Strategy Game

Ilmari Oivanen

25.4.2023

General description

My project is a fantasy themed turn-based strategy game with an AI opponent. The player commands a group of different characters with different skills to use in combat. For example some characters can support the others with positive effects such as healing or damage boost while other characters excel in dealing a lot of damage to the enemies. In total the amount of characters is bigger than the maximum group size of three, so the player can build their team in multiple ways. There are multiple stages to choose from with different backgrounds and stage effects that alter the gameplay.

I originally planned to make this game a roguelike but I changed my plans and decided to make the game a single match against the AI, instead of multiple subsequent encounters. The player can choose their characters and the stage, but the AI opponent will get a number of random characters so that the party sizes match. This means the player can choose to play 1v1, 2v2 or 3v3.

Characters, skills, and stages were implemented extensibility in mind so the addition of new content should be fairly easy. Progress during a match can be saved to a save file and an existing save file can be loaded at any time. By editing the save file the user can change the game state, stage, and the characters in either party. Characters' skill cannot be changed this way.

User interface

The program is not executable as that was not required. It can be launched by running the GameApp object in src/main/scala/GameApp.scala. When launched, the game menu will open. Some screenshots of the GUI can be found in the appendixes.

In the game menu screen, there are four buttons. The user can continue a game, start a new game, save a game file and load a game file. The game will react and inform the user if there's no game to continue or if saving/loading the game failed. The user can return to the menu by pressing Esc on the keyboard anytime. If continuing a game is possible, the game view will open and the match will continue from the point of saving or exiting. If a new game is started, old match data is deleted and the stage selection screen is shown.

In the stage selection screen, there are buttons for each available stage and a "next"-button. The user can choose the stage by pressing the stage buttons. When a stage has been selected, the "next"-button can be pressed to get to the character selection screen.

In the character selection screen, there are buttons for each available character, a button for clearing the user's party, and a button to start the game. When at least one character has been added to the party by pressing the buttons, the user can start the game which will open the game view.

