Quiz

1. What is ‘decoupling’?
2. Why is it important?
3. What filetypes are useful for storing game related data?
4. Why should we store our game data outside of our framework?
5. What are the advantages of using game tools?
6. What are the disadvantages of developing game tools?
7. Why would we use an open source engine?
8. What is a scene graph?
9. If each object in our scene is in a gameobject class and exists as part of a scene graph, what benefit does this provide?

# OBJ Loading

Do not begin this tutorial until you have completed all previous tutorials as it will require the use of texture coordinates and vertex normals. Be sure to read this entire tutorial (especially the FAQ at the end) before attempting to load in your own models.

## Brief Overview of the OBJ file format

The only parts of the OBJ file we’re interested in are:

* Vertex positions – lines beginning with “v”
* Texture coordinates – lines beginning with “vt”
* Vertex Normals – lines beginning with “vn”
* Faces/Indices – lines beginning with “f”

Everything else, we can ignore. This *includes* ignoring the .mtl (“material”) file that modelling software often gives us when creating OBJ files. There are only 3 things inside the .mtl file that we could possibly use at the moment which are the ka, kd, and ks values and these represent the ambient material, diffuse material and specular material, respectively.

However if we are sampling textures, then that texture sample is precisely our diffuse material, so we don’t need to read in kd! Specularity is almost always white/light-grey so we might as well just use the specular material values from tutorial 6 for all our objects, so we won’t need ks either! That leaves the ambient material (ka) and there’s no point in reading in an entire file for 3 floats especially since ambient material is just a general brightening up of the object. Some developers (such as Frank Luna who wrote the recommended reading text for this module) just re-use the diffuse material for their ambient material.

## Brief Overview of the Code

Inside the Code folder, you’ll see the OBJLoader header and source files and the Load() function is the only function you’ll need to call yourself. It simply reads in the filename you passed in, attempts to load the file and reads the vertex positions, texture coordinates, vertex normals and the indices **only** as that is only data inside OBJ files that you need for this project.

After reading in all of the data we have a small problem… OBJ files aren’t really made for optimal rendering but rather for optimized storage and so uses 3 small index buffers instead of 1 large one that your DirectX framework uses. The code then works to merge the 3 buffers into 1 using maps and more vectors. You don’t need to know how a mapping works, just know that it’s used here for fast lookup speed (which is O(1), the fastest you can get!). After merging the index buffers the code then creates the DirectX vertex and index buffers and then creates the MeshData structure using the vertex and index buffers. This last part of the code should look very familiar to you as it is doing precisely the same thing as your InitVertexBuffer() and InitIndexBuffer() methods.

## How to use the Code

Copy and paste the OBJLoader.h and OBJLoader.cpp files from the Code folder to your project folder and make sure they’re in the same folder as your other .h and .cpp files. Then inside Visual Studio, right click the project file in the solution explorer and go to Add->Existing Item and import the two files. Before we can actually start using it though, we need to make two quick changes to our existing code.

### 1) Placing our structures into a separate file

If you already have your structures placed into a separate header file, you can safely ignore this step but just make sure the file is called “Structures.h”, though you will need to remove the SimpleVertex structure located in the OBJLoader.h file.

The OBJLoader needs to know what our SimpleVertex structure looks like and this is currently placed in Application.h, we *could* #include our Application.h file at the top of the OBJLoader.h file but then we’re coupling together a lot more data than we need to. We also need to use our OBJLoader inside the Application files and if we’re #including both header files inside each other then we would run into a cyclic problem of file-A needing file-B needing file-A (needing file-B, and so on). The solution is to simply create a new file, “Structures.h”, and cut & paste the SimpleVertex struct inside there. You may also need to include the directX header files so it knows what XMFLOAT3/2 variables look like. **Don’t forget to add #include “Structures.h” at the top of your Application.h file!**

**Currently a SimpleVertex example structure is in the OBJLoader.h file, you will need to either use this structure as is and delete your SimpleVertex, or delete this one and use your own.**

### 2) Modifying the SimpleVertex structure

For fast look up speed, OBJLoader uses a map data structure but in order for the SimpleVertex struct be useable by the map, we need to overload the < operator so the map knows which vertices go “before” one another inside the map, this code is shown below.

This is what your “Structures.h” file should look like:

#pragma once

#include <windows.h>

#include <d3d11\_1.h>

#include <directxmath.h>

struct SimpleVertex

{

XMFLOAT3 Pos;

XMFLOAT3 Normal;

XMFLOAT2 TexC;

bool operator<(const SimpleVertex other) const

{

return memcmp((void\*)this, (void\*)&other, sizeof(SimpleVertex)) > 0;

};

};

Along with any other structures you may have already placed there.

