**Bomb Game**

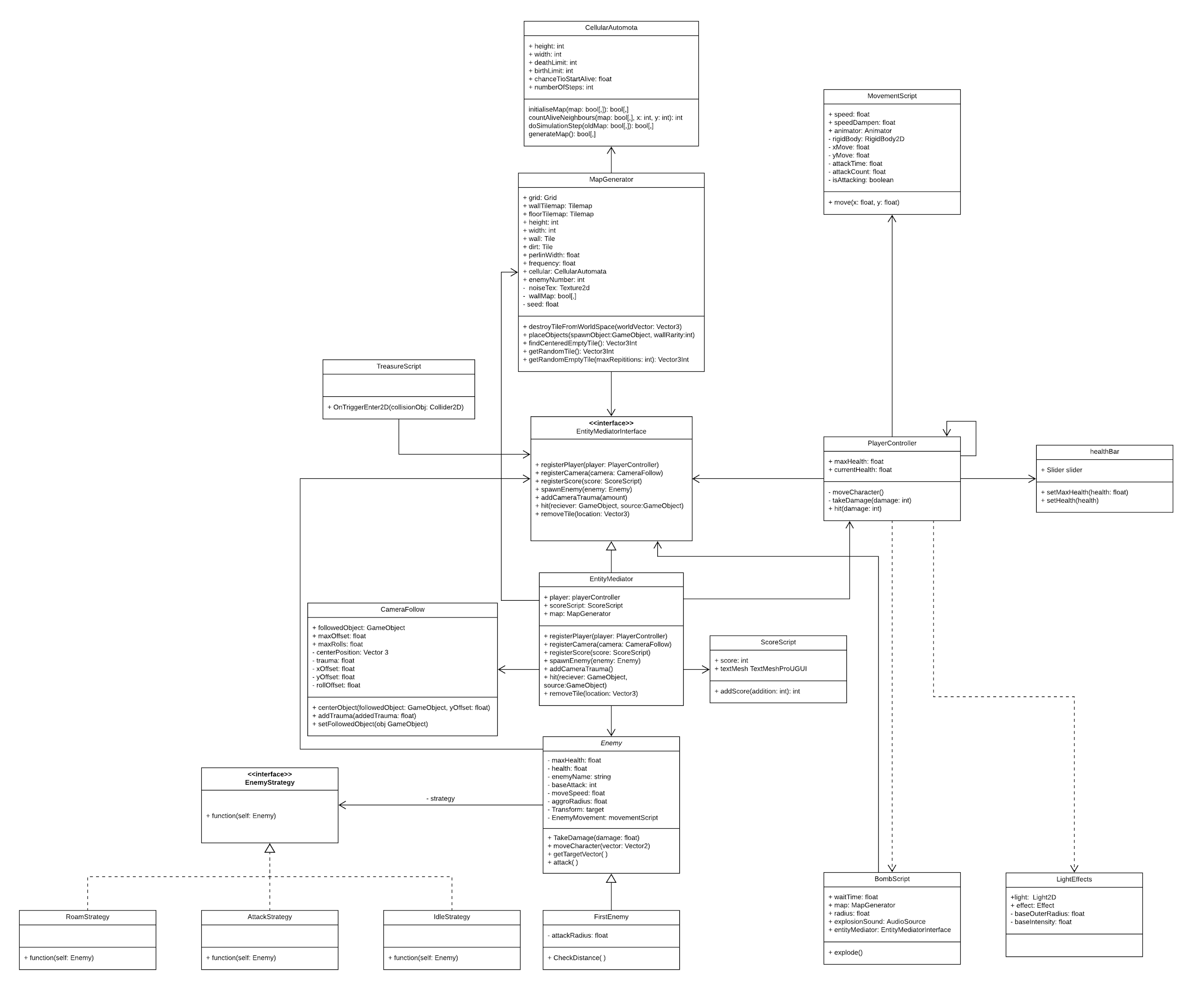
By:  
Bradley Stukas, Benjamin Gonzalez, Yumi Park

