

Inspecting the 3D Pipeline, Part II

By Casey Muratori (cmu@d6.com)

Tell me if this scene sounds familiar: in the early stages of development, you run your game, and you see that your 3D test cube is spinning around the world's axes instead of its own local axes. So you try changing the order of a few lines of code, and run it again. Now it's spinning about its local axes, but it's spinning in the wrong direction... or at least, you *think* it's the wrong direction. Are you using the wrong sign on the angle of rotation? Then you look closer and realize that not only is it spinning the wrong way, but you're looking at all the backfaces! So maybe there's actually a sign error in the rotation matrix. But wait - maybe the backface-culling code just has a greater-than sign where it should have a less-than sign... did you check that code?

Although I'm sure none of us wants to admit it, we've all gone through this type of confusion before. While projecting, clipping, and rasterizing are all difficult, there's nothing quite as elusive as gaining a firm understanding of 3D rotations. So, in this installment of [Inspecting the 3D Pipeline](#), we're going to try to dispel the confusion by picking apart 3D rotation matrices and seeing what's really happening inside. We'll look at many different ways in which rotation matrices can be conceptualized, and see how each can be applied to the practical problems involved in every 3D pipeline.

Before we begin, let's run through the typical set of rotation operations a 3D pipeline might involve. We start with a number of objects, each of which can be independently rotated. As part of the game's state, we store (in one form or another) the current rotation of each of these objects. When it comes time to render a scene, we must loop through each object in turn and generate a rotation matrix that puts the object at its correct orientation relative to the camera we're using to view the scene. Typically, this involves at least two different rotation matrices: one which rotates the object from its non-rotated state to its current orientation in the world, and one which rotates it from its current orientation in the world to its orientation relative to the camera.

Thus, as the 3D pipeline runs its course, we conceptually move through three different spaces: *object-space*, where the points of an object are aligned along the x, y, and z axes with no rotation; *world-space*, where the points of all the objects and orientations of all cameras are expressed relative to some global reference frame; and *camera-space*, where the points of all objects are expressed relative to the camera viewing the scene. It is very important to remember these spaces as we look more closely at rotation matrices, because as you'll see, much of the confusion about 3D rotations can be avoided if we pay attention to the spaces we're rotating to and from. With that in mind, let's get started.

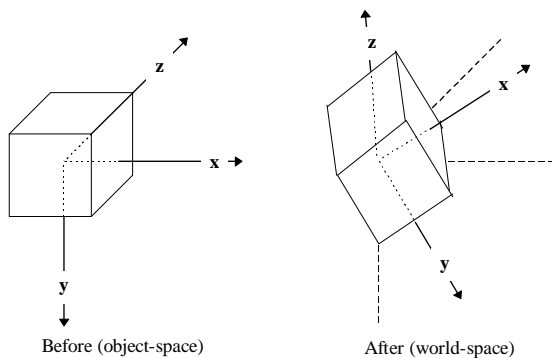


Figure 1. Object-space to world-space rotation

The Columns

All of our objects, by definition, start off in object-space. They've had no rotation applied; we've basically just loaded in their geometry and taken it as-is. If we were to render them all in this way, they'd all be pointing in their original directions, all the time. So, the first rotation matrix we're likely to need is that of the type shown in Figure 1 - a matrix that rotates an object to some other orientation in the world. This way, we can place our objects at different orientations relative to the world axes - we can rotate from object-space to world-space (which I'll call an *object-to-world* rotation).

When we apply such a rotation, think of it as rotating the object's axes away from the world axes into some new orientation. As you can see from Figure 1, it's extremely simple to conceptualize a rotation as a set of rotated axes for an object. If you see how the object is aligned along its axes when it is not rotated, you can very easily picture what the object would look like if you knew the direction of its rotated axes in world-space. Now, if we could figure out how those axes relate to a rotation matrix, we'd be all set: not only could we decipher any rotation matrix by looking at the direction of the axes, we'd also be able to build arbitrarily complicated rotations as long as we knew what vectors we wanted our objects to point along.

Fortunately, the gods of mathematics made it easy for us. It just so happens that, for any given rotation matrix, the vectors formed by its three columns *are* the rotated axes for the objects it transforms. If you think of the 3x3 matrix as partitioned like this:

$$\begin{vmatrix} x_x & y_x & z_x \\ x_y & y_y & z_y \\ x_z & y_z & z_z \end{vmatrix} = \begin{vmatrix} \mathbf{x} & \mathbf{y} & \mathbf{z} \end{vmatrix}$$

then the column vectors \mathbf{x} , \mathbf{y} , and \mathbf{z} are the new axes your object will point along. It's that simple!

