Software Design Document

for

<Alpha Danmaku Rush>

Team: [TeamAlpha]

Project: [Alpha Danmaku Rush]

Team Members:
[Xiangjun Ma]
[Can Cheng]
[Tianqi Zhen]
[Guangbei Yi]

Last Updated: [3/23/2024 9:07:44 PM]

Table of Contents

[Update the Table of Contents]

Document Revision History

Revision Number	Revision Date	Description	Rationale
0.1	3/24/2024	Milestone 2 Document	Implement basic logic

1. Introduction

Our game Alpha Danmaku Rush is a Danmaku Hell game. Gameplay of Danmaku Hell is similar to that of other action games, in that it involves the player having to dodge a very large number of bullets as well as attack their enemies through good strategy and agility. Alpha Danmaku Rush is a game in which players control a character to move freely in eight directions on the screen, avoid the enemies' attacks of different designs, and encounter the Boss in the middle and later stages of the game. The players will be able to obtain props to strengthen them in order to overcome the boss's special attack mode. The bullet attack mode can increase the game's difficulty and playability. Team members will test it and maintain the difficulty level accordingly. Our purpose is to adopt good design strategies, design a complete and mature game architecture. We will make the level interpreter more flexible, increase the game's scalability, and publish it as an independent desktop game.

1.1 Architectural Design Goals

Availability

1. Use GitLab CI/CD to automatically build, test and deploy code commits. This helps in reducing downtime and ensures that the game is always ready for testing and playing.

2. Create the game architecture in a way that can handle errors gracefully. Implement error handling and logging systems to detect and potentially recover from errors without causing the game to crash.

3. Embrace an architecture where components like the level interpreter, game engine and user interface are loosely connected. This makes updating and maintaining the game easier allowing individual modules to be updated without disrupting the gameplay.

4. Manage resources in memory usage and asset loading to prevent crashes and ensure smooth gameplay even on less powerful computers. Optimize. Use loading/unloading techniques, for game resources to maintain responsive gameplay.

Testability

1. Create an automated testing framework that includes unit tests, integration tests and end, to end tests. For example, we have unit tests for game elements like player movements and enemy behaviors, integration tests for game features such as collision detection and level advancements, and end-to-end tests for gameplay scenarios.
2. Utilize objects and dependency injection when testing isolated components to avoid reliance on their dependencies. This approach is especially beneficial when examining game logic AI actions and interactions among game elements without needing the entire game engine to be operational.

