Software Design Document

for

The Matrix

Team: TeamMatrix

Project: The Matrix

Team Members:
Afoke Abogidi
David Kelly
Kulpreet Singh

Table of Contents

Document Revision History

Revision Number	Revision Date	Description	Rationale
1.0	March 15th, 2022	Deliverable 2 Systems Design Document	Initial prototype design with Model View Controller Architectural strategy implementation featuring an introduction to the Matrix system, a general overview of the project, the quality design goals and architecture of the project and descriptions of the various subsystems within the system.
1.1	March 27th, 2022	Final diagrams	Applied factory patterns after code changes

List of Figures

List of Tables

1. Introduction

The Matrix is a standalone monogame-based game application. It is a bullet hell game which is a subgenre of shooters, a game style that originated in the mid 1990s. In this game style, the entire screen is mostly filled with enemy projectiles and features a player combating different types of enemies while dodging their projectiles. The matrix features a gameplay that occurs in phases. These phases see the enemies attempt to take down the player spacecraft by shooting in different combative patterns. Each phase is unique and more complex than the previous and can sometimes feature a stage boss with even more powerful and elaborate attacks. The player spacecraft can shoot to attack enemies and bosses and can dodge the shots targeted at it. The player can also achieve power-ups during the course of the game to enable special abilities to stay protected from enemy attacks.To play the game successfully, a player must survive all forms of attacks from all the different enemy types by maintaining a health point of above zero and completing all phases. The purpose of this project is to learn and apply programming design principles, patterns and system architectures in the creation of flexible, extendable and reusable software components and systems.

1.1 Architectural Design Goals

With the goal of effectively managing the complexity of The Matrix system and designing highly flexible and modular sub-systems and reusable components while also reducing the risks of defects within the system, several qualities are of paramount importance. These goals ensure a bridge between the user requirements and the technical requirements of our system and reduce risks associated with a system of this nature.

1.1.1 Performance Design Quality

One of the major quality goals of this project is to ensure the execution of actions within the game are at a highly responsive rate. This involves timely responses, efficient resource utilization as well as the capacity and consistency of the services provided by the system.To meet this requirement the Matrix system properly manages its resources by designing an efficient collision detection system which is our most computational and resource intensive system to utilize a partitioning mechanism for creating entity object collections. Using these collections of objects instead of iterating through individual entities is a lot more effective and involves far less time and resource consumption, thereby speeding up the overall game processing time.

Performance of the Matrix system is also enhanced by proper model coordination. All entities of the system display properly coordinated movements as a result of an efficient and reusable movement system designed. The system also features an input system for the player inputs that is highly extensible to accommodate additional input types if need be. These highly efficient services are designed as individual subsystems and components to allow for flexibility, extensibility and to high responsiveness of the system.

1.1.2 Availability Design Quality

Another major goal The Matrix system focuses on, is providing the required services at least 98% of the time. This is usually expressed as the ratio of the available system time to the total working time. For the Matrix system to achieve this goal, the system aims to minimize the number of faults that occur in the system and the impact any faults might have on the system. This is achieved through thorough fault prevention and detection and an extensive fault recovery procedure. The system adopts the Model View Controller architectural design to properly separate and allocate responsibilities as one of the strategies to easily detect faults. Proper data modeling including the single responsibility design principle is also adopted, this allows for proper exception handling and helps recognize and handle faults in the system. With faults detected earlier and handled properly to include retrying and redundancy logic, the system recovers quickly and failures which can bring down a part or the entire system are avoided.

Functionalities of the system that are highly intensive and more prone to faults are designed as subsystems with proper management of resources needed. This reduces coupling and increases cohesion and allows for much easier prevention and detection.

2. Software Architecture

UML sequence diagrams showing major use cases of the The matrix game are featured below:

AI spawns enemies

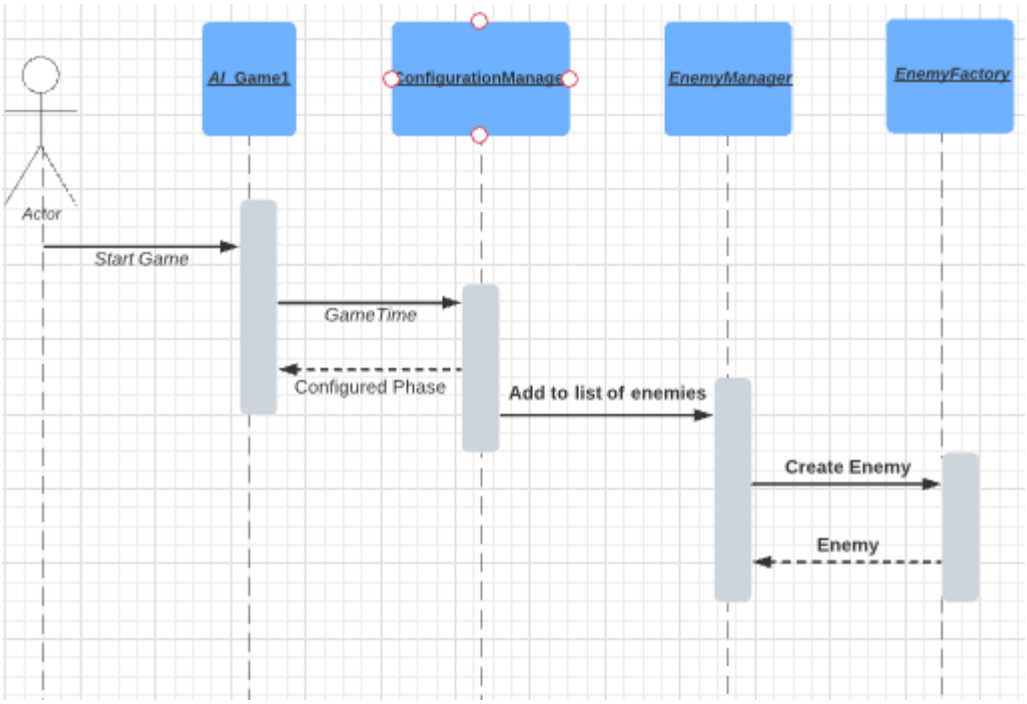

Figure 2-1

User controls player to move

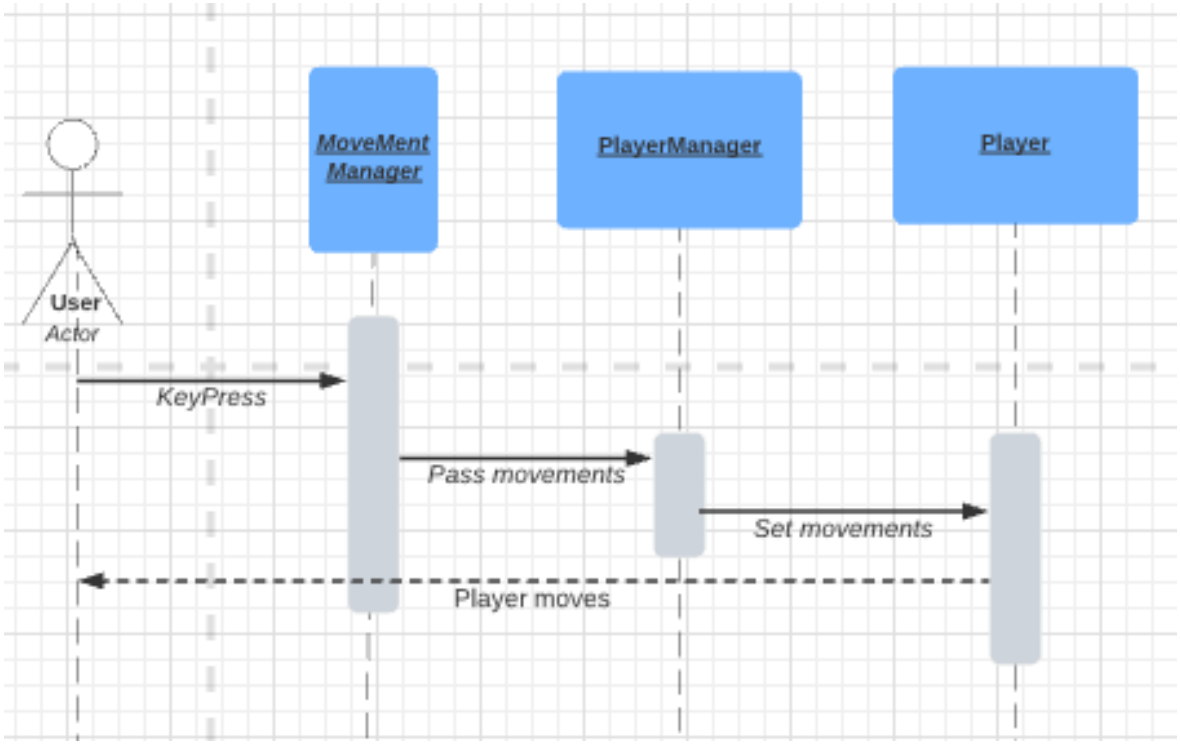

Figure 2-2

Player is hit by enemy bullets

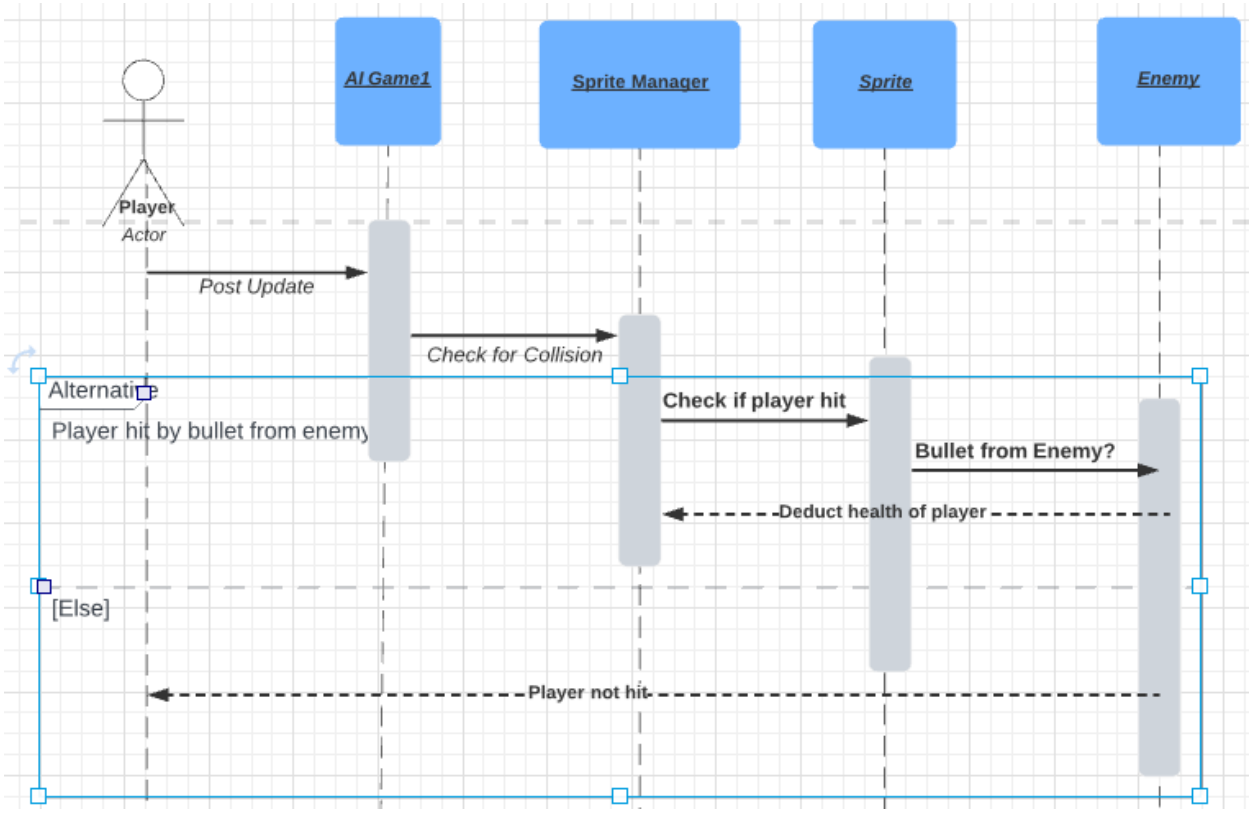

Figure 2-3

2.1 Overview

We are using the MVC architecture pattern. The reason for this is that it allows us to implement a separation of concerns. We have a user interface, a set of controllers and then models that hold the data.

The Game class uses Monogame which sits on the Microsoft xna framework. The class creates a graphic device manager that is associated with the game instance. This class takes to the UI and is considered our “View”. Updates and Draw methods use the controllers to get data from the models and then will update the View.

The controllers get data to fill the models of player bullets, projectiles, bombs, enemies, menu’s, and player.

Gametime provides a snapshot of timing values we use to get time to determine when an enemy will appear in the view. We also use it to call the Die() when lifespans are depleted.

A game loop is provided by Monogame which calls Update(Gametime) and Draw(Gametime).

A menu system will allow a user to start the game, configure settings or exit the game. The configuration will allow a user to change direction keys for the player from arrows (default) to WASD.

A scoring system will keep track of scores during the game. Each enemy type has a different point system.

A health system will monitor the players health along with enemies. It will take longer to kill the boss and final boss compared to the basic enemies.