1. **Introduction**  
     
    Our project was to make a game in the style of a rogue-like dungeon crawler. Players would enter short instances where they must complete the game without dying. Upon completion or failure, players will receive rewards and experience depending on their progress, which can be carried over as advantages in future sessions. Our intent with this project was to apply the patterns and structures we learned over the semester to something we were passionate about, while also gaining experience in utilizing game engine software. We feel confident in our capability of using java as it has been the primary language used to teach object-oriented programming, and so Unity’s utilization of C# was a good opportunity to work with a different object-oriented language.

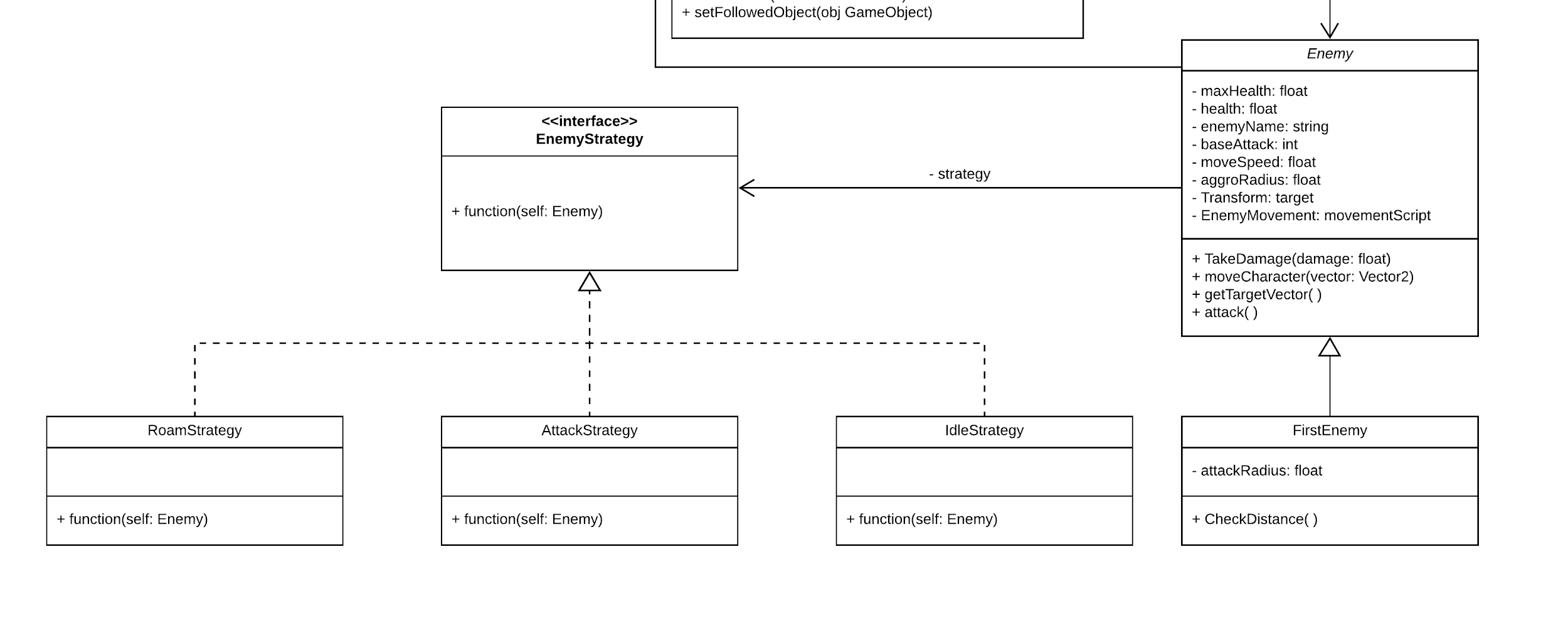
Some features of this type of game include random generation of dungeons, which includes the area that the player walks around in, and the enemies inside. Because our levels are randomly generated, we do not need to hand-craft individual levels, however our future plans include an alteration to our dungeon generation algorithm that would spawn in different types of enemies, items, and overall dungeon pattern. In our current progress we have a type of terrain manipulation in the form of bombs, as well as a basic combat system with close range combat. In the future it is planned to add different classes which utilize different forms of terrain modification such as digging as well as different types of attacks and play styles. We also feature enemies with flexible AI which have different strategies depending on the type of enemy and their current health. Some key features to be developed here include bosses, a variety of enemies, and new enemy strategies that take into account the player’s weapon and class.

