­­

Further game and graphics concepts report

ANTHONY STURDY



Table of Contents

[User Guide 3](#_Toc26956494)

[Controls 3](#_Toc26956495)

[Features 3](#_Toc26956496)

[Engine 3](#_Toc26956497)

[GameObject 3](#_Toc26956498)

[Hierarchical Scene Graph 3](#_Toc26956499)

[Loading Scene from JSON 4](#_Toc26956500)

[Camera 4](#_Toc26956501)

[Vector3 4](#_Toc26956502)

[Shaders 5](#_Toc26956503)

[Default Shader 5](#_Toc26956504)

[Lighting 5](#_Toc26956505)

[Specular and Normal Mapping 5](#_Toc26956506)

[Game 6](#_Toc26956507)

[GameObject (Plane) 6](#_Toc26956508)

[Ship Controller 7](#_Toc26956509)

[Boat Shader 8](#_Toc26956510)

[Third Person Camera 8](#_Toc26956511)

[First Person Camera 8](#_Toc26956512)

[Skybox 8](#_Toc26956513)

[Water Shaders 8](#_Toc26956514)

[Fog 8](#_Toc26956515)

[Application Architecture 9](#_Toc26956516)

[Class Diagram 9](#_Toc26956517)

[Test Documentation 9](#_Toc26956518)

[References 9](#_Toc26956519)

# User Guide

### Controls

* 1-6 Switch Cameras
* W Move Ship Forward
* A and D Steer Ship (only works when moving)
* Arrow Keys Rotate Camera
* Q and E Zoom Camera (only third person camera)
* F1 Wireframe mode

# Features

## Engine

### GameObject

The GameObject class stores data about objects such as Mesh Data, Position, Rotation, Scale, Material, Textures and ShaderType.

Loading models and textures are also handled in the class constructor. The constructor takes a string for the model directory and the diffuse texture directory, then the Mesh, Diffuse texture, Specular texture and Normal texture are all initialised.

1. //Load model into MeshData
2. **if**(modelDir != "")
3. mesh = OBJLoader::Load(**const\_cast**<**char**\*>(modelDir.c\_str()), \_device);
5. //Load Texture
6. CreateDDSTextureFromFile(\_device, textureDir.c\_str(), nullptr, &diffuseTex);
7. CreateDDSTextureFromFile(\_device, AddSuffixBeforeExtension(textureDir, L"\_normal", L".dds").c\_str(), nullptr, &normalTex);
8. CreateDDSTextureFromFile(\_device, AddSuffixBeforeExtension(textureDir, L"\_specular", L".dds").c\_str(), nullptr, &specularTex);

The code snippet above shows how the model and textures are loaded from the constructor. The mesh is handled by the OBJLoader library and the textures are loaded using CreateDDSTextureFromFile(), I have a function which adds \_normal and \_specular before the extension, which automatically loads the textures without having to pass more parameters into the constructor.

### Hierarchical Scene Graph

The hierarchical scene graph allows me to have parent and child objects in my scene. This allows my scene to be organised and have objects move, rotate and scale relative to their parent object.

Each SceneGraphObject holds a pointer to the GameObject it represents, it’s parent object, a sum of the total position, rotation and scale of all the parent objects and a list of child objects.

When a UpdateTransform() is called on a SceneGraphObject, the position of it’s parents are local position are applied to the GameObject it represents. The order of which the transformations are applied is important for the Scene Graph to work as intended. The order of transformations is as follows:

1. Local Scale
2. Parent Scale
3. Local Rotation
4. Local Position
5. Parent Rotation
6. Parent Position

In a different order, the results of the transformation could be wrong. The UpdateTransform() function is recursively called on all of the SceneGraphObject’s child objects.

### Loading Scene from JSON

The scene is stored as a JSON file and loaded upon the start of the application. To load and handle the JSON data, I used a library by GitHub user ‘nlohmann’. A link to the library can be found in the references at the bottom of the report. Using the library simplified the process of reading the JSON file greatly.

The .json file is first loaded as a string, then parsed as a json object (which is handled by the library).

