Graphics – Tutorial for Week 4

Start an Implementation Log

The work you do from this week onwards will be working towards your final submission for assessment 2. As part of that final submission, you will be required to submit an implementation log that details the work you have done on this assessment from this week onwards.

Your implementation log should include the work you do in the practical sessions for this module, as well as the work you do on your own time. Each log entry must contain:

- The date
- The number of hours worked on this module
- what was implemented, how you tested it, problems encountered and how you fixed those problems.

Your implementation log will be used during the marking of your assessment, as well as being worth 20% of the marks (see the rubric in the assessment specification) so it is very important. There is no specified layout for your log or any required word count. However, it must be comprehensive.

Introduction to the exercises this week

This week, you will be implementing a scene graph and using this to render a robot made out of cubes.

Starting Point

I have provided a starting point for the work this week (and for the rest of this module) in DirectXFramework.zip. This provides an implementation of a basic framework (DirectXFramework) that initialises Direct3D 11 and renders scenes. The overrides of the Initialise, Update, Render and Shutdown methods exposed by the Framework class are used to make the appropriate actions on the scene graph.

Included with the starting point is a class called DirectXApp that inherits from DirectXFramework. It provides the starting point for two overrides of additional virtual methods in DirectXFramework. These are:

CreateSceneGraph:	Used to build the nodes for the scene graph and attach them to the graph.
UpdateSceneGraph:	Used to apply any updates to any of the nodes in the scene graph before the scene graph is recursively updated.

You will see an example of how these can be used in the lecture this week.

Implementing Scene Graph

Note that the starting point provided will compile, but it will not link since no implementation of the SceneGraph class has been provided. Your first task is to implement SceneGraph.

An implementation of SceneNode is provided (in SceneNode.h). No .cpp file is provided for this since all implementation is provided in the header file.

The header file for SceneGraph (SceneGraph.h) has been provided. You need to implement the methods for the class in SceneGraph.cpp.

The methods should perform the following functions:

Initialise — Call the Initialise method on each child node. If *any* node returns false, then Initialise should return false. If all child nodes return true, then Initialise should return true.

Update — Update the cumulative world transformation for itself (i.e., call:

```
SceneNode::Update(worldTransformation);
```

and then call the Update method for each child node, passing the combined world transformation to those nodes.

Render — Call the Render method on each child node.

Shutdown — Call the Shutdown method on each child node.

Add — Add the specified node to the collection of child nodes.

Remove — Remove the specified node from the scene graph. If the node has children, call Remove on all child nodes.

Find — If we are the node being searched for, return a pointer to ourselves. If not, call Find on all child nodes. If the call to Find succeeds, return the pointer to the found node otherwise return nullptr.

If you look at the code for the Initialise method in DIrectXFramework.cpp, you will see that the main scene graph is created just before a call to CreateSceneGraph. Then the scene graph's Initialise method is called.

Implement a SceneNode type to render a Cube

The next step is to implement a class that inherits from SceneNode which creates a cube of the same size as the cubes rendered last week (i.e. 2 units on each side). You might call this class CubeNode. You should be able to specify the colour used for the material colour of the cube in the constructor, as well as the name of the node. You can create this class using the hard-coded vertices and indices for a cube as you have previously seen or you could use the ComputeBox method from the GeometicObject class you saw last week (this would give you a head start on exercise 2 for this week).

In the Initialise method, you need to do all of the initialisation steps (build the geometry, specify the vertex layout, load and compile the shaders and build the constant buffer. You do not need to build the swap chain, etc, since that is handled by DirectXFramework.

In the Render method, you should render the cube using the _worldTransformation matrix as its position in world space. Note that you should NOT clear the render target or depth stencil buffer in the Render method for the cube since this is done before the scene graph is rendered in DirectXFramework. In addition, you should NOT make a call to Present() since this is also done in DirectXFramework. All you need to is perform the steps needed to render the cube.

To implement these methods, you will need access to the Direct3D device and device context information. Methods have been provided in DirectXFramework to retrieve this information.

