COMP 3004B

#3 – System Architecture and Design

Team name: Apple Pie

Project name: Stone

Team member:

Chupeng Shen 101079598

Junren Jiang 100999408

Zac Zhuang 101068659

Xiannan Chen 100995076

Architectural Description

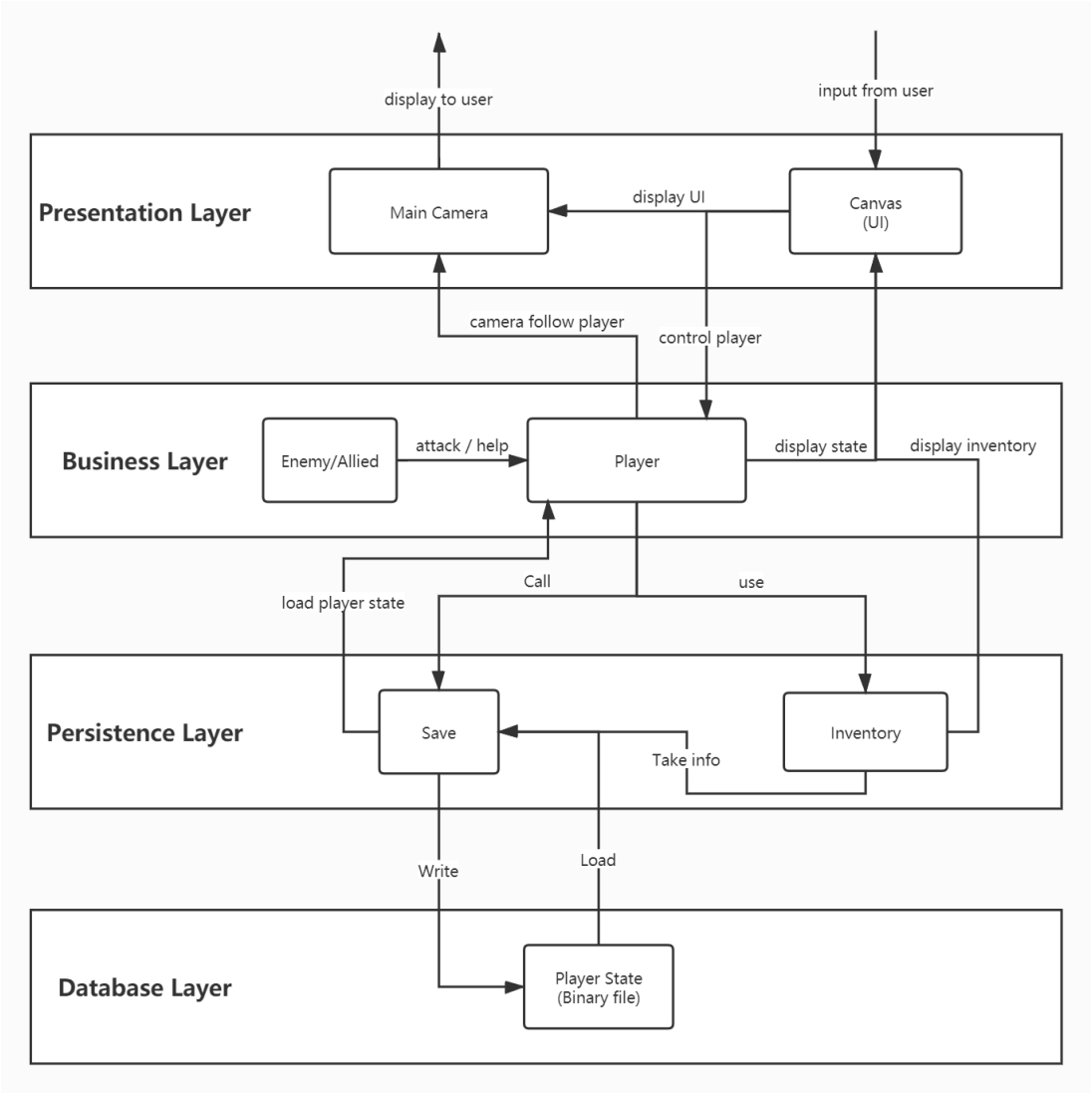

Figure 1. Improved Layered Architecture

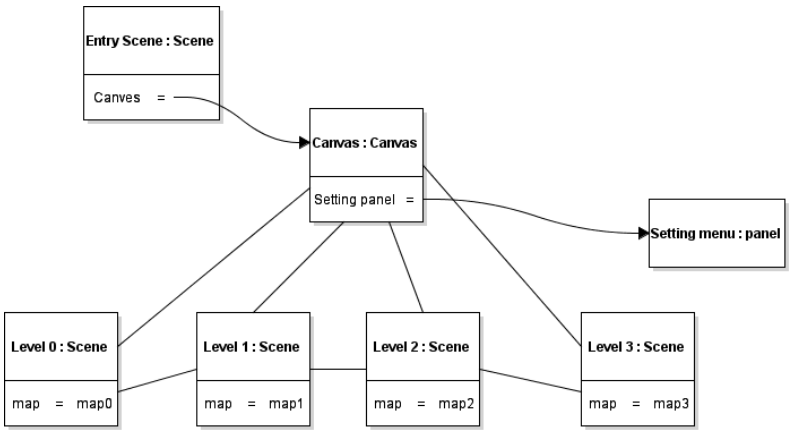

Figure 2. Basic Object-Oriented Design

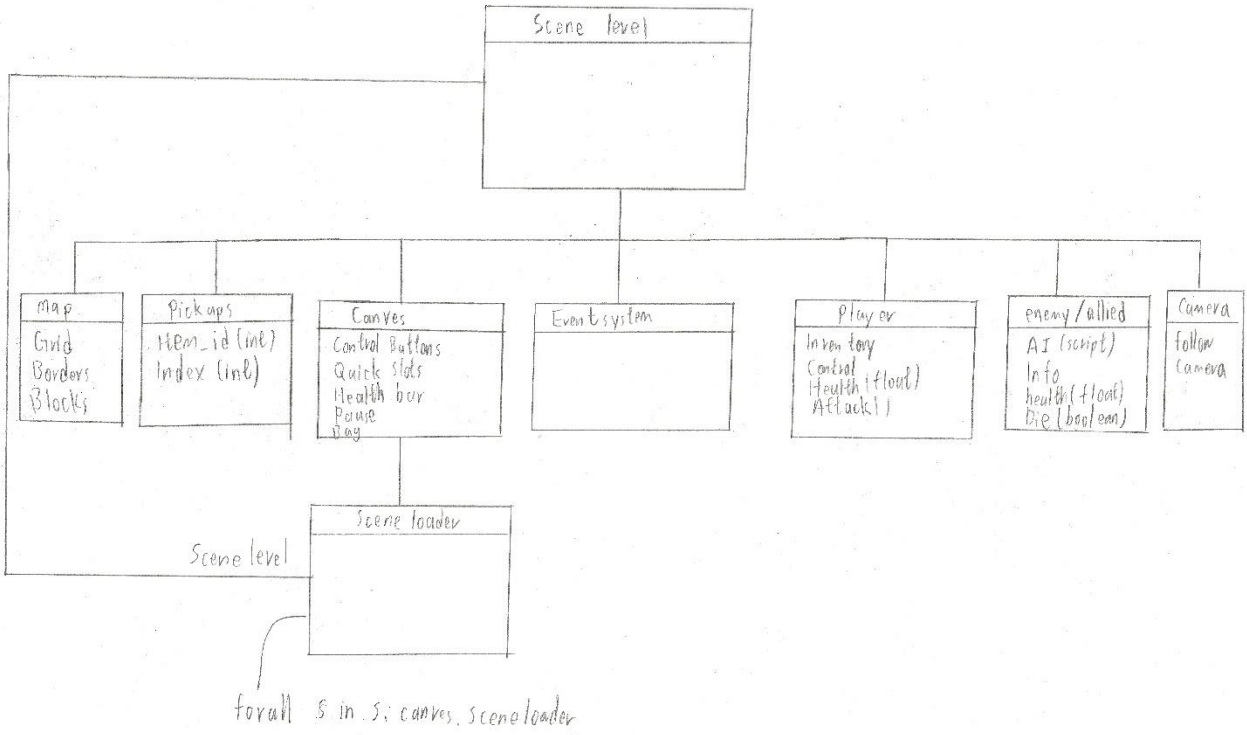

Figure 3. Composite Pattern

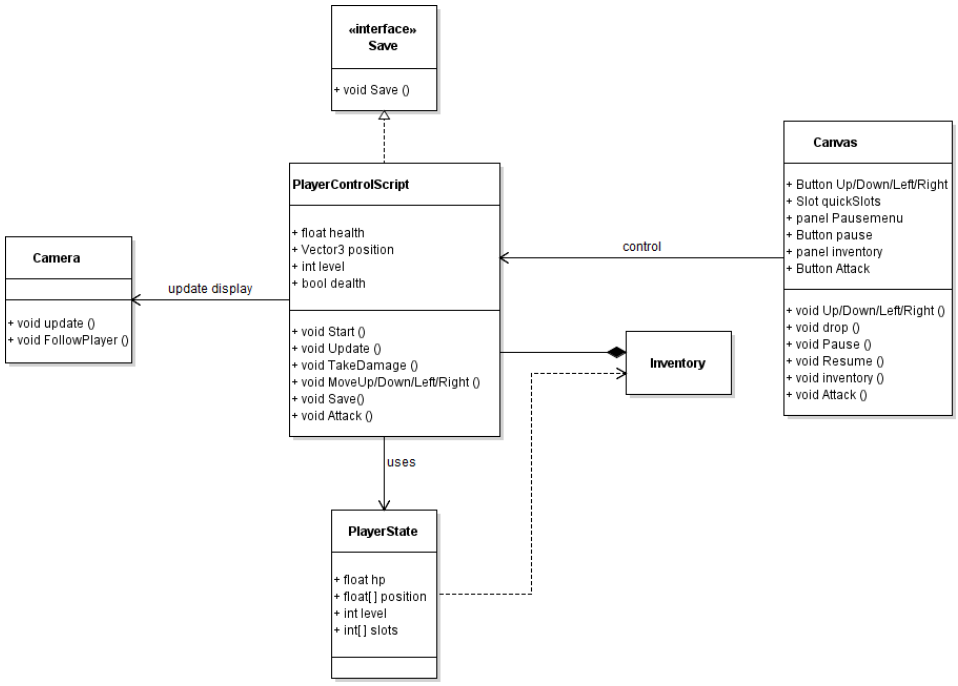

Figure 4. MVC Pattern

The project of this assignment is improved based on the given Layer Architecture (Figure 1). The main purpose is to show the framework of our entire project for the convenience of subsequent reference and editing. As the most common architecture style, each layer of the layered architecture has a clear division of labor. As long as you figure out which layer (feedback layer, data layer, etc.) you want to write and then connect with the interface during the development process, you can easily determine the logic between the layers without much details of other layers