Programming report
Asteroids Assignment
Advanced Object Oriented Programming

Maria Sophia Stefan (s3413896)
Anda-Amelia Palamariuc(s3443817)

October 30, 2018

1 Problem description

The final assignment is represented by the well known arcade game "Asteroids". In the beginning, we deal with a functional, single-player version of it. This version of the game allows the player to move around and shoot at randomly spawning asteroids, thus earning points. In the end, the game should keep its existing single player functionality and also have two more game states : multi-player and spectator. Both of them are going to be implemented by networking. The left-hand one will have multiple players joining and participating in the game, whilst the right-hand one will let you spectate an ongoing multi-player game.

2 Problem analysis

Given that the original code was divided into a model-view-controller pattern, we will discuss the further extensions that need to be implemented for each component:

- **Model**: the given model should be extended such that it will support a multi-player and spectate game state. Therefore, we may want to keep the existing Game class as an abstract superclass that deals with common components for both single-player and multi-player states and then divide it into two separate subclasses: one for single-player and one for multi-player. As far as the multi-player model is concerned, we want to have multiple players and their corresponding ships, different updates when the game is over (i.e. awarding the last standing player an extra point) and the existing point awarding system (when hitting and asteroid), slightly modified by adding the possibility of being awarded a point when destroying an opponent's ship. The latter arises the question: how will we know which player destroyed another ship? how will we know which player aimed at an asteroid? In order to tackle this issue, we are considering having an unique ID number that will correlate a player with its ship and the bullets that he fires.

- **Controller**: the given single-player game expects no input from the user, but we would like the user to give his nickname and to pick a desired color. Also, when the user decides to host a game, he should be able to select the number of desired opponents, meanwhile when a user wants to connect (either join/ spectate) to multi-player game he should also input the ip address and port number of the existing host.
  Besides dealing with user input and spaceship actions (which are controlled by the keyboard in the given single-player game), we have to deal with networking. We would like to have a player that is also the host, which will deal with messages from all the other players (joiners and spectators). The messages will be treated accordingly and the host's local game will be updated and then sent back to the rest of the players. This leaves us with other two networking components besides the host : the joiner(s) and the spectator(s) which will update their local game upon receiving the host's game. At

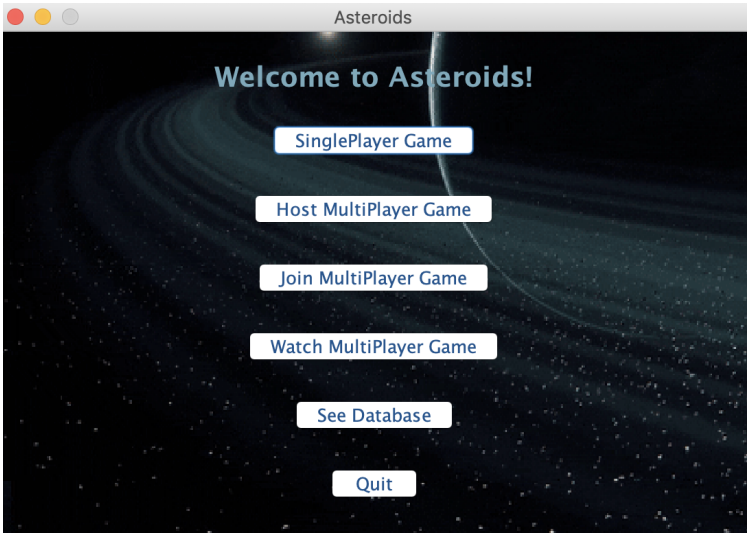
this point, it becomes clear that all clients should send messages to connect/disconnect and that the joiner (which has the possibility to move around) should notice the host about its current movements.

