

***A Frenetic First-Person Shooter (FFPS)***

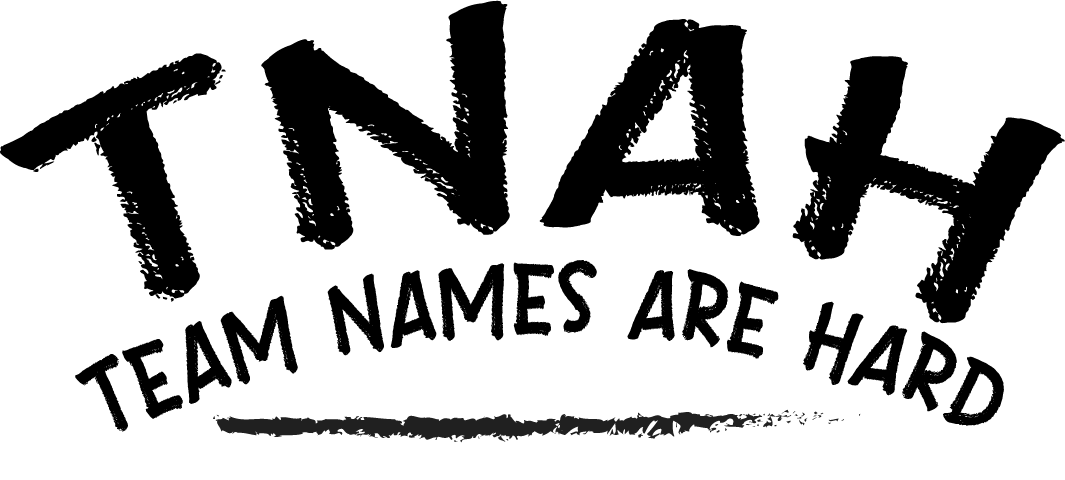
**Game SoftWare Design Document**

Version 2.0

Produced by Dylan Blereau, Bryce Standley, and Christopher Logan

Created: March 31, 2021

Updated: May 28, 2021



# **Game Engine Architecture**

## **Third Party API**

Below are all the third-party API that were used to develop the ZOOM Engine.

### **OpenGL**

OpenGL (Open Graphics Library) is a strong open-source programing API that allows for rendering 2D or 3D graphics. We use it to render 3D graphics, using a core-version of the API and set it up using GLAD. The way the API is used allows us to replace it or add functionality for a different API, as there is a façade for rendering.

### **GLFW**

Graphics Library Framework is an open-source library with the primary objective of managing and creating windows. It can be used with OpenGL, OpenGL ES and Vulkan, making it a good overall choice if there is any need for multiple APIs to be used. In our engine it has a façade for the window, and it will handle the window, inputs, and other buffers.

### **GLAD**

Glad is a loading generator that allow for allows us to pick the specific needs for OpenGL that we require. We are using OpenGL 3.2 with core profiles, allowing for more flexibility.

### **Assimp**

The Open Asset Import Library (also known as Assimp) is a powerful open-source library that is capable of loading in 3D models of various formats. Assimp’s ability to load in 3D model types such as OBJ and the more advanced formats such as FBX (the model format we are using) quickly made it a library that we knew we would require in our engine.

### **OpenGL Mathematics (GLM)**

GLM is a header only C++ mathematics library that is based on OpenGL’s GLSL shading language. The fact that GLM provides a simple and easy way to define and use various vectors and matrices made it an extremely attractive third-party tool to use. While we don’t façade GLM in our engine, we felt that the need to do so was unnecessary as GLM is already abstracted enough to point where it is easily usable.

### **Lua/LuaBridge**

As per the requirements of the assignment, lua scripts are used to as a quick and efficient way read in variables and game data. They are used in various classes throughout the engine and allow for changes to be made to the engine without having to recompile. Lua paired with LuaBridge aid in being able to development a game engine that is inherently reusable and customizable. Lua scripts aren’t just handy to the developers, but also the users that can use the basic language to tinker aspects of the engine to their liking. The luaManager class is responsible for the instance of lua in the engine, allowing classes to call it and make use of lua scripts.

### **ImGui**

ImGui is a header only library that enables quick and easy access to creating and drawing graphical user interfaces inside C++ and a given graphics API. ImGui just needs to be imported into a project and the required API headers included, once imported, ImGui adds simple functionality to be able to create and design windows and UI elements for the application. ImGui wasn’t abstracted within the engine due to its simplicity and multi-API system allowing the engine to chose on the fly what API to use with ImGui.