1. **void** SceneGraph::LoadFromFile(std::string fileDir) {
2. std::ifstream inFile;
3. inFile.open(fileDir);
4. **if** (!inFile.is\_open())
5. **return**;
7. std::stringstream ss;
8. ss << inFile.rdbuf();     //https://stackoverflow.com/questions/2602013/read-whole-ascii-file-into-c-stdstring
10. json j = json::parse(ss.str());
11. CreateSceneGraphObject(nullptr, j["0"]);
12. **int** i = 0;
13. }

The first object (named ‘0’ in the file) is passed into the CreateSceneGraphObject() function which then checks for tags to input Model Directory, Position, Rotation, Scale, etc. into a new GameObject’s constructor. Once the GameObject is created, a SceneGraphObject is also created. A loop is then performed for every child item on the current json object, if an item is found which is not any of the GameObject parameters, it must be a child object. The function is then recursively called using the child object it finds.

### Camera

The Camera class in my application stores the Eye, At and Up vectors, as well as the near and far depth plane. It also contains the view and perspective matrix used for rendering, which are both calculated in the Update() function.

The Application class stores a vector of cameras, there is also a Camera pointer called currentCamera, which is what gets updated and is used for rendering. The currentCamera pointer is changed when the user presses 1-6 on the keyboard.

### Vector3

The vector3 class was part of the week 2 additional tutorial. It stores 3 float values: x, y and z, and also has numerous functions which may be convenient when using a vector3. For example, it has functionality for calculating the dot product, cross product, normalising, squaring the vector, getting a distance between two vectors and getting the magnitude of the vector.

### Shaders

The shader class stores information about a shader, which can then be enabled in the Draw() function. Each Shader is initialised at the beginning of the application and is stored in a vector, the vector is then labelled using an enum called ShaderType. The individual GameObjects store a ShaderType which is then used to specify what shader is enabled in the Draw() function:

1. Shader\* s = shaders[gameObjects[i]->GetShaderType()];
2. \_pImmediateContext->VSSetShader(s->GetVertexShader(), nullptr, 0);
3. \_pImmediateContext->VSSetConstantBuffers(0, 1, &\_pConstantBuffer);
4. \_pImmediateContext->PSSetConstantBuffers(0, 1, &\_pConstantBuffer);
5. \_pImmediateContext->PSSetShader(s->GetPixelShader(), nullptr, 0);

### Default Shader

The default shader is what is assigned to an object when no other shader is specified. The features of the default shader are Texturing, Ambient lighting, Diffuse lighting, Specular lighting and fog.

### Lighting

Lighting is calculated in the pixel shader. Specular lighting is calculated using the direction of the light, the normal of the surface it hits and the position of the camera (eye). Firstly, the reflection vector is calculated between the light direction and surface normal. Then the dot product is calculated between the camera position and the reflection vector, which is then checked to see if positive and multiplied to the power of the SpecularPower float specified in the constant buffer. Lastly, the specular amount calculated previously is multiplied by the material settings and then multiplied by the texture map sample to give a final specular result.

1. float3 r = reflect(-LightVecW, normalW);
2. **float** specularAmount = pow(max(dot(r, input.eyePos), 0.0f), SpecularPower);
3. float3 specular = specularAmount \* (SpecularMtrl \* SpecularLight).rbg \* txSpecular.Sample(samLinear, input.Tex);

Next, the diffuse is calculated by getting the dot product of the light direction and surface normal, which is multiplied by the material values.

1. **float** diffuseAmount = max(dot(LightVecW, normalW), 0.0f);
2. //Compute diffuse colour
3. float3 diffuse = diffuseAmount \* (DiffuseMtrl \* DiffuseLight).rgb;

The last type of lighting I calculate is the ambient lighting. Ambient lighting is global lighting without a direction, and is calculated just by using the material values passed into the constant buffer.

