# Functionality Design Document

## Current Implementation

### Shader/VertexShader/PixelShader Classes – Engine/Framework Scope

The Shader class and its subclasses encapsulate the necessary data for storing and loading shader files. The Shader abstract base class has the functionality for dealing directly with the buffer component. The VertexShader initializes a vertex shader (who would’ve thought?) and takes in an input layout to be able to map the vertex data to the shader semantics. The pixel shader does the same but without the need for the input layout.

### Buffer/VertexBuffer/ IndexBuffer/ConstantBuffer Classes – Engine/Framework Scope

The Buffer class is the abstract base class for the different buffer types. The functionality of each is similar but differs in simple ways. The first subclass, VertexBuffer, stores vertex data of a specific vertex type, as well as a stride of the vertex types length. This is set to the most complex vertex type (Vertex\_PosTexNormTanBinorm) but in the future, I wish to make this dynamic (or remove the different vertex types but that is not a dynamic option). The index buffer class is almost identical to the vertex buffer but there is no need for the template as there is only one index type (DWORD). This also means there is no need for the stride attribute here. The constant buffer class has the additional functionality to map the constant buffer resource to the deviceContext, which is necessary for passing data to the shaders every frame.

### Texture/Color Classes – Engine/Framework Scope

The texture class handles the loading and storing of texture data. It can load any bitmap or DDS texture file. This makes it reusable for the various textures a mesh could have. If a texture is missing or fails to load the system creates a 1x1 pixel colour Texture using the custom Color class. This is a simple class for handling the byte data of colours.

### Model/Mesh Classes – Engine/Framework Scope

The Model class stores information required to render models, which have been loaded from files. It uses the external library Assimp, to form a scene graph of the meshes in the file. The system can load any supported type, including .obj, .fbx, and .blend types. These meshes are stored and cycled through upon a draw call. The mesh data itself is stored in the Mesh class. This class is where the actual draw function is called, with the vertex and index buffers being set here, as well as loading in textures. It contains a list (vector) of all the textures related to the mesh, including diffuse, specular and normal maps. The model class handles the loading of the textures into Texture objects.

### GameObject/RenderableGameObject Classes – Engine/Framework Scope

The GameObject class encapsulates all the data about an in-game object. This is an abstract base class, used by the light objects, the camera and the RenderableGameObject. The RenderableGameObject is a subclass of GameObject. It has the additional functionality of a model attribute. This stores a model type which can be drawn to the screen.

### Light/DirectionalLight/PointLight Classes – Engine/Framework Scope

The Light class is an abstract class which is inherited by the subtypes of light. The Directional light inherits directly from the GameObject class. I have used a macro and a new function call to repurpose the inherited position data as directional data for this light type. The point light has additional attenuation attributes and inherits from the RenderableGameObjectclass. This is because it has an attached lightbulb model, to help visualize where the light is in development builds.

### Camera Class – Engine/Framework Scope

The camera class contains the camera-specific functionality. This includes the creation of the view projection matrix. A camera takes in projection values, such as the FOV, and produces a view matrix and a projection matrix, which can then be passed through the pipeline. It inherits from GameObject class.

**Config Class – Engine/Framework Scope**

The Config class is used to store and process the initial setup data. The graphics handler takes in a config type as part of its constructor (or Initialize function). The config type is populated from a config file, with information such as window size and vsync settings.

### Application Class – Engine/Framework Scope

The Application class is where the main game loop functionality occurs. Any frame updates occur here, and this is where input functionality is determined, i.e. when a is pressed walk left. It has attributes of GraphicsHandler, Window, Keyboard and Mouse types.

### GraphicsHandler Class – Engine/Framework Scope

The graphics handler is the wrapper for the main DirectX pipeline. It has functions for initializing the DirectX functionality such as the swap chain and rasterizer states, initializing the shaders and input layouts of the system and finally for initializing the scene, which involves loading game objects setting positions and modifying start values of constant buffers.

