CS 314: 2D Transformations Continued, Homogeneous Coordinates, Starting 3D

Robert Bridson

September 11, 2008

1 More 2D Transformations

1.1 Other 2×2 transformations

Another important class of transformations are **reflections**, where we flip points over the x-axis or the y-axis like a mirror image, simply by negating either the x or y coordinate. For example, to reflect over the y-axis you can use this matrix:

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} -1 & 0 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix}$$

In fact this is just a special case of a scaling, where all the scale factors are ± 1. It's easy to see this reflection matrix is also orthogonal. As it happens, any orthogonal matrix can be expressed as a product of reflections and rotations.

The final special class of 2×2 matrix transformations is **shearing**. A y-shear leaves x-coordinates unchanged but shifts y-coordinates up or down proportional to x, skewing squares into parallelograms for example. To skew by a slope s (i.e. turning horizontal lines into lines with slope s), this can be written as

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ s & 1 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix}$$

An x-shear, which shifts x-coordinates left or right proportional to y but leaves y coordinates unchanged, is similar:

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} 1 & s \\ 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix}$$

These aren't particularly useful for most graphics needs, but do arise occasionally.

While all of the primitive transformations seen so far have been presented as distinct, the truth is that once you start composing them into sequences of transformations (i.e. multiplying matrices together) they're not unique. For example, any matrix can be represented as a rotation followed by a scaling followed by another rotation[1]

1.2 Inverting Transformations

Another useful convenience that comes along with expressing transformations with matrices is that it's simple to invert the transformation, just by taking the matrix inverse: if $\vec{x}' = M\vec{x}$, then $\vec{x} = M^{-1}\vec{x}'$. (This assumes of course that the matrix is invertible: the only transformation we've discussed which could be singular is the scaling, if a scale factor is exactly zero.)

Why would you want to do this? A typical scenario is in user interaction. Say we want to make it possible for a user to click on a triangle to select it for editing; the operating system will provide some means for finding the pixel coordinates of the mouse cursor in the rasterized image we drew, but we need to figure out which point that corresponds to in the user's coordinate system. The inverse transformation matrix instantly tells us this.

1.3 Change of Basis

If you remember your linear algebra, transforming vectors by multiplying with invertible matrices is closely tied to the notion of **change of basis**. This is just a different way of thinking about what we're doing with transforms, which sometimes is a lot more natural than thinking "operationally" about a sequence of rotations and scalings and so forth.

Instead of thinking of a point as a vector of real numbers, we can think more abstractly that it's a primitive geometric object. Once you pick an origin point and a set of basis vectors, which together form a **coordinate system**, you can describe the point numerically with its coordinates in that system. The origin has coordinates $(0, 0)$, points that lie in the direction of the first basis vector (which we usually call the x-axis) from the origin have coordinates of the form $(x, 0)$, points that lie in the direction of the second basis vector (the y-axis) from the origin have coordinates of the form $(0, y)$, and so on.

[1] If you're interested, this actually goes under the name of Singular Value Decomposition or SVD, one of the most useful matrix factorizations but often ignored in linear algebra courses; the Shirley textbook has quite a lot of material on it if you want to know more.

The transformation matrices we've discussed so far provide a way of taking a point expressed as coordinates in one basis, and computing its coordinates relative to a different basis. This of course comes with the restriction that the origins of the input and the output coordinate systems are the same, i.e. we're just changing the basis vectors, but we'll relax that in the next section. The inverse matrix transforms back from the second basis to the first.

A scaling corresponds to an output coordinate system with the same origin, and whose basis vectors are parallel to the input coordinate system but of different lengths.

A rotation corresponds to an output coordinate system with the same origin, and whose basis vectors are the same length but rotated by some angle.

A reflection corresponds to an output coordinate system with the same origin, but where one of the basis vectors has been flipped to point the opposite way.

A shear corresponds to an output coordinate system with the same origin, but where the basis vectors have been made non-orthogonal relative to the input coordinate system.

This viewpoint leads to another way to look at the entries in a transformation matrix: the columns are just the coordinates (in the output coordinate system) of the basis vectors of the input coordinate system. This isn't hard to see: the coordinates of the x-axis basis vector in the input coordinate system are just $(1, 0)$. If you transform that vector to get its coordinates in the output coordinate system, you get:

$$\begin{pmatrix} m_{11} & m_{12} \\ m_{21} & m_{22} \end{pmatrix} \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} m_{11} \\ m_{21} \end{pmatrix}$$

which is obviously the first column of the matrix. Similarly the y-axis, which has input coordinates $(0, 1)$, transforms to be equal to the second column.

