Advanced Programming: Gradius

Design Report

Sergio Fenoll
University of Antwerp
2017-2018

Model

The Model holds all the data needed by an Entity to 'exist' in the game world. Attributes like *health* and the *position* are stored in the Model, but not, for example, the *texture*. These things, related to the visual representation, are stored outside the Model.

The Model stores data and handles the game logic of a given Entity. The Model changes state on its own (AI) or based on instructions from the Controller (player) and updates the View when necessasry.

In my project, the Model is (erroneously) implemented with two objects. Because of an initial misunderstanding of the working of MVC, that the Model should only hold game data (the state) and the Controller should adjust it, the Model exists as an Entity class containing a ModelData struct. ModelData is a POD struct, all its members might as well be stored in the Entity class (and the Entity class should be renamed to Model for completeness' sake). This design error is intentionally left in the code to show what I have learned about the MVC design pattern while working on this project.

Some parts of the game logic are handled by the Game class. This class is an extension of the Model, as it holds all the different Entities and handles certain aspects of the game logic, like collision detection. It made more sense to place it here along with more utility-like behaviour (drawing end- and death screens, ...) than separating the behaviour.

View

The View stores information related to the representation of a Model. A texture which represents the Model is stored and drawn to the screen when necessary. The View holds no other data related to the Model. It has no effect on the Model and in fact depends on it to be able to run. The Model, on the other hand, is completely independent from the View and does not require its existence.

Observer Pattern

Altough the Observer pattern is technically implemented (the Model holds a list of Views which are notified of state changes), there is no real purpose for it in my implementation. No Model holds multiple Views, thus there is little need for a list Views. The concept of notifying the View (instead of the View polling the Model for updates) did come in handy, the View isn't coupled to the Model, it only receives it's data whenever it must (re-)draw the Model.

Controller

The Controller handles events generated by the player input and tells the Model to adjust itself based on those events. It has no further effect on the Model, it depends on the Model to be able to run. The Model does not require the Controller to exist.

Singleton Pattern

The Singleton Pattern was implemented using templates following the method described here: https://stackoverflow.com/questions/41328038/singleton-template-as-base-class-in-c

Using this method made sense, the base Singleton class hides the constructor from the user and only allows the user to call getInstance().

Utilities

Stopwatch

To make sure the state changes happen at the same interval on every machine, the Stopwatch utility is used. It sets the framerate to 60 FPS and makes sure wait after every tick until the tick legnth is reached (16.7 nanoseconds for 60 FPS). It is also used to compute the speed of entities, speed is computed by dividing the defined speed (in the config file) by the framerate.

Transformation

The Transformation utility is used to turn pixel coordinates into game coordinates and vice versa. The pixel coordinates are bound by the window size, the game coordinates by the game field (which is 4x3). Game coordinates are used internally to represent the Models, pixel coordinates are used by SFML to draw the Views.