Jasper Slaff
Math 2270 Spring 2017

Using Homogeneous Coordinate Systems to Perform Linear Translations, Rotations, and Scaling on 3 Dimensional Objects in Computer Graphics and How These Operations Relate to Robotics

Intro:

-Homogeneous Coordinates

Homogeneous coordinates are ubiquitous in computer graphics because they allow common vector operations such as translation, rotation, and scaling to be represented as a matrix by which the vector is multiplied. The same applies to their presence in robotics. The use of these coordinates allows us to use matrices to accurately determine an object's location relative to our own. This skill is invaluable in robotic movement. Knowing where you are relative to an object or vice versa can allow you to move to the object's location by simple matrix multiplication. This will be talked about more in-depth in later sections.

Translations:

-Graphics

Computer graphics can be represented by a multitude of objects on a screen all with their own coordinate system and location relative to the "world" so to speak. This concept is known as object oriented programming. Based on the program you are running, there can be any number of objects represented in this world and each one can be moved to the location of another using matrix translation. For example, say we have an object like a ball that is at location (x,y,z) in the 3 dimensional world that is represented in your program. If we think of these coordinates as a column vector like so:

$$\begin{pmatrix} x \\ y \\ z \end{pmatrix}$$

Then we can move it along a coordinate plane simply by multiplying it by the matrix;

$$A=\begin{pmatrix} 1 & 0 & 0 & a \\ 0 & 1 & 0 & b \\ 0 & 0 & 1 & c \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

In this example, **a** is the distance you would like to move along the x-axis **b** along the y-axis and **c** along the z-axis. However, when we try and multiply these together or, more specifically, a

nx**n** matrix by a vector that has less than **n** entries, the multiplication is invalid. To account for this, our matrix becomes:

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

This allows us to multiply the vector by the matrix without affecting the vector space by simply adding 1 times the **a**, **b**, or **c** values.

-Robotics

This concept applies in robotics as well. We can do the same translations and get the same results, however, here we think of the "world" as the origin or the point that doesn't move on the robot. For example, in the paper by Jennifer Kay (Introduction to Homogeneous Transformations & Robot Kinematics) she mentions that if you have a robotic arm where the origin is a part of a 3 dimensional coordinate system and you have a joint at distance **L1** from the origin and a gripper that is the distance **L2** from the joint along the x-axis, we could calculate the distance between the origin and the gripper with the simple matrix translation:

$$\begin{pmatrix} 1 & 0 & 0 & L1+L2 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Applying this to the origin would move the x-axis by **L1**+**L2** and arrive at the location of the gripper.

<u>Rotations:</u>

-Graphics

Using the same idea that I talked about in the Translations section where objects have their own coordinate system, we can use the same matrix multiplication to perform rotations to the origin object in order to get it oriented the same way as the target object. For example, there are a few matrices that we can multiply a vector by to get different results.
The matrix:

$$\begin{pmatrix} \cos\Theta & -\sin\Theta & 0 & 0 \\ \sin\Theta & \cos\Theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

will rotate the object along the z-axis by Θ.
The matrix:

$$\begin{pmatrix} \cos\Theta & 0 & \sin\Theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\Theta & 0 & \cos\Theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

will rotate the object along the y-axis by Θ.
Finally, the matrix:

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\Theta & -\sin\Theta & 0 \\ 0 & \sin\Theta & \cos\Theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

will rotate the object along the x-axis by Θ. Using this, we can rotate objects relative to the "world" space of the program. This means that if we want to flip a car in a game or rotate a face to see all sides, all we have to do is multiply the coordinate vector that represents the location of the object by these matrices. Then it will rotate the object in that space by Θ with respect to the relevant axis.

-Robotics

This idea of rotating an object along an axis is extremely important when talking about robotic movement. For example, in the gripper problem, we imagine that the gripper was connected to a joint and that this joint could rotate about its z-axis (meaning the axis that appears to be coming out of the page). If I wanted to rotate the gripper 90 degrees so that I could approach an object like a small box from above, I would simply have to rotate the gripper's z-axis so that its gripping end was facing downward. After that, it would be a simple matter of translating the gripper downward to be at the box's location and closing the gripper. However, applying the 90 degree rotation is the focus of this section. Since we are applying it to the z-axis, we can use the matrix given above for the z-axis rotation. 90 degrees translates to pi/2 radians. cos(pi/2) is 0 and sin(pi/2) is 1 thus we are given the matrix:

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

The reason that the values are negated is because we were following the right hand rule and rotating the arm clockwise. This is assuming, of course, that you wanted to rotate the arm clockwise. If you are rotating the arm counter clockwise, you simply switch the 1 and the -1 in the sine value locations.

Scaling:

-Graphics

Scaling is a fairly easy-to-grasp concept and is used in computer graphics to portray distance from an object in 3D space. To scale a vector, you simply apply scalar multipliers to the vector. For example, our generic vector from earlier scaled to twice its size would be:

$$\begin{bmatrix} 2x \\ 2y \\ 2z \\ 1 \end{bmatrix}$$

The one on the bottom remains the same, meaning that the object has simply extended the range of its vertices. If we were to have a 2 at the bottom, the vector would be the same as it was without the scalar applied. This application would, of course, have to happen to all of the object's vertices. If it were applied to the center of the object, it would only double the distance it was from the origin or the center of the "world" space. Applying this to each vertex extends the object's size and makes it appear to grow. This creates the illusion that the object is getting closer in some contexts. These same concepts could be applied to computer vision.

Conclusion:

Here, I will present my source code and demonstrate how 2D graphics can use translations, rotations, and scaling to create a 3D illusion.