

**Software Design Document for:**

# **OOber Taxi**

**Revolutionising the taxi game!**

“We’re not trying to reinvent the wheel with this one”™

All work Copyright ©2018 by Carré

Written by Jack Matters, Cordell Smith

Version # 2.00

Thursday, May 31, 2018

Table of Contents

[OOber Taxi 1](#_Toc515620432)

[Design History 4](#_Toc515620433)

[Version 1.00 4](#_Toc515620434)

[Version 2.00 4](#_Toc515620435)

[Game Engine Architecture Overview 5](#_Toc515620436)

[General Overview 5](#_Toc515620437)

[Philosophy 5](#_Toc515620438)

[StarUML 5](#_Toc515620439)

[Third Party API 5](#_Toc515620440)

[OpenGL 5](#_Toc515620441)

[Bullet Physics 5](#_Toc515620442)

[Assimp 5](#_Toc515620443)

[Lua 5](#_Toc515620444)

[Created Classes 5](#_Toc515620445)

[Game Control Engine 5](#_Toc515620446)

[Game Asset Factory 6](#_Toc515620447)

[Game World 6](#_Toc515620448)

[Input Manager 6](#_Toc515620449)

[Physics Engine 7](#_Toc515620450)

[Texture Manager 7](#_Toc515620451)

[Script Manager 7](#_Toc515620452)

[AI Engine 8](#_Toc515620453)

[External File Formats 9](#_Toc515620454)

[Scripts 9](#_Toc515620455)

[Script Location 9](#_Toc515620456)

[Script Type 1 - CameraInit 9](#_Toc515620457)

[Script Type 2 - ModelsInit 9](#_Toc515620458)

[Script Type 3 - TerrainsInit 10](#_Toc515620459)

[Script Type 4 - TexturesInit 10](#_Toc515620460)

[Script Type 5 - WindowsInit 11](#_Toc515620461)

[Resources 11](#_Toc515620462)

[Game objects 11](#_Toc515620463)

[Textures 11](#_Toc515620464)

[Terrains 11](#_Toc515620465)

[Shaders 11](#_Toc515620466)

[“StarUML Appendix” 12](#_Toc515620467)

# **Design History**

During the design process of our game engine, the ideas that we intend to bring into fruition will be documented with the intent to allow us to examine the success and failures of our project. Beginning with version 1.00, we will be adding smaller 0.10 increments or even 0.01 indicating minor changes pertinent to the project. When a significant milestone is reached, this will be set during the project planning stage, it will be indicated by intervals of 1.00.

## **Version 1.00**

Version 1.00 is the initial start on the game software design document. An example game document has been used as the foundation for this one, mainly copying the layout for use with this document.

## **Version 2.00**

Version 2.00 is the final release of the game software design document. It contains the details of all the currently implemented functionality as well as the architecture and software design methods used.

# **Game Engine Architecture Overview**

## **General Overview**

### **Philosophy**

Our philosophy when designing the Carre game engine was to apply clean, well structured software patterns that was easy to interpret, whilst still maintaining the principles of low coupling and high cohesion.

### 

### **StarUML**

Please refer to appendix ‘StarUML Appendix’

## **Third Party API**

### **OpenGL**

OpenGL is our main 3D rendering API. More specifically modern openGL using shaders. The API is behind a facade class that allows for specific openGL functions to be called behind the scenes, reducing code clutter.

### 

### **Bullet Physics**

Bullet physics is currently not behind a facade. It is coded into the Physics Engine, making it difficult to swap out for another API. In the next assignment, we will aim to facade it in a way that swapping APIs is relatively easy and would require minimal alteration.

### **Assimp**

Assimp import library is the main library used to read in many object types. Assimp has the capability to identify many different file types, read the information and allow it to be accessed and stored for further use.

### **Lua**

Lua scripts can manage a lot of the functionality within your engine without having to recompile the entire project. All game logic and configuration files should be exported to scripts and compiled at run time. This also allows for ease of update by non-programmers. Lua scripts are loaded through calling the script manager class, all scripts must be placed in the script resource folder (path found below). A script may be loaded by calling the LoadScript() function and passing in the name of the lua script that you would like to load. This has not been fully implemented for this assignment but a working framework is attached for demonstration.

