very much like xy coordinates; however, we tend to stay away from the poles. Near the North Pole, the latitude behaves normally, but the longitude could vary a large amount while corresponding to a tiny distance traveled. Recall how a wall map of the world looks near the poles: Greenland is enormous and Antarctica wraps across the entire bottom (assuming it uses a projection that keeps longitude lines straight). The poles themselves are the kinematic singularities: At these special points, you can vary longitude, but the location on the Earth is not changing. One of two DOFs seems to be lost.

The same problem occurs with 3D rotations, but it is harder to visualize due to the extra dimension. If the pitch angle is held at $\beta = \pi/2$, then a kind of "North Pole" is reached in which α and γ vary independently but cause only one DOF (in the case of latitude and longitude, it was one parameter varying but causing zero DOFs). Here is how it looks when combining the yaw, pitch, and roll matrices:

$$\begin{bmatrix} \cos\alpha & 0 & \sin\alpha \\ 0 & 1 & 0 \\ -\sin\alpha & 0 & \cos\alpha \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & -1 \\ 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} \cos\gamma & -\sin\gamma & 0 \\ \sin\gamma & \cos\gamma & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} \cos(\alpha-\gamma) & \sin(\alpha-\gamma) & 0 \\ 0 & 0 & -1 \\ -\sin(\alpha-\gamma) & \cos(\alpha-\gamma) & 0 \end{bmatrix}. \tag{3.27}$$

The second matrix above corresponds to pitch (3.17) with $\beta = \pi/2$. The result on the right is obtained by performing matrix multiplication and applying a subtraction trigonometric identity. You should observe that the resulting matrix is a function of both α and γ, but there is one DOF because only the difference $\alpha - \gamma$ affects the resulting rotation. In the video game industry there has been some

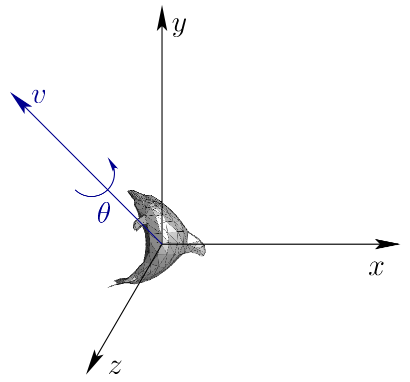


Figure 3.9: Euler's rotation theorem states that every 3D rotation can be considered as a rotation by an angle θ about an axis through the origin, given by the unit direction vector $v = (v_1, v_2, v_3)$.

back-and-forth battles about whether this problem is crucial. In an FPS game, the avatar is usually not allowed to pitch his head all the way to $\pm\pi/2$, thereby avoiding this problem. In VR, it happens all the time that a user could pitch her head straight up or down. The kinematic singularity often causes the viewpoint to spin uncontrollably. This phenomenon also occurs when sensing and controlling a spacecraft's orientation using mechanical gimbals; the result is called *gimbal lock*.

The problems can be easily solved with *axis-angle* representations of rotation. They are harder to learn than yaw, pitch, and roll; however, it is a worthwhile investment because it avoids these problems. Furthermore, many well-written software libraries and game engines work directly with these representations. Thus, to use them effectively, you should understand what they are doing.

The most important insight to solving the kinematic singularity problems is Euler's rotation theorem (1775), shown in Figure 3.9. Even though the rotation axis may change after rotations are combined, Euler showed that *any* 3D rotation can be expressed as a rotation θ about some axis that pokes through the origin. This matches the three DOFs for rotation: It takes two parameters to specify the direction of an axis and one parameter for θ. The only trouble is that conversions back and forth between rotation matrices and the axis-angle representation are somewhat inconvenient. This motivates the introduction of a mathematical object that is close to the axis-angle representation, closely mimics the algebra of 3D rotations, and can even be applied directly to rotate models. The perfect representation: *Quaternions*.

Two-to-one problem Before getting to quaternions, it is important point out one annoying problem with Euler's rotation theorem. As shown in Figure 3.10, it does not claim that the axis-angle representation is unique. In fact, for every 3D rotation other than the identity, there are exactly two representations. This is due to the fact that the axis could "point" in either direction. We could insist that the axis always point in one direction, such as positive y, but this does not fully solve the problem because of the boundary cases (horizontal axes). Quaternions, which are coming next, nicely handle all problems with 3D rotations except this one, which is unavoidable.