Model View Controller

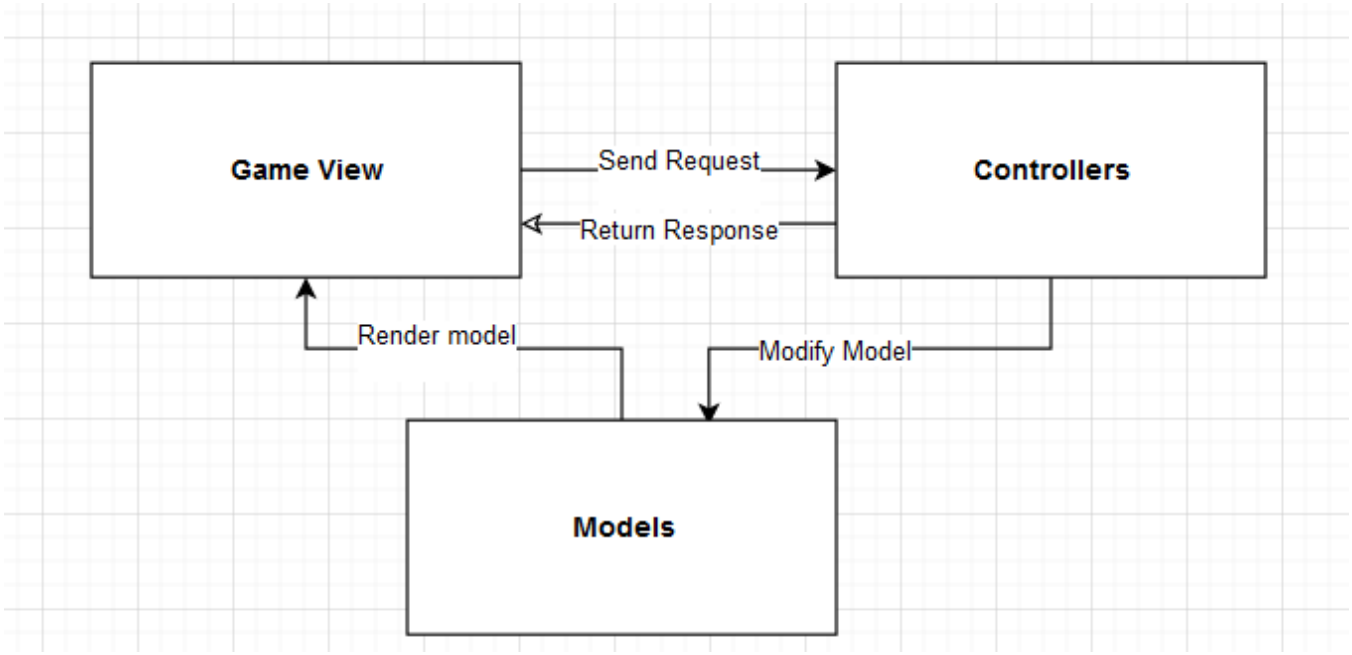

Figure 2-4

Game Component Diagram

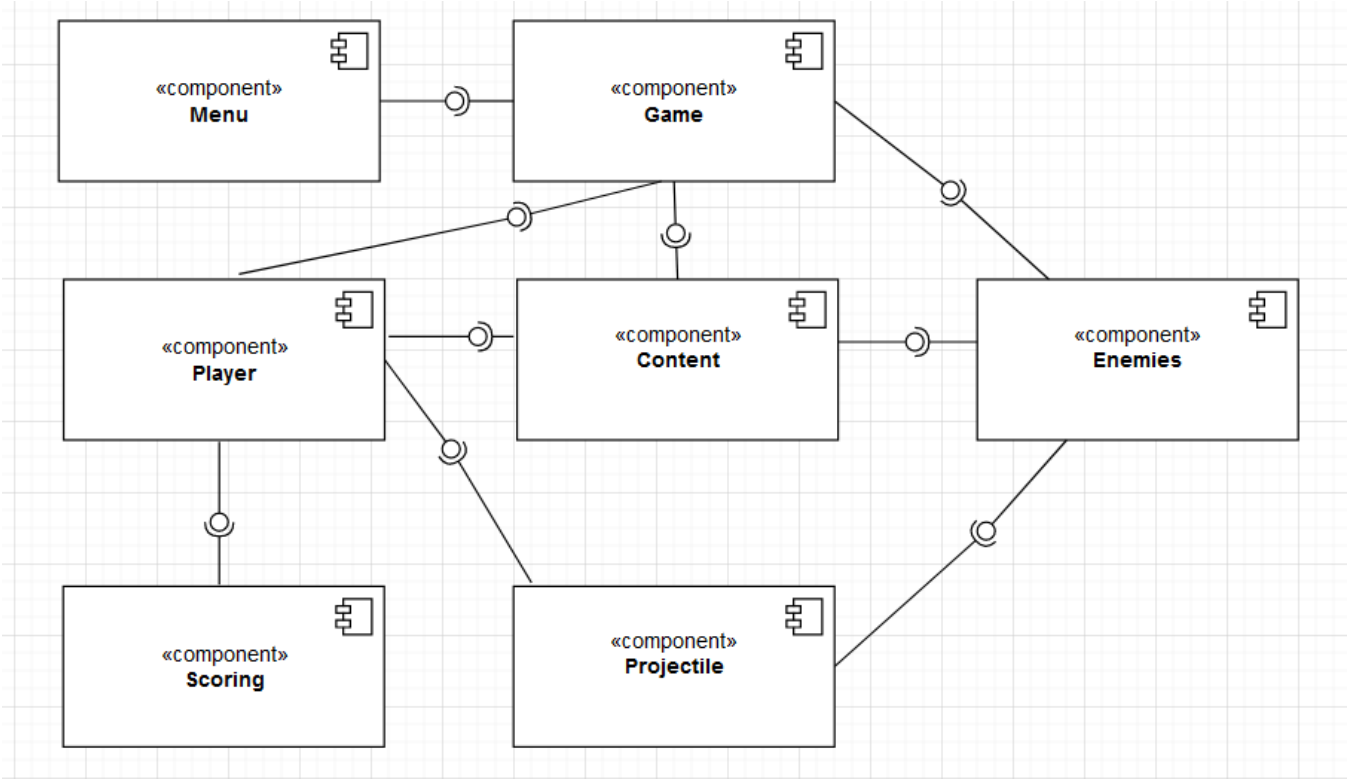

Figure 2-5

Enemy Sub Component Diagram

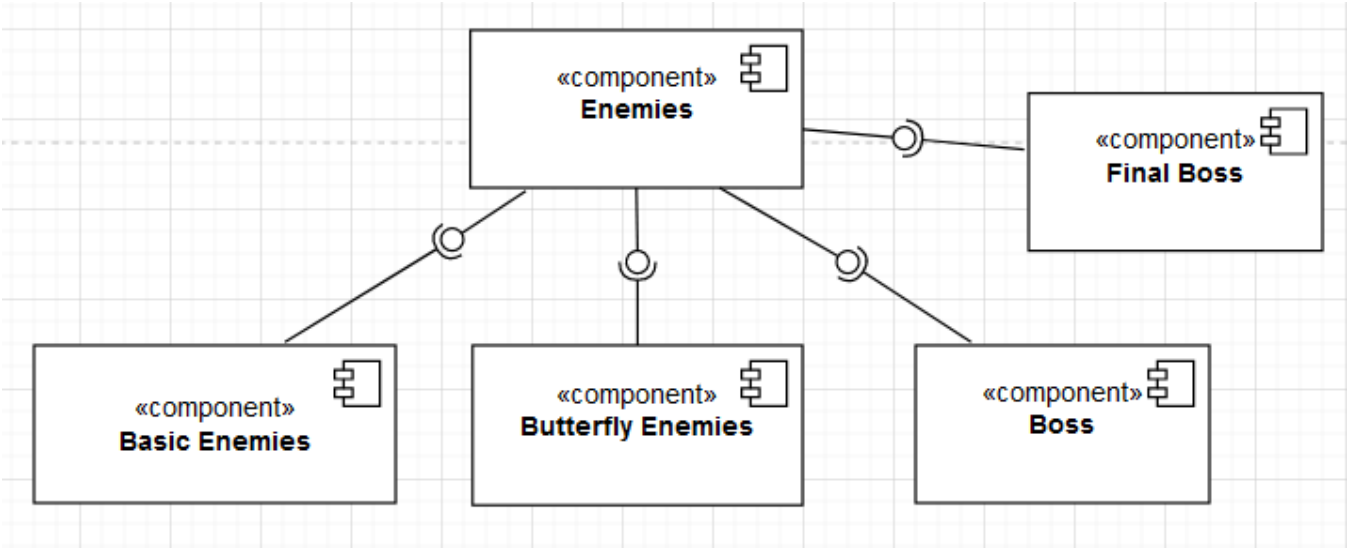

Figure 2-6

2.2 Subsystem Decomposition

The Game View serves as the application/entry game's point. From here, one can begin to configure the game, load content, draw, and update.

2.2.1 Game View Subsystems