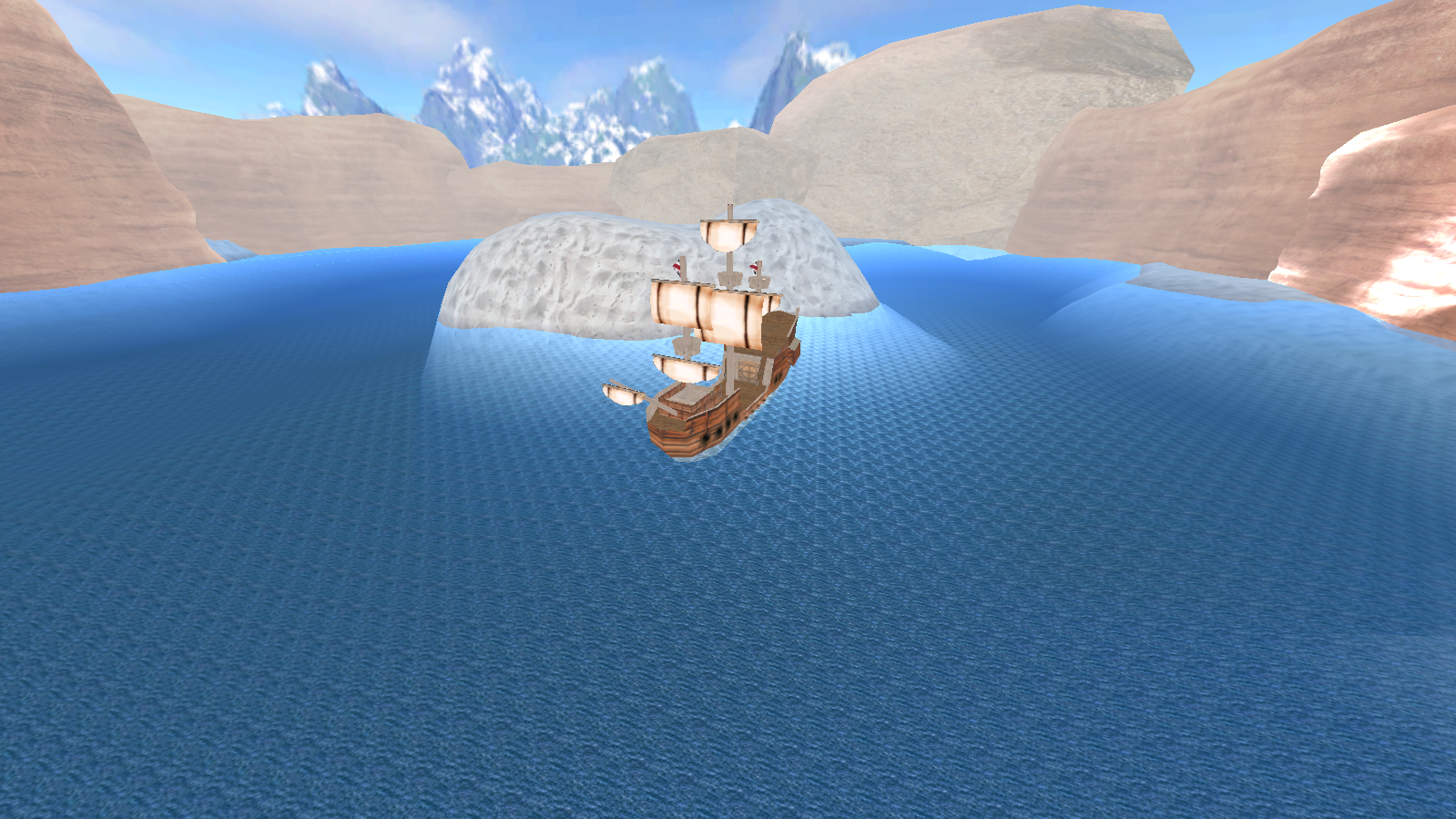
1. float4 texCol = txDiffuse.Sample(samLinear, input.Tex);

### Specular and Normal Mapping

Specular lighting, as shown above, samples the specular texture and multiplies the calculated specular amount by the sample. The specular map is a black and white texture which ranges from 0-1, so completely black means no specular, and completely white means full specular.

For normal mapping, I sample the normal texture and add the values to the Normal value which is input to the pixel shader. Here is the difference without normal mapping, and with normal mapping.





The first one is specular and diffuse texturing only, the second has specular, normal and diffuse.

