Graphics Part II: Code Tutorial for Week 4b

Dr P. Perakis

Preparation

We have provided a starting point for the work this week (and for the rest of this module) in `DirectXFramework.zip`.

Introduction

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

The provided implementation of the basic framework (`DirectXFramework`) 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.

Implementing Scene Graph

Note that the starting point provided will not compile 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.

Note: A fully working code for the implementation of the scene graph is provided to you in the `Exercise_04_1.zip` *file. Study the provided code, so that you can implement a* `SceneNode` *type to render a Cube.*

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.

```
CubeNode::CubeNode(std::wstring name) : SceneNode(name)
{
        _name = name;
        _materialColour = Vector4(0.0f, 1.0f, 1.0f, 1.0f);
}
```

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).

```
void CubeNode::BuildGeometry()
{
/* your code goes here… */
// Cube
}
```

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`.

```
bool CubeNode::Initialise()
{
        // access the device
        _device = DirectXFramework::GetDXFramework()->GetDevice();
        // access the device context
        _deviceContext = DirectXFramework::GetDXFramework()->GetDeviceContext();

        BuildGeometry();
        BuildShaders();
        BuildVertexLayout();
        BuildConstantBuffer();
        BuildRasteriserState();

        return true;
}
```

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. All you need to is perform the steps needed to render the cube.

```
void CubeNode::Render()
{
        //DirectXFramework::GetDXFramework()->GetCamera()->SetViewMatrix();
        Matrix _viewTransformation = DirectXFramework::GetDXFramework()-
>GetViewTransformation();
        Matrix _projectionTransformation = DirectXFramework::GetDXFramework()-
>GetProjectionTransformation();

        Matrix completeTransformation = _worldTransformation * _viewTransformation *
_projectionTransformation;

        CBUFFER cBuffer;
        //Set light
        cBuffer.AmbientLightColour = Vector4(0.3f, 0.25f, 0.25f, 1.0f);
        cBuffer.DirectionalLightVector = Vector4(11.0f, 0.0f, -10.0f, 1.0f);
        cBuffer.DirectionalLightColour = Vector4(1.0f, 1.0f, 1.0f, 1.0f);
        //Set material
        cBuffer.MaterialColour = _materialColour;
        //Set transforms
        cBuffer.WorldViewProjection = completeTransformation;
        cBuffer.World = _worldTransformation;

        // Update the constant buffer
        _deviceContext->VSSetConstantBuffers(0, 1, _constantBuffer.GetAddressOf());
        _deviceContext->UpdateSubresource(_constantBuffer.Get(), 0, 0, &cBuffer, 0, 0);

        //Render the cube
        UINT stride = sizeof(Vertex);
        UINT offset = 0;

        _deviceContext->IASetVertexBuffers(0, 1, _vertexBuffer.GetAddressOf(), &stride,
&offset);
        _deviceContext->IASetIndexBuffer(_indexBuffer.Get(), DXGI_FORMAT_R32_UINT, 0);
        _deviceContext->IASetPrimitiveTopology(D3D11_PRIMITIVE_TOPOLOGY_TRIANGLELIST);
        _deviceContext->DrawIndexed(_indexCount, 0, 0);
}
```

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. 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 the Cube into Scene

You will need finally to add the cube into the scene:

```
void DirectXApp::CreateSceneGraph()
{
      SceneGraphPointer sceneGraph = GetSceneGraph();

      // Add your code here to build up the scene graph

      // Create a cube node using your full lighting, shader, and triangle setup
      SceneNodePointer cube = make_shared<CubeNode>(L"CubeNode");

      // Add the cube to the scene
      sceneGraph->Add(cube);

}
```

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 with slight modifications.