Once you understand this, you can directly construct a transformation matrix just by figuring out what you want the new coordinate system basis vectors to be—once you have the coordinates of the old basis vectors relative to the new coordinate system, simply write down a matrix with those as the columns. Keep in mind: it's important not to get mixed up here and do it the other way (write down the coordinates of the new basis vectors in the old coordinate system) as then you would end up with the inverse transformation instead.

2 Translation

We now can build a reasonably flexible graphics system, where the user specifies triangles with their own chosen measurements (in their own coordinate system) and can easily rasterize to any size of image (the display's coordinate system), and in addition can make adjustments like rotating around the origin or reflecting across an axis. However, this is still seriously limited: what if the user wanted to rotate around a point other than the origin, or reflect over a line that doesn't go through the origin, or just plain move the triangles in some fixed direction? What if their coordinate system has a different origin than the bottom left corner of the image?

All of this can be done with **translation**. Translation by a fixed vector (x_t, y_t) just means adding it to the coordinates of every point, shifting all geometry by that vector:

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} x + x_t \\ y + y_t \end{pmatrix}$$

Unfortunately, vector addition can't be expressed as a multiplication with a 2×2 matrix.

But matrices are just so elegant—in terms of being to compose sequences of transformations, invert general transformations, etc.—that we're not going to give up on them. Instead we'll introduce one of the most important tricks in computer graphics, which later will prove even more useful in 3D: **homogeneous coordinates**.

What this means is that we'll add a third coordinate to our 2D points, which for now will always be the number 1:

$$\begin{pmatrix} x \\ y \end{pmatrix} \rightarrow \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$$

Even though we're talking about 2D points, we're going to (at least conceptually) store them as 3D points that all happen to lie on the $z = 1$ plane. In the previous sections we built 2×2 transformation matrices for scaling, rotation, etc. that we now expand to 3×3 matrices as follows:

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix} \rightarrow \begin{pmatrix} a & b & 0 \\ c & d & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

You should verify that multiplying this 2×2 matrix by a regular 2D vector is exactly equivalent to multiplying the expanded 3×3 version by the expanded vector, getting back an expanded vector with a 1 as its last coordinate.

What's really cool about this is that we can now write down a 3×3 matrix that encodes translation in 2D, as follows:

$$\begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} = \begin{pmatrix} x + x_t \\ y + y_t \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & x_t \\ 0 & 1 & y_t \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$$

This new translation transformation matrix uses the third column to store the translation vector:

$$T = \begin{pmatrix} 1 & 0 & x_t \\ 0 & 1 & y_t \\ 0 & 0 & 1 \end{pmatrix}$$

Another good exercise for you is to verify that the inverse of this matrix, T^{-1}, is just translation in the opposite direction, i.e. by the vector $(-x_t, -y_t)$. If you translate a point by (x_t, y_t) and then translate that by $(-x_t, -y_t)$, you end up with the original point, which is equivalent to multiplying by the identity matrix (leaving coordinates unchanged).

Now that translation is just another matrix, we can compose a sequence of translations and rotations and scalings etc. in any order with simple matrix multiplication just as before. We can invert this total transformation easily as well.

This gives us a lot more transformation power. For example, instead of requiring the user to specify their triangles in terms of a rectangle with $(0, 0)$ as one corner, they can put the image rectangle anywhere they want: to transform user coordinates to pixel coordinates in the image we'll just need to start with a translation which moves everything back relative to the origin. Let's do that in more detail!

2.1 Using Translations for Coordinate Systems

Translations (and the other transformations) let us transform between arbitrary coordinate systems, i.e. ones with different origins as well as different basis vectors.

As an example, say the user wants to work on a nice and symmetric square stretching from $x = -1$ to $x = +1$, and similarly from $y = -1$ to $y = +1$, i.e. the domain $[-1, 1] \times [-1, 1]$. Here their origin $(0, 0)$ is actually in the centre of the image, and the bottom left corner is instead at user coordinates $(-1, -1)$.

If we immediately introduced a scaling transformation to make this the pixel width and the height of the output image, we'd find that the centre of the user's space gets erroneously mapped to the bottom left pixel, and three quarters of their space doesn't even show up in the image.

Therefore we start the transformation with a translation that puts the user's bottom left corner at $(0, 0)$, ready for a scaling later. That is, we translate all points with the vector $(1, 1)$. This maps:

- the bottom left corner at $(-1, -1)$ to $(-1 + 1, -1 + 1) = (0, 0)$,
- the top left corner at $(-1, 1)$ to $(-1 + 1, 1 + 1) = (0, 2)$,
- the bottom right corner at $(1, -1)$ to $(1 + 1, -1 + 1) = (2, 0)$,
- and the top right corner at $(1, 1)$ to $(1 + 1, 1 + 1) = (2, 2)$.

