CS 314: 3D Transformations

Robert Bridson

September 16, 2008

1 3D Transformations

1.1 More on 3D Rotations

Last time we saw our first 3D rotation, around the x-axis in the yz-plane:

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

This was mostly copied from the 2D rotation specified as ϕ radians counterclockwise in the plane. However, the notion of clockwise versus counterclockwise doesn't entirely make sense in 3D: the right way to specify the direction of a rotation is through the use of something like the right-hand-rule.

A rotation around the x-axis with a right-hand-rule means that if you placed your hand around the x-axis with your thumb pointing in the direction of positive x, then your fingers would curl in the direction of a positive ($\phi > 0$) rotation. You can double check that the matrix above does that—though it may take a while drawing diagrams on paper to see it.

Continuing on, we can introduce the right-hand-rule rotations around the other axes. For the y-axis, the matrix is:

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

and for the z-axis the matrix is:

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

The z-axis rotation is directly equivalent to the 2D rotation in the xy-plane.

1.1.1 General Rotations

Just as in 2D (and in fact, as in any dimension), any orthogonal matrix represents either a rotation around some axis or a rotation followed by a reflection. In particular if the columns of a 3×3 orthogonal matrix U satisfy a cross-product rule, namely the third column is the cross-product of the first with the second

$$\vec{u}_3 = \vec{u}_1 \times \vec{u}_2$$

then the matrix is a rotation that rotates the standard basis vectors $(1, 0, 0)$, $(0, 1, 0)$ and $(0, 0, 1)$ to the columns of U. (The other possibility for the last column is $-\vec{u}_1 \times \vec{u}_2$, which would indicate a reflection has also taken place. This boils down to determining if the columns of U satisfy a left-hand-rule or a right-hand-rule; if it has flipped relative to the original set of basis vectors, then a reflection has happened.)

This gives one approach for determining a 3D rotation around any arbitrary axis $\vec{r}$ by ϕ radians, espoused by the Shirley text.

Given $\vec{r} = (r_1, r_2, r_3)$, a unit-length vector, you can first construct an orthonormal set of basis vectors that includes $\vec{r}$, which satisfies the cross-product rule. For example, the second vector $\vec{s}$ can be chosen as

$$\vec{s} = \frac{(r_2, -r_1, 0)}{\sqrt{r_1^2 + r_2^2}}$$

if at least one of r_1 and r_2 is nonzero, and otherwise take $\vec{s} = (1, 0, 0)$. There are many other ways to construct such a vector $\vec{s}$: in class and in the text there was a different construction. Finally, the third basis vector is $\vec{t} = \vec{r} \times \vec{s}$. (As an exercise, you can further confirm that any orthonormal basis satisfying the cross-product rule $\vec{t} = \vec{r} \times \vec{s}$ also satisfies $\vec{r} = \vec{s} \times \vec{t}$ and $\vec{s} = \vec{t} \times \vec{r}$.) Build an orthogonal matrix Q^T with $\vec{r}$, $\vec{s}$ and $\vec{t}$ as columns (or alternatively Q with $\vec{r}$ etc. as rows); Q is now a rotation matrix which rotates the vector $\vec{r}$ to $(1, 0, 0)$, since its inverse Q^T obviously maps $(1, 0, 0)$ to $\vec{r}$.[1]

Now, with this Q in place, we can rotate around $\vec{r}$ by

- first apply Q to change the basis to one where $\vec{r}$ is along $(1, 0, 0)$,
- next use a rotation of ϕ around the new x-axis,
- and finally change the basis back to what it used to be with the inverse of Q, i.e. Q^T.

[1] To match the Shirley text's description on page 148, our Q is the same as the text's R_{uvw}^T.

This is a product of three rotations, so it's a rotation itself.

However, it turns out through other means the final result of all of this can be derived into a somewhat simpler form:

$$\begin{pmatrix} \cos\phi + (1-\cos\phi)r_1^2 & (1-\cos\phi)r_1r_2 - r_3\sin\phi & (1-\cos\phi)r_1r_3 + r_2\sin\phi \\ (1-\cos\phi)r_1r_2 + r_3\sin\phi & \cos\phi + (1-\cos\phi)r_2^2 & (1-\cos\phi)r_2r_3 - r_1\sin\phi \\ (1-\cos\phi)r_1r_3 - r_2\sin\phi & (1-\cos\phi)r_2r_3 + r_1\sin\phi & \cos\phi + (1-\cos\phi)r_3^2 \end{pmatrix}$$

This relies, of course, on $\vec{r}$ being unit length: if it's not, you need to first **normalize** it by dividing by its length. Obviously this isn't going to work if $\vec{r} = 0$.