Why does it work out this way? Well, let's take the most straightforward approach. Consider the three vectors that describe the axes of your object in object-space. If we rotated each of these vectors by the rotation matrix, we'd get where they'd end up after the rotation, right? Let's see what happens when we try this for the x-axis of our object, $|1\ 0\ 0|^T$ (if you are unfamiliar with the superscript "^T", you'll want to take a quick look at the sidebar before continuing):

$$\begin{vmatrix} \mathbf{x} & \mathbf{y} & \mathbf{z} \end{vmatrix} \begin{vmatrix} 1 \\ 0 \\ 0 \end{vmatrix} = 1\mathbf{x} + 0\mathbf{y} + 0\mathbf{z} = \mathbf{x}$$

As you can see, we end up with only the first column of the matrix - it is the new x-axis, since no other column of the matrix contributes. If you try y and z, you'll see that they work out the same way.

Still not convinced? Try this: begin with the definition of one of the object's points, \mathbf{p} , in object-space. Its coordinates are given as distances along the three principle axes of the object. We might say that, to find \mathbf{p} , we start at the origin of the object,

Vectors and the Transpose Operator

In 3D graphics, the word "vector" is often used without qualification. However, when dealing with matrices or complicated equations, it is often necessary to understand that there are really two different ways a vector can be written: as a column, or as a row. When a vector is written as a single letter, it is assumed to be a column matrix:

$$\mathbf{v} = \begin{vmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{vmatrix}$$

Now, if we wish to refer to \mathbf{v} as a row, we need to make that clear in the notation. So, we need an operation that makes a column into a row. That operation is called *transposition*, and it is represented by the *transpose operator*, a superscript T:

$$\mathbf{v}^T = \begin{vmatrix} v_1 & v_2 & \cdots & v_n \end{vmatrix}$$

It can also be thought of as making a row into a column:

$$\mathbf{v} = \begin{vmatrix} v_1 & v_2 & \cdots & v_n \end{vmatrix}^T$$

The transpose operator is not restricted to vectors - it is simply the general operation of exchanging columns for rows, or vice versa. For example, a two-dimensional matrix also has a transpose, given by the exchanging of its rows for its columns:

$$\mathbf{A}^T = \begin{vmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{vmatrix}^T = \begin{vmatrix} A_{11} & A_{21} & A_{31} \\ A_{12} & A_{22} & A_{32} \\ A_{13} & A_{23} & A_{33} \end{vmatrix}$$

Another important aspect of the transpose operator is that it shows the equivalence of the dot product and a matrix multiplication. With a matrix multiplication, you multiply the rows of the first matrix by the columns of the second. So, the dot product can be easily expressed as a matrix multiply by using the transpose operator:

$$\mathbf{u} \cdot \mathbf{v} = \mathbf{u}^T \mathbf{v}$$

If you would like to read more about vector and matrix operations, a good place to start would be with an introductory linear algebra book, such as [Introduction to Linear Algebra](#) by Gilbert Strang.

move p_x units along its x-axis, p_y units along its y-axis, and p_z units along its z-axis - then we've arrived.

Now apply the rotation:

$$\mathbf{p}' = \begin{vmatrix} \mathbf{x} & \mathbf{y} & \mathbf{z} \\ p_x \\ p_y \\ p_z \end{vmatrix} = p_x \mathbf{x} + p_y \mathbf{y} + p_z \mathbf{z}$$

Look at the right side of this equation - it says, to find the rotated point \mathbf{p}' , start at the origin, move p_x units along \mathbf{x} , p_y units along \mathbf{y} , and p_z units along \mathbf{z} . Sound familiar? It's doing the same thing we did when we originally defined \mathbf{p} , only instead of using the object's local axes, it's using the axes defined by the columns of the matrix. It is putting the point exactly where it should be if the object's axes were not coincident with the world axes, but were coincident with the vectors \mathbf{x} , \mathbf{y} , and \mathbf{z} - exactly what we thought should happen.

The Rows

Looking at a matrix as a collection of columns provided us with a good way to understand object-to-world rotations. However, a world-space to camera-space (which I'll call *world-to-camera*) rotation is decidedly different. It is not orienting an object in world-space. Instead, it is rotating objects already in world-space to be oriented relative to a particular camera.

