# Graphics – Week 5

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). This depends on you having got the work from all previous weeks done so make sure you finish that before starting this.

I have provided the built version of ASSIMP for you as well as classes described in the lecture to you. Note that you should think of these as a starting point. 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. 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.

These files are provided in Week5_Files.zip. The first thing about this 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 (Course Resources), 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. |
| WICTextureLoader.h and .cpp | Classes provided by the DirectX Toolkit to load images and create textures. |

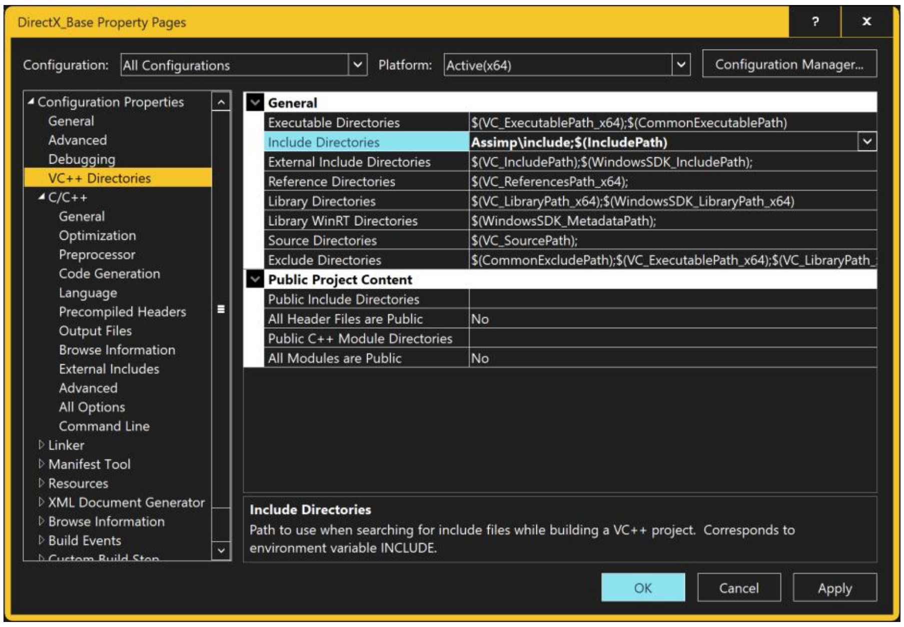
You should copy them to the folder containing your code and add the ResourceManager and Mesh files to your project.

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 Assmp folder in to the same folder as your source files, you should just add the following to the Include Directories:

   Assimp\Include

   You can see this below:


DirectX_Base Property Pages

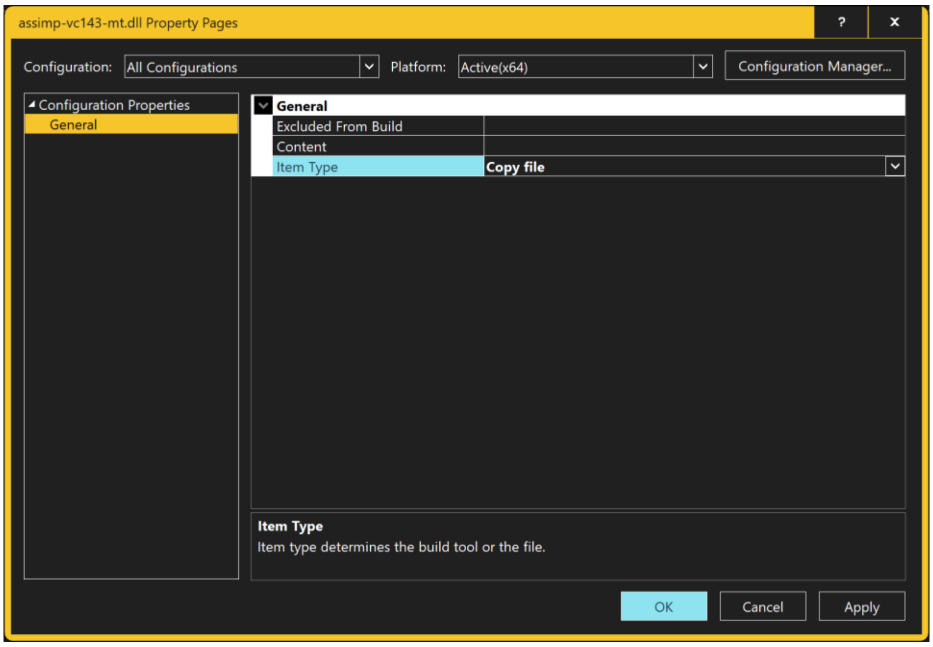
Configuration: All Configurations    Platform: Active(x64)    Configuration Manager...

