GAME DESIGN DOCUMENT

PROJECT SHOOTER

AUTHOR: IZZUDDIN AHSANUJUNDA

Project Shooter is a simple shooting range game made using Unity3d 4.1.2f. Player need to shoot at enemies to accumulate points as much as possible. In order to provide obstacles, enemies that could fire back at player are spawned at random.

All scripting are written in C# and constructed in object-oriented programming paradigm. Making every function, logics, and AIs reusable to any other objects that are in need of similar working logic, or even another project all together.

GAME STRUCTURE

CONTROL

The user navigate a player object using RIGHT and LEFT arrow key to move the player horizontally. The player object will move accordingly.
In the desired position, user could make the player shoot a bullet using UP arrow key.
Beware that some enemy could retaliate and launch a counter attack by shooting bullets to player's position.

SCORING

By default, all enemies worth 100 points each. But an attacking enemy worth 50 points bonus.
Try to avoid any incoming attacks from the attacking enemies by moving the player to avoid enemies bullets, or commit a retaliating action by shooting at the bullets. Successful attack on enemy bullet would provide a better score than shooting at any enemy.

GAME ELEMENTS

The Player

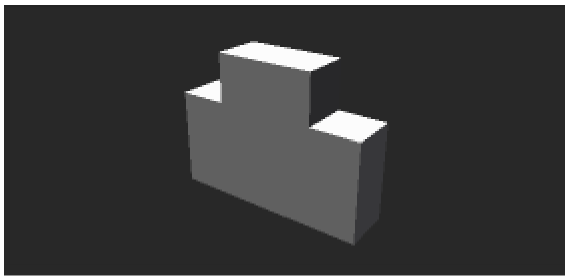


Figure 1 The Player

This is the object that represents the user. It waits for users input to move. It has several scripts attached to make it listens to user's input, calls the bullet objects to be fired, and to call debris objects when collided with enemy bullets to simulate an explosion.

The Enemy

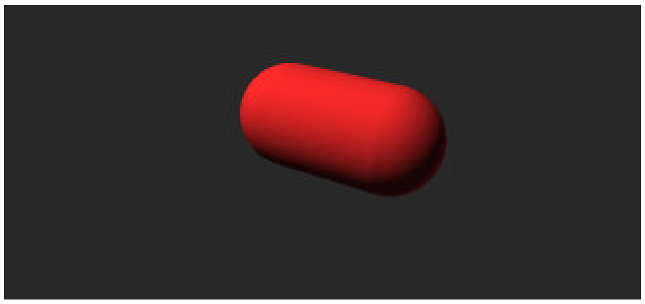


Figure 2 The Enemy

This is the enemy objects. It is spawned at random by the game. By default, this is the most common type of enemy spawned by the system. However, the probability of which type of enemy should be spawned could be predetermined easily.

The Attacking Enemy

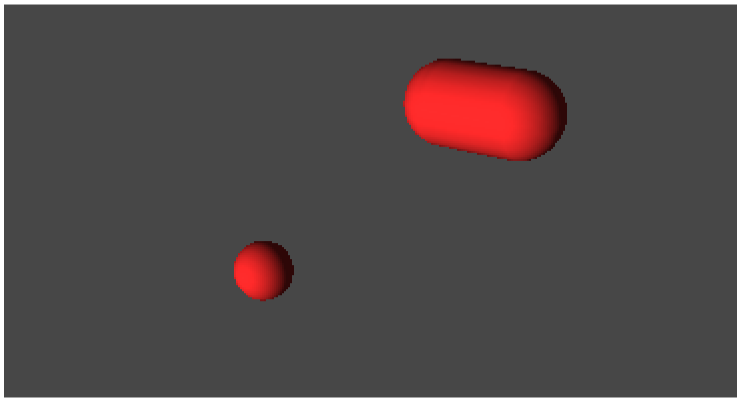


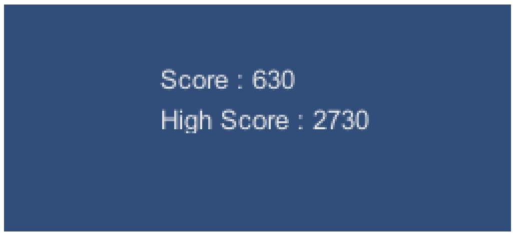
Figure 3 The Attacking Enemy Right After Launching it's Bullets

The attacking enemy is basically just like the enemy, but it's overridden to have an extra function to capture the player's position at certain time, and launches the bullet at the captured position. Both the Attacking Enemy Object and the Bullet Object could be destroyed if fired upon. Both worth more points than the enemy.

The Enemy Spawner

The enemy spawner is an empty Game Object in Unity3d, which has a spawner script attached to it. From it's position, the enemy spawner could spawn random enemy with predetermined spawn ratio, direction, speed, and types.

The Scoring System



```
Score : 630
High Score : 2730
```

Figure 4 Scoring Label

The scoring system is a GUI label which is attached to a script that keeping tracks to player's in-game scoring.

