# Conquest

Daniel Wilenius
907925
Automaatio- ja informaatioteknologia
2020/2021
24.6.2021

## General Description

I made a turn based strategy game where the target is to conquer the enemy headquarter and protect your own. The playing field is a grid of hexagons (tiles).

You start off with placing your headquarter where you want on the game field and the AI is going to place it's headquarter in the upper left corner. After placing your headquarter you can now deploy troops which costs energy. You start off with 10 energy which you can spend to deploy troops or move your existing ones.
The game then progresses in turns,  ending the turn regains 2 energy points and makes the AI react to your actions as it sees fit.

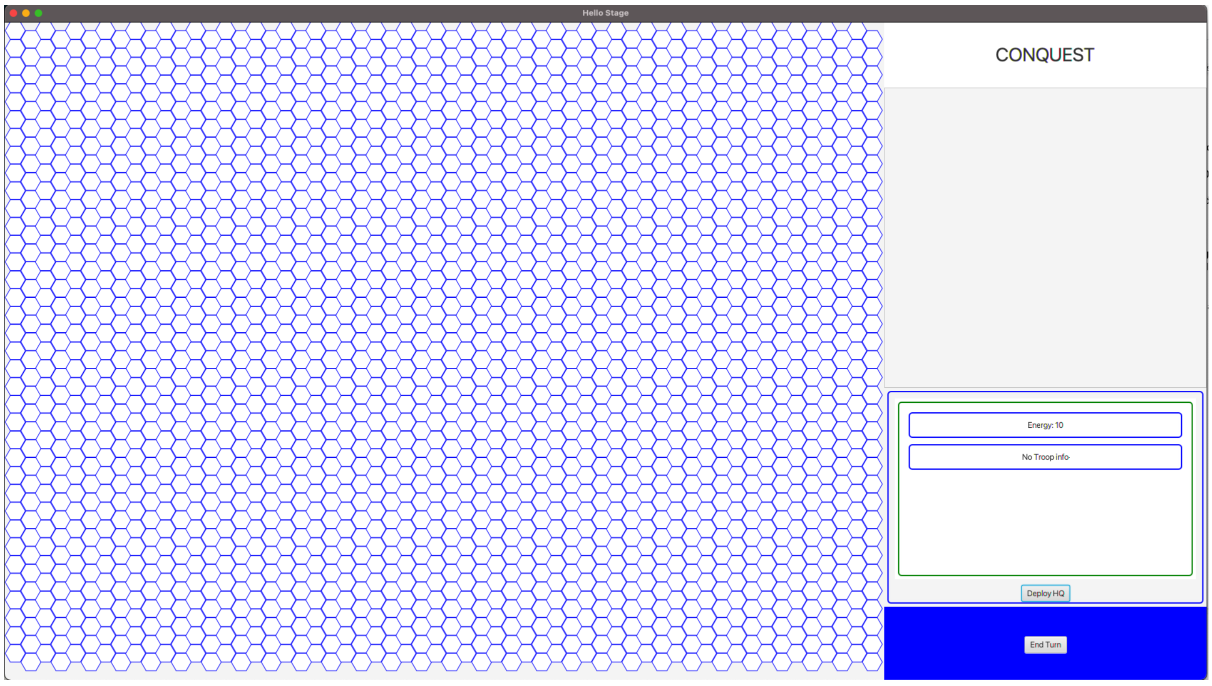
The troops can fight when they are in neighbouring tiles, the fighting mechanism has a random factor to determine the amount of damage each attack does, attacking the troop will make it retaliate by striking back but the troop that initiates the fight will deal damage first and thus have the advantage.

To conquer the enemy headquarter you need to get one of your troops next to it and conquering it works the same as attacking troops.

When starting the project I wanted to complete it at difficult level but in the end I think the project is completed at a medium-difficult level.

