Advanced Object Oriented Programming
Programming report
Asteroids

Andreea Glavan Ai Deng
s3083691 s3198928

November 2, 2017

1 Problem description

(1) **Short description of the assignment:**
The final assignment, Asteroids, required turning a single player game into a multi player game, with the following features:

i. A main menu containing all the game modes
ii. Single player mode
iii. Spectator mode
iv. Multi player mode
v. A database containing the high scores

Some of these features required more implementations, such as joining a multi player game, hosting a multi player game, a multi player game score system, and differentiating players through nicknames and colors.

(2) **Analysis of the assignment and the work:**
We will look through them one by one:

i. This requires an GUI menu which should contain some buttons for users to be able to choose the game mode they prefer.
ii. The original program fulfilled this requirement. We would just need a button as an event listener.
iii. For Spectator and Multi-player modes, we set up the our game design as below:
  - The host waiting for other joiners' connections. He has the right to start the game anytime. However there must be at least 2 players or less than 6 players.
  - When the room exceeds its capacity 5, the joiner would not be able to join the game anymore. Otherwise he could join anytime during the game.
  - The last standing player would gain one point. The game would be reset automatically after that, so the game continues. All players' score would be written recorded into DB.
  - However, if the last standing player is achieved by disconnections of **all** other players, he would not gain any point. And the players scores would not be written in DB.
  - In the middle of the game, if all joiners disconnect with the host, the game aborts. The host has to wait for new player joining the game. Then the game would start again.

we would use a server client model to achieve this. The server should be in charge of the game, and the clients are the joining players as well as the spectators. We use UDP for this due to the high number of packages which contain the game state being sent.

The server would deal with the game itself, containing the game, the high score database, as well as array lists with all the players and spectators addresses. What is more, the server contains its own player object, allowing the host to play as well. The host updates the game based on received input and, using the observer pattern, sends the new game state to all the joiners and spectators. At the end of the game, the winners high score is added to the database through the server.

The clients are able to send connect and disconnect requests to the server, based on which they become a new player or a spectator. While both players and spectators receive the state of the game, players are also able to modify it. We used a key listener for the players and send the way each players ship is supposed to move based on that to the server, which in makes the actual change in the game.

iv. The database needs to contain the highest score of each player, based on the number of games he has won. We would use ObjectDB JPA for this, having the scores as the entity class which contains the *nickname* and an *integer* representing the score.

2 Design Description

(1) **programs structure:**

(a) **Choice:**
The program structure was done keeping in mind the necessary functionality for each element of the architecture. The program is split into four packages:

i. Model
ii. View
iii. Controller
iv. Database

It also contains the main class Asteroids separately. The main class is similar to the previous version, dealing with initializing a new start menu.
The Model package contains 2 sub models. One is the game model, which deals with all GameObjects. The other is the server-client model which contains all our networking classes: server, and clients(Spectator/Joiner), serverIO and clientIO. The View package contains all GUI classes that an user would see when he plays the game. The Controller package contains all action classes that would be able to change the game and the view. And Database package deals with entity class and entity manager class.

(b) **Substantiate our choice:**
We justify for all the classes in each of the package as below:

i. Model

- GameModel: This model contains all our Gameobject classes. We keep all GameObject class in the original Model package as they serve the purpose of a Game.
  However the game now it allows multiple players (in our case spaceships) to take part, these spaceships are now stored in an array list. Due to this change, methods such as linking controller now take an extra parameter, the index of the desired ship. What is more, this class now contains serialize and deserialize methods for turning the game state into a byte array, which is used for sending datagram packets to and from the server.
- NetworkingModel: under this model we have classes as below:
  - Spectator Class: The spectator class is a simple client class which sends only one message to the server as a connection request. This class only contains a game, which is updated according to the new versions it receives from the server, unable to make changes to it. In the menu bar for spectator, there is a disconnect option which sends

a disconnect request to the server such that this address no longer receives the game state.