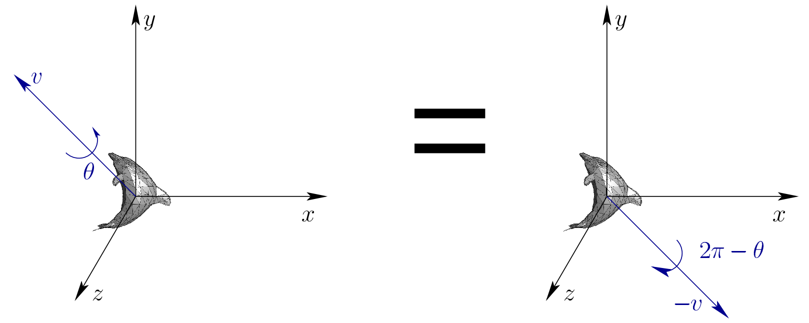


Figure 3.10: There are two ways to encode the same rotation in terms of axis and angle, using either v or $-v$.

Quaternions were introduced in 1843 by William Rowan Hamilton. When seeing them the first time, most people have difficulty understanding their peculiar algebra. Therefore, we will instead focus on precisely which quaternions correspond to which rotations. After that, we will introduce some limited quaternion algebra. The algebra is much less important for developing VR systems, unless you want to implement your own 3D rotation library. The correspondence between quaternions and 3D rotations, however, is crucial.

A quaternion h is a 4D vector:

$$q = (a, b, c, d), \tag{3.28}$$

in which a, b, c, and d can take on real values. Thus, q can be considered as a point in $\mathbb{R}^4$. It turns out that we will only use *unit quaternions*, which means that

$$a^2 + b^2 + c^2 + d^2 = 1 \tag{3.29}$$

must always hold. This should remind you of the equation of a unit sphere ($x^2 + y^2 + z^2 = 1$), but it is one dimension higher. A sphere is a 2D surface, whereas the set of all unit quaternions is a 3D "hypersurface", more formally known as a *manifold* [1, 5]. We will use the space of unit quaternions to represent the space of all 3D rotations. Both have 3 DOFs, which seems reasonable.

Let (v, θ) be an axis-angle representation of a 3D rotation, as depicted in Figure 3.9. Let this be represented by the following quaternion:

$$q = \left(\cos\frac{\theta}{2}\ ,\ v_1 \sin\frac{\theta}{2}\ ,\ v_2 \sin\frac{\theta}{2}\ ,\ v_3 \sin\frac{\theta}{2} \right). \tag{3.30}$$

Quaternion	Axis-Angle	Description
$(1,0,0,0)$	$(\text{undefined},0)$	Identity rotation
$(0,1,0,0)$	$((1,0,0),\pi)$	Pitch by π
$(0,0,1,0)$	$((0,1,0),\pi)$	Yaw by π
$(0,0,0,1)$	$((0,0,1),\pi)$	Roll by π
$(\frac{1}{\sqrt{2}},\frac{1}{\sqrt{2}},0,0)$	$((1,0,0),\pi/2)$	Pitch by $\pi/2$
$(\frac{1}{\sqrt{2}},0,\frac{1}{\sqrt{2}},0)$	$((0,1,0),\pi/2)$	Yaw by $\pi/2$
$(\frac{1}{\sqrt{2}},0,0,\frac{1}{\sqrt{2}})$	$((0,0,1),\pi/2)$	Roll by $\pi/2$

Figure 3.11: For these cases, you should be able to look at the quaternion and quickly picture the axis and angle of the corresponding 3D rotation.

$$\begin{tikzcd}
(a,b,c,d) \arrow[r, leftrightarrow, "\text{equivalent}"'] \arrow[d, leftrightarrow, "\text{inverses}"'] & (-a,-b,-c,-d) \arrow[d, leftrightarrow, "\text{inverses}"] \\
(a,-b,-c,-d) \arrow[r, leftrightarrow, "\text{equivalent}"'] & (-a,b,c,d)
\end{tikzcd}$$

Figure 3.12: Simple relationships between equivalent quaternions and their inverses.

Think of q as a data structure that encodes the 3D rotation. It is easy to recover (v,θ) from q:

$$\theta = 2\cos^{-1} a \text{ and } v = \frac{1}{\sqrt{1-a^2}}(b,c,d). \tag{3.31}$$

If $a = 1$, then (3.31) breaks; however, this corresponds to the case of the identity rotation.

