# Figuring out vector conventions from code

Sometimes you find yourself in codebases you’re unfamiliar with, trying to make changes to 3D math code. It’s better if we try to understand the conventions used in the codebase rather than wasting time trying various combinations of transformation orders until we stumble upon something that happens to work.

A lot of the confusion comes from the choice of row or column vectors and the resulting matrix transformation orders. You should read Fabian Giesen’s excellent post [1] about this subject.It follows from Fabian’s post that you can deduce the vector conventions used if you see a transform concatenation taking place in the existing code. Consider a model that is to be transformed from its local space into camera space via world space. We have two basic homogeneous 4x4 matrices involved:



We can glean one piece of information from how those matrices are combined to form a composite transform that accomplishes both.

If you see composite transformation matrix being computed like this:

And a point is later transformed like this:

You can be reasonably sure that the codebase treats vectors as 1x4 matrices (row vectors) which go on the left when multiplying.

If the two matrices are multiplied in the reverse order, you can assume that the convention is to treat vectors as 4x1 matrices (column vectors), which go on the right when multiplying.

All this means is that when you’re trying to figure out where to slot in your additional transformation, you should think about the “transformation pipeline” of matrix multiplication pairs like this:

* Row vectors used: **point** 🡪**xform1** 🡪 **xform2**🡪 **xform3** 🡪 …
* Column vectors used: … 🡨 **xform3** 🡨 **xform2** 🡨 **xform1** 🡨 **point**

Understanding the convention means can pretty easily see where to insert your added transformation in the chain. Note that regardless of the convention used, the point being transformed enters the pipeline and is transformed by the first transform, then the second. The steps are the same. But the order in which the matrices are multiplied is reversed.

Also, note that the above doesn’t have anything to do with in-memory storage order of the matrices. Whether we store the matrices in column or row-major order in memory doesn’t affect the above discussion at all. A matrix multiply will always dot the rows of the left hand side with the columns of the right hand side, regardless of how those numbers that make up those rows and columns are stored in memory. Fabian covers that subject in much better detail than I can in his post [1].

However, there is another important implication that comes from the choice of row or column vectors: it affects what parts of our matrices we consider the basis vectors:

* If we choose row vectors, and multiply *vAB*, then the matrices *A*/*B* must have their basis vectors in the **rows**.
* If we instead choose column vectors and multiply *BAv*, the result only makes sense if the matrices *A*/*B* have their basis vectors in the **columns**.

Again, this is not talking about memory layout order of the rows or columns, but simply something that falls out of the choice of row or column vectors and the order of transformations.

We can visualize this clearly in pure mathematical notation. I’ll use 3x3 matrices for clarity. Let’s start with the case where we treat vectors as row matrices:

Here we’re transforming a vector by a very simple 3x3 matrix that just swaps some axes around. Because the vector is simply a unit length along the *x* axis, we expect it to just pick out the basis vector of the new space. The basis vector it picks is the first row of the matrix.

If we flip the thing around, and treat the vector as a column matrix, we have a different situation:

In this case, our *x* unit vector has picked out the first column, which constitutes the basis vector of the space when multiplying with column matrix vectors.

[1] <https://fgiesen.wordpress.com/2012/02/12/row-major-vs-column-major-row-vectors-vs-column-vectors/>