Graphics Part II: Code Tutorial for Week 5

Dr P. Perakis

Preparation

We have provided a fully working code for the implementation of the scene graph and a `CubeNode` class to render a Cube with texture in the `Exercise_04_3.zip` file. This code can be used as a starting point for the following work, or you can use any working code you have up to now. We have also provided the `Exercise_05_0.zip` file which you can use as starting point. You will also need to use the files provided in the `Exercise_05_Files.zip` file.

Introduction

This week, the goal is to get a complete model loaded from a file and rendered on the screen. The model will be loaded using the Open Asset Import Library (ASSIMP).

We have provided the built version of ASSIMP for you as well as other supporting classes you have to use in current implementation. These files are provided in `Exercise_05_Files.zip`.

However, you might want to look at the code in `ResourceManager.cpp` that handles the ASSIMP structures since ASSIMP is not well documented and there was quite a bit of trial and error, as well as looking at ASSIMP code, to figure out what was really required.

You should not think of these as finished articles. There may be bugs in this code and it certainly could benefit from enhancements. Feel free to completely ignore it if you wish and write your own resource manager.

The Resource Manager

The first thing about the files provided in `Exercise_05_Files.zip` file that you need to note is that it does include the executable code for ASSIMP in the form of a dynamic link library (DLL). Because you will be downloading this from a site on the internet, by default, any executable files in it will be blocked from executing. To prevent this happening, download the zip file and BEFORE unzipping it, right click on the zip file in File Explorer, select Properties and then select Unblock. Now you will be able to unzip it as normal.

The files included are as follows:

`Assimp\`	The folder containing the lib, bin and include files needed for ASSIMP.
`ResourceManager.h` and `.cpp`	A simple resource manager that loads meshes from files using ASSIMP.
`Mesh.h` and `.cpp`	The classes that represent meshes, sub-meshes and materials.
`MeshNode.h` and `.cpp`	The classes that represent mesh node, for using it in scene graph.
`WICTextureLoader.h` and `.cpp`	Classes provided by the DirectX Toolkit to load images and create textures.
`white.png`	A white texture file for use as a default texture

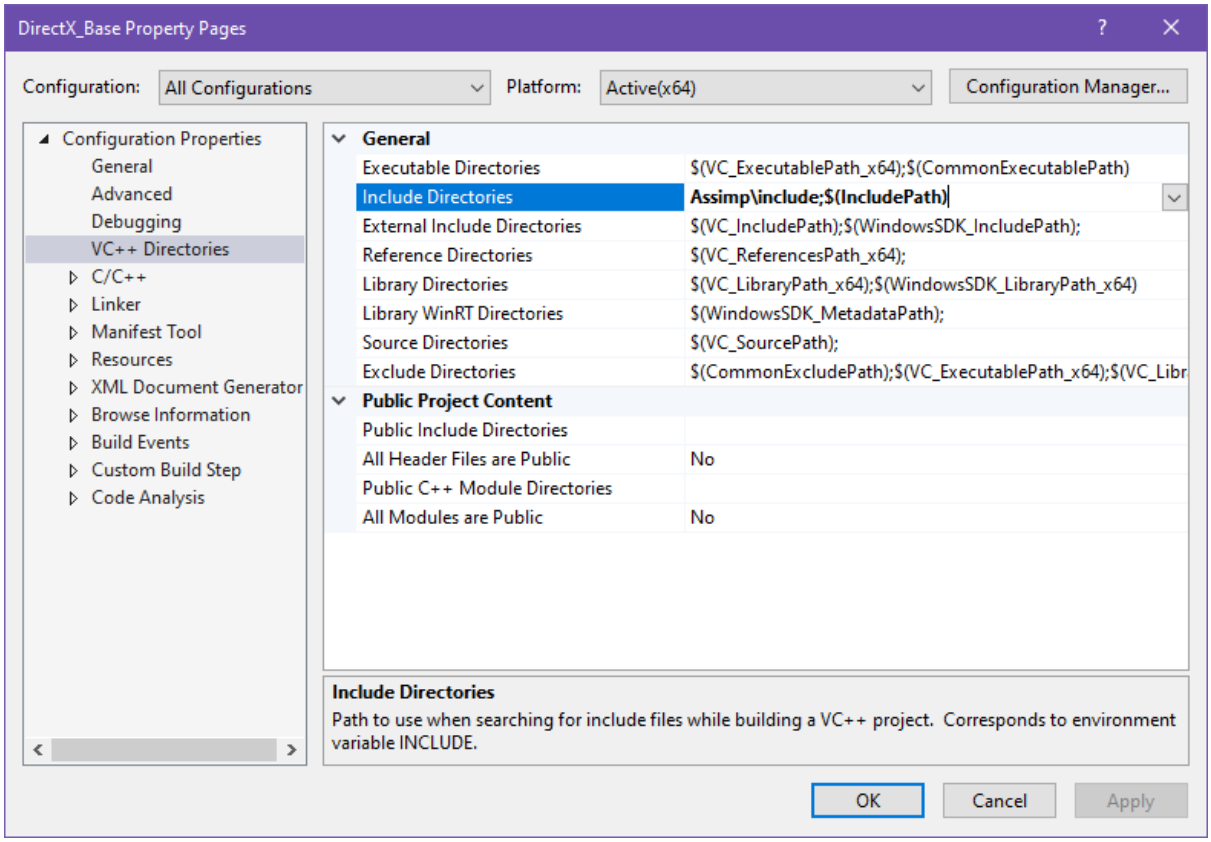
You should copy them to the folder containing your code and add the `ResourceManager` and `Mesh` files to your project.

