Arno Troch
S0181354
arno.troch@student.uantwerpen.be

Space invaders: design choices

In this report, I will talk about some of the design choices and patterns I used in the Space Invaders project. First of all, let's start with the project's namespaces. There are two main namespaces, which are *entity* and *game*.

The first namespace contains, as you might have guessed, the class structure for entities. There is one main *Entity* class, which is abstract and whom every other class inherits from. This class contains basic member variables which every entity should have, like a position, a dimension and health points. It also has a pure virtual function, which returns a path to the texture that should be used to represent the entity.

The actual entities can be split further into two groups: entities that are able to move, and entities that aren't. The latter ones are concrete subclasses inheriting directly from Entity (in this case, *Shield* is the only 'static' entity). The former ones inherit from an abstract class named *Creatur*e. This class is still an entity, which means it inherits from the *Entity* class, but contains some extra member variables, like a moving direction and a velocity, and also contains member functions to actually move, but also to check for possible moves. In this project, the entities that inherit from this class are *Bullet*, *Invader* and *Player*.

Now let's talk about the *game* namespace. This namespace contains all classes that together form the core components of the game, while also housing the exception classes and some other utility classes. The exceptions and utility classes are grouped in their own namespaces (nested in *game*), which are respectively called *exception* and *utils*.

The most important classes in the *game* namespace are undoubtedly the model, view, and controller, which are bundled together in the *Game* class and are clearly implemented using the MVC design pattern, but also use the Observer and State pattern.

The MVC pattern is implemented as follows: the controller contains the main game loop and checks for user input on each cycle. The controller is able to change the model according to the given input, i.e. it can change the moving direction of entities (depending on input or AI decisions), tell the model to reset when a new level has to be loaded, and even change its state. Important to note is that the controller does not actually move the entities or check for collisions. That's a job for the model. The model is basically 'the database' of the game, which also contains game logic like movement and collision detection. The model uses the State design pattern, which means that it keeps track of a 'game state', but it is also a part of the Observer design pattern.

The game state is an essential component for the view, the controller and, of course, the model itself. The view will render different scenario's per state, the controller will check (and thus accept) different input depending on the state and the model may update differently (if at all) depending on its state.

The model also is a *Subject* that is being observed by one or more *Observers* (which in this case is only the view). Every time the model made important changes to itself (i.e. altering entities, game state, etc.), it will notify its subjects, which on their turn will update themselves if necessary (which in this case means: change the visual representation of the model).

The view gives a graphical representation of the model and will only update itself when its subject (the model) tells it to, as explained before. It contains objects such as an SFML window, a resource library (which is one of the classes in *utils*), and member functions to represent every game state.

Those were the most important components of the *game* namespace. As I said before, this namespace does not only contain exception classes for handling problems with the graphical representation and loading new levels, or utility classes such as the *Transformation* class, the

Stopwatch class and the resource library that was mentioned earlier, but their design and implementation is pretty straightforward, so I won't be discussing them in further detail.