

# Model Loading

The previous chapter was all about lighting up our game world. There, we discussed all the different types of light effects and learned how to include various types of light sources in the game world. In this chapter, we'll look at loading a model into our game world. We'll also learn how to set up Assimp, which is the model-loading library that we're going to use to load models into our game world. Assimp is an industry-standard library. It supports a wide array of models, from Wavefront OBJ to Collada. You'll also learn about how to create the `Model` class for the Assimp model loader; we can use that class to load our models into our game world.

In this chapter, we'll cover the following topics:

- Setting up the Assimp library
- Creating the `Mesh` class and `Model` class
- How to load models in the game world using the `Model` class

So, let's get started... We'll begin by learning about how to set up the Assimp library.

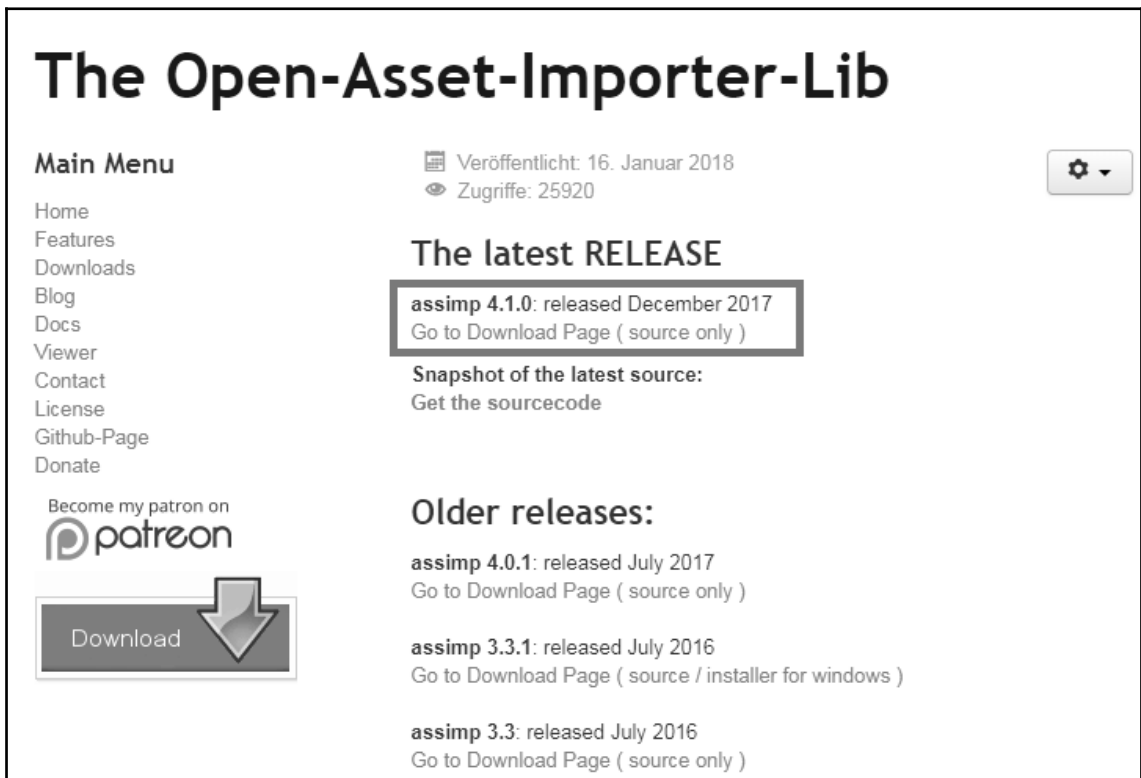
## Setting up the Assimp library

Here we are going to cover how to set up Assimp, which is a model-loading library, on the Windows and macOS platforms using Visual Studio and Xcode. In this section, we'll learn how to download the Assimp library and link it to our project to load models into our game. Let's get started by downloading the Assimp library.

## Downloading the library

Let's take a look at the following steps:

1. Download the Assimp library at [assimp.org](http://assimp.org).
2. Go to the **Downloads** option. In here, you will get to see various versions of libraries; unless you need a specific version, such as an older version, it's best you download the latest version:



The screenshot shows the website for "The Open-Asset-Importer-Lib". The main heading is "The Open-Asset-Importer-Lib". On the left is a "Main Menu" with links: Home, Features, Downloads, Blog, Docs, Viewer, Contact, License, Github-Page, and Donate. Below the menu is a "Become my patron on patreon" button and a "Download" button with a downward arrow. On the right, it says "Veröffentlicht: 16. Januar 2018" and "Zugriffe: 25920". The "The latest RELEASE" section highlights "assimp 4.1.0: released December 2017" with a "Go to Download Page ( source only )" link. Below that is a "Snapshot of the latest source: Get the sourcecode" link. The "Older releases:" section lists "assimp 4.0.1: released July 2017", "assimp 3.3.1: released July 2016", and "assimp 3.3: released July 2016", each with a "Go to Download Page" link.



Sometimes the version will be available on GitHub or SourceForge, which is fine. Whichever it is, just download the ZIP file that is available. It's always recommended to download the `.zip` file.

- 
3. Download **CMake**. CMake allows us to actually generate a Visual Studio solution that will generate the library and DLL files we need to link into our Visual Studio OpenGL project.
  4. Go to <https://cmake.org/>, then to **Download** options, and scroll down to the latest release:

Binary distributions:

Platform	Files
Windows win64-x64 Installer: <b>Installer tool has changed. Uninstall CMake 3.4 or lower first!</b>	cmake-3.12.1-win64-x64.msi
Windows win64-x64 ZIP	<b>cmake-3.12.1-win64-x64.zip</b>
Windows win32-x86 Installer: <b>Installer tool has changed. Uninstall CMake 3.4 or lower first!</b>	cmake-3.12.1-win32-x86.msi
Windows win32-x86 ZIP	<b>cmake-3.12.1-win32-x86.zip</b>

5. Download either the 64-bit or 32-bit, as per your system requirements.



If you are on a 64-bit machine, you can download 64 or 32. If you're on a 32-bit machine, you can only download the 32-bit version. Don't select the installer; just select the `.zip` extension. For this book, we have used the 64-bit version.

You can download the 32-or 64-bit version; it doesn't matter as long as you have the right version.

6. Once you've downloaded the files, extract both folders.



For Mac users, you can download the macOS version of the CMake library.

## Setting up Assimp for the Windows platform

In this section, we'll take a look at how to set up Assimp on the Windows platform.