There are a few things you need to do to configure your project to build and use ASSIMP. These are as follows:

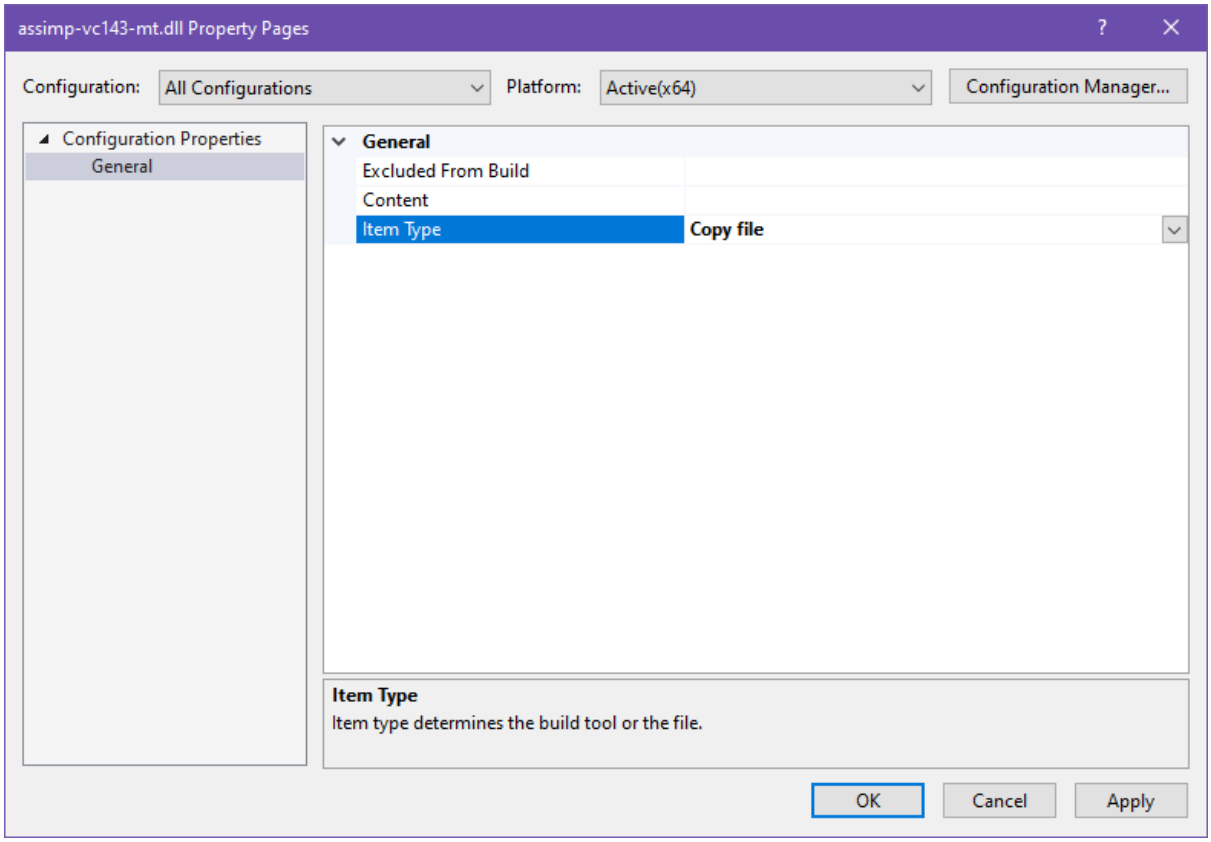
1. Add the `Assimp\include` folder to the C++ Include directories for the project. Open the properties for your project and look for VC++ Directories. Assuming you have copied the `Assimp` folder into the same folder as your source files, you should just add the following to the Include Directories:

```
Assimp\Include;
```