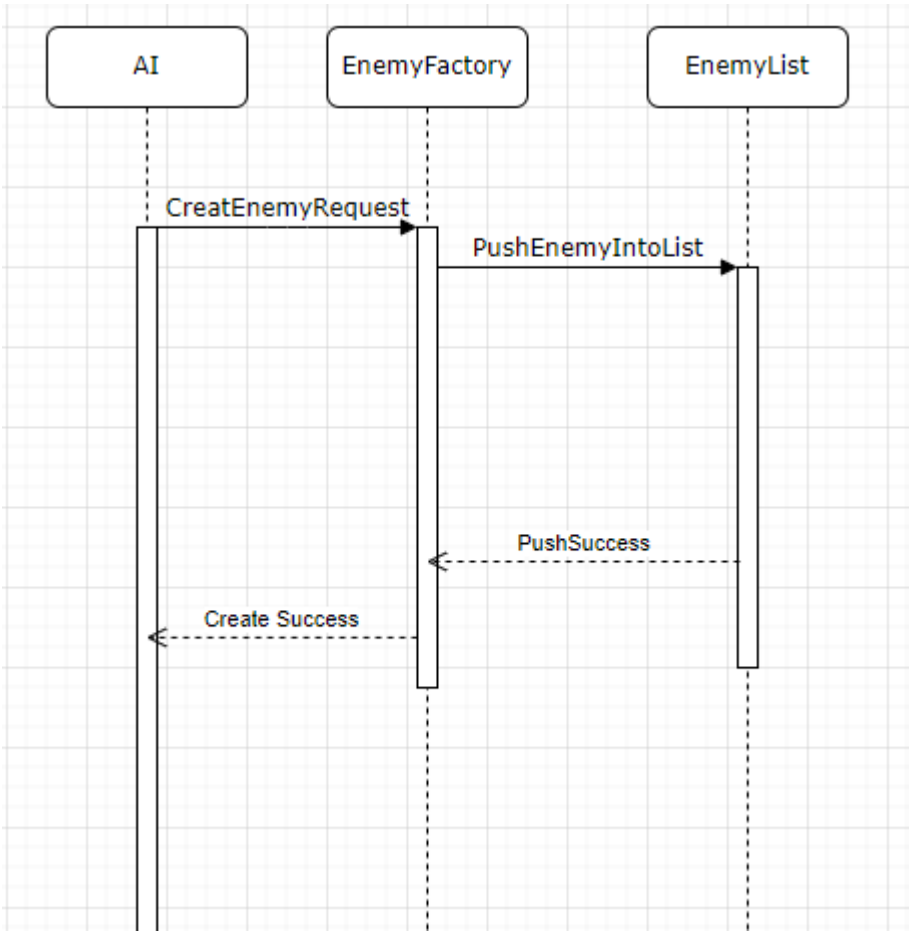
3. Embrace test-driven development (TDD) methodologies to prioritize testability from the beginning. Crafting tests before implementing functionalities not only ensures that the code aligns with its design and functions correctly but guarantees that the game is constructed with testability in mind.
4. Integrate logging mechanisms and debugging tools into the game structure. These tools should offer logs for issue identification and resolution. Consider including features like an overlay, within the game to showcase real-time metrics, hitboxes, or other pertinent information during testing.

2. Software Architecture

2.1 Overview

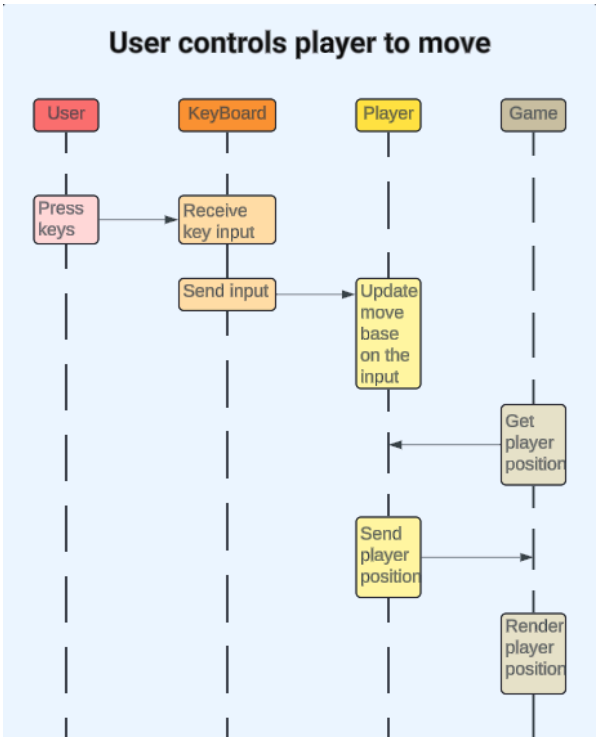
[In this subsection, provide a brief overview of the software architecture. Introduce the architectural pattern that you use in your design model, and state the rationale in choosing that pattern. Briefly describe the overall decomposition of your system in terms of subsystems.

Provide a UML diagram (component diagram or class diagram) that shows your subsystem decomposition based on the architectural pattern that you selected (either Multi-layered, or MVC architecture) The UML component diagram should illustrate the architectural pattern that you adopted. Clearly identify the different subsystems, which will be further elaborated in Section 2.2.]

