

Square Strategy

Leo Blomqvist 100684892

Computer Science (Bachelor level), 1st year

18.4.2024

1. General description

The end product is a very small-scale strategy game in which the player and an AI complete turns controlling a team of four characters with different stats and that have abilities with differing effects like statuses. The battle continues until all the characters in one team have fully depleted hp, which is when the player may close the game.

The battle happens on a grid-like structure that contains randomly placed obstacles.

Originally the idea was to have different levels but due to time constraints variability is handled with randomness. The starting orientation of a battle is different each time the game is launched.

In my opinion my project is completed on the difficult level since it contains all the elements required for the topic, is quite easily extendable through minimal extra work and has a pleasant enough GUI to enhance playability.

2. User interface

The game is simply launched using object Main in the source code. After launching the screen will display a battleground of size 20 x 15 with randomly placed characters and obstacles and a sidebar that contains information about the characters in the battle.

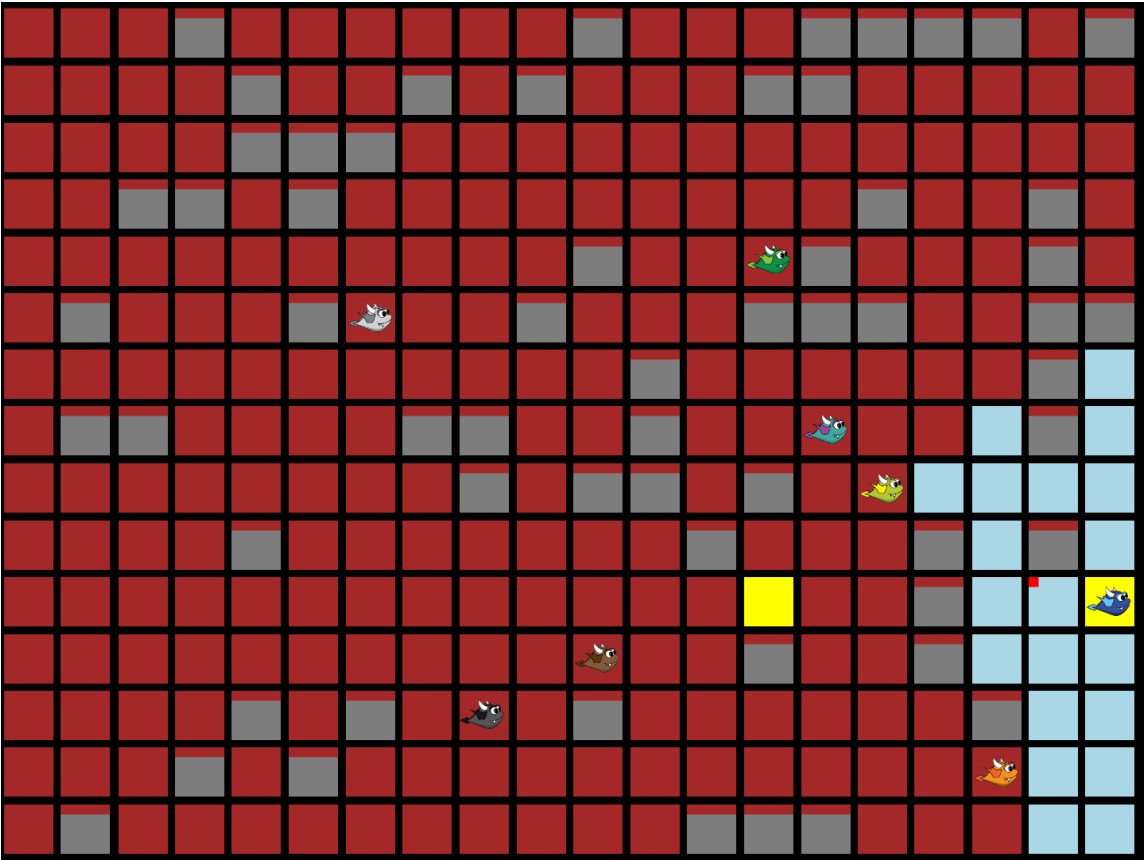
When hovering the mouse over any square it will turn yellow to indicate that, if clicked, the mouse would try to select that square. However, clicking only does something when hovering over a square that contains one of the characters belonging to the team of the player. Doing this “locks” the square so that it will always be displayed as yellow until it’s unlocked.










Selecting one of these squares (that contain a character) will also display some of the squares around it as light blue to indicate that by clicking one of them, the selected character will move to that square. In addition, some buttons, and the character selected will appear on the sidebar.