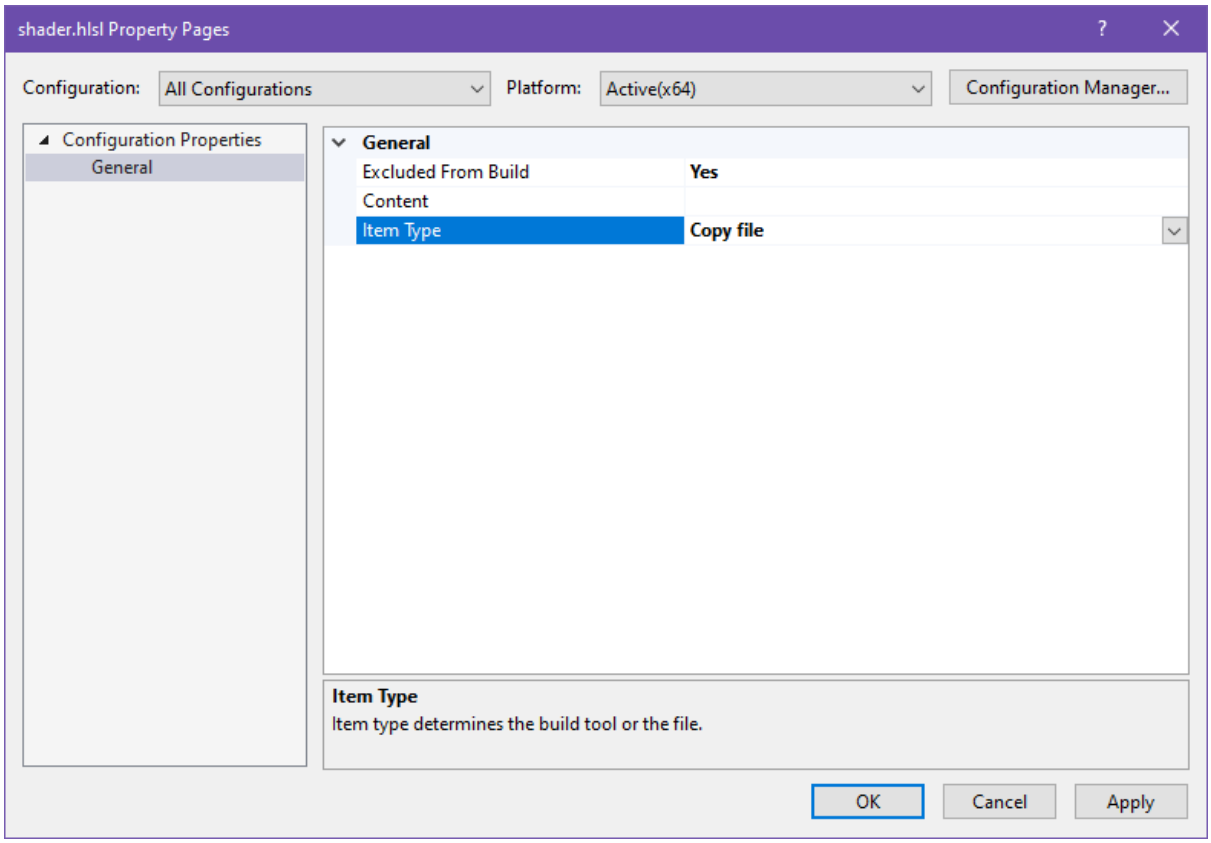
You can see this below:



2. Copy the file `assimp-vc143-mt.dll` from the folder `Assimp\bin\release` to the same folder as your source code files. Now add this to your project as an existing item so that it shows up in the Solution Explorer. You now need to change the properties on this item so that Visual Studio copies it to the same folder as your executable file whenever you do a project build. To do this, right-click on the dll in Solution Explorer and select Properties. In the Item Type field, change this to Copy File as you can see below:



While you are doing this, you should also make the same property change for your shader HLSL file so that this is always copied to the same place as your executable.