- **View:** we have to extend the given panel in order for it to paint multiple game objects (such as ships), also we would like to see the name and the score of each player. On top of this, we can bring improvements as far as the whole user-interface experience is concerned: add pop-ups before closing a frame and wait for closing confirmation, add a menu from which the user can choose his desired game state, display the host's ip-address and port number, as well as the number of expected opponents and the number of connected ones, display a message for joiner's to wait until everyone is connected and the host is ready to start the game and also display messages when winning/losing the game.
- **Database:** When playing a multi-player game, we would like to keep track of previous highscores for each player. In this case, every player should pick an unique name and if their current score is a "high-score", then it will be updated in the existing database for future reference. In addition to this, the data base should be displayed and an option for this be included in the aforementioned menu.

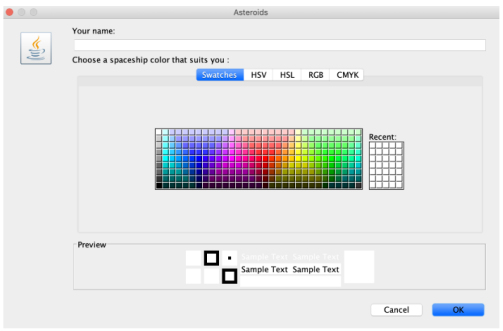
3 Design description

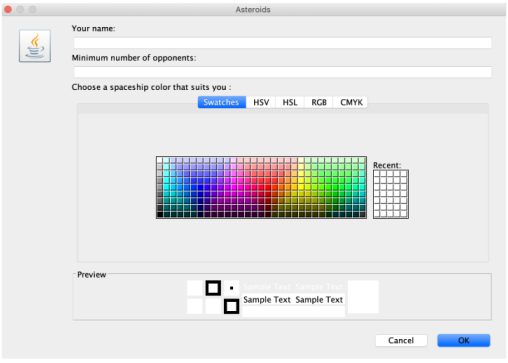
The presented project stands mainly on the model-view-controller design principle. We used the provided code as a foundation and build on from that. We will further on discuss about our GUI components, the database, the networking functionality and our contribution towards improving the provided code.

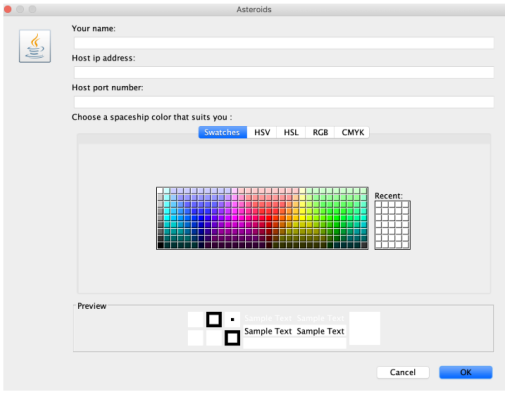
1. **User interface:** The user interface has multiple components:
   - Main menu: gives you the possibility to play a single-player game, to host a multi-player game, to join an ongoing multi-player game, to spectate one, to check your highscore in the database and to quit the game. From this point on, whatever the user chooses, the main menu will be disposed and the user will be redirected accordingly.



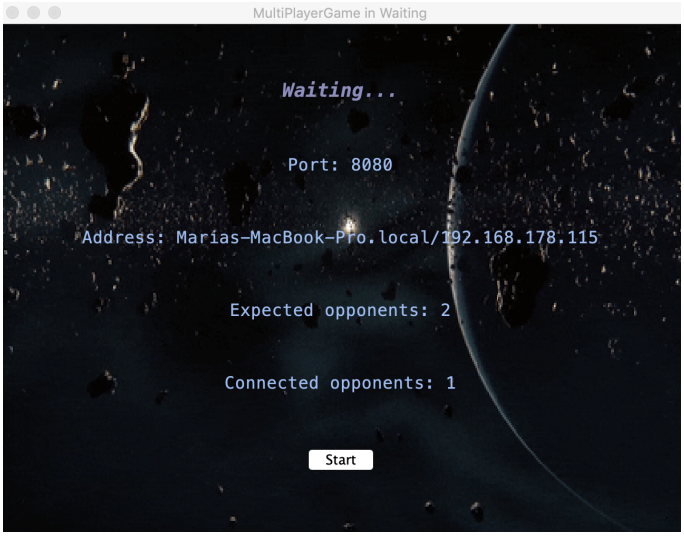
   - All four game states chosen from the main menu will show a specific frame where the user should input its preferences. From left to right, we have the ones for: single-player game state (asking for the user's nickname and color choice), host multi-player game state (asking for the user's nickname, desired minimum number of opponents and color choice), join multi-player game state (asking for the user's nickname, host's ip-address and port number (for networking connection purposes) and color choice) and last, but not least, the spectate multi-player game state which asks for the host's ip-address and port number.