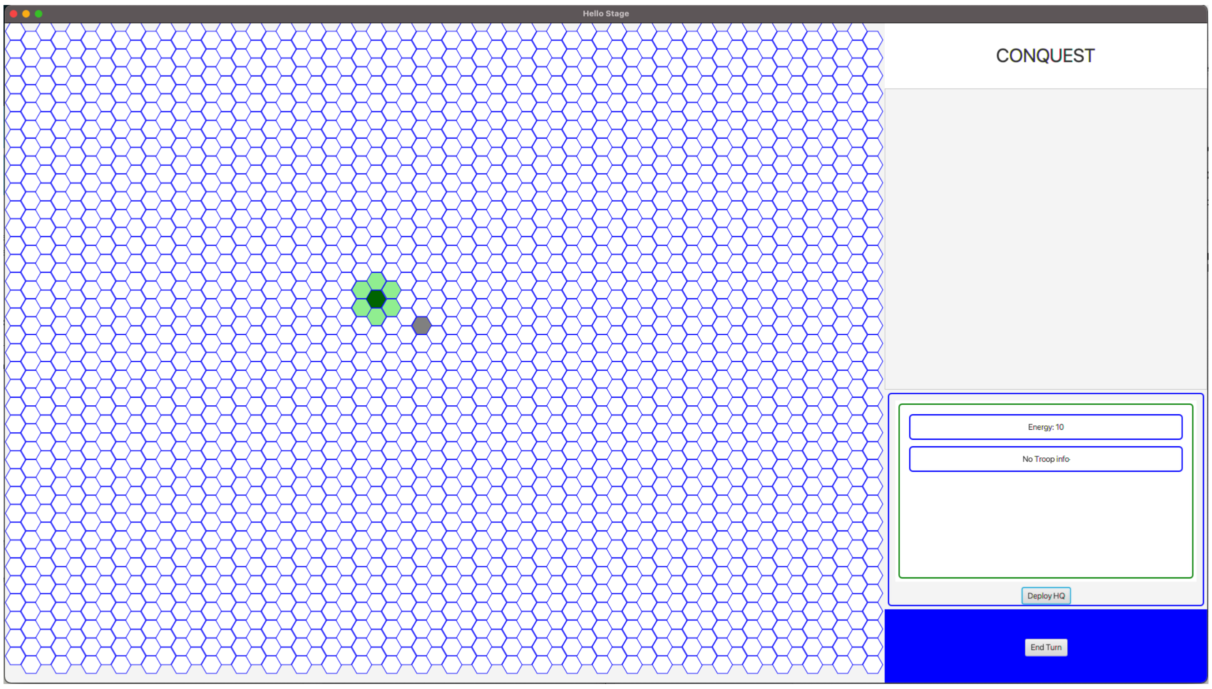
## User Guide

You start the game by running the Main Object. This window should pop up.

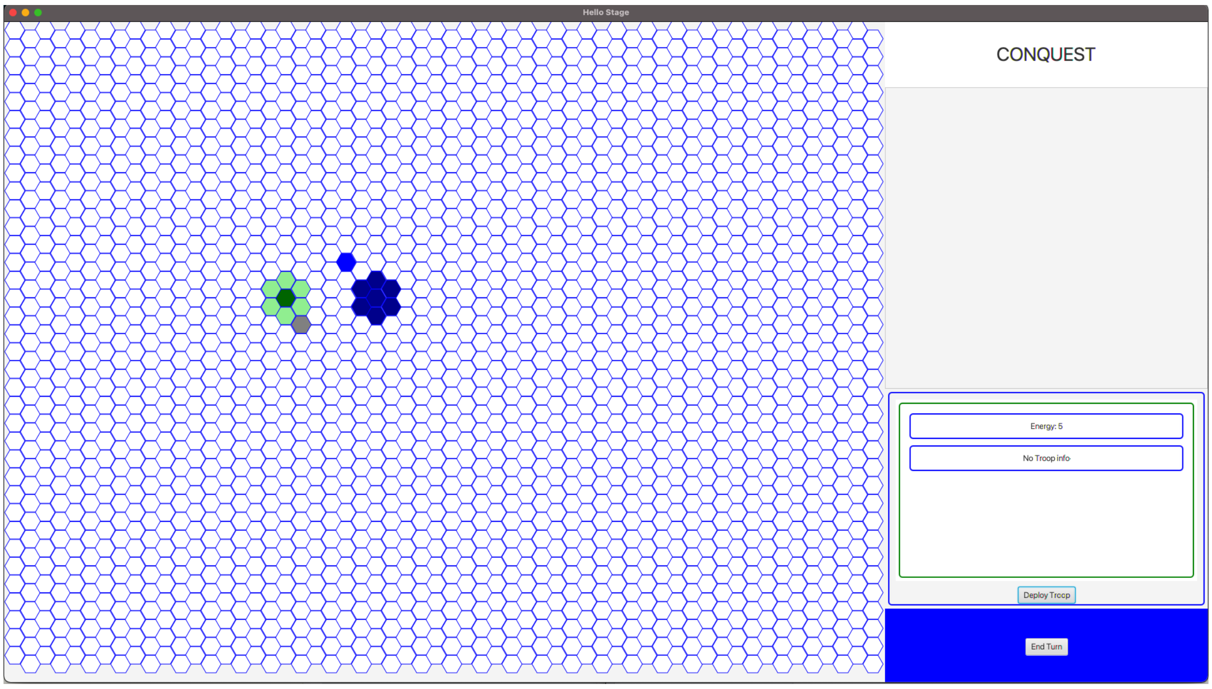


On the right you have the title of the game, under that the event log, under that the information section which will display information about your energy amount and any selected troops health. Lastly you have the two buttons Deploy HQ which is going to deploy your HQ ones you have selected a tile and ones you have deployed it it will become the deploy troop button. Under that is the end turn button which when pressed will make the AI react to your actions.