You now have the mappings $(v,\theta) \mapsto q$ and $q \mapsto (v,\theta)$. To test your understanding, Figure 3.11 shows some simple examples, which commonly occur in practice. Furthermore, Figure 3.12 shows some simple relationships between quaternions and their corresponding rotations. The horizontal arrows indicate that q and $-q$ represent the same rotation. This is true because of the double representation issue shown in Figure 3.10. Applying (3.30) to both cases establishes their equivalence. The vertical arrows correspond to inverse rotations. These hold because reversing the direction of the axis causes the rotation to be reversed (rotation by θ becomes rotation by $2\pi - \theta$).

How do we apply the quaternion $h = (a,b,c,d)$ to rotate the model? One way is to use the following conversion into a 3D rotation matrix:

$$R(h) = \begin{bmatrix} 2(a^2+b^2)-1 & 2(bc-ad) & 2(bd+ac) \\ 2(bc+ad) & 2(a^2+c^2)-1 & 2(cd-ab) \\ 2(bd-ac) & 2(cd+ab) & 2(a^2+d^2)-1 \end{bmatrix}. \tag{3.32}$$

A more efficient way exists which avoids converting into a rotation matrix. To accomplish this, we need to define quaternion multiplication. For any two quaternions, q_1 and q_2, let $q_1 * q_2$ denote the product, which is defined as

$$\begin{aligned} a_3 &= a_1a_2 - b_1b_2 - c_1c_2 - d_1d_2 \\ b_3 &= a_1b_2 + a_2b_1 + c_1d_2 - c_2d_1 \\ c_3 &= a_1c_2 + a_2c_1 + b_2d_1 - b_1d_2 \\ d_3 &= a_1d_2 + a_2d_1 + b_1c_2 - b_2c_1. \end{aligned} \tag{3.33}$$

In other words, $q_3 = q_1 * q_2$ as defined in (3.33).

Here is a way to rotate the point (x,y,z) using the rotation represented by h. Let $p = (0,x,y,z)$, which is done to give the point the same dimensions as a quaternion. Perhaps surprisingly, the point is rotated by applying quaternion multiplication as

$$p' = q * p * q^{-1}, \tag{3.34}$$

in which $q^{-1} = (a,-b,-c,-d)$ (recall from Figure 3.12). The rotated point is (x',y',z'), which is taken from the result $p' = (0,x',y',z')$.

Here is a simple example for the point $(1,0,0)$. Let $p = (0,1,0,0)$ and consider executing a yaw rotation by π. According to Figure 3.11, the corresponding quaternion is $(\frac{1}{\sqrt{2}},0,\frac{1}{\sqrt{2}},0)$. The inverse is $q^{-1} = (\frac{1}{\sqrt{2}},0,-\frac{1}{\sqrt{2}},0)$. After tediously applying (3.33) to calculate (3.34), the result is $p' = (0,0,1,0)$. Thus, the rotated point is $(0,1,0)$, which is a correct yaw by $\pi/2$.

3.4 Viewing Transformations

This section describes how to transform the models in the virtual world so that they appear on a virtual screen. The main purpose is to set the foundation for graphical rendering, which adds effects due to lighting, material properties, and quantization. Ultimately, the result appears on the physical display. One side effect of these transforms is that they also explain how cameras form images, at least the idealized mathematics of the process. Think of this section as describing a virtual camera that is placed in the virtual world. What should the virtual picture, taken by that camera, look like? To make VR work correctly, the "camera" should actually be one of two virtual human eyes that are placed into the virtual world. Thus, what should a virtual eye see, based on its position and orientation in the virtual world? Rather than determine precisely what would appear on the retina, which should become clear after Section 4.4, here we merely calculate where the model vertices would appear on a flat, rectangular screen in the virtual world. See Figure 3.13.

