SOFT351 Programming for Entertainment Systems: Coursework1 – 40%

This is an individual piece of work "Create a Flying Thing" together with a basic mesh loader.

The ultimate aim is for you to show that you produce a flying object like a mutant flying tiger; something where parts of the object move relative to other parts. You might have also seen the flying pig, and also a helicopter program. I will demo those sometime; do remind me.

You don't have to create a flying animal, and it doesn't even have to fly. Anything where components move relative to other components will meet the requirements, such as a helicopter with a spinning rotor, or a vehicle with rotating wheels. However, *you may not use animated meshes* at this point (anyway we have no supported animated mesh format in DirectX10 onwards). The whole point is that I want you to manipulate the matrices yourselves.

Most of you will use DirectX11. You are welcome to have a look at DirectX12 (I have no sample code at the time of writing, so you are largely on your own). You may use OpenGL (or Vulkan) instead of DirectX if you wish. However, you *must* use "new" OpenGL, the stuff that looks just like DirectX, i.e. vertex buffers, index buffers and shaders. That probably means OpenGL4 on the desktop (I know the computers in Babbage209 support that). In this case you may "nick" stuff that gives you roughly the same level of functionality as DirectX. It seems that "new" OpenGL has no maths library (does Vulkan?), and you are therefore allowed to "nick" a maths library for OpenGL such as GLM.

Part1: Flying Thing 60% (of 40% for this piece of work)

You may use any of the sample code on the module site as a starting point. If you want to use other sample code as a starting point, please ask first. This year I will also allow you to use my `Thing3D` framework as a starting point (if you want to), but the `Thing3D` framework comes with a heath warning. It has been hastily written, and is largely undocumented beyond the comments in the program code.

If you do use my framework, make your flying thing into a single subclass of a `Thing3D`. **Do not use a separate `Thing3D` to represent body, wing1, wing2 etc.** The reason is that the `Thing3D` re-calculates its own world matrix, which will make it hard to make your wings move relative to the body. As I keep saying, don't try to be too clever; let the matrices do the work for you.

Where to begin: preamble.
Most of you will probably be using DirectX11 (or 10 if you wish). You can use as much (or as little) of my sample programs as you wish, and you can also use "new" OpenGL (as above). You must use C/C++ as your programming language. **No C#, no XNA**.

What You Must Do (deliverables)

1. Start with the sample ".sdkmesh" meshes published in the module folder, or with any project you have been working on (you should have some flying stuff by now).

2. Make a couple of wings (or wheels or whatever). These can also be instances of a `CDXUTSDKMesh` in the same way that the tiger (`g_p_meshTiger`) is. You will have to use the above methods of a `CDXUTSDKMesh` to load and render the meshes for the wings.

3. Make the wings flap (or a rotor spin) relative to the body. I think many of you have already done this by now, but in case not....

   To the un-enlightened, DirectX does not appear to support relative coordinates in the way that other higher level APIs seem to. It doesn't need to. To make the wings move relative to the body, you need

- Construct a matrix (`matTigerWorld` here), which contains all the rotation and translation information for the body.

- Thee the rendering device that `matTigerWorld` is the world matrix: You will see that this is the matrix that is passed to the shader. You must also multiply the world, view and projection matrices together (in that order) and pass them to the shader. Use a constant buffer to pass the matrices to your shader.

- Render the tiger's body mesh `renderMesh(pd3dDevice, &g_MeshTiger);` (Or if you are using a Thing3D you will find the rendering in Thing3D::RenderForMyImplementation()).

- Construct another matrix containing the rotation information for the wing, say `matWing1`. You might need to translate your wing as well, but the translation will ultimately be relating to the body.

- Multiply your wing matrix and your body's world matrix to make, say, `matWing1World`. Give the new matrix to the shader (again use a constant buffer) and render the mesh. You will need to think carefully about the order in which you multiply all these matrices. If you get it wrong, the results are, um, interesting.

- Render your wing, rotor or whatever.

- You may need to find a way of making your wing flap up and down, rather than just continuously rotate.

4. Add some extra input functionality (i.e. some extra key presses) to translate the tiger, as well just rotate it.

   Your object should also be able to move forward, though exactly how it does so depends on the dynamics of whatever your object is. If you are using a Thing3D, you will already have the method `Thing3D::moveForward()`, but it would not really be appropriate as is for, say, a helicopter. And can dragons and similar creatures hover or even fly sideways and backwards like a helicopter? You decide.

5. Add some sound to the flapping wings (or whatever). You may use the DXUT classes. We haven't covered sound yet.

6. My DirectX demos usually fall over in a heap with memory de-allocation issues. One thing C# or Java programmers will miss is a garbage collector, which we do not have here. In C++, it is good practice to put some code in place to delete an object whenever you create it. My code is sloppy in that it does not do so. If you are using the `Thing3D` framework cleaning up nicely could be quite a headache, and is a soft deliverable, but do have a go.

   You will notice that all the helper classes Microsoft have written for us (such as the `CDXUTSDKMesh`) have properly written destructors. In addition, the sample framework gives us functions (or methods, depending on which one you use) which pair up with each other. E.g. in the skeleton framework you have, `OnD3D11CreateDevice ()`, where I create most of the objects, which pairs with `OnD3D11DestroyDevice ()`. Use this, together with the properly written destructors in the Microsoft helper classes, to prevent the usual "Nigel demo errors" as the application quits.