## **Created Classes**

### **Game Control Engine**

The game control engine is the main hub to which all the components of the game engine communicate with and satisfies the controller component of the MVC design pattern. Within this class, all of the engines assets are initialised within an Initialize() function. Anything you wish to be loaded prior to the creation of a new game world should be loaded in this function first for future access and possible game changes. When all the game engine components have been initialized, the main game loop is started and called using the GameLoop() function. This is where any changes made as a result of user input or game logic such as game states, objects, animation, physics or sound and should be passed into the m\_gameWorld object to be updated and rendered for the user to see. When the user exits the application by closing the window or pressing ‘esc’, the main game loop will end and then code management will proceed to take place calling the Destroy() function. Anything that is created by all of the game components are destroyed and deleted in this function to free up any memory that has been used.

### **Game Asset Factory**

The GameAssetFactory class utilises the factory method design pattern in order to allow the separation between object creation and type. Game assets such as terrains, objects and NPCs can be created and loaded in through the CreateAsset() function that takes the type of asset you wish to create as a first parameter and the file path where the data is contained as the second parameter. The Terrain, NPC and Object class inherit from the parent class IGameObject. The type you specify in the the create asset function can be of type OBJ\_TERRRAIN, OBJ\_NPC and OBJ\_OBJECT and using a switch case, will return a newly created object of that type containing the loaded data.

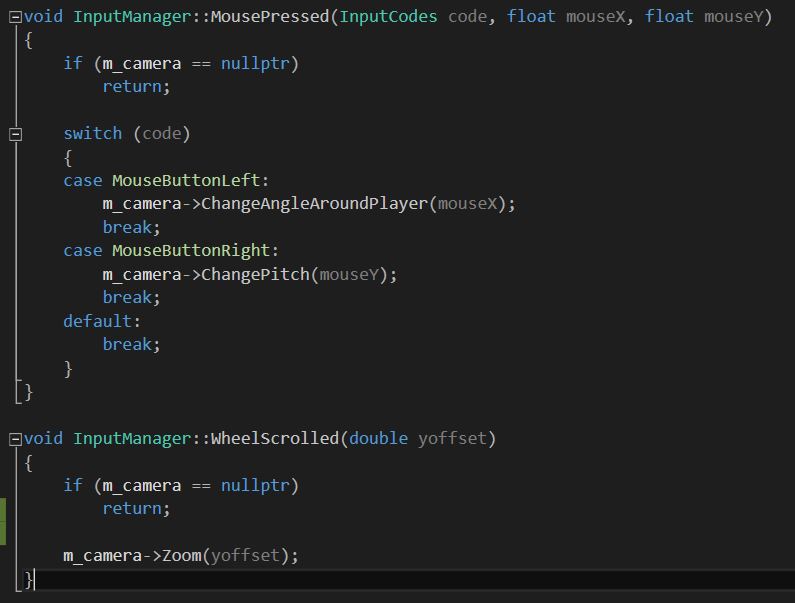
### **Game World**

The game world class represents all of the data coming together to create an instance of a game world. This includes terrain, objects, characters and any other assets loaded to create a scene. This is the View part of the MVC pattern and is where all the data is displayed visually.

### **Input Manager**

The input manager class contains various functionality that allows for the player to interact with the View component of the MVC pattern using the keyboard and mouse. Within the InputManager class, there is a struct containing key codes that are relative to the integer value that GLFW uses to read each individual key on the keyboard as well as mouse buttons and movement. The class handles this data during the main game loop, constantly polling for input events to take place. The KeyPressed(Keycode code) determines what functions are called by passing the code into a switch statement. This is where the player movement occurs when the ‘WASD’ keys are pressed. Each key represents a switch case and runs the appropriate function to move the player object in the world. The mouse movements are also tracked and using their own set of functions.





### 

### **Physics Engine**

The Physics Engine is responsible for all physics related simulations, including collision detection. All game object data is passed into this class via the GameWorld class in the Initialize() function in the form of a data structure. Also during this step, ActivateAllObjects() is called to ensure that every rigid body object is active throughout the entire program. During the main game loop, Simulate() is called from the Update() function in the GameWorld class. Simulate() performs all physics calculations during a step simulation. After the simulation, the updated game object rigid body locations are passed back to the GameWorld class to be drawn. This process happens every step until the program is ended.