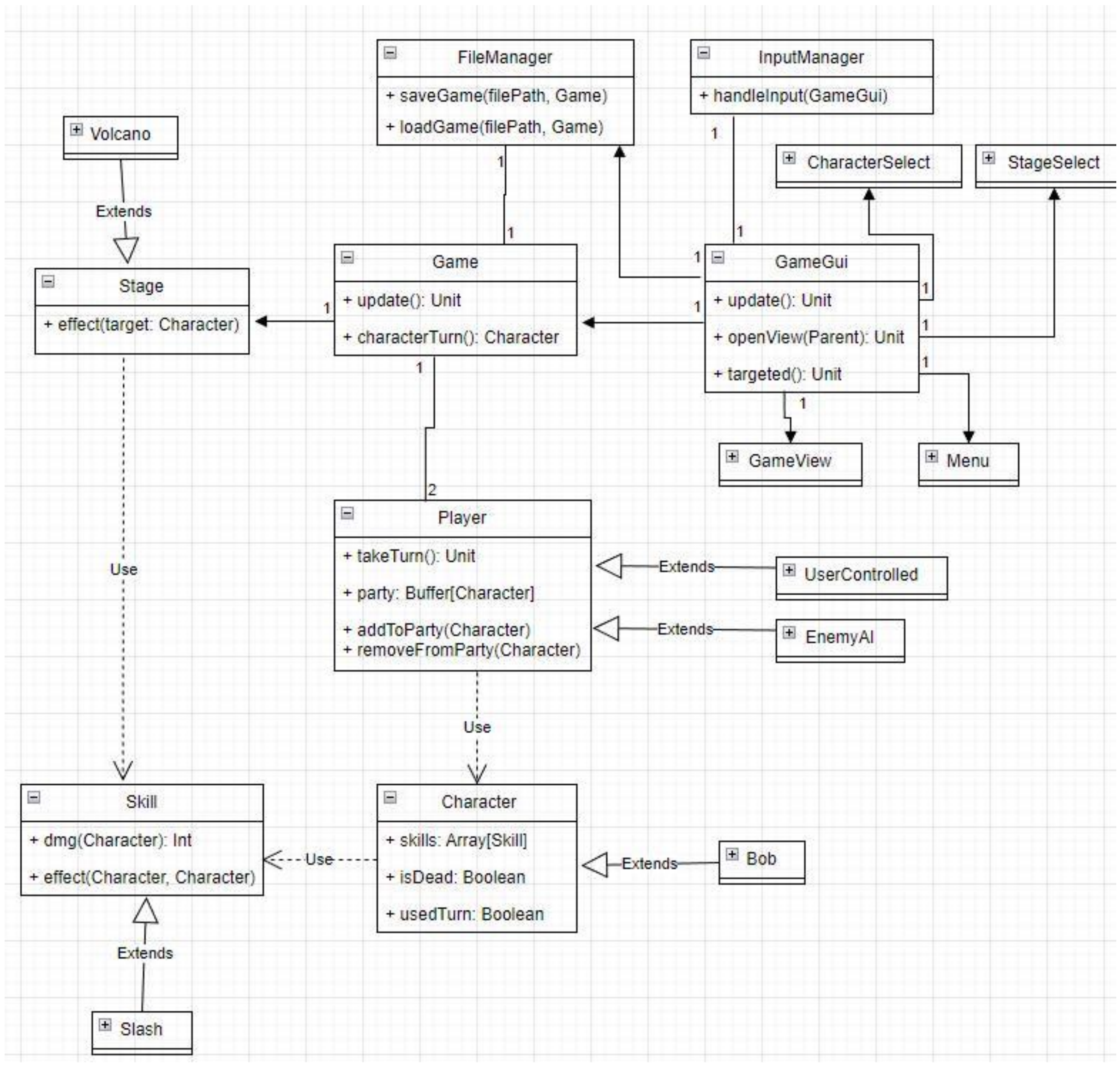
The game view consists of three parts. The stage takes the most space on screen and shows the characters on a background given by the stage. Here the game will also show textual descriptions about attacks and stage effects. On the bottom right there are four buttons for the characters' skills. The text on each button will show the name of the skill, the mana cost of the skill and whether the skill is an attack (#A) or a buff (#B). These will change when the character in control changes. On the bottom left there is textual info about the characters in battle.

Playing the game requires that the user presses the skill buttons to use skills on the characters. The user controls different characters between turns. Turn order depends on the current speed stat of the characters which is not visible to the user. The character in turn is highlighted by a black shadow effect surrounding the character. Skills can be targeted by clicking on the characters with a mouse. The targeted character is highlighted by a red shadow effect surrounding the character. Clicking on something that's not a viable target will randomly choose a character to be the target. The targeting effect is not shown at the start of the game and it will disappear when a targeted character is killed. If the user doesn't click on anything, the targeting effect will not appear, but the first enemy in the enemy's party is targeted when a skill is casted. Targeting must be done before using a skill, and the user can target any character with any skill, regardless if the effect is positive or negative. The user's character's turn will end after using a skill and a new turn is started. The enemy AI uses its characters' turns automatically and the result is shown by the aforementioned textual descriptions on the stage.

When the game is over, a "Game Over"-window will pop up and tell whether the user won or not. Closing the window will return the user to the game menu.

Program structure

I left most of the skill and character classes out from the diagram for clarity.



I’m fairly happy how my program structure turned out since I was very unsure when designing it in the project plan. I didn’t actually refine the structure that much even though my final program devised from the plan a bit. I decided to not have a GameObject class as the only proper objects in my program were the characters. I originally also had a helper object for using turns. That would have probably been useful since this was a turn based game after all. Some of the methods could have been moved there from the Game class for example.

