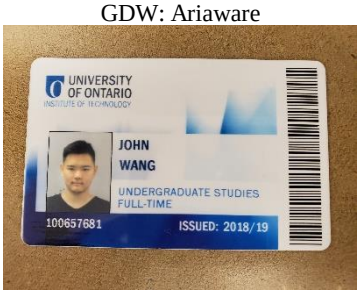
INFR 3110U: Group Assignment 1

Boris Au
ID: 100660279
GDW: Ariaware

John Wang
ID: 100657681
GDW: Ariaware



Victor Zhang
ID: 100421055
GDW: Ariaware

TABLE OF CONTENTS

Work Distribution:

- *Boris*
    - *Graphical Assets*
    - *GUI*
    - *DLL Creation*
    - *Report*
- *Victor*
    - *Saving & Loading DLL Implementation*
    - *Singleton Implementation*
    - *Report*
- *John*
    - *Game Mechanics*
    - *Command Implementation*
    - *Factory Implementation*
    - *Report*

***Abstract*— This document outlines how to use our level editor along with describing its features. It also explains the implementation of the different design patterns used in order to accomplish the assignment task.**

I. Introduction

Our level editor allows all members of our GDW team to place objects anywhere in the scene and save the data into a text file. The editor is designed in a intuitive way, such as the use of traditional keyboard shortcuts as well as mouse click and drag functionality.

A. How to use:

1) To place objects, the user can click on the desired item in the bottom toolbar that they want to place and click anywhere in the scene to place it. Holding SHIFT button while clicking the scene with an equipped objects allows the user to keep placing multiple instances of the object within the scene.

2) To delete items, the user can click on any object and press the DELETE button key or in the bottom right toolbar. Deleting multiple objects is possible as long as all objects are selected by either holding SHIFT button and clicking or using click and drag. The undo button or CTRL+Z undos you last action and the redo button or CTRL+Y, redos any previous undos.

3) Pressing the save button or 'O' button will save the objects positions into the save text file and the load button or 'I' button will load the previously saved map from the same text file.

B. Controls

1) Users can move the player around the scene using WASD buttons in order to test collisions or to visualize scale.

2) The scroll wheel enables zoom functionality and holding down the mouse wheel while dragging pans the camera around the scene. Moving the mouse cursor toward the edges also move the camera in the desired direction.

3) The camera can also be independently controlled by the arrow keys.

II. Use of Program

To meet the objective, we have implemented functions to interact with the objects of the program through interfacing with the UI, or through hotkeys set up according to typical RTS-style game functions. This program has the functions to:

A. Save and Load

1) The 'I' key is a shortcut for loading

2) The 'O' key is a shortcut for saving

B. Place 4 objects

1) 1-4 can be used as hotkeys for placing objects

2) The 'Shift' key can be held for persistant object placement

C. Select one or more object(s)

1) A Selection box can be used by dragging the mouse

2) 'Ctrl-LMB' and 'Shift-LMB' can be used for multiple-selections

3) Clicking off an object will deselect it

D. Delete and upgrade selected objects

1) The 'Delete' key can be used to delete objects

2) The 'G' key is a shortcut for upgrading

3) The player cannot be deleted

4) Objects can only be upgraded 5 times

5) 'Shift + Delete' deletes all objects (except the player), and clears the undo and redo stacks

E. Undo and Redo all actions, except saving and loading

1) 'Ctrl+Z' is a shortcut for Undo

2) 'Ctrl+Y' is a shortcut for Redo

F. Move a player character

1) The movement of the player character is controled with 'WASD' keys

G. Control a RTS-Style Camera

1) The 'Q' and 'E' keys, and the mouse wheel scrolls controls zooming

2) The Keyboard's Arrowkeys, the mouse's scroll wheel, moving the mouse to the edge of a screen moves the camera along the horizontal plane.

3) Double clicking an object will allow the camera to attempt to center and follow that object. This can be undone by clicking off the object.

H. Game Pausing

1) The 'P' key is a shortcut for pausing

III. Design Patterns

A. Command Design Pattern

The command design pattern is implemented in the placement, deletion and upgrading of objects. With the exception of deleting and placing the player, all objects are able to do and undo any action requested of them. Furthermore, each action invoked will be stored in the undo stack, and each undone action will be stored in the redo stack, able to redone. This is achieved through the command design pattern. It is able to do this by making sure that all commands have similar structure, with a command interface called ICommand which contains an execute command, an unexecute command, and a cleanup command for when any references to the command is lost. Each command is invoked externally, and takes in a GameObject. The placement and deletion commands function very simply. It only disables or re-enables the GameObject when executed or unexecuted without deleting anything, so that the GameObject's parameters remain accessible. It is only when the command is being cleaned up, and the object is in a deleted phase, will it be deleted, as at that point, it will be unnecessary to re-access the parameters of the object. To upgrade the object, the command simply makes the scale of the object bigger and updates the Y position to be aligned to the ground. It simply makes the object smaller, then updates the Y position when a unexecute command is invoked. The functions that call the commands are hooked to UI Button selections.