### **Texture Manager**

The texture manager class is responsible for the loading and storing of all textures. It uses a singleton design pattern so it can be accessed anywhere in the game engine, and so there is only ever one instance of a Texture Manager. All textures are to be stored in the TexturesInit.lua file for loading. When a texture is loaded in, the Texture Manager first checks to see if it has previously been loaded in. If it has, it will return the assigned textureID. If not, it will proceed to load in the texture, assign it a texture ID, and add it to the unordered map of loaded textures. The relative file path is used as the key, and the texture ID is the value. When searching for a loaded texture, the Texture Manager first checks the map. If it is found, it will return the texture ID of the required texture. If it hasn’t been previously loaded, it will load it in, add it to the map, then return the textureID. This setup ensures that a texture is only ever loaded once.

### **Script Manager**

The script manager class is responsible for the loading of all scripts, and for the storing of all data they contain. It uses a singleton design pattern so it can be accessed anywhere in the game engine, and so there is only ever one instance of a Script Manager. The design of the script manager isn’t exactly a good one, but with the amount of work to do and the time restraints, we opted to go for something that works over something that works and is efficient and of good design. The Script Manager contains five different functions for the loading of the five different lua scripts. The different functions are LoadWindowInitLua(), LoadCamInitLua(), LoadTexturesInitLua(), LoadModelsInitLua(), and LoadHeightmapsInitLua(). The first function reads in all data related to window initialization such as screen width, height, and title. The second function reads in all data related to camera initialization such as the xyz position, FOV, and near and far planes. The second function loads in all textures, and calls the Texture Manager singleton for each texture read from the script. The fourth function loads all model data into a data structure which is then used in the GameControlEngine Initialize() function to initialize all model locations, collision bodies, and other data. The fifth function is very similar to the fourth, except that it holds slightly different data. All data is loaded from script earlier in the Initialization process.

### **AI Engine**

The AI Engine is responsible for the handling and logic of all AI. The AI Engine is split up into four classes. The State class is a virtual template class that defines the three functions that each state will go through; Enter(), Execute(), and Exit(). The Enter() function is where all initialisations for a state happen, the Execute() function is the main loop of a state, and the Exit() function is the cleanup when exiting a state. The StateMachine class is a template that handles the transitioning between the different states, and keeps track of which state an AI is currently in, the previous state it was in, and other useful features. It is also responsible for running a state using the Update() function. The AllStatesFSM class is where all the code is done for the aforementioned Enter(), Execute(), and Exit() functions for every state within the FSM. It contains a different class for each different state, all stored within the same file for easy management. The last class is the ComputerAI class which handles all the data and logic to do with the AI controlled object. It keeps track of data such as current position, current velocity, health, and other useful and important information. The ModelsInit script has a parameter that determines if a model is given AI or not. If a model is to be given AI, it is done so in the GameControlEngine class when all model data is initialized.