### Window/Keyboard/Mouse Classes – Engine/Framework Scope

The Window class encapsulates the WinAPI code for creating a window. It handles all the messages the system receives. The Keyboard class handles input with functionality for KeyDown, KeyUp and Char data, as well as controlling repeat presses. The Mouse class does much the same, except instead of keyboard keys, it handles mouse buttons. This includes scroll wheels for any mouse wheel interval. The system can capture mouse input as a position or as a delta from the last position.

### ConstantBufferTypes Header – Specific Application Scope

This header contains a series of structs which are used with the shaders. These structs represent the constant buffer data which each shader requires. The system is currently hardcoded to use a certain type of constant buffer, but I hope to make this dynamic soon. An exemplar type is CB\_vertexShader which contains the attributes; worldMatrix and worldViewProjectionMatrix, which are used for transforming from world to view space and so on. These types are specific to the requirements of the application.

### VertexTypes Header – Specific Application Scope

Similarly, to the constant buffer types, this header contains many struct types. Each struct represents a vertex data layout, which matches’ and input layout. This allows us to populate our vertices with more information to pass to the shader, such as texture coordinates and colours. These types are specific to the application's requirements.

### Shader Files – Specific Application Scope

The shader files are the shaders themselves. They are written in HLSL and change how the data is rendered to the screen. The most obvious use of shaders is to perform lighting calculations, making scenes feel more immersive. This would be performed in the pixel shader. The vertex shader controls the modification of vertices, so a tessellated mesh, for example, would be created in a vertex shader. I have many different shaders all specifically used for this application. Changing a shader may require the creation of a new input layout or vertex type, making them application-specific.

## Future Features

## Classes:

### Bindables (and its children) Class – Engine/Framework Scope

The Bindables class would be an abstract base class, which would have many children. These children would be wrappers for the various DirectX ‘bindables’. This includes rasterizer state, shaders, buffers, blend states, depth stencils etc. Each game object would then store a list of pointers to the Bindables and bind them during the draw call.

### Loader Class – Engine/Framework Scope

The loader class would handle all the data loading. This would allow for efficiency when loading models and textures, as if the model has already been loaded before it does not need to be fully loaded again. The same goes for textures. By storing lists of the data, the classes which use it, i.e. Mesh, would contain a pointer to a model’s data and textures. If any of these models/textures are modified during runtime, new instances of them are created and stored, so that the original data is not corrupted (effectively the data from the file is const)

### Audio Emitter Class – Engine/Framework Scope

The audio emitter would be a new subclass of GameObject. It would be used to have location-based audio, with features such as attenuation based on the current camera distance.

### Skybox (Cubemap) Class – Engine/Framework Scope

The Cubemap or skybox is a unique type of item which is equivalent to a cube with textures on its interior. It would inherit from the GameObject class and incorporate the existing texture class to load its faces. The skybox class is engine specific but the textures for the skybox, as well as the shader used, are application specific.

### SpotLight Class – Engine/Framework Scope

Like the directional and point lights, the SpotLight class is another type of light. It has both a direction and attenuation, so it inherits functionality from both. It will also have attributes for range and radius (or angle of spread) to dictate the cone. It may require a change in how the directional light works as it will need to have a position and a direction. This may mean just switching the directional light to override the rotation vector instead of the position vector.

### Boat Class – Specific Application Scope

The Boat class would inherit from RenderableGameObject. It would be specifically designed to load and control a boat model in the environment. It may have some additional attributes such as velocity.

### WaterMesh Class – Specific Application Scope

The water mesh would be another application specific class, inheriting from RGO. This would have the functionality required to tesselate and manipulate a plane to achieve a water-like surface, with waves and buoyancy etc.

### Cloud Class – Specific Application Scope

The cloud class would be another form of RGO. This would create random clouds using some form of noise or other randomness which would change of time giving them volume and creating a dynamic feel

## Functionality:

Additional functionality could include:

* Support for multiple lights of the same type – Engine/Framework Scope
* Loading scenes from files (not just the components but the layout as well) – Engine/Framework Scope
* Greater control over shaders (bind shaders to models) – Engine/Framework Scope
* Export scene configurations so that tweaks can be made at runtime. – Engine/Framework Scope
* 3D picking of GameObjects for object specific controls. – Engine/Framework Scope
* Multiple viewports (cameras) – Specific Application Scope