An eye's view Figure 3.14 shows a virtual eye that is looking down the negative z axis. It is placed in this way so that from the eye's perspective, x increases to

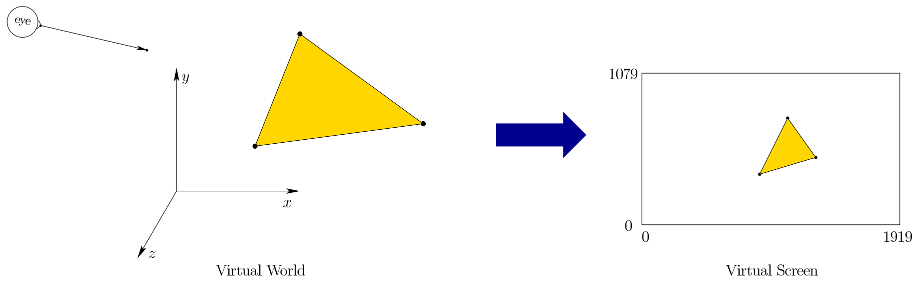


Figure 3.13: If we placed a virtual eye or camera into the virtual world, what would it see? Section 3.4 provides transformations that place objects from the virtual world onto a virtual screen, based on the particular viewpoint of a virtual eye. A flat rectangular shape is chosen for engineering and historical reasons, even though it does not match the shape of our retinas.

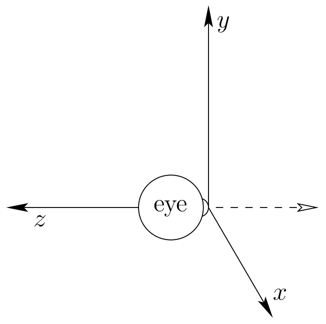


Figure 3.14: Consider an eye that is looking down the z axis in the negative direction. The origin of the model is the point at which light enters the eye.

the right and y is upward. This corresponds to familiar Cartesian coordinates. The alternatives would be: 1) to face the eye in the positive z direction, which makes the xy coordinates appear backwards, or 2) reverse the z axis, which would unfortunately lead to a left-handed coordinate system. Thus, we have made an odd choice that avoids worse complications.

Suppose that the eye is an object model that we want to place into the virtual world $\mathbb{R}^3$ at some position $e = (e_1, e_2, e_3)$ and orientation given by the matrix

$$R_{eye} = \begin{bmatrix} \hat{x}_1 & \hat{y}_1 & \hat{z}_1 \\ \hat{x}_2 & \hat{y}_2 & \hat{z}_2 \\ \hat{x}_3 & \hat{y}_3 & \hat{z}_3 \end{bmatrix} . \tag{3.35}$$

If the eyeball in Figure 3.14 were made of triangles, then rotation by R_{eye} and translation by e would be applied to all vertices to place it in $\mathbb{R}^3$.

This does not, however, solve the problem of how the virtual world should appear to the eye. Rather than moving the eye in the virtual world, we need to move all of the models in the virtual world to the eye's frame of reference. This means that we need to apply the *inverse* transformation. The inverse rotation is R_{eye}^T, the transpose of R_{eye}. The inverse of e is $-e$. Applying (3.26) results in the appropriate transform:

$$T_{eye} = \begin{bmatrix} \hat{x}_1 & \hat{x}_2 & \hat{x}_3 & 0 \\ \hat{y}_1 & \hat{y}_2 & \hat{y}_3 & 0 \\ \hat{z}_1 & \hat{z}_2 & \hat{z}_3 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & -e_1 \\ 0 & 1 & 0 & -e_2 \\ 0 & 0 & 1 & -e_3 \\ 0 & 0 & 0 & 1 \end{bmatrix} . \tag{3.36}$$

Note that R_{eye}, as shown in (3.35), has been transposed and placed into the left matrix above. Also, the order of translation and rotation have been swapped, which is required for the inverse, as mentioned in Section 3.2.

Following Figure 3.4, there are two possible interpretations of (3.36). As stated, this could correspond to moving all of the virtual world models (corresponding to Figure 3.4(b)). A more appropriate interpretation in the current setting is that the virtual world's coordinate frame is being moved so that it matches the eye's frame from Figure 3.14. This corresponds to the case of Figure 3.4(c), which was not the appropriate interpretation in Section 3.2.

Starting from a look-at For VR, the position and orientation of the eye in the virtual world are given by a tracking system and possibly controller inputs. By contrast, in computer graphics, it is common to start with a description of where the eye is located and which way it is looking. This is called a *look-at*, and has the following components:

1. Position of the eye: e

2. Central looking direction of the eye: $\hat{c}$

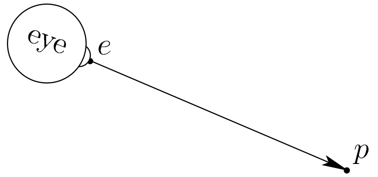


Figure 3.15: The vector from the eye position e to a point p that it is looking at is normalized to form $\hat{c}$ in (3.37).