### 

# 

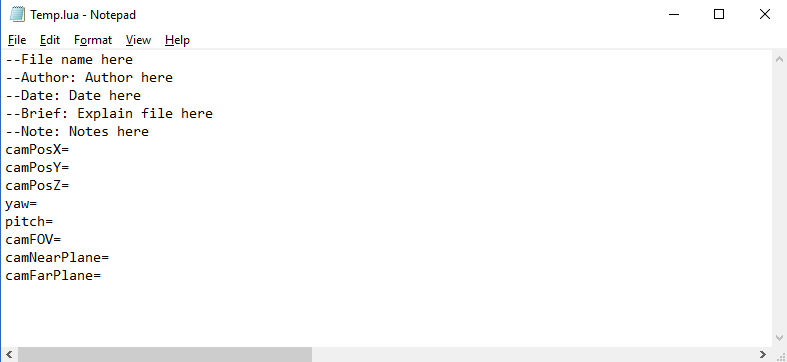
# **External File Formats**

## **Scripts**

### **Script Location**

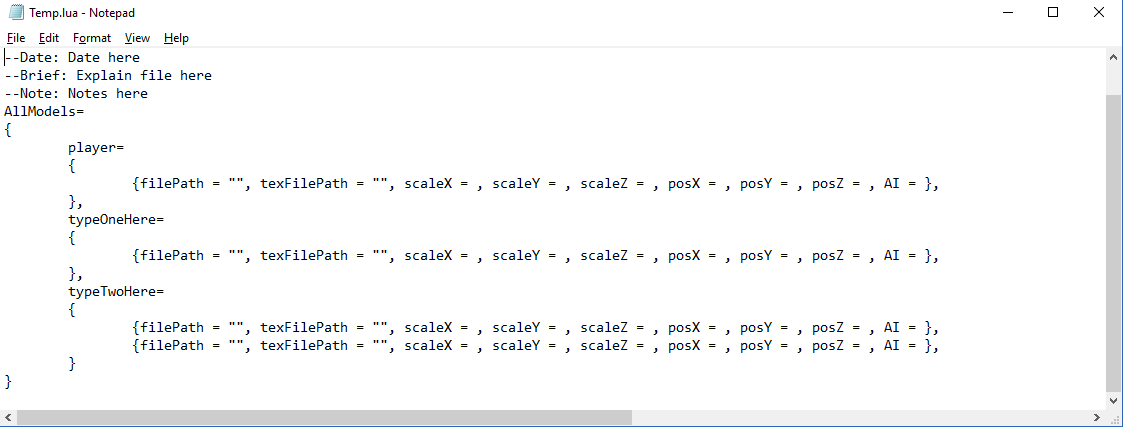
All scripts are to be placed in “$(SolutionDir)CarreGameEngine\res\scripts”. All script files are to be placed here, without any subfolders. Recommend naming script files something meaningful. Any test following ‘--’ will be counted as a comment, and won’t be read by the Script Manager.

### **Script Type 1 - CameraInit**



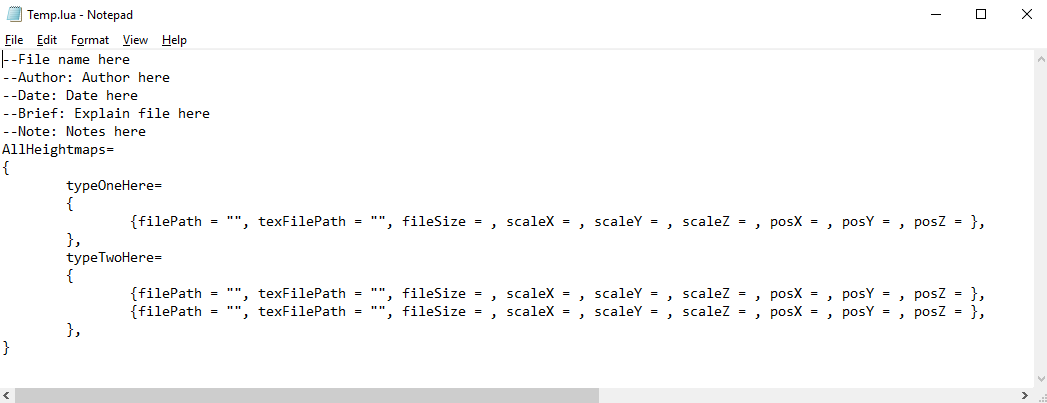
All parameters take integer float value, and they should be placed directly after the equals sign (camFOV=0.0). Any other data entered in this script will not be read