The GUI is entirely operate using your mouse clicks.

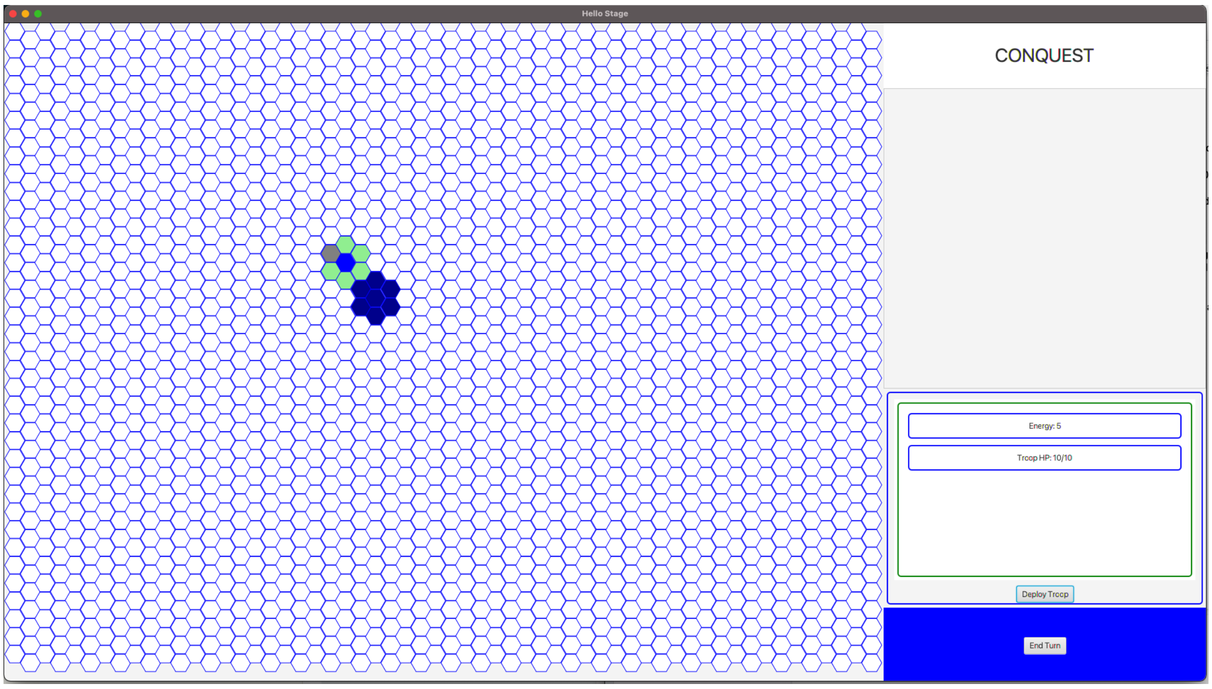


Here I have selected a tile and I'm hovering over a different one, selecting a tile will turn it dark green and the neighboring tiles light green. Hovering over a tile turns it grey.



Here I have placed the HQ using the Deploy HQ button on the tile I had selected above. After that I've selected another tile close to my HQ that I have enough energy to place troop at and placed the troop, this is the lighter blue single tile. I have also clicked on another tile in this picture which you can see is selected based on the dark green color.