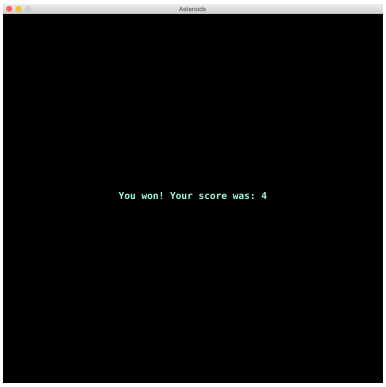
- When the game is not connected yet, the host will display a waiting frame which contains its ip-address and port number, as well as the minimum number of opponents required for the game to start. It will also show the number of connected opponents, which will update whenever there is a new joiner.

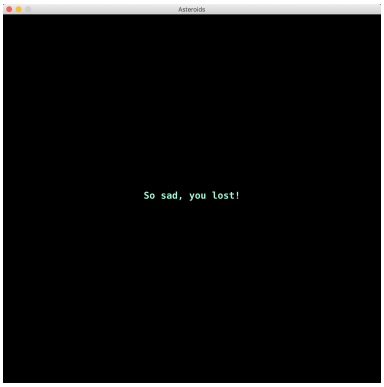


The moment the number of connected opponents is greater or equal to the number of expected opponents, the Start button will be enabled and the multi-player game will begin. In the meanwhile all joiners will display the following message:

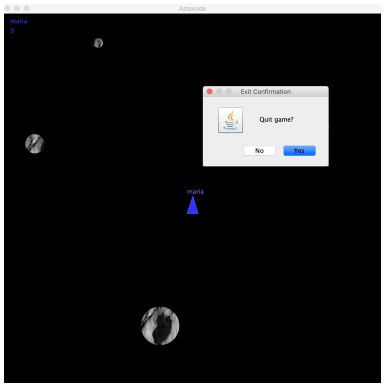
In addition to this, if a spectator wants to join a game that is not yet connected, it will display a black screen until the host decides to start the game.

- When a game ends, the user will see one of the two following messages:



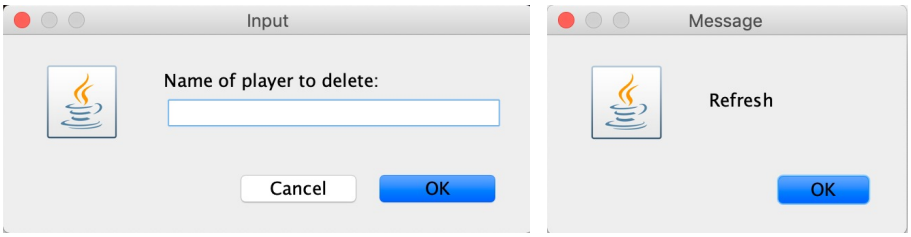


To sum up, exiting the game frames will result into a pop-up asking for an exit confirmation and the user will return to the main menu.

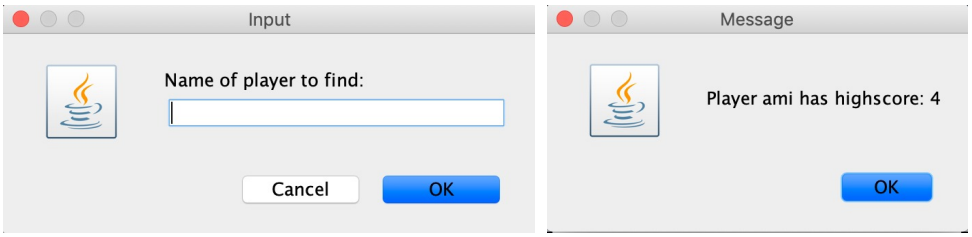


2. **High-score database:** The game also has a high-score database implemented using persistence. The database has as entity the Player class and as Id its name. Note that if a player uses the same nickname, he will not be added again to the database and his score will only be updated if it's a new high-score. The database highscores for each player are updated in the multi-player game model, hence there is no database implementation for the single-player game state.
Going back to the main menu, when choosing the option to see the database a new frame will pop up, which also includes two more possible actions other than seeing all the players-highscores entries.