3. Up direction: $\hat{u}$.

Both $\hat{c}$ and $\hat{u}$ are unit vectors. The first direction $\hat{c}$ corresponds to the center of the view. Whatever $\hat{c}$ is pointing at should end up in the center of the display. If we want this to be a particular point p in $\mathbb{R}^3$ (see Figure 3.15), then $\hat{c}$ can be calculated as

$$\hat{c} = \frac{p - e}{\|p - e\|}, \tag{3.37}$$

in which $\|\cdot\|$ denotes the length of a vector. The result is just the vector from e to p, but normalized.

The second direction $\hat{u}$ indicates which way is up. Imagine holding a camera out as if you are about to take a photo and then performing a roll rotation. You can make level ground appear to be slanted or even upside down in the picture. Thus, $\hat{u}$ indicates the up direction for the virtual camera or eye.

We now construct the resulting transform T_{eye} from (3.36). The translation components are already determined by e, which was given in the look-at. We need only to determine the rotation R_{eye}, as expressed in (3.35). Recall from Section 3.2 that the matrix columns indicate how the coordinate axes are transformed by the matrix (refer to (3.7) and (3.8)). This simplifies the problem of determining R_{eye}. Each column vector is calculated as

$$\begin{aligned} \hat{z} &= -\hat{c} \\ \hat{x} &= \hat{u} \times \hat{z} \\ \hat{y} &= \hat{z} \times \hat{x}. \end{aligned} \tag{3.38}$$

The minus sign appears for calculating $\hat{z}$ because the eye is looking down the negative z axis. The $\hat{x}$ direction is calculated using the standard cross product $\hat{z}$. For the third equation, we could use $\hat{y} = \hat{u}$; however, $\hat{z} \times \hat{x}$ will cleverly correct cases in which $\hat{u}$ generally points upward but is not perpendicular to $\hat{c}$. The unit vectors from (3.38) are substituted into (3.35) to obtain R_{eye}. Thus, we have all the required information to construct T_{eye}.

Orthographic projection Let (x, y, z) denote the coordinates of any point, after T_{eye} has been applied. What would happen if we took all points and directly

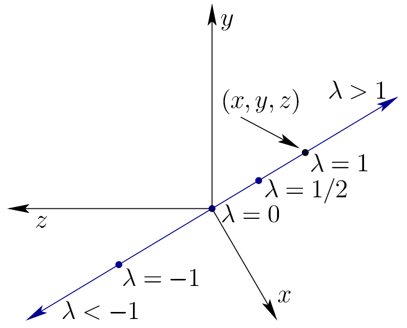


Figure 3.16: Starting with any point (x, y, z), a line through the origin can be formed using a parameter λ. It is the set of all points of the form $(\lambda x, \lambda y, \lambda z)$ for any real value λ. For example, $\lambda = 1/2$ corresponds to the midpoint between (x, y, z) and $(0, 0, 0)$ along the line.

projected them into the vertical xy plane by forcing each z coordinate to be 0? In other words, $(x, y, z) \mapsto (x, y, 0)$, which is called *orthographic projection*. If we imagine the xy plane as a virtual display of the models, then there would be several problems:

1. A jumble of objects would be superimposed, rather than hiding parts of a model that are in front of another.

2. The display would extend infinitely in all directions (except z). If the display is a small rectangle in the xy plane, then the model parts that are outside of its range can be eliminated.

3. Objects that are closer should appear larger than those further away. This happens in the real world. Recall from Section 1.3 (Figure 1.23(c)) paintings that correctly handle perspective.

The first two problems are important graphics operations that are deferred until Chapter 7. The third problem is addressed next.

Perspective projection Instead of using orthographic projection, we define a *perspective projection*. For each point (x, y, z), consider a line through the origin. This is the set of all points with coordinates

$$(\lambda x, \lambda y, \lambda z), \tag{3.39}$$

in which λ can be any real number. In other words λ is a parameter that reaches all points on the line that contains both (x, y, z) and $(0, 0, 0)$. See Figure 3.16.