Now you should be able to build your solution with the new files.

How you use the `ResourceManager` is up to you.

One way would be to add it to the `DirectXFramework` class, using:

```
#include "ResourceManager.h"
```

You cannot add it as a static variable or class variable since the constructor needs to be able to access the device and device context information for the framework.

So you can declare it in the class as:

```
shared_ptr<ResourceManager> _resourceManager;
```

You would also need to provide a static method in `DirectXFramework` to retrieve the shared pointer to the resource manager so that nodes can access it by creating a getter for it:

```
inline shared_ptr<ResourceManager> GetResourceManager() { return _resourceManager; }
```

Then you create a new instance of the resource manager inside the `DirectXFramework::Initialise()` after the initialisation of the DirectX framework is complete, but before `CreateSceneGraph` is called using:

```
_resourceManager = make_shared<ResourceManager>();
```

The MeshNode class

For using the model mesh in the scene graph you need to use the `MeshNode` class.

Once you have a reference to the shared pointer to the resource manager in your node class, you can retrieve the mesh inside the `MeshNode::Initialise()` for the object in your Initialise method by doing the following:

```
_resourceManager = DirectXFramework::GetDXFramework()->GetResourceManager();
_mesh = _resourceManager->GetMesh(_modelName);
```

where `_modelName` is the name of the main model file for your object (in my case it was `airplane.x`).

To ensure proper cleanup, you should do the following in the `Shutdown` method for your node:

```
_resourceManager->ReleaseMesh(_modelName);
```

Rendering the model

The general process to render an object once you retrieved its mesh is:

1. Setup the common fields in the constant buffer that will be common for each sub-mesh (such as the transformation matrices, lighting, colours etc).
2. For each submesh:
    a. Populate the constant buffer fields that are specific for the sub-mesh. These will be things such as the material diffuse and specular colour, shininess, etc and send the constant buffer through to both the vertex shader and pixel shader.

We have not needed this yet because we only use the material colour in the shader, so a default white material colour suffices.

For now, we have been doing all of our calculations in the vertex shader, but we will soon start to do calculations in the pixel shader as well that will need access the constant buffer (such as per-pixel lighting). So, we need to add `PSSetConstantBuffers` to our rendering code in our node classes to make the constant buffer visible to our pixel shader. Put this line just after your call to `VSSetConstantBuffers` for the vertex shader.

b. Retrieve the vertex buffer and index buffer from the sub-mesh and pass them through to DirectX. Note that the vertex format always has a position, normal and texcoord field, even if no texture is being used so that we do not have to mess with changing vertex formats and can use a common vertex shader.

c. Do all of the other calls needed (setting the vertex layout, primitive topology, etc).

d. If the sub-mesh has texture coordinates (use the HasTexCoords method to find out), select the pixel shader that uses a texture and pass the texture retrieved from the material through to the pixel shader.

e. If the sub-mesh does NOT have texture coordinates, select the pixel shader that does not use a texture. Instead of creating two different pixel shaders we have added a default texture white.png for any case that the material texture is not available.

f. Now draw the sub-mesh using DrawIndexed.

A proposed simple implementation of the rendering function is included in the `MeshNode` class.

The Shader

As you will have realised from the above, you now need to have two pixel-shaders (they can be in the same file – they just need different names). One will multiply the calculated colour with the colour obtained from the texture while the other one will just use the calculated colour. Your node class will need to add code to compile and load another pixel-shader from the HLSL file.

Instead of doing so we can make modifications to the `ResourceManager::InitialiseMaterial` function so that it loads a default white texture when the texture file is not available for any reason.

```
if (texture == nullptr)
{
// Create a default white texture for use when texture file is not available.
// This happens for materials that do not provide a texture, and creates problems
// unless we provide a totally different shader just for those cases.
// That might be more efficient, but is a lot of work at this stage for little gain.
// If neither white.png is available, then the default texture will be null, i.e.
black.
	if (FAILED(CreateWICTextureFromFile(_device.Get(),
					_deviceContext.Get(),
					L"white.png",
					nullptr,
					texture.GetAddressOf()
							)))
	{
		texture = nullptr;
	}
}
```