### Adding libraries to the project folder

Check out the following steps to link the libraries to our project:

1. After extracting the folders, open up the `ASSIMP` folder. In that folder, create a folder and name it `build`.
2. Open up the Command Prompt and type in `cd`, which is the command to change the directory. Press *spacebar* and then drag over the `build` folder that you created in the `ASSIMP` folder. You will be able to notice the following:

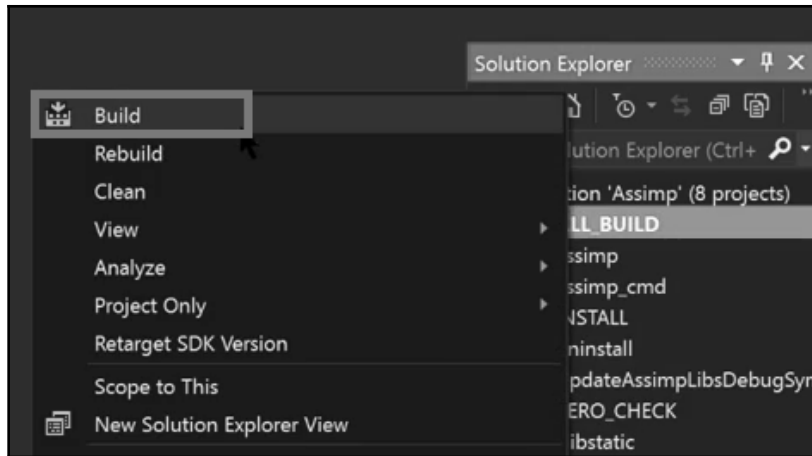
```
C:\Users\FrahaanAmanAziz>cd C:\Users\FrahaanAmanAziz\Desktop\ASSIMP
\build_
```

3. As you can see, the `build` folder is empty, but this is where our project will get generated for Assimp. Press *Enter* and you will get the following:

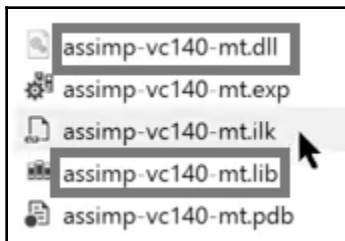
```
C:\Users\FrahaanAmanAziz>cd C:\Users\FrahaanAmanAziz\Desktop\ASSIMP
\build

C:\Users\FrahaanAmanAziz\Desktop\ASSIMP\assimp-3.3.1\build>
```

4. Go to the `cmake` folder and then to the `bin` folder. Then drag and drop `cmake.exe` to the Command Prompt, press *spacebar*, and then go back to the `ASSIMP` folder.
5. Within that folder, there is another `assimp` folder, which consists of various files and folders. Drag and drop this particular `assimp` folder into the Command Prompt and hit *Enter*. This will help build a Visual Studio project for us within the `build` folder.
6. Once the project has been successfully built, go to the `build` folder and then the `Assimp.sln` file.
7. Open up the `Assimp.sln` file in Visual Studio, right-click the **ALL BUILD** option in the **Solution Explorer** window, and then just click on the **Build** option:



8. This will just take a moment or so. Wait for it to build your project successfully and then you'll be given a `.dll` and `.lib` file; we'll be hooking those up to our Visual Studio OpenGL project.
9. Once the project has been built successfully and the files have also been generated, you can close Visual Studio.
10. Go to the `build` folder, then go to `code`, and then go to the `Debug` folder. Within the `Debug` folder, you'll get the `.dll` and `.lib` files:

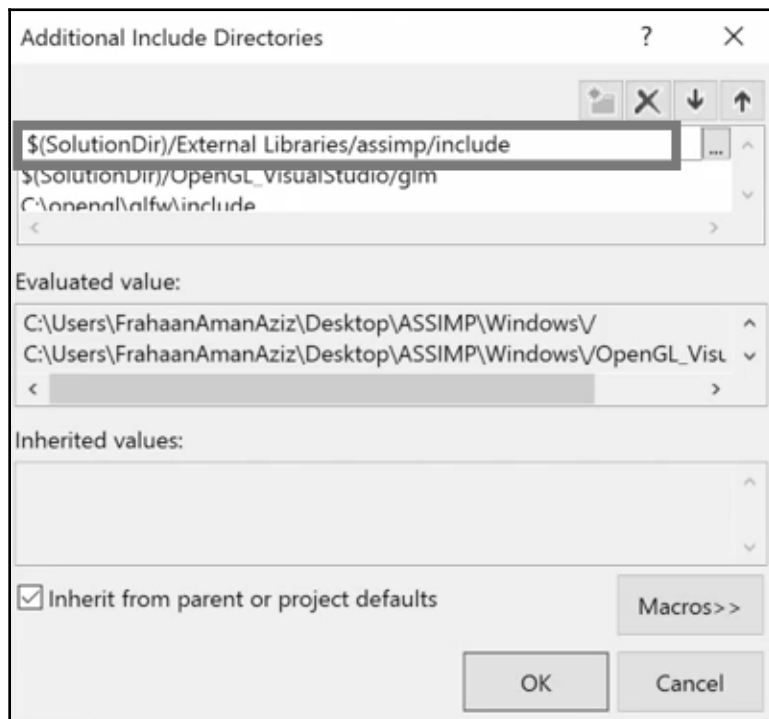


11. Copy all the files present in the `Debug` folder and then go to your Visual Studio OpenGL project, which is present in `Windows` folder. Create an `External Libraries` folder in it and go to the `Windows` folder. Within `External Libraries`, create a folder called `assimp`. Within the `assimp` folder, create a folder called `lib` and paste in all the files that we have copied.
12. Go to the `assimp` folder that we extracted. Within that folder, copy the `include` folder and paste it within `External Libraries\assimp`.

## Linking libraries to the Visual Studio project

Once you have added the libraries to the project folder, it's time to link these libraries to our project in Visual Studio, perform the following steps to do that:

1. Open the Visual Studio project. Right-click on the name of your project in the **Solution Explorer** window.
2. Go to **Properties | C/C++ | General**; then go to the **Additional Include Directories** option, click on the dropdown, then click on **<Edit>**.
3. Click new icon and type in `$(SolutionDir)/ExternalLibraries/assimp/include`:

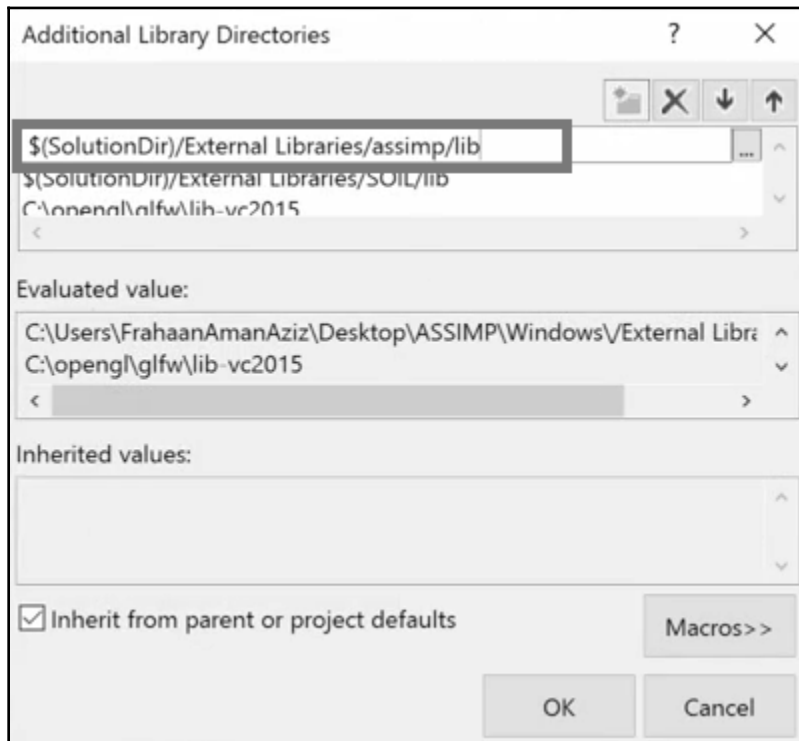


In the preceding screenshot, `SolutionDir` refers to this `.sln` file.