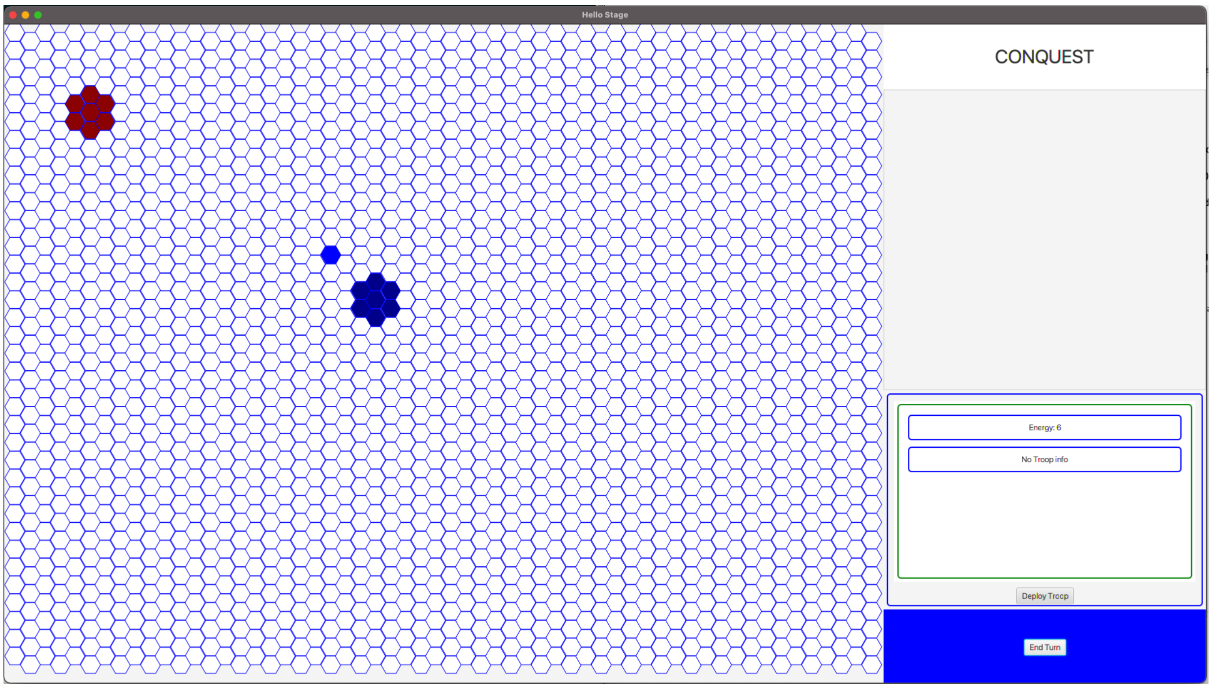
The energy cost of placing a troop is the amount of steps from the center of your HQ + 2.