## **Summary of Classes**

### **Bounding Box**

The bounding box class controls and generates bounding box information for a given object. These bounding boxes are then used to assist with collisions. For example, the player, enemies and static objects such as trees and rocks all have bounding boxes for use in collisions. The tokens that feature in the game world also make use of collisions, which don’t stop the player from moving into them, but instead remove themselves from the scene highlighting that they’ve been collected. The water also uses collisions, which allows the water to kill the player when their bounding boxes collide. The bounding box class provides enumerated collision tags for each collision type.

### **Buffer**

The buffer allows for buffers that hold vertex information to be easily stored, it is used via open gl and only held in the open gl version of the renderer, It is used to make many buffers for animation, easily allowing for the memory to be held, deleted and sent to the gpu.

### **Manager**

The manager holds simple information such as difficulty, tokens, timers, and other smaller things that need to be easily accessed by other components in the engine.

### **MD2**

The MD2 class is responsible for loading rendering and animating MD2 models. It uses the filepath to the model and its shaders and sets the model up.

### **State**

The state class is an abstract class that provides a common interface for all state classes. It is templated so that this class be reused in any state machine. The class contains an enter function which is executed once upon entering the state, an execute function which will continuously execute until the state is changed, and an exit function which is executed once when exiting out of the state.

### **StateMachine**

The stateMachine class is a templated class that encapsulates all state related data and functions, allowing for a cleaner integration of an FSM that can be used in the entity’s class. The stateMachine comprises of a pointer to the owner of the Finite State Machine, as well as pointers to the stateMachine’s current, previous, and global states. The class contains functions that allow you to get and set states, handle messages sent by other entities, as well as an update function that will update the states of the Finite State Machine.

### **GUI**

The GUI class is the base class for each derived GUI class present in the TNAH Engine. It uses an init function that initializes ImGui and calls a draw function to draw the relevant information to screen.

### **DebugGUI**

A derived class that inherits from the GUI base class. This class is responsible for setting up and displaying the debug menu.

### **GameGUI**

A derived class that inherits from the GUI base class. This class is responsible for setting up and displaying the player information on the screen such as the Points score, Kill Count etc. It also displays the results screen menu when the player dies.

### **MainMenuGUI**

A derived class that inherits from the GUI base class. This class is responsible for setting up and displaying the ZOOM main menu. The main menu consists of multiple sub menus settings menu, pause menu, difficulty menu, controls menu etc. The role of this Main Menu is to manage the transitions between all these menus.

### **EndScreenGUI**

A derived class that inherits from the GUI base class. This class is responsible for setting up and displaying the exit screen highlighting the TNAH and ZOOM development team. This class displays the exit screen whenever needed.

### **Engine**

The engine class is the main driving for of the engine. The class holds and sets up the scene, renderer, and window, bringing together each core of the engine, whilst fitting it into the Model-View-Controller design.

The engine has only a default constructor when this constructor is called it will first set up a lua state from the lua manager. It will then store some default values for height, width, name, and amount, these are stored in case there is an error with the engine.lua file. It will then attempt to read the engine.lua file, and if that works it will read in the name of the engine, width and height of the window, and the number of scenes in the engine. Once it has done this it will attempt to initialize GLFW to see if it is available, if that attempt works, it will get the window to GlfwWindow to be used in the game. It will do that same thing with glad, initializing it and if that works, it will set the renderer to be OpenGL. Once it has done both of those it will then proceed to load the game scene/world.

It also features a destructor, that will delete the pointers on exit.

There are two functions within the class, one function that handles the lua loading of the world explained above, exposing the MakeGameObject function from scenes to lua, allowing us to create them easily. The other scene is our running scene, this is called in main and runs the game loop. The running loop will first check if the scene is initially running, as it would be set to false if any issues arose in the startup of the engine. It then will run a loop based of the window status of the GlfwWindow, which will stop once the window has been exited. Within the loop there is firstly a set float that gets set every frame called deltaTime, this handles the time since the previous frame. It then calls a window update, which updates the variables from the window and sets lens values, whilst also calling projections and buffers. It then gets the exit screen from the current scene and grabs the status from the window for that variable, setting its value to it. It will also handle the position of the player and camera, updating the players position to fit the camera, and the players to the camera as well with any adjustments. It will also call for the game keyboard and mouse inputs, which handle movement and viewing, also exiting and wireframes. The final thing it will handle is calling a buffer from the window.

### **GameAssetFactory**

The GameAssetFactory is a class that is responsible for creating new game assets for the game to use. It has two parameters, a Boolean to check if a player game object has been made, and a Renderer pointer, so that the facade rendering can be passed to the game objects.

It has a single constructor which takes in the Renderer pointer, setting it to its own within the constructor. The constructor also sets the Boolean to equal false, as a player has not been created yet.