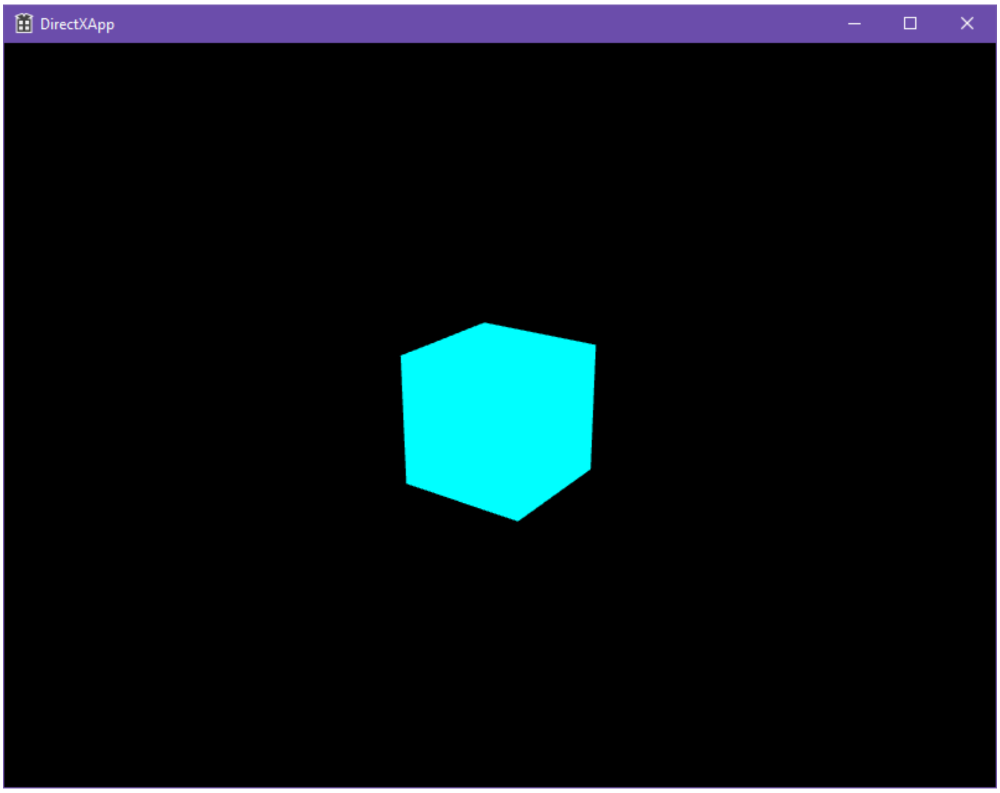
```
cbuffer ConstantBuffer
{
       matrix  worldViewProjection;
       matrix  worldTransformation;
       float4  materialColour;
       float4  ambientLightColour;
       float4  directionalLightColour;
       float4  directionalLightVector;
};

struct VertexIn
{
       float3 InputPosition : POSITION;
       float3 Normal         : NORMAL;
};

struct VertexOut
{
       float4 OutputPosition       : SV_POSITION;
       float4 Colour               : COLOR;
};

VertexOut VS(VertexIn vin)
{
       VertexOut vout;

       // Transform to homogeneous clip space.
       vout.OutputPosition = mul(worldViewProjection, float4(vin.InputPosition,
1.0f));

       // calculate the diffuse light and add it to the ambient light
    float4 vectorBackToLight = -normalize(directionalLightVector); // directional
light
    //float4 vectorBackToLight = -normalize(directionalLightVector -
mul(worldTransformation, float4(vin.InputPosition, 1.0f))); // point light

       float4 adjustedNormal = normalize(mul(worldTransformation, float4(vin.Normal,
0.0f)));
    float diffuseBrightness = saturate(dot(adjustedNormal, vectorBackToLight));

    vout.Colour = materialColour * saturate(ambientLightColour + diffuseBrightness *
directionalLightColour); // ALL

    //vout.Colour = materialColour; // material only
    //vout.Colour = saturate(ambientLightColour); // ambient only
    //vout.Colour = saturate(diffuseBrightness * directionalLightColour); // Specular
only

    return vout;
}

float4 PS(VertexOut pin) : SV_Target
{
        return pin.Colour;
}
```

Note: A fully working code for the implementation of the scene graph is provided to you in the `Exercise_04_2.zip` *file. Study the provided code, so that you can proceed with Exercise 1.*