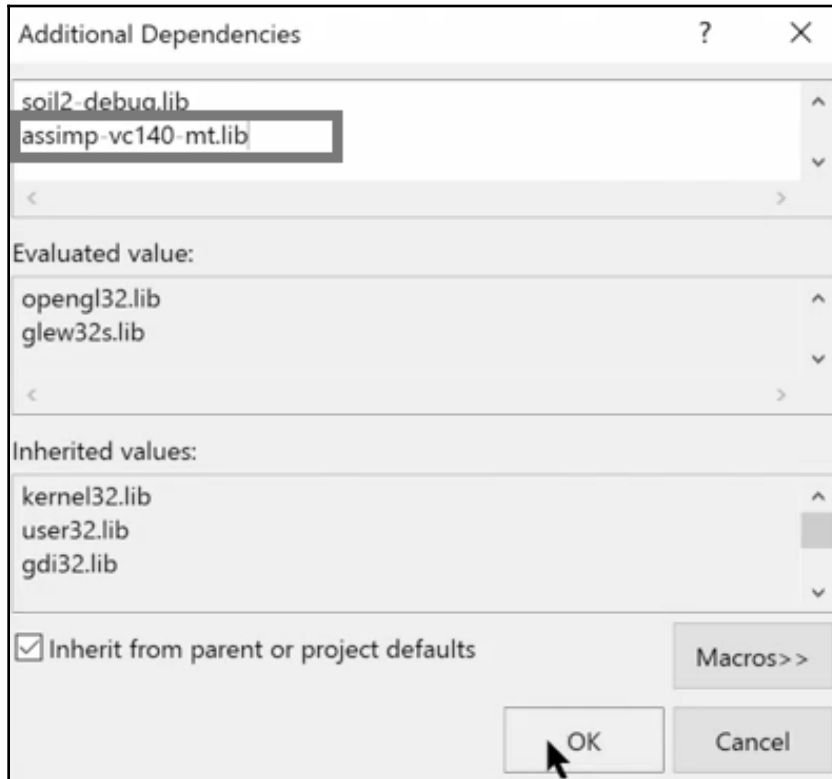


In the preceding steps, we linked the libraries with the help of relative linking. If you are comfortable with absolute linking, you can use that.

4. To save some time, copy the path mentioned in the previous step, go to **Properties** | **Linker** | **General**, then click on **Additional Library Directories**, click on the dropdown, and then click on **<Edit>**.
5. Click new icon, paste the path into it, and make the following change to it:



6. Go to **Linker | Input** and click **Additional Dependencies**. Click the dropdown and then **<Edit>**. We need to specify our library file:



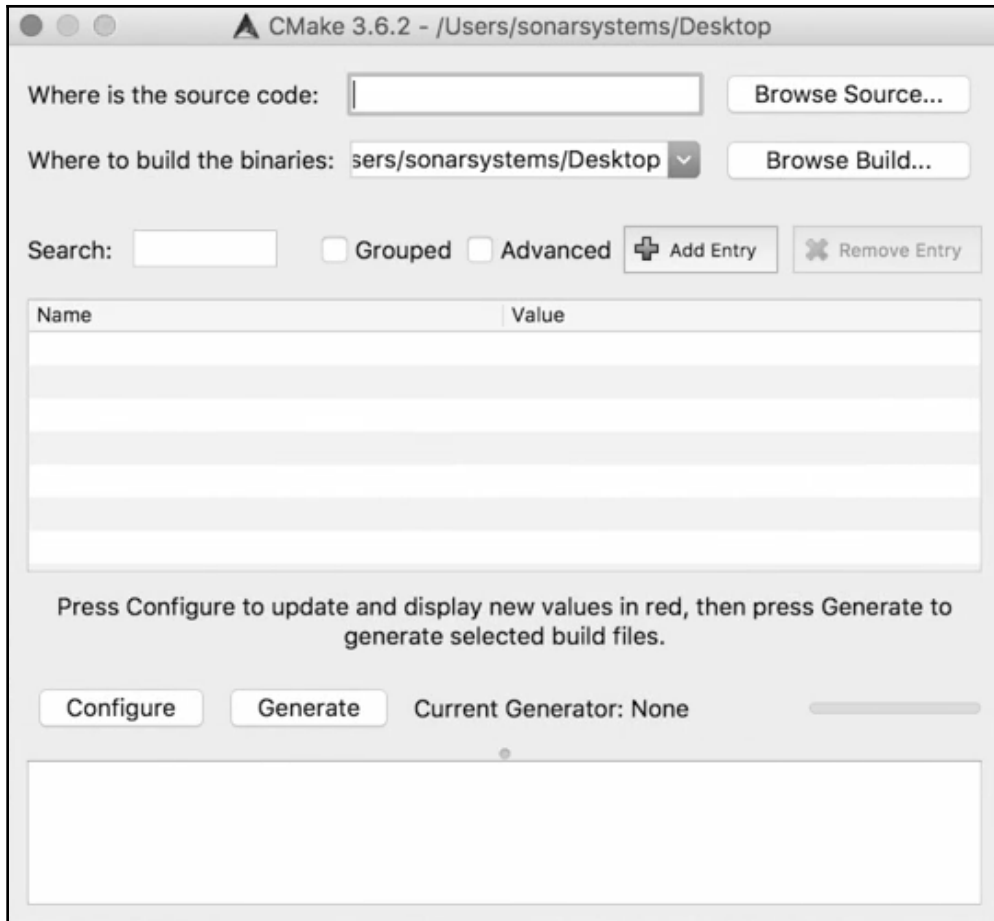
7. Click **Apply** and Click **OK**.

## Setting up Assimp for the Mac platform

Let's take a look at how to set up Assimp for an Xcode project on the Mac platform. Perform the following steps:

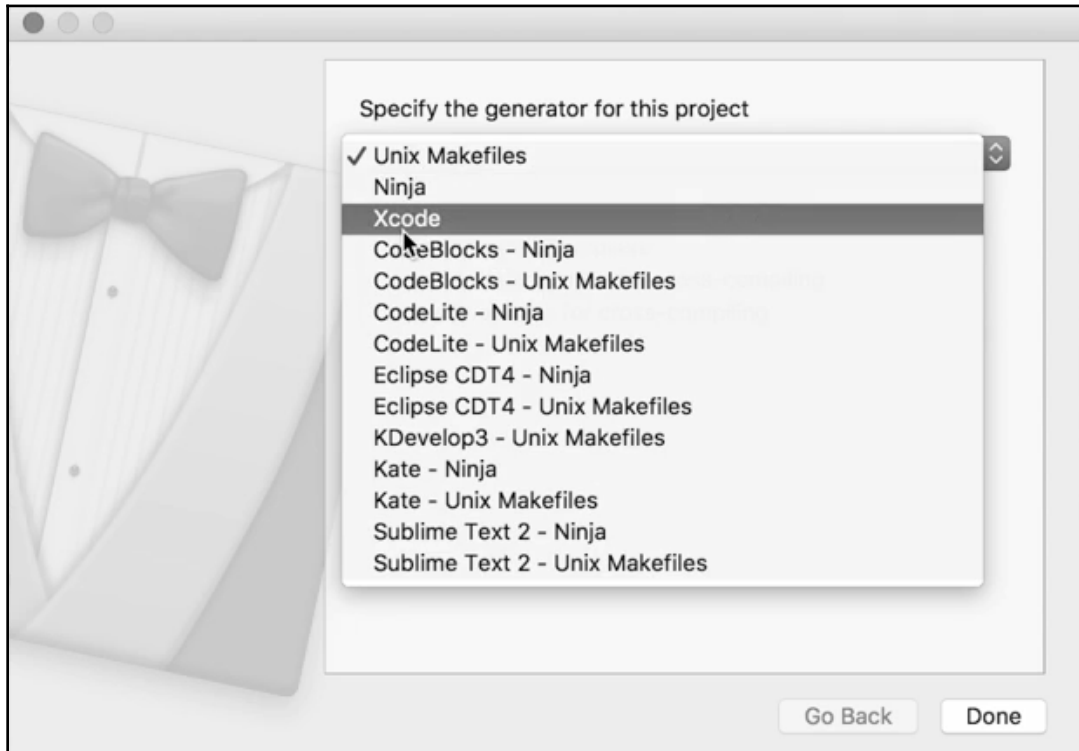
1. Download the Assimp and CMake files. We saw how to download these files in the preceding section. Download and extract both files onto your system.
2. Open up CMake. Once it's open, you will get a window similar to this:





3. Click on the **Browse Source...** button. Go to the location where you have downloaded and extracted the Assimp files and then press on the **Open** button.
4. Click on the **Browse Build...** button. Here you will have to provide the location where you want to build your libraries.

5. Click the **Configure** button and a window will pop up, similar to the following screenshot:



Here you will have to select your developing environment. For us it is Xcode, so we will select that.

---

## Linking up the libraries to the project

In this section, we'll take a look at how to link up the libraries to our project on the Mac platform. To include the `DYLIB` in our existing project, we need to perform the following steps:

1. Go to main project folder on the Xcode, then go to **Build Phases**, and then to the **Link Binary** in the **Libraries** section by clicking on the + sign.
2. Click on the **Add Other** button.
3. Press `Cmd + Shift + G`, type in `/usr/local/cellar`, and click the **Go** button.
4. Go to the `Assimp` folder, choose the version folder that you've got.
5. Go to the `lib` folder and select `libassimp.3.3.1.dylib`, or whatever version you have got, but remember to select the `.dylib` file only.
6. Click on the **Open** button.

And that's how we include the `DYLIB` file in our existing project.

With this last step, we are done with linking libraries to our project. Now we'll move on to creating the `Mesh` class.

## The Mesh class

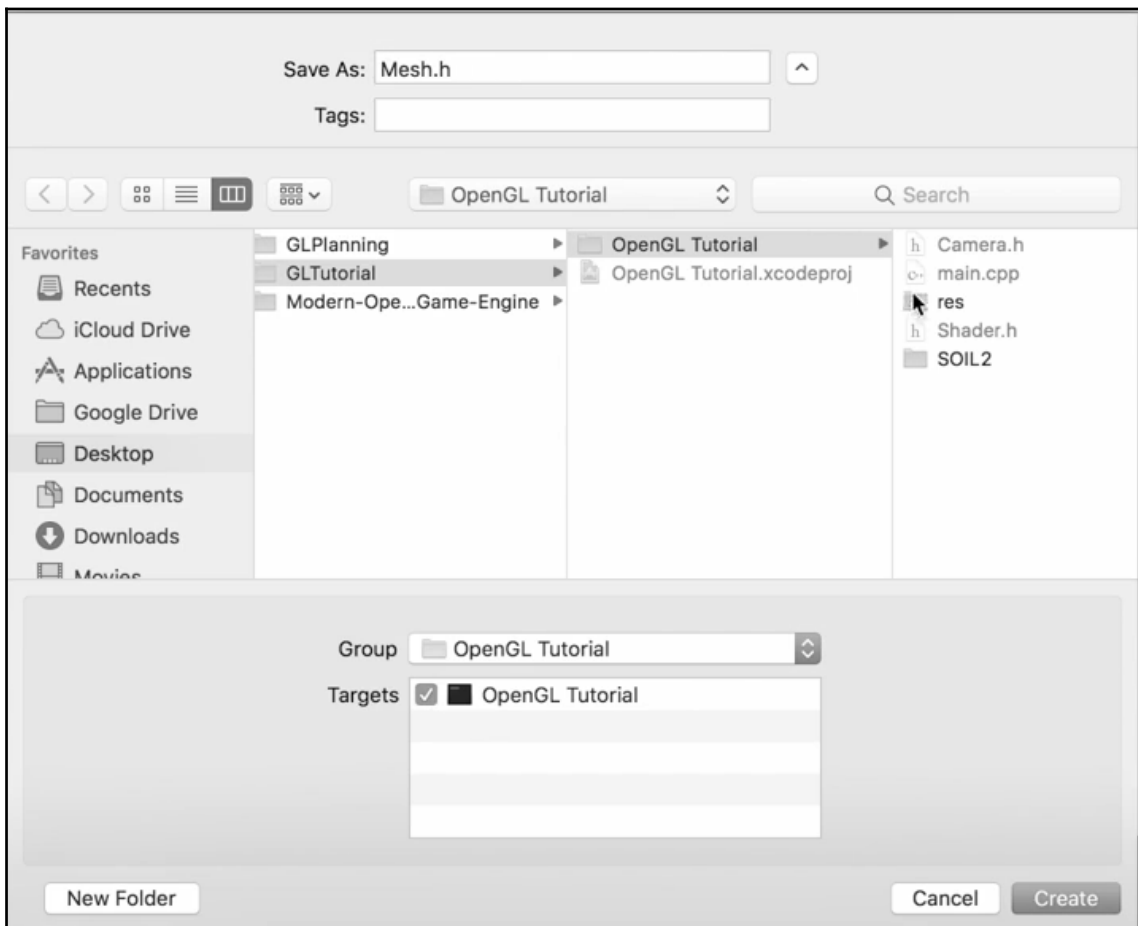
Now we are ready to create the `Mesh` class. The `Mesh` class is essentially a simple container that stores data, such as texture coordinates, vertices, and normals. We had studied this in the preceding section. Here, we are going to create a model object that will essentially be a container for several mesh objects.

So, let's begin with creating the `Mesh` class.

## Creating the Mesh.h header file

Let's take a look at the following steps to create `Mesh.h` header file:

1. On Xcode, head over to your OpenGL project, right-click on it, and select **New File....**
2. Select the header file and then click on the **Next** button.
3. Name the file `Mesh.h` and select **OpenGL** on **Targets**:



4. Click on the **Create** button and your empty header file will be created.

---

## Coding the Mesh.h header file

In this section, we'll be coding our `Mesh.h` header files.

### Adding necessary header files

Follow these steps to add header files to your code:

1. Add some header files to your code:

```
#pragma once

#include <string>
#include <fstream>
#include <sstream>
#include <iostream>
#include <vector>

#include <GL/glew.h>
#include <glm/glm.hpp>
#include <glm/gtc/matrix_transform.hpp>
```

2. Add using namespace `std` to the code.

## Creating Vertex and Texture Struct

Take a look at the steps to create `Struct`:

1. We'll be creating the `Struct` datatype, called `Vertex`, and we'll add the following lines of code to it:

```
struct Vertex
{
    // Position
    glm::vec3 Position;
    // Normal
    glm::vec3 Normal;
    // TexCoords
    glm::vec2 TexCoords;
};
```

2. Create another Struct, called `Texture`, and add the following lines of code to it:

```
struct Texture
{
    GLuint id;
    string type;
    aiString path;
};
```

## Creating the Mesh class

Now let's create the `Mesh` class. Follow these steps:

1. In the class `Mesh`, create a bunch of public variables:

```
class Mesh
{
public:
    /* Mesh Data */
    vector<Vertex> vertices;
    vector<GLuint> indices;
    vector<Texture> textures;
```

2. Create the constructor, called `Mesh ( )`, and to this add the following lines of code:

```
// Constructor
Mesh( vector<Vertex> vertices, vector<GLuint> indices,
vector<Texture> textures )
{
    this->vertices = vertices;
    this->indices = indices;
    this->textures = textures;

    // Now that we have all the required data, set the vertex
    buffers //and its attribute pointers.
    this->setupMesh( );
}
```

In the preceding lines of code, with `this->`, we're just passing in the data for the vertices, the indices, and the textures, and assigning it to the mesh's variables so it knows what data it needs. Next, we added `this->setupMesh( )`; to set the vertex buffers and the attribute pointers.