B. Factory Design Pattern

The factory design pattern is responsible for creating the GameObjects able to be placed in the scene. By using this, we are able to encapsulate the creation process, without needing

to worry about any of the creation initialization procedure in our main code and let us easily place more options for different object variants to be created easily. Furthermore, it lets us keep track of the total amount of objects that were created within the game, and allow us to assign a unique id to each object created, without worrying about duplicate ids from undoing deleted objects. This works by using an abstract class to define the structure for a building and determine the id, then creating an inherited class that takes in a position to instantiate a prefab of a predefined object. Differing based on the inherited class, and assigning it a position in the game world based on the position that was passed into it. It will also inform the object of the object's id. Finally, once it's done with all that, it will return the object as a GameObject to the invoker of the factory, allowing it to do whatever it might need to do. Though this creates the prefab, it is not responsible for placing the prefab into the screen. In the CommandManager script when a request for a new object is received, a switch statement will be activated, receiving an enumeration (enum) to determine which object needs to be taken from the factory. With this enum type, combined with the current mouse ray-casted position in the game world, the switch statement will request the factory pattern to create a return an object of the given parameters. To create a new object variant, the user will simply define the constructor for a new building inheriting from the Building class by swapping the instantiated prefab reference, add a new enum to the list, and add a case for the enum in the switch statement in the command pattern class.

C. *Singleton Design Pattern (DLL)*

1) *Use in C++*

In the C++ DLL, Singleton is used for the MapLoader class/dll to create a static instance of itself because there won't be a need for multiple instances of the class. Since the class functions can be called many times, there is no need to create instances for each time that the functions are called.

2) *Use in C#:*

In the Unity Project, the Singleton Pattern is used multiple times for the CommandPattern class (which should be renamed to GameManager), and the SelectionManager class. It's included in these 2 classes and none others because these two classes must only have a single instance, or the game will break. Furthermore, it's because these two classes must be referenced by each other and many other classes a lot, meaning that it would save the creation of multiple connections if they were to exist in statically defined memory. The implementation of this pattern in an object inheriting from is simply to delete any unity editor created versions of the class itself, and create a new, static instance of itself that can be accessed through an Instance Get function, but not set. Since monobehaviour constructors already cannot be called, there's no point in re-making it private. The code to achieve this was taken from online [4].

D. *Game Loop Pattern*

Unity runs with a default game loop implemented within the system, it is easy to tell that this is happening because in every C# script that is created in unity, they include a default Start() and Update() functions. The Start() function is for the developer to initialize anything that is needed before the running of the Game Loop while Update() function is for any events happening inside the Game Loop. The Game loop is running at 1 loop per frame where Update() gets called in every loop. From the update, we can introduce checkers and function calls to access other functionality such as adding an if statement for keyboard button presses which then goes off to trigger the respective functions for that specific frame/game loop iteration. The best benefit of this is that the rest of the game is still running without any delays or waiting for any other functions until we active one of the triggers and more functions are called. Taking this to a higher level if the trigger event is very heavy and takes too long, it can affect the loop runtime longer and decrease the frame rate causing lag. We can then take that long function and run it on a different processing thread while keeping the same frame rate for the game loop.

IV. UML For Factory Pattern

The Pattern we are showing is the Factory pattern, in a UML Diagram. [5]

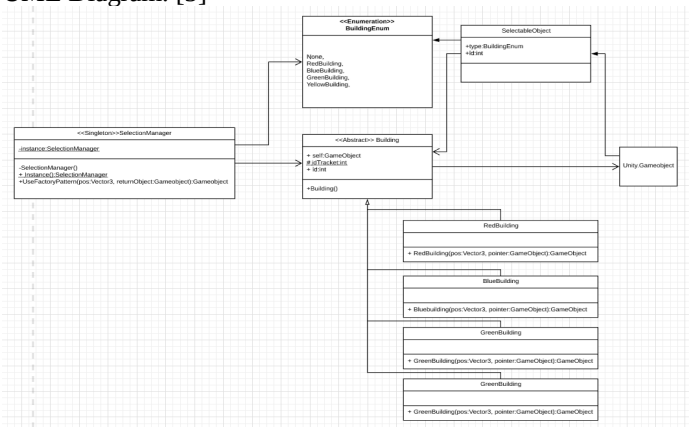


Fig. 1. UML Diagram **https://prnt.sc/pf2pot**

V. Implementation

A. *Saving and loading the level*

Both saving and loading the map in our editor involves interfacing with a text file. We created a DLL using C++ called MapLoader.dll to facilitate the saving and loading of data. Our map data text file holds 1 int and 3 floats separated by a ',' in each line for each object. The first element specifies the object type and the subsequent three values contain the X, Y, and Z global positions. Upon activating the saving function, the program will loop through all the active Objects in the AllObjects List and send the type and position data into the text file using the C++ DLL one by one. Upon calling the load function, both command lists are cleared and the scene is cleared of all additional GameObjects. The saved file is read fully and loaded into vectors inside the MapLoader Class then each Saved GameObject is created and added to AllObjects List while getting the type and position data from the MapLoader class using the DLL.