These buttons will each do something when clicked. They may be used to end the character’s turn, unlock the current locked square or select, rotate and use an ability that the character has. Ending a character’s turn, using an ability or moving it to one of the light blue squares also unlock the current locked square.

On the battleground squares with gray boxes are considered obstacles and characters can't pass through them. Also, once an ability is selected, its area of effect will be displayed as small red squares in the upper left corner of all affected squares.

This is basically how the game is played. The player will move (and use an ability) all of their characters during their turn until each character can't do anything anymore this turn or their turn has ended prematurely with use of the relevant button. Then the player will see a short animation that shows what the AI has done on its turn and the cycle will repeat until one team loses.



| | |
|---|---|
|  Jouko Armor: 20 Statuses: <div></div> |  Marko Armor: 40 Statuses: <div></div> |
|  Pertti Armor: 10 Statuses: <div></div> |  Tarmo Armor: 15 Statuses: <div></div> |
|  Marko Return End Turn Pyromania Stab Burst Rotate Use Ability | |
|  Johanna Armor: 20 Statuses: <div></div> |  Martta Armor: 40 Statuses: <div></div> |
|  Piia Armor: 10 Statuses: <div></div> |  Tiina Armor: 15 Statuses: <div></div> |

3. Program structure

The fundamental structure of the program is very similar to the one that was originally planned but for example some method implementations are in another trait or class than originally planned. The class `Battle` is still pretty much the glue between all the parts of the program as it is also the source of all information for the GUI classes through the object `Main` which has the current `Battle`-instance as a variable. The central play-method is also contained here. This method controls the turns of between the player and the AI and keeps all information around the classes up to date and correct.

The class `Battleground` is quite similar to what was originally planned and it contains the grid-like structure of squares on which the battle takes place. The class has three very important methods, namely: `getSquare` which returns a square of desired coordinates from the grid, `squaresWithinRadius` which returns the squares that can be reached in a given number of steps from a given starting square, and finally `squaresAlongPath` which returns the step-by-step path that one must take to reach a given square from another.

The class `Square` describes one of the squares in the whole battleground of the battle. It has specified coordinates and can for example investigate whether or not it contains a `Character`-instance or simply any other actor. It can also find its own neighbors from the battleground that it's part of. In addition, it can find a given amount of squares in a given (enum) `Direction` relative to itself. It could be possible to implement some of the methods in `Square` in `Battleground` and vice versa, however, the current setup feels quite intuitive, and the code in these classes is quite dry at all as a result of the current implementation.

The trait `Actor` describes the qualities that are common to all actors that can occupy a `Square`-instance. Most importantly the movement of all actors is done with the `followPath`- and `pushedTo`-methods along with the variables `onTheMove` and `currentSquare`. All actors also know which battle they are a part of. This is because it proved useful in making the GUI classes implementations more simple and it has been useful elsewhere as well, such as the trait `Ability`.

