Transforms

Computer Graphics - SET08116

EDINBURGH NAPIER UNIVERSITY

Outline

Coordinate Spaces

Consider that we have a 3D coordinate space *C* that has an origin and three coordinate axes. Any point in this coordinate space can be defined using $< x, y, z >$, with these values indicating the distance travelled along each of the three relevant axes to reach the point.

Suppose now we want to exist in a different 3D coordinate space, *C'*. The original point, **P** can be expressed as the having coordinates $< x', y', z' >$ in this coordinate space. These new coordinates can be expressed as linear functions of the original coordinates:

$$\begin{aligned}
x'(x, y, z) &= U_1 x + V_1 y + W_1 z + T_1 \\
y'(x, y, z) &= U_2 x + V_2 y + W_2 z + T_2 \\
z'(x, y, z) &= U_3 x + V_3 y + W_3 z + T_3
\end{aligned}$$

Linear Transformations

The previous slide created a *linear transformation* from *C* to *C'*, and we can define this as a matrix calculation:

$$\begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = \begin{bmatrix} U_1 & V_1 & W_1 \\ U_2 & V_2 & W_2 \\ U_3 & V_3 & W_3 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} + \begin{bmatrix} T_1 \\ T_2 \\ T_3 \end{bmatrix}$$

There are a number of values to consider here:

- The vector $< x', y', z' >$ represents the distance traveled along the axes of *C'* to reach **P**
- The vector **T** represents the translation of the origin of *C* to *C'*
- The vectors **U**, **V** and **W** represent the change in orientation between the coordinate spaces.

Linear Transformations

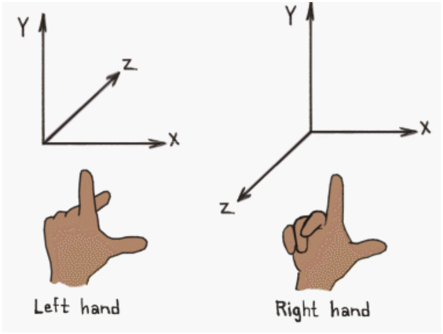
If the transformation matrix is invertible, then the following equation holds:

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} U_1 & V_1 & W_1 \\ U_2 & V_2 & W_2 \\ U_3 & V_3 & W_3 \end{bmatrix}^{-1} \left(\begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} - \begin{bmatrix} T_1 \\ T_2 \\ T_3 \end{bmatrix} \right)$$

We can concatenate multiple transformatations by multiplying the matrices together. This is the fundamental part of what we are doing. When we work with vectors, we move from object space to world space (via the model matrix) to camera space (via the view matrix) to screen space (via the projection matrix). Concatenating matrices allows us to do this in one operation.

Handedness

It is also worth considereding now the handedness of our space. coordinate system defined by the three vectors $\mathbf{V}_1$, $\mathbf{V}_2$ and $\mathbf{V}_3$ has a property called *handedness*. This can be illustrated as follows:



Handedness

There are a two properties that of interest when discussing handedness:

- If $(V_1 \times V_2) \cdot V_3 > 0$ then the coordinate space is right handed.
- If $(V_1 \times V_2) = V_3$ then the coordinate space is right handed orthonormal. An orthonormal coordinate space means that the vectors that define the space are orthogonal and are unit length.

We can change the handedness of space by reflecting the matrix. This is done using a *reflection transform*, which is defined as follows:

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & -1 \end{bmatrix}$$

Scaling Transforms

To scale a vector **P** by a factor of *a*, we calculate the following:

$$\mathbf{P}' = a\mathbf{P}$$

This calculation can be represented in matrix form as follows:

$$\mathbf{P}' = \begin{bmatrix} a & 0 & 0 \\ 0 & a & 0 \\ 0 & 0 & a \end{bmatrix} \begin{bmatrix} P_x \\ P_y \\ P_z \end{bmatrix}$$

For example, to scale $< 5, 10, 20 >$ by 3, we perform the following operation:

$$\mathbf{P}' = \begin{bmatrix} 3 & 0 & 0 \\ 0 & 3 & 0 \\ 0 & 0 & 3 \end{bmatrix} \begin{bmatrix} 5 \\ 10 \\ 20 \end{bmatrix} = \begin{bmatrix} 15 \\ 30 \\ 60 \end{bmatrix}$$

Nonuniform Scaling

The previous approach is called *uniform scaling*. Sometimes we may want to scale an vector by different values across the different axes. This is called *nonuniform scaling*. This time the main diagonals consist of the different scale values:

$$\mathbf{P}' = \begin{bmatrix} a & 0 & 0 \\ 0 & b & 0 \\ 0 & 0 & c \end{bmatrix} \begin{bmatrix} P_x \\ P_y \\ P_z \end{bmatrix}$$