The used shader is the following:

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

struct VertexIn
{
	float3 InputPosition : POSITION;
	float3 Normal        : NORMAL;
	float2 TexCoord      : TEXCOORD;
};

Texture2D Texture;
SamplerState ss;

struct VertexOut
{
	float4 OutputPosition	: SV_POSITION;
	float4 Colour		: COLOR;
	float2 TexCoord		: TEXCOORD;
};

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.TexCoord = vin.TexCoord;

	return vout;
}

float4 PS(VertexOut pin) : SV_Target
{
	//return pin.Colour;
	return pin.Colour * Texture.Sample(ss, pin.TexCoord);
}
```

Loading the model into scene

We have provided a bi-plane model for trying (`BiPlane.zip` – the model is `airplane.x`). This is a relatively straightforward model. If you start using other models, you might find yourself having to more work to get them to render correctly, particularly if they include transparent parts as we have not dealt with transparency and opacity in this module yet or looked at blending states. We will cover that later.

We can now load the airplane and make it rotate around a center by applying simple transformations.

Loading the plane and applying an initial translation transformation in the `DirectXApp::CreateSceneGraph()` function:

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

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

	// Create a plane road from a model file
	SceneNodePointer  nodePlane = make_shared<MeshNode>(L"Plane",
L".\\BiPlane\\airplane.x");
	nodePlane->Initialise();
	// Add the plane to the scene
	sceneGraph->Add(nodePlane);

	//Apply an initial translation
	Matrix ModelTrans = Matrix::CreateTranslation(-10.0f, 0.0f, 0.0f);
	sceneGraph->Find(L"Plane")->SetWorldTransform(ModelTrans);

}
```

Making the model rotate

Applying the rotation transformation in the `DirectXApp::UpdateSceneGraph()` function:

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

	// This method is called at the frame rate frequency set in the Framework class
(by default,
	// 60 times a second).  Perform any updates to the scene graph needed for the
next frame.

	// Update our model's world transformations
	float rotationAngle = -0.5f;
	//Compute the composite Transformation for Plane
	Matrix ModelYRot = Matrix::CreateRotationY(rotationAngle * XM_PI / 180.0f);

	Matrix worldTransformation = sceneGraph->Find(L"Plane")->GetWorldTransform();
	worldTransformation = worldTransformation * ModelYRot;

	sceneGraph->Find(L"Plane")->SetWorldTransform(worldTransformation);

}
```

We also need a getter to retrieve the world transformation of the model. We add this as a public member of the `SceneNode` class:

```
Matrix GetWorldTransform() const
{
	return _localTransformation;
}
```

Note: A fully working code for the implementation of the loading of a model is provided to you in the `Exercise_05_1.zip` *file.*

Making the Propeller Rotate

At the moment, the propeller on the bi-plane does not rotate and it would be nice if it could. We have separated out the propeller from the rest of the model and provided it in `BiPlane-SeparateProp.zip`. You will need to load the two models (`airplane.x` and `prop.x`) separately.

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

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

	// Create a plane node from a model file
	SceneNodePointer  nodePlane = make_shared<MeshNode>(L"Plane",
L".\\BiPlane_SeparateProp\\airplane.x");
	nodePlane->Initialise();
	// Add the plane to the scene
	sceneGraph->Add(nodePlane);

	// Create a propeller node from a model file
	SceneNodePointer  nodePropeller = make_shared<MeshNode>(L"Propeller",
L".\\BiPlane_SeparateProp\\prop.x");
	nodePropeller->Initialise();
	// Add the plane to the scene
	sceneGraph->Add(nodePropeller);

	//Apply an initial translation
	Matrix ModelTrans = Matrix::CreateTranslation(-10.0f, 0.0f, 0.0f);
	sceneGraph->Find(L"Plane")->SetWorldTransform(ModelTrans);
	sceneGraph->Find(L"Propeller")->SetWorldTransform(ModelTrans);

}
```

To rotate the propeller, you will need to rotate it around the Z axis relative to the airplane. There is a slight complication in that the X and Y co-ordinates of the propeller are not centred on the origin. They are actually centred on `x = 0.034129738` and `y = 0.420158183`. You will need to factor this in when figuring out the total transformation needed for the propeller.