It's shifted the user's domain $[-1, 1] \times [-1, 1]$ to $[0, 2] \times [0, 2]$. Now we can apply a scaling with factors $m/2$ and $n/2$ to map this set to the pixel domain $[0, m] \times [0, n]$.

Writing this out in matrix form, the complete transformation is:

$$\begin{aligned} M &= \begin{pmatrix} \frac{m}{2} & 0 & 0 \\ 0 & \frac{n}{2} & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 1 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{pmatrix} \\ &= \begin{pmatrix} \frac{m}{2} & 0 & \frac{m}{2} \\ 0 & \frac{n}{2} & \frac{n}{2} \\ 0 & 0 & 1 \end{pmatrix} \end{aligned}$$

I've multiplied it out for you in the second line, so you can see what the matrix will be ultimately, but usually you'd let the computer do that and not do it by hand. Also note it's very important to get the translation and scaling matrices in the correct order: the first thing we want to do to an input point is to translate it, and only then do we rescale it, so the translation matrix needs to be on the right. It would be a good thing for your brain at this point to stop reading and verify, by matrix multiplication on a piece of paper, that this transformation does map the user's bottom left corner $(-1, -1)$ to the bottom left pixel $(0, 0)$, and the user's top right corner $(1, 1)$ to the top right pixel (m, n).

Now we've worked through that example, let's tackle something a bit more general, which is sometimes called the windowing transform (see section 6.3.1 in the Shirley text). This maps a given axis-aligned rectangle $[a, A] \times [b, B]$ to another rectangle $[c, C] \times [d, D]$. (The reason for the name "windowing" is thinking of this as starting with a window on the geometry at $[a, A] \times [b, B]$, which we want to map to an output window $[c, C] \times [d, D]$ for display—in particular, this sort of thing is very important for

user-interfaces with windows!) In other words, it will take as input points with coordinates in the range $a \leq x \leq A$ and $b \leq y \leq B$ and return output coordinates in the range $c \leq x' \leq C$ and $d \leq y' \leq D$. We just went through an example of this with $a = -1$, $A = 1$, $b = -1$ and $B = 1$ specifying the user's domain, and $c = 0$, $C = m$, $d = 0$, and $D = n$ specifying the output image in pixel coordinates.

The first step in windowing is to use a translation, which takes the bottom left corner of the input at (a, b) to the origin $(0, 0)$. (You might be wondering why we do this first, instead of a translation which directly moves it to (c, d): the reason is that the next step will be a scaling, which keeps the origin $(0, 0)$ fixed but can move around every other point.) This is translation by the vector $(-a, -b)$, giving the following matrix:

$$\begin{pmatrix} 1 & 0 & -a \\ 0 & 1 & -b \\ 0 & 0 & 1 \end{pmatrix}$$

You can check this moves the input rectangle $[a, A] \times [b, B]$ to an intermediate rectangle $[0, A-a] \times [0, B-b]$.

The next step is to rescale this intermediate rectangle so that it has the same width and height as the output rectangle. Its width is $C - c$ and its height is $D - d$, so the scaling matrix we need is:

$$\begin{pmatrix} \frac{C-c}{A-a} & 0 & 0 \\ 0 & \frac{D-d}{B-b} & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

This will transform our intermediate rectangle to $[0, C - c] \times [0, D - d]$.

Finally, now that the intermediate rectangle has the right width and height for the output, we translate it so that the corners match up: the translation vector clearly needs to be (c, d). The accompanying matrix is:

$$\begin{pmatrix} 1 & 0 & c \\ 0 & 1 & d \\ 0 & 0 & 1 \end{pmatrix}$$

That's the last step: we're done.

Again, being careful about the order of these operations, the total transformation is:

$$M = \begin{pmatrix} 1 & 0 & c \\ 0 & 1 & d \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} \frac{C-c}{A-a} & 0 & 0 \\ 0 & \frac{D-d}{B-b} & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & -a \\ 0 & 1 & -b \\ 0 & 0 & 1 \end{pmatrix}$$

If you want to multiply this out by hand to see the final result, go ahead; the Shirely text also contains the product. Again, the computer would usually do this for you, however, after you specified three distinct operations (translation, scaling, translation).