which affect the development of the current layer. This is also the architecture that best represents the development of ideas in the Unity development process. Our architectural design can also use the object-oriented design style, because the high degree of freedom of Unity can help us easily achieve class reuse, which can be implemented in various places where needed.

In the presentation layer, users can get all the information they want to know on the output of the camera. Among them, the user interface is attached to the camera and is responsible for receiving all available requirements of the user and passing the calling function to the player class. Both players and AI belong to the business layer. The player is the core of the program, and almost all other classes are related to the player. When the user is in the game, the camera will always follow the player's actions and record various animations, while the player returns status information to the UI. Players use the inventory system to store personal information. When saving data, all data of the inventory system will be retained in the local database as important parameters. Enemies are independent objects and are handled only logically.

In the persistence layer, the save system uses the file as the program's database and loads the previously saved state in the database when the file is read. Inventory is one of the saved parameters, which is called as the personal information to the required place when saving and reading. The database contains the abstract data of the program, including the player's position, status, and AI information.

While entering a game, the user's first visual interface is to enter the scene. All canvases that belong to the scene make the function call to a specific canvas, for example, the setting interface belongs to the canvas, starting the game and loading the game to enter the main scene of the program: the level. Each level is linked using a function call. Players can use it to move between scenes, and they can return to the entering scene at any time in the level. Levels differ in the map design of the game (including enemies, allies, grids, colliders, etc.).