---

## Defining the Draw method

Now let's define the `Draw` method:

1. Create the `Draw` method, which will be public. We'll add the following lines of code to it:

```
// Render the mesh
void Draw( Shader shader )
{
    // Bind appropriate textures
    GLuint diffuseNr = 1;
    GLuint specularNr = 1;

    for( GLuint i = 0; i < this->textures.size( ); i++ )
    {
        glActiveTexture( GL_TEXTURE0 + i ); // Active proper
texture          unit before binding
        // Retrieve texture number (the N in diffuse_textureN)
        stringstream ss;
        string number;
        string name = this->textures[i].type;

        if( name == "texture_diffuse" )
        {
            ss << diffuseNr++; // Transfer GLuint to stream
        }
        else if( name == "texture_specular" )
        {
            ss << specularNr++; // Transfer GLuint to stream
        }

        number = ss.str( );
        // Now set the sampler to the correct texture unit
        glUniform1i( glGetUniformLocation( shader.Program, ( name
+          number ).c_str( ), i );
        // And finally bind the texture
        glBindTexture( GL_TEXTURE_2D, this->textures[i].id );
    }
}
```

2. We've done the `for` loop. We've still got some code left to add to the `Draw` class. We need to set each mesh's shininess property to a default value. Default values are always good to prevent variables from messing up. So, we'll add the following lines of code:

```
// Also set each mesh's shininess property to a default value (if
you //want you could extend this to another mesh property and
```

```
possibly //change this value)
glUniform1f( glGetUniformLocation( shader.Program,
"material.shininess" ), 16.0f );
```

3. Add the code to draw the mesh:

```
// Draw mesh
glBindVertexArray( this->VAO );
glDrawElements( GL_TRIANGLES, this->indices.size( ),
GL_UNSIGNED_INT, 0 );
glBindVertexArray( 0 );
```

4. Set everything back to its default values once you're finished the configuration, (It's just a good practice):

```
for ( GLuint i = 0; i < this->textures.size( ); i++ )
{
    glActiveTexture( GL_TEXTURE0 + i );
    glBindTexture( GL_TEXTURE_2D, 0 );
}
```

## Defining the private section of the Mesh class

We're now done with the `Draw` method, and that's everything in the public section of the `Mesh` class. Next, we're going to define private section of the `Mesh` class:

1. Add a few variables:

```
/* Render data */
GLuint VAO, VBO, EBO;
```

We've looked at what all of these variables do in the initial chapters. If you're not too sure what these variables do, feel free to go back for a refresher.

2. To initialize all of the buffer objects and arrays, begin by typing `void setupMesh( )`. We called `setupMesh` in the constructor. This isn't something that the developer really would be calling manually. Once whatever is using the `Mesh` class as an object has constructed it, it will then call the `SetupMesh` method.
3. In `setupMesh( )`, we need to create the buffers and arrays, so we'll add the following lines of code:

```
void setupMesh( )
{
    // Create buffers/arrays
    glGenVertexArrays( 1, &this->VAO );
```



---

```
glGenBuffers( 1, &this->VBO );
glGenBuffers( 1, &this->EBO );
glBindVertexArray( this->VAO );
```

4. Load the data into the vertex buffer, so we're going to add the following lines of code:

```
glBindBuffer( GL_ARRAY_BUFFER, this->VBO );

// A great thing about structs is that their memory layout is
// sequential for all its items.
// The effect is that we can simply pass a pointer to the struct
// and it translates perfectly to a glm::vec3/2 array which
// again translates to 3/2 floats which translates to a byte
// array.

glBufferData( GL_ARRAY_BUFFER, this->vertices.size() * sizeof(
Vertex ), &this->vertices[0], GL_STATIC_DRAW );
```

5. Duplicate the preceding lines of code and make the following highlighted changes:

```
glBindBuffer( GL_ELEMENT_ARRAY_BUFFER, this->EBO );
glBufferData( GL_ELEMENT_ARRAY_BUFFER, this->indices.size() *
sizeof( GLuint ), &this->indices[0], GL_STATIC_DRAW );
```

6. We can move on to setting the vertex attribute pointers. The first one we are going to set up is the vertex positions:

```
// Vertex Positions
glEnableVertexAttribArray( 0 );
glVertexAttribPointer( 0, 3, GL_FLOAT, GL_FALSE, sizeof( Vertex ),
( GLvoid * ) 0 );
```

7. Set up the vertex normals:

```
// Vertex Normals
glEnableVertexAttribArray( 1 );
glVertexAttribPointer( 1, 3, GL_FLOAT, GL_FALSE, sizeof( Vertex ),
( GLvoid * ) offsetof( Vertex, Normal ) );
```

8. Set up texture co-ordinates:

```
// Vertex Texture Coords
glEnableVertexAttribArray( 2 );
glVertexAttribPointer( 2, 2, GL_FLOAT, GL_FALSE, sizeof( Vertex ),
( GLvoid * ) offsetof( Vertex, TexCoords ) );
```

9. Unbind the vertex array:

```
        glBindVertexArray( 0 );
    }
};
```

So, this is it for creating the `mesh` class. In the next section, we look at creating a `Model` class that will contain a number of these `mesh` classes.

## Getting rid of some old code

As we will be working on the source code from Chapter 5, *Types of Light Sources and Combining of Lights*, there are some lines in the code that we need to get rid of from our main code. In this section, we won't be using our old lighting code or the cube code that we had discussed in the initial chapters. So, go to your `main.cpp` file and get rid of the build and compile-shader code, vertices-defining code, the code that defines `cubePositions`, all the attributes-defining code, and the `SOIL` code. Also get rid of the `lightingShader.Use()` code.

In the `while` loop, firstly, we need to get rid of the `lightPos.x` and `lightPos.z` code, and then we will get rid of all the lighting-related code until we define the `SwapBuffer` code. We'll also get rid of `glDelete` as we no longer will be deleting our arrays and buffers.

Now that we have got rid of all the unwanted code, we'll get started with the our model-loading code.

## Modelloading

In the previous sections, we went through the setup process of Assimp, and we also created the `mesh` class. Now, in this section, we are going to be looking at creating the `Model` class for the Assimp model loader, to start loading in models. The `Model` class basically consists of many meshes.

We will now get started with creating our `Model` class. But before that, some information about the code files for this chapter would be useful; you can find the code files at the following link: <https://GitHub.com/PacktPublishing/Learn-OpenGL/tree/master/Chapter07>.

