Final Project of Fundamental Objected-Oriented Programming

陳明信, 汪昊新, 林耘平, 王風意
B08902020, B08902047, B08902075, B08902119

July 1, 2021

1 Introduction

CSIE OS C- is an old school 2-D fighting game. It's the simplest yet the most nostalgic version of fighting games. The goal of this final project is to remind us of the genuine happiness in our childhood by playing around with this arcade game.

As long as you start playing the game, you'll immediately fall in love with the excitement and the pleasure that ***CSIE OS C-*** brings to you. Swinging swords and pulling triggers at the zombies are so easy and satisfying that we believe everyone will enjoy it, regardless of any of their previous gaming experiences.

2 How to Play Our Game?

First, open your terminal and change directory to our `FOOP-FINAL` directory. Entering `make`, the compilation will start and the game will show up on your screen just after a few seconds.

In the main page of our game, there is a Tutorial button, click it to learn how to control the players in the game. Below the tutorial button are three buttons that guide us to three different worlds in which we can go on an adventure.

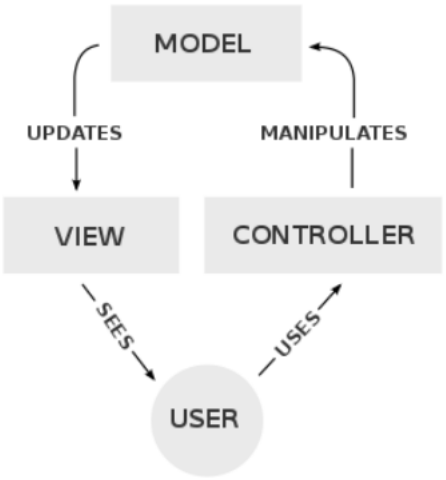
After selecting the world, it's time to choose the character that catches your eyes the most. Clicking the character card, we can start the game immediately with the chosen character. Note that every character has different skill set. For instance, some characters may not have the shooting skill.

In this game, you can move your character in three directions: right, left or jump. Whenever you see a zombie coming to you, feel free to attack it by swinging swords, kicking legs or pulling the trigger! While you kill the zombie coming towards you one after another, the upper-right progress bar would show your the current progress in the game accordingly. After the progress bar runs to the end, the screen would turn dark and you'll be facing the

boss! Defeat it and you will see the winning animation with a button to click to end the game!

3 Relations Between the Classes

First of all, we design ***CSIE OS C-*** by adopting the well known MVC software design pattern. MVC is the abbreviation of the names of three basic components: model, view and controller. Below we'll demonstrate the classes that we designed, how they interact with other classes, and which part they belong to in the MVC design pattern.



Besides the MVC design pattern, we use finite state machines to keep track of the current state and the state transitions of each sprite.

3.1 Model

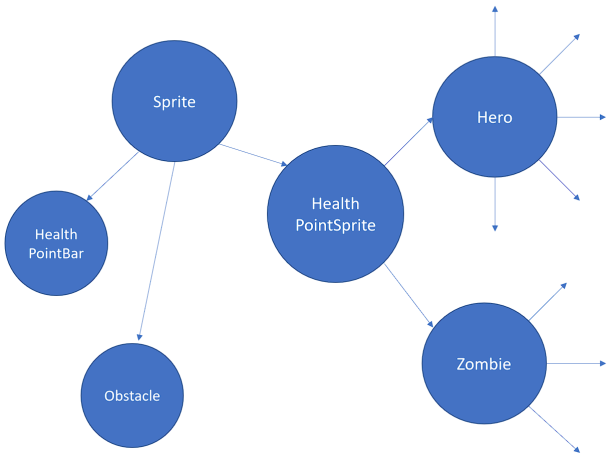
Every object in this game is built upon the base class `Sprite`, including the characters controlled by us, the zombies that try to attack us and even the obstacles that appear on the map.

`Sprite` has all the common attributes of characters, zombies and obstacles, such as locations, aliveness, etc. It also has some functions that will be called by the **controller** module at every tick, like `render()` and `update()`.

`HealthPointSprite` is the subclass of `Sprite`. Any object that needs a health bar is built upon this. Both `Hero` and `Zombie` are inherited from `HealthPointSprite`. But the story doesn't end here.

All six characters that can be chosen from the character selection page are subclasses of `Hero`, while `MaleZombie`, `FemaleZombie` and `Boss` are inherited from `Zombie`.

