FiOnaOPπcman

> Fundamental Object Oriented Programming (CSIE 1214) — Final Project Report

> We organize the report in our manner. To let the TAs grade more easily, we mark a "⭐" in the section header of each necessary item.

⭐ Team Members & Work Contribution

- Team FiOnaOPπcman

Student ID	Name	Contribution
B06902001	陳義榮	View, Controller, State
B06902029	裴梧鈞	Map, Weapon, Game Loop
B06902059	謝宜儒	Renderer, Props, Pacman
B06902067	許育銘	Weapon, State, Game Loop

⭐ How to play our games?

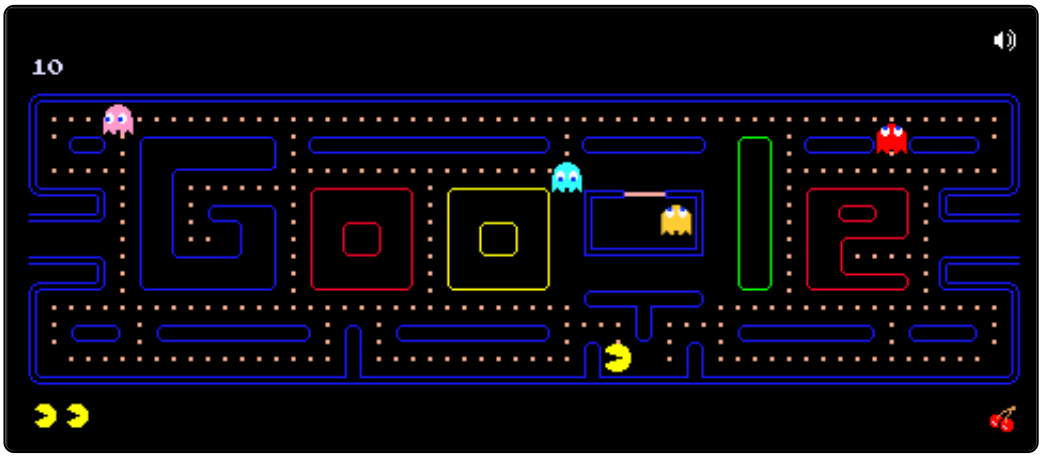
- We've written a `Makefile` and you can
  1. compile the game using `make`, and
  2. run the game using `make run` (1 manual player) or `make run2p` (2 manual players)

⭐ Java Environment

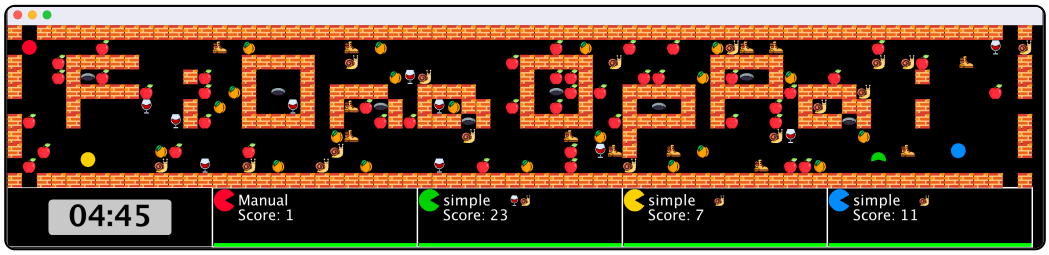
- Java SE 11 (the same as the one we used in previous homeworks)
- Other packages: none

Introduction to our Games

Our game is extended based on a classic game – Pacman. In the original game, the (only) *Pacman* tries to eat as many points on the ground as possible while escaping from the monsters' chasing at the same time. When the pacman eats a special big point, the Pacman will enter a special attack mode and it's able to attack the monsters. The game ends either when the pacman eats all the points on the ground, or the pacman is caught by a monster. The figure below is a screenshot of the original game.



To extend the game, we first changed it into a multi-player game and removed the monsters. Without the monsters, the game would be less fun since it would be too easy. To address this issue, we added some weapons to our game, enabling the pacmans to attack each other. We also added some props in our game to make the game more complex, including different fruits with more points, boots that makes a pacman walk faster, and Prof. Lin's favorite, wine, that makes a pacman dizzy.



⭐ Software Design

Before hitting the first keystroke in our repo, we spent two afternoons discussing the software design. We followed the same concept – the Open-Closed Principle to design each component of our game.

General

We followed the MVC architecture to design our game. The game is mainly splitted into three parts

1. Model – controlls all the game logics.
2. View – displays the game on the screen.
3. Controller – receives signals from the user and send it to the Model.

With Java's easy-to-use concurrent APIs, the Model and the View are able to run in their own threads, improving the user experience as the FPS is less likely to drop.

Model

Interface

We defined three main interfaces to make our classes well-organized, namely **Locatable**, **Pickable**, and **Tickable**.

- **Locatable**: Everything that can be located on map should implement this interface, including **Pacman**, **Prop**, **Weapon**, etc.
- **Pickable**: Everything that can be picked up should implement this interface, including **Prop** and **Weapon**.
- **Tickable**: Everything that may be updated every tick should implement this interface, including **Pacman**, **Weapon**, and **State**, etc.

Game

A **Game** acts as a container for our game. It has instances of following classes as its class members:

- **View**: renders everything
- **World**: It handles the interactions between **Pacman**, **Prop**, and **Weapon**, and updates what should be rendered.
- **Pacman**: represents a player in the game
- **Timer**: determines when the game should end

World

A **World** consists of a **Map**, several **Pacman**s, and several objects (**Prop**s and **Weapon**s). It deals with the interaction between them and update the states of them. To be specific, each update in the **World** contains the following 5 steps:

1. Pre-update tasks (implemented as `onRoundBegin()`)
2. Each pacman decides its action
3. Finalizes pacmans' actions
4. Randomly generates props & weapons on map
5. Post-update tasks (implemented as `onRoundEnd()`)

Map

A **Map** consists of $H \times W$ **MapGrid**s, where H is the height and W is the width. The whole **Game** is played on a **Map**.

To support complex maps, we carefully design our **Map** class to make it have the following significant methods:

1. `canPass(Coordinate from, Direction direction);` : determines whether a pacman can go to the next grid when he's currently stepping on *from* going *direction* way.
2. `nextCoordinate(Coordinate from, Direction direction);` : what is the next grid the pacman will go when he's currently stepping on *from* going *direction* way.

The details are implemented in the underlying **MapGrid**s mentioned in the next section. With the two methods, the main game loop does not need to determine the grid the **Pacman** is heading for and whether it can pass the grid.

MapGrid

MapGrid is an abstract class, whose child classes include the following:

- **Road**: a grid on which pacmans can walk arbitrarily
- **Wall**: a grid that acts as an obstacle, which pacmans cannot pass through
- **Highway**: Highways come in pairs. In each pair, pacmans can travel from one **Highway** to the other **Highway**. There's a 120 ticks (2 seconds) cooldown after the highway is used.

Pacman

Each **Pacman** represents a player in the **Game** (which can either be a manual or an AI player). A **Pacman** has instances of following classes as its class members:

- **Coordinate**: represents the location of a pacman on a **Map**
- **Weapon**: can be used to attack another pacman
- **Controller**: controls the pacman's **Action**, e.g. go up/down/left/right or attack
- **State**: A **Pacman** may have several states, each of which affects the pacman's behavior, e.g. a **Drunk** state may force the pacman to move in the opposite direction
- **AttackCallback**, **TakeCallback**, **DeadCallBack**, ..: callbacks that should be executed when a certain event occurs

Weapon

A weapon is first generated on a **Map**. Then, it may be picked up and used by a pacman. **Weapon** is an abstract class, whose child classes include the following:

- **HarmingWeapon**
  - **BoxingGlove**, **Sword**, **Spear**: When a pacman is hit by these weapons, its HP would be reduced.
- **StateChangingWeapon**
  - **Explosion**: When a pacman is hit by this weapon, a **Stunned** state would be added to its states.

Prop

A prop is first generated on a **Map**. Then, it may be picked by a pacman. Once the prop is picked up, it will take effect on the pacman immediately. **Prop** is an abstract class, whose child classes include the following:

- **PointProp**
  - **SmallPointProp**, **BigPointProp**: When picked up, it will increase the picker's score.
  - **WineProp**: When picked up, it will increase the picker's score and add a **Drunk** state to the picker's states.
- **SpeedProp**

- **SpeedUpProp**, **SlowDownProp**: When picked up, it will either increase or decrease the picker's speed.

State

Each **Pacman** can have several states at the same time. It behaves differently according to the states.

State is an abstract class, whose child classes include the following:

- **Normal**: the default state for a pacman
- **Dead**: When a pacman's HP is reduce to 0, a **Dead** state will be added to the pacman. Within this state, the pacman cannot perform any action.
- **Revive**: After death, a pacman will be given a **Revive** state. Within this state, the pacman is immune to any harm.
- **Drunk**: When a pacman picks up a **Wine**, a **Drunk** state will be added to its states, making the pacman move in the opposite direction.
- **SpeedChange**: When a pacman picks up a **SpeedUpProp** or a **SlowDownProp**, a **SpeedChange** state will be added to its states, making its speed either faster or slower.
- **Stunned**: When a pacman is hit by a **Explosion** weapon, a **Stunned** state will be added to it states, making it unable to take any action.

Callback

Callbacks are interfaces that require "something" to happen at certain moments. For example, a **TakeImageCallback** requires a `onTakeDamage()` function to be implemented concretely, and this function will be executed whenever a pacman takes damage.

Listed below are the callbacks and their corresponding required functions:

- **AttackCallback**: `onAttackBegin()`, this function should be executed before a pacman attacks
- **DeadCallback**: `onDie()`, this function should be executed when a pacman dies
- **DecideCallback**: `onDecide()`, this function should be executed when a pacman decides it action
- **SwitchImageCallback**: `onSwitchImage()`, this function should be executed when a pacman switches the to-be-rendered image
- **TakeDamageCallback**: `onTakeDamage()`, this function should be executed when a pacman takes damage

View

View

View is the assigner of the entire game view. It extends **JFrame** and contains all the **Renderer**s described below. In every game tick, it asks the renderers to render the views. When the game ends, the game result, which ranks the pacmans, will then be rendered.

Renderer

Renderer is an abstract class, whose child classes include the following:

- **PacmanRenderer**: renders a **Pacman**
- **MapRenderer**: renders the **MapGrid**s in a **Map**
- **PropRenderer**: renders a **Prop**
- **WeaponRenderer**: renders a **Weapon**
- **FootPanel**: renders the information (name, score, HP…) of a **Pacman** at the bottom of the window
- **TimePanel**: renders the remaining time at the bottom of the window

Controller

There are three kinds of **Controller**s in total. Each **Pacman** is attached with a **Controller**, which determines a pacman's action whenever a pacman is able to make a decision, i.e., is neither moving nor attacking.

- **KeyboardController**: A **Pacman** with a **KeyboardController** uses inputs from a keyboard to decide its action.
- **SimpleController**: A **Pacman** with a **SimpleController** searches for the nearest props in its neighorhood and goes in that direction.
- **RandomMoveController**: A **Pacman** with a **RandomMoveController** decides its action by random!

Discussion and Feedback

⭐ The advantage of our design

1. Since we followed the OCP by using several callbacks in our game, we can easily add new props, weapons and pacman states to our game. Therefore, it is easy to design new effects on players.
2. We tried our best to follow the MVC design pattern — most important the model part — from other parts. With this, we can transfer our game to other UI systems, such as mobile apps, web pages and even Nintendo Switch.
3. Since a renderer is uniquely associated with an "object" in the game, its only responsibilty is to render a "object" according to its relevant states. Hence, a renderer is always "told" what to render, instead of deciding what to render. If we want to add new elements to the game, we can simply add new corresponding renderers.
4. Developers can add their own controllers. The new controller only has to inherit the base class, and overrides the `decide()` method. Developers can design a very strong AI that beats almost everyone, or a monkey controller that just wanders around.

⭐ The disadvantage of our design

1. To pay tribute to the original game, we designed our game map using "grids" (integer coordinates). With the grid-style design, it's more easily to deal with the elements originated from the pacman game. For example, it's relatively easier to deal with the pacmans' moving, props generation and consumption. However, when dealing with the weapon system, it's hard to determine whether a "hit" occurs. On the

other hand, we are only able to handle the short-range weapons; we cannot properly model the long-range weapons such as guns with bullets or bows with arrows because it's hard to determine whether a bullet hits a pacman. That's the reason why we only designed short-range weapons.

2. In our weapon model, we only determined if the weapon hit the other pacmans in one tick, while the animation takes several ticks. In this way, it's easier to judge if a weapon makes successful attacks, but it causes inconsistency between animation and attack determination. Since currently available weapons' animation are short, the inconsistency doesn't matter. But if we want weapons with longer animation, such as arrows, the consistency would be an important issue. We have to make successful-attack decision for many ticks (along with the animation) without successfully attack the same pacman more than one time. Besides, the attack judgement is done once the owner uses it, thus we can't make idle weapons such as landmines.
3. (It's not related to architecture design) Each props and weapon has different probability of generation. However, the current probability distribution may not be the best. Besides, the balance between different weapons can be improved. If having enough time and given others' feedback (which is hard during the lockdown), we can fine-tune the objects and make the game experience better.