## Game

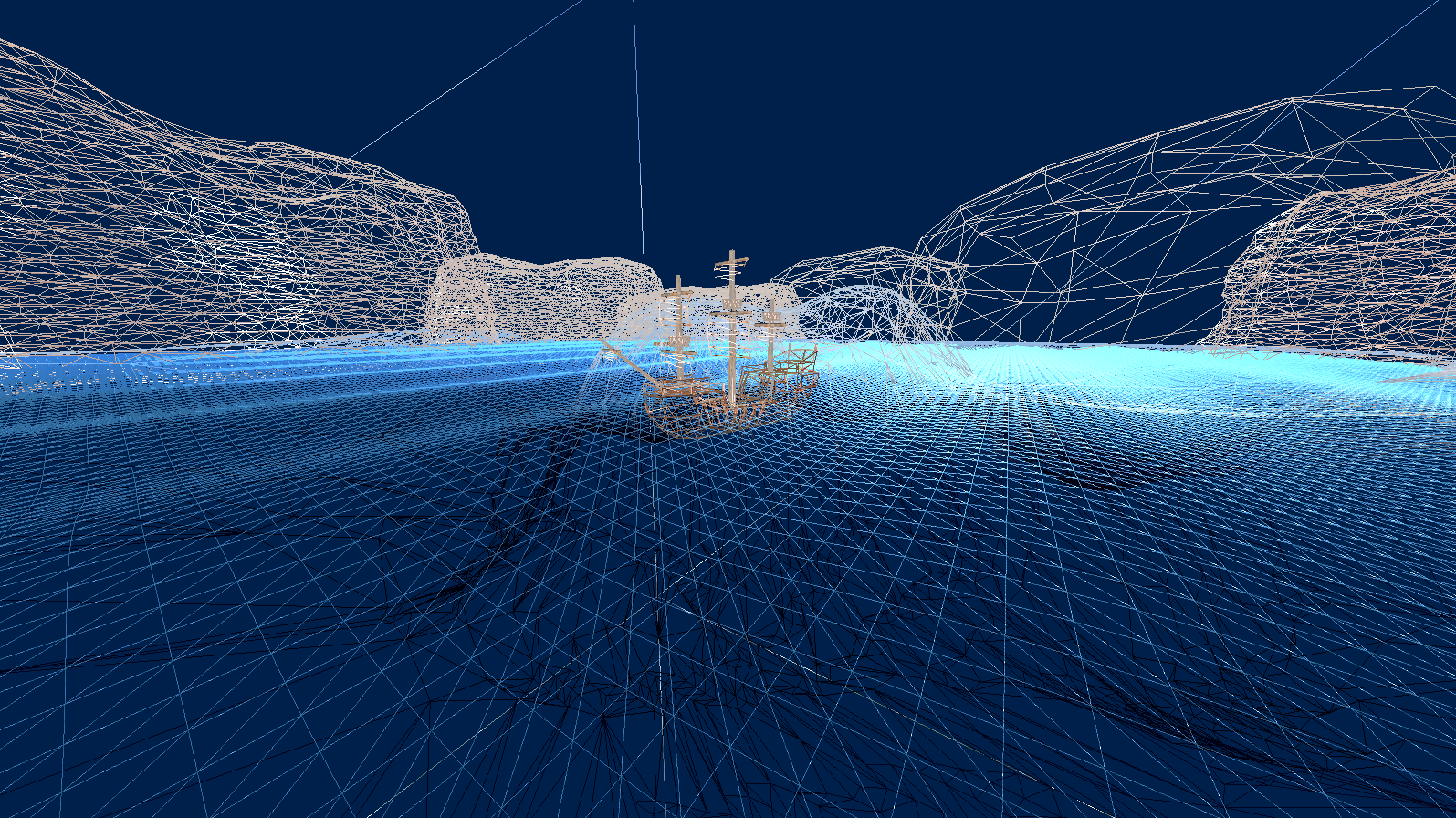
### GameObject (Plane)

The GameObject\_Plane class is a child of the GameObject class. The difference is that the GameObject\_Plane doesn’t load a model, but instead generates the mesh. A size is input to the constructor which is the width and height of the plane. Firstly, the object generates the mesh vertices, which is simply a nested for loop up through the width and height, and for each iteration, a SimpleVertex is pushed with the position and texcoord being the values of the current iteration.

The indices are generated second. It also uses a nested loop but instead of looping the full width and height it loops width-1 and height-1. This is because instead of each iteration representing one point of the mesh it’s representing one square, and access the next two vertices on the X and Y axis. So if it were to loop the full width it’d try and access outside the range of the vector.

1. std::vector<unsigned **short**> returnVec;
3. **for** (unsigned **int** y = 0; y < depth - 1; y++) {
4. **for** (unsigned **int** x = 0; x < width - 1; x++) {
5. //Triangle 1
6. returnVec.push\_back((y)\* width + (x));
7. returnVec.push\_back((y + 1) \* width + (x));
8. returnVec.push\_back((y)\* width + (x + 1));
10. //Triangle 2
11. returnVec.push\_back((y)\* width + (x + 1));
12. returnVec.push\_back((y + 1) \* width + (x));
13. returnVec.push\_back((y + 1) \* width + (x + 1));
14. }
15. }
17. **return** returnVec;

The MeshData object is then populated with this data and the object is rendered as normal. This is what the mesh looks like in wireframe:



### Ship Controller

The ShipController class takes a GameObject as input to the constructor, which is the object it moves on input. The controller is simple, and moves the ship forward when W is pressed, if the player is moving forward A and D can be used to rotate the ship left and right. The player can’t rotate when the ship is stationary as I feel it would look unrealistic.