The class also has a single function, named GetGameObject, which returns a GameObject pointer, and which takes in 7 parameters, using these to determine what type of game object to create, and return. The parameters are a string for the type of game object required, a string for the model path, two strings for the vertex and fragment shaders, a float for the scale, a vec3 for the position, and a float for the speed. When called it will check if the type matches player, enemy, token or static, if it does not match one of these, it will return a nullptr. It will also return a nullptr if there has already been a player object created, as the game requires just one.

It will them apply the entered parameters as required with different game objects, setting them all to them. After it has set it to a GameObject pointer of required type, it will return it back.

### **GameObject**

The GameObject class stands as the parent class to the abstracted child GameObject classes that follow. This class essentially acts as the main one that allows for new classes to be created using it, it will never be used itself in the engine, only the abstract versions.

The class holds many variables and functions, with most of the functions being setters and getters for the variables. The main variables of the class will be the Model, which is the model for the game object, using this class to set the model, texture, and meshes. This was originally only going to be a part of a few of the objects, but design decisions lead us to determine each game object will use it, even player as they will use the model to render a gun for their position. The next variable is the Renderer pointer, which is used so that the model can have access to the renderer so that it can set the model and meshes for future rendering.

There are also variables for position, rotation, and scale, with the position and rotation being vec3 that take the x, y, z position and rotation of the object, allowing for setting of positions and rotations. The scale is a flat float, as most models will need to be scaled equally on the x, y, z axis. Each of these variables mentioned has setters and getters for them.

The class has 2 constructors, a default class that sets the values to an internal preset, a class that only takes in the Renderer pointer and then sets the rest to an internal preset. When making game objects the engine only utilizes the GameAssetFactory to create and set them that was explained in the GameAssetFactory section.

The other function that is used for the GameObjects is the Update function, which takes in a float for deltatime. This function is used for functionality of the different abstract classes as it is a virtual function.

### **Player**

### The player abstract class is used for the player themselves; it uses an update loop to update the state machine. It has a render function that will render the gun and bullet models. It also features other functions and variables used within the AI, which is used for token management.

### **Enemy**

The enemy abstract class is the same as the player at this time, with the update function being used to update the state machine of the enemy. The enemy also features specific functions and variables for health, animations, states, alive and ones used by AI such as wander, pursue, flee, evade. Each of these functions are uses within a lua script used specifically for the enemy.

### **Static**

The static abstract class does not have any functionality currently, with its main function being a model that stands in the world for the user to bump into.

### **Token**

The token class is the only one that has functionality in the update function, as it is used to spin the tokens around the x axis.

### **EnemyStates**

The enemyStates is a header file comprised of all the enemy states that feature in the enemy Finite State Machine.

### **Wander**

The Wander state represents the wander state of the enemy. This state handles the enemy behavior that occurs in the wander state. The logic it uses is written in lua.

### **Alert**

The Alert state represents the alert state of the enemy. This state handles the enemy behavior that occurs in the alert state. The logic it uses is written in lua.

### **Chase**

The Chase state represents the chase state of the enemy. This state handles the enemy behavior that occurs in the chase state. The logic it uses is written in lua.

### **Flee**

The Flee state represents the flee state of the enemy. This state handles the enemy behavior that occurs in the flee state. The logic it uses is written in lua.

### **Attack**

The Attack state represents the attack state of the enemy. This state handles the enemy behavior that occurs in the attack state. The logic it uses is written in lua.

### **Die**

The Die state represents the die state of the enemy. This state handles the enemy behavior that occurs in the die state. The logic it uses is written in lua.

### **Global**

The Global state represents the global state of the enemy. This state handles the enemy behavior that occurs in the global state. The logic it uses is written in lua.

### **PlayerStates**

The playerStates is a header file comprised of all the player states that feature in the player Finite State Machine.

### **doubleDamage**

The doubleDamage state represents the doubleDamage state of the player. This state handles the player behaviour that occurs in the doubleDamage state. The logic it uses is written in lua.

### **doublePoints**

The doublePoints state represents the doublePoints state of the player. This state handles the player behaviour that occurs in the doublePoints state. The logic it uses is written in lua.

### **HealthRefill**

The HealthRefill state represents the HealthRefill state of the player. This state handles the enemy behaviour that occurs in the HealthRefill state. The logic it uses is written in lua.

### **SpeedUp**

The SpeedUp state represents the SpeedUp state of the player. This state handles the player behaviour that occurs in the SpeedUp state. The logic it uses is written in lua.

### **Main**

The Main state represents the main state of the player. This state handles the player behaviour that occurs in the main state. The logic it uses is written in lua.

### **Death**

The Death state represents the die state of the player. This state handles the player behaviour that occurs in the death state. The logic it uses is written in lua.