  - Joiner Class: The joiner class is similar to the spectator class in the sense that it also contains its own game which is updated according to the packet received from the server after sending a connect request. However, this client implements key listener, sending messages back to the server in the form of a a datagram packet containing four booleans, one for each possible key, each indicating whether the key was pressed or not.
  - Server Class: The server class contains the real game, as well as array list of all the addresses for spectators and joiners, and the database for high scores. The server has its own player object which can make changes to the game, and also deals with updating the game according to the movements received from the joiners. Once started, the server waits for a packet from either a spectator or a joiner. Based on this the address is added to the correct array list and the input is processed in the method processInput. This method deals with connect and disconnect requests, and in the case of a datagram packet containing the moves of a player, changes the current state of the game according to those moves for the ship belonging to that player. The server game has an observer, and with every update the new updated game is serialized and sent to every spectator and joiner. What is more, this class contains a server network IO, which, similarly to the client IO, deals with the server socket, port and the sending of data. The database connected to this class is updated with the high score of the wining player, as soon as there is just one player left alive.
  - ClientNetWorkIO class: This class contains the client socket as well as the server address. It serves a purpose to deal with all the communication a client needs to take care, including the receiving of datagram packets, as well with the sending of messages and datagram packets to the server.
  - ServerNetWorkIO class: This class is similar to ClientNetworkIO except it serves a purpose for the server instead of a client.

ii. View

- AsteroidsFrame and AsteroidsPanel classes: We keep these 2 classes as they serve a purpose to show the game for the users. They are adapted to use colors and to paint every ship presenting for each player in the game, as well as the corresponding nicknames.
- GameMenu class: This class contains the 5 game-mode buttons: single player, hosting multi player, joining multi player, viewing the high scores and spectating multi player. This class servers a purpose for users to be able to choose the game mode they intend to participate.
- HostFrame class: This class servers a purpose for a host to be able to see all the connections of joiners.
- ScorePanel class: This class serves a purpose for any host who wants to view the score.

iii. Controller

- ServerController class: A keylisterner which server a purpose for controlling the host's spaceship.
- JoinerController class: A keylisterner which server a purpose for sending joiners' commands.
- Button classes: These classes take care of all the buttons and corresponding actions in the view.

iv. Database

- Entity Class Score: This score class which contains the *nickname* and an *integer* representing the score. It records down the score, the number of games this player has won, and the corresponding player(distinguished by *nickname*).
- ScoreDB Class: This class contains an entity manager factory and an entitiy manager. They serve a purpose to add, update and persist instance of Score to our database, making

sure that the database only contains the highest score of each player. The method *addWinnerScore* is called at the end of every game in each server and checks if the *nickname* of the each player is already in the database, in which case the score needs to be updated, otherwise a new entry is created. The method *getAllScores* is used for the *ScorePanel*, allowing for the contents to be displayed.

(2) **Design patterns:**

(a) **Choice:**
We have included the following design patterns in our project

i. **MVC Design Pattern:** MVC is our main design pattern. The game is the model; the GUIs are the view; All keylisteners and buttons are the controller.

ii. **Composite Pattern:** In our game there are different kinds of GameObjects There is one abstract class GameObject to define the general methods for all gameobjects and then we extends it to all different gameobjects. Also we use composite pattern for paint a panel. We paint components of the panel separately, spaceships, asteriods...

iii. **Observer Design Pattern:** In our project, Game has AsteroidsPanel, Server as observers; Server has Score-panel and Host-Frame as observers.

(b) **Justify our Choices:**

i. **MVC Design Pattern:** This comes quite straightforward as game player needs to be able to view and control the game. The view comes from the game model, and the controller change the state of the game mode.

ii. **Composite Pattern:** Composite pattern should be applied here as all components of a game/-panel behaves as the similar/single object.

iii. **Observer Design Pattern:** AsteriodsPanel needs to be updated according to the game while server is interested in the change of the game so he would be able to send the change the the spectators and joiners. HostPanel needs to be updated according to the server's connections; ScorePanel needs to be updated according to the change of the database. These three situations makes the use of the observer design pattern prefect here as observer design pattern is useful when an object is interested in the state of another object and want to get notified whenever there is any change.

(3) **Networking Functionality:**

(a) **Choice**:
We use protocol UDP for this project due to the high number of packages which contain the game state being sent and received. There are three major stages:

i. Request/reject connections: In both spectator and joiner mode: after they type the IP address and port number of which server they wish to connect to, a connection request would be sent from their client-socket to the server-socket. The server will accept all connections from spectators while he could reject the connection from joiners if the game reaches the capacity. In this case, the server would send a connection rejecting message back to the server.

ii. During the game: the server, being the only controller of the game, would send the game to all clients whenever there is a change happening in the game. All the clients would be able to update their view from the game they have received. The joiners, which also wish to control one of the spaceship in the game, they would send the corresponding messages to the server. When the server receives the message, he will update his own game according to the messages. Then he will send the updated game back to all clients.

iii. Disconnect: When the clients want to disconnect with the server, they would again send a disconnect message to the server and close their clientsockets. When the server receives this message, he will stop sending any updated game to corresponding addresses.