The main audience for our game would be players who are interested in this genre of dungeon crawling rogue-like video games. Because of the rogue-like nature, there is a natural difficulty curve that tends to appeal to players who are skilled with this type of game. However, we hope to target less-experienced players as well as an introduction to rogue-likes and dungeon crawling. Our goal is to appeal to a general audience with a brighter color palette and an overall non-threatening aesthetic along with an easier learning curve.

1. **System Design**

The design patterns we used in our project included, Strategy for the enemy a.i., State for the player animation, Singleton for the player controller, and Mediator for connecting the main part of our project.  


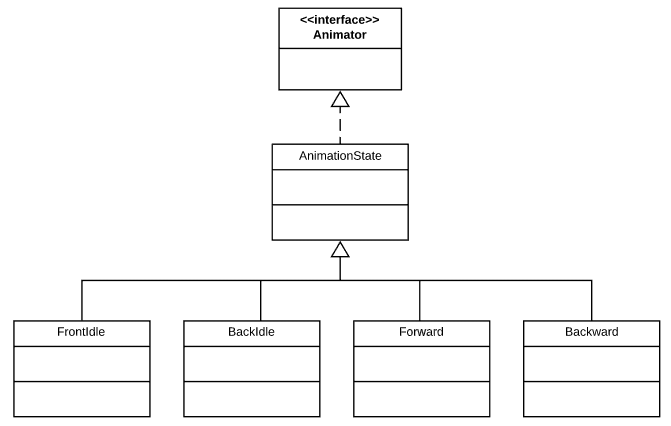
* 1. **Strategy**

****

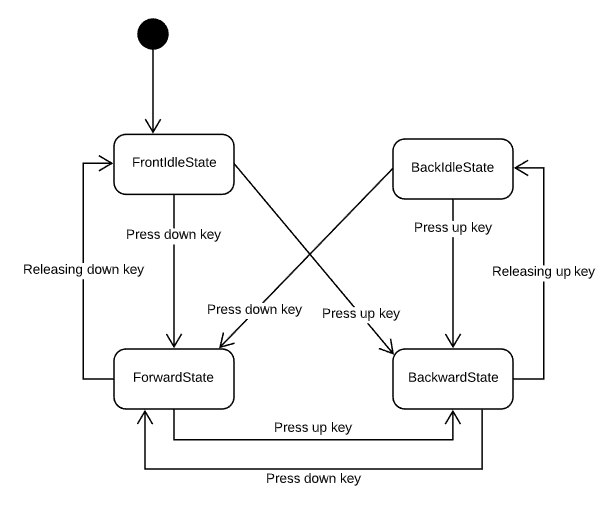
The strategy design pattern was used for controlling the action of the enemy. By doing this the enemy is able to change its strategy based on its current condition. Currently, the strategies are not used in that complex way, but if more enemies were to be added then it would be very easy to both create new strategies and implement existing strategies. In theory, since the strategies, if they can, just use the methods of the enemy in general so that they can be as reusable as possible.

* 1. **State**

Our state design pattern was used in how we handled player animations. This allowed us to have different animations to show what direction the player was actually walking. Each new state was dependent on what button the user was pressing along with an idle state if the user stopped giving button inputs.