B. *GameObject Selection and Manipulation Process*

To manipulate a GameObject, it must be selected. To do this, there are 2 object lists that we need to keep track of. The Selected Objects list and the Active Objects list. All objects visible in the scene are in the active objects list. If they are in this list, they will be selectable in a multitude of ways. The Selected Objects list is a sublist of the Active Objects list. If they are selected, any command invocations will be applied to them. This works by moving them to the selected objects list and simply using a foreach loop to apply the commands for all

objects in the Selected Objects list. If an object is deleted, it will automatically be removed from the Active Objects list, and thus the Selected Objects list as well. When a command is executed for a group of Selected Objects, individual commands will be issued for each object within the group. This means that if the user were to delete a list of objects, but only want to undo the deletion of a few, they will not need to worry about undoing all the deletions, if they were to press the undo button, as undo invocations are separated per object. To separate the selectable objects from non-selectable objects, and to define functionality for all selectable objects, the SelectableObject class is placed on all objects that can be selectable. It will hold information about the object including its type, whether it's deletable, it's level and it's id. It also responds to being selected by activating a halo behaviour around any selectable object, and disables this behaviour when the object is deselected.

C. Player

The player character is a modified selectable object. It is given a rigidbody, movement script, and it is excused from being deleted or placed.

VI. Graphic Assets

The editor allows the placement of four different objects.