Below is brief graph of the relationships between these classes.



3.2 View

`Gameview` is a subclass of `Jframe`, which is a class of the `javax.swing` package. `GameLoop` calls it every dozen of milliseconds to render the game scene according to the different scenarios that happen during the game play.

For instance, it renders out the homepage and the character selection page before we enter the game. During the game play, it shows every detail of the game on the screen, including the characters and their movements, the map that the characters are standing at, and even the flying bullets in the air.

Rendering out the maps, the homepage, the pause page or the character selection page is rather easy. Basically, what it does is only print out the given images. However, rendering the motions of each character is a whole different story.

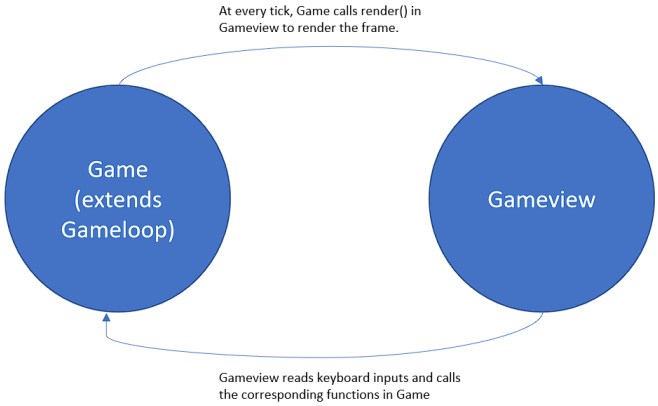
According to the MVC design pattern, `Gameview` is not in charge of the states and the motions of the characters. Thus, instead of letting `Gameview` keep track of every detail of the game, `Gameview` assign the rendering job to the designated classes and tell them to render themselves. The two components, `view` and `model`, must work together to render out the whole game!

3.3 Controller

Another job that `Gameview` is responsible of is listening to the keyboard and mouse inputs. Note that this is a separate thread from `Game`. `Game`, which inherits `GameLoop`, loops every dozen of milliseconds to update the game and calls the methods inside `Gameview` to render the scene. However, there is another independent thread of Gameview that keeps listening to the input signals.

Whenever it receives the signals, it calls the according functions in `Game`, such as `Game.move()`, `Game.jump()` to control the character's motion. It then calls the corresponding functions in the `Model` module to make real effect on the characters.

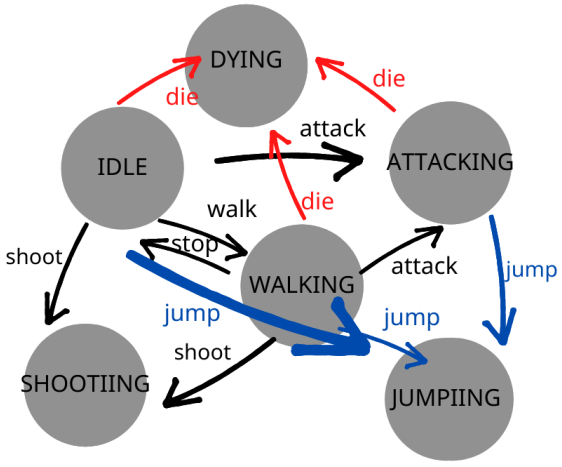
Below is a brief graph of how **View** and **Controller** collaborate.



3.4 Finite State Machine

`State` is a simple interface with update and render two functions. `Sequence` contains a list of extended states class, and implements the `State`, using `update()` to choose the next image in the given folder. `ImageState` is a class implement `State` which would render the image.

`FiniteStateMachine` is used in all the sprite-kind class, including `hero` and `zombie` which extends `HealthPointSprite`. By calling `fsm.addtransition(from(State old-state).when(Event e).to(State new-state))` in `HealthPointSprite`, we first set relations between the given two states with one-way transition. Each State is a `WaitingPerFrame` which implements `State`, which can keep the same image for given numbers of loop. Then, uses `fsm.trigger(Event e)` to transfer among each type of states. Below is a brief graph of how our finite state machine works.



3.5 States

All the classes in this package inherit `Sequence` or `CyclicSequence` in the package `fsm`. Each state would be given a sequence of `ImageState`, which would be updated and render sequential image to animate the character. `CyclicSequence` would repeatedly render the sprites cycle, like `IDLE`, `WALKING`, `JUMPING` and `DYING`, while `Sequence` would only do the sprites cycle once, like `ATTACKING` and `SHOOTING`.