### **Global**

The Global state represents the global state of the player. This state handles the player behaviour that occurs in the global state. The logic it uses is written in lua.

### **ExitScreen**

The exit screen class is a class that displays the exit screen showing the TNAH and ZOOM development team. This screen is displayed in the game by pressing the ‘X’ key.

### **Window**

The window class is a class that is design with abstraction in mind, to allow for the use of different windowing applications by adding the functionality to a child class, essentially using a façade for it. The base windows class provides a basis for functionality to be added by another class. It features many functions that allow for the window to be correctly set, for inputs and updates.

The window will use an initialize function, which will take in a string and 2 integers, for the name and the window size. This will then initialize the window and those factors. It will also have functions for both keyboard and mouse input. It will then have update functions for things such as the camera, and a general update function that provides more functionality within it. As stated, this is a parent class designed without functionality in mind and would use a child to provide the functionality.

### **GlfwWindow**

This provides Glfw functionality to the window class. It will provide functionality to each of the classes mentioned in window, by using glfws window capability’s, and using its input capabilities, as well as an input class to set the inputs. This class will have a dedicated camera, that will be used for the position and viewing of the player. This class also provides restart, projection and buffer functions that are called in the update loop, which also stands as a time setting point.

### 

### **Windows**

The windows file is used to store the Window and any following abstracted classes header files.

### **Renderer**

The Renderer is the parent class used as a basis for rendering in our engine. It is designed to be abstracted to the point where different calls, such as meshes, terrains and skyboxes are all capably rendered, and created. The class features many virtual functions, that will then be expanded and used by child classes (OpenGL).

The renderer features many functions that allow for different things to be setup and loaded into memory. The first major one is the setting up of our terrain, we have a function called Terrain Setup, which takes in a vector of vec3, vector of ints, VAO, VBO and EBO. All of these come together to provide us will all we need to be able to render a terrain, which will be done itself in any child classes, which will add the functionality that will read in the different positions, setting them to the VAO, VBO and EBO, as they are passed by reference. The same can be applied to the skybox setup, but we are passing in a vector of floats, a vector of strings, the VAO, VBO, texture ints and a shader. That allows us to render the box with the floats, set the textures with the string paths, and set the shader, whilst also have the VAO and VBO set. The last thing that we can setup from the renderer is a mesh, the mesh set up again takes VAO, VBO and EBO as reference integers, and takes in a vector of vertex and a vector of integers. This again allows us to have the values we would need to then add functionally to store and generate meshes into memory.

The class features many rendering functions. The first one is a Mesh renderer called RenderModel, this will take in a shader, a matrix4 and mesh variable, allowing for the shader to be used and manipulated for the model, the matrix4 providing scaling, rotations and other transformative aspects of the mesh, and the mesh providing the actual mesh rendering. We can also render terrain and skybox, which will only require the size and the vertex array object integer to render, as everything will be handled by a loading in functions in the renderer.

The rendering also allows for other things, such as gathering a texture from a file, setting shaders for use, and exit screen information.

### **OpenGL**

OpenGL is a child class of Renderer, as the renderer is designed to support multiple APIs. It will provide actual functionality to the different functions in renderer, that can be used in the game itself.

### **Input**

The input class holds integers that set the inputs for interacting with the game via a keyboard. The class is used by the window to set what keys are used for different things. The class a single constructor, and 7 variables which are for forward, back, left, right, wireframeOn, wireframeOff and exit. Each of these values are integers and are set in the constructor.

The constructor gets the lua instance and the attempts to read a lua file called input.lua. If this file is opened and read, it will attempt to gather the integers and set them to their corresponding variable. This allows for the inputs to be easily changed through the script file instead of the engine itself. If the input.lua file is not found, the class will revert to a default input scheme. This is forward is w, back if s, left is a, right is d, wireframeOn is arrow up, wireframeOff is arrow down and exit is escape.

### **luaManager**

The luaManager class is an extremely small but powerful class that provides the classes that use it the ability to utilize lua scripting.

The class utilizes the singleton design pattern which ensures that only one instance of luaManager can exist, and that instance can be retrieved by the luaManager getInstance () function. The luaManager has a get function named getLuaState () which returns a new luaState if its currently set to null or returns the current state if not null. It also has a basic constructor that assigns a new lua\_State. To use the luaManager in a class, it is as simple as using a lua\_State \*L and using it to point to the LuaManager: getInstance ()::getLuaState(). Doing so will give the class access to that single instance of lua. You can then use the luaL\_dofile(L, script ) to read in a lua script and do whatever you like. In most instances of the luaManager in the engine, it accesses the given lua script and assigns variables in the c++ code the values that exist in the lua script.