Using the component interaction in the MVC pattern, we use the composite design and MVC design (Figure 3&4) to explain the design direction inside the program. The reason for using a composite structure diagram is that it is suitable for representing the interrelationship between a class containing an inner class and an external interface. For example, in our design ideas, the settings interface can be called up during the game and the start interface. Any changes to the settings options are synced to wherever they are brought up. We need to store the last saved data of the player locally for use on the next load, so how to associate the saved data with the interface is our biggest challenge in editing this model.

Figure 3 and 4 are detailed design patterns of the level scene. The scene loader under the canvas will load other scenes. Relationships at each level are synthesized in this way. The canvas acts as a controller to obtain input, including functional instructions such as move, attack, and pause. The player control exists as the core of the model, it is responsible for handling the user's control over the player. The player state is an object file that is imported and exported from the database when it needs to be saved and be loaded. The camera always follows the player and is responsible for feedback on the player's animation, personal information, and other intuitive things.

System Design

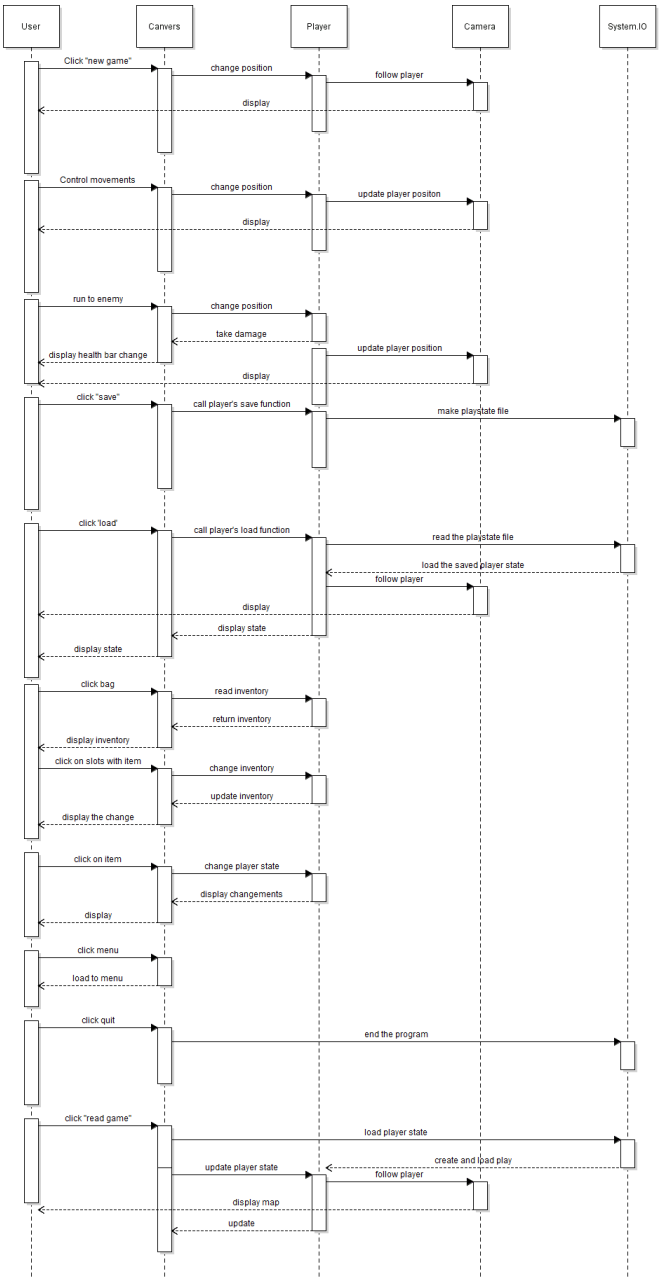

Figure 5. Sequence Diagram

Figure 6. External Visible

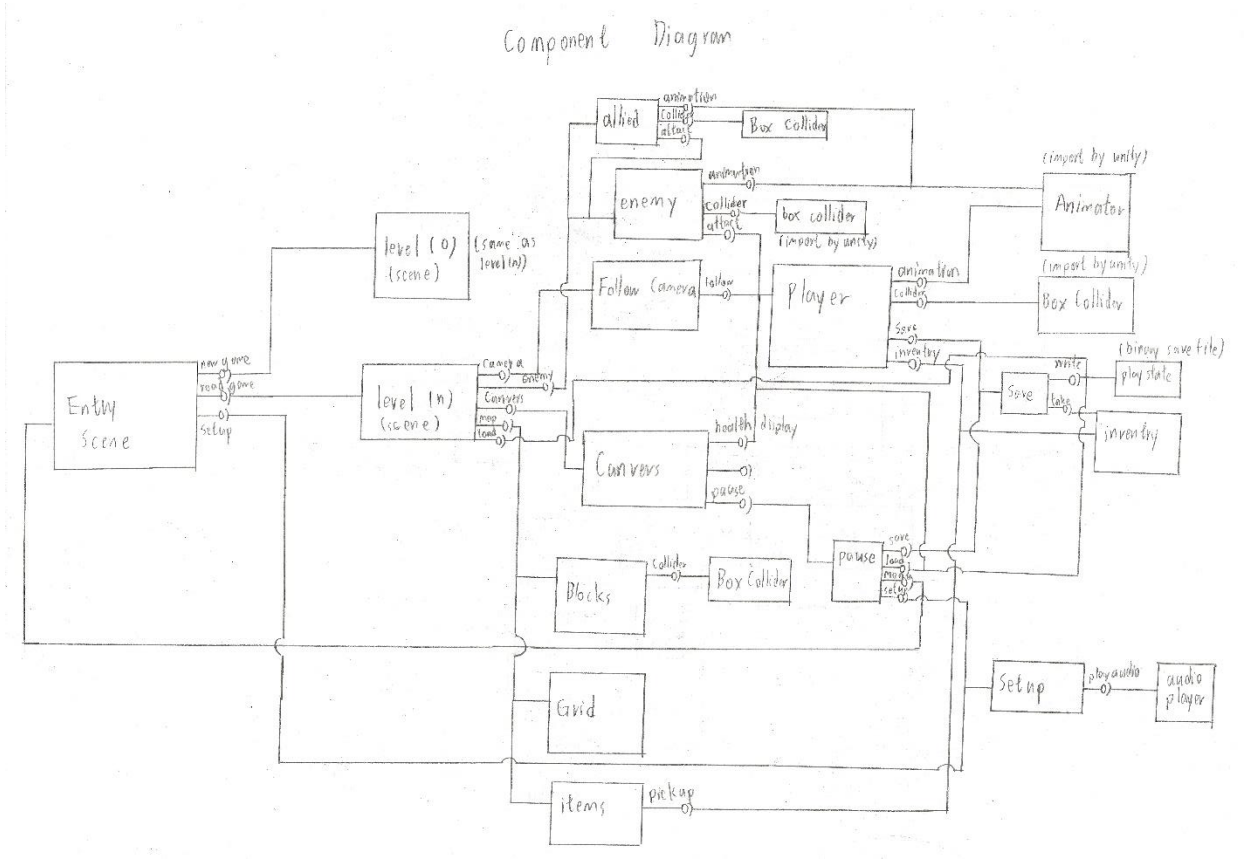

Figure 7. Component Diagram

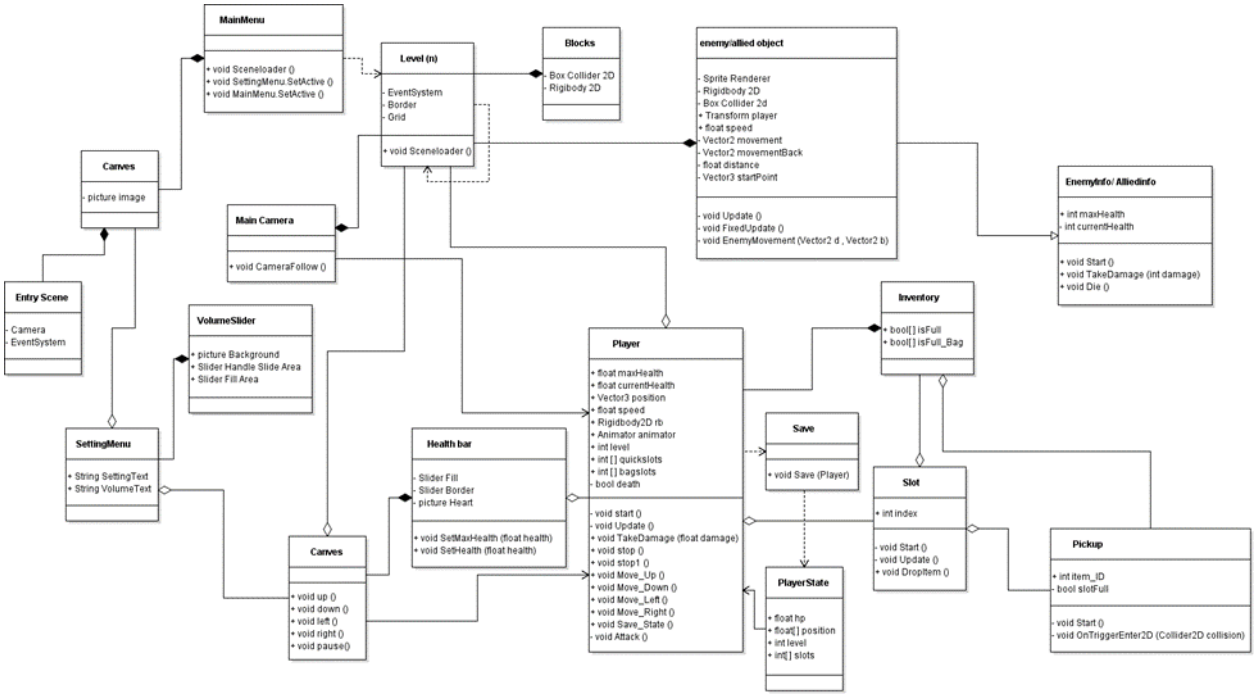

Figure 8. Class Diagram