Algorithms

The most important algorithm is definitely the one used to define how the AI picks a move and a target. In general the AI prioritizes finishing off low hp enemies, healing allies and using buffs or other set up moves on allies when at high health. When at low health the AI will choose any move that deals the most damage such as an explosion. The algorithm has a few different steps:

First set up the skills:

1. Get the character in turn
2. Check if the character has any skills that can be casted (have mana)
3. Sort the skills into attacks, nerfs, heals, or other (buff, nonheal)
4. Get the attack skill with most damage (no negative effects like self

damage) Then choose the skill:

5. if the skill with most damage kills any of the user’s characters, choose that
6. else if someone on AI’s party is low, choose a heal skill (if exists)
7. else if character in turn is healthy:
    a. if exists choose other skill
    b. else if exists choose nerf skill
    c. else choose skill with most damage
8. else choose skill with most damage (regardless of negative effects)

9. if chose attack then lowest hp enemy
10. else if chose heal then lowest hp ally
11. else if chose nerf then highest hp enemy
12. else chose other then highest hp ally
13. Use the chosen skill on the chosen target

I didn’t think of any alternate solutions, but I could’ve made the AI algorithm even better by checking the characters’ base stats to optimally buff magic users’ magic damage for example. Also, I could’ve given the AI the turn order so that it would know who’s next in turn and maybe update the algorithm to prioritize that character instead.

Data structures

I mainly used mutable buffers and arrays to store and process the data. Buffers were used to keep track of changing game data such as stage effects and the characters in player’s parties. Buffers were also used by the file manager to help read and load the data from a save file, and by the GUI to remove and add visual elements. I chose buffers as their size was not fixed and they were easy to manipulate from different parts of the program by appending, prepending or inserting data, which was needed since I wanted to, for example, add characters to a user’s party by clicking a button on the GUI. I did consider queues to keep track of different skills casted during turns but decided not to since that was not needed and to keep things simpler as I was more familiar with Buffers.

Arrays were used to store data that was set up at launch and wasn’t necessarily meant to be changed during gameplay. For example, characters’ skills were stored in an array. The array functioned as the character’s skill set which was fixed in size. I originally began using arrays as I was familiar with them even though for some reason I thought they were immutable.
After realizing that they were in fact mutable, I considered replacing some of them with immutable collections such as lists. In the end I continued to use arrays since they were suitable for my needs and allowed replacing data when needed. This was probably not ideal and I definitely could have replaced some of them. I mainly got the needed info about collections from the Scala standard library and from posts on Stack Overflow [1, 2].

Files and Internet access

The program only deals with XML files as I didn’t have the time for polished graphics. When saving the game, the program will write the game data to a XML file called “save1”. The file has been included in src/main/savefiles/save1.xml and it has proper data in it to also serve as an example of the format. Without altering the code, the program won’t support other save files and will overwrite save1 if a new save is made. I also included a copy of save1 as save2 in case save1 was overwritten. In this case save2 needs to be renamed to save1 in order to load the data.

Testing

The program was mostly tested using the graphical interface or command line. I did not build separate test classes and the final version of the project doesn't include unit tests. The program in general was simple enough that these somewhat worked in development but this also means most of the project isn't properly tested even though it passed my testing with the command line and the graphical interface. For example, the behavior of the AI should have been unit tested to be sure the algorithms work correctly.

The planned testing process was pretty rough as I wasn't very familiar with testing at the time. And since I didn't spend enough time on implementing tests at any point I ended up not properly testing important parts of the program. I initially planned to start implementing proper tests early in development but since I started working on the project so late I cut the time needed there to finish the project in time.

Known bugs and missing features

Even though I found a lot of bugs and surprisingly big defects after initially completing the project, I think I somehow managed to fix all of the ones I found. Currently I am not aware of any bugs or defects. However, I won't be too surprised if some bugs linked to the turn system are found as I reworked that multiple times and had a lot of problems with it when trying to fix other bugs.