On a related note, you can use translation and the other transformations to convert between arbitrary coordinate systems with different origins. The first step is a translation that moves the origin of the input coordinate system to the origin of the output system, expressed as a vector with coordinates in the input system. Then multiply by the rotation or scaling or whatever other non-translation matrix it is that encodes the basis vectors of the input coordinate system relative to the output coordinate system.

2.2 Using Translations for General Rotations

We can use similar tricks to rotate an object about an arbitrary point (x, y) instead of just the origin $(0, 0)$, which is what the plain rotation matrix we saw earlier was restricted to.

The game plan is almost the same as the windowing transform: we'll first translate everything so that (x, y) gets mapped to $(0, 0)$, then we'll do the rotation, and finally we'll do another translation to map $(0, 0)$ (which is unchanged by rotation) back to (x, y). The end result is that the point (x, y) will remain fixed by this total transformation, and everything else will rotate around it.

The first step is a translation to move (x, y) to $(0, 0)$: clearly we need the translation vector $(-x, -y)$. The matrix is:

$$\begin{pmatrix} 1 & 0 & -x \\ 0 & 1 & -y \\ 0 & 0 & 1 \end{pmatrix}$$

The second step is the rotation around $(0, 0)$:

$$\begin{pmatrix} \cos(\phi) & -\sin(\phi) & 0 \\ \sin(\phi) & \cos(\phi) & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

The last step is a translation that takes $(0, 0)$ back to (x, y), with the translation vector (x, y):

$$\begin{pmatrix} 1 & 0 & x \\ 0 & 1 & y \\ 0 & 0 & 1 \end{pmatrix}$$

Putting this all together in the correct order gives:

$$M = \begin{pmatrix} 1 & 0 & x \\ 0 & 1 & y \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} \cos(\phi) & -\sin(\phi) & 0 \\ \sin(\phi) & \cos(\phi) & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & -x \\ 0 & 1 & -y \\ 0 & 0 & 1 \end{pmatrix}$$

You can multiply this one out by hand too, but it's a bit ugly: again, it's better to let the computer do it for you after you specify the translation, rotation and translation steps individually.

3 A Very Primitive 3D Rendering Pipeline

Before extending all of our transformation work up to 3D, we'll take a sneak peek at how a full 3D rendering pipeline might look. We've already covered the final stages to some degree, namely 2D transformations to pixel coordinates, rasterization of 2D triangles, and output as an image. Let's stick in a very simplified set of operations for the earlier stages, which take place in 3D.

We'll do this by rendering models from the 'front view' (one which should be familiar to anyone who has seen side views and top views in engineering blueprints). All this boils down to is a **projection** from 3D to 2D by throwing away the z coordinate. That is, we map a 3D triangle to a 2D triangle that we can transform and rasterize simply by ignoring the third z coordinates of its vertices and just using the x and y coordinates. For now let's ignore the issue of homogeneous coordinates, just adding them in where needed.

This is far from complete—if you were to implement this now and try it out, you would hit a big problem in the output that we'll spend time fixing later (hint: the order in which triangles are rasterized is important). Nonetheless, it gives us a rudimentary skeleton of the pipeline that we can start enhancing, now at the start.

Leaving the projection step (throwing out z) fixed in stone, which is actually very close to what the full-fledged OpenGL pipeline does, how could we render objects from something other than the front view? What if we wanted the top view, say?

We can do this conceptually by rotating the model so its top side faces the front (along the z-axis): i.e. use a 3D transformation (a rotation) to transform the model before sending it to the rest of the pipeline. So let's start taking a look at 3D transformations, which are fairly straightforward extensions of all the 2D transformations we've just seen.

3.1 3D Rotations

The simplest form of 3D rotations is just rotating around one of the axes. This simplifies to a 2D rotation in the plane orthogonal to the axis.

For example, to rotate ϕ radians around the x-axis, i.e. in the yz-plane, we know the x coordinates of points should remain unchanged and the y and z coordinates should be transformed as if by a 2D

rotation. It's easy to see this can be expressed as:

$$\begin{pmatrix} x' \\ y' \\ z' \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos(\phi) & -\sin(\phi) \\ 0 & \sin(\phi) & \cos(\phi) \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix}$$

You can probaby write down something plausible for rotation around the z-axis too, and maybe the y-axis, but before we go that far, we should realize there's a potential ambiguity lurking here.

In 2D, we specified we were rotating ϕ radians *counterclockwise*: in 3D unfortunately the notion of clockwise versus counterclockwise is ambiguous. If you look at a transparent clock from behind, its hands actually are moving counterclockwise apparently. We'll tackle this complication next time.