The project that we choose to design is a mobile game. It is a traditional RPG (Role-Playing Game) named “Stone,” based on the Android system (version 4.1 or higher). Since the programming is based on C# language on Unity platform, due to the clear design structure and convenient platform, implementing subsets or other functions is easier than other languages. For design patterns, we are using MVC design pattern (Model-View-Controller) and composite design pattern for our structural design.

As shown in the component diagram, the relationship between components is clearly demonstrated. Starting from the entry scene (External visible interface 1), the user may choose from four different options: choose a new game to enter level 0; choose load game to enter level n, the scene where the user left the last time they were executing the program; choose setup to enter a panel where the user can adjust the sound level; or choose quit to exit from the program. In all scenes, there are five main components followed by different smaller components. Also, our mobile game will be on the client side. All objects will be saved as local files, which means our game will not be connected to a server.

The first main component is the camera, the camera is set to follow the character’s movement that the user controls. The player component follows the camera component, which has four different components: animator, box collider, saving function, and an inventory system. Both animator and box collider have been imported by Unity platform. The saving function is to save the current progress of the user, which can also be accessed from the pause menu, which will be introduced later. The inventory system is the backpack of the character, where the item that has been picked up by the character will be stored.

The second main component is AI (Artificial intelligence), including two types of AI: allied ai and enemy AI. They have the same components under their class, which are animation, colliding, and attack action. Like I have mentioned, the animation and colliding are handled by the Unity platform’s animator and box collider. On the one hand, the attack action is connected to the health system of the user, and it affects the deduction and the recovery of the character’s health point. On the other hand, it is also connected to the health display on the canvas component since the health bar on the canvas will be the most straight-forward method to show the user their character’s health condition.

The third main component is the canvas (External visible interface 2). Canvas component is different from other components in some way; the display under the canvas component is on different layers, yet it can be displayed at the same time. The health bar display will be at the top left corner of the canvas, following the character’s movement (the camera), it also connects to the health status of the character, displaying the current health status of the character. The second component under canvas is the inventory system, which I have discussed previously. The third component is the pausing menu when the pausing menu has been activated, the game will be paused, and a menu will pop up on the canvas, it will show several options for the user, the user may save, load, return to the menu, or to change the volume level.