So, for example, we can scale $< 5, 10, 20 >$ by $< 2, 4, 8 >$ to get:

$$\mathbf{P}' = \begin{bmatrix} 2 & 0 & 0 \\ 0 & 4 & 0 \\ 0 & 0 & 8 \end{bmatrix} \begin{bmatrix} 5 \\ 10 \\ 20 \end{bmatrix} = \begin{bmatrix} 10 \\ 40 \\ 160 \end{bmatrix}$$

Scaling on an Arbitrary Axis

We can also scale on an arbitrary set of axes rather than just the ones used to define the the coordinate space. To do this, we need to define the vectors **U**, **V** and **W** to act as the axes to scale upon. The equation we use is:

$$\mathbf{P}' = \begin{bmatrix} U_x & V_x & W_x \\ U_y & V_y & W_y \\ U_z & V_z & W_z \end{bmatrix} \begin{bmatrix} a & 0 & 0 \\ 0 & b & 0 \\ 0 & 0 & c \end{bmatrix} \begin{bmatrix} U_x & V_x & W_x \\ U_y & V_y & W_y \\ U_z & V_z & W_z \end{bmatrix}^{-1} \begin{bmatrix} P_x \\ P_y \\ P_z \end{bmatrix}$$

So, for example, if we use the scale $< 8, 4, 2 >$ on vector $< 5, 10, 20 >$ using $\mathbf{U} =< 0, 0, 1 >$, $\mathbf{V} =< 0, 1, 0 >$ and $\mathbf{W} =< 1, 0, 0 >$ we get the following:

$$\mathbf{P}' = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} 8 & 0 & 0 \\ 0 & 4 & 0 \\ 0 & 0 & 2 \end{bmatrix} \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix}^{-1} \begin{bmatrix} 5 \\ 10 \\ 20 \end{bmatrix} = \begin{bmatrix} 10 \\ 40 \\ 160 \end{bmatrix}$$

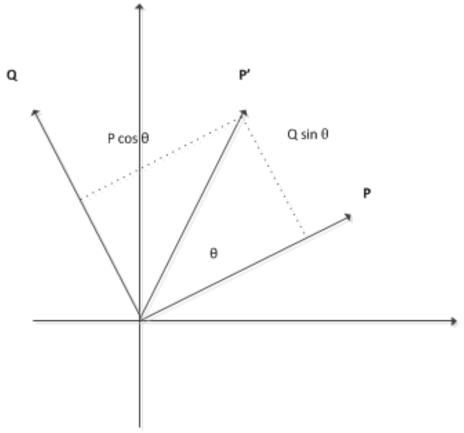
Rotation Transforms

Another goal we have is to rotate a coordinate system, thereby reorientating it. A rotation of θ around an axis **A** is considered to be a counterclockwise rotation while the axis faces us. What we will do is define a standard method to view this.

Let us consider 2D rotation. If we have a 2D vector, **P**, we can determine the $\frac{\pi}{2}$ rotation of this vector as $< -P_y, P_x >$. If you think about what we are doing, this makes sense. Let us call this new vector **Q**.

Rotation Transforms

Any rotation less than $\frac{\pi}{2}$ is just a linear combination of the two vectors, **P** and **Q**. We can visualise this as follows:



Rotation Transforms

We can now define an equation by using basic trigonometry:

$$\mathbf{P}' = \mathbf{P}\cos\theta + \mathbf{Q}\sin\theta$$

By seperating the components of $\mathbf{P}'$, we get the following equations:

$$\begin{aligned} P'_x &= P_x\cos\theta - P_y\sin\theta \\ P'_y &= P_y\cos\theta + P_x\sin\theta \end{aligned}$$

We can create a matrix form of this equation as follows:

$$\mathbf{P}' = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \mathbf{P}$$

Rotation Transforms

The matrix generated works for a 2D rotation, and therefore is only defined for a 2D vector. For a 3D vector, we need to extend the 2×2 matrix to a 3×3 matrix. We do this by setting the other values to match an identity matrix. This gives us the 3D rotation matrix around the Z-axis as:

$$\mathbf{R}_z(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Using what we have learned, we can define X and Y-rotation matrices:

$$\mathbf{R}_x(\theta) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta \\ 0 & \sin\theta & \cos\theta \end{bmatrix}$$

$$\mathbf{R}_y(\theta) = \begin{bmatrix} \cos\theta & 0 & \sin\theta \\ 0 & 1 & 0 \\ -\sin\theta & 0 & \cos\theta \end{bmatrix}$$

Translation Transforms

So far, we have built a collection of 3×3 matrices which allow us and rotate our 3D vectors. However, we have so far not dealt with translation - or the movement - of our vectors. Translation doesn't affect the orientation or scale of a vector, and is therefore easier to deal with.

The equation to transform a point **P** from one coordinate space to another can be defined as:

$$\mathbf{P}' = \mathbf{MP} + \mathbf{T}$$

For multiple transforms, we get the following equation:

$$\begin{aligned}\mathbf{P}' &= \mathbf{M}_2(\mathbf{M}_1\mathbf{P} + \mathbf{T}_1) + \mathbf{T}_2 \\ &= (\mathbf{M}_2\mathbf{M}_1)\mathbf{P} + \mathbf{M}_2\mathbf{T}_1 + \mathbf{T}_2\end{aligned}$$

Four-Dimensional Transforms

As we apply more transforms, the concatenated calculation becomes more complex. To overcome this, we build four-dimensional (4×4) matrices, and work with four-dimensional *homogeneous coordinates*.

