Advanced Programming
Project Report

Yousra Smits
20201267

Aug. 2024

1 Namespaces

The code is seperated into 3 parts, using namespaces to seperate these parts accordingly

1. View namespace:
   (a) Responsible for creating a UI to facilitate interaction with the user and logic of the game
   (b) Provides needed information on the score, lives left and game state in general
   (c) Provides instructions on how to interact with the project in a simple way
2. Singletons namespace:
   (a) Holds the singletons that can be used in both the Logic and View namespace. Made for ease of access
3. Logic namespace:
   (a) The back-end/core of the game. It gets updated on request from the main Game loop
   (b) In charge of simulating the game and providing the View namespace with the needed information to match the back-end with the representation of the game

2 Logic API

The Logic namespace can be used for different representations if you so please. It provides simple instructions and ways to get access to the current state of the game.

The World class is your main access point to the logic of the game and acts as an API. Following instructions can help you with getting the data you need to make your own representation of Pacman:

Function	Description
World(int level)	Creates an instance of the world, based on a given level (*)
Update()	Updates the state of the game, takes no input
GetFullMap()	Get a 2D vector containing data of every tile in the game and the entity that is present
GetUpdates()	Get all the changes that happened in the previous update call in the form of a queue of events (events defined in logic/Event.hpp)
GetOutputData()	Get a struct holding the current scores, lives and if the game is over or not, also holds if the player has won (if they meet certain conditions)
SetPlayerDirection()	Sets the direction of Pacman, seperated from the update so this can happen whenever input gets handled (in case you want to handle input more frequently and update the game less frequently)

3 Design Patterns used

3.1 Model-View-Controller

The model is the level class, it holds all the data of the current level. The controller is a mix of the World class and the Simulator. The World class works like a facade and holds the Camera and the Simulator. The Simulator is the main manipulator of the data while the Camera gets a 'view' over the level when requested. As for the View, that is the Game class, it is in charge of handling input and providing the info to the user.

3.2 Facade

I used the facade design pattern to be able to design the 'API'. It hides the complexity of the Logic namespace to outside classes/users. World is the facade for classes outside the Logic namespace while Simulator is a facade to the World class and holds the complex calculations and processes of how Pacman works and only provides the necessary data to the World class for it to be passed.

It does make a 'God class' out of simulation but this did make development a lot easier as collision checking is done outside of the models (pacman, ghost, etc.) and the models only do simple calculations based on their state. It also allows for more flexibility in case I want to add a new object since I will only have to add onto the simulation and not go to the individual classes in order to be able to tell it what it needs to do with this new object.

3.3 State

States have been used in two places:

1. For the Game View to shift between screens easily and go to new levels.
2. For the dynamic entities within the Logic namespace, since, based on the state they are in, the entities will behave differently

Both of these have their own Statemachine/Statemanagers that holds the current active state, adds and removes states. Adding and removing states are done by states themselves.

3.4 Singleton

I used the singleton design pattern for the Random Number Generator and the Stopwatch since it is best there is only a single object of that class. It helps avoid confusion, for example, when dealing with how much time has passed in the frame. Thanks to getInstance you can just make sure you get the same object that is used elsewhere.

3.5 (Entity) Factory

For the View I made an entityFactory that holds all the entities used in the Game State/Level State. This way they can easily be created, accessed and updated when needed. I did not go for linking the Logic entities with the view entities to create a clear separation. I felt like it was not needed to couple the two since that could create some issues in development when adding or changing a logical entity. It would also make the code less readable and more complex.

3.6 No Observer?

The reason I did not use the Observer Design pattern is because of its complexity. I felt like using a queue to give the needed updates was a bit more easier to implement and maintain. Adding an event only requires a few lines of extra code and doesn't need defining of a new subclass and adding listeners to said event.