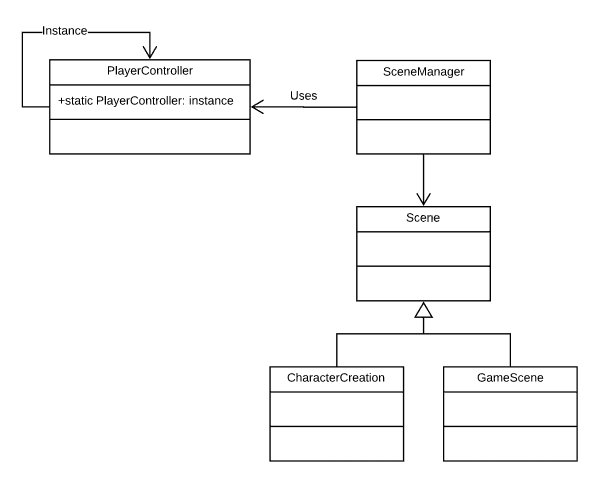
State pattern class diagram



Animation state diagram

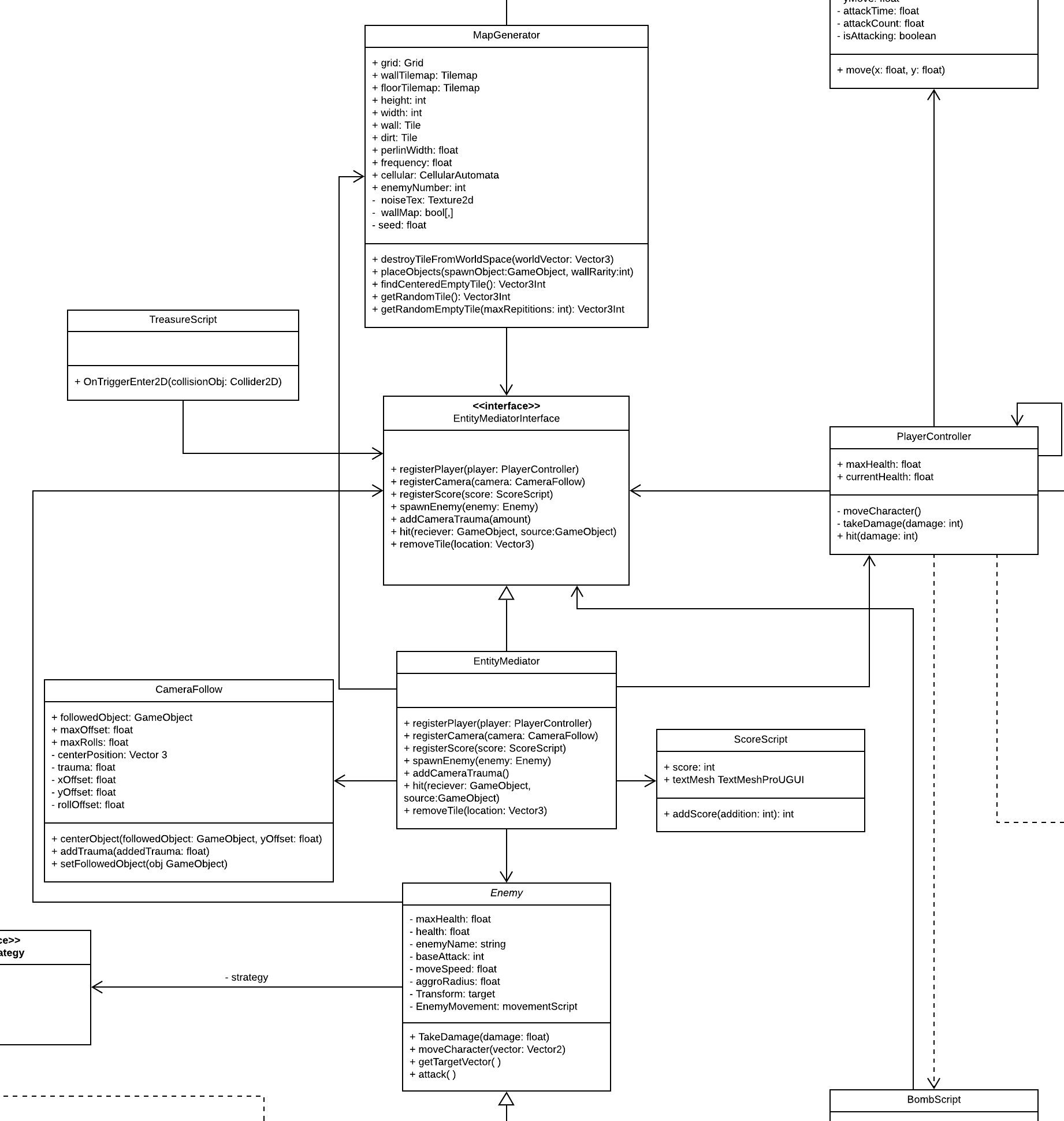
* 1. **Singleton**

Our game utilizes the singleton pattern in its character creation. In general, scripts can be called multiple times across different scenes and reinitialize variables by creating new instances. In our example, our goal is to bring a character that was generated in the character selection screen to the game scene so the player’s choice is carried over. If we did not use singleton, transfers back and forth between scenes would create multiple instances of the player character.



* 1. **Mediator**

We ended up using the mediator pattern in order to reduce the amount of strong coupling in our system. We wanted this because as we added more and more to the project, there started to be situations in which when we would change something in one script it would sometimes break things in other scripts. This was never a huge problem due to the size of the program making it easy to track down the problem, but it was becoming more of a problem the bigger the program got. Because of this we created a mediator interface and class to handle interactions between separate entities. This doesn’t necessarily mean that every interaction between separate entities is going through the mediator though. Some relationships that are limited to a very small amount of connected classes are left as strongly coupled relationships. One example of this is the relationship between the character controller and the movement script.

****

Mediator class diagram

1. **System Implementation**

**Enemy actions - strategy pattern**

When implementing the strategies different methods had to be used. The roam strategy for example was created by casting out a ray in front of the enemy and having it follow the direction of that ray. The ray would then be rotated if it collided with any walls. Sometimes enemies would get stuck using this method if they got into a position where they were both going perpendicular with a