4 | Asteroids

Given is a working single player version of the old arcade game “Asteroids”, see figure 4.1 for a screenshot. In this game the player controls a spaceship in 2D space. The goal is to not crash as long as possible, while asteroids continue to spawn. The player crashes by colliding with asteroids. The controllable spaceship is equipped with a gun that allows the player to destroy the asteroids, making it easier to keep flying. However, when asteroids are destroyed, they will spawn two smaller asteroids.

The supplied game allows the user to turn, shoot, and accelerate. A score system has been implemented, i.e. every destroyed asteroid will increase the score by 1. And there are menu buttons for restarting and quitting the game. Your task is to extend this program, such that the program offers multi player functionality over UDP.

Before you begin, study the provided source code and JavaDocs, such that you become familiar with the provided product. It is advised to play the game and try to recognise behaviour you've seen in the source code. You can package the JAR and generate the JavaDocs yourself using the command below. The JAR can then be found in the directory `target/` and the docs in `target/site/apidocs/`.

```
mvn clean package javadoc:javadoc
```

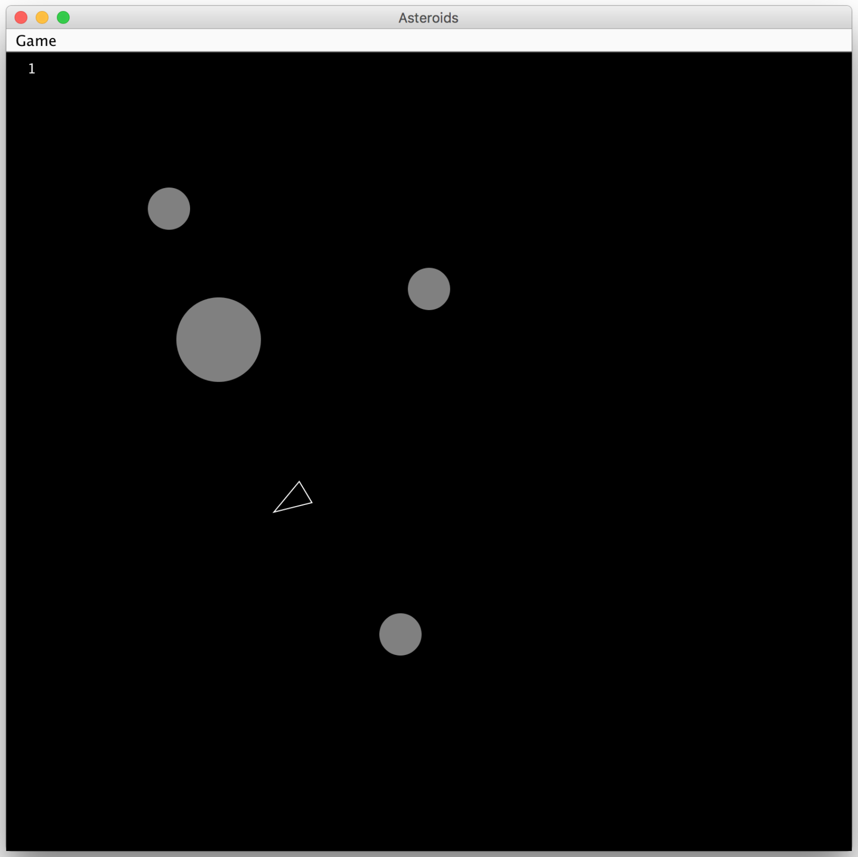

Figure 4.1: Screenshot of the provided Asteroids game.

4.1 | The assignment

4.1.1 | Requirements

Below you will find a list of minimal requirements for the extensions you will have to implement.

- Stable multi player functionality across multiple machines using UDP.
- A main menu with at least four options:
  - Start single player game.
  - Host multi player game.
  - Spectate multi player game.
  - Join multi player game.
- Spaceships should not collide with each other, i.e. they should be able to pass freely through one another. Bullets, however, should collide with and thus destroy all spaceships.
- A multi player game round should end when only one player remains alive, assigning a single point to the sole survivor.
- Each player should have a nickname and colour.
- High score persistence.
- Optionally, you may implement multiple different game modes.

To do: Update the GUI such that it always boots to a main menu, from which the four game modes are selectable. The host should display its IP address and port number. Selecting spectate or join should make the program ask for the aforementioned IP address and port number.

Make sure that it is possible to enter a nickname somewhere that is remembered by the game for as long as its running.

Optional: Make the player choose a colour as well. If you decide not to implement this, make the host in multi player mode assign a random colour to each connected player.

4.1.2 | Spectating

We suggest you start by implementing spectator functionality. In other words, that you will be able to see how someone else plays the game. Both on the host and client side you should start a normal game, except on the client side, you will not initialise a `Player` object, so that the client will not listen to any key presses and thus will not modify the model. Both game instances will have to update the game model on the same interval to introduce a so called game tick.

Every game tick the host should send the state of the model to the client. The client side should update the game model once it receives an update from the host. This way, the models of the host and all clients will be equal. However, it may occur that packages get lost. In this case the client side should update the model itself (using the `update` method).

The client should be able to send a spectate request to the host, as well as a disconnect request. Due to the variance in requests you should device a manner to differentiate between them.

To do: Create two classes that implement `Runnable`: `Server` and `Spectator`. The `Spectator` class should send (dis)connect requests to the `Server` class. And the `Server` class should keep track of the addresses of the spectators, such that it knows where to send the model to.

Hints: When using UDP, the packages may not arrive in the same order as they were send. Therefore you should add a reference number to each package, such that the spectator can ignore old packages and accept new packages.

Eventually multiple threads will need to access the model. You will have to make sure that this will not cause issues.

Think about how you can implement inheritance and apply where appropriate.

4.1.3 | Joining

After implementing the spectator you should be able to view what is happening in someone else's game. So essentially, the only thing you need to implement still is make the client capable of changing the model.

To do: Make sure that it is possible for multiple spaceships to exist in a single game object. In multi player mode, make sure that a point is awarded to the last standing player, instead of asteroid kills. Awarding a point here should reset the game, such that all players may join in once more.

Edit the view so that each player (spaceship) is displayed in a different colour. Also make sure that the scoreboard indicates the nickname of the players and prints each score in the colour affiliated with the corresponding player.

Create a new class `Joiner`, that sends the required information to the host so that the model may be updated accordingly. Make sure that the server keeps track of this information as well. Once again, this class should be able to send (dis)connect requests.

4.1.4 | Administration

Finally, each host should connect to a database in which it stores high scores attached to nicknames. This database should be persistent and reloaded every time the program starts. It is advised to use ObjectDB for this[1].

To do: Add possibilities to the GUI to retrieve the contents of the database and display said information in a menu. Also, store after each session or round the high scores of the players in the database.

4.2 | Submission

In order to submit the assignment, you are to create a pull request from the development branch into the master branch before the deadline[2]. After the deadline you will be enrolled into a demo slot, and we will communicate to you when you are expected.

You are expected to write a report on the assignment. The report should be easily findable within your repository. In the report we expect a short analysis of the problem and a more detailed design description. This design description should cover how the entire program is structured, where you have applied what patterns, and why you have made the decision that you have.

[1]See www.objectdb.com and the lecture slides.
[2]See Nestor for deadline.