Now we can place a planar "movie screen" anywhere in the virtual world and see where all of the lines pierce it. To keep the math simple, we pick the $z = -1$

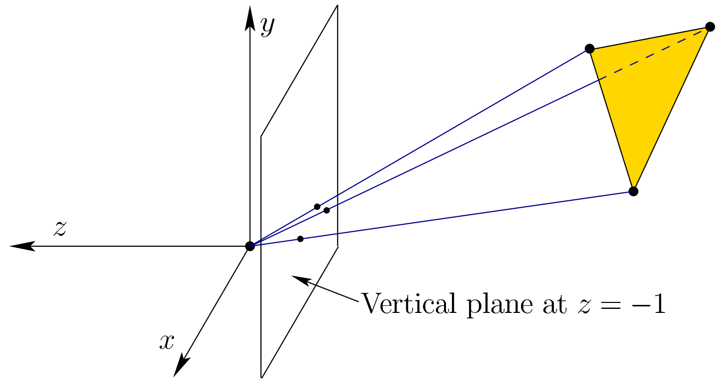


Figure 3.17: An illustration of perspective projection. The model vertices are projected onto a virtual screen by drawing lines through them and the origin $(0, 0, 0)$. The "image" of the points on the virtual screen corresponds to the intersections of the line with the screen.

plane to place our virtual screen directly in front of the eye; see Figure 3.17. Using the third component of (3.39), we have $\lambda z = -1$, implying that $\lambda = -1/z$. Using the first two components of (3.39), the coordinates for the points on the screen are calculated as $x' = -x/z$ and $y' = -y/z$. Note that since x and y are scaled by the same amount z for each axis, their aspect ratio is preserved on the screen.

More generally, suppose the vertical screen is placed at some location d along the z axis. In this case, we obtain more general expressions for the location of a point on the screen:

$$\begin{aligned} x' &= dx/z \\ y' &= dy/z. \end{aligned} \tag{3.40}$$

This was obtained by solving $d = \lambda z$ for λ and substituting it into (3.39).

This is all we need to project the points onto a virtual screen, while respecting the scaling properties of objects at various distances. Getting this right in VR helps in the perception of depth and scale, which are covered in Section 6.1. In Section 3.5, we will adapt (3.40) using transformation matrices. Furthermore, only points that lie within a zone in front of the eye will be projected onto the virtual screen. Points that are too close, too far, or in outside the normal field of view will not be rendered on the virtual screen; this is addressed in Section 3.5 and Chapter 7.

3.5 Chaining the Transformations

This section links all of the transformations of this chapter together while also slightly adjusting their form to match what is currently used in the VR and com-

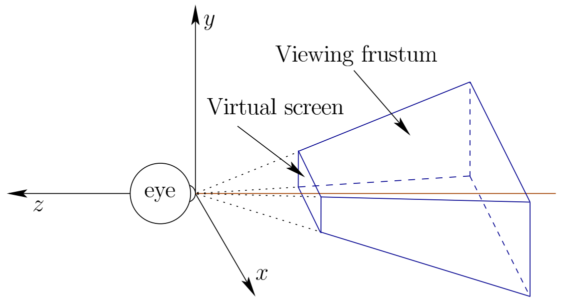


Figure 3.18: The viewing frustum.

puter graphics industries. Some of the matrices appearing in this section may seem unnecessarily complicated. The reason is that the expressions are motivated by algorithm and hardware issues, rather than mathematical simplicity. In particular, there is a bias toward putting every transformation into a 4 by 4 homogeneous transform matrix, even in the case of perspective projection which is not even linear (recall (3.40)). In this way, an efficient matrix multiplication algorithm can be iterated over the chain of matrices to produce the result.

The chain generally appears as follows:

$$T = T_{vp} T_{can} T_{eye} T_{rb}. \tag{3.41}$$

When T is applied to a point $(x, y, z, 1)$, the location of the point on the screen is produced. Remember that these matrix multiplications are not commutative, and the operations are applied from right to left. The first matrix T_{rb} is the rigid body transform (3.23) applied to points on a movable model. For each rigid object in the model, T_{rb} remains the same; however, different objects will generally be placed in various positions and orientations. For example, the wheel of a virtual car will move differently than the avatar's head. After T_{rb} is applied, T_{eye} transforms the virtual world into the coordinate frame of the eye, according to (3.36). At a fixed instant in time, this and all remaining transformation matrices are the same for all points in the virtual world. Here we assume that the eye is positioned at the midpoint between the two virtual human eyes, leading to a *cyclopean viewpoint*. Later in this section, we will extend it to the case of left and right eyes so that stereo viewpoints can be constructed.