This Matrix Subsystem is used mainly by the controllers which inturn controls the Game View. The Game View is responsible for displaying relevant information like the player's health, enemy movement and projectile movement for both the player and enemy.

Game View Subsystems

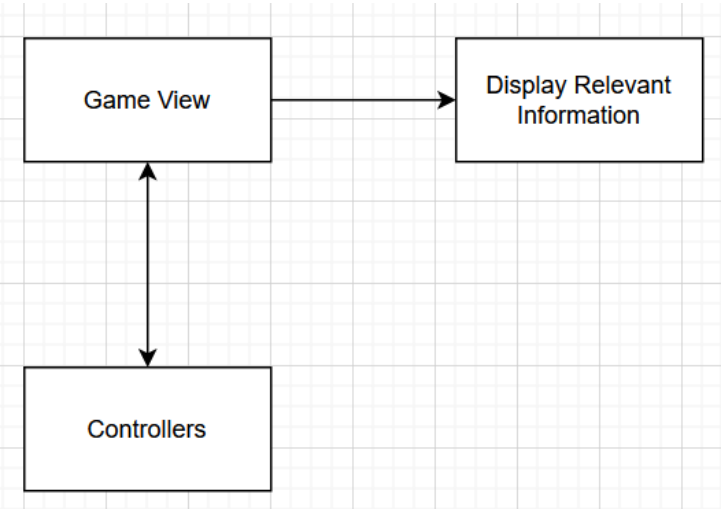

Figure 2-7

2.2.2 Controller Subsystem:

The controller is in charge of all aspects of the enemy's and players actions, including the creation of enemies, bullets, and the patterns that the enemies/player will shoot in.