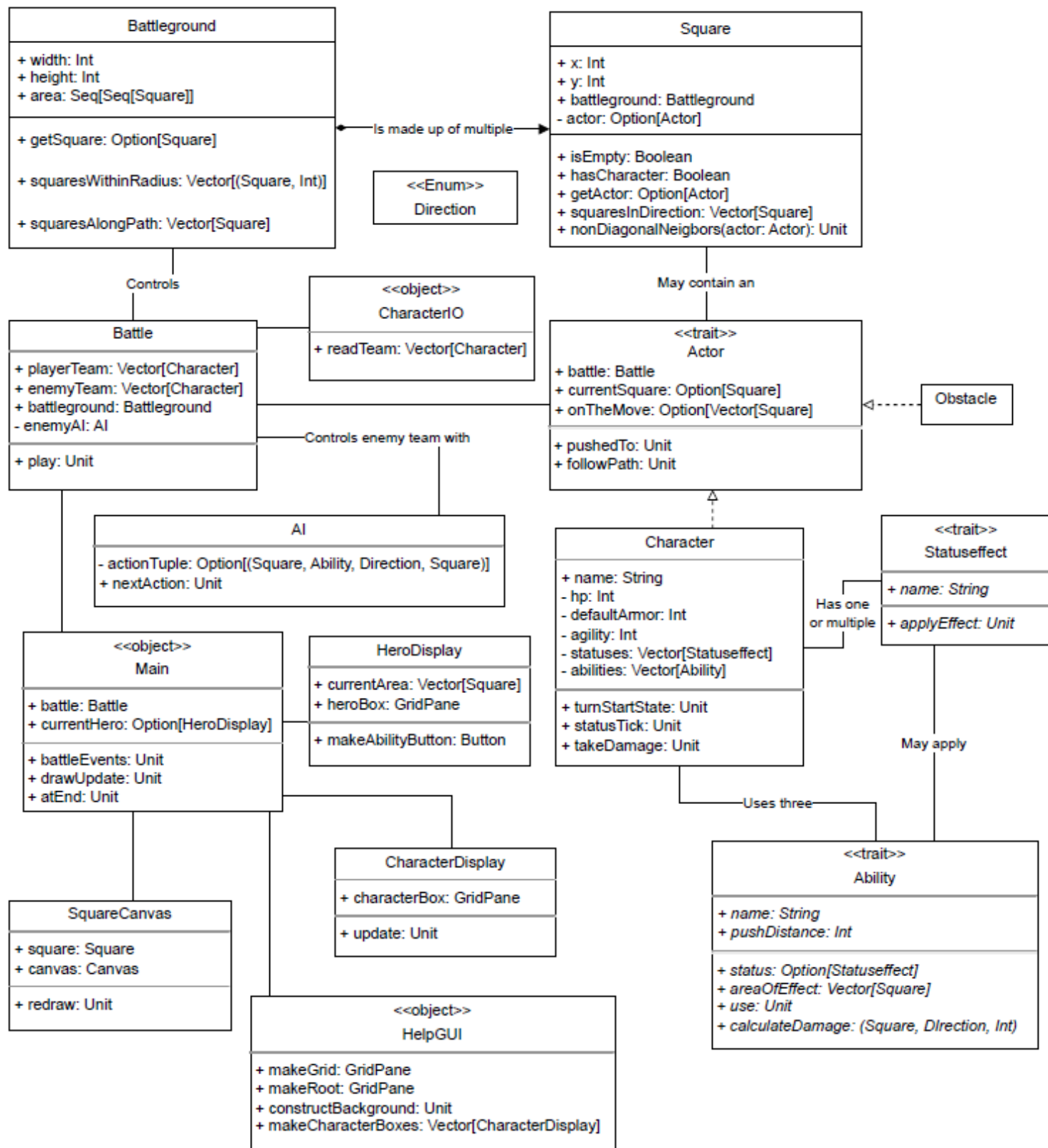
The class `Character` then is a subclass of trait `Actor` but contains lots of extra information about a controllable character. It also has some helpful methods to keep all information correct during a battle. For example, the `turnStartState`-method which sets some of the variables associated with the character to a state that is correct for the start of a new turn for the player or enemy AI. Characters can also make their obtained statuses tick and have themselves take damage. While on the subject of characters, the object `CharacterIO` and its `readTeam`-method is responsible for reading character information out of external files before the start of a battle.

The trait Ability is a very important one since the main meat of the program is the usage of abilities and battling with them. Each ability object that extends this trait will implement its own area of effect and what exactly it will do when it hits a target. This trait is very important also because it gives big help to the enemy AI with the method calculateDamage. The trait Statuseffect describes effects that may be inflicted upon characters when applied by abilities. They have a set duration which ticks down each turn until the status will be removed.

The class AI has only one relevant public member which is the nextAction-method. The Battle instance calls this method to execute an action for one character of the enemy team. An action can be either movement to a square calculated inside the method or the usage of an ability that the current character has.

The GUI is controlled by the object Main which has a Battle-instance to draw information from as mentioned before. The object HelpGUI contains a couple methods that are useful in handling the GUI, especially the makeGrid-method which can return a GridPane-instance of specified proportions. Main also includes the battleEvents-method which handles most actions that the player can do with the mouse. In addition, the drawUpdate-method makes sure all the most recent and accurate information is displayed on the GUI.

The classes HeroDisplay and CharacterDisplay are used to present character information in a neat and tidy way. HeroDisplay-instances are also used to control some of the characters' functionalities in the GUI.



4. Algorithms

In my opinion the program contains two major algorithms worth describing in this section. Firstly, the pathfinding used to move characters around each other and obstacles is not quite trivial. The algorithm can be seen as having two steps:

First, select the starting square and assign it the number it takes for you to move to that square (naturally 0), then take all the non diagonal neighbors of that square (character can't move diagonally) and assign them the next number (1). Then take all squares that have been assigned one, and take all of their non diagonal neighbors and assign the next number (2). Continue this cycle (while avoiding squares that have been assigned a number in a previous step) until your whole collection of squares contains the one you want to move to.

Then all you have to do is look at the number assigned to your destination and move backwards to an arbitrary square that has been assigned the next lowest number. Do this until you reach zero while taking note of the squares you passed through and you have created a path that avoids any and all obstacles.