### **Script Type 2 - ModelsInit**



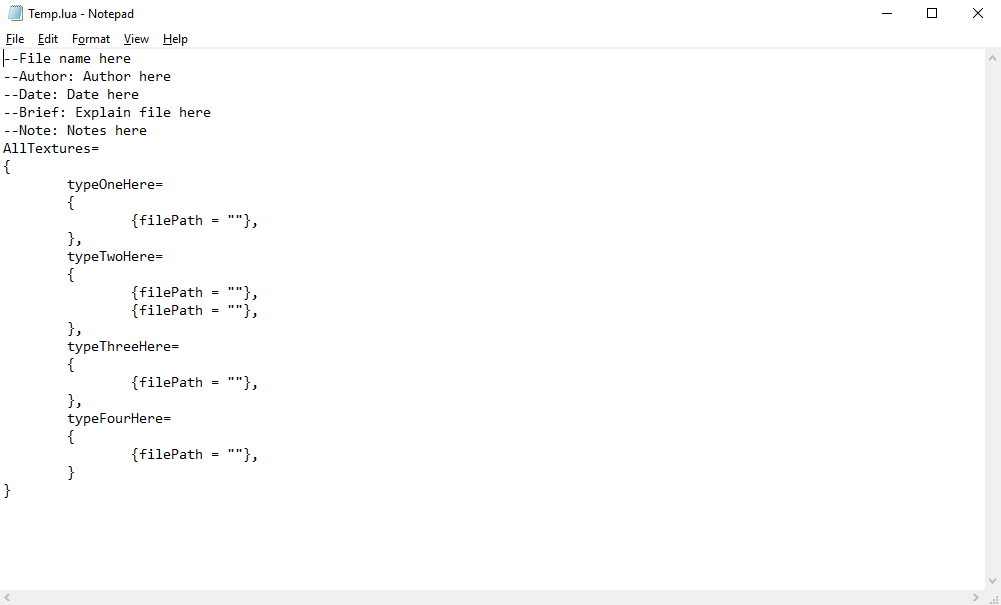
The player parameter **must** be present and **must** be spelt as it is in the image. That parameter is searched for specifically to identify the player model (if there is any). The typeOneHere and other similarly named placeholders should be replaced with the name of the model that the following data is related to. Some models might be repeated, so all similar models should be placed together, like how typeTwoHere has multiple data entries. The correct file paths, relative to the CarreGameEngine.vcxproj file, should be placed between the quotation marks. The AI parameter should be followed by either true or false (AI = false). All other parameters take float values and should be placed with a space after the equals sign (scaleX = 5.0). Incorrect file paths will result in a texture not being loaded. Any other data entered in this script will not be read.

### **Script Type 3 - TerrainsInit**



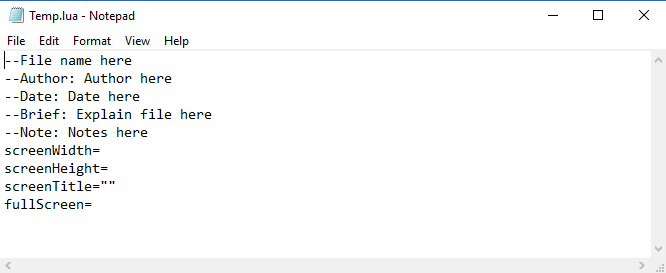
The typeOneHere and other similarly named placeholders should be replaced with the name of the terrain that the following data is related to. Some heightmaps might be repeated, so all similar heightmaps should be placed together, like how typeTwoHere has multiple data entries. The correct file paths, relative to the CarreGameEngine.vcxproj file, should be placed between the quotation marks. All other parameters take float values and should be placed with a space after the equals sign (scaleX = 5.0). Incorrect file paths will result in a texture not being loaded. Any other data entered in this script will not be read.

### **Script Type 4 - TexturesInit**



The typeOneHere and other similarly named placeholders should be replaced with the name of the game object that the following textures are related to. A object (.obj) file will most likely have several textures related to it, so all should be placed together, like how typeTwoHere has multiple file paths. The correct file path, relative to the CarreGameEngine.vcxproj file, should be placed between the quotation marks. Incorrect file paths will result in a texture not being loaded. Any other data entered in this script will not be read.

### **Script Type 5 - WindowsInit**



The screenWidth and screenHeight parameters take integer values, and they should be placed directly after the equals sign (screenWidth=1920). The title of the screen should be placed between the quotations for screenTitle. The fullScreen parameter should be followed by either ‘true’ or ‘false’ (fullScreen=false). Any other data entered in this script will not be read.

## **Resources**

### **Game objects**

All game objects are to be placed in “$(SolutionDir)CarreGameEngine\res\objects”. All related objects are to be placed in their own subfolder, the name of which is at the user’s discretion. All other objects are not to be placed in a subfolder. Recommend naming game objects something meaningful.

### **Textures**

All textures are to be placed in the same location as the game objects or terrains that use them. Their relative file path is placed in the TexturesInit.lua file for reading in. Recommend naming the texture files something meaningful.

### **Terrains**

All terrains are to be placed in “$(SolutionDir)CarreGameEngine\res\terrain”. All terrain files are to be placed here, without any subfolders. Recommend naming terrain files something meaningful.

### **Shaders**

All shaders are to placed in “$(SolutionDir)CarreGameEngine\res\shaders”. All shader files are to be placed here, without any subfolders. Recommend naming shader files something meaningful.

# 

# **“StarUML Appendix”**

This appendix provides a StarUML overview of our Game engine.

