Advanced Object Orriented Programming Final Project Report

Kobe Moerman (s3385507)

Cornelis Zwart (s3331660)

October 28, 2018

1 Problem Description and Analysis

The assignment consists of (using a functional single-player asteroids game) allowing a multi-player use of the game across multiple machines using UDP. To achieve this, we have to allow a user to host a multi-player game, and join or spectate a specific game by giving the program the desired IP and port. Furthermore, a database is implemented to store a player's high-score.
We have decided to implement the following rules in our game.
A player wins the game when he is the last spaceship alive. Each time this happens his score gets updated in the database. Additionally, a spaceship that has been destroyed also sends his score to the database (but does not obtain the bonus of a game win). Notice that the score in game represents how many other spaceships he has destroyed. This way the player is motivated to participate in the action and to win the game.
When a player creates a lobby, there are no asteroids spawning and firing bullets is disabled. Only once there are enough players for a game to start (at least 4 players), do asteroids spawn. Firing is still disabled for another 10 seconds such that each player can get into position.

To complete this assignment there are several functionalities that have to be taken into consideration.
First, we want to update the GUI and add a menu frame. By doing this it allows the player to select at his own will which game feature he wants to try out. We have to make sure it is done in a clear and user-friendly way.
Furthermore, we want the user to be able to create a game lobby for others to join or spectate. To implement this we have to be able to send packets through a specified IP and port number. As a result, each user needs to be able to send its game data and receive packets that contain game updates.
Likewise, there will be multiple spaceship models inside the game. We then have to update the Game class accordingly to allow several instances of a spaceship to exist. The rules also have to be different for single-player and multi-player.
Finally we have to be able to connect to a database to update each player's high-score.

2 Design Description

Before anything, we separated the classes into a MVC (Model-View-Controller) design. By doing this we have a clear understanding what each class is responsible for in the program. The model contains the main data and updates the controller of any changes. The view is simply there to represent the visualization of the model data. Then the controller acts on both the model and the view. It updates the view whenever data changes which allows to separate view and model packages.
We then proceeded to implement a Server package as it's purpose differs from the MVC pattern. As such our Client and Server are distinct which allows us to find specific features with more ease. In addition, as our classes for creating packets got noticeable, we decided to give it it's own package. As a result, our

server package mainly handles the packets whereas the packet package is responsible for the game data. Finally, we also separated the database and tests from the rest of the code by giving them their own package. This is simply done for clarity.

While working on this assignment, we have implemented pattern designs from which a few are discussed below.

Prototype pattern: In the Game class we make use of this pattern. At each game update we check for destroyed objects. As a result we take a new list and add to it a clone of all elements that are not destroyed. Since receiving a clone of the object is less costly than creating a new one, we improve significantly the performance of our program. By doing this we require less time and resources to update the game.

Factory pattern: This can be seen in the ManagerServer class. We de-serialize the packet data into an Object. To determine the type of the Object we type cast it with the Packet abstract class. This one implements the interface PacketAction which itself contains the method `int action();`. As a result, each of the classes that extend from Packet (PacketLogin, PacketSpectate, PacketControl) override this method and return a specific and unique integer. So, with the use of a switch statement, the program can take different actions depending on which packet the Object is an instance of.

Decorator pattern: We have implemented this several times (GameObject, ManagerPacket, Player). The most significant implementation is for the Player class. Depending on whether the game is single-player or multi-player, the controls have to be used differently. As a result, for each key press we update the boolean values accordingly but also the abstract method that updates the ship (updateShip()). In PlayerMulti, which extends from Player, it sends these controls over to the server. In the case of PlayerSingle, also extending from Player, it directly updates the the ships' data inside of the game.

Bridge pattern: Once again this can be seen with the Player class. This is however also the case for ManagerPacket. After each run method it calls the receivePacket abstract method. In all three classes that extend from this (ManagerServer, ManagerMultiplayer, ManagerSpectator) it uses the method differently. In such a way each of these classes can vary independently.

Observer pattern: We use this in the Messenger class. It allows a number of observer objects to see an event.

For the networking functionality of our program we adopted the following implementation.
We create a class that serializes each packet we want to send. As such, every other class that implements serializable will receive this data. Furthermore the Packet class contains the IP and port of the sender as well as the functions it wants to send. Each class that extends from this one can change it's parameters accordingly. The spectate packet only contains the IP and port it is connected to. The login packet does this as well with the ability to update the username.
When creating a multi-player game we initialize this one and add a player. The player can only send commands over onto the server; that is accelerate, left, right, fire. As such, each player is responsible for it's own spaceship. At each game tick, the server will send a serialized version of the Game model to all the connected players.

3 Evaluation

Essentially our program follows all requirements and works without problems. We are very satisfied with how our GUI looks. We also think that our Player class, to separate single and multi, is accomplished very neatly. Similarly, we are happy with how we implemented the server functionality but do think that it could be done in a more compact way. Furthermore we payed close attention to the game rules so that it is a good experience for all players.

One of the features we did not implement is the ability for multi-player games to restart this one. For players to play another game they have to host a new server. This leads to another feature we decided to neglect due to time constraints. Once the game is over on a specific IP and port, a player can't use this address anymore as we did not disable the server or reinitialize the game. A similar bug that the user might encounter is that he can create a server on the same IP and port as another user. This will simply make him join the server instead of warning him that this game is already active.

There are also some improvements we could make to our code. What seems most obvious to us is to make different classes extend from the Game class. These would simply be responsible for the rules of the game and therefore would make it easier to add a new game-mode or even change one independently of others. Of course, with more time, we would have loved to attempt and implement many more fun game-modes (i.e. Asteroids.io where your score is how many spaceships you have destroyed but, once destroyed you get reset). Furthermore we would add a list of destroyed spaceships that would be painted on the last location they got destroyed. However, these would not be able to influence the game in any way. This is just to keep the score of each spaceship on the screen and restart the game by initializing all destroyed spaceships. Finally, as mentioned above, we would fix all server features and try to implement it more clearly.

4 Team Work

During this project we have both worked on different aspects of the program.

Kobe was responsible for at first establishing a connection between the client and a local-hosted server. Then, he created the GUI of the game and worked on making the game multi-player friendly (including the implementation of the game rules for both single and multi-player). He then proceeded to refactor the whole project (meaning the addition of patterns, test cases, and comments) and established a connection to a database.

Cornelis was in large part responsible for allowing the program to send packets between the client and the server over a specific IP and port. Furthermore, he implemented the option for the user to create a game, join a game, and spectate a game over the server.