Coursework report

Software Development 3 (SET09101)

Ivan ROGER (40285021)

I. Introduction

I.1. Abstract

This coursework was about the creation of game and the implementation of multiple programming techniques such as design patterns and threads.

This game has a really simple principle. The player acts as a space ship called the Mother Ship. He has to fight against three enemies: the SpaceFighter, SpaceShooter and SpaceCruiser. The only differ by their appearance. The outcome of a fight is ruled by the Mother Ship operationnal mode (Attack or Defence). For more details refer to the rules in Appendix 2.

You can find all the code for this coursework on the github repository: https://github.com/Ivan-Roger/NAP09101_CW

I.2. General conception

One of the main rules that I followed during the creation of this software is that the functionality must be divided and independent from the User Interface. This can be seen in the packages : there is a package for the functionnality called **core** with multiples sub packages and another package called **ui** where there is the main interface for Uis and packages for each implementation.

II. Programming techniques

This game has been developed in manner that it implements most of the techniques taught during this module. Here after is a list of the most notable ones.

II.1. Inheritance

In this game there are multiple class that are composed of the same element for instance the enemies are Space Ships, and the `MotherShip` too. So an abstract class was used to represent a `Ship`. And another one extending it was used to represent the `EnemyShip`.

Some other class also use inheritance, `Events` and `Actions` for instance. This allowed me to "program to the interface" meaning that I put the code that is shared in a single method in mother class and all the child classes can use it. It avoids code duplication and makes it easier to create a new child class.

Sometimes on the other hand, you do not want to write code in the mother class and just want to say that you want child classes to provide this method. In this case you can use an *interface* to list the methods that you want publicly accessible, regardless of how they work.

II.2. Pattern: Strategy

Sometimes when need a class to behave in different ways. In this case the `MotherShip` has two different behaviours, either it is in an offensive operational mode or in a defensive mode. We could simply use two different methods with a boolean to select which to use but if we want to add a new mode we would have to modify the entire code.

This is when the strategy patterns is useful. It consists of an *interface* specifying the methods that differ between modes and as many child classes as modes. Each class overriding the interface method with it's own behaviour. The englobing object (the mother ship) has a variable typed with the variable so that any of the child classes can be used (this is called *polymorphism*). When the mother ship has to fight it calls the fight method of the current behaviour object which performs the action. And this behaviour can be changed at runtime by changing the value of the behaviour variable.

II.3. Pattern: Factory

The previous patterns is really useful but when we want to create the mother ship object we also have to create a new behaviour object. And if the mother ship had multiple *Strategy patterns* the constructor would quickly become a big mess.

In this case the *Factory pattern* can prove very helpful. This patterns uses another class with *static* methods to help in the creation of an instance of an other object. In this example the *Factory pattern* has a `create` method that takes the value of an *enumeration* to select the behaviour to set at the construction of the object.

An enumeration (or *Enum*) is a pseudo class used to define a constant set of values. This one defined the two operational modes: attack and defence. Depending on the value of this enum the factory class will select the corresponding behaviour to use.

II.4. Pattern: Command

In software like this one sometimes it gets risky and complicated to allow other classes to perform modification instantly or without control. This is a case where the *Command pattern* can be used.

The command pattern is composed by a controlling class and multiple action classes. Each action class inherits from a common interface and implements it's action method. External classes can create action object and submit them to the controller who is going to execute them when it considers it's the time.

In this game when a ship spawn or moves they use this pattern. They create an action and submit it to the wrapper (which transfers it to the worker). The action are only executed when the controller decides to. (More details about the worker in the section II.6 – Threads).

II.5. Pattern: Observer

This game has been created so that it can work with any user interface. (Details in section III – User interfaces). Those interfaces need to update themselves to stay up to date with the internal state of the game. To do so I used the *Observer pattern*. This pattern is composed of one observable class and observer classes. Observers are registered to the observable class and when the observable object is modified it notifies it's observer by calling an `update` method. This method is implemented in each observer thanks to an interface.