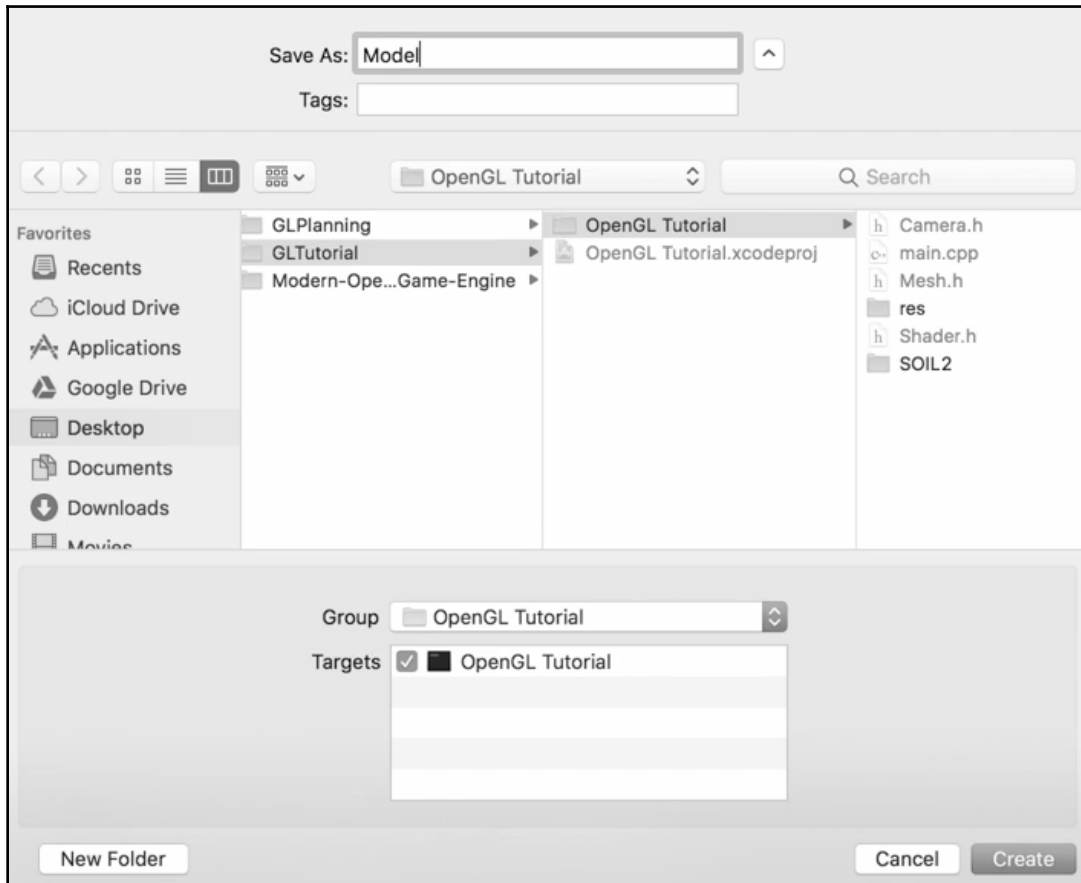
---

The folder placed on this link consists of the `res` folder, in which there is a folder called `models`. This `models` folder consists of a lot of files related to the model that we'll be loading in our code.

## Creating the Model.h header file

So, let's begin with creating our `Model.h` header file:

1. Right-click on your `project` folder.
2. Click on **New File** and click on the **Header** file option, because we will put the class within the header and we need to make sure that we add the header files to our target folder. We're going to name the header file `Model.h` and then click on the **Create** button:



## Coding the Model.h header file

Now we'll begin coding our `Model.h` file, take a look at the following steps:

1. Add the basic things, such as strings and streams:

```
#pragma once

#include <string>
#include <fstream>
#include <sstream>
#include <iostream>
#include <map>
#include <vector>
```

2. Include OpenGL, SOIL (which is a local library), the Assimp library, Mesh files, and other necessary files:

```
#include <GL/glew.h>
#include <glm/glm.hpp>
#include <glm/gtc/matrix_transform.hpp>
#include "SOIL2/SOIL2.h"
#include <assimp/Importer.hpp>
#include <assimp/scene.h>
#include <assimp/postprocess.h>

#include "Mesh.h"
```



OpenGL will assume we are using GLEW to load an extension, and not something such as `glLoadGen`. But if we are using `glLoadGen`, we need to replace it.

3. Add using namespace `std`, and then we will also add the `GLint TextureFromFile` function:

```
using namespace std;

GLint TextureFromFile( const char *path, string directory );
```

4. Create the `Model` class, wherein we will define our `public` section first with the `Model` constructor and then use the `Draw` method to draw the model:

```
class Model
{
public:
    /* Functions */
```

---

```

// Constructor, expects a filepath to a 3D model.
Model( GLchar *path )
{
    this->loadModel( path );
}

// Draws the model, and thus all its meshes
void Draw( Shader shader )
{
    for ( GLuint i = 0; i < this->meshes.size(); i++ )
    {
        this->meshes[i].Draw( shader );
    }
}

```

In the preceding code, we used the shader file. Then we iterated the `for` loop over the meshes and drew the meshes.

## Defining the private section of the Model class

Next, we'll implement the arguments in `private`. Perform the following steps:

1. Create some variables in which we can store the Model data:

```

private:
    /* Model Data */
    vector<Mesh> meshes;
    string directory;
    vector<Texture> textures_loaded;
    // Stores all the textures loaded so far, optimization to make
    // sure //textures aren't loaded more than once.

```

2. Create the `loadModel` method that we called in the constructor. In this method, we need to read the file using `Assimp`, so we'll add the following lines of code:

```

// Loads a model with supported ASSIMP extensions from file and
// stores the resulting meshes in the meshes vector.
void loadModel( string path )
{
    // Read file via ASSIMP
    Assimp::Importer importer;
    const aiScene *scene = importer.ReadFile( path,
        aiProcess_Triangulate | aiProcess_FlipUVs );

```

In the preceding code, we used `Assimp`, with the namespace as `Importer`. Then we had a constant `aiScene`, which is basically the `assimp` scene.

3. Check whether there any errors while reading the file via assimp by adding the following lines of code:

```
// Check for errors
if( !scene || scene->mFlags == AI_SCENE_FLAGS_INCOMPLETE || !scene-
>mRootNode ) // if is Not Zero
{
    cout << "ERROR::ASSIMP:: " << importer.GetErrorString( ) <<
endl;
    return;
}
```

4. If there are no errors, we assume that the model has successfully loaded, and now we have to retrieve the directory path of the file path:

```
// Retrieve the directory path of the filepath
this->directory = path.substr( 0, path.find_last_of( '/' ) );
```

5. Process all of Assimp's root nodes recursively:

```
// Process ASSIMP's root node recursively
this->processNode( scene->mRootNode, scene );
}
```

So, that is it for the load model method.

6. Implement the `processNode` method and add the following lines of code to it:

```
void processNode( aiNode* node, const aiScene* scene )
{
```

7. Process each mesh located at the node:

```
// Process each mesh located at the current node
for ( GLuint i = 0; i < node->mNumMeshes; i++ )
{
    // The node object only contains indices to index the actual
objects in the scene.
    // The scene contains all the data, node is just to keep stuff
organized (like relations between nodes).
    aiMesh* mesh = scene->mMeshes[node->mMeshes[i]];

    this->meshes.push_back( this->processMesh( mesh, scene ) );
}
```

In the preceding code, the node object contains indices to index the actual objects in the scene. The scene contains all the data, and the node is just to keep relations between the nodes organized.