The direction of the ship uses the ships Y rotation, then converts that rotation to a vector and moves the ship in the direction of the vector.

### Boat Shader

For the ship to ‘bob’ on the water and look natural, I had to create another shader for objects floating on the water. This shader uses the position in the world, then applies the same sin values which the wave has at that point.

The wave shader uses time and the current vertex’s X position for the wave:

1. **float** frequency = 0.3f;
2. **float** magnitude = 0.4f;
3. output.Pos.y += sin(gTime + (Pos.x \* frequency)) \* magnitude;

I couldn’t do this for the boat otherwise the boat would be wavy, so instead I get the X position from the world matrix and use that as the X position so there is only 1 X position for every boat vertex:

1. output.Pos.y += sin(gTime + (World[3][0] - 4) \* 0.3f) \* 0.4f;

### Third Person Camera

The third person camera uses the Scene Graph for rotation. There is an invisible cube on the boat which acts as a pivot point to rotate, then there is another invisible child object of that cube positioned away from the boat which is where the cameras position is. As the parent cube rotates, the child cube rotates around the boat. This is rotation is then controlled in the Application class.

### First Person Camera

The first person camera uses the same system as the Third Person Camera, but swaps the Eye position and Look At objects, and also uses opposite left and right controls.

### Skybox

The Skybox is a cube with inverted faces, the scale is large enough to fit the whole map inside, and the position of the cube is locked to the position of the currentCamera to give that it’s far away.

### Water Shaders

The water uses multiple shaders to give the effect of depth. The first shader is applied to the water mesh, which applies the waves to the surface of the water.

The second mesh does most of the work to make the water look like water, and is applied to the Seabed object. In the pixel shader, if the position is less than 0 (below the water), a gradient is applied which interpolates between the original output colour and a deep blue based on how far below the surface the position is. Vertices below sea level also have a sin wave on the position to give the appearance of warping by the waves in the water.

### Fog

Fog is calculated based on how far away the vertex is from the camera. Similarly to the water, the original output colour is interpolated to a grey fog colour.

1. **float** dist = input.Pos.z / input.Pos.w; //Distance to pixel being rendered
2. float4 fogCol = float4(0.8f, 0.8f, 0.8f, 1.0f); //Colour of fog
3. outCol = lerp(fogCol, outCol, saturate(dist \* 25));

# Application Architecture

To keep my program abstract, I tried to use non-specific classes and use object orientation where applicable.

