COSE50581 Tutorial Week 2

Before starting the rest of the tutorial, your tutor will explain the process of adding in extras cubes, and making sure those cubes are using different world matrices.

You should now have:

* At least two cubes displayed on screen with no facets rendered incorrectly
* The cubes rotating differently on screen

If you have not finished these tasks, please ask your tutor for help fixing any problems you are having.

## Depth/Stencil View

A depth/stencil view lets the graphics pipeline check all the pixel fragment’s depth/stencil values on a render target. A pixel fragment is a pixel that has the potential of being written to the screen. Imagine we have a sphere and a box. The sphere is behind the box, so as they are passed through the rendering pipeline, the pixels from the sphere AND the pixels from the box are both put onto the render target.

These pixels are called Pixel Fragments. When a pixel reaches a certain point in the pipeline, it compares the pixel fragments depth value with the pixel fragment already in that position. If the new pixel fragments depth value is less than the pixel fragment that was already there, then the pixel fragment that was already there will be discarded, and the new pixel fragment will be kept on the render target.

So back to the sphere and box, say the sphere is rendered first. There is no geometry on the render target yet, so the entire sphere will be rendered. But when the box's pixels get rendered, we compare the depth values of the pixels from the box and the pixels from the sphere. Since the box is in front of the sphere, the spheres pixel fragments are discarded, and the box's pixel fragments are kept on the render target. After all geometry is drawn, the pixels left on the render target are the ones displayed on the screen.

The stencil part of the depth view is for advanced techniques, such as mirrors, which we will learn later.

If the two cubes you implemented last week overlap at all you may have noticed that they do not look like they are correctly positioned in the 3D scene, this is because we have not set a depth buffer yet.

## Setting up the Depth/Stencil buffer

1. We need two new interface objects. One is to store the depth/stencil view, the other is to store the depth/stencil buffer. Add these as private variables in your Application class:

ID3D11DepthStencilView\* \_depthStencilView;

ID3D11Texture2D\* \_depthStencilBuffer;

1. Now go to the InitDevice function. Before we create our depth/stencil buffer, we need to define it. We can do that by filling out a D3D11\_TEXTURE2D\_DESC, like when we filled out a back buffer. The texture format used gives space for two variables; 24 bits for the depth, and 8 bits for the stencil. The bind flags allow this texture will be bound to the output merger (OM) pipeline stage as a depth/stencil buffer.

D3D11\_TEXTURE2D\_DESC depthStencilDesc;

depthStencilDesc.Width = \_WindowWidth;

depthStencilDesc.Height = \_WindowHeight;

depthStencilDesc.MipLevels = 1;

depthStencilDesc.ArraySize = 1;

depthStencilDesc.Format = DXGI\_FORMAT\_D24\_UNORM\_S8\_UINT;

depthStencilDesc.SampleDesc.Count = 1;

depthStencilDesc.SampleDesc.Quality = 0;

depthStencilDesc.Usage = D3D11\_USAGE\_DEFAULT;

depthStencilDesc.BindFlags = D3D11\_BIND\_DEPTH\_STENCIL;

depthStencilDesc.CPUAccessFlags = 0;

depthStencilDesc.MiscFlags = 0;

1. Now that we have defined our depth/stencil buffer, we need to create it. We can create it using the CreateTexture2D method of the device interface. After we have created a depth/stencil buffer, we need to create the depth/stencil view that will be bound to the OM stage of the pipeline. We do this by calling CreateDepthStencilView() from the device interface. The first parameter is our depth/stencil buffer description, the second parameter is for a depth/stencil state (we don't have one so set it to nullptr) and the third is the returned depth/stencil buffer.