The luaManager may not look like it is doing too much, but its ability to allow aspects of the engine to be adjusted and customized through external files makes it one of the most impressive and important parts of our engine.

### **Mesh**

The mesh class itself doesn’t hold any functionality; all it does is hold all the information needed for it. It has a constructor that takes in a vector of vertex vector of unsigned ints, a vector of texturemMesh and a matrix4 transformation. It takes all this information from the model which retrieves that information from the Assimp loader and is the storage of it. That information is used by the renderer to get the VAO, VBO and EBO for the mesh, and that can be passed into the renderer again to display the mesh.

### **Model**

The model class is used to load in and render models for game objects. It is designed in a way where you can use either FBX or MD2, depending on what you specify, as MD2 is capable of being animated using keyframes. It provides those functions, and ones to interface into the MD2 class to run animations and render them.

### **FBX**

The model is used to setup the meshes for specific models and load the data for them. We are using Assimp to load in our models. Our model class consists of two constructors, as standard default one, and one that takes in the file path and Renderer Pointer, with the renderer used to setup the meshes.

The model is loaded in using load Model, which again uses the path given in the contractor. Within load model we use Assimp to import the files information, we then get the node, scene and transformation information from the imported file and pass it to another function which processes the nodes. The node process runs through which loops through firstly the meshes, using process mesh to get the information for the meshes to be created, getting the vertices, normal, texture-coordinates, faces, and materials, then sending them to a mesh to be created and set, then returning them, adding them to a mesh vector for that class. It does this until it has gone through each mesh, then it will run through a loop of the children of the node, recursively going through the process node again. Once this is done the whole model and all of it meshes, textures and normal will be available for rendering.

### **Camera**

The camera class is responsible for the first-person camera class that features in the engine. The class contains two constructors both of which require various parameters to be used, such as the position, up axis, yaw, and pitch. The first constructor vec3 for position and up which have already been assigned values, while the yaw and pitch also make use of values that have already been assigned. It also then uses a defined vec3 for the front of the camera, while assigning global variables for the movement speed, mouse sensitivity and zoom. At the end of function it calls the updateCameraVectors() function which calculates the front vector based on the cameras current euler angles.

The second constructor is largely the same but allows you to manually define the xyz values of the position and up vectors in the form of various floats. It also grants you the ability to define your own value for the yaw and pitch, however it still uses the same already defined values for the front vector, movement speed, mouse sensitivity and zoom.

There are a few more handy functions present in the camera class. GetViewMatrix() returns the view matrix while the ProcessKeyboard, ProcessMouseMovement and ProcessMouseScroll allow the first-person camera to move around, look around and zoom in and out respectively.

### **Scene**

The scene stands as the class that holds all the games functionality, holding the game Objects, skybox, terrain, and setting them up to run. The scene has 2 constructors, a default basic constructor, and a constructor that takes in a string for the name, and a pointer to the renderer from the engine. The scenes main interaction is with the engine, as the engine will tell it if the player has move, or if there is any inputs that will adjust it, as of now that would only be camera movement and the exit screen. The scene also a destructor, that will clear the pointers on deletion. The main functionality of the scene is to run in the engine, it is designed to allow for multiple scenes. The main starting function of the scene is the initialization, this initilsation will do a few main things. It will set up the terrain, initializing it and setting up its textures, it will also pass the required information to the renderer, allowing it to be rendered. It will do the same for the skybox, setting it up so that it can be sent to the renderer to be added to memory. It will do the same for the final exit screen, and setup the GameAssetFactory. The run function for the scene runs the current scene and its game objects, using the renderer to display the game objects, terrain and skybox, the game objects are set in the engine class via a scene script that calls another function in the scene called makeGameObject, which uses the game asset manager to general new objects. The next function is updatePlayer, which updates the players current position based on the camera. The game also features collision detection, allowing it to check if the player has collided with different objects, grabbing their tags, and providing functionality based of that.

The main idea behind the scene is it is essentially the Model in model-view-controller, as it doesn’t have any of the inputs directly attached, and only utilizes the renderer to store things, never doing anything it would need outside of it, whilst being the holding point for all the games information.

### **Shader**

The shader class is responsible for shaders in the engine, allowing you to open and use shaders, as well as set uniforms values of various types which can be used in the shaders. The Shader constructor takes in the path, of the vertex and fragment shader (geometry is allowed but set to nullptr in this case) and subsequently retrieves, creates, and attaches the shaders. Its complexity is heavily abstracted by what otherwise appears as a Shader constructor that merely takes the paths of its shaders to work, but it works well. The use() function abstracts the glUseProgram call and uses the unsigned int ID that was used to create the Shader program within the constructor. The shader also contains various set functions such as setBool(), setInt() etc. These functions much like use() hide the gl calls and are used to set uniforms of the given type by simply provide a name and variable of the given type.