3.6 World

Objects like hero, zombies, obstacles, background, boss and progress bar are all handled in `World` class. Beside managing all these objects, one important thing that `World` does is randomly generate zombies at the edge of the map.

`World` uses `update()` to update all the items handled in the class. It then called `update()` of each sprite. We also add gravity to make it look more lifelike, and move all the items in order to create a moving background.

3.7 Homepage, RoleSelect, Pause Page, Tutor Page

These four classes, which are all subclasses of `JPanel`, are responsible for different scenes other than the main gaming one. The program starts at `Homepage`, where it offers a `Tutor Page` and three buttons to let players choose the map in which the character would be placed in. Then, players can select the character in `RoleSelect`. After the game starts, players can pause the game by pressing `esc` and `Pause Page` will pop out immediately. They can resume, restart or quit the game.

The components are mostly arranged by `LayoutManager`, so when the window resizes, the layout won't be distorted drastically.

When `MouseEvent` in components happens, it will change the state of the page and the event will be dispatched to `Canvas`. Then the event handler will call `clickButton` in `game` to check the state changes to control the game flow and page visibility.

3.8 Hero

`Hero` is a subclass of `HealthPointSprite` with a finite state machine. The animation can be rendered with the help of `FSM`. The character can be rendered in any given sizes as well. All the human-controlled classes are subclasses of `Hero`, some of them with a shooting skill has an inner-class `bullet`.

3.9 Zombie

`Zombie` is also a subclass of `HealthPointSprite`. Actually, `Hero` and `Zombie` share many attributes and methods. The main difference between `Hero` and `Zombie` is the way of control-

ling. Human players control `Hero` by keyboard inputs, while an AI controls all the motions of `Zombie`. AI's strategy is simple. It basically runs toward the `Hero` and tries its best to attack `Hero`.

3.10 Obstacle

Unlike zombie and hero, `Obstacle` inherited sprites, and needs different collision handler since it won't move and should also deals with the gravity.

3.11 Background

We refer to Super-Mario's moving background, which means the background moves with the hero, but doesn't move if it tries to go backward. It is not feasible to find an infinite long background image, we let image like grass, bricks and cloud move the opposite direction as hero does. While `Background` class is only for this target, it won't collide with any sprite.

4 Advantages and Disadvantages of Our Design

4.1 Advantages

1. **MVC**
   This design pattern helps us separate jobs to different threads and improve the effectiveness of our software development.

2. **Extend All Objects from `Sprite`**
   There are plenty of benefits of extending all objects from `Sprite`. For instance, the speed of development is fast. Adding an additional object requires fewer lines of code.

3. **Finite State Machine**
   For characters that has multiple states, using a finite state machine to make state transition is the simplest and the most elegant way to maintain the state information.

4. **Extend All Selection Pages from `JPanel`**
   We can customize the layout and components such as `JButton`, `JLabels` on every page, and add different mouselistener on each page's child component.

5. **Open Closed Principle**
   We can easily extend more hero and enemy.

4.2 Disadvantages

1. **Spaghetti Programming**
   We didn't plan every detail well before programming. Many lines of code were added

to the code base without being carefully examined beforehand.

2. **Repetitive Work**
   In order to avoid repetitive works, we could further reduce the total lines of code and simplify the dependencies among the classes if we refactor our code better.

5 Other Packages We've Used

1. java.awt.*
2. javax.swing.*

6 Responsibilities of the Team Members

6.1 陳明信 B08902020

1. Design the state/method `Shoot()` for `HealthPointSprite`.
2. Design and implement `Boss`, including the presence, motions and dying animation.
3. world.move() and progress bar
4. moving background

6.2 汪昊新 B08902047

1. Game page layout management(Homepage, Pause Menu, Role Selection, Game Tutorial)
2. Mouse event detection and damage number display during attack
3. Gameflow control with regard to page status.
4. world themes selection system for the 3 scenes.

6.3 林耘平 B08902075

1. Design the state/method `Jump()` for `HealthPointSprite`.
2. Design and implement Zombie, including their presence, motions and dying animation.
3. Design the earlier version of `Pausepage`.
4. Write the report.

6.4 王風意 B08902119

1. interaction between sprites and obstacles
2. collision between Hero and Zombies
3. implements miscellaneous heros that extends the `Hero` class
4. refine the `Jumping` state system to function normally.