\_pd3dDevice->CreateTexture2D(&depthStencilDesc, nullptr, &\_depthStencilBuffer);

\_pd3dDevice->CreateDepthStencilView(\_depthStencilBuffer, nullptr, &\_depthStencilView);

1. The last thing we need to do to implement depth in our program is to bind it to the OM stage of the pipeline. Before we called the OMSetRenderTargets function and set the third parameter to NULL, since we didn't have a depth/stencil view. We have one now, so we can change that NULL parameter to our depthStencilView object we created from above. Locate this function in your code and change it appropriately.
2. Don’t forget to release COM objects, so make sure you add the following lines to your CleanUp method:

if (\_depthStencilView) \_depthStencilView->Release();

if (\_depthStencilBuffer) \_depthStencilBuffer->Release();

1. Finally we need to make sure we clear the depth/stencil view every frame, like we do with the render target view. We do this by calling the ClearDepthStencilView() method of the device context interface, this call should take place in your Draw function. The first value in the method call is the depth/stencil view we want to clear, the second is an enumerated type, or-ed together, specifying the part of the depth/stencil view to clear, the fourth parameter is the value we want to clear the depth to. We set this to 1.0f, since 1.0f is the largest depth value anything can have. This makes sure that all the objects are drawn on the screen. If we were to set 0.0f here, nothing would be drawn to the screen since all the depth values of the pixel fragments would be between 0.0f and 1.0f. The last parameter is the value we set the stencil to. We set this to 0 since we're not using it. Add the following line after you clear your render target:

\_pImmediateContext->ClearDepthStencilView(\_depthStencilView, D3D11\_CLEAR\_DEPTH|D3D11\_CLEAR\_STENCIL, 1.0f, 0);

You should now see that any cubes you have on screen are now organised in the correct manner, and have no depth issues. If needed, add in more cubes to ensure that the depth buffer is working correctly.

## Transformations

Next we will learn how to transform geometry in a 3D scene. Transformations make use of matrices, which the direct3D math library conveniently provides functions to help us out with.

Transformations in Direct3D use a 4x4 Matrix, Matrices in the direct3D math library are of the XMMATRIX type. Direct3D has a number of helper functions already included in the library, you can find more information on these built-in functions below:

### Scaling

Scaling Matrices will scale the size of an object in 3D space. The direct3D math library has functions that will compute transformations for us. The Scaling Matrix can be computed with the following function:

XMMATRIX XMMatrixScaling

(

FLOAT ScaleX, // x=axis scale

FLOAT ScaleY, // y-axis scale

FLOAT ScaleZ // z-axis scale

)

The result of this function is a returned matrix, which will be used to find the World space matrix.

### Rotating

Rotation Matrices are used to rotate an object in 3D space. There are 3 different matrices for rotating around each of the x, y, and z-axis's. Here are the three functions to create each of the rotation matrices:

XMMATRIX XMMatrixRotationX (

FLOAT Angle //Rotation angle in radians

)

XMMATRIX XMMatrixRotationY (

FLOAT Angle //Rotation angle in radians

)

XMMATRIX XMMatrixRotationZ (

FLOAT Angle //Rotation angle in radians

)

### Translating

There is one more type of transformation which we can use in our 3d scenes, translation. Translations are used to move an object in 3D space. The directX function is given below:

XMMATRIX XMMatrixTranslation (

FLOAT OffsetX, // Units translated on the x-axis

FLOAT OffsetY, // Units translated on the y-axis

FLOAT OffsetZ // Units translated on the z-axis

)

### Combining Transformations

We can combine multiple transformation matrices into one matrix. We do this by multiplying each one together. We have to do it in order to get the desired outcome. If we have a scaling matrix called 'S', a rotation matrix called 'R', and a translation matrix called 'T'.

O = S \* R \* T.

This will scale the object, rotate it around the origin, and then translate it. Ordering the matrix combinations differently will give different results. If we were to put the translation matrix first:

0 = T \* S \* R

Our object would be rotating around where it was originally, not where it is now. It would create an orbit effect instead of a spinning effect.

We know that the World matrix describes the objects position, rotation, and size in the world compared to the other objects in the scene. To create a world matrix for an object, we use transformations. Each object in the scene needs to have its own world space matrix.