7. Try adding some other interesting and creative effects. I leave this largely to you, but an easy effect to include would be a flat ground plane and a shadow (with the wings flapping, of course). In fact, you will need a ground for the physics bit anyway. Sky boxes or spheres are relatively easy to add and very effective at enhancing the look of your project.

   Or maybe shadows or shader effects, such as having the wings bend as they flap, nice lighting effects, even a tiger with fur (I'm working on that).

   …and finally (this part), the deliverable is *not a playable game* at this point, but you can reuse anything you feel pleased with in the second assignment.

Part2: 40% (of 40% for this piece of work) write a basic mesh loader

You could even call this Part0. I have made it Part2 as to create a decent loader is, um, challenging, and that and in particular combining your loader with your part1 is how you get your first class grade.

As you will know, the ".sdkmesh" file format is not supported beyond the M$ sample framework, it would be nice to have a loader that works with more common file formats. Something like the WaveFront ".obj" files which are common, easy to understand and human readable. So, create a simple loader that works with simple obj meshes. The file format you support is up to you.

Some sample code to get you started now exists, look in "Tutorial and Sample code/ Dx 11 / Microsoft Tutorials adapted by Nigel" for "Start of OBJ loader".

The sample program opens an obj file and creates a vertex and index buffer from it. Look at the function `LoadMesh()` at the bottom of the program. It opens a Wavefront obj" file, creates a list of vertices (it just looks for all lines starting "v"), then a list of indexes (it just looks for all lines starting "f"), and assumes the mesh is triangulated and the faces are represented as vertex/texture-coordinate/normal faces. The sample will fail with any other representation. To see what this means, consult the Wiki.

Once the lists of vertices and indexes have been created, a vertex buffer and index buffer are created and the rest should look moderately familiar.

However, my sample **does not**

Read normal vectors
Read texture UV coordinates
Read the materials (the lighting properties) from the associated ".mtl" file
Create any textures (which are indicated in the associated ".mtl" file)
Handle meshes with subsets (meshes within meshes or groups in ".obj" jargon)

...And it only works with meshes in exactly the right format. It is a get yourself started point, nothing more. I certainly don't expect you to create a loader that works with all meshes, keep it simple. Look in the "build your own loader" lab.

Check my sample code very carefully. According to the Wiki, ".obj" index numbers of vertices etc. start at 1, where my program indexes from 0. I'll let you think about this....

Also, for those struggling to get this to work, here's another thing. Look at the simple triangulated textured cube (now published on the DLE in "resources / meshes / OBJ meshes". You will see that the number of texture UV coordinated exceeds the number of vertices. This is not a problem so long as we aren't indexing non-existent vertices (which we aren't).

However, this does mean that looping `numVertices` times to create the vertex buffer / index buffer pair is going to fail. I think we need loop `numFaces` times. You will then find you are duplicating a vertex a few times in the vertex buffer. This will cause no problems (he writes with optimism) and is consistent with the way the textured cube is hard-coded into the M$ sample code on which I based my sample.

Hand In

This year you will submit using the DLE", the new UoP Electronic coursework submission system. You will submit:

A single zip archive containing

1. The Visual Studio project (not just the ".exe"). *Please remove the "debug" folder and the ".sdf" (intellisence) file, and also the. ipch (pre compiled headers, of you are using pre-compiled headers) folder*, all of which are big, and all of which will be re-created by Visual Studio. But do include any resources such as meshes and sounds that your project needs.

   On that subject, you are permitted to hand submit two projects, the "flying thing" and the mesh loader. However, the holy grail is to have the loader and flying thing as a single project, so that the objects that move around are meshes you load yourself. The top grades will be awarded to such a solution.

2. A brief "paper" write-up, which can be in the form of either a Word document or a PDF document. This is intended to be a light touch on the documentation.

- Which version Visual Studio and which DirectX SDK you used (or whatever else).
- What I am looking at from an end user's point of view. How do I work it?
- What I am looking at from a programmer's point of view. How your program fits together, the flow through your program. But this is *not* intended to be an exercise in formally documenting program code.
- Anything else which will help me to understand how your prototype works.
- An evaluation of your prototype and your thoughts about what, if anything, you might do differently knowing what you now know. And feel free to blow your trumpet and draw my attention to anything you are particularly pleased with.

Assessment

There is a basic marking grid published alongside this document. This coursework contributes 40% to the module's overall grade. The module's learning outcomes, as defined in the Definitive Module Record, assessed here are:

LO1: demonstrate knowledge of the concepts and issues of programming in a high performance real-time graphics environment.

Due In before: 1200 Wednesday 8th November 2017. To be returned 6th December 2017

Hand in to: Electronic submission to DLE. Please note: at busy times, the DLE can become very slow. The slowness of the system is not a valid reason for late submission, so please don't leave it to the last 5 minutes! Get a submission "in the bank" early, you can submit as many times as you wish up to the deadline.

Also note, only DLE submissions can be accepted. The DLE can time out if you are submitting big archives over slow connections. Reduce the size of your submission by removing the folders "Debug" and "pch" (if you are using pre-compiled headers) and also the intellisense ".sdf" file.

Nigel Barlow October 2017.