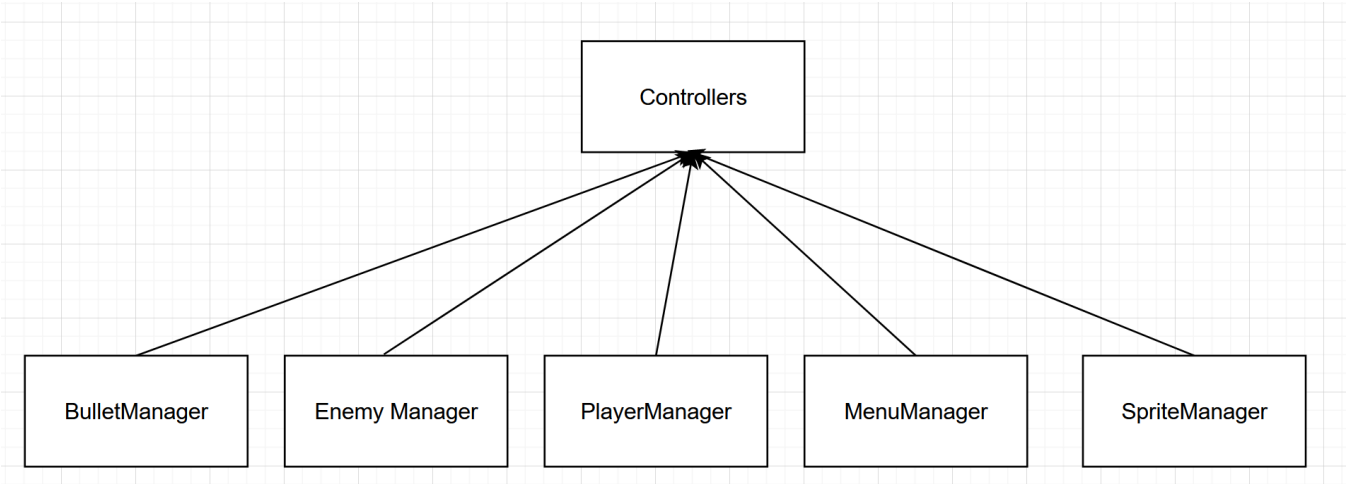

Figure 2-8

2.2.3 Model Subsystem:

The models include the most important aspect of the game. They contain the build aspect of the Enemies, the bosses (mid and final), the player characteristics and the various types of firing projectiles such as bombs and bullets.

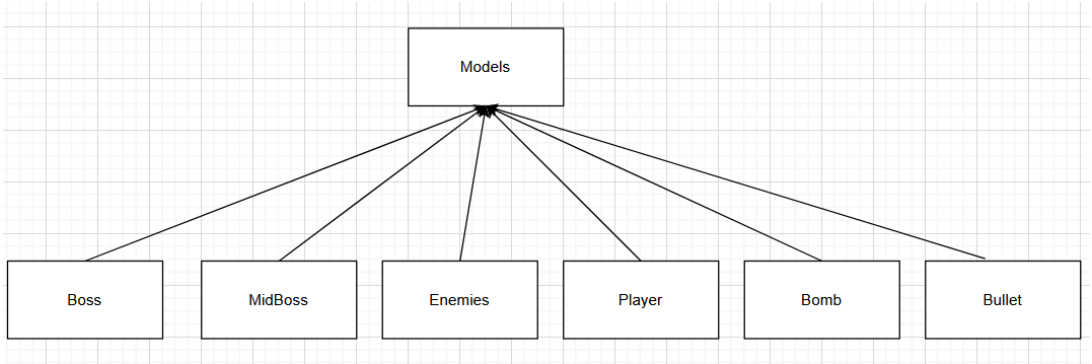

Figure 2-9

2.2.3.1 Player Movement Characteristics

The user actions subsystem will capture all user actions such as movements. At the time of this writing, the following button clicks are captured:

User Action	Description
Up Arrow (or W)	Move the player sprite up
Down Arrow (or S)	Move the player sprite down
Right Arrow (or D)	Move the player sprite to the right
Left Arrow (or A)	Move the player sprite to the left

Table 2-1

We have given the option to either have the movement as Arrow Keys, and WASD keys for players comfortable with that movement pattern.

2.2.4 Additional Accessories

The user actions subsystem will capture all user actions such as button clicks. At the time of this writing, the following button clicks are captured:

User Action	Description
Spacebar	Slow the game down
Left Mouse Click	Fire Projectile

Table 2-2

2.2.5 Design Patterns

Abstract Factory Design Pattern

The abstract factory method contains two factories: Enemy Factory and Projectile Factory as subclasses inheriting from it.