As the above picture suggests, by pressing `D` on the keyboard you can delete entries by name and by pressing `F` on the keyboard you can find an entry by name.
When the user wants to delete an entry a pop-up asking for the name of the player will appear, the database needs to refresh in order to display modified content. The refresh pop-up will take you back to the main menu from which you can select to see the database again.



When you want to find an entry in the database, a pop up will appear asking you for the name. In case the name exists in the database, it will return the score of the player, otherwise it will let you know that there is no such entry in the database.



3. **Networking:** The entire networking functionality can be found in the `net` package included in `aoop.asteroids.controller`. The root of the networking functionality is represented by the `Network` class which contains methods to serialize/ deserialize data, as well as to receive and send a packet via a DatagramSocket. In this class, each network component gets bound to a DatagramSocket: the host's server address has a specific port number and the local host's ip-address, while the joiner's address and the spectator's address are bound to a random available ip-address and port number. Furthermore, we will briefly discuss the role of each network component:

(a) **HostServer:** this class implements the singleton pattern to prevent binding the host two times to the same address and DatagramSocket in case the same user decides to host two different multi-player games in the same time. Therefore, one unique instance will be available when running the program. The host server will keep track of its clients, which are added whenever they send a "join", respectively "spectate" request. Since the host server is an observer of its local game model, it will send the updated version to all clients when it's notified.
Moreover, in order to assure that the clients' game threads do not run before the host's game thread starts running, the host will also send "start" requests to its clients (when the user presses the "start" button when all opponents are connected: in `WaitingFrame` class).
The host will receive five types of different packets and messages and will decide upon further action: a spectator will at first send a "spectate" request, thus being added to the clients list, from which it will be removed the moment it will send a "disconnect" message, a joiner will at first send its local player, which will be interpreted as a "join" request, thus it will be added to the clients list and its local player will be deserialized and added to the local game model together with its ship. Further on, a joiner will also send a message containing its local player's actions, which will be deserialized and passed to the corresponding ship. In the end, if a joiner wishes to disconnect it will send a stop message together with his unique ID number and his address will be removed from the clients list and its ship and score will be removed from the local game model.

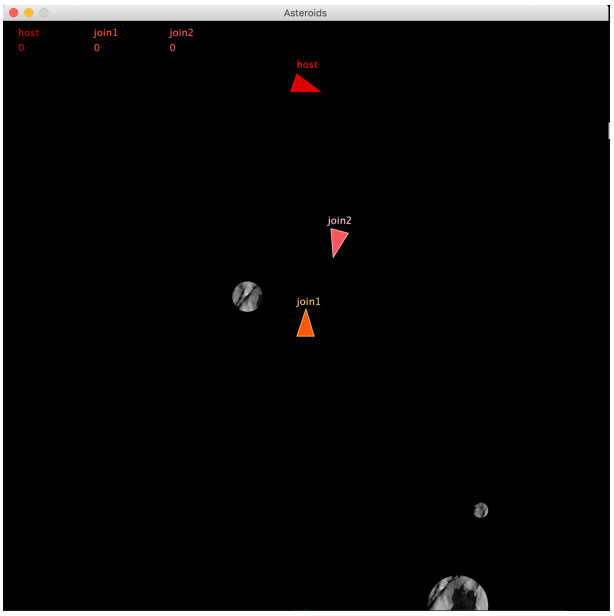
(b) **JoinerClient:** This client will at first send one "join" request to the host server containing the serialized local player. Then it will listen to the "start" request from the host server, thus starting the thread of its local game model and also for updated game models from the host server. Upon receiving an updated game model, it will update its own. This class will keep track of the local player's actions and if they change in any way, they will be passed to the host server. If the local player decided to exit the game, a "stop" request will be sent to the host, together with the local player's unique id, the client's thread will stop and the socket will close.