- Configuration Properties
  - General
  - Advanced
  - Debugging
  - VC++ Directories
  - C/C++
    - General
    - Optimization
    - Preprocessor
    - Code Generation
    - Language
    - Precompiled Headers
    - Output Files
    - Browse Information
    - External Includes
    - Advanced
    - All Options
    - Command Line
  - Linker
  - Manifest Tool
  - Resources
  - XML Document Generator
  - Browse Information
  - Build Events
  - Custom Build Step

| General | |
|---|---|
| Executable Directories | $(VC_ExecutablePath_x64);$(CommonExecutablePath) |
| Include Directories | **Assimp\include;$(IncludePath)** |
| External Include Directories | $(VC_IncludePath);$(WindowsSDK_IncludePath); |
| Reference Directories | $(VC_ReferencesPath_x64); |
| Library Directories | $(VC_LibraryPath_x64);$(WindowsSDK_LibraryPath_x64) |
| Library WinRT Directories | $(WindowsSDK_MetadataPath); |
| Source Directories | $(VC_SourcePath); |
| Exclude Directories | $(CommonExcludePath);$(VC_ExecutablePath_x64);$(VC_LibraryPath_ |
| **Public Project Content** | |
| Public Include Directories | |
| All Header Files are Public | No |
| Public C++ Module Directories | |
| All Modules are Public | No |

**Include Directories**
Path to use when searching for include files while building a VC++ project. Corresponds to environment variable INCLUDE.

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:


assimp-vc143-mt.dll Property Pages

Configuration: All Configurations    Platform: Active(x64)    Configuration Manager...

- Configuration Properties
  - General

| General | |
|---|---|
| Excluded From Build | |
| Content | |
| Item Type | **Copy file** |

**Item Type**
Item type determines the build tool or the file.

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 bulid 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. 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;
```

and then create a new instance of the resource manager after the initialisation of the DirectX framework is complete, but before CreateSceneGraph is called using:

```
_resourceManager = make_shared<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.

Alternatively, you can declare in in your main application file and create it in CreateSceneGraph as above. Then you would need to pass the pointer to the resource manager into the constructor for any node class that needs to access it.

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

```
_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);
```

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 matrices, lighting, 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 as we have been doing all of our calculations in the vertex shader, but you 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 you need to add a line to your rendering code in your node classes to make the constant buffer visible to your pixel shader. The line to add is:

       ```
       _deviceContext->PSSetConstantBuffers( ,  ,
                                 _constantBuffer.GetAddressOf());
       ```

Put this line just after your call to VSSetConstantBuffers. Note that if, after you have moved all of your colour calculation to the pixel shader, your window shows as all black, it will be almost certain that you have missed this line out (I know because I forgot it initially when writing the examples for this week and it took me a while to figure out why my window was all black!).

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 coordinatess, select the pixel shader that does not use a texture.

f. Now draw the sub-mesh using DrawIndexed.

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.

## Testing The Code

I have provided the bi-plane model shown in the lecture on Course Resources for you to try (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.

## Making the Propeller Rotate

At the moment, the propeller on the bi-plane does not rotate and it would be nice if it could. I 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. 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.