Lastly, the checkCompileErrors function is an important debugging function that will display in the console the compilation errors in the provided shaders (if they have any).

The shader class is crucial for using the various shaders present in the engine, with the Terrain and Skybox both using a Shader object to make use of their shaders.

### **Skybox**

The Skybox class is a class that is responsible for creating and setting up a skybox. When it comes to variables it has a Shader object used to assign the skybox with a shader, a vector of strings named cubeFaces which takes in the path to the image that will be drawn to each of the skybox faces, as well as a vector of floats that make up the vertex positions of the skybox. Also present in the class are unsigned ints that refer to the skybox’s VAO, VBO and texture.

Function wise, the Skybox class isn’t overly complex. It has two different constructors, with both handling things quite differently. The first constructor (the one used in the engine) doesn’t take any parameters and utilizes lua scripting to get the paths to the skybox face images and shaders.

Once the values have been read in from the lua script, the paths to the cubeFaces are placed into the aforementioned cubeFaces vector while the vertex and fragment shader paths are passed to the Shader objects constructor and then assigned to the skyShader variable. The constructor then calls the SetSkyVerts function which contains an array of floats that make up the vertices of the skybox.

This approach with the lua scripting makes the skybox far easier to customize and tweak when it comes to the images and shaders used. No further compilation is required to do so either, simply change the image paths of the skybox faces to another image if you like and it will still work.

The second constructor essentially does what the prior constructor does but without the lua scripting. It can be used to hardcode in the paths to your cubeFaces and shaders, but you will have to manually change the paths in code and recompile if you decide to make changes. This secondary constructor was used initially but is no longer used as the lua scripting way is far more efficient.

The skybox class also has a GetCubeFaces and GetSkyVerts function, which return the cubeFaces and skyVerts vectors. These functions are used by the renderer in the scene class when setting up the skybox.

### **Terrain**

The terrain class is responsible for loading in a heightmap for a file and generating a terrain from it. To simplify the usage of the Terrain when it is instantiated as an object the Init() function contains all the functions required to setup the terrain object. The first function called in the Init() function is the luaLoader() function. The luaLoader() function gets the instance of the lua\_State from the luaManager singleton class and uses that to get the values from the terrain.lua script. The values from the lua script are then assigned to the variables in the class, providing them with details such as the path to the textures and the heightmap. Once all the details have been read in and assigned the function ends and returns to the Init().

LoadHeightfield(filename, size) is called next and reads in the heightmap using the filename and size that had been grabbed from the lua script. The same occurs with the setScalingFactor(scaleX, scaleY, scaleZ) function, which scales the x,y, and z-axis according to the values that were obtained from the lua script.

Lastly, the generateTerrain() function is called, which in turn calls various functions which generate the vertices (generateVertices(vertex)), colors (generateColors(vertex)), textures (generateTextures(vertex)), indices (generateIndices(Indices)) and normals (generateNormals()) of the terrain. The vertex data that has been generated and pushed back into a vector in these functions are then placed in a totalData vector which is then used to set up and ultimately draw the terrain.

There are a lot of functions that make up the terrain class, but it is abstracted behind function calls that make it rather easy to understand a use in. The fact that it makes use of lua scripting also means it is inherently reusable and customizable, allowing for different heightmaps to be used simply by changing the paths and values in the script.

### **TextureLoader**

The TextureLoader class is a very basic class that only provides one function. This singular function is loadTexture(string path), a function that takes in the path to the texture and loads it.

If the path points to a valid image that can be used as a texture it will be bonded to a texture, and if it doesn’t it will print to the console stating the texture failed to load at the given path. The function returns an unsigned int textureID, which is then pushed back into a vector of texIDs and used to load in the textures. An instance of TextureLoader is used in the terrain to class to load in the textures that are used for multitexturing and detail mapping. The textures taken in by the TextureLoader object in this case depends on what has been defined in the terrain’s lua script. It is also used in the Scene class when initializing the game terrain.

### **Time**

A simple time class that is used to calculate the delta time of the project. It does this by having 2 values, a delta time one and a last Time on. Each frame the engine will call an update, that then updates the deltatime by sending the since the last frame too the update function of Time. This then calculates deltatime via currentTime – lastTime. This value is used quite often, specifically with the camera, and with the movement of objects in the future as well.

### **View**

The view class holds information that is then used for other classes, allowing us to gather it from here instead of getting many individual values. These values will values used for the projection, view and model matrix that is used on the camera, and the camera position, and skybox positions.