A 3D point **P** is transformed into a 4D homogeneous coordinate by setting the fourth component, w, to 1. That is, $< P_x, P_y, P_z >$ becomes $< P_x, P_y, P_z, 1 >$.

We create our 4×4 transform matrix, **F** as follows:

$$\mathbf{F} = \begin{bmatrix} M_{11} & M_{12} & M_{13} & T_x \\ M_{21} & M_{22} & M_{23} & T_y \\ M_{31} & M_{32} & M_{33} & T_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Inverting a Four-Dimensional Transform Matrix

We have looked at using Gauss-Jordan elimination to invert our matrices. If you have read around determinants, you will also see that we can use these to perform faster inversion for small matrices.

If we look at our original transformation equation, we can invert it using the following equation:

$$\mathbf{P} = \mathbf{M}^{-1}\mathbf{P}' - \mathbf{M}^{-1}\mathbf{T}$$

If we apply this to our four-dimensional matrix, we can simplify our inversion to the following:

$$\mathbf{F}^{-1} = \begin{bmatrix} M_{11}^{-1} & M_{12}^{-1} & M_{13}^{-1} & -(\mathbf{M}^{-1}\mathbf{T})_x \\ M_{21}^{-1} & M_{22}^{-1} & M_{23}^{-1} & -(\mathbf{M}^{-1}\mathbf{T})_y \\ M_{31}^{-1} & M_{32}^{-1} & M_{33}^{-1} & -(\mathbf{M}^{-1}\mathbf{T})_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Quaternion Mathematics

The final form of transform approach we will look is *quaternions*. Quaternions are useful as they enable us to save on storage over a rotation matrix, require fewer arithmetic operations, and allow us to produce smoother animation.

A quaternion can be defined as follows:

$$\mathbf{q} = < w, x, y, z > = w + xi + yj + zk$$

A quaternion is often written as $\mathbf{q} = s + \mathbf{v}$ where s represents the scalar or w-component of $\mathbf{q}$ and $\mathbf{v}$ the vector component.

Multiplying Quaternions

To multiply two quaternions together, we use the following equation:

$$\begin{aligned}\mathbf{q}_1\mathbf{q}_2 =&(w_1w_2 - x_1x_2 - y_1y_2 - z_1z_2)\\ &+(w_1x_2 + x_1w_2 + y_1z_2 - z_1y_2)i\\ &+(w_1y_2 - x_1z_2 + y_1w_2 + z_1x_2)j\\ &+(w_1z_2 + x_1y_2 - y_1x_2 + z_1w_2)k\end{aligned}$$

This is worse than trying to remember the actual calculation for the cross product. Luckily, we can simplify this calculation by using the scalar-vector form of the quaternions:

$$\mathbf{q}_1\mathbf{q}_2 = s_1s_2 - \mathbf{v}_1\cdot\mathbf{v}_2 + s_1\mathbf{v}_2 + s_2\mathbf{v}_1 + \mathbf{v}_1\times\mathbf{v}_2$$

Quaternions as Rotations

We won't go into the reasons why the following is the transformation matrix for a quaternion. However, read the maths text to get an idea.

We can represent a rotation, θ, around an (unit length) axis, **v**, in quaternion form as follows:

$$\mathbf{q} = \cos\frac{\theta}{2} + \sin\frac{\theta}{2}\mathbf{v}$$

Multiplying these quaternions together provides concatenated rotations. We can convert this to a matrix form as follows:

$$\mathbf{R_q} = \begin{bmatrix} 1-2y^2-2z^2 & 2xy-2wz & 2xz+2wy \\ 2xy+2wz & 1-2x^2-2z^2 & 2yz-2wx \\ 2xy-2wy & 2yz+2wx & 1-2x^2-2y^2 \end{bmatrix}$$

Note

When looking at the matrices glm creates in the debugger, they will not look as you expect. This is because the data is arranged in a manner like this:

$$\begin{bmatrix} M_{11} & M_{21} & M_{31} & 0 \\ M_{12} & M_{22} & M_{32} & 0 \\ M_{13} & M_{23} & M_{33} & 0 \\ T_x & T_y & T_z & 1 \end{bmatrix}$$

This is just how the matrix is laid out in memory (you can consider the matrix transposed). Considering that vectors are laid out in their transposed format, this does make sense. You just need to realise this when you look at the matrices in the debugger.

Recommended Reading

Chapter 4 of the mathematics text goes into far more detail of these concepts than we do here. Review the chapter, and attempt some of the questions within it. Some important parts we haven't covered are arbitrary axis rotation and spherical linear interpolation using quaternions.