Space Invaders

Report by Mateo Sierens

Introduction
For this project I decided to base myself on using sprites from Undertale, even though it has nothing to do with Space Invaders. The reason for this is that it's a great game and has some really nice soundtracks that I included as well. This also motivates me to come back to this project later to make changes and include a boss battle with a lot of features, as this is easily done thanks to the design patterns used. And that's also where this report is all about: the design patterns and structures I used to realise this game and the reasoning behind my choices.

Model-View-Controller
The pattern which the project concludes of the most is the Model-View-Controller pattern (MVC). I decided to splitting this up in some base and derived classes. Model is referred to as Entity, View and Controller are both just referred to as View and Controller. For each entity (e.g. PlayerShip) what I did is make an class for said entity that derives from Entity, one that derives from Controller and one that derives from View. This made it easy for adding new entities, as I wouldn't need to touch the files of other entities (except if these entities interacted with each other in some way). The Controller base class contains a shared pointer to the entity it manages, the Entity base class contains its coordinates on the screen and the View class contains the sprite and texture of the entity it needs to draw. By giving input the user can control the Controller's functions, which will trigger the entity it controls to update. The Controller base class also contains an update function, which doesn't need any input, and will be called every game loop to automatically update its entity (e.g. make a bullet that is shot move further on the window). The controller will do this by calling some functions from its entity. The way the entity then *notifies* the view to update on the window is by another pattern we call the Observer pattern. This way we make sure the user can only use the controller for controlling the model (entities in this case) and can automatically see the view.

Observer pattern
In the last paragraph we talked about the Observer pattern for updating the View class. The way this Works is with 2 base classes: an Observer class and a Subject class. A subject (also called "observable") will be observed by observers. It has a function *notify* to notify the observer that is watching it when something happens. With the notify it gives an event argument (e.g. "update") and it will tell this to every observer observing it. If the observer doesn't recognize the event, it ignores it, but if it does recognize the event it will execute the necessary code. This way when something happens in the entity classes in my project that concerns the view, then there will be a notification to the observers observing it to update the view. The entities in my project are the subjects, so they derive from the Subject class and the views are the observers, these derive from the Observer class. When an entity and a

view are created, the entity adds the view to its list of observers. This way the window is always up to date when something happens to the entity.

Transformation and Stopwatch with Singleton
Another well known design pattern used in this project is the Singleton pattern. It is an object made so that through the whole program there is only 1 instance of it. This is useful for the transformation and stopwatch in my project as there is no need for more than 1 instance of this and it is accessible throughout the whole program. By making it constructor private and making a static method instance, it can overcome the ability to make more than one instance of it. The transformation class in this project stands in for transforming window coordinates to the right game coordinates by taking the window size into count. This way the whole game can use the [-4,4]x[-3,3] coordinate system for different window sizes.
As stated in the assignment the Stopwatch class will be used to help keeping the time between two game ticks. At first I didn’t have this implemented, and after testing my project on the lab computers the utility of this idea came to light: Everything was so much slower than when testing on my pc, and this was due to the effect of how fast a processor can go over the *game loop*. By doing some reading online I found a way to go against this with the Stopwatch class. This class will keep 2 time points with the C++ *chrono* library: the current time point (of the beginning of the loop) and the previous time point (of the beginning of the loop). It will also be accountable for computing the elapsed time in milliseconds between these two time points, as this is needed for the computation of the *lag*. Now we keep a self chosen value called MS_PER_UPDATE, which is 16 in this project as this is the value for 60 frames per second. The *lag* value will keep count of how much the program is behind on the real time, and will update this until it is back on track. Better processors will have less lag, which results in smoother moving of entities on the window, but this way we will have a game that runs same speed on every pc.

The Big Picture
Now all that is left is explaining how all of this works together. The game loop (which every game consists of) is stored in the Game class. The Game class will hold a lot of stuff, like backgrounds and music for win/loss/gameplay, ints and bools for some game logic but most importantly the game objects. We keep track of a shared pointer to the controller of the PlayerShip. This way we can easily move the player when there is input and handle events like shooting bullets. We also keep a vector of EnemyShip controllers to make it easy to move them in sync. All the other controllers are stored in a list of shared pointers to Controllers, which the update function will be called on each update. As this function is pure virtual it will call the update function of the derived controllers. We also keep a vector with shared pointers to View objects, where all the entities views will be stored in. As sprites and textures are stored in the base view class we just need to call the getSprite() function for every object in this vector and the correct sprites will be drawn on the screen.

Some other features that I added are multiple levels (3 are available), each level has more enemies than the previous level and will let enemies shoot bullets more frequently. As stated at the beginning of the report I did not only use sprites from Undertale, but also music. There is a song that plays during the game (that goes over the levels as levels don’t last too long), a song when you end up with a game over and a song that plays when you win. The game over and win each have an image included, with a self written fade in. The way this fade in works is that it changes the transparency of the window with a certain factor each game loop until it is fully visible.

Levels use json files, and are read with the nlohmann json parser, which is included in the project under the file json.hpp. An exception is thrown if the level json file isn't found in the working directory. I also added exceptions for when textures and music aren't found, and when the font (which is used for the text representing the lives) isn't found.