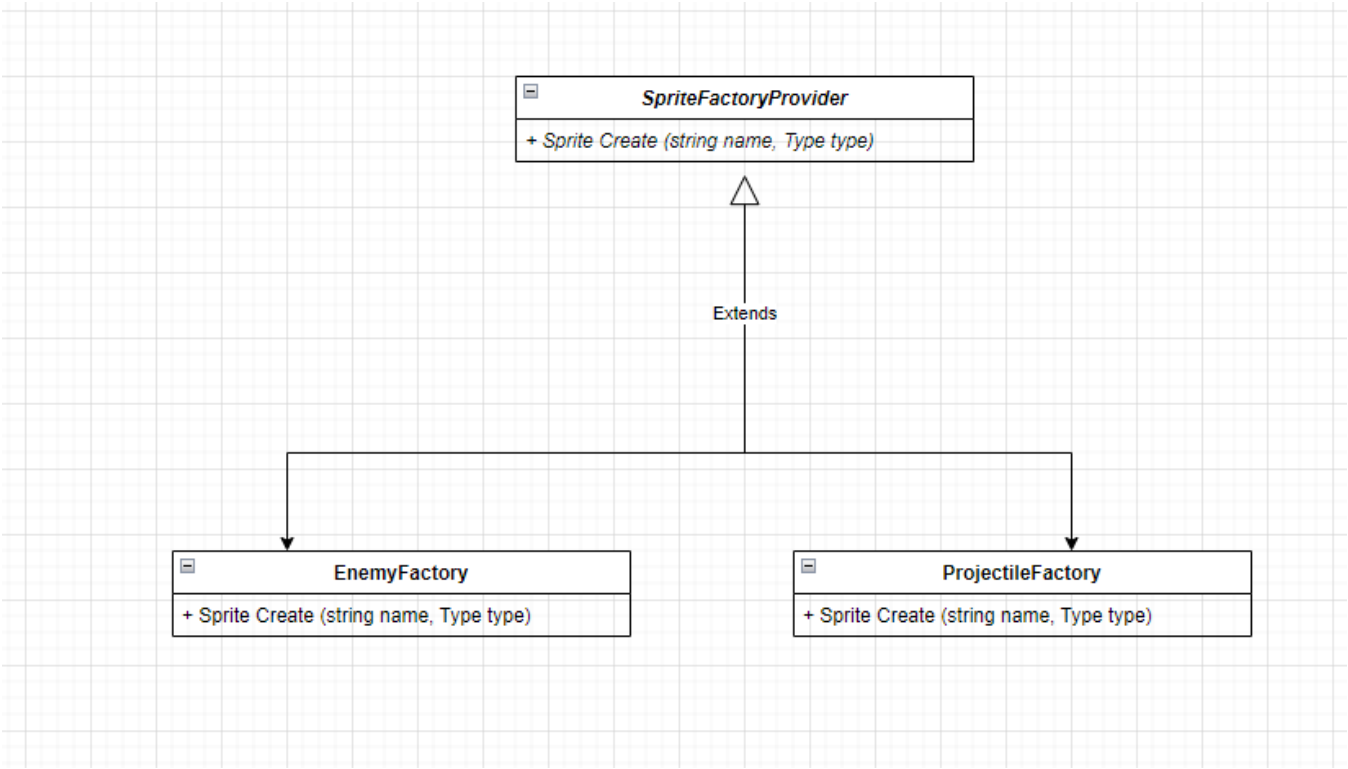

Figure 2-10

Factory Method Design Pattern

The factory method Create is overridden by the subclasses EnemyFactory and ProjectileFactory to create their corresponding concrete enemy product and projectile.

Singleton Design Pattern

This consists of a single class that is responsible for creating an object while making sure that only a single object is created. For example, the singleton pattern is utilized in the creation of the Final Boss and Mid Boss as seen in the class diagram below.

