Affine Transformations
Computer Graphics
Scott D. Anderson

1 Linear Combinations

To understand the power of an affine transformation, it's helpful to understand the idea of a "linear combination." If we have two things, A and B, such that it makes sense both to multiply them by scalars and to add them together, a linear combination of A and B is any expression like the following:

$$\alpha A + \beta B$$

where α and β are any real numbers. In other words, a "linear combination" of A and B is the sum of a number multiplied by A and a number multiplied by B. For example,

$$3A - 2B$$

is a linear combination of A and B.

We've seen this kind of expression before, when we looked at parametric equations of lines; any point on a line between A and B is a linear combination of A and B. In that case, A and B were points, and the two weights add to 1:

$$\alpha A + (1 - \alpha)B$$

2 Affine Transforms

The essential power of affine transformations is that we only need to transform the endpoints of a segment, and every point on the segment is transformed, because lines map to lines. Hence, the transformation must be linear. What "linear" means in this context is the following:

$$f(\alpha p + \beta q) = \alpha f(p) + \beta f(q)$$

What this equation means is that the mapping f, applied to a linear combination of p and q, is the same as the linear combination of f applied to the p and q. Thus,

> In order to transform every point on a line segment, it's sufficient to transform the endpoints. In particular, to transform a line drawn from A to B, it's sufficient to transform A and B and then draw the lines between the transformed endpoints.

Fortunately, matrix multiplication has this property of being linear.

3 Matrix Multiplication

Matrix multiplication is a shorthand for equations with just adding and multiplying. The following is the product of a matrix and a (column) vector.

$$\begin{bmatrix} ax + by \\ cx + dy \end{bmatrix} = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

This is useful when the second thing is a vector or point in our model and the matrix accomplishes some transformation we want to do. So, it's useful to separate the object from the transformation.

Matrix multiplication is done in an odd way. You go across the rows of the matrix on the left and down the columns of the matrix on the right, multiplying matching elements. Then add up the products of the matching elements. So, multiplying a 2×3 by a 3×2 we get a 2×2:

$$\begin{bmatrix} \sum_i a_{1i}b_{i1} & \sum_i a_{1i}b_{i2} \\ \sum_i a_{2i}b_{i1} & \sum_i a_{2i}b_{i2} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \end{bmatrix} \times \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \\ b_{31} & b_{32} \end{bmatrix}$$

Let's do an example.

$$\begin{bmatrix} 1\cdot 2+3\cdot 6+5\cdot 10 & 1\cdot 4+3\cdot 8+5\cdot 12 \\ 7\cdot 2+9\cdot 6+11\cdot 10 & 7\cdot 4+9\cdot 8+11\cdot 12 \end{bmatrix} = \begin{bmatrix} 1 & 3 & 5 \\ 7 & 9 & 11 \end{bmatrix} \times \begin{bmatrix} 2 & 4 \\ 6 & 8 \\ 10 & 12 \end{bmatrix}$$

So, finishing the arithmetic, the matrix product is

$$\begin{bmatrix} 70 & 88 \\ 178 & 232 \end{bmatrix}$$

4 Translation, Rotation and Scaling

In this section, we'll look at how the three major affine transformations are accomplished using matrix multiplication. We'll start with the easiest one and work up.

4.1 3D Scaling

We can easily do scaling with a matrix multiplication.

$$\begin{bmatrix} s_x x \\ s_y y \\ s_z z \end{bmatrix} = \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & s_z \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

This matrix results in the x coordinate being multiplied by s_x, and so forth. Since all the matrix entries are zero except for the three scale factors, all we have to tell OpenGL about is the three non-zero entries.

The OpenGL equivalent of this is:

```
glScalef(sx,sy,sz);
```

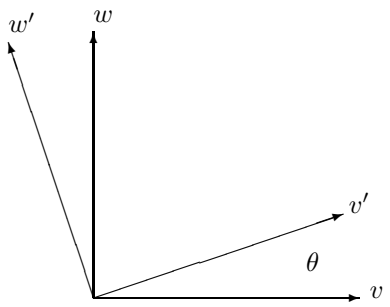
It may seem silly to do a big matrix multiplication instead of three simple multiplications, but it's really useful to be able to do everything with matrix multiplications, as we'll see by the end of this document.

5 2D Rotation

Rotation changes only the directions of the coordinate system (these are called *basis vectors* in linear algebra), but not the origin. We'll rotate *vectors* but the math is the same as "rotating" a point. You can think of rotating a point as rotating the vector from the origin to the point.