Canonical view transform The next transformation, T_{can} performs the perspective projection as described in Section 3.4; however, we must explain how it is unnaturally forced into a 4 by 4 matrix. We also want the result to be in a canonical form that appears to be unitless, which is again motivated by industrial needs. Therefore, T_{can} is called the *canonical view transform*. Figure 3.18 shows a *viewing frustum*, which is based on the four corners of a rectangular *virtual screen*.

At $z = n$ and $z = f$ lie a *near plane* and *far plane*, respectively. Note that $z < 0$ for these cases because the z axis points in the opposite direction. The virtual screen is contained in the near plane. The perspective projection should place all of the points inside of the frustum onto a virtual screen that is centered in the near plane. This implies $d = n$ using (3.40).

We now want to reproduce (3.40) using a matrix. Consider the result of applying the following matrix multiplication:

$$\begin{bmatrix} n & 0 & 0 & 0 \\ 0 & n & 0 & 0 \\ 0 & 0 & n & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} nx \\ ny \\ nz \\ z \end{bmatrix}. \tag{3.42}$$

In the first two coordinates, we obtain the numerator of (3.40). The nonlinear part of (3.40) is the $1/z$ factor. To handle this, the fourth coordinate is used to represent z, rather than 1 as in the case of T_{rb}. From this point onward, the resulting 4D vector is interpreted as a 3D vector that is scaled by dividing out its fourth component. For example, (v_1, v_2, v_3, v_4) is interpreted as

$$(v_1/v_4, v_2/v_4, v_3/v_4). \tag{3.43}$$

Thus, the result from (3.42) is interpreted as

$$(nx/z, ny/z, n), \tag{3.44}$$

in which the first two coordinates match (3.42) with $d = n$, and the third coordinate is the location of the virtual screen along the z axis.

Keeping track of depth for later use The following matrix is commonly used in computer graphics, and will be used here in our chain:

$$T_p = \begin{bmatrix} n & 0 & 0 & 0 \\ 0 & n & 0 & 0 \\ 0 & 0 & n+f & -fn \\ 0 & 0 & 1 & 0 \end{bmatrix}. \tag{3.45}$$

It is identical to the matrix in (3.42) except in how it transforms the z coordinate. For purposes of placing points on the virtual screen, it is unnecessary because we already know they are all placed at $z = n$. The z coordinate is therefore co-opted for another purpose: Keeping track of the distance of each point from the eye so that graphics algorithms can determine which objects are in front of other objects. The matrix T_p calculates the third coordinate as

$$(n + f)z - fn \tag{3.46}$$

When divided by z, (3.46) does not preserve the exact distance, but the graphics methods (some of which are covered in Chapter 7) require only that the distance

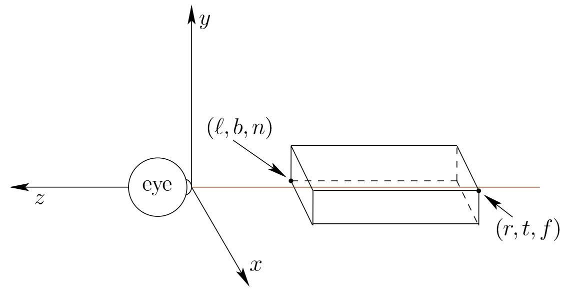


Figure 3.19: The rectangular region formed by the corners of the viewing frustum, after they are transformed by T_p. The coordinates of the selected opposite corners provide the six parameters, ℓ, r, b, t, n, and f, which used in T_{st}.

ordering is preserved. In other words, if point p is further from the eye than point q, then it remains further after the transformation, even if the distances are distorted. It does, however, preserve the distance in two special cases: $z = n$ and $z = f$. This can be seen by substituting these into (3.46) and dividing by z.