GAME SYSTEM

Class Diagram

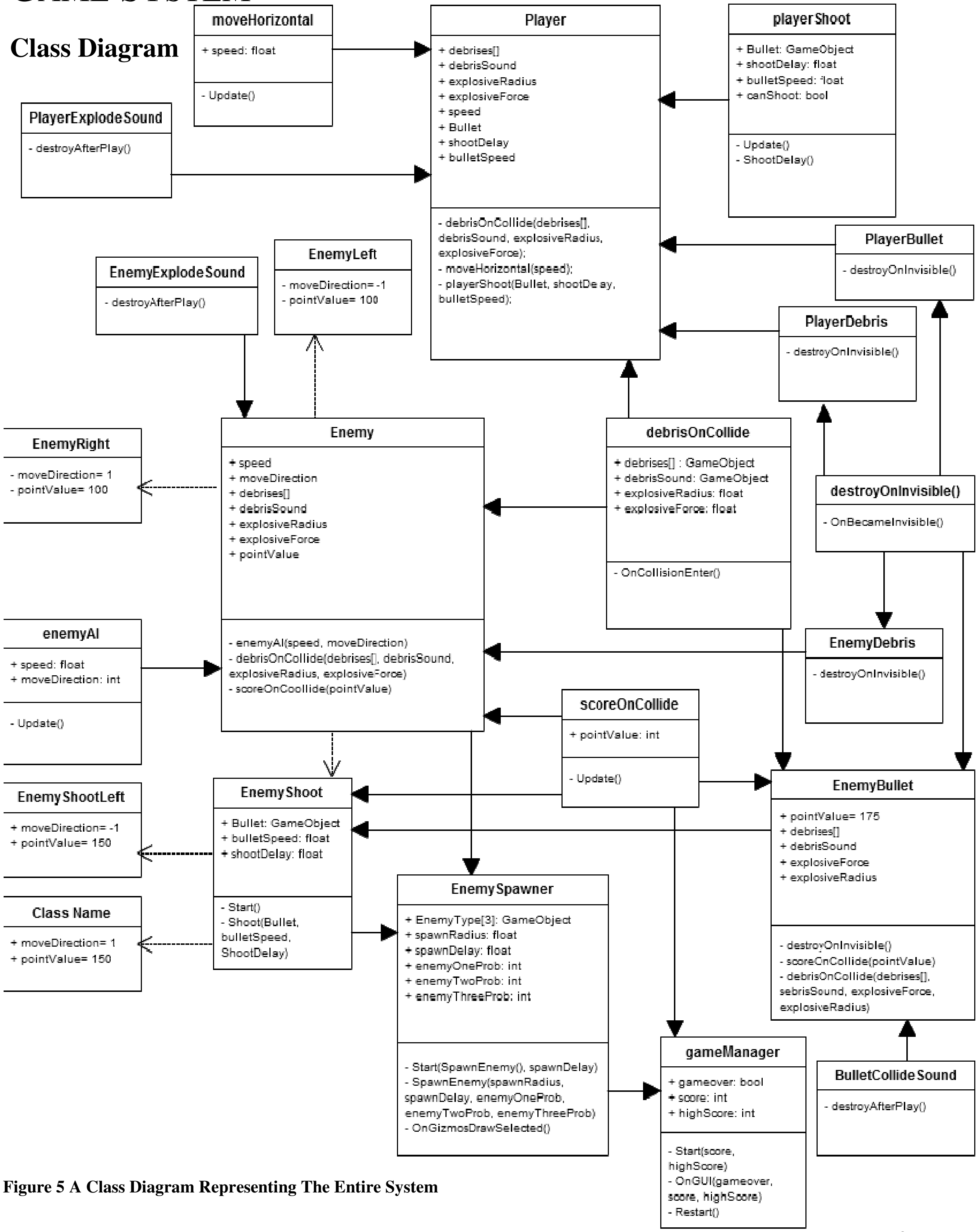


Figure 5 A Class Diagram Representing The Entire System

Figure 5 above shows how the game constructed in a system layer. As mentioned, this game is constructed in object-oriented programming paradigm, and this diagram shows how exactly it is done.

Let's focus on `Player` class to provide better explanation.

This class represents the player object in the game. Most of it's functions or methods are constructed as part of other classes, and then this `Player` class inherit those other class so it could use their methods. An arrow pointing to `Player` class indicates inheritance made by `Player` class from other classes.

`Player` class has several attributes, all of which are attributes required by parent classes the `Player` class inherits, for example `debrises[]` array, `debrisSound` game object, `explosiveRadius` float, and `explosiveForce` float are all attributes of `debrisOnCollide` class. For `Player` class to work perfectly, all of this attributes had to be inherited along with methods from `debrisOnCollide`. The same thing applies for every attributes from every classes in which the `Player` class inherits. That's where the various attributes of `Player` class come from.

Aside from inheritance, this game also uses override features of object-oriented programming. Easiest example would be to focus on `Enemy` Class.

Just like `Player` class, `Enemy` class uses inheritance from other classes to construct it's methods and attributes. And after the `Enemy` class is built, this class is used to construct various behavior of enemies, such as enemies that moving from right to left, and the other one that moving from left to right. To do this, `EnemyLeft` class and `EnemyRight` class use all of `enemy` class's attributes and methods, and then override only `moveDirection` attribute. `EnemyLeft` overrode it to "-1" while `EnemyRight` overrode it to "1". This simple gesture results in two completely different kind of enemy, although both of them built from the exact same base.

If this pattern followed, all of other classes and their OOP relations in the class diagram from figure 5 could be understood perfectly.