Now extend your current implementation to use multiple transformations to create a solar system effect using cubes for the planets! Your solar system should have the following features:

1. A ‘sun’ planet spinning around the origin (0, 0, 0).
2. At least two other planets spinning, but also rotating around the central ‘sun’
3. At least two moons spinning, but also rotating around the planets which are rotating around the ‘sun’.

All the objects in your solar system should be scaled appropriately.

## Render States

In Direct3D, things stay in their current state. Nothing goes back to its "default" state. An example is our primitive topology. Once we set it, it stays a triangle list until we explicitly change it. Like our vertex and index buffers. Once we bind them to the pipeline, they stay there until we change them.

Render states encapsulate settings that can be used to configure Direct3d; we are now going to set up the rasterizer’s state using ID3D11RasterizerState.

1. First we need a declaration of the interface object which we will use to set the render state, this should be added as a private variable inside your application class.

ID3D11RasterizerState\* \_wireFrame;

1. Next, make sure you add appropriate code inside your cleanup method as before.
2. Now we need to declare a new D3D11\_RASTERIZER\_DESC structure called wfdesc. This will hold the definition of our rasterizer state we want to create.

The parameters inside this structure we are interested in are:

FillMode – We can use D3D11\_FILL\_WIREFRAME for wireframe rendering or D3D11\_FILL\_SOLID for solid rendering.

CullMode - We can put D3D11\_CULL\_NONE to disable culling, D3D11\_CULL\_FRONT for front face culling (so the front face will not be rendered), or D3D11\_CULL\_BACK to cull the back face, which is default.

FrontCounterClockwise - This is either set to true or false, true means that if the triangles vertices are rendered clockwise to the camera, then that is the front face. False is the opposite.

Creating a rasterizer state to do wireframe rendering is done as follows:

D3D11\_RASTERIZER\_DESC wfdesc;

ZeroMemory(&wfdesc, sizeof(D3D11\_RASTERIZER\_DESC));

wfdesc.FillMode = D3D11\_FILL\_WIREFRAME;

wfdesc.CullMode = D3D11\_CULL\_NONE;

hr = \_pd3dDevice->CreateRasterizerState(&wfdesc, &\_wireFrame);

Add the code above at the end of your InitDevice function. This code starts by making sure the memory is cleared. Then it sets the fillmode to D3D11\_FILL\_WIREFRAME so Direct3D renders our cubes in wireframe, finally it sets cullmode to D3D11\_CULL\_NONE so we can see the backs of the cubes as they spin.

After we have filled out the D3D11\_RASTERIZER\_DESC structure, we need to create the new render state. The render state will be bound to the RS stage of the pipeline, so we create the render state with the ID3D11Device::CreateRasterizerState() method. The first parameter is the description of our render state, and the second is a pointer to a ID3D11RasterizerState object which will hold our new render state.

1. Now that we have created the render state, we need to set it, or bind it to the RS stage of the pipeline. We can do this by calling the method ID3D11DeviceContext::RSSetState(). The only parameter here is the render state we want to bind.

Sometimes objects will use different render states in your scenes, so setting the render state when initializing your scene is not usually what you want. You will need to set the render state before every object or group of objects that want to use that render state. To use the default render state, all you have to do is pass nullptr to this function. We set the render state like this:

\_pImmediateContext->RSSetState(\_wireFrame);

Now enable wireframe rendering using the code above, you should see your solar system in wireframe!

1. Create a new rasterizer state for solid objects, and try rendering some objects in wireframe and some as solid objects.
2. Finally, add a key to switch all objects from solid to wireframe. This will be useful for later tutorials when you are working with more complex geometry.

### Additional Task

1. Create a randomly generated asteroid belt (of cubes) around one of the planets. The belt should have at least 100 different cubes all spinning and rotating around the planet they orbit. You may want to use the c++ rand() function to help you.