To transfer the data I created multiple `Event` classes to tell the user interfaces to update themselves. Each event has it's own data and information and all the observer registered are receiving the same `Event` object. The different event types are listed in an *Enum* and each event class has a method to tell what it's type is. This allows the user interface to know what event it is and if it not relevant, to ignore it. This could also be improved by providing a list of event types when subscribing to the observable and then to only dispatch the event if it matches the selection.

Here are some of my event classes: `GameStartEvent`, `NewTurnEvent`, `ShipSpawnEvent`, `ShipMoveEvent`, etc ...

II.6. Threads

Since the user interfaces and the game are completely separated I also used threads so that the game can process and run in a different thread without having to wait for the user interfaces.

To implement theses threads I used the natives methods `wait` and `notify`. The game thread is located in a class called `GameWorker` that is a loop started at the beginning of the game. This loop `wait`s at the begging of each turn to be woken up by the user (via the `notify` method).

II.7. Exceptions

Sometimes there are some pre-condition that are not respected or the result of a method call isn't what we expected. To deal with theses problem I used specific exceptions.

For instance, if a parameter in a method isn't valid I throw an `InvalidArgumentEx`. This allows the caller to catch the exception and deal with the problem to solve it. If the exception is not caught soon enough then it is sent to interfaces and if it is considered as critic the game is interrupted.

III. User interfaces

The user interfaces beeing completely independent from the core there was the possibility to connect any UI to the game via the observer pattern (CF. Section II.5 – Pattern: Observer).

However I had to make sure all UIs where compatible with the core so there is a common interface implemented by all the UIs called `UiWrapper`. This interface defines the observer methods to listen to the events (CF. Section III.1 – Events). And the common methods used by the core in the different UIs.

III.1. Events

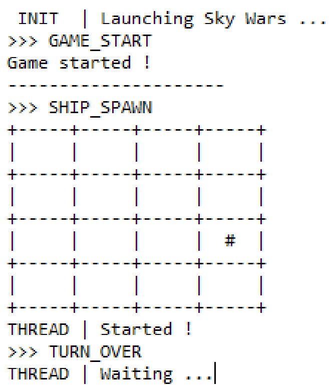
As explained in section II.5 I used events to tell the user interfaces what to update. I will describe the events I designed and explain what they are used for:

- ***EXCEPTION:*** This event is sent when an exception is not caught soon enough. The UI should show to the user an error message. If the exception is critic the game will be closed.
- ***GAME_START:*** Dispatched when the method `startGame` is called.
- ***NEW_TURN:*** After the player calls the `play` method.
- ***SHIP_SPAWN:*** When a new ship is placed (`ShipSpawnAction`).
- ***SHIP_MOVE:*** When a ship moves (`ShipMoveAction`).
- ***FIGHT_START:*** When there is at least one enemy and the mother ship in the same tile.
- ***SHIP_DESTROYED:*** During a fight, when the `destroy` method is called on a ship.
- ***SHIP_REMOVED:*** Most of the time after the *SHIP_DESTROYED* event. When the animation of destruction should be finished (500ms).
- ***FIGHT_END:*** After the fight ends, contains a boolean to tell if the player won.
- ***TURN_OVER:*** After the fight as ended, tell the UI to allow the player to start a new turn.
- ***GAME_OVER:*** When the player won or lost.
- ***GAME_QUIT:*** When the method `stopGame` is called.

III.2. Text Interface

At first when I was building the board I just just working with a text user interface. But this has been useful afterwards because it allowed me to make sure that the game was working with any user interface (multiples at the same time and even none).

This text UI was just printing the board and the number of enemy on each cell.



```
 INIT  | Launching Sky Wars ...
>>> GAME_START
Game started !
---------------------
>>> SHIP_SPAWN
+-----+-----+-----+-----+
|     |     |     |     |
+-----+-----+-----+-----+
|     |     |     |     |
+-----+-----+-----+-----+
|     |     |     |  #  |
+-----+-----+-----+-----+
|     |     |     |     |
+-----+-----+-----+-----+
THREAD | Started !
>>> TURN_OVER
THREAD | Waiting ...
```

Illustration 1: Text UI

III.3. Graphical Interface

The Graphical User Interface (or GUI) took quite a long time to design. I had to take into account every information that to be displayed. I didn't use the Eclipse interface creation tool. I used layout managers to dispose elements around.