Secondly, the algorithm the AI uses to control its subjects. It handles the characters one at a time so let's look only at how it does that. Firstly, the character may only move to some of the squares on the battlefield. Secondly, only some of those squares will allow it to have an effect on something with one of its abilities. Consequently, it is useful to look at the abilities of the current character in isolation and see how effective they may be:

A target can only be hit by an ability in the squares, that if the target itself would be using the ability, would be hit by the ability. Thus, it's best to reduce the amount of possible movement squares to those which would be hit by a target if it used the ability in its current location. Then, for each of the possible locations of movement you could simply see what the current ability would do if it were used there. This is where the AI makes some abstractions about the effectiveness of abilities.

Naturally damage done can be represented by the amount of damage itself, however, I decided that applying an effect would equate to 50 points of damage, and pushing an object (not just characters) would equate to 10 points of damage. This way the AI can also get itself unstuck if one of its characters gets walled off.

When all of the possible squares have been checked with each ability we simply pick the pairing of square and ability that was the most effective. At the end of its turn the AI character wants to avoid as much harm as possible, so it simply finds the square that is on average the furthest away from all of the player's characters.

This is done with the Pythagorean theorem: $a^2 + b^2 = c^2$

5. Data structures

Since I'm not yet fully familiar with the advantages of each data structure that Scala contains I decided to favor the one quality I appreciate, namely immutability.

This led me to mostly use vectors all around the program, which I recognise might not be optimal in terms of resource management especially when it comes to vectors containing a huge amount of squares. In these instances an array might have been more correct, but I do appreciate the uniformity of the end product. Arrays could definitely have been used in the class Battleground to store all the squares. This I implemented with sequences, which maybe originally happened because arrays didn't have a method that I wished to use. Some methods also use buffers with for- or while-loops to store info things such as redundant squares. In the end, I think using mostly vectors did help me in some cases of debugging because I didn't have to consider the factor of mutability.

6. Files and Internet access

The program uses files for one thing which is creating instances of the class Character. More precisely, one file creates an entire team of four characters, thus, two files have to be read to initiate a battle. The file format is shown at the of this section. In it, all attributes that a character has are defined by a 3-letter id (this simplifies the extraction of information). After each attribute name and value, follows a colon (:). This is what the program uses to identify when one attribute is read. The same principle works when abilities are split at the comma (,). Finally, the four characters are split by a row of three dashes (---) so that the characters can be easily separated from each other and the readability of the file increases.

```
1  NAM Johanna:
2  PIC greenboy:
3  HPO 1000:
4  ARM 20:
5  AGI 20:
6  ABI Pyromania, Stab, Burst:
7  ---
8
9  NAM Martta:
10 PIC yellowboy:
11 HPO 1500:
12 ARM 40:
13 AGI 5:
14 ABI Pyromania, Stab, Burst:
15 ---
16
17 NAM Piia:
18 PIC brownboy:
19 HPO 700:
20 ARM 10:
21 AGI 30:
22 ABI Pyromania, Stab, Burst:
23 ---
24
25 NAM Tiina:
26 PIC blackboy:
27 HPO 1200:
28 ARM 15:
29 AGI 15:
30 ABI Pyromania, Stab, Burst:
```

7. Testing

A huge part of the testing was done through the GUI as it was also one of the parts that was implemented near the beginning and gradually evolved when new parts of the internal logic were implemented. Testing occurred naturally as the logic and GUI progressed. Also, because I intentionally made the program controllable only with the mouse, I did not have to worry too much about false and empty inputs and the like. The only part of the program that I used unit tests on is file reading and thereupon based character creation. All in all unit testing was minimal but in my opinion the nature of the program steered me naturally toward GUI-based testing.

8. Known bugs and missing features

As far as I'm aware, the current version of the program doesn't have any bugs. Don't quote me on that however, since testing could always be more thorough. Regarding missing features, I would say that there definitely is a whole lot that could be done to improve the playability and variability of the game. In addition, the GUI could have some more functionalities and display some more information like what a chosen ability will do or whether or not a character has ended its turn.

Originally I also planned to have cooldowns for abilities, which could be implemented but it would be more interesting after more abilities have been added. Also the strategic aspect of the game could improve greatly with cooldowns and being able to use, for example, two abilities in one turn. This would, however, also require some changes in the implementation of the AI but I gather it would not be all that horrible.