Additional translation and scaling After T_p is applied, the 8 corners of the frustum are transformed into the corners of a rectangular box, shown in Figure 3.19. The following performs a simple translation of the box along the z axis and some rescaling so that it is centered at the origin and the coordinates of its corners are $(\pm 1, \pm 1, \pm 1)$:

$$T_{st} = \begin{bmatrix} \frac{2}{r-\ell} & 0 & 0 & -\frac{r+\ell}{r-\ell} \\ 0 & \frac{2}{t-b} & 0 & -\frac{t+b}{t-b} \\ 0 & 0 & \frac{2}{n-f} & -\frac{n+f}{n-f} \\ 0 & 0 & 0 & 1 \end{bmatrix}. \tag{3.47}$$

If the frustum is perfectly centered in the xy plane, then the first two components of the last column become 0. Finally, we define the canonical view transform T_{can} from (3.41) as

$$T_{can} = T_{st} T_p. \tag{3.48}$$

Viewport transform The last transform to be applied in the chain (3.41) is the *viewport transform* T_{vp}. After T_{can} has been applied, the x and y coordinates each range from -1 to 1. One last step is required to bring the projected points to the coordinates used to index pixels on a physical display. Let m be the number of horizontal pixels and n be the number of vertical pixels. For example, $n = 1080$ and $m = 1920$ for a 1080p display. Suppose that the display is indexed with rows running from 0 to $n - 1$ and columns from 0 to $m - 1$. Furthermore, $(0, 0)$ is in the lower left corner. In this case, the viewport transform is

$$T_{vp} = \begin{bmatrix} \frac{m}{2} & 0 & 0 & \frac{m-1}{2} \\ 0 & \frac{n}{2} & 0 & \frac{n-1}{2} \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} . \tag{3.49}$$

Left and right eyes We now address how the transformation chain (3.41) is altered for stereoscopic viewing. Let t denote the distance between the left and right eyes. Its value in the real world varies across people, and its average is around $t = 0.064$ meters. To handle the left eye view, we need to simply shift the cyclopean (center) eye horizontally to the left. Recall from Section 3.4 that the inverse actually gets applied. The models need to be shifted to the right. Therefore, let

$$T_{left} = \begin{bmatrix} 1 & 0 & 0 & \frac{t}{2} \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} , \tag{3.50}$$

which corresponds to a right shift of the models, when viewed from the eye. This transform is placed after T_{eye} to adjust its output. The appropriate modification to (3.41) is:

$$T = T_{vp} T_{can} T_{left} T_{eye} T_{rb} . \tag{3.51}$$

By symmetry, the right eye is similarly handled by replacing T_{left} in (3.51) with

$$T_{right} = \begin{bmatrix} 1 & 0 & 0 & -\frac{t}{2} \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} . \tag{3.52}$$

This concludes the explanation of the entire chain of transformations to place and move models in the virtual world and then have them appear in the right place on a display. After reading Chapter 4, it will become clear that one final transformation may be needed after the entire chain has been applied. This is done to compensate for nonlinear optical distortions that occur due to wide-angle lenses in VR headsets.

Further Reading

Most of the matrix transforms appear in standard computer graphics texts. The presentation in this chapter closely follows [8]. For more details on quaternions and their associated algebraic properties, see [6]. Robotics texts usually cover 3D transformations for both rigid bodies and chains of bodies, and also consider kinematic singularities; see [7, 11].

Bibliography

[1] W. M. Boothby. *An Introduction to Differentiable Manifolds and Riemannian Geometry. Revised 2nd Ed.* Academic, New York, 2003.

[2] M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf. *Computational Geometry: Algorithms and Applications, 2nd Ed.* Springer-Verlag, Berlin, 2000.

[3] J. Gallier. *Curves and Surfaces in Geometric Modeling.* Morgan Kaufmann, San Francisco, CA, 2000.

[4] C. M. Hoffmann. *Geometric and Solid Modeling.* Morgan Kaufmann, San Francisco, CA, 1989.

[5] C. L. Kinsey. *Topology of Surfaces.* Springer-Verlag, Berlin, 1993.

[6] J. B. Kuipers. *Quaternions and Rotation Sequences.* Princeton University Press, Princeton, NJ, 1999.

[7] S. M. LaValle. *Planning Algorithms.* Cambridge University Press, Cambridge, U.K., 2006. Available at http://planning.cs.uiuc.edu/.

[8] S. Marschner and P. Shirley. *Fundamentals of Computer Graphics, 4th Ed.* CRC Press, Boca Raton, FL, 2015.

[9] M. E. Mortenson. *Geometric Modeling, 2nd Ed.* Wiley, Hoboken, NJ, 1997.

[10] D. E. Muller and F. P. Preparata. Finding the intersection of two convex polyhedra. *Theoretical Computer Science*, 7:217–236, 1978.

[11] M. W. Spong, S. Hutchinson, and M. Vidyasagar. *Robot Modeling and Control.* Wiley, Hoboken, NJ, 2005.