Figure 2 shows what this means in terms of axes. The camera's axes are described in world-space, and our world-to-camera rotation matrix rotates the world so that the camera's axes become the new world axes. In essence, we are making everything's orientation relative to the camera.

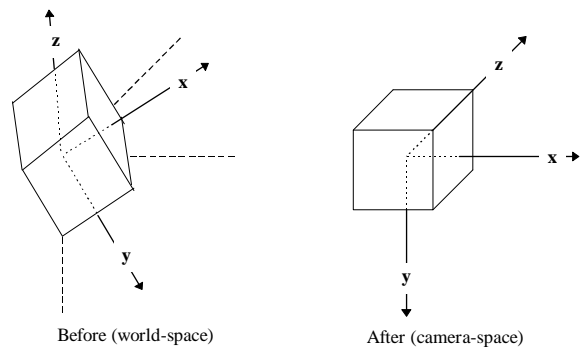


Figure 2. World-space to camera-space rotation

As with the previous section, we want to find some way of relating the axes of the camera to the elements of the rotation matrix we'd use to view from that camera. As you might have guessed from the section heading, the relation is just as simple as it was for the column picture... for a world-to-camera rotation, the axes of the camera appear as the rows of the matrix. Think of the 3x3 rotation matrix like this:

$$\begin{vmatrix} x_x & x_y & x_z \\ y_x & y_y & y_z \\ z_x & z_y & z_z \end{vmatrix} = \begin{vmatrix} \mathbf{x}^T \\ \mathbf{y}^T \\ \mathbf{z}^T \end{vmatrix}$$

\mathbf{x} , \mathbf{y} , and \mathbf{z} are the camera's axes.

To see why this is the case, we can use similar techniques to those we used for the column picture. As we just said, if our rotation is indeed the world-to-camera rotation, it will rotate the camera's axes to coincide with the world axes. So, if we rotate the camera's axes by our rotation matrix, and the world axes come out, our assumption about the rows must be correct. Let's try that for the camera's x-axis, \mathbf{x} :

$$\begin{vmatrix} \mathbf{x}^T \\ \mathbf{y}^T \\ \mathbf{z}^T \end{vmatrix} \mathbf{x} = \begin{vmatrix} \mathbf{x}^T \mathbf{x} \\ \mathbf{y}^T \mathbf{x} \\ \mathbf{z}^T \mathbf{x} \end{vmatrix} = \begin{vmatrix} 1 \\ 0 \\ 0 \end{vmatrix}$$

In the previous article, we looked at the orthogonality of the principle axes, and made some observations about dot-products between them. Those observations come into play here: we know \mathbf{y} and \mathbf{z} are both orthogonal to \mathbf{x} (by definition), so their dot-product is 0, and any unit-vector dotted with itself is 1. So, $\mathbf{y}^T \mathbf{x}$ and $\mathbf{z}^T \mathbf{x}$ both become

0, and $\mathbf{x}^T \mathbf{x}$ becomes 1, leaving us with $|1\ 0\ 0|^T$, the world x-axis (if you're unfamiliar with the $\mathbf{v}^T \mathbf{v}$ form of the dot product, see the sidebar). If you try the y and z axes, you'll see that they work in the same fashion.

As with the column picture, we can verify our assertions in more than one way. Take a point \mathbf{p} , this time in world-space. Now apply the camera-space rotation:

$$\mathbf{p}' = \begin{vmatrix} \mathbf{x}^T \\ \mathbf{y}^T \\ \mathbf{z}^T \end{vmatrix} \mathbf{p} = \begin{vmatrix} \mathbf{x}^T \mathbf{p} \\ \mathbf{y}^T \mathbf{p} \\ \mathbf{z}^T \mathbf{p} \end{vmatrix}$$

The result is a single vector comprised of dot-products. We know the dot product maps one vector onto another; for a unit vector \mathbf{u} , and any other vector \mathbf{v} , it answers the question, "if \mathbf{u} were an axis, what would \mathbf{v} 's coordinate be along that axis?" In that light, the above multiplication is answering the question "if \mathbf{x} , \mathbf{y} , and \mathbf{z} were axes, what would \mathbf{p} 's coordinates be on those axes?" In this way, our rotation results in a projection of \mathbf{p} onto the axes \mathbf{x} , \mathbf{y} , and \mathbf{z} . We are left with what the coordinates of \mathbf{p} would be if the rows of our transform matrix were the new world axes - exactly what we thought should happen.