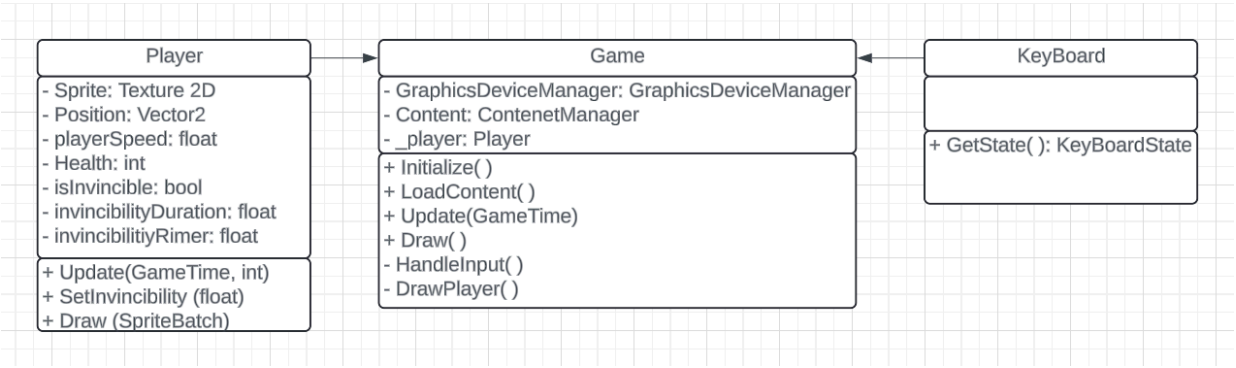
- **AI Spawns enemies**
- **Sequence Diagram**
- 

- **User controls player to move**

Sequence diagram:

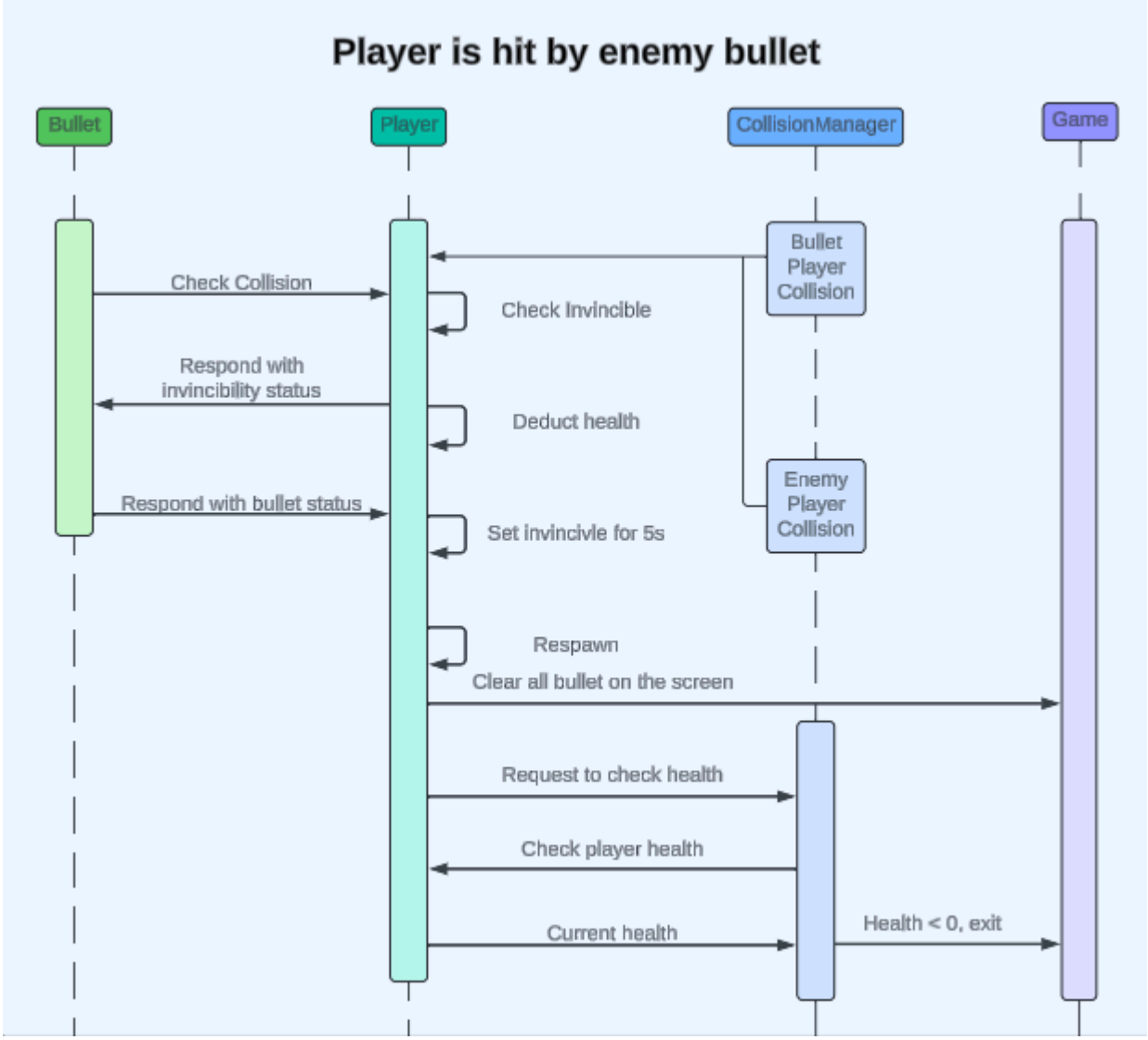


UML diagram:

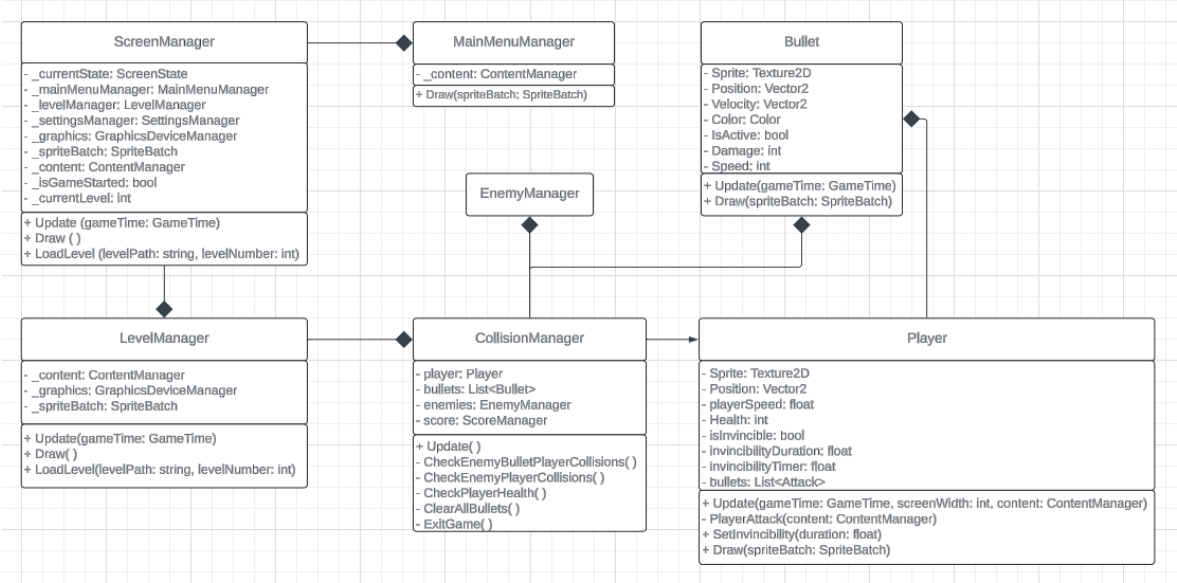


- **Player is hit by enemy bullet**

Sequence diagram:


Player is hit by enemy bullet
Bullet
Player
CollisionManager
Game
Bullet Player Collision
Check Collision
Check Invincible
Respond with invincibility status
Deduct health
Enemy Player Collision
Respond with bullet status
Set invincivle for 5s
Respawn
Clear all bullet on the screen
Request to check health
Check player health
Current health
Health < 0, exit

UML diagram:



2.2 Subsystem Decomposition

The project now mainly has five subsystems, the following table is a simple overview of each subsystem. Movement and Input management subsystems are mainly defined under the enemy and the player subsystem to coordinate different partial functions of those.