5.1 Rotating Basis Vectors

The basis vectors are (1,0,0), (0,1,0) and (0,0,1). Because they have such a simple form, it's fairly easy to figure out how to rotate them. Let's look at rotating just one.



You may be able to figure out from your knowledge of trigonometry that the new vectors are:

$$v' = \begin{bmatrix} \cos\theta \\ \sin\theta \end{bmatrix} \text{ and } w' = \begin{bmatrix} -\sin\theta \\ \cos\theta \end{bmatrix}$$

We can express this computation using *matrix multiplication*:

$$v' = \begin{bmatrix} \cos\theta \\ \sin\theta \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

and

$$w' = \begin{bmatrix} -\sin\theta \\ \cos\theta \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

Thus, we can determine the rotated form of the basis vectors by multiplying them by this rotation matrix. Since every vector can be defined by a linear combination (there's that terminology again) of the basis vectors, and matrix multiplication is linear, this rotation matrix will work for any vector. Let's see one example.

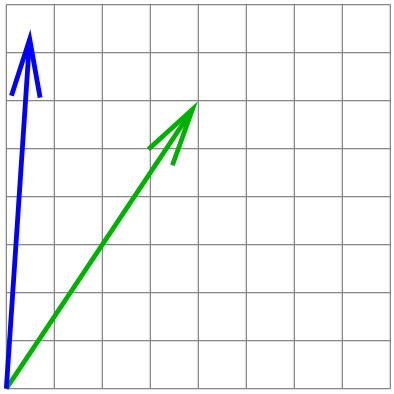
5.2 Example of Rotating a Vector

Let's start with the vector $v = (2, 3)$. You can think of this as an arrow from the origin to the point (2,3). In general, we can think of any vector as a *direction*, like "north by northeast" is a direction. Here, our direction is "to the right (positive x) and up (positive y), but more up than right." In fact, it's a lot like "north by northeast."

What does it mean to rotate this vector? It's like turning a ship: you're now heading it a different direction. Let's turn this vector by 30 degrees to yield v':

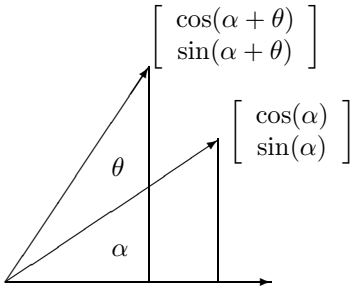
$$\begin{aligned} v' &= \begin{bmatrix} \cos(30) & -\sin(30) \\ \sin(30) & \cos(30) \end{bmatrix} \begin{bmatrix} 2 \\ 3 \end{bmatrix} \\ &= \begin{bmatrix} 0.866 & -0.5 \\ 0.5 & 0.866 \end{bmatrix} \begin{bmatrix} 2 \\ 3 \end{bmatrix} \\ &= \begin{bmatrix} 2 * 0.866 - 3 * 0.5 \\ 2 * 0.5 + 3 * 0.866 \end{bmatrix} \\ &= \begin{bmatrix} 0.232 \\ 3.6 \end{bmatrix} \end{aligned}$$

Here is a picture. The green arrow is the original vector $v = (3, 2)$. The blue arrow is the result after rotating by 30 degrees.



5.3 Arbitrary Unit-Length Vectors

What about rotating an arbitrary unit-length vector? Suppose it's at an angle of α with the x axis. We want to rotate it to an angle of $\alpha + \theta$.



By some theorems of trigonometry having to do with sines and cosines of the sums of angles, we find:

$$\begin{bmatrix} \cos(\alpha+\theta) \\ \sin(\alpha+\theta) \end{bmatrix} = \begin{bmatrix} \cos\alpha\cos\theta - \sin\alpha\sin\theta \\ \sin\alpha\cos\theta + \cos\alpha\sin\theta \end{bmatrix}$$

But this is the same thing as the matrix multiplication.

$$\begin{bmatrix} \cos(\alpha+\theta) \\ \sin(\alpha+\theta) \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} \cos\alpha \\ \sin\alpha \end{bmatrix}$$

Notice that this is the same rotation matrix we used in the last section. In fact, the general 2D rotation matrix is the following. It can be used to rotate any vector or point.