- 
8. Recursively process each of the children of the meshes – if there are actually any:

```
// After we've processed all of the meshes (if any) we then
recursively process each of the children nodes
for ( GLuint i = 0; i < node->mNumChildren; i++ )
{
    this->processNode( node->mChildren[i], scene );
}
```

9. And that's all for `processNode`. We will now create the `processMesh` method, which actually returns a mesh. The first thing to do in this method is to create some variables, which will include the vertices, the indices, and the textures: all the basic stuff that we have pretty much done before:

```
Mesh processMesh( aiMesh *mesh, const aiScene *scene )
{
    // Data to fill
    vector<Vertex> vertices;
    vector<GLuint> indices;
    vector<Texture> textures;
```

10. We'll walk through each of the mesh's vertices now, and then we'll want to loop over it:

```
// Walk through each of the mesh's vertices
for ( GLuint i = 0; i < mesh->mNumVertices; i++ )
{
    Vertex vertex;
```

11. In the loop, declare a placeholder vector, since Assimp uses its own `vector` class and doesn't directly convert to it:

```
glm::vec3 vector;
```

12. And for the positions, in the `for` loop, we are going to set the position as follows :

```
// Positions
vector.x = mesh->mVertices[i].x;
vector.y = mesh->mVertices[i].y;
vector.z = mesh->mVertices[i].z;
vertex.Position = vector;
```

13. And, similarly, we need to mention the normals as follows :

```
// Normals
vector.x = mesh->mNormals[i].x;
vector.y = mesh->mNormals[i].y;
vector.z = mesh->mNormals[i].z;
vertex.Normal = vector;
```

14. Check the texture coordinates. To do that, add the following code. Check, with the help of the `if` statement, whether the mesh consists of texture coordinates or not. If not, we'll implement the `else` condition:

```
// Texture Coordinates
if( mesh->mTextureCoords[0] ) // Does the mesh contain texture
coordinates?
{
    glm::vec2 vec;
    vec.x = mesh->mTextureCoords[0][i].x;
    vec.y = mesh->mTextureCoords[0][i].y;
    vertex.TexCoords = vec;
}
else
{
    vertex.TexCoords = glm::vec2( 0.0f, 0.0f );
}
}
```

A vertex can contain up to eight different texture coordinates. We thus make the assumption that we won't use models where a vertex can have multiple texture coordinates; we always take the first set as zero, so we added `mTextureCoords[0]`. In the `else` statement, we just assign the default values of `0.0f` and `0.0f` if the mesh doesn't have any texture coordinates.

15. Push back the vertex that we have calculated:

```
vertices.push_back( vertex );
```

16. Walk through each of the faces in the mesh and retrieve the corresponding vertex indices. Add a `for` loop to our code and initialize it, and then inside the `for` loop, create an Assimp face variable:

```
// Now walk through each of the mesh's faces (a face is a mesh its
triangle) and retrieve the corresponding vertex indices.

for ( GLuint i = 0; i < mesh->mNumFaces; i++ )
{
    aiFace face = mesh->mFaces[i];
```



- 
17. Inside this loop, we'll add another `for` loop to retrieve all indices of the `face` variable and store them in the `indices` vector:

```
for ( GLuint j = 0; j < face.mNumIndices; j++ )
{
    indices.push_back( face.mIndices[j] );
}
```

18. Process the material, because so far we've just been processing the meshes and the vertices. Add an `if` statement to the code to check whether the `mMaterialIndex` mesh is greater than 0:

```
// Process materials
if( mesh->mMaterialIndex >= 0 )
{
    aiMaterial* material = scene->mMaterials[mesh->mMaterialIndex];
}
```

19. Assume a convention for the sampler name in the shaders. Each diffuse texture should be named as `texture_diffuseN`, where `N` is basically the sequential number ranging from 1 to the max sampler number, with the same applying to the other texture. So, basically, diffuse is `texture_diffuseN`, specular is `texture_specularN`, and normal is `texture_normalN`. In order to handle the diffuse map, perform the following:

```
// Diffuse maps
vector<Texture> diffuseMaps = this->loadMaterialTextures( material,
aiTextureType_DIFFUSE, "texture_diffuse" );
textures.insert( textures.end( ), diffuseMaps.begin( ),
diffuseMaps.end( ) );
```

20. To handle the specular maps, perform:

```
// Specular maps
vector<Texture> specularMaps = this->loadMaterialTextures(
material, aiTextureType_SPECULAR, "texture_specular" );
textures.insert( textures.end( ), specularMaps.begin( ),
specularMaps.end( ) );
```

21. So, outside of this `if` statement, we're going to return a mesh object that is extracted from the mesh data:

```
// Return a mesh object created from the extracted mesh data
return Mesh( vertices, indices, textures );
```

22. Load the material textures, which is going to be returning a vector of textures. This will check all material textures of a given type and load the textures if they're not loaded already. The required info is returned as a texture struct:

```
vector<Texture> loadMaterialTextures( aiMaterial *mat,
aiTextureType type, string typeName )
{
vector<Texture> textures;
```

23. Loop over the material that we created here:

```
for ( GLuint i = 0; i < mat->GetTextureCount( type ); i++ )
{
aiString str;
mat->GetTexture( type, i, &str );
```

24. Create a variable named `GLboolean` and set it to `false` to check whether the texture was loaded before. If so, continue to the next iteration – skip loading a new texture, create a `for` loop, and push in the texture that has been loaded:

```
GLboolean skip = false;

for ( GLuint j = 0; j < textures_loaded.size( ); j++ )
{
```

25. Check the `if` condition to see whether the texture has been loaded and returns the textures:

```
if( textures_loaded[j].path == str )
{
textures.push_back( textures_loaded[j] );
skip = true; // A texture with the same filepath has already
been loaded, continue to next one. (optimization)

break;
```

26. After the `for` loop, we are going to check whether the texture has been loaded, and load it if it hasn't been already:

```
if( !skip )
{ // If texture hasn't been loaded already, load it
Texture texture;
texture.id = TextureFromFile( str.C_Str( ), this->directory );
texture.type = typeName;
texture.path = str;
textures.push_back( texture );
```

---

```

        this->textures_loaded.push_back( texture ); // Store it as
texture      loaded for entire model, to ensure we won't unnecessary
load          duplicate textures.
    }

    }

        return textures;
    }
};

```

27. If you remember, right at the top, we created a function declaration but we didn't implement it anywhere in the code. So, we're going to implement that now. Generate a texture ID, and load the texture data appropriately:

```

GLuint TextureFromFile( const char *path, string directory )
{
    //Generate texture ID and load texture data
    string filename = string( path );
    filename = directory + '/' + filename;
    GLuint textureID;
    glGenTextures( 1, &textureID );

    int width, height;

```

28. We need an unsigned character that will store the image data:

```

unsigned char *image = SOIL_load_image( filename.c_str( ), &width,
&height, 0, SOIL_LOAD_RGB );

```

29. Assign a texture to the particular ID. To do that, we need to bind textureID to texture2D:

```

// Assign texture to ID
glBindTexture( GL_TEXTURE_2D, textureID );
glTexImage2D( GL_TEXTURE_2D, 0, GL_RGB, width, height, 0, GL_RGB,
GL_UNSIGNED_BYTE, image );

```

30. Generate the Mipmap for the preceding texture ID:

```

glGenerateMipmap( GL_TEXTURE_2D );

```

### 31. Specify the parameters:

```
// Parameters
glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT );
glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT );
glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
GL_LINEAR_MIPMAP_LINEAR );
glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
glBindTexture( GL_TEXTURE_2D, 0 );
SOIL_free_image_data( image );

return textureID;
```

And now we are done coding our `Model.h` file.