Now we can finally start using the OBJLoader. First, let’s grab from of the OBJ models I’ve provided in the **Test Models** folder so copy & paste some of them to your project folder (again, the same place the .h and .cpp files are). Then do the following:

* Add #include “OBJLoader.h” to your Application.h file
* Add a MeshData variable to your Application.h file, I’ve called mine “objMeshData” for this example
* Inside your Initialize() function, **anywhere after** the InitDevice() function call, use one of the following function calls:
  + For models made with 3ds max:

objMeshData = OBJLoader::Load("sphere.obj", \_pd3dDevice);

* + For models made with Blender:

objMeshData = OBJLoader::Load("sphere.obj", \_pd3dDevice, false);

And that’s it, you use that MeshData structure the exact same way you have with the other MeshData structures, by passing it to a GameObjects Initialise method (You may want to just try rendering the object first without using GameObject). The first param is the filename, change this to whatever model you want to load. The 2nd param is just your device pointer and the Boolean value is whether to flip the texture cords or not (defaults to true, which is why it’s not needed for 3ds max function calls).

## Things that *may* go wrong

Think of this as a mini FAQ

### 1) My mesh won’t draw! It loads in fine but nothing shows up when you call draw()!

Currently in your framework you use unsigned short variables to store the indices. These are small in memory size but can only hold up to a value of ~65k. Meaning if you have more than 65k indices in your mesh, the values will overflow and not work as expected. The tutorials before had you using cubes and planes which didn’t really need many indices but now you have the ability to make your own mesh and some of you may make meshes with too many triangles. The solution is to use a different mesh with fewer triangles (which means lowering the quality).

### 2) Problems with the SimpleVertex structure in the OBJLoader::Load() function

I’m assuming your SimpleVertex looks like mine, shown above in the “Modifying the SimpleVertex Structure” section. You may have called the variables different names or have them in a different order. The solution is to just use the same code as above. Please also make sure that your shader input is in order of position, normal, tex coord (don’t worry about position being a float4) and that your inputlayout also goes in the order of position, normal, tex coord (they should be if you are following along correctly).

### 3) When creating my own models, I can’t get it to export correctly

Check the pictures inside the “Export Settings” folder to see which export settings I used for the test models. It’s imperative that you use triangles, export normal and tex coords and actually have tex coords for your model.

### 4) Textures don’t quite look correct

The current texture you have is a square and it’s quite difficult to map a square to objects such as spheres and donuts without some weird warping effects. Unfortunately I’m no artist so I don’t have much experience manually unwrapping texture coordinates and so I just used the default texture unwrappings that 3ds max and blender gave me, so blame them! If you want to unwrap the texture for the models correctly then watch these videos:

For blender users - <https://www.youtube.com/watch?v=9Ha2rFR6qO0>

For 3ds max users - <https://www.youtube.com/watch?v=oYW0y8y2hk0> (this is the first of a series of unwrapping tutorials)

### 5) My texture is upside down

In the OBJ format, there is no difference between a model that used a left hand coordinate system and a model that uses a right hand coordinate system. There is literally no way of knowing which coordinate system this obj model was made in (or exported in) which is annoying since one has texture coordinates starting from the bottom left and the other – top left. If your texture is being mapped upside down, try flipping the Boolean value when calling the OBJLoader::Load() function.

# Task 2

Understanding and planning our architecture is one of the more difficult parts of games programming.

Create a UML Class Diagram that outlines the current state of the project, with correct relationships between classes. If you are unsure as to how to format a UML diagram, take note of this tutorial which covers the basics you will need, aim for the specification layout rather than the implementation .

https://www.visual-paradigm.com/guide/uml-unified-modeling-language/uml-class-diagram-tutorial/

# Research Question / Additional Task

We briefly touched on game programming patterns this week.

Can you create a conceptual UML Class Diagram based on if we were to implement the component pattern in the current implementation, you do not have to implement this pattern, just visualize how it would look. (Don’t worry if you get it wrong, this is part of understanding how to plan a project)

Which pattern would be most suitable if we wanted to add achievements/trophies?

Some of you may have already created a class to generate Plane objects based on passing in how large we want that plane to be (procedural generation), which pattern closely resembles this one/ could be used to improve the current implementation?

If you have not implemented this yet, find which pattern you think would be best to follow for implementation.