9. 3 best sides and 3 weaknesses

In my opinion one major strength of the current program is how easy the implementation of new abilities and status effects would be. You can see in the current code that especially for abilities the central trait has a huge implementation but the actual abilities could be fit into six simple lines if desired. I think I have done a great job at making universal methods that can be called with just a few extra parameters to have a completely different end result. This is also partly due to some of the methods in class Square that come in handy. Secondly, I think that the pathfinding system was in the end done quite elegantly with recursion, and a fully reliable and accurate pathfinding was fit into two methods (them being `squaresWithinRadius` and `squaresAlongPath`). Finally, I think that the animation of movement is quite clever, as all it took was creating a simple animation timer, moving actors on their path one square at a time. In the case of the AI, all it took was to remember to do nothing when any of the characters is still moving.

In terms of weaknesses, I think I touched on one in the section about data structures. I believe that my use of different data structures is quite weak to say the least, but my excuse is that I haven't yet taken the course called "Data Structures and Algorithms". Hopefully something will be learned there. Otherwise, I think that the trait Actor and its subclasses probably could have a cleaner implementation not containing so many tiny methods and variables. This would probably sort itself out given some time dedicated to it. In addition, some of the code might be quite hard to read as I didn't have too much time to document it and the object Main could definitely still be divided into smaller subsections of the program.

10. Deviations from the plan, realized process and schedule

I started implementation from classes Battleground and Square as I had planned, because they seemed to be the basis for almost everything else. Quite quickly after that though I started to work on the fundamentals of the GUI since it felt natural to test all of the internal logic with it. This was not in the original plan.

After that I worked for quite long on the pathfinding algorithm (with obstacles) until coming up with the current solution. After that I started to implement the class Character to be able to properly design a GUI that would show all relevant information.

I believe that at this point I already had the barebones of the class Battle but it had not really been used for much as of yet. Then I worked on the abilities and on assigning their use to the GUI. I had pretty much finished everything except Battle, AI and Statureeffect. The latter two were a consequence of the first since they could not be properly tested without a proper turn structure.

All in all, I would say that I had a decent idea about the order of implementation but some clear things I missed, like the fact that taking turns would probably have to be last, and the consequences of that. I would also say that the total amount of work was maybe not as much as I imagined but I also didn't have as much time as I'd imagined, so every possible moment had to be used. This is also why I wouldn't feel comfortable estimating the workflow of the two-week sprints as I basically used my time as best I could and some weeks I simply had no time at all.

Maybe the biggest point of learning was how important the GUI is to testing and how much more motivating it is to code the internal logic when you can immediately see the result on screen.

11. Final evaluation

All in all, I would say that the program is quite good, although I maybe hoped that I would have had time to implement more stuff (such as abilities and statuses) to make the actual game more fun. Although on the flipside, I think my code is quite dry for the most part and that the extensions I wish to make can be made with as little effort as I could hope.

As previously discussed, the chosen data structures are a major weak point and in addition to that I really think that the biggest downfall is that there are a lot of features that could have been added like saving a game file, a level select screen, a smoother user experience and in general more content. Also I would say that the AI-implementation is not the most friendly to changes in implementation, like being able to use two abilities in one turn, but I also think it could be much worse.

In addition, it is possible that some of the classes could be cleaned up by refactoring some of the internal logic so as to not need so many variables, but I have already done quite a bit of that in the process of implementing new functionalities. In summary, I'm happy about the amount of features that I managed to implement in at least a passing manner and about the actual design of my game, which was also quite a big part of this specific topic.

If I started from the beginning, I would definitely look up a proper pathfinding algorithm immediately so as to not waste so much time on it and I would

implement the Actor-trait better and more universally from the beginning because in the end I had to make quite a few changes to it along with the Character-class.

12.References and code written by someone else

Materials for the following courses:

Programming 1, Programming Studio 1, Programming 2, Programming Studio 2









Inspiration for pathfinding:

<https://theory.stanford.edu/~amitp/GameProgramming/AStarComparison.html>

In addition documentation for ScalaFX, Scala, JavaFX, and some pages on Stack Overflow (no copied code)

13.Appendices

One more screenshot of a battle in progress:

| | |
|---|--|
|  Jouko Armor: 20 Statuses: <div><div></div></div> |  Marko Armor: 25 Statuses: Burn, Bleeding <div><div></div></div> |
|  Pertti Armor: -5 Statuses: Burn <div><div></div></div> |  Tarmo Armor: 0 Statuses: Burn <div><div></div></div> |
| | |
| | |
|  Johanna Armor: 5 Statuses: Bleeding, Burn <div><div></div></div> |  Martta Armor: 25 Statuses: Burn <div><div></div></div> |
|  Pia Armor: 10 Statuses: <div><div></div></div> |  Tiina Armor: 0 Statuses: Burn <div><div></div></div> |