## Creating new shaders

Let's create new shaders for model loading by following below-mentioned steps:

1. Go to the `shaders` folder and create a couple of new shaders. We will duplicate the shaders from what we've already got. We then rename them to `modelLoading.vs` and `modelLoading.frag`.
2. Go to the vertex shader, `modelLoading.vs`, and type the following highlighted terms:

```
#version 330 core
layout ( location = 0 ) in vec3 position;
layout ( location = 1 ) in vec3 normal;
layout ( location = 2 ) in vec2 texCoords;

out vec2 TexCoords;

uniform mat4 model;
uniform mat4 view;
uniform mat4 projection;

void main( )
{
    gl_Position = projection * view * model * vec4( position, 1.0f );
    TexCoords = texCoords;
}
```

- 
3. For the fragment shader, take a look at the following highlighted code and carry out the necessary changes:

```
#version 330 core

in vec2 TexCoords;

out vec4 color;

uniform sampler2D texture_diffuse;

void main( )
{
    color = vec4( texture( texture_diffuse, TexCoords ) );
}
```

We're done with creating new shaders for model loading.

So, now we are ready to actually load our model. So, we will go to `main.cpp` and carry out certain modifications to it.

## Loading the model in the main code

Let's take a look at following steps to understand how to load the model in the main code:

1. Include the new `Model.h` header file in our existing code:

```
#include "Model.h"
```

2. Before our projection matrix code, we need to add code to set up and compile our shaders:

```
// Setup and compile our shaders
Shader shader( "res/shaders/modelLoading.vs",
"res/shaders/modelLoading.frag" );
```

3. Load the model:

```
// Load models
Model ourModel( "res/models/nanosuit.obj" );
```

This `nanosuit.obj` file consists of information about our model that we'll be loading in our game world.

4. In the `while` loop, after we've defined `glClear`, we want to add code to use the shader:

```
shader.Use( );
```

5. Create a `ViewMatrix`:

```
glm::mat4 view = camera.GetViewMatrix( );

glUniformMatrix4fv( glGetUniformLocation( shader.Program,
"projection" ), 1, GL_FALSE, glm::value_ptr( projection ) );

glUniformMatrix4fv( glGetUniformLocation( shader.Program, "view" ),
1, GL_FALSE, glm::value_ptr( view ) );
```

6. Draw the loaded model:

```
// Draw the loaded model
glm::mat4 model;

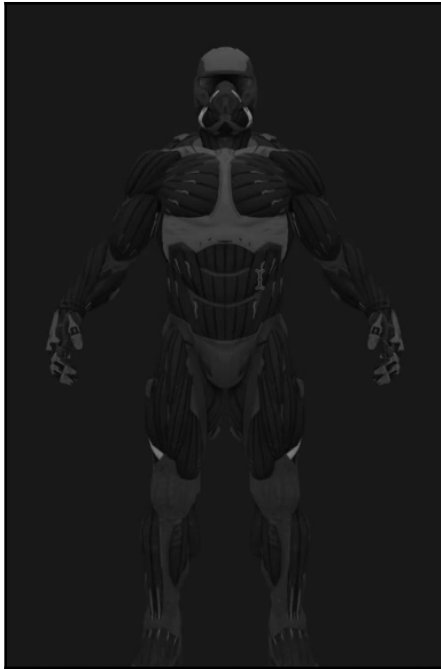
model = glm::translate( model, glm::vec3( 0.0f, -1.75f, 0.0f ) );
// Translate it down a bit so it's at the center of the scene

model = glm::scale( model, glm::vec3( 0.2f, 0.2f, 0.2f ) ); // It's
a bit too big for our scene, so scale it down

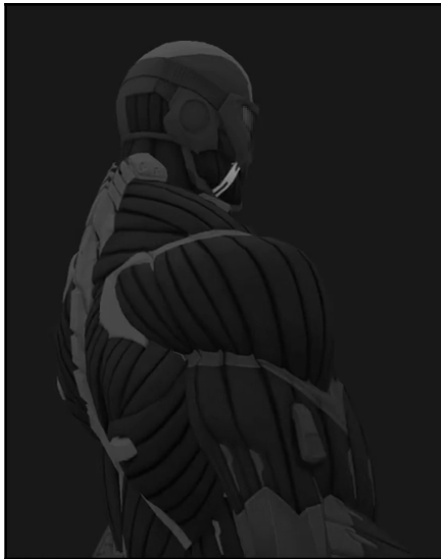
glUniformMatrix4fv( glGetUniformLocation( shader.Program, "model"
), 1,
GL_FALSE, glm::value_ptr( model ) );

ourModel.Draw( shader );
```

And now we are ready to run the code and load the model in our game world. If your code complies without any errors, you will get to see the following output on your screen:



We can move this model around using our camera class:

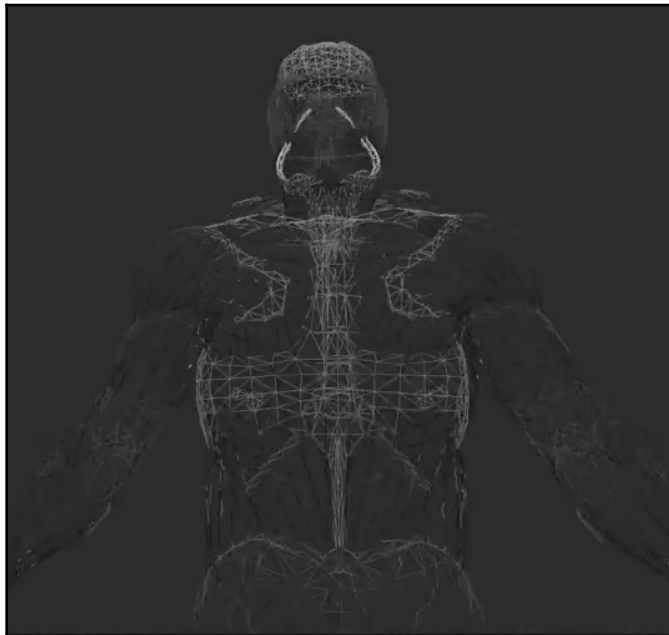


So, that is it for model loading. Now that you've got this code all sorted, you can start loading a bunch of other models. It is recommended going to the Assimp website to see what models they support. There's just a whole heap of model types they support, and it's fantastic.

There's one more thing that we can do with the model: we can try to implement the wireframe for our model. So, after we define the model-loading code, add the following line to set the polygon mode for our model:

```
Model ourModel( "res/models/nanosuit.obj" );  
  
glPolygonMode( GL_FRONT_AND_BACK, GL_LINE );
```

Save this code and you'll get the wireframe version of our model:



You can observe that the wireframe is basically just the lines – the polygons – that make up our shape. You can see all the different intricacies of our shape.



---

## Summary

In this chapter, we learned how to set up Assimp (Open Asset Import Library) on Windows using CMake for all our model-loading needs. We also covered setting up the Assimp library on macOS X. Then we created a cross-platform `Mesh` class and the `Mesh.h` header file. We also explored how to load a 3D model into our game and generated the wireframe of our model.