(b) **Justification of the choice:**
The start of the game is defined by successful connections; the middle of the game is defined by the playing/change state of the game; the end of the game from the client side is defined by the disconnection. Hence it is obvious and reasonable we split and distinguish the information being sent and received based on these three phases.

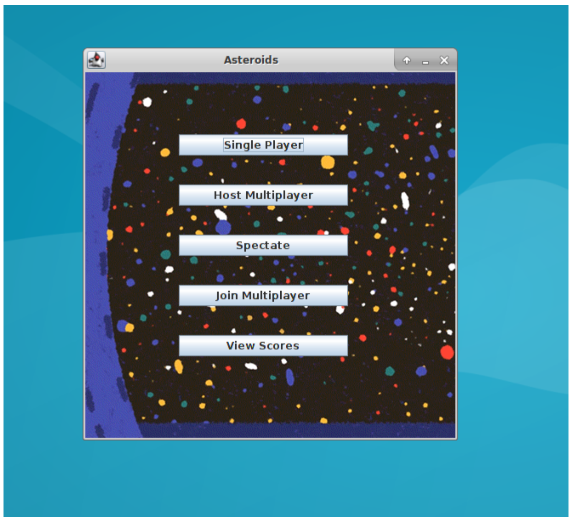
(4) **Comment:** we stick with *javadoc* format of comment. The doc can be found in *target/site/apidocs/* with command **mvn clean package javadoc:javadoc**.

3 Evaluation

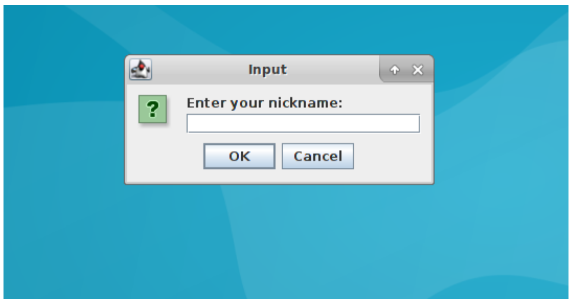
(1) **Stability of the implementation:**
Overall, the game runs smoothly for both single and multiple players, allowing a wide variety of actions ranging from hosting, to joining and spectating. What is more, there is a database containing the high score which is updated at the end of every game with the winner's score. The final version of the game runs smoothly without any bugs.

Some screen-shots of the final program are shown below:

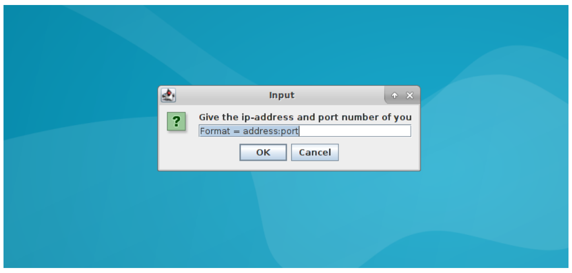
i. Start of the program:



ii. Ask user information:

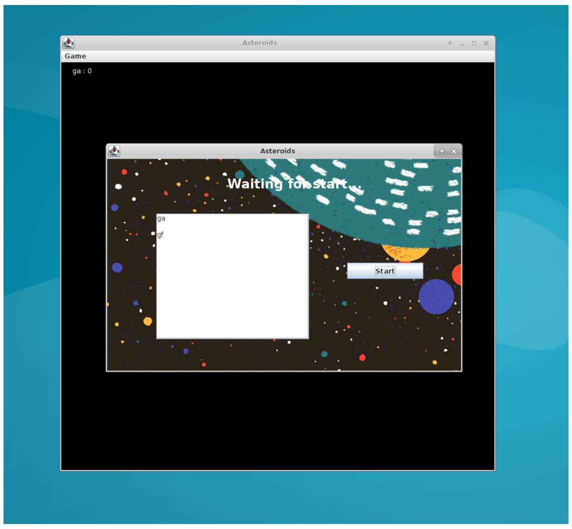


Enter your nickname:


Give the ip-address and port number of you

Format = address:port

iii. The host waiting for other joiners' connection:

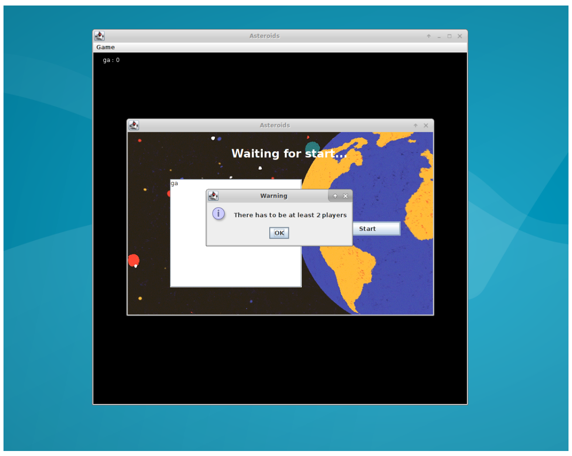

ga : 0

Waiting for start...

ga

gf

iv. When there is less than two players, the host wants to start the game:

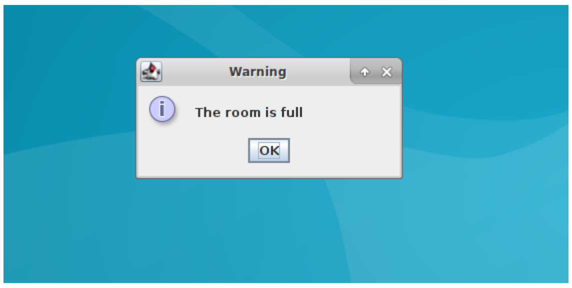

ga : 0

Waiting for start...

ga

There has to be at least 2 players

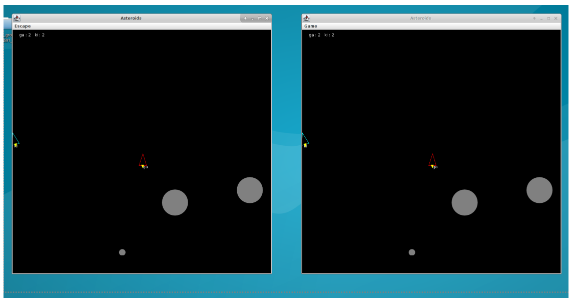
v. When the client wants to join a game which is already full of capacity:


Warning

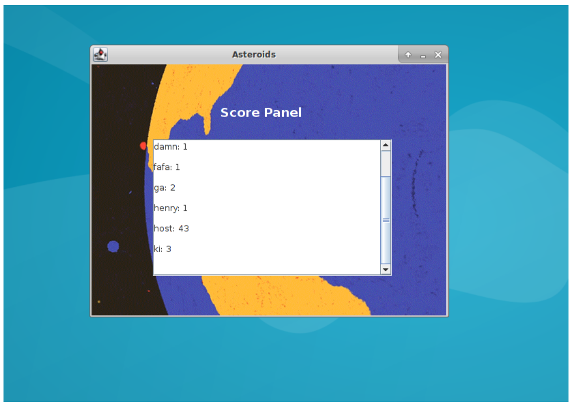
The room is full

OK

vi. Running of multi-player game:



vii. The ScorePanel:


Asteroids

Score Panel

damn: 1
fafa: 1
ga: 2
henry: 1
host: 43
ki: 3

(2) **Unimplemented feature:** We did not take care of the lost of packages in our project as we found it not very necessary as we tested our game for thousands of times. Such circumstances did not happen at all. Also if there is package lost, it is not a big problem, as soon newer packaged would be received.

(3) **Improvements:**
If we have been given more time, we would improve our projects from below aspects:

(a) For the joiner we would have added TCP for sending the connection and disconnection request. The reason TCP would be beneficial in this case is because, this request only gets sent once. The use of TCP makes sure this message would have been received.
(b) We would have liked to include other game modes and features, such as 1v1 matches.
(c) We could involve more design patterns to make our project structure clearer and more solid.
(d) a missing feature is clients dealing with lost packages. The reason this is not included is due to the fact that the game is constantly updated, thus one missed package not being significant.

4 Team Work

There are 6 main phases in getting this whole project done for our team:

i. Decide which networking protocol we would be used and what information should be sent between clients/server at different stages of the game. We get this done in the lab session under the help of TAs.

ii. Come up with the main structure to implement our idea. Again we get this done altogether, especially for classes like *Server*, *Joiner* and *Spectator*.

iii. Make the main structure and classes work. This took us a bit long and we did it separately in our private time. We took turns to look at, go over and make change of the code

iv. Add Database. We both tried to implement this part on each one's own class design and idea. In the end we stick with the implementation which works.

v. Debug the code to make the game run more smoothly. This part was mainly done by Ai Deng.

vi. Refactoring. This part was mainly done by Andreea Glavan.