(c) **SpectateClient:** This client will at first send one "spectate" request to the host server. After it will be added to the host's clients list, it will expect updated versions of the game, thus updating its own. If the spectator decides to exit the game, it will send a "disconnect" request to the host server, the client's thread will stop and the previously used socket will close.

4. **Overall design concerns:** On top of it all, we included several improvements to the existing code and model: the root of the game model remains the same, but now it has two more subclasses for each game state. Besides the abstract game class' functionality, they implement their own. Each player will have his own id number, assigned when added to the game model (part randomly, part based on his chosen color) and based on his id number he will be removed, added, awarded points etc. One player shares his id number with his corresponding ship, as well as with every bullet that his ship fires. This way, during the multi-player game points will be awarded to each player that aimed at an asteroid or at an opponent's ship. Also, asteroids used to spawn in the proximity of the only existing ship, now they spawn in the proximity of a randomly chosen spaceship from the game model. The single-player game automatically restarts when the player dies, meanwhile if a joiner or the host dies, they will continue as spectators of the game. A joiner can disconnect during the game, but it cannot disconnect while he waits for the game to start, meanwhile the host cannot disconnect in the middle of the game. In the end, there will be only one winner, to whom another point will be granted. The winner will see his score and all the other players should consult the high-score database in order to see their overall progress after everyone has exited the game.

4 Evaluation

Taking into account the required set of features, we managed to implement them all and also add quite a few extra functionality, not only user-interface wise, but also as far as the database options and the "game rules" are concerned. In the multi-player state the game looks as follows:



In the upper part of the screen the name and current score of each player is visible.
We have implemented the game mechanics such that two game objects of the same type can pass through each other, whilst bullets collide with both asteroids and spaceships and spaceships collide with asteroids.
The game is stable both locally and network-wise. If we were to have more time, we would have improved the latter by using both UDP and TCP, mainly because TCP is more reliable. A good idea would have been to establish the connection between the server and the clients using TCP and then send information via UDP. Testing the code (at least the constructors for each class) would have been another good practice that we did not manage to implement.
Another weird behavior that we noticed while testing our code comes from the database frame when pressing the keys for removing/finding a specific player. Most of the time this does not bring any problems, but sometimes if the keyboards are pressed multiple times at a fast rate, nothing responds. If the user returns to the main menu and then tries again, everything works. Unfortunately, we did not manage to find the cause of this issue and this is another thing we would have liked to have the necessary time to improve. In the same light, maybe the database frame should have been scrollable, but since the user can delete and search for entries, this is not that necessary.

5 Team work

During the development process we came across a few challenges, from which we can mention: if the user decides to host a multi-player game and then goes back to the main menu and selects the option to be a host again, which basically will translate into instantiating the host server twice, a few exceptions will arise because the ip-address and the port number to which the first instance of the host server is bound is already in use. In order to solve this problem, we came up with the idea of implementing a singleton pattern for the host server, which will result in only having one instance of the host server. Moreover, in order to update the view in certain circumstances (such as the number of already connected opponents), as well as sending the host's updated game to all clients, we needed to implement the Observable pattern. This way, we add view components and networking components as observers of the game model.
While working on this assignment, we had to go through all object oriented programming principles and techniques presented in this course. Moreover, we feel that this assignment consolidated the foundation we have built before.
We constantly worked together on each part of the assignment: brainstorming the ideas, implementing them and writing the report. We found that working in pairs for this specific assignment proved itself to be extremely helpful when dealing with lack of ideas, errors etc.

6 Conclusion

To conclude, we extended the given single-player game into a fully functional multi-player game which lets the user choose between four actions: play a single-player game, become the host of a multi-player game, engage himself into an ongoing multi-player game and spectate an ongoing multi-player game. His high-scores gained while playing a multi-player game are successfully stored inside an easily accessible database. We would say that we are content with our work since the program functions properly and the overall user experience is very good thanks to the improvements made GUI-wise.