- A static method has been provided to access the current instance of DirectXFramework. You can call this as follows:

```
DirectXFramework::GetDXFramework()
```

- Once you have this, GetDevice() and GetDeviceContext() methods are available to return ComPtrs to the appropriate interfaces. For example, to access the device, you can use:

```
ComPtr<ID3D11Device> device =
            DirectXFramework::GetDXFramework()->GetDevice();
```

  To access the device context, you can use:

```
ComPtr<ID3D11DeviceContext> deviceContext =
      DirectXFramework::GetDXFramework()->GetDeviceContext();
```

In your CubeNode, you do not need to provide an override of the Update method of SceneNode since it already contains the required functionality. You only need to provide an override of the Shutdown method of SceneNode if you need to perform additional functionality in the Shutdown method.

Other Useful DirectXFramework Methods

There are some other methods implemented in DirectXFramework that you might find useful as you are using it. These are:

const Matrix& GetProjectionTransformation()	`Return projection matrix.
void SetBackgroundColour(Vector4 colour)	Set background colour for the scene

A starting point for a Camera class has been created in Camera.h and Camera.cpp. At the moment, it does very little other than initialise the view matrix and retrieve the view matrix. We will add to this in a few weeks, but it seemed a good time to add it now. The Camera object is created in DirectXFramework and when you are rendering your nodes, you can retrieve the view matrix from the camera using:

```
Matrix viewTransformation =
    DirectXFramework::GetDXFramework()->GetCamera()->GetViewMatrix();
```

Adding Shader Files

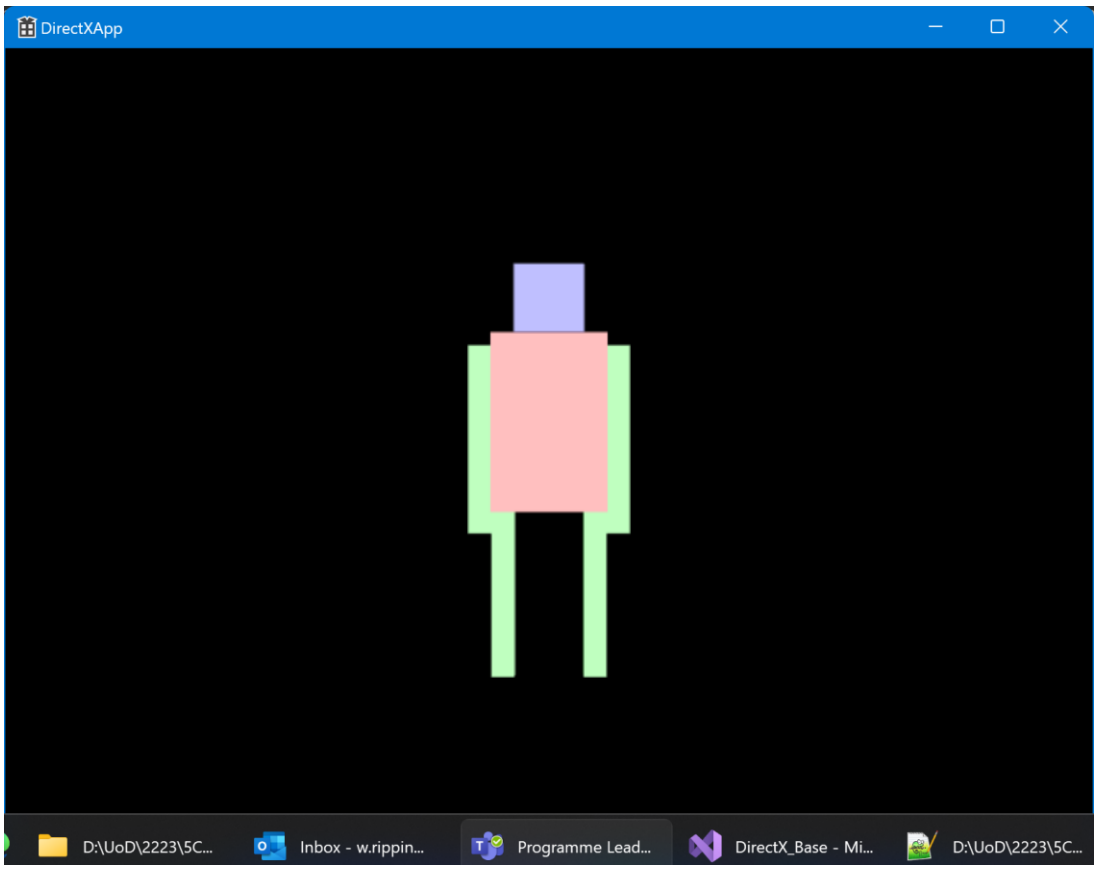
You will notice that the starting project does not contain any shader files. This is because the vertex and pixel shaders are handled by each node and it is quite possible that different types of node will need different vertex and pixel shaders.