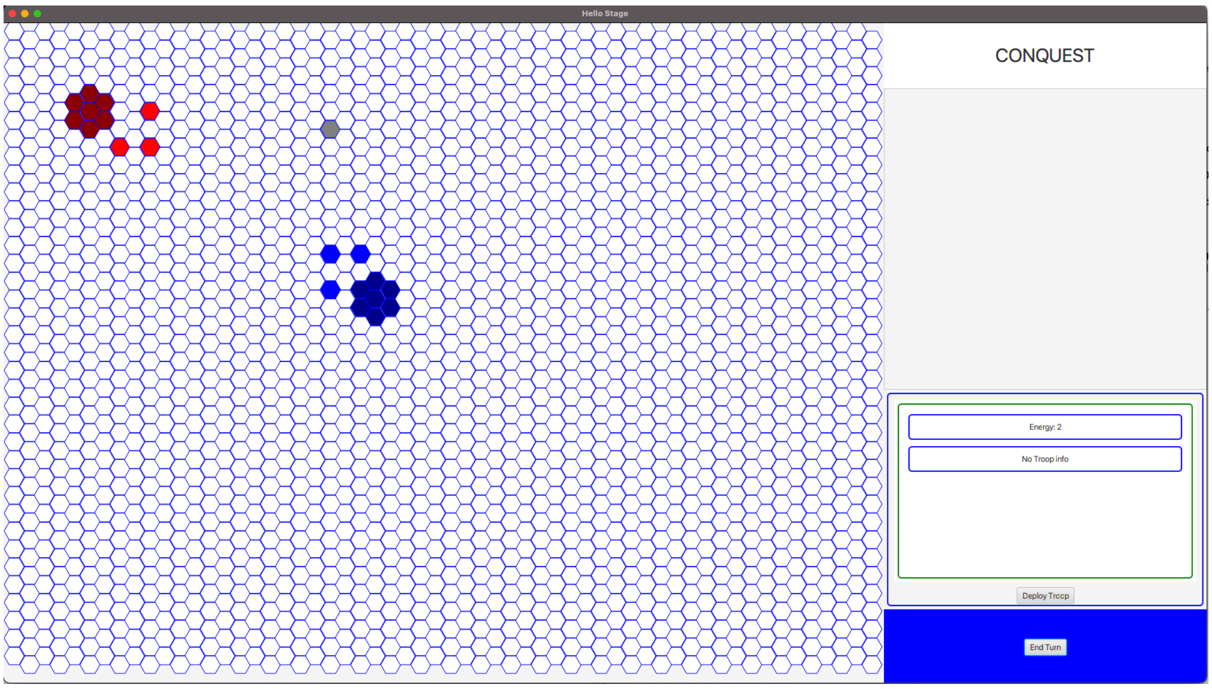
Energy: 5

Troop HP: 10/10

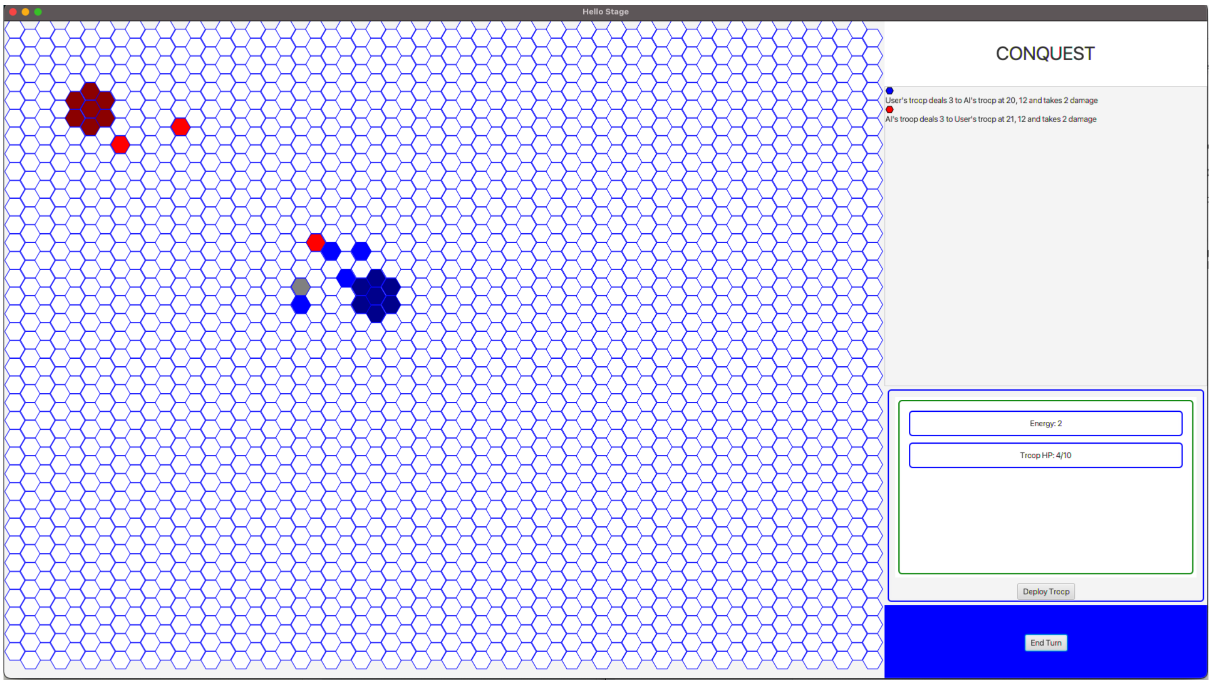
Here I have selected my troop, the light green tiles around it are tiles it is able to move to and the grey one is the one of those that I'm currently hovering over. Clicking that tile will move the troop to it which costs 1 energy. You can also see that the troop information box now displays the current HP of the troop.



After moving the troop I've ended the turn, here we can see that the AI has acted and placed its HQ marked by the dark red tiles.



Here I've placed a few more troops on the field and so has the AI now it's time to try to move towards the AIs headquarter and conquer it without leaving your own unguarded.