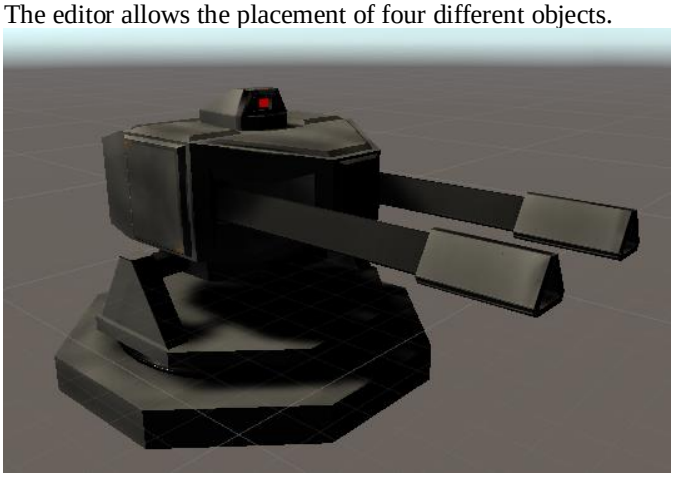


Fig. 2. Turret Model

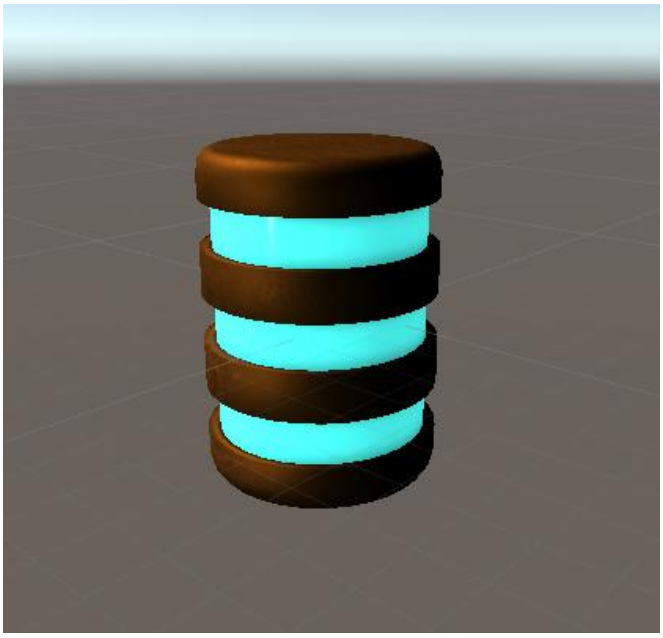


Fig. 3. Battery Model

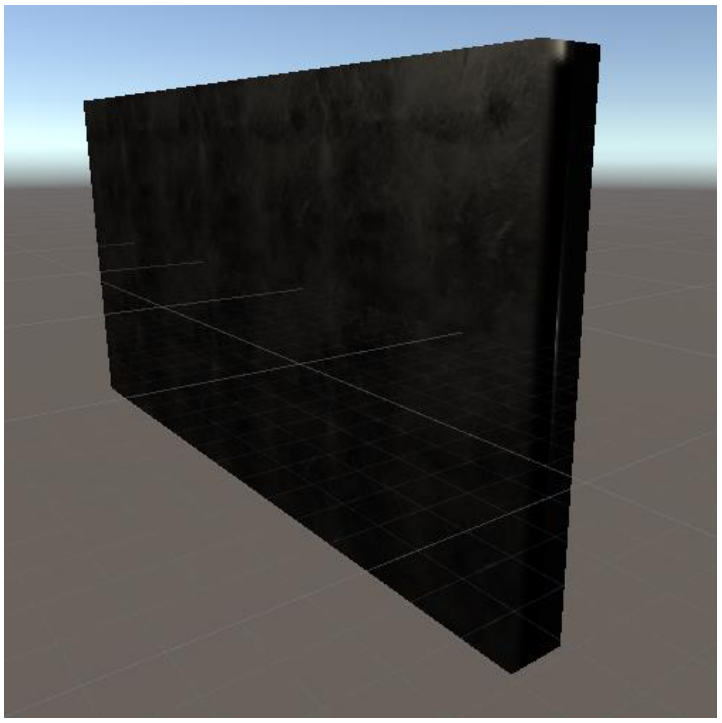


Fig. 4. Vertical Wall Model

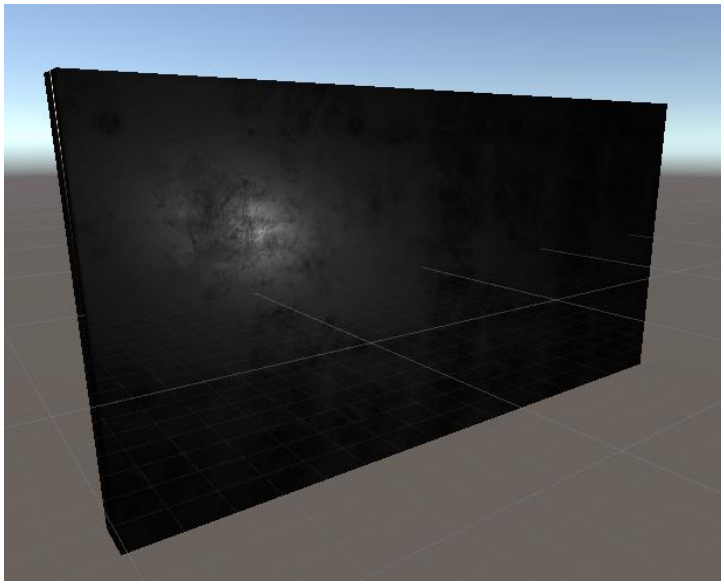


Fig. 5. Horizontal Wall Model

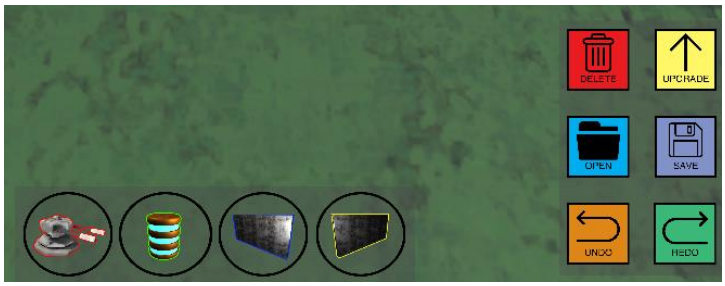


Fig. 6. Graphic User Interface (GUI)

Fig. 7. The Player's Character Model (From Unity Asset Store) [3]

VII. External Code

This project has within it, some code written by people not in the group. The instances where external code is used is as followed:

1) The camera used in the code has code taken from an Asset in the Unity Asset Store [1]. *The code has been modified and adjusted for this project.*

2) The GUI display code for the selection box is taken from a youtube tutorial [2]

VIII. Screenshots and References

[1] **https://assetstore.unity.com/packages/tools/camera/rts-camera-43321**

[2] **https://www.youtube.com/watch?v=iN24fuZEF_k&list=PLREj8Ib34tkYX_3ROT50Q_079sF5AqId9&index=18**

[3] **https://assetstore.unity.com/packages/3d/characters/robots/space-robot-kyle-4696**

[4] **https://gamedev.stackexchange.com/questions/116009/in-unity-how-do-i-correctly-implement-the-singleton-pattern**

[5] UML Diagram for Factory Pattern https://prnt.sc/pf2pot