wall and their ray cast was missing it. Because of this, a very small constant rotation to the left was added so that enemies who get stuck could slide off if they get stuck. In the future, a more effective version of this strategy would be to use two ray casts that are pointed to the left and right of the forward vector and decide which way to go based on which vector is getting a hit. Doing it this way would make it so that the enemy would have an easier time getting through some currently hard to get through passages, such as a slowly sloping left turn in a tight passage. The other strategies were far more simple to implement because they were either just doing nothing for idle, or running towards the player and attacking for the attack strategy. The enemies act by calling the function method of their strategy class every frame which decides what the enemy will do. The enemy will also change its strategy in some cases depending on what’s happening.

**Map generation**

The random map generation was done using cellular automata which is a process by which cells grow and die. Using this process creates very believable tunnels because the cells like to grow close to one another.

Ex.[](https://thumbs.gfycat.com/BarrenTidyClownanemonefish-max-1mb.gif)

(click on the picture)  
The map was also then made random by using different starting values for the process each time. We were also able to use a set of parameters to change how the blocks formed in order to get the specific type of caves we wanted. After the map is generated(as a two dimensional array of boolean) a tilemap, with the property that every tile has a hitbox, is filled by just placing a tile in the grid spots in which the coordinates put into the boolean area are true. The treasure is then placed by looking for open spots on the map in which there are many blocks surrounding the location. This is because doing this way tends to spawn treasure at the end of passages. The enemies are then spawned by taking random locations and spawning an enemy there if the location is open.