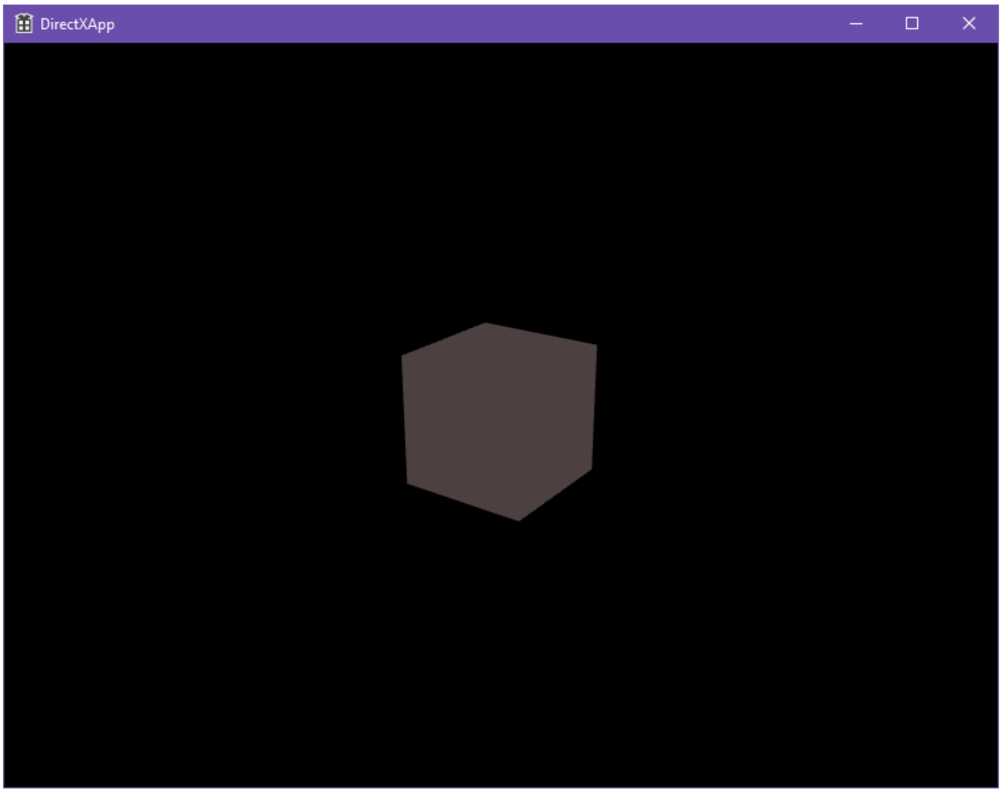
The output of the executable

After compiling and running the executable, the application window should look like this:

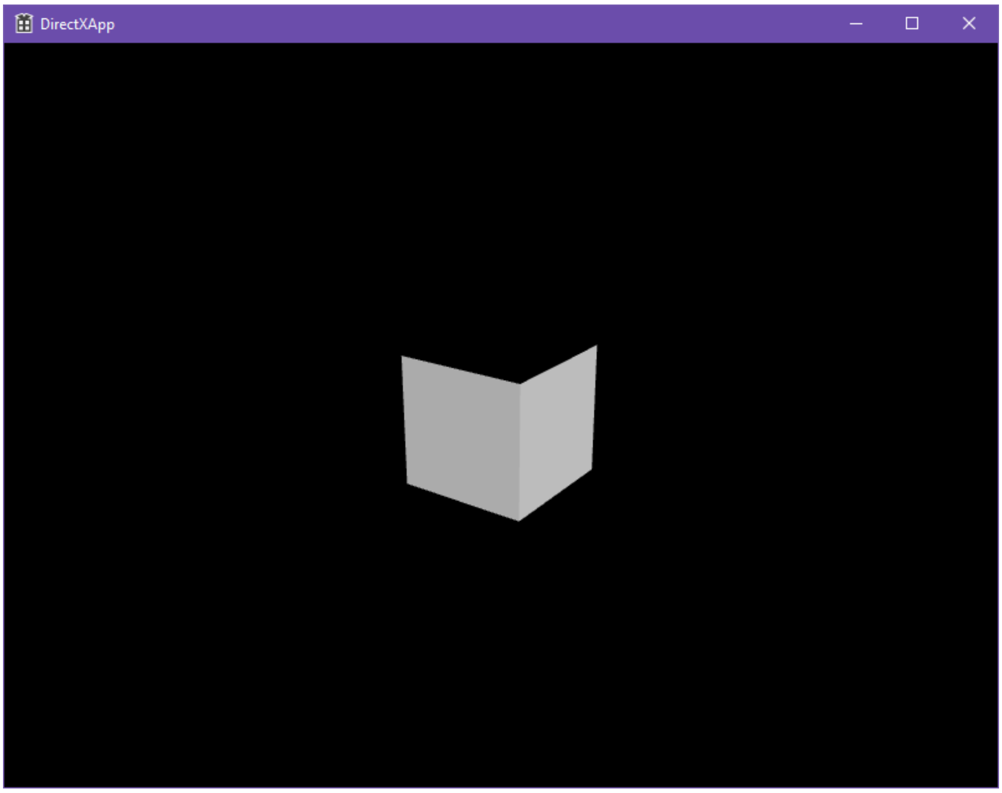
Material Colour only!