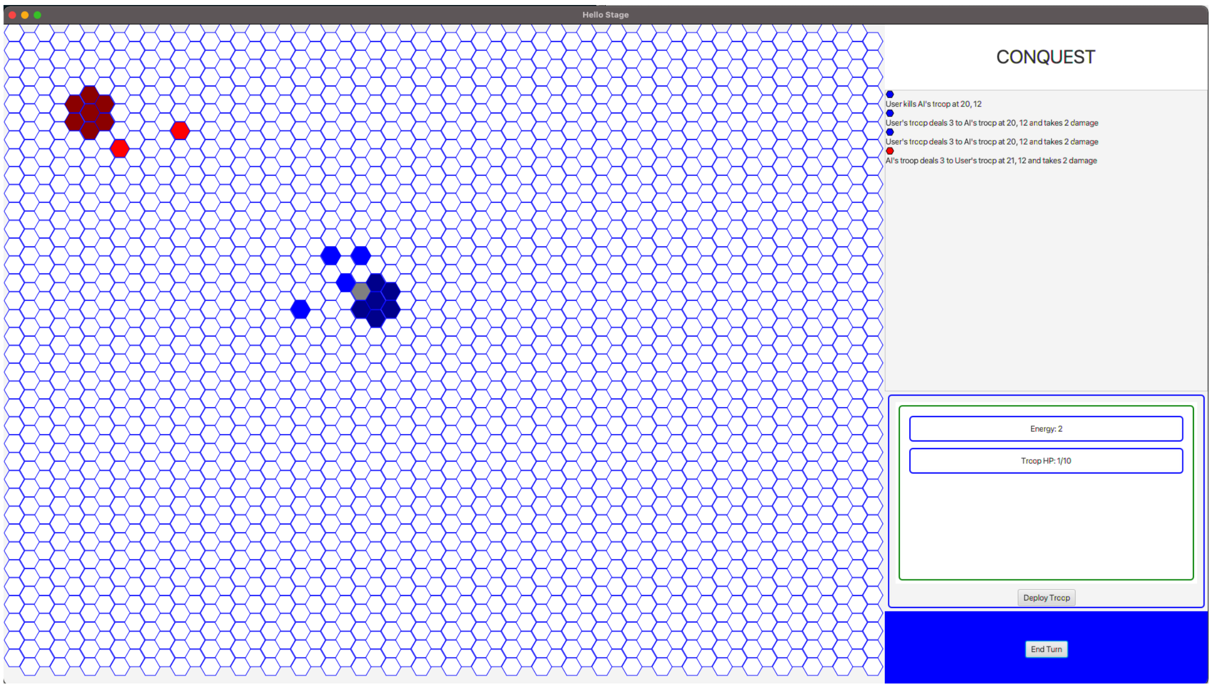
CONQUEST

● User's troop deals 3 to AI's troop at 20, 12 and takes 2 damage
● AI's troop deals 3 to User's troop at 21, 12 and takes 2 damage

Energy: 2

Troop HP: 4/10

Here the AI has started to attack my troops, you can see in the event log under the title what has happened. Since the red troop and blue troop are next to each other they can fight. To do this select the troop you want to attack with and click on the enemy troop, just like trying to move to that tile.



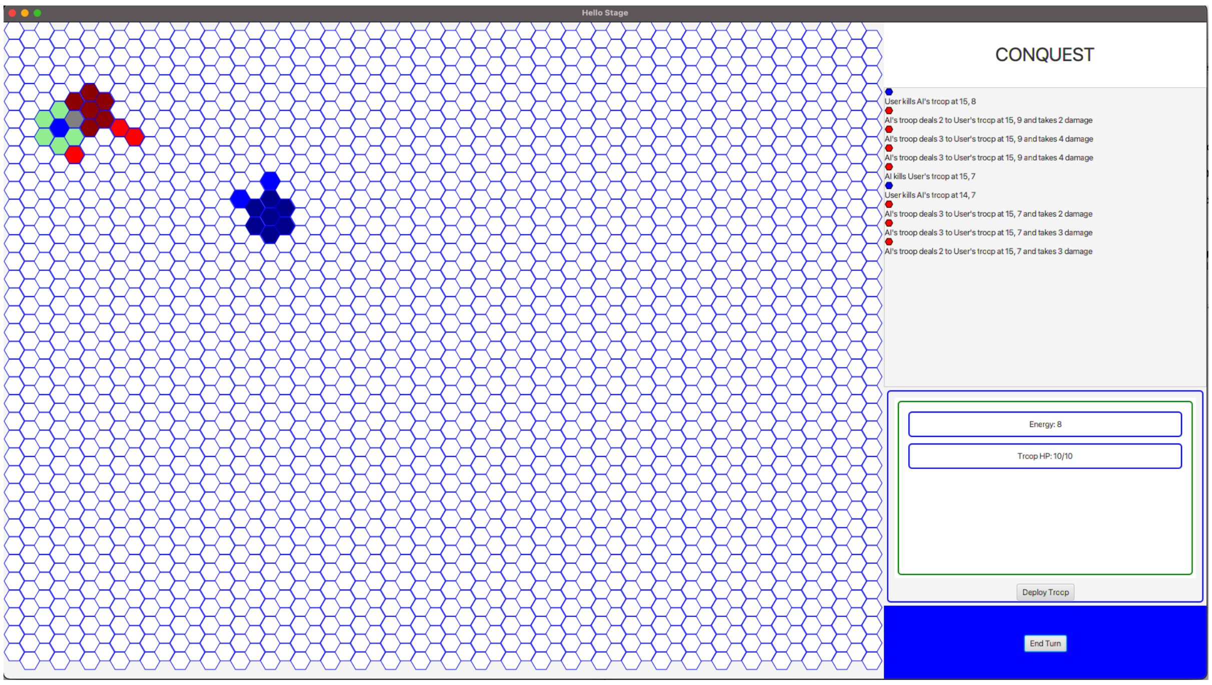
CONQUEST

● User kills AI's troop at 20, 12
● User's troop deals 3 to AI's troop at 20, 12 and takes 2 damage
● User's troop deals 3 to AI's troop at 20, 12 and takes 2 damage
● AI's troop deals 3 to User's troop at 21, 12 and takes 2 damage

Energy: 2

Troop HP: 1/10

Here I have killed the enemy troop.



```
User kills AI's troop at 15, 8
AI's troop deals 2 to User's troop at 15, 9 and takes 2 damage
AI's troop deals 3 to User's troop at 15, 9 and takes 4 damage
AI's troop deals 3 to User's troop at 15, 9 and takes 4 damage
AI kills User's troop at 15, 7
User kills AI's troop at 14, 7
AI's troop deals 3 to User's troop at 15, 7 and takes 2 damage
AI's troop deals 3 to User's troop at 15, 7 and takes 3 damage
AI's troop deals 2 to User's troop at 15, 7 and takes 3 damage
```

I've managed to sneak in behind the AIs troops and selected my troop that is threatening the AIs HQ clicking the highlighted grey tile will win me the game by conquering the HQ.