There are multiple small features that I planned but didn't have time or resources to implement. Most notable are however proper character sprites and stage backgrounds, animations, and a skill system with more variety. A better skill system or a new status system would have allowed for more varied skills with effects that lasted for a certain amount of turns. One missing feature that I could have implemented after the last turn system update is a multi target skill but I didn't want to add new skills or change the old ones. For some of the visual features, I kept the code and structure that would have allowed implementing them in some way in the future.

3 best sides and 3 weaknesses

There are a few things in the project that I'm pretty satisfied with. The save file management, the character system and the targeting system. Even though the game only supports one save slot for now, I think the FileManager class turned out great. Learning XML was satisfying and when saving and loading a file, basically no game progress is lost. With more time the file format could have been expanded in a way that supports changing things like characters' skills and their values by editing the save files. I consider the character system good as it's fairly extensible compared to how much data each character holds. Same would apply to the skill system if it allowed more versatility. Lastly the targeting system was one of the first things I implemented and though it previously had some problems, it's a nice solution to avoid fixed or random targeting.

The three biggest weaknesses are probably the general game loop, the GameGui class, and the visual side of the program on the whole, even though it didn't have to be particularly polished. I designed the game loop in a way that is very dependent on the user doing things and because of that a lot of things needed to be updated or handled separately or in different ways when the AI opponent used his turn for example. A better way of implementing the game loop would have probably included separate threads and concurrent programming which I tried but was not successful with. The weaknesses in the GameGui class are partly linked to the game loop as it handles most of the users input and updates the game. The general weakness of the class however would probably be how crowded it is. I reckon by implementing some things differently one could relocate big parts of the class to other GUI classes, the input manager or some other class. The third big flaw is the visual polishing. If I had the time to even add proper backgrounds and character sprites the game would look much more like a proper game. Animations are probably the biggest missing visual feature and implementing them would have not only made the game look better but also make the gameplay a lot clearer. If I started using AnimationTimer in ScalaFX in the beginning, the whole game loop might have been different.

Deviations from the plan, realized process and schedule

I only made real progress on the project after five weeks or so simply because I had way more time during the last few weeks. Timewise the process deviated so much from the initial plan, that it’s hard to remember the dates things were implemented. The order however in which the project was finally implemented was fairly similar. I started by implementing the GUI and continued by making the game loop, the core systems like characters and skills, the enemy AI and finally the file manager. There was a lot of polishing and fixing broken systems between all of these, and especially the enemy AI and file manager were implemented very late in development. Working on the GUI between each step probably took the most time and was unexpected. I would still say I spent more than a hundred hours on this project in total which is close to the rough estimate of 120.

I would like to say I learned that you must start working on big projects early to avoid daily ten hour crunches, but that is probably not true especially since I was genuinely very busy at the beginning and starting the project late wasn’t that bad. It wasn’t optimal but it could’ve been worse.

Final evaluation

I already assessed most of the shortcomings and they were all caused by the lack of time. In case I was correct in that the program has no major bugs, the biggest flaws are the testing of the program, the game loop and the skill variety. Visual polishing and the use of collections are also some weak aspects. Except for the game loop, these were the things I deemed lowest priority if I wanted to finish the project in time. In hindsight, I could’ve cut some content or GUI features to have time for testing for example.

Proper testing could be added to the program easily with separate test classes. The main game loop obviously cannot be changed without reworking major parts of the Game and GameGui classes. It will also affect many other parts of the program. Skill variety would be a fairly easy fix with some updates to the Skill class or a new Condition class to allow any temporary effect. Adding some visual polishing would be easy but animations might need a bigger overhaul to the GameGui class.

If I started the project again from the beginning I would spend a lot more time on researching game design patterns and strategies before starting. For example I found a blog post on creating a turn based game and designing the game loop [3]. Info on vital design patterns was not that useful if I had already written a lot of code. Spending more time on exploring the ScalaFX documentation would have also helped as I found many useful methods that could’ve simplified some earlier problems [4].