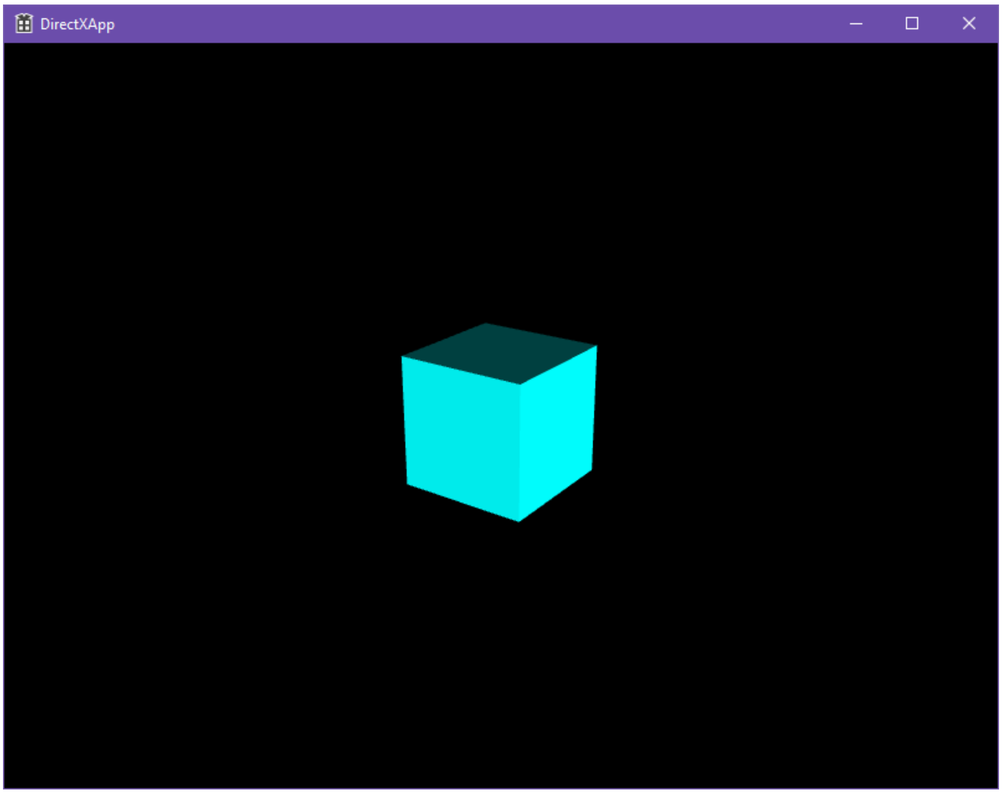
Ambient Lighting only!



Diffuse lighting only!



Final colouring/shading!


DirectXApp