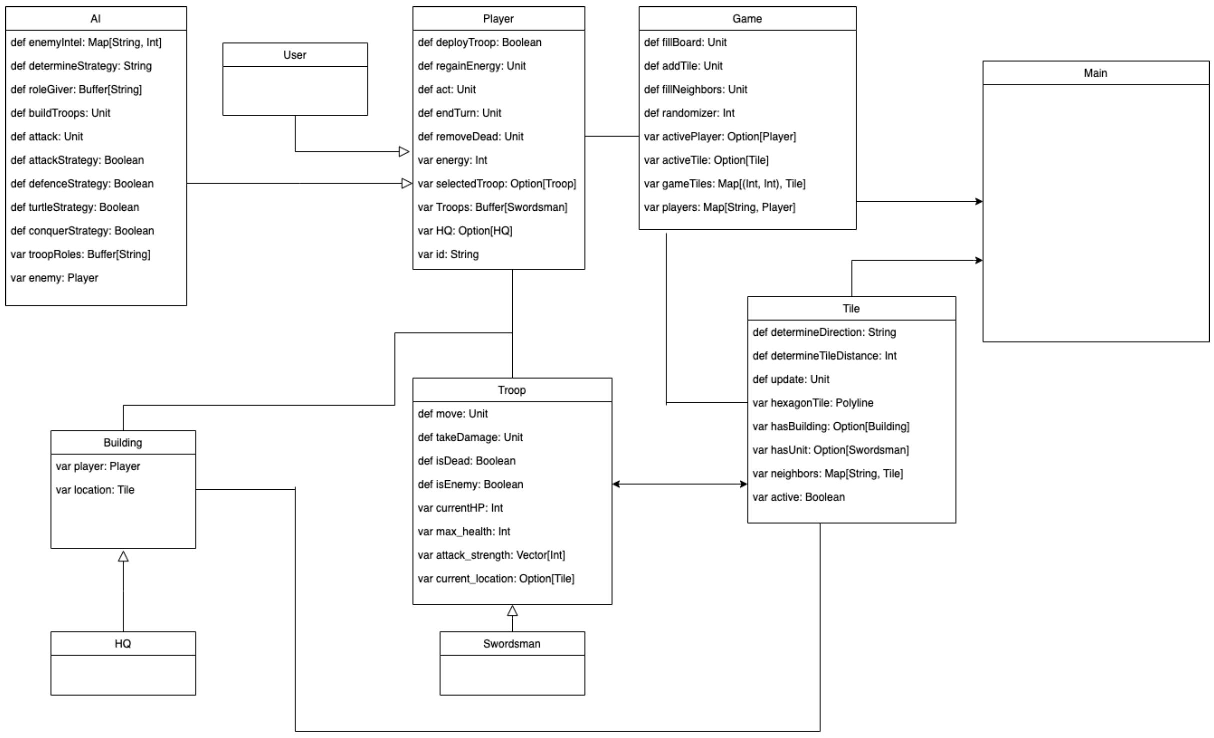
**Recommended strategy**
From what I've found the best strategy is to try to sneak one troop in behind the AI and keeping your other troops close enough to the AIs troops so that they're distracted by them.

If you want to play a shorter game I recommend placing your HQ close to the upper left corner, which is where the AI places its HQ.

The difficulty of the AI can be adjusted by changing the amount of energy it regains each turn. This can be found under the regainEnergy method in the AI class.

**Programme Structure**

The programme has 5 classes and 1 object
- Game
- Player
- Tile
- Troop
- Building
- Main [Object]


AI
def enemyIntel: Map[String, Int]
def determineStrategy: String
def roleGiver: Buffer[String]
def buildTroops: Unit
def attack: Unit
def attackStrategy: Boolean
def defenceStrategy: Boolean
def turtleStrategy: Boolean
def conquerStrategy: Boolean
var troopRoles: Buffer[String]
var enemy: Player
User
Player
def deployTroop: Boolean
def regainEnergy: Unit
def act: Unit
def endTurn: Unit
def removeDead: Unit
var energy: Int
var selectedTroop: Option[Troop]
var Troops: Buffer[Swordsman]
var HQ: Option[HQ]
var id: String
Game
def fillBoard: Unit
def addTile: Unit
def fillNeighbors: Unit
def randomizer: Int
var activePlayer: Option[Player]
var activeTile: Option[Tile]
var gameTiles: Map[(Int, Int), Tile]
var players: Map[String, Player]
Main
Tile
def determineDirection: String
def determineTileDistance: Int
def update: Unit
var hexagonTile: Polyline
var hasBuilding: Option[Building]
var hasUnit: Option[Swordsman]
var neighbors: Map[String, Tile]
var active: Boolean
Troop
def move: Unit
def takeDamage: Unit
def isDead: Boolean
def isEnemy: Boolean
var currentHP: Int
var max_health: Int
var attack_strength: Vector[Int]
var current_location: Option[Tile]
Building
var player: Player
var location: Tile
HQ
Swordsman

Here’s a UML diagram of the class structure with most of the functions and variables.