### **Debugger**

The debugging class provides a simple and quick way to enable and disable debugging functions within the engine itself. Within the debugging class are functions used to quickly convert objects into simple strings to be printed to the console for debugging. One example of this in action is printing the vec3 position of an object to the console, the debugging class implements a DebugVec3 function that takes in a vec3 object as a parameter and outputs a string that ready to be printed to the console rather than manually having to print each XYZ component of the vec3 to the console each time. The class also provides global settings to enable or disable select types of console outputs for other classes like general debugging outputs or collision debug information.

The debugger class also added functionality to the engine to be able enable or disable select toggles or data on screen. This is done though a debugger window that the user can toggled by pressing the mapped key at runtime. This window can toggle all debugger toggles as well as toggle player and FPS data being displayed in the game view.

### **Entity Manager**

This class is used to store the different game objects that sends messages to each other, specifically used by the enemies in our game. The messages are integer messages, and there are two ways of sending them, either directly to another id, or to the whole of a type, being enemy. This stores the gameobjects that are used for the messages by using an integer, game object map that is then searched through using a vector that holds the keys to the game objects. This can be used to retrieved new game objects, delete old ones and clear it in general.

### **Message Dispatcher**

This class is used to send messages to different game objects. The class uses the entity manager to find the game objects, their ids or type and send them messages that can be called from other game objects. These messages can be handled by them. The messages are a telegram, that has a message id, sender id, and receiver id, being dispatched from the message dispatcher class.

### **SettingsManager**

The settings manager takes care of simple settings tasks within the enegine, such as full screen.

### **Singleton**

A templated class that uses the singleton design pattern to provide one instance of any specified class. One example of the singletons use can be seen the enemyStates header, which contains a singleton of each enemy state, which in turn ensures there is only one instance of each state.

### **Weapon**

Weapon holds the weapon logic for the playes gun, allowing it to fire, shoot and time the weapon shots with the animation being shown.

# **External File Formats**

## **Lua Scripts**

All scripts that are used in the game engine can be found in the following directory: **$(SolutionDir)ICT397-Project-Engine/res/scripts**

### **Script - engine.lua**

Text

Description automatically generated

The engine script takes in the information that is required to setup the window system, and the scene systems. With 3 of the variables being integers, and the other being a string.

The height and width allow the user to input the variables that will define the initial size of the window, in the example you can see we are running with 720 height and 1280 width. It also allows for you to setup the name of the window, which is the name, as an example the script has “ICT397 – Game Engine”. These will define the aspects of the window.

The amount is the amount of scenes in which the engine will create. This also relates to scene scripts, as it will tell the engine how many scene scripts to check for as they define the game objects of a scene.

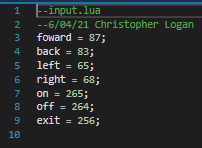
### **Script – gameobjects**

Each game object will have a script that can define more aspects of it. We following the naming convention “typeofgameobject\_name.lua”, such as out “player\_default.lua” script which is used for the player. These scripts have enemy, player, static, water and token ones which are used to populate our world and provide intelligence.

Our static scripts only have the shaders and models used within them. Our token scripts have the shader, model and speed at which they spin within them. The player script will have health, damage, FSM, models, shaders, weapon and bullet models as well, having a lot more there then the other types, which is why we separated them. Our final one is the enemy, which will have health, damage, distance, accuracy, FSM, models and shaders also defined within it.

These game objects are found in “./res/models/scripts/gameobjects/” directory

### **Script - input.lua**



The input script defines the different keys which are used in our game engine by taking in multiple variables for different key inputs. The example above has 7 different key inputs, for movement, exiting and other things such as wireframes. They are each an integer as that is what you set the keys too. The keys are set based of documentation for GLFW, and the keys that relate to those integers.

Forward is defined as 87, which is the code for w, applying that to the forward key. Back is defined as 83, which assigns s to that key. Left is defined as 65, which applies a as the key. Right is defined as 68, which defined d as the key. On, which is for turning the wireframe on, is defined as 265, which is the up arrow, and off which is for turning wireframe off, is defined as 264 for the down arrow key. The final key is for exit, which is 256 which sets up the exiting of the program.

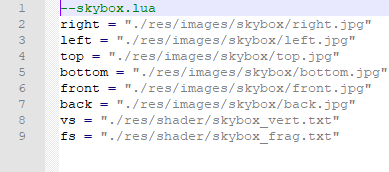
The script is setup this way so that setting up new inputs only requires checking which integer relates to which key on the keyboard.

### **Script - scene.lua**

The scene script defines the spawning of gameobjects in our game world. This is done via exposing a scene pointer for that specific script, and then calling MakeGameObject from it. This function will then create a game object based of the values we enter. The structure goes as follows, game object script from above, scale, position x, y, z and speed. Each of these will be used to determine factors about the new objects being created.