You will need to add your shader file to the project. For this exercise, you can use the same shader file as you used last week. Copy the shader.hlsl file to the source file directory for the project and add it to the project in Visual Studio. Now you want to exclude it from the build so that Visual Studio does not try to compile it (remember it is compiled at run-time). Right click on the hlsl file in Visual Studio's solution explorer and select Properties. In the dialog box that is displayed, you will see the top item is "Excluded from Build". Change this to Yes and then press OK. This will now prevent Visual Studio from trying to compile your shader file at when you build your project.

Exercise 1 - Building a Cube Robot

Now it is time to test your code. One way would be to do a animation of cubes as seen in the lecture, but a slightly more interesting one is a robot made up of cubes. This will also give you more experience of combining matrices to apply different transformations.

The robot will be very crude, consisting of a very blocky body, head, arms and legs. This is what my version looked like, but you can choose whatever colours you want to use.



You should create a cube for each of the six components and then apply the following initial world transformations to get components of the right size and in the right place.

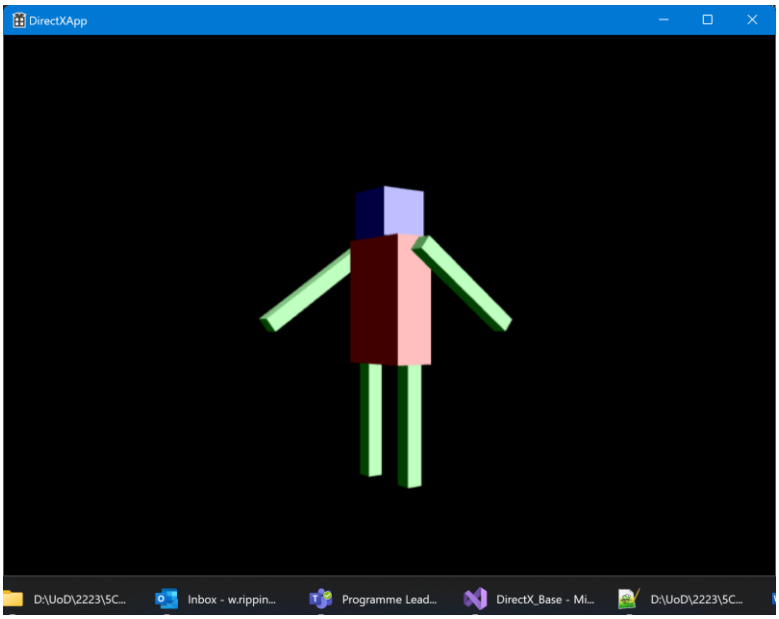
Component	Scale	Translation
Body	5, 8, 2.5	0, 23, 0
Left Leg	1, 7,5, 1	-4, 7.5, 0
Right Leg	1, 7,5, 1	4, 7,5, 0
Head	3, 3, 3	0, 34, 0
Left Arm	1, 8.5, 1	-6, 22, 0
Right Arm	1, 8.5, 1	6, 22, 0

These are the transformations I used, but feel free to change them if you wish.

For this to display correctly, I also modified the position and focal point of the camera by changing the vectors in Camera.cpp. I used the following values:

```
Camera Position      Vector3(0.0f, 20.0f, -90.0f);
Focal Point          Vector3(0.0f, 20.0f, 0.0f);
```

Once you have your robot displaying correctly, now try making updates to UpdateSceneGraph so that the robot rotates around the Y axis, while swinging its arms in different directions. For example:



Then try making it do other things.

Note. The way the directional lighting is being handled here is not really ideal since each instance of the cube node has it's own directional light information. Ideally, the information about the lights would be held by the DirectX Framework. We will add that later.

Also, you have hopefully realised that we will have 6 copies of the vertex buffer for the cube, 6 copies of the index buffer and so on. This is not very efficient. We will look at more efficient resource management next week.

Exercise 2 – Using additional geometric objects

Now try using different geometric objects (using the GeometricObject class you were introduced to last week) to use different shapes for the parts of the robots body. For example, I used a sphere for the head and a cylinder for the body below: