A deeper dive into the methodology used in Space Invaders.

Tobias Wilfert 20181439

About the MVC:

I implemented an MVC pattern to allow me, "to completely separate the visualization from the game logic". The pattern that I implement works as following. The Model class holds a deque of levels as well as all items that are level independent such as the counters and the player. The Model and the entities in it are not aware of how they will be displayed to the user but rather only hold the information such as the position of each entity in the game logic dimension as well as the values of the counters.

The Controller class is tasked with handling input from the user as well as spawning bullets from the eniems, updating the position of all bullets and enemies, and deleting entities from the Model that no longer exist, like killed aliens or collided bullets.

The View aspect of the MVC turned out to be not as nice as possible due to restrictions from SFML. Every frame before we draw the updated entities to the window that the user sees we are forced to call 'window.clear()' which clears all entries of the screen, in return forcing us to draw all entities in the Model again even if the position has not changed at all. This makes it obsolete for the Controller to keep track of which items have changed, as we need to draw all items again anyways, so the View asks the Model for a deque of pointers to the base type of the class hierarchy. The deque contains one entry for each entity currently in the model. The view is then able to determine to which derived class the entity pointed to by the pointer belongs and will cast it to its proper type to retrieve all the extra specialized information it holds. Depending on the type of the entity the View will then draw the entity to the window for the Viewer to see.

It is important to note that the Model and entities don't know how they look and the View does not know how they work this allows that when changing the GUI library from SFML to something this can be done through changes in the View alone, the Model and the View are thus separated like intended.

About the Class hierarchy:

The class hierarchy used was designed with easy upgradeability in mind and aiming to minimize redundancy. All classes are derived from an abstract base class Entity. The Entity class holds only the bare minimum that all entities in the Model need: position and size as well as a virtual function 'getEntityType()' that will allow a pointer to this abstract base class to be asked what the actual type of the entity is it is pointing to. This allows us to store all the entities in the model in a list of pointers to this abstract base class and then ask and cast them to the proper type if we need to edit or retrieve data from them.

Both the Counter and CollideObject class are derived from Entity the distinction between them being Counters do not care if they collide with anything as they are part of the UI rather than the actual game field. Counter are ontop also capable of holding a score while collideObjects have attackpoints, healthpoints and a bound in order to detect if they collide with another object.

Further Immortal and Moral are driven from the CollideObject class. Immortal is a class for the Sky and the Ground that can be collided with but don't take damage upon impact and are thus immortal while all remaining entities of the Model can die.

From the Mortal class, the Shield, Player and MoveObject class are derived. The player is different from the Shield as it can be in a state of respawning and different to the MoveObkects as it does not have a predetermined MovePattern that it will follow the entire game.

From the MoveObject class the last two classes of the Model are derived, Enemy and bullet both hold specific data items such as type and color that will be used by the View to display them in a certain way. As both are derived from the MovePattern class they hold a Movepattern that determines the way they will move by specifying directions, steep size, speed and pattern to follow.

In this hierarchy, every Class holds only the information that is needed for it to work and no more. Adding a feature to the game can be done by creating a new class and placing it in the correct spot in the hierarchy when positioned properly the only extra work needed then is to tell the View how to draw this new Class.

Functions are derived and overwritten to make the entire hierarchy work in a way that should be straight forward and easy to understand.

About the level files.

The level files are xml files. They are parsed using a free xml Parser taking from the internet named TinyXml I did not change the files the Parser uses and do not claim I wrote them. They are provided with comments at the beginning of the files crediting the creators.

Each file contains one level composed of MovePatterns that the latter entities in the file can reference via the index of the MovePatern starting with the number zero. Enemies can and should also be provided if none are the player wins by default. The enemy's position, size, attackpoints, healthpoints, move pattern, type and color can all be changed by the user to his liking. The last part of each level is the Shields which again can have variable size, position, attack points, and health points. The XML level files are designing in a way to give the user as much flexibility as possible.