References

[1] Scala standard library for Scala 3

https://www.scala-lang.org/api/3.2.1/,

[2] Stack Overflow

https://stackoverflow.com/

[3] Bob Nystrom, A Turn-Based Game Loop

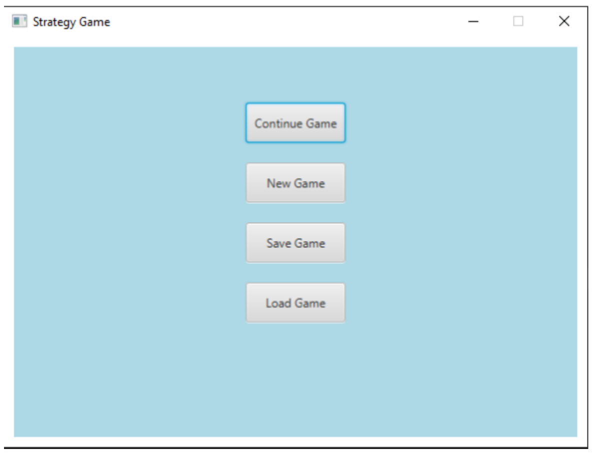
https://journal.stuffwithstuff.com/2014/07/15/a-turn-based-game-loop/

[4] ScalaFX documentation

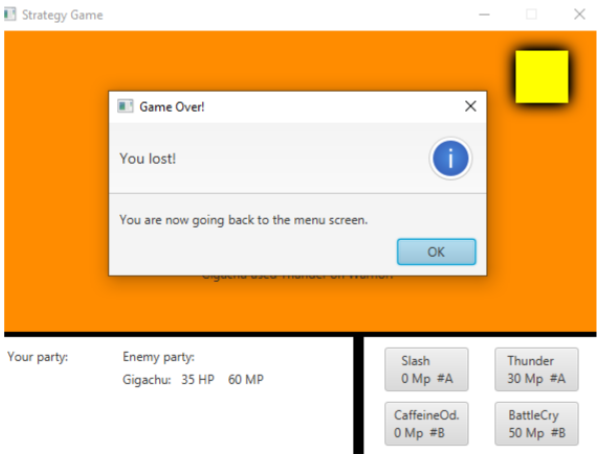
https://www.scalafx.org/

Appendixes

Menu:



Game over:



You lost!

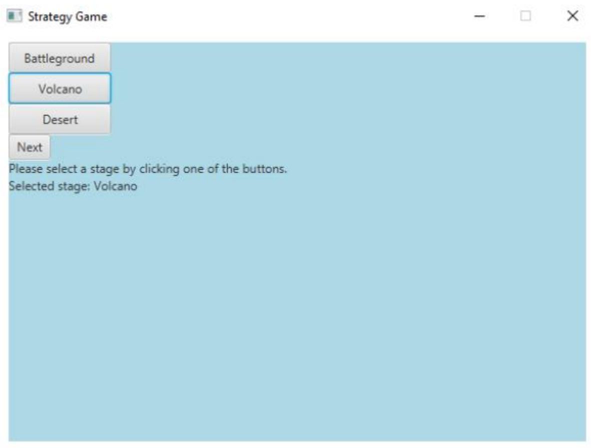
You are now going back to the menu screen.

Your party:

Enemy party:

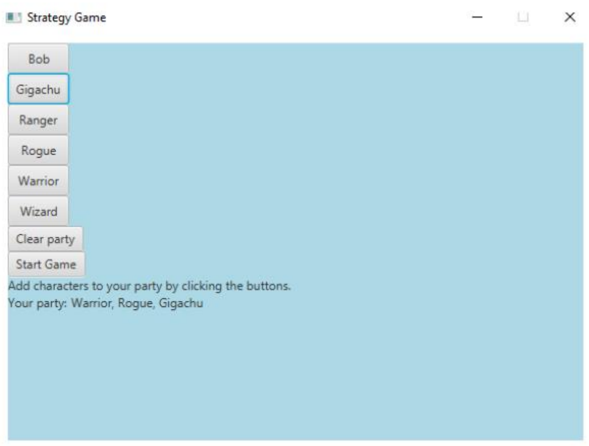
Gigachu: 35 HP 60 MP

Stage select:



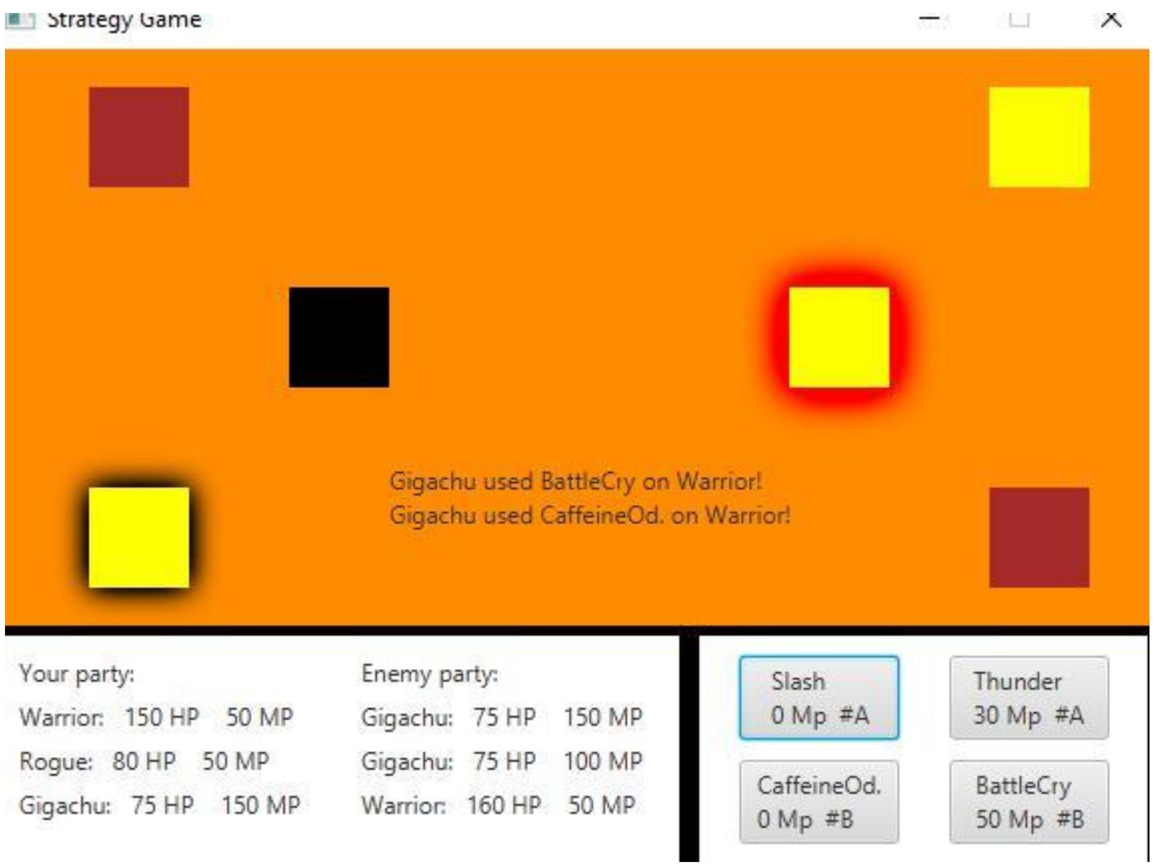
Please select a stage by clicking one of the buttons.
Selected stage: Volcano

Character select:



Add characters to your party by clicking the buttons.
Your party: Warrior, Rogue, Gigachu

Game view



Gigachu used BattleCry on Warrior!
Gigachu used CaffeineOd. on Warrior!

Your party:			Enemy party:		
Warrior:	150 HP	50 MP	Gigachu:	75 HP	150 MP
Rogue:	80 HP	50 MP	Gigachu:	75 HP	100 MP
Gigachu:	75 HP	150 MP	Warrior:	160 HP	50 MP