**Objective**

This homework assignment will be using what was taught in lecture as well as building off of the techniques you learned from the last homework assignment. This assignment is very similar to the first one, only now you’ll be creating shapes in three dimensions instead of just two. Most of the setup (class structure, windows app, etc) should be pretty similar. The movement of the geometry will also be similar, only with a third movement axis for translation and rotation.

Included in this assignment are the following:

* Example EXE of the final working program.

**Homework Requirements**

This homework assignment will be using what was taught in lecture as well as building off of the techniques you learned from the last homework assignment. This assignment is very similar to the first one, only now you’ll be creating shapes in three dimensions instead of just two. Most of the setup (class structure, windows app, etc) should be pretty similar. The movement of the geometry will also be similar, only with a third movement axis for translation and rotation.

Included in this assignment are the following:

* Example EXE of the final working program.

**Homework Instructions**

**NOTE: DO NOT USE SHAREDDEFINES.H FROM LAB 1.**

This file is specifically for use with lab 1. You'll need to create new constant buffers for this lab. Here's a tip, you'll need one constant buffer that contains ONE world matrix (not an array like lab 1), and another that contains the view and projection matricies. It's in your best interest to NOT use the constant buffers that came with lab 1.

**Creating the geometry**

* Much like the last assignment the geometry is going to be a series of triangles, only this time they will be used to make cubes and a 5 point star. Winding order matters here, if it’s not correct parts of your model will look invisible or inside out, so keep that in mind.
* I’d suggest starting by trying to either create the cube or the grid. The star is a bit complex and will be easier to do once you have a grasp of how it worked with the cube. If you’re stuck with any of the geometry, drawing it out will help more than you think.
* As for the cube grid, if you did the linelist outlines for the triangles last time, we’ll be using linelists again here. It may look complicated but it really isn’t, there are multiple ways you could approach this.

**Moving the geometry**

* Each object will need its own matrix, which will be updated with an X,Y, and Z velocity, as well as a rotation. When the object reaches the side of the cube, the object should “bounce” off of it. This can be done simply by reversing the applicable X, Y, or Z velocity. This is just like what needed to be done for the last lab, just in three dimensions instead of two.

**Moving the camera**

* Moving a camera is just like moving geometry, however one final step needs to be done before we use it as a camera matrix.
* Any world matrix becomes a camera matrix by inversing it. Vise versa, any camera matrix becomes a world matrix by inversing it. So to manipulate a camera matrix, we can simply, inverse it, do our translations and rotations, and inverse it back. Too easy.
* We can create a camera matrix using the XMMatrixLookToLH function, which will create a Left Handed camera matrix from a position, forward vector (the direction it’s looking), and an up direction (usually the World’s up direction, which is {0,1,0}).
* As for the actual input for movement, we’ve all used GetAsyncKeyState… and we’ll be using it again.

**Creating the vertex shader**

* Oh boy, the new stuff! This is fairly simple, and will use the same pixel shader as the last lab.
* The sample vertex shader we went over in lab is a good example of what we’re going to need to do here today. We’ll be making sure our input vertex, which has a float3 POSITION and a float4 COLOR, will be converted into world, view, and projection space, and then passed to the rasterizer stage.
* Looking at our pixel shader, it says it only takes in a color, but if you remember from lecture, there’s something else that is expected by the rasterizer stage.