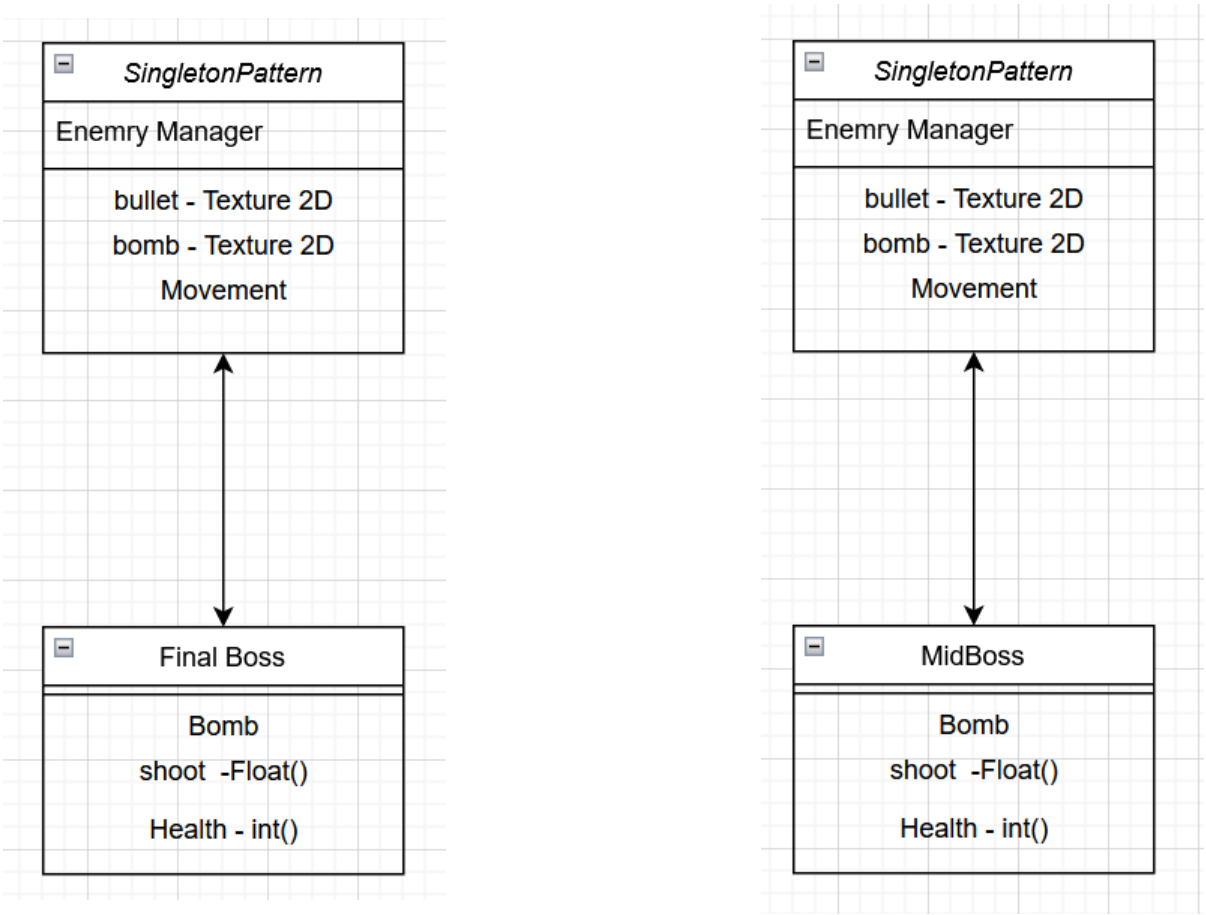

Figure 2-11

3. Subsystem Services

The Matrix system follows the Model View Controller architectural design pattern which is characterized by three major subsystems, the view, the controller and the model. Several dependencies exist between these three major subsystems and as such, there are several interfaces to simplify and improve the interactions between these subsystems.

3.1. Game View Subsystem

This subsystem is responsible for the user's view, it is dependent on the controller subsystem and provides the following services

- Display Update Service: This service renders the updated model data to the user. It is dependent on the data provided by the controller on request.
- Update Retrieval Service: This service requests for and retrieves a response on the data of the systems model.

3.2. Controller Subsystem

This subsystem is dependent on the game view subsystem for specific requests and the Model Subsystem for data needed to fulfill requests coming from the game view subsystem. It provides several interfaces for controlling the game models. Controller services provide the following services.

- Menu Manager: This manages the different menu items and their individual functionality.
- Enemy Manager: This manages the creation and update of the different enemy types and their properties.
- Player Manager: This manages the properties and player.
- Projectile Manager: This manages the creation and state of the various projectile types for the different entities utilizing them.

3.3 Model Subsystem

This subsystem is dependent on the controller. It provides the needed game data required by the controller in a well organized structure which is further utilized to complete the game view subsystem's request. It also features several interfaces including:

- Arts model: This provides the texture to load for a given entity
- Enemy model: This provides the different types of enemy objects used by the controller
- Projectile model: This provides the different types projectiles available in the Matrix system
- Colliadable model: This provides a shared interface of objects that are colliadable.
- Score model: This keeps track of the score state of the player entity
- Player model: This is the player object.

Model Subsystem

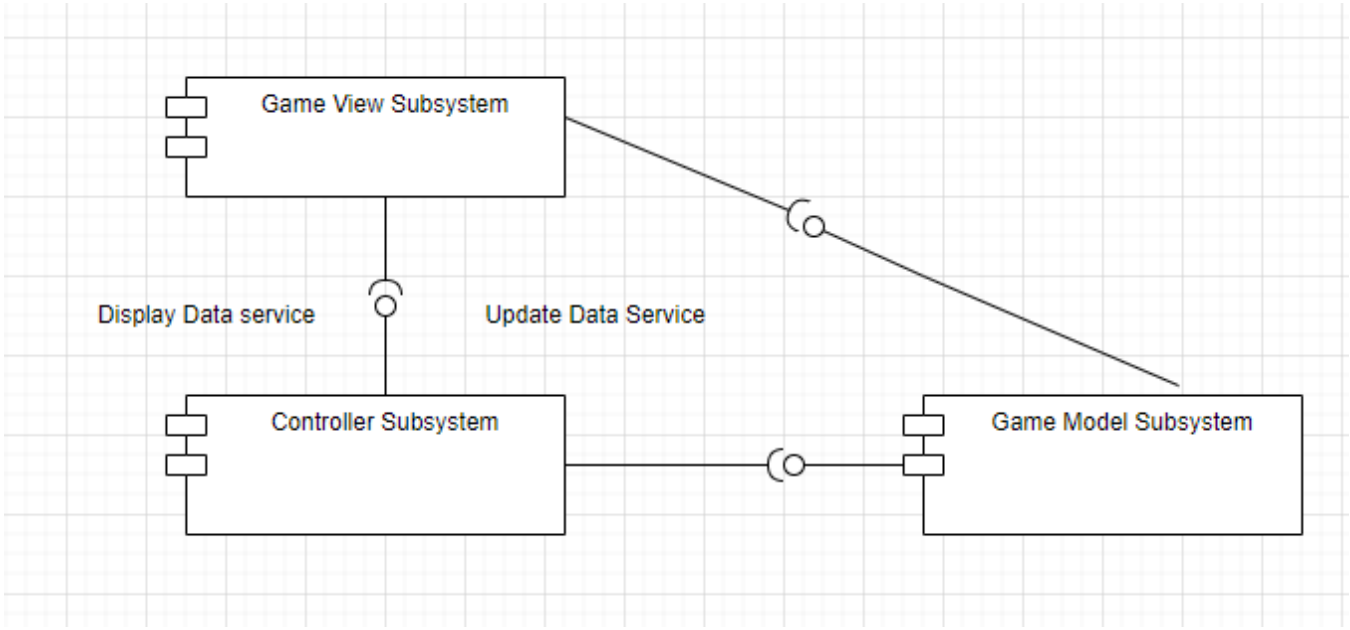

Figure 2-12