1.1.2 General Orientations

Rotation matrices go hand-in-hand with the idea of **orientation**. If we take some object and put it in a new orientation, i.e. rotate it, that new orientation can be represented as some general rotation matrix. However, this isn't the most concise or convenient representation of the orientation: imagine trying to type in a matrix by hand that has to be orthogonal. Specifying an axis $\vec{r}$ and a rotation angle ϕ is better, but also has problems in some situations: it can be hard to visualize what the necessary axis is for large general rotations.

A very common approach to specifying general rotations uses so-called **Euler angles**, which boil down to the fact that any 3D rotation can be written as a sequence of a rotation around the x-axis by some angle ϕ_1, then a rotation around the y-axis by ϕ_2, and then a rotation around the z-axis by ϕ_3. The order of the axes isn't critical as long as it's consistent. This approach to specifying orientation is especially useful for **navigation** in 3D: designing interfaces for programs where the user can interactively move around in 3D, such as when modeling 3D geometry or playing a first-person shooter game—motions of the mouse might be directly connected to some of these Euler angles. For example, the angle ϕ_1 could represent tilting the view up or down, the angle ϕ_2 the heading in the horizontal plane, and the angle ϕ_3 if needed would represent rolling to the left or right.

1.2 Scaling and Shearing

Scaling and shearing can be generalized from 2D in the obvious way. A scaling is expressed with a diagonal matrix:

$$\begin{pmatrix} s_1 & 0 & 0 \\ 0 & s_2 & 0 \\ 0 & 0 & s_3 \end{pmatrix}$$

A shearing takes the identiy matrix plus one off-diagonal entry: for example, a shearing of the x-axis proportional to z takes the form

$$\begin{pmatrix} 1 & 0 & s \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

for some slope parameter s.

1.3 Translation

To include 3D translations (i.e. adding a fixed vector to any input) we can use exactly the same homogeneous coordinate trick we did before, expanding 3D vectors into 4D with an extra 1 (the homogeneous coordinate) tacked on the end. The previous 3×3 transformation matrices can be expanded to 4×4 the same way, e.g. for the y-axis rotation:

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

The new translation matrices add the translation vector to the fourth column of the 4×4 identity matrix. Translation by (x, y, z) is expressed as:

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

With this in place, we can do the same composite operations (such as rotating around a point other than the origin) as we worked out in 2D.

2 Different Spaces

With all this in place, it's proved helpful to think in a number of different coordinate systems for the rendering pipeline.

The input to rendering (the 3D geometry we're trying to draw in the image) is implicitly given in **world space**: it's assumed those coordinates are relative to the 3D world's natural origin and basis vectors. For example, in a game set in a Martian army base, the world space origin might be at the centre

of the map, the x-axis could be one metre long and point east, the y-axis one metre long and point up, and the z-axis one metre long and point south (to make it right-handed—note: I got it wrong in the lecture and said north). The world space coordinates of 3D geometry in the map would be in metres, with the first coordinate how far east of the centre of the map the point is etc. World space coordinates should be independent of the rendering: they don't care whether we want a top-view, side-view, or any other view on the geometry; it's what you would store in the "map file".

The next coordinate system or space to consider is **camera space**, also known as **eye space**. A very good way of thinking about 3D rendering is actually putting a virtual camera (or your own eye) in the virtual world at a particular location, oriented a particular way, and simulating the light it would record. Camera space is a coordinate system with its origin at the position of this virtual camera, with basis vectors chosen so the camera's x-axis points to the right in the image, the camera's y-axis points upwards in the image, and the camera's z-axis points out of the image at the viewer. The camera can potentially see all the geometry with negative z-values (camera z, not necessarily world space z). Typically camera space is built just from rotations and translations, so that camera coordinates are measured in the same units as world space (e.g. metres).

The transformation that takes world space coordinates and produces camera space coordinates is called the **model-view** matrix. This is the 4×4 matrix which tells the renderer how to convert world-space coordinates of geometry into the desired viewpoint for the image.