Subsystem	Description
Player	Setting Player's position and updating position when receiving movement order.
Enemy	Handle generating enemy. There are two types of regular enemies: enemyA and enemyB, mid boss, and final boss. Besides deciding time and type an enemy will be spawned, it will also set bullets to the enemy.
Bullet	Create bullets.
Movement	Handle enemy movement
Input management	Receive Keyboard Input

2.2.1

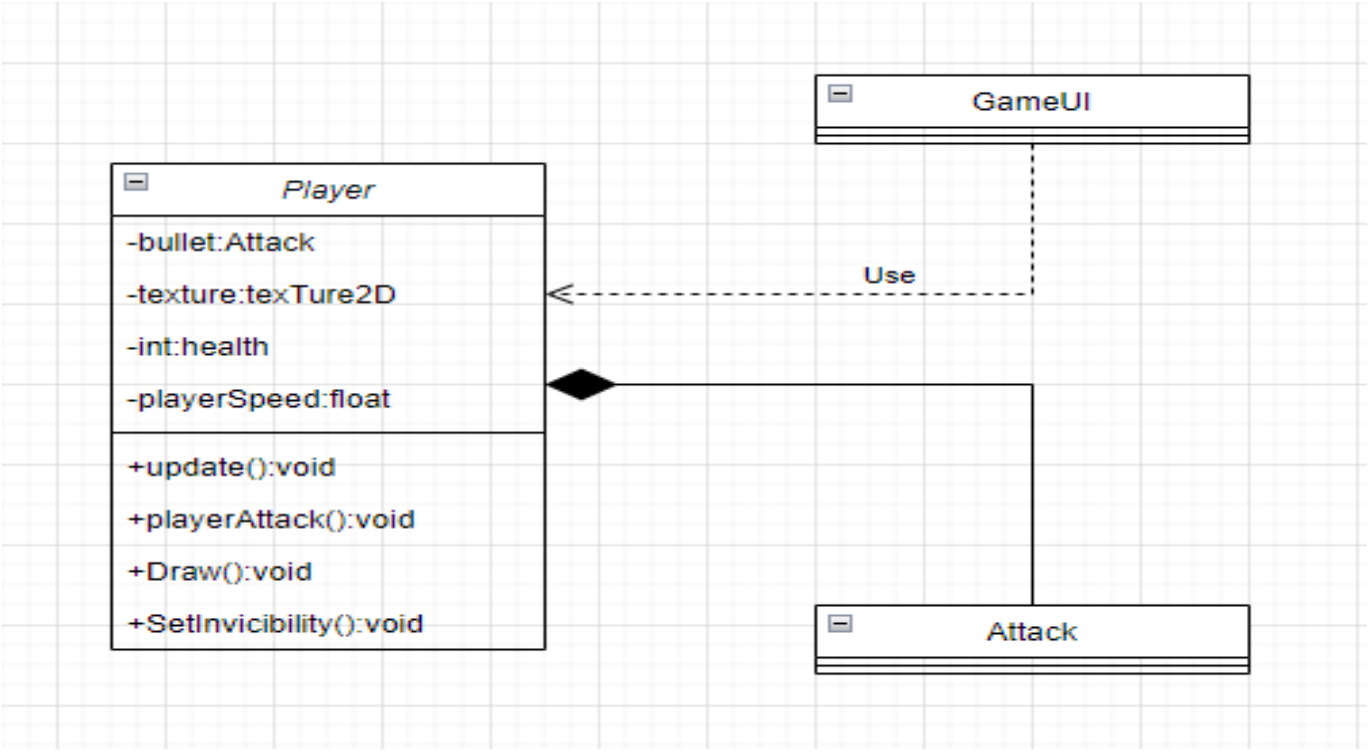


Figure:Player Subsystem UML

2.2.2

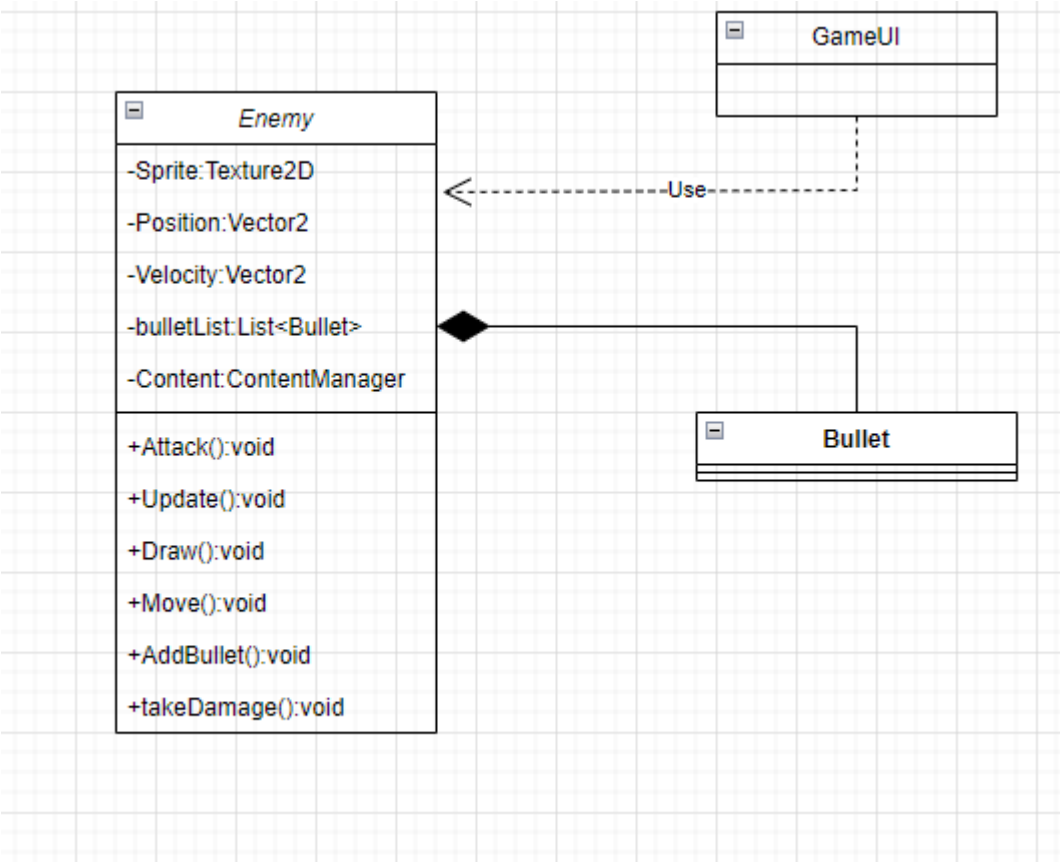


Figure: Enemy Subsystem UML

2.2.3

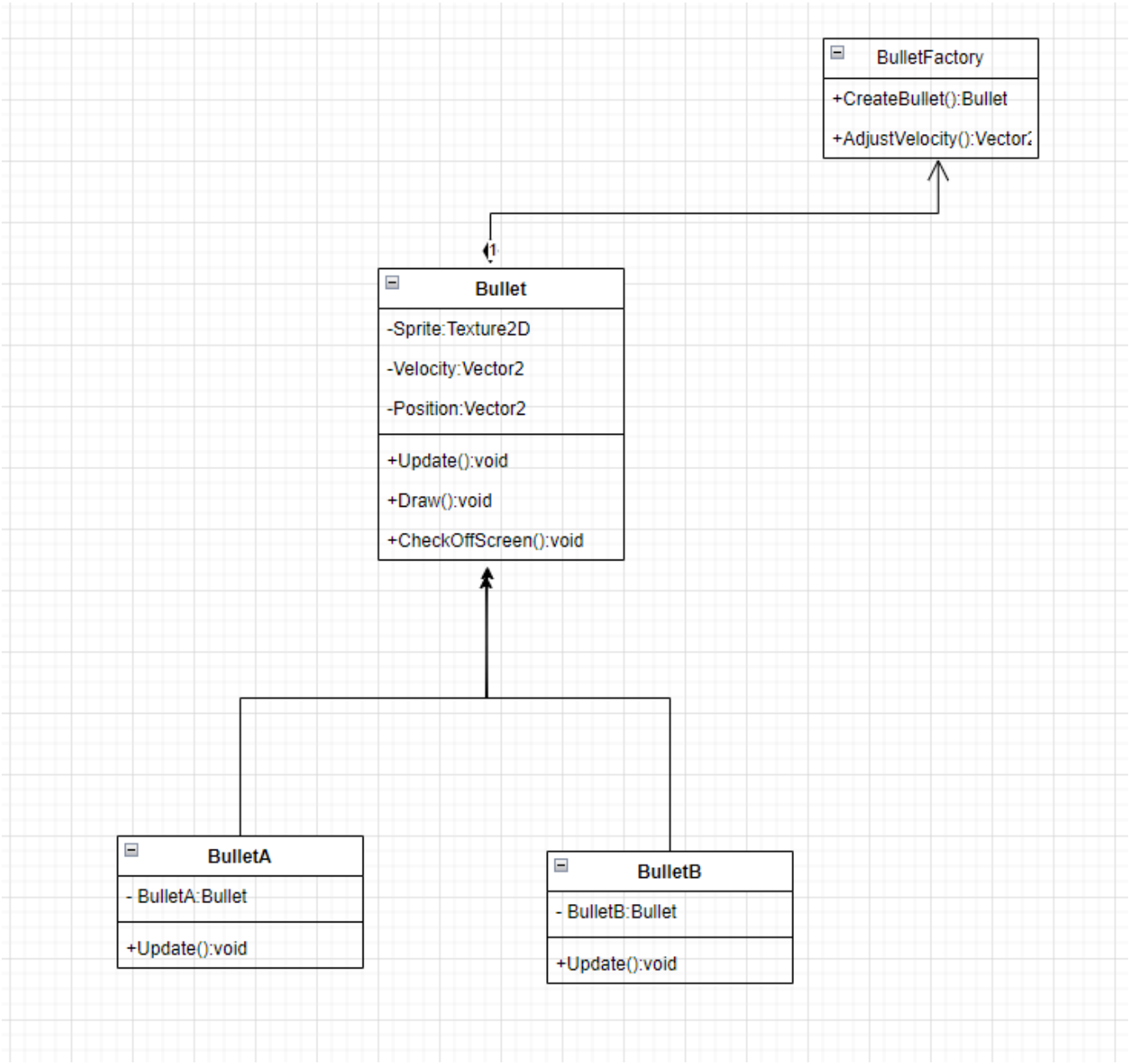


Figure: Bullet Subsystem UML

2.2.4

The program is using multilayer architecture. It can be divided into three layers: GameUI layer, object creation layer(including player and enemies), and bullet creation layer; they have clear “having” relationship and do not need to manage data across different layers.

To handle different types of enemies and bullets, we are using factory patterns to instantiate them and push them into corresponding lists to manage them. To follow the command pattern and single responsibility principle, we also have several manager classes to handle specific events in the game.

3. Subsystem Services

Player Subsystem: This subsystem is mainly used to manage any orders the player gives to their character, including moving and firing bullets, and generate controllable character image in the beginning of the game. Input control is a subsystem of this subsystem. It also uses attack class to generate attack.

Enemy Subsystem: This subsystem handle generating enemies in the game progress. It use bullet subsystem to generate a bullet list attribute. In the main game loop, it have access to the game time to generate specific type of enemies during the game progress. It has movement subsystem to manage movement of enemies.

Bullet subsystem: This subsystem is used to generate bullets. Depends on type of characters(player's character, regular enemies, and boss) it supposed to instantiate different type of bullet as concrete product and push them into corresponding lists.

Please see diagram in part 2.2 for the reference of subsystems.