**Main Object**

The Main object is responsible for creating the basic structure of the GUI and updating the GUI components during the gameplay. It will also when first initiated create a new Game and add the players to the game. Creating a new Game will generate the tiles to place on the game field and output them in to the variable tileMap inside the Main object which is a Map that stores the tiles with a key consisting of the coordinates.  All the tiles will then be added to the scene along with the rest of the structure for the GUI.

The object has a few methods, addLabel, updateGUI, updateActionBox, eventLabel, energyLabel, troopInfoLabel and endGame.

addLabel is just a simple helper function for the other functions that takes a string of text and returns a label with that text.

updateGUI is a method that refreshes the GUI by updating all of its components. It will also check for if the headquarter has been placed which if true will replace the Deploy Headquarter button with the Deploy Troop button.

updateActionBox is a helper function for the method updateGUI method that will update the ActionBox content.

eventLabel is a method that when called will add a Label to the eventBox, this method is called by other parts of the programme to add events to the event log.

energyLabel is a method that will return a Label that contains information about the current energy amount when called.

troopInfoLabel is a method that will return a Label that contains information about the current health of a selected troop, if no troop is selected it will return a Label “No troop info”

Finally the endGame method will change the scene to either the winScene or the loseScene depending on which player it gets as an input.

**Game class**

The Game class is mainly responsible for initiating the game by creating the playing field and keeping track of the current state of the game.
A few important variables are the activeTile, activePlayer and gameTiles which many of the other classes use but they are stored in the game class.

The class also has 4 methods, addTile, fillBoard, updateGUI and randomizer.

addTile is a helper function for the fillBoard method which takes an x and y component as well as a game as input. It then creates a new tile using these inputs and adds it to the gameTiles Map.

fillBoard is a method that takes as input from the settings of the Main Object an amount of Tiles to create in the x direction and y direction and then uses a while loop and the addTile function to add all the tiles.

updateGUI is a method that loops through all the game tiles and resets the state of these. It is called when a turn is ended.

Randomizer is a method that takes an integer as input to indicate what the maximum integer is that it should return. And returns a random integer between 0 and the input. This method is used to make the damage in the fighting mechanism luck based.

**Player class**

The Player class is an abstract class that serves as a base for the User class as well as the AI class. Shared methods for there classes are removeDead, deployTroop, regainEnergy and endTurn. Each players troops, headquarter and energy is also stored here as well as selectedTroop.

removeDead is a method that removes all dead troops from a players troop Buffer.

deployTroop is a method that checks whether a the player has enough energy to deploy a troop at a certain Tile and removes the required amount of energy if successful.

regainEnergy is a method that adds energy after each turn to the player.

endTurn is a method that ends the turn and calls the other relevant functions for this event.

**AI class**

The AI class has the same general structure as the User class with some added methods, these are enemyIntel, determineStrategy, roleGiver, attack, attackStrategy, defenceStrategy, turtleStartegy, conquerStrategy, buildTroops,  and act.

enemyIntel is a method that creates a Map with different information about the current situation on the playing field.

determineStrategy is a method that uses the enemyIntel to determine which strategy the AI should use and returns the strategy it thinks is best suited for the current situation as a String.

roleGiver is a method that gives each troop the AI deploys a general role. This is simply used to make the troops establish a basic formation.

Attack is a method that takes as input one of the AIs troops and one of the users troops and fights it until death. This is a helper function for the different strategy methods.

attackStrategy is a method that executes the AIs actions if it has determined that it should use the attack strategy. It will loop through the AIs and Users troops and assign targets for each of its troops which it will then pass on to the attack function.

defenceStrategy is a method that executes the AIs actions if it has determined that it should use the defence strategy.

turtleStrategy is a method that executes the AIs actions if it has determined that it should use the turtle strategy.

conquerStrategy is a method that executes the AIs actions if it has determined that it should use the conquer strategy. This strategy will identify which of the AIs troops is closest to the enemy headquarter and spend all its energy to move towards it to conquer it.

buildTroops is a method that is called from each of the strategy methods in case the AI thinks that it needs to build troops before acting on the strategy.

Act is a method that is run each time it’s the AIs turn, on the first round it will place its headquarter and after that it will determineStrategy each round and act accordingly.

**Tile class**

The Tile class takes care of the playing field