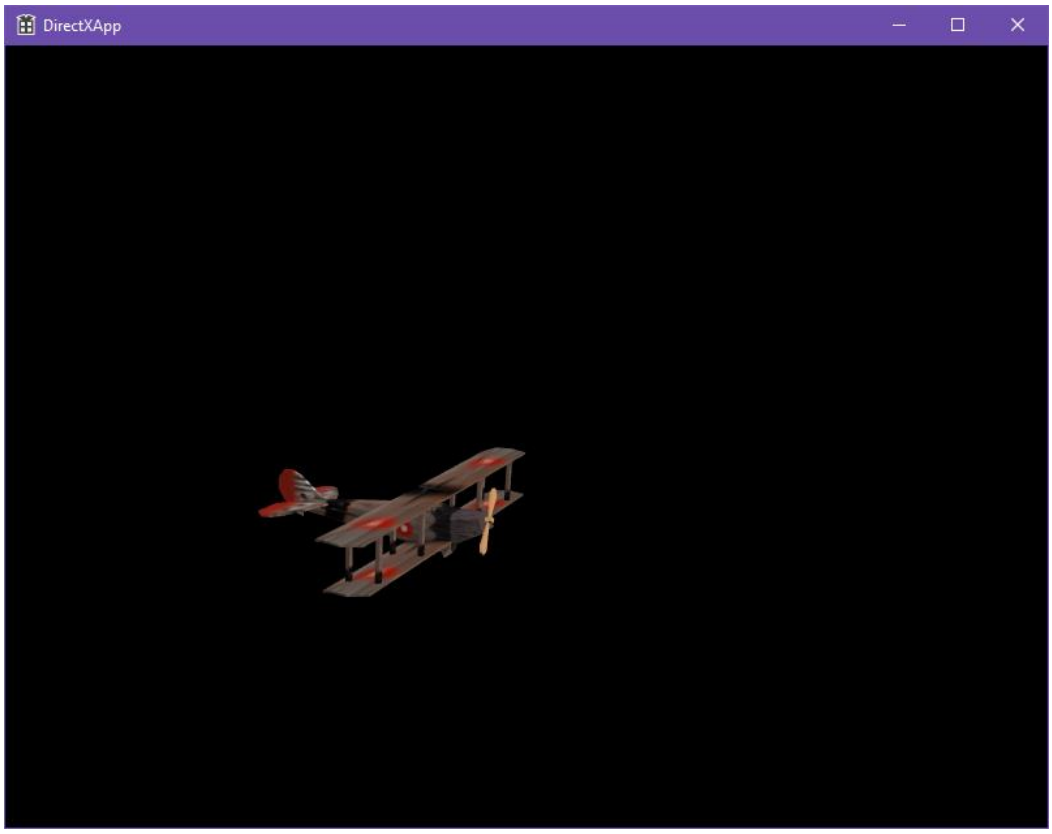
```
void DirectXApp::UpdateSceneGraph()
{
	SceneGraphPointer sceneGraph = GetSceneGraph();

	// This method is called at the frame rate frequency set in the Framework class
(by default,
	// 60 times a second).  Perform any updates to the scene graph needed for the
next frame.
```

