Architecture

Cohort 1 Group 2

Yusuf Almahmeed
Jane Boag
Charmaine Chamapiwa
Charlie Coyne
Anupam Cunden
Bertie Kerry
Leo Xu-Latti

This page does not count toward the total page number

1: Introduction

The architecture of this project evolved throughout the course of its development. First was a theoretical architecture designed based off of the initial impressions we had from the product brief and meeting with our client. Then as we began to work on the project, we started to finalise our design into the working architecture we are using.

To properly explain the architecture, and its changes over time, we will be using UML diagrams. UML is a standardised visual language used to represent relationships between components of a program, amongst other things. Primarily we will be using this to show the relationship between Controller, Model and View components of the game; The various classes that compose those components; and at the most fine looking at individual functions when relevant.

Initially these UML diagrams were sketched out on paper in an iterative process, so that we could visualise various ideas, however for the sake of clarity these diagrams have been recreated with the online tool draw.io, which has tools specifically designed for UML diagrams.

2: Initial Architecture

2.1: Brief and Requirements

First we were given the project brief. It was a short document that did not include a massive amount of concrete details on the specifics of the project. It did however give the outline of how the project should be ‘shaped’ so to speak.

We have a player character, one who will be navigating a map and collecting upgrades. Their success will be measured by how quickly they navigate the maze as well as a score counter. This already gave us a few ideas of how we should go about structuring the project from a technical perspective.

First we would likely need a player class, both to handle movement and navigation, as well as so that the speed and other properties of the player avatar could be modified by in-game events. To this end an event class would also likely be useful. This class would define the skeleton of what an event was, an individual sub groups of events, such as positive, negative, and secret, events would then inherit from this parent class.Of course we would also require classes to handle more generic features, such as the map and camera.

2.2: Initial architecture concepts

At the beginning, we evaluated two architectural concepts which were Entity-Component-Systems (ECS) and Controller Model View (CMV). Both are effective structures for game development, but we needed to decide which one best complemented our project.

Many popular platforms, such as Unity and LibGDX, use ECS as their architectural foundation. ECS is divided into three main parts, Entities include game objects like the player or any obstacles. The component section is made up of attributes such as position or velocity. Finally, the systems section stores game logic and operates on groups of components. Overall, this concept improves modularity and performance; however, it does make it more complex which can make it less suitable for smaller projects.

In contrast, CMV also consists of three sections but it has a different structure. The three sections include the player interaction and input (controller), the game's logic (model), and the user-interface (view). It is important to note that the Model is independent of the other components which makes it easier to modify and maintain. For instance, if we wanted to change the UI design, we would not have to make any changes in the game's logic. This concept of separation well supports smaller games like ours due to its simplicity and adaptability.

2.3: Narrowing down the design

After evaluating both ECS and CMV, our team decided to use the CMV architecture. We found that it best fits the project because of its straightforward approach and flexibility. Such features of CMV allow for frequent change which also aligns with our agile software engineering method.

As previously mentioned, the separation of components (controller, model, and view) is a key element of CMV which makes it useful for our project. This concept ensures that each part of the system is responsible for a specific function. In turn, the separation allows different components to be modified and tested without unintentionally affecting something else. This also made it easier for members of the team to work on different sections of the game without affecting each other's work.

Another major advantage CMV offers is its simplicity and reusability. Since each component is separated into different classes, they can be reused and implemented in future versions of the game with minimal changes. The clear classes make the code easier to read and debug. This is especially useful when another team will be taking over our project. Overall, the CMV architecture allowed our team to work in a clear and efficient manner, which supported collaboration and minimised development issues.

3: Working Architecture

3.1: CMV Overview

The CMV model separates the program into three sections: controller, model and view.
In our program the controller consists of GameControler; the model component consists of the classes: GameState, Player and MapModel; and the view component consists of GameRenderer.

A full UML diagram can be found on the GitHub page under figure1.

3.2: Class Overview

GameState 3.2.1

Keeps track of the game's state, fairly self explanatory. Specifically tracks if the game is paused, over, won or has started via a set of booleans. Other classes can call on this information as well as needed. It only has two functions, one to update the values, and one to reset them.

MapModel 3.2.2

The map model holds all the data relating to the game's maps, such as collisions, teleport zones, win zones and the visuals of the maps. It has two main sets of functions, a set of private functions that load the various zones onto the map, as well as a set of public functions that check to see if the player is within any of the zones.

Player 3.2.3

The play class is by far the largest class in the entire game. This holds all information about the player character, which primarily consists of the player's speed and the animations for the avatar itself. Its functions consist primarily of methods to alter the players animations, alongside a function to change the speed of the player, should they interact with a speed boost.

TeleportZone 3.2.4

TelportZone is a very small class that defines the behaviour of teleport zones that allow the player to pass through doors.

Controller 3.2.5

The controller class, perhaps unsurprisingly, handles the controls! It holds the gamestate, player, renderer and map data. It has several functions, once to listen to handle input, a

couple to handle updating the game-logic and camera, and then a function to restart the game.

GameRenderer 3.2.6

This function deals with rendering things onto the screen itself. As such it holds all of the game assets and various ui elements, fonts, etc. Importantly it also holds both cameras. One for the player, one for the ui. It only has two functions, one to render and asset, on to dispose of an asset.

Powerup 3.2.7

This is a small class that constructs the powerups scattered around the game and assigns one of two types with a string - speed boost or time increase.

Hindrance 3.2.8

This is another small class that constructs the hindrances - penalties that reduce the time the player has left.

Key 3.2.9

Constructs the key for a hidden locked door event.

Score 3.2.10

Defines the players score and uses a few methods to update score continuously according to the timer, and decrease or increase it according to interaction with hindrances or powerups.