You can see the window captures in appendix 3.

I created multiple classes extending native elements to represent items I needed. This is the hierarchy of my elements:

GraphicUI [*UiWrapper*] is the wrapper for the interface,
GameFrameUI [*JFrame*],

- BoardUI [*JPanel*]: Composed of BoardTileUI [*JPanel*]
- ControlsUI [*JPanel*]: Composed of multiple native elements in differents Panels

The Board is a grid that contains an array of Tiles. Each tile (`BoardTileUI`) contains a cell for the `MotherShip` and three cells for each type of `EnemyShip`. Each ship has a *png* image to represent it displayed in a `JLabel` as an Icon. The labal is set to visible or not depending on the presence of the ship. Right next to each enemy icon there is a counter to display how many enemies of this kind are on this tile.

Since the size of the board can be configured the window size adapts it's size to make sure that tiles keep the same size.

On the right side are the controls and statistics about the game. In a large font at the top we can see which we are currently in. Underneath it there are two counters display the number of enemies currently alive and the number we have killed.

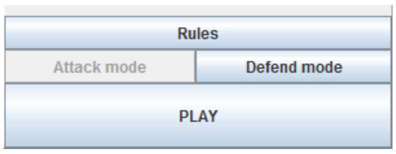


Illustration 2: Control buttons

Next there is a log of what appends in the turn. When a ship spawns the board is updated and the log is appended with a message. When a fight starts another message is displayed and at each enemy we kill the log is updated.

These are updated thanks to the events sent. (refer to section III.1 – Events)

During a fight there is some delay at the start to make sure that the player sees the content of the tile. Then the fight starts and ships are destroyed one after the other by replacing their icon by a blow icon. (CF. Illustration 3).

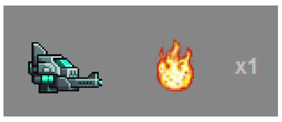


Illustration 3

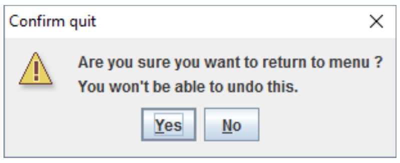
The counter is decremented as the ships are destroyed.

If the player wants to change the mother ship operational mode he can select by clicking the Attack or Defence button (CF. Illustration 2). He can also open the rules with the corresponding button.

When the turn is over (Event: *TURN_OVER*) the “PLAY” button is released and the player can click it to start the new turn. As soon as the button is hit the new turn starts and the buttons is disabled.

When there are no other enemies on the board the player wins, if the mother ship is destroyed during a fight the player loses. I both cases the *GAME_OVER* event is dispatched and the result is displayed in the GUI. The play button is replaced by a “QUIT” button that takes the player back to the menu.

If the player tries to close the window during the game a pop up appears asking a confirmation (CF. Illustration 4). If the play says yes the game is closed and the Menu is shown again. If the game is over when closing the window no confirmation is asked.


Confirm quit

Are you sure you want to return to menu ?
You won't be able to undo this.