The Transposed Matrix

We've come to an interesting place in our understanding of rotation matrices: we know how to look at them as columns, we know how to look at them as rows, but we haven't really looked at the two pictures together. Surely, since they are both rotations, there must be *some* link between them... so let's see if we can find it.

Take an object-to-world rotation matrix. This matrix gives us the orientation of a particular object in world-space. Now, suppose we wanted to view the world from this object, effectively making it into a camera. What would we do?

Well, using the column picture, we know that the columns of our object's object-to-world matrix are the axes of the object. Using the row picture, we know we can build a world-to-camera matrix if we have the axes of the "camera", which in this case is our object. We can take the columns out of the objects current matrix, plug them in as *rows* of a rotation matrix, and poof! We're left with the world-to-camera rotation that views the world from our object. In short, to turn an object-to-world rotation into a world-to-camera transform for the same object, all we have to do is exchange the matrix's columns for its rows.

Similarly, if we had a camera's world-to-camera rotation, and we wanted to make that camera into an actual object in the world, we could build the camera's object-to-world rotation in the exact same way. Follow essentially the same process we just used: extract the camera's axes from its world-to-camera rotation matrix, then plug them into the object-to-camera matrix. You'll see that all you end up doing is exchanging the rows and the columns.

Exchanging a matrix's rows with its columns is called *transposing* the matrix (see the sidebar for more details), and it provides the link between the columns and the rows that we were looking for. Think about what we just did: we found that you could convert between object-to-world rotations and world-to-camera rotations just by transposing the given rotation matrix. That in itself is a link. But we can see an even more interesting relation between the rows and columns once we realize one additional thing about object-to-world rotations and world-to-camera rotations: they are perfectly opposite operations.

Consider it: the object-to-world rotation rotates the objects axes away from the origin into the current orientation of the object. The world-to-camera rotation rotates the objects axes away from its current orientation into the world axes. They are opposite, or *inverse*, operations. Chances are, you've seen it written elsewhere that "the inverse of a rotation matrix is its transpose". Well, the reason is nothing more than the coupling between the rows and columns that we've just seen.

Matrix Concatenations

In the previous sections, we attacked rotation matrices as static entities - we took a given type of rotation, and saw what the components of the rotation matrix meant in a more tangible way. Now we're going to change gears slightly and look at matrices in a more dynamic way. Specifically, we're going to see what happens when different rotation matrices are combined, or *concatenated*, to form new rotations.

There wouldn't be much to talk about if there was only one way to combine any two rotations. But, as we've all heard a million times, matrix multiplication is non-commutative - the order in which the matrices are multiplied affects the resulting matrix. So, in order to fully understand the concatenation of rotations, we need to get the order straight.

Take the following example:

$$\mathbf{p}' = (\mathbf{KLMN})\mathbf{p}$$

Here we have four rotation matrices, multiplied together, that transform a single point. In which order are these rotations being applied? Is **K** happening first, or is **N** happening first? Well, because matrix multiplication is associative, we can make the situation clearer by associating the terms differently:

$$\mathbf{p}' = \mathbf{K}(\mathbf{L}(\mathbf{M}(\mathbf{Np})))$$

Now, we can easily see what's happening. **N** occurs first, rotating the point **p** to a new point. Then **M** rotates this new point, then **L**, and finally **K**. So, returning to the original question, when we see a set of rotation matrices concatenated together, they will be applied *from right to left*.

As a side-note, there are some interesting tricks we can perform once we understand the order of rotation concatenations. For example, suppose we have a spaceship object that points along its z-axis in object-space, and we're keeping its current orientation as a rotation matrix, **R_S**. Now, if we want our spaceship to do a roll, we have two options. We can figure out what the spaceship's z-axis is in world-space, build an arbitrary-axis rotation matrix, **R_A**, that rotates about that axis, then pre-concatenate it onto the spaceship's current matrix, yielding:

$$\mathbf{R}'_S = \mathbf{R}_A \mathbf{R}_S$$

We'd be wasting a lot of time and energy if we went that route. Instead, we can build a primary z-axis rotation matrix, **R_Z**, and post-concatenate it:

$$\mathbf{R}'_S = \mathbf{R}_S \mathbf{R}_Z$$

Since **R_Z** happens first, it will happen before the spaceship is rotated to its current orientation - so the rotation about the primary z-axis will occur when the object's z-axis is still aligned with it. In essence, we are using the order of multiplication to go back in time to the objects original state (object-space), apply a rotation, then let all the other rotations happen afterwards.