**Bombs**

Bombs can be placed by the player and explode tiles when near them. The way it does this is by calling the mediators kill tile function using an array of different vector 3 world vectors. This kill tile function then uses the map generator’s destroy tile based off of world location function to translate the world space vector to a coordinate on the grid and destroy it if there's a tile there.

|  |  |
| --- | --- |
|  |  |

**Collectible objects**

To implement objects the player can collect, we utilize object hitboxes. Within unity, objects can be assigned a designated space which represents the object’s form within the game space. Both the collectible object and player character are assigned hitbox spaces, and a script is written such that when the player character’s hitbox collides with the object’s, an event is triggered which destroys the object and increases the player’s score.

|  |  |
| --- | --- |
|  |  |

**Animations - State**

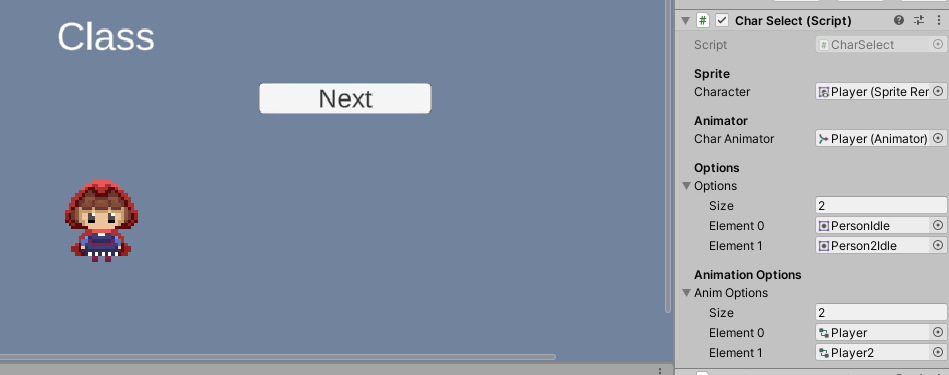
Unity provides the framework for implementing animations to this game through their animator class. Animations can be added to an existing object and connections between different animation states are written in code. Variables are instantiated in the animator which are used as triggers to move to different animation states.

In the below example, VertMove and isAttacking are parameters that have been defined in the animator which are utilized in the transitions between idle, attack, and forward. Within our script, when the player presses the “R” key, the update function sets isAttacking to true, which triggers the transition between idle or forward to the attack animation.