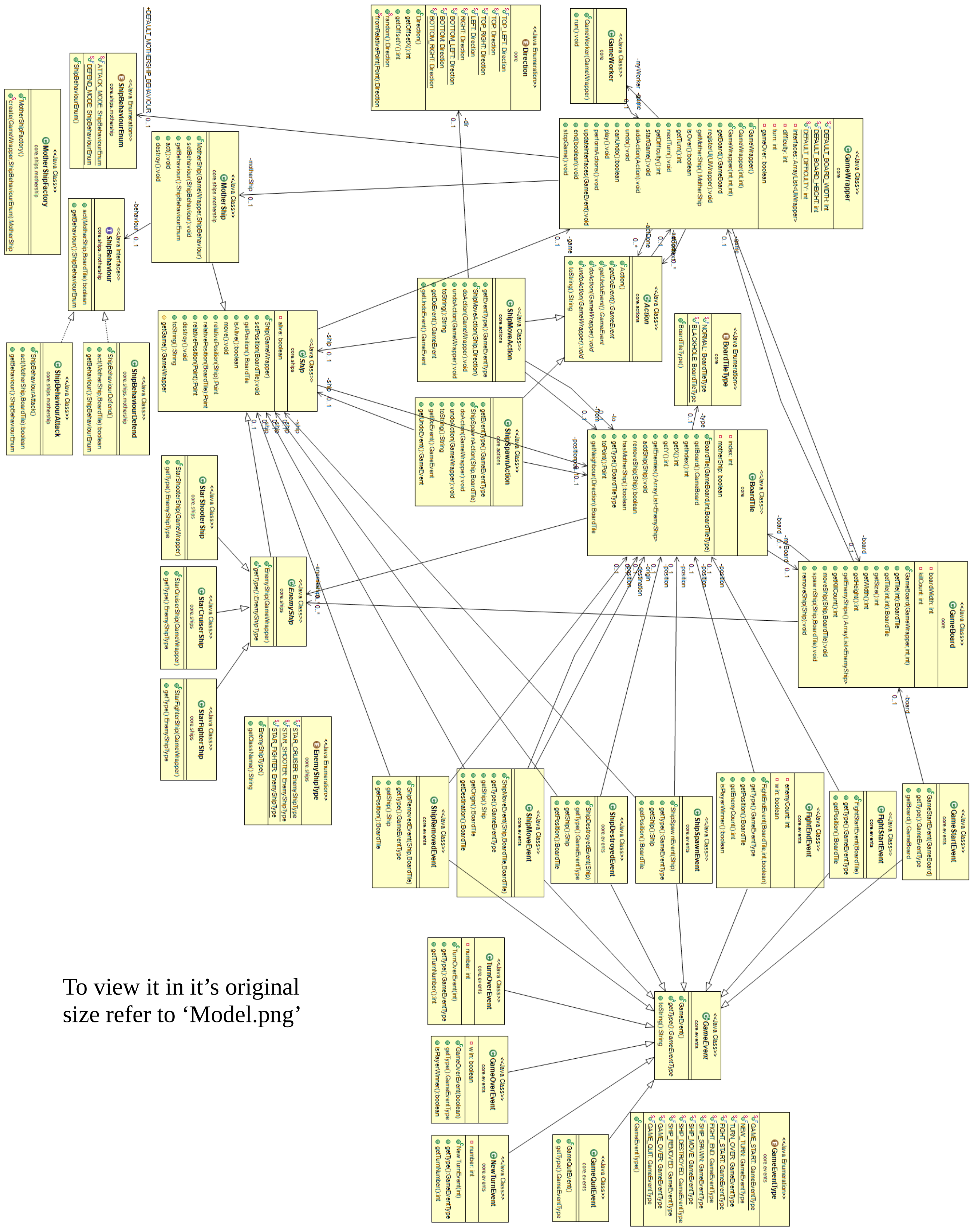
Illustration 4: Capture of the Pop Up when closing Game