The fourth main component is the map. Map is the biggest component of all five components. It contains three components: blocks, grid, and items. The Grid component is for setting the map into small individual boxes, makes map editing move convenient and efficient. The blocks are obstacles on the map, also imported by the box collider function by the Unity platform. The items are objects that can be used by the player or to be stored in the inventory for later use.

From the perspective of our software design architecture, our program follows a loose coupling approach while our project development progresses, which ensures any related requirement changes later. Secondly, software development follows an agile development model. Lastly, we have achieved our basic

design purpose in our first demonstration. We have demonstrated our basic functions.
For the other design pattern that we are using, MVC pattern is what we have applied (figure 7). In this pattern, we have Player State, inventory, and the Player Control Script as the Model object, camera as the View object, and Canvas as the Controller object.

Starting from the Model object, it contains Player State, inventory, and the Player Control Script, the Player Control Script is the core of model object. The Player state contains the basic status of the character that the user is controlling, such as health point, character's current location in the scene, which level the character is in, and the inventory slots in the character's backpack. Furthermore, the relationship among the components in the Model object is that when a user is saving their game progress, the current player status will be saved into player state component as an object file. For the View object, we have the camera on the current canvas and the character, updating every movement and action that the user is making, feeding back all the images back to the user. The Control object is done by the canvas component, all the options that are available for the user to choose is on the current canvas that the user is on, which means the canvas component is where the game takes all the user input and interaction with the level.

As we have demonstrated in the sequence diagram, it is closely connected to the MVC design pattern. When the user chooses to enter a new game, the canvas will be changed to a new level, activating the camera to follow the movement of the player, every time the user makes a movement, the canvas will refresh and update the new position of the player and return to the stage where the system wait for the user to make their next move, this process will keep happening recursively. When the player approaches an enemy, the player will enter an action phase, where player's health will be deducted if being attacked by the enemy, the result will be reflected on the health bar on the canvas instantly and continuously. If the save function has been called, a new binary player state file will be created and stored. Later, when the load function has been called, the system will read the binary file stored in the player state and load it to the last position that the player has been, and make the camera follows, returning to the phase where all movement will be returned recursively. As for the bag of player, the process is the system will read what is in the inventory and return it to the canvas, display it to the player. Then, when the player interacts with the item in the inventory, a change will be made and instantly update the inventory, if a health recovery item has been clicked, the player state will also be updated and reflected through the health bar on the canvas.

As we have mentioned in the architectural section, the presentation layer is the most direct interface that the player will interact with when the user enters the game. It contains the canvas(UI), which shows the actions that the user is allowed to take and the main camera, which will be following the character throughout the whole game progress. When the user makes an action, the command will be delivered to the next layer, which is the business layer. On the one hand, the character will respond to the action that the user has made and interact with the level, including obstacle collisions and enemy or allied AI. On the other hand, the user can choose to open the inventory interface to display the current status and inventory of the character. When user choose to save their progress, the command will be forwarded to the persistence layer, where stores the save function and inventory data of the character, the save function will save all the information that is connected to the player, such as the current inventory and health point, then the saved data will be written into binary files and be stored in the player state file, which is located in the database layer.

Assignment of tasks for each team member:

Zac Zhuang : Health bar display and damage taken. Map design, blocks on the map with box collider, map border. Editing and embellishing picture. Manage the prefabs. Analysis description of project structure.

Junren Jiang : Movement and the animation of movement. Attack and the animation of attack. The hit box for attack. Setting panel and its configuration. Exit the program. Analysis layer architectural style, composite pattern.

Xiannan Chen : Movement button. Inventory system that includes slots, player inventory pickup event handler, items and inventory sorting. Enemy AI that includes enemy's attack, health and behaviour logic. Optimize the animation.

Chupeng Shen : Camera script. Player's box collider edits. Pause menu with pause function. Main menu's button. Save and loading for play object. Scene loader. Inventory panel. Diagrams for this project. Assigning tasks for group member. Github's management.