|  |  |
| --- | --- |
|  |  |
|  |

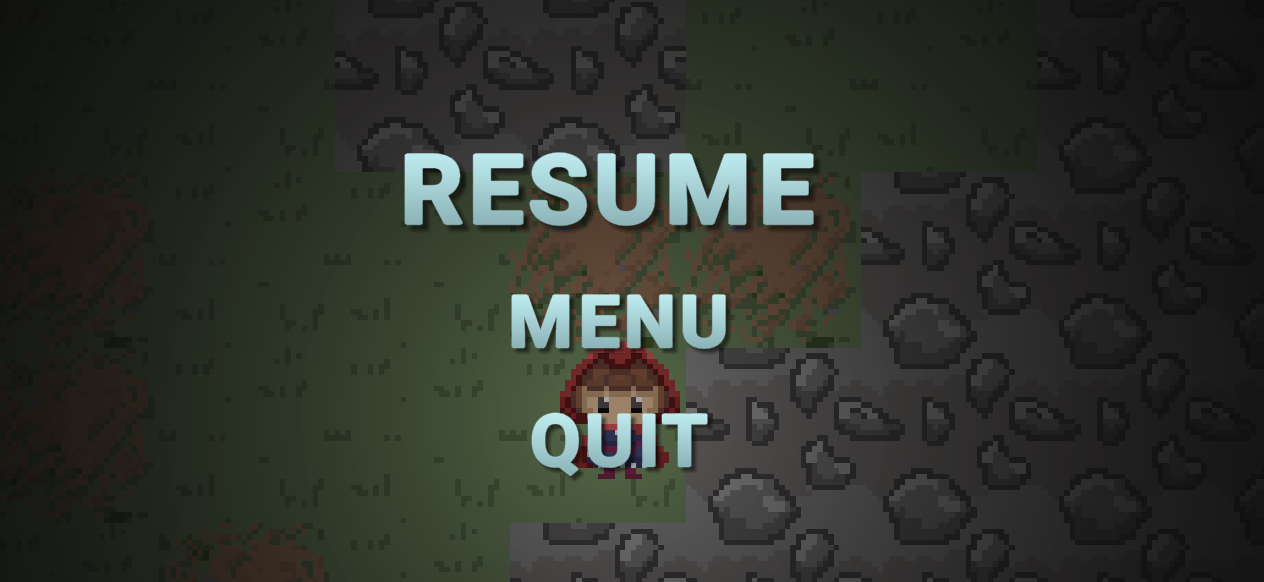
**Character selection**

In the character selection, players have the option to choose between two different characters. These characters correspond to different sets of sprites and animations which must be assigned to the primary player object. This is accomplished by creating an arraylist of sprites and animation sets that is cycled through by clicking the “next” or “prev” buttons. Starting the game locks the selection to that instance of the player object, which is then brought over to the game scene.



**UI**

For our UI we had the standard menu and pause menu for the game. The main menu is it’s own scene where a user can press play, options, and quit. When the user is in the game they are also able to press ESC to pause the game and show a pause menu with the options of resuming the game, heading back to the main menu, and quitting the game. I set the main menu as its own scene to keep things clean and it would allow us to do more complicated things with the main menu without it having a bad effect on the game. Implementing these menus included making buttons for each option a user could press and having a script attached to each button telling it where to go like pressing play on the main menu would head to the character creation scene. The pause menu while in the game has the same kind of implemented buttons with scene redirects as the main menu but also stops the game while in the pause menu.



**Main Menu UI Pause Menu UI**

1. **Lessons Learned**

Bradley Stukas: I wouldn’t necessarily call this a lesson learned because I already knew this, but this project did reinforce the importance of having multiple perspectives when working on a team. One example of this would be how Yumi used one sprite for the floor tiles when I was planning on using 8. This was because the games that I made in the past were from a different orientation where you do have to have more tiles for different relationships the tile can have to others, but that didn’t apply as well to this style of game. I guess another lesson I learned is the importance of weak coupling, since strong coupling did cause a couple of actual problems in our program. Honestly, although many of the design patterns don’t apply as well to games, the ones that do I’m really happy I learned and I could definitely see myself using in the short future on other game related projects.

Benjamin Gonzalez: With the current events going on in the world and everything going online, we were thankfully able to keep going with our project as planned. Some lessons I learned really revolved around Unity and how design patterns could be implemented well in Unity. Before I knew a little bit about Unity but now have learned a lot more in using Unity. It was interesting having parts of the game be split up to different team members using the Unity collab feature and seeing how our updated parts worked when one of us updated their part of the game. Learning the new design patterns this semester opened up a lot of new ideas on alternative ways of implementing the game and it was neat being able to actually apply them in a project. Overall learning as a team is something we all go through and it’s facing the challenges of them that are very important for any group project.

Yumi Park: I already felt confident working with object-oriented languages, so even though I had little experience with C# I didn’t feel too challenged using it. I’m glad I was able to gain an immense amount of understanding with the unity engine and the basic principles of designing a game. The biggest thing I realize I learned through this project is that the design patterns we learned in class are basic structures that have a variety of applications in different forms. It isn’t so much that our class diagrams must look and behave exactly as we have learned them, but their general application, benefits, and costs should be taken into account in regards to how we build our programs or software. This all goes in line with how efficiently computers execute code, readability, adaptability, and its effects on the software. Being able to recognize the design patterns and having a foundation for designing software structure for a given situation is the most important thing I feel I learned through this class and project. I also learned that I have a general cycle of frustration when I am learning and adapting to something I’m unfamiliar with.