The script loops through the enemies, static objects, tokens and also has 1 water so that these are populating the game world, with 30 enemies, 60 tokens and many static objects.

### **Script - skybox.lua**



The skybox scripts purpose is to take in the path to each of the images that comprise its six faces, as well as the paths to its vertex and fragment shader. All the variables in this script accept strings, with the string being the relative path to the data required.

In the example above you can see that the skybox faces (right, left, top, bottom, front, back) are given paths to the images it will use to texture the skybox, while the paths to its vertex and fragment shaders are provided to the vs and fs variables respectively.

Designing the script in this fashion allows users to be able to create a skybox using different shaders or images if they wish to do so, allowing for simple and efficient customization.

### **Script - terrain.lua**



The terrain script takes in all the information required to create the terrain. Firstly, it takes in a string path to the raw file that contains the heightmap, before taking in an integer value that denotes the size of the terrain, which above is 3072 (the heightmap is 3072x3072).

The xScale, yScale, and zScale variables scale their respective axis by the value provided in the script. In the current terrain script, the y-axis is scaled by 5 while the x-axis and z-axis maintain the default scale of 1.

The tex1, tex2, tex3, and tex4 values take paths to the images that will be used to texture them. These textures make up the multitexturing that will be easily seen when launching the ZOOM demo. The detail Map variable also takes a path to an image, which will serve as the detail Map that is textured to the entire terrain.

Lastly, the vs and fs variables take in the paths to the terrain’s vertex and fragment shader.

Providing all this information to the engine through scripts allows for terrain to be quickly and easily adjusted and customized. If the user wanted, they could provide their own textures, detail Map, shaders and even heightmap by providing paths to them and adjusting the size and scale of terrain to fit.

## **Other Resources**

Alongside the Lua scripts, all the other resources used in the game engine are in the following directory in a dedicated file: **$(SolutionDir)ICT397-Project-Engine/res**

While resources could technically be placed anywhere, we deemed it important to place our resources properly and efficiently in an easy to understand file structure, which in this case is the “res” (resources) folder.

### **Models**

All the models that are usable in the engine are in the file titled “models” in the resource’s directory. Models are then separated into various folders based on their type. The “environment” folder as its name suggests contains models for all the models that will decorate the environment, such as trees and stones. The “group” folder contains some miscellaneous photos used for the group exit photo. The “tokens” folder contains the models that are used as the power-up tokens. The “weapons” folder contains the data required to make the player’s shotgun, and lastly the “zarlag” folder contains the data that makes the enemy model.

When making game objects to place into the game scene via the scene.lua script, providing the script with the relative path to the model you want to use on that game object will read that model in.

**For example:** “./res/models/tokens/Free\_Hit.fbx” when provided to the scene.lua script as the “modelfolder” variable in the MakeGameObject call will draw the Free Hit token model. The models are mainly loaded in using the gameobjects scripts foun

### **Images (Textures)**

Images that serve as textures for use in the engine are in the file titled “images” in the resource’s directory. Images are separated into two different files, with “skybox” featuring the images required to texture a skybox, and “terrain” featuring the images required to texture terrain.

Both the terrain and skybox have their own lua scripts (terrain.lua and skybox.lua) which are provided with the paths to these images to use them as textures.

There are also various images in the images folder that serve as assets for the UI menus.

**For example:** “./res/images/terrain/detailMap.jpg” will provide the terrain.lua script access to the detail map image. Other images are used in the game object scripts and also by the md2 and fbx files.

### **Shaders**

All shaders that are used in engine are available in the “shader” folder in the resources directory. Currently, there are vertex and fragment shaders for the models, skybox, and terrain respectively, which are all used to specify how the GPU goes about rendering them.

Various lua scripts, such as terrain.lua, skybox.lua, and take in the paths to these vertex and fragment shaders, allowing them to be used.

**For example:** “.res/shader/skybox/skybox\_vert.txt” and “res/shader/skybox/skybox\_frag.txt” can be assigned to the vertex shader and fragment shader variables in skybox.lua. they are also defined in the game object scripts for our different game objects.

### **Heightmaps**

All heightmaps that are available for use in the engine (of which there are a couple to try) can be found in the “heightmaps” folder in the resources directory. The heightmap present in the folder named “1k.tga” serves as the heightmap that the terrain is generated from.

In future, if more heightmaps are generated, they will be placed in the folder for use. Specifying the heightmap for use is done in the terrain lua script, with the path of the heightmap used to find the heightmap file and generate the terrain.

For example: "./res/heightmaps/1kheightmap-3.raw" when provided to the heightmap variable in terrain.lua will result in that heightmap being used to generate the terrain.