IV. Appendix

IV.1. Rules

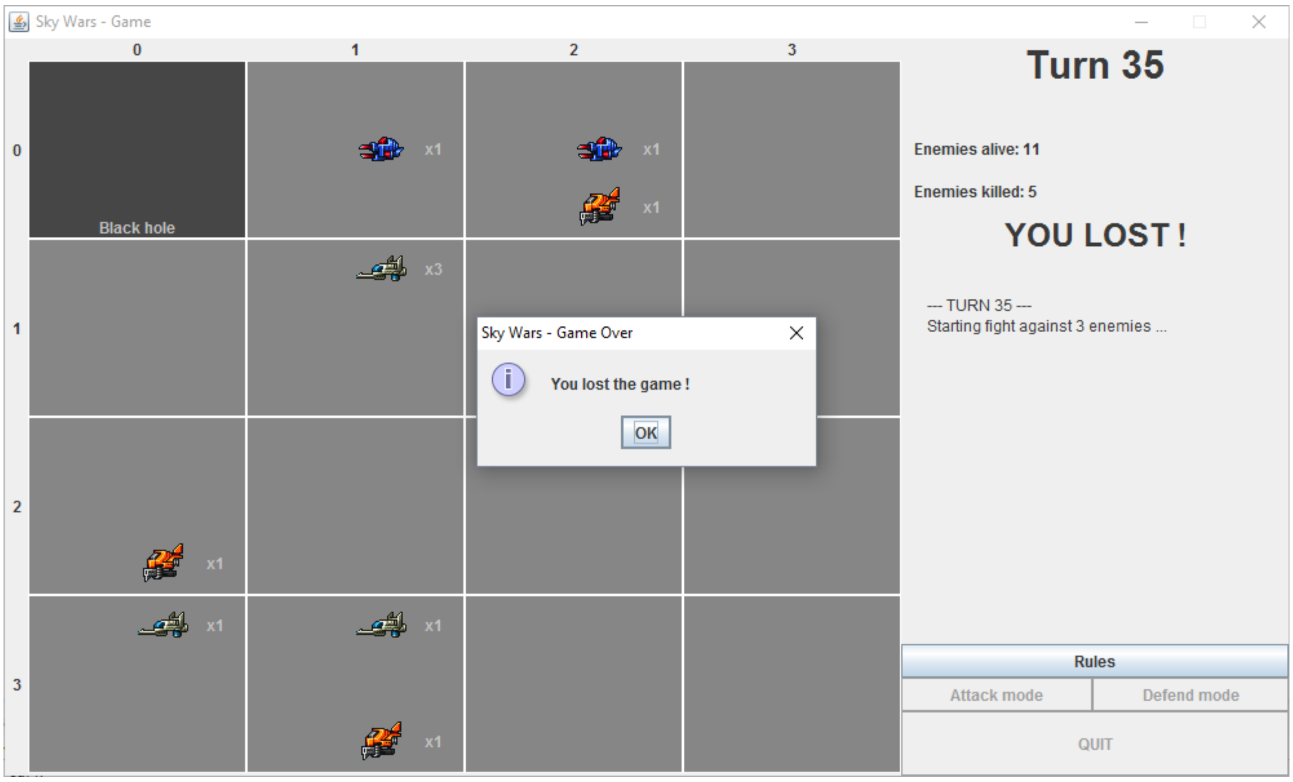
```
                              By Ivan ROGER
                             ----------------

This game is simple. It's aim is to stay alive as long as you can. You are playing as a
spaceship called the Mother Ship.

You will have to fight against three kinds of enemies coming out of the black hole:
SpaceShooter, SpaceCruiser and SpaceFighter. They only differ by their appearance.

You are going to move through the space grid to fight and run away from your enemies. To
do so you only have to hit the play button. The Mother Ship will move randomly.

If you enter in the same cell than an enemy you will have to fight against him.
The outcome of this fight depends on the Mother Ship operational mode.
You can switch between the offensive mode and the defensive mode.

By default the attack mode is enabled. This means that you can destroy up to three other
enemies. If there are more you will die.

In the defense mode you can defend against one or two enemies. More and you die.

GOOD LUCK, And have fun.
```

IV.2. UML Diagram



To view it in it's original size refer to 'Model.png'