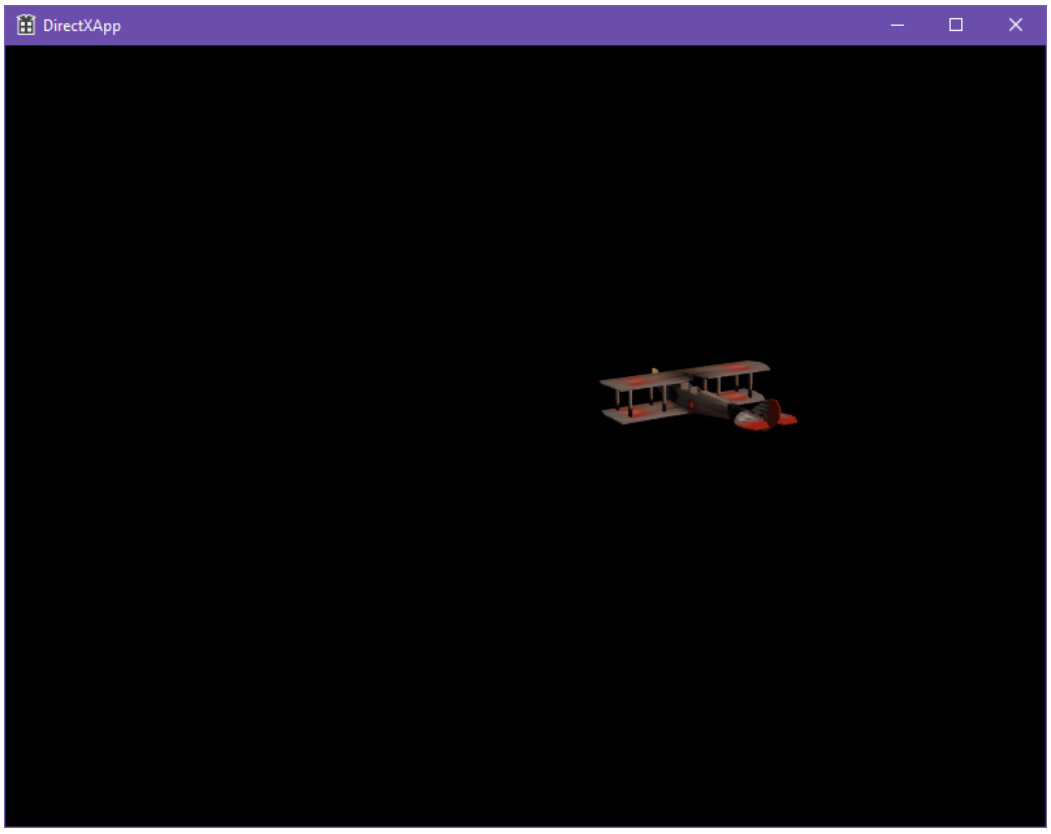
```
        // Update our model's world transformations
        float rotationAngle = -0.75f;
        //Compute the composite Transformation for Plane
        Matrix worldTransformation, ModelYRot, ModelZRot;
        SceneNodePointer Model;

        ModelYRot = Matrix::CreateRotationY(rotationAngle * XM_PI / 180.0f);
        Model = sceneGraph->Find(L"Plane");
        worldTransformation = Model->GetWorldTransform();
        worldTransformation = worldTransformation * ModelYRot;

        Model->SetWorldTransform(worldTransformation);

        ModelZRot = Matrix::CreateTranslation(-0.0341f, -0.4202f,
0.0f)*Matrix::CreateRotationZ(30.0f * rotationAngle * XM_PI /
180.0f)*Matrix::CreateTranslation(+0.0341f, +0.4202f, 0.0f);
        //ModelZRot = Matrix::CreateRotationZ(0.0f * rotationAngle * XM_PI / 180.0f);
        Model = sceneGraph->Find(L"Propeller");
        worldTransformation = Model->GetWorldTransform();
        worldTransformation = ModelZRot * worldTransformation * ModelYRot;

        Model->SetWorldTransform(worldTransformation);

}
```

The output of the executable

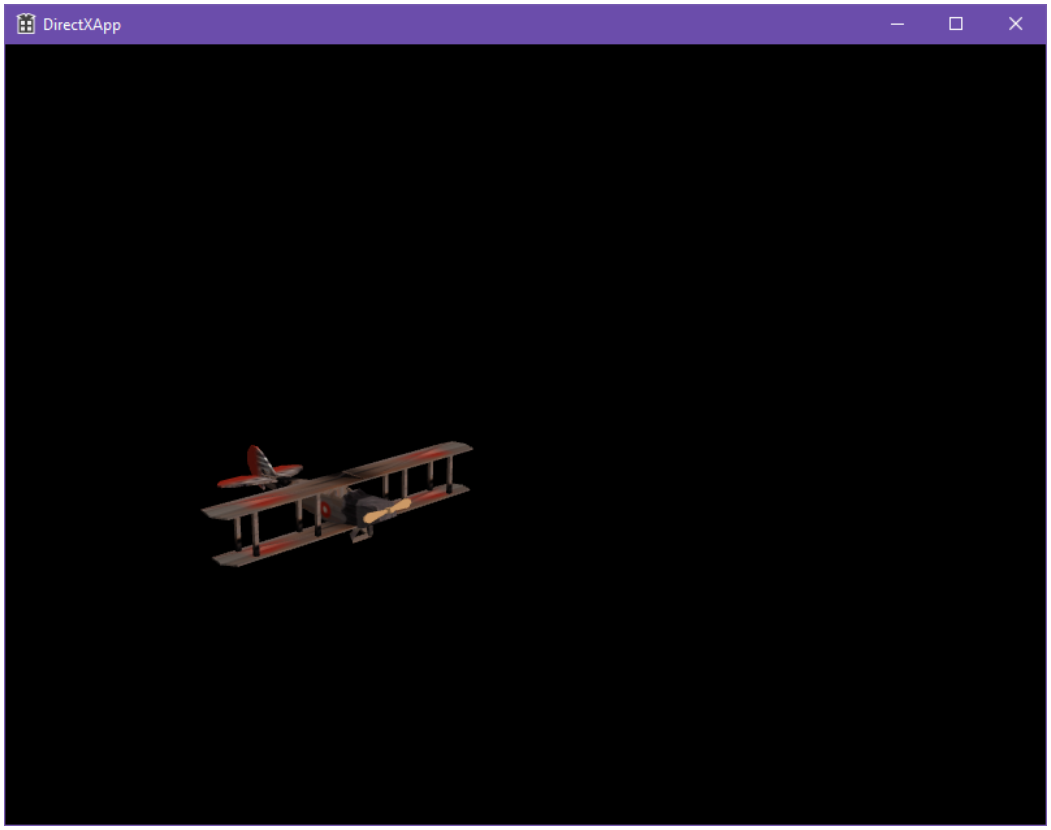
After compiling and running the executable, the application window should look like this:

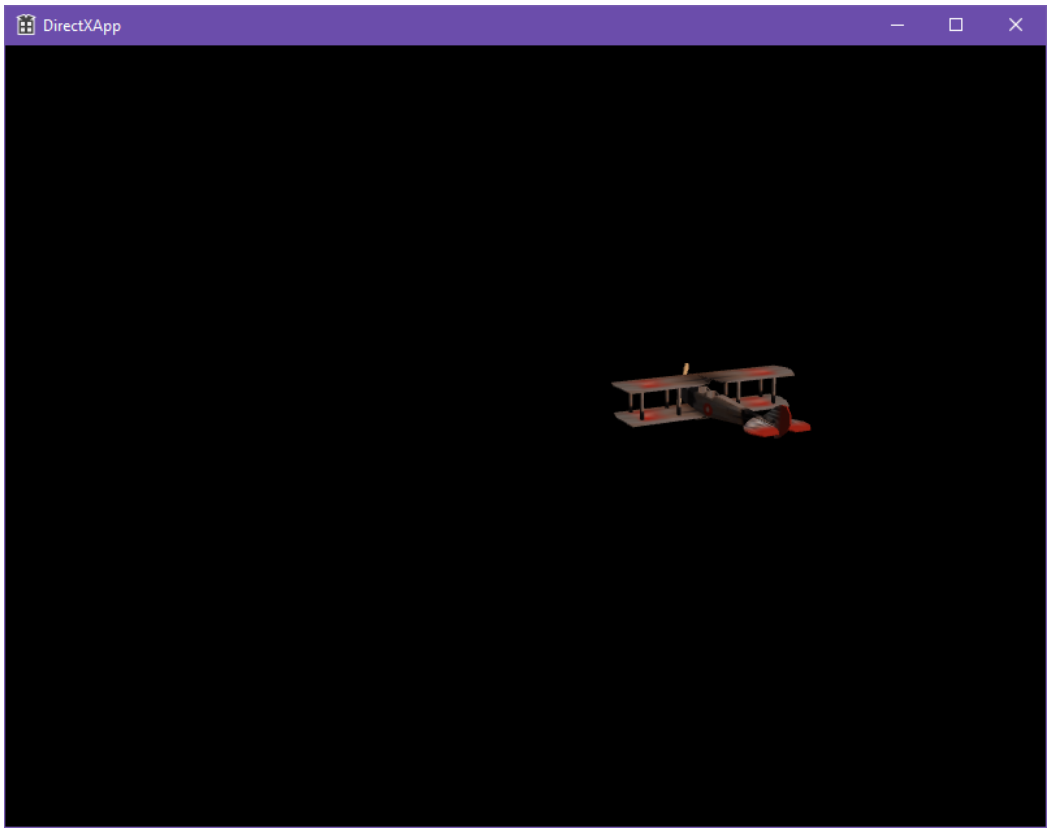
A plane model with texture rotating around the origin





A plane model with texture and rotating propeller rotating around the origin





Exercise

Can you load and add to the scene another plane from the provided model `Bonanza.3ds`?

Can you make it rotate at a higher level in the opposite direction?