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

We can also think of this as a 3D rotation around the z axis, using the following 3×3 matrix. This is, in fact, the way that OpenGL does rotations around the z axis.

$$\begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

Generally, there is such a matrix for rotation around any vector, not just around the z axis. We'll let OpenGL figure out what the matrix is. The OpenGL equivalent of rotating by θ around the z axis is:

```
glRotatef(theta,0,0,1);
```

5.4 2D Translation

Translation can't be done with matrix multiplication (without a trick), so it has to be done with addition.

$$\begin{bmatrix} x + \Delta x \\ y + \Delta y \end{bmatrix} = \begin{bmatrix} \Delta x \\ \Delta y \end{bmatrix} + \begin{bmatrix} x \\ y \end{bmatrix}$$

But we really want to use multiplication, for reasons we'll see in a moment, so we use *homogeneous coordinates*.

5.5 Homogeneous Coordinates

Homogeneous coordinates adds an extra component, conventionally called w, to each vector or point. You can try to think of it as a spatial dimension, with the usual points and vectors being a subspace of it, but I find that difficult and unhelpful. I think it's easier to just think of it as a representational trick.

Using homogeneous coordinates, a *point* has $w = 1$, while a *vector* has $w = 0$. This lets us tell points and vectors apart, which is pretty cool. Here's an example. If we have points $P = (7, 6, 5)$ and $Q = (1, 2, 3)$, we can give the homogeneous coordinates thus:

$$\begin{aligned} P &= (7, 6, 5, 1) \\ Q &= (1, 2, 3, 1) \\ v &= P - Q = (6, 4, 2, 0) \end{aligned}$$

But, how can you remember which has $w = 0$ and which has $w = 1$? I find it easier to recall that a vector, v, from Q to P is defined as follows:

$$\begin{aligned} v &= P - Q \\ Q &= P + v \end{aligned}$$

Whatever the w component of the points P and Q are, the w component of the vector v has to be zero. Therefore, the w component of any vector is zero and the w component of a point is 1.

How do homogeneous coordinates help us with translation? If we want to translate something by $(\Delta x, \Delta y, \Delta z)$, we can use the following matrix:

$$\begin{bmatrix} x + \Delta x \\ y + \Delta y \\ z + \Delta z \\ w \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & \Delta x \\ 0 & 1 & 0 & \Delta y \\ 0 & 0 & 1 & \Delta z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix}$$

Take a minute to work out the matrix multiplication to see that this 4×4 matrix is, in fact, a translation matrix in homogeneous coordinates.

The OpenGL equivalent of this translation matrix is:

```
glTranslatef(deltax,deltay,deltaz);
```

6 Transformations in 4D

So we can do translation by multiplying by a 4×4 matrix. How does this affect rotation and scaling? It turns out to be pretty straightforward. Since we just want to keep the w component the same, we just expand our earlier matrices with a row and column of zeros except for a 1 in the corner.

Thus, the 4×4 scaling matrix is just:

$$\begin{bmatrix} s_x x \\ s_y y \\ s_z z \\ w \end{bmatrix} = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix}$$

We can similarly expand the rotation matrix to 4×4 to leave the w component alone.

7 Concatenation of Matrices

In this section, we'll finally learn why we've gone to such lengths to turn every affine transformation into a matrix multiplication, instead of representing each transformation in a more direct way.

Suppose our program, such as the Fence program, has used a large number of transformations and then sends a bunch of vertices down the pipeline. How is this computed?

One way to compute is to take a vertex and apply one matrix, then another, then another, and so forth. To compute the transformed vertex, v', from the original vertex, v, the computation looks something like:

$$v' = S(R(T_1(T_2(T_3(R_2(T_4(v))))))))$$

If you're doing this on many vertices, there's a better way. First, combine all the matrices into one. The computation above becomes:

$$v' = (SRT_1T_2T_3R_2T_4)v$$

In other words, the OpenGL pipeline represents *all* of the transformations in one 4×4 matrix. A single matrix can represent any sequence of any number of translations, rotation, scalings, in any order. This is a *tremendous* simplification. That's why OpenGL (and every graphics system) uses homogeneous coordinates and represents every transformation using a matrix.

The concatenation (product) of all transformation matrices are captured in a matrix called the *current transformation matrix* (CTM). Thus, when you do `glTranslatef` or `glRotatef` or `glScalef`, what you're really doing is multiplying the CTM, like this:

$$\text{CTM} \leftarrow \text{CTM} \times M$$

where M is the appropriate translation, rotation or scaling matrix, like one of the matrices we saw above. When you do `glPushMatrix` and `glPopMatrix`, you're saving and restoring the CTM.

There are even OpenGL calls for multiplying the current matrix by an arbitrary 4×4 matrix.