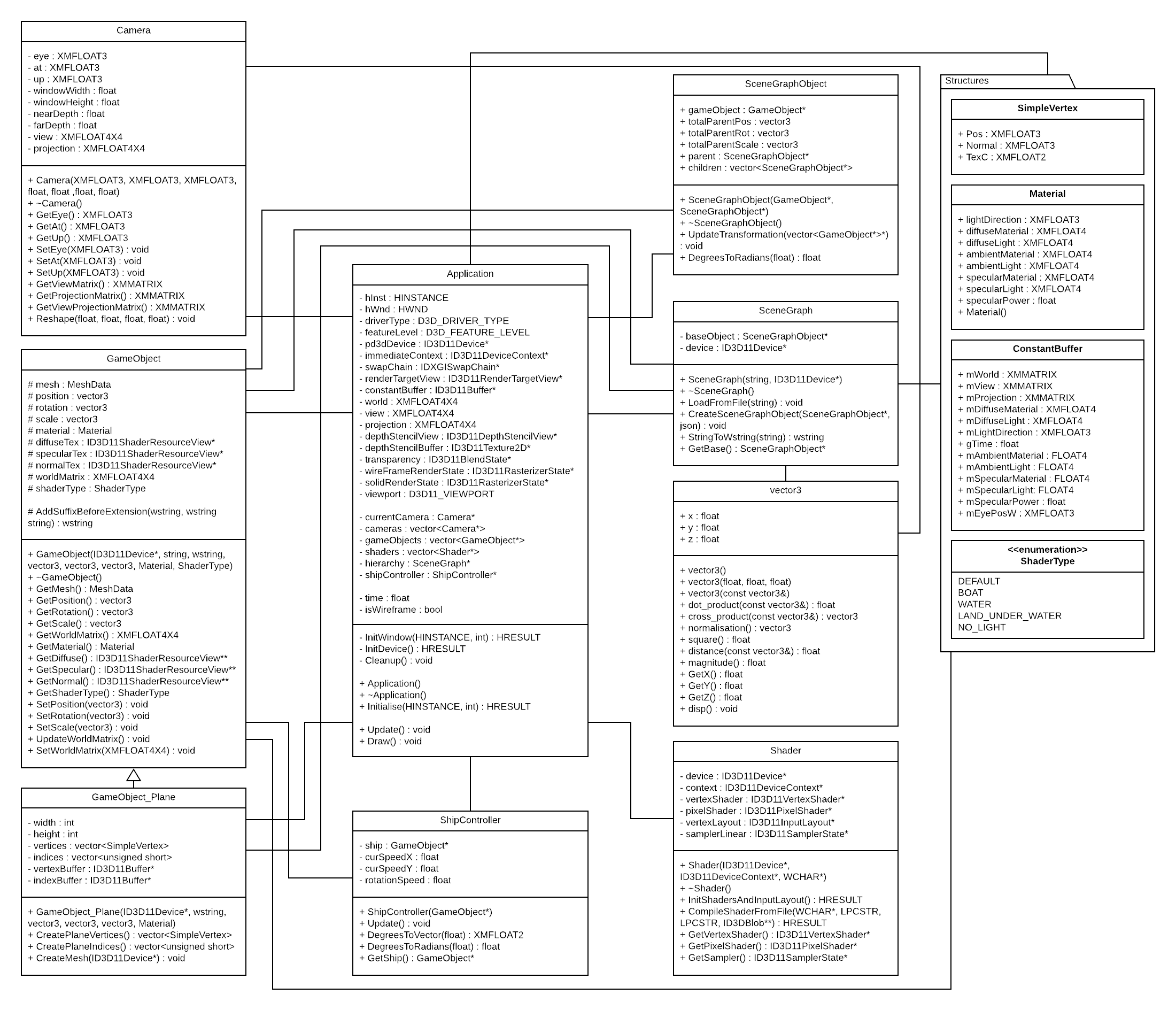
For example, instead of using a different class for each object, I have a general GameObject class which loads and object and store data about objects. I then created a child class of the GameObject class for the Plane variation as it functions differently to the parent class.

Instead of compiling a shader per-object, I compile all the shaders once at the beginning of the program then use an Enum to select which shader is used for that object.

The camera class stores information about the camera such as the view and projection matrix, the application has a pointer to a camera which is used for rendering, this pointer is swapped between different camera objects depending on the user’s input.

The ShipController is a separate class to the actual ship GameObject. It stores a pointer to the Ship GameObject to set position and rotation, but all input and logic is controlled from the ShipController class.

# Class Diagram



# Test Documentation

|  |  |  |
| --- | --- | --- |
| Description | Expected Output | Actual Output |
| Ship moves forward in correct direction | Moves forward in the direction it is facing | Moves forward in the direction it is facing |
| Ship Steers on input only when moving forward | Ship slowly rotates | Ship slowly rotates |

# References

* “SandMountain.obj” Model, “Mountain.dds” Texture: <https://www.turbosquid.com/3d-models/free-obj-mode-canyon-cliff-desert-mountain/988758>
* “Rock1New.obj” Model, “rock.dds” Texture: <https://free3d.com/3d-model/low-poly-rock-4631.html>
* “Ship2.obj” Model, “pirateship.dds” Texture: <https://www.turbosquid.com/FullPreview/Index.cfm/ID/469717>
* “Canoe.obj” Model, “canoe.dds” Textures: <https://www.turbosquid.com/FullPreview/Index.cfm/ID/1393215>
* “pirateship.dds” Texture: <https://www.textures.com/download/woodplanksbare0450/117196>
* “pirateship.dds” Texture: <https://www.textures.com/download/woodplanksoverlapping0072/114067>
* “pirateship.dds” Texture: <https://www.textures.com/download/rooftileswood0009/8855>
* “pirateship.dds” Texture: <https://slm-assets1.secondlife.com/assets/11652790/view_large/Porthole_Frame_with_Shadow__1_001.jpg>
* “pirateship.dds” Texture: <https://www.textures.com/download/doorsmedieval0615/124318>
* “water.dds” Texture: <https://www.textures.com/download/waterplain0012/9438>
* “sand.dds” Textures: <https://3dtextures.me/2017/11/17/sand-001/>
* “skybox.dds” Texture: <http://www.custommapmakers.org/skyboxes.php>
* JSON Loading Library: <https://github.com/nlohmann/json>