IV.3. Graphical UI



Turn 35

Enemies alive: 11

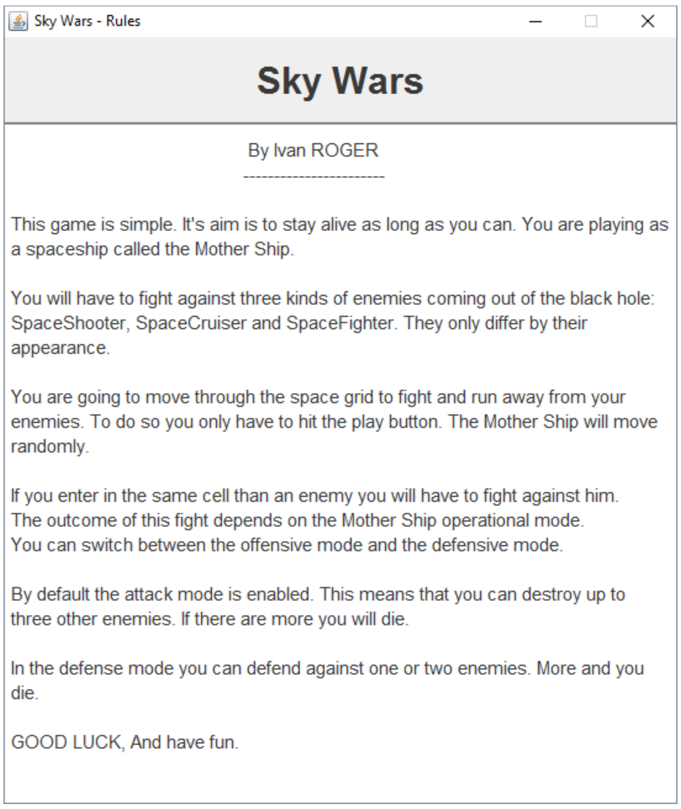
Enemies killed: 5

YOU LOST !

--- TURN 35 ---
Starting fight against 3 enemies ...

Sky Wars - Game Over

You lost the game !



Sky Wars

By Ivan ROGER

This game is simple. It's aim is to stay alive as long as you can. You are playing as a spaceship called the Mother Ship.

You will have to fight against three kinds of enemies coming out of the black hole: SpaceShooter, SpaceCruiser and SpaceFighter. They only differ by their appearance.

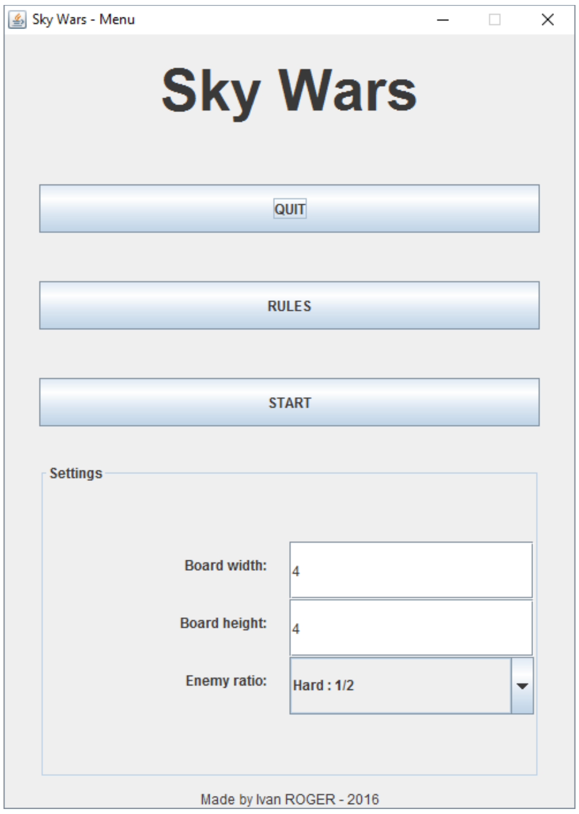
You are going to move through the space grid to fight and run away from your enemies. To do so you only have to hit the play button. The Mother Ship will move randomly.

If you enter in the same cell than an enemy you will have to fight against him.
The outcome of this fight depends on the Mother Ship operational mode.
You can switch between the offensive mode and the defensive mode.

By default the attack mode is enabled. This means that you can destroy up to three other enemies. If there are more you will die.

In the defense mode you can defend against one or two enemies. More and you die.

GOOD LUCK, And have fun.



Settings

Board width: 4

Board height: 4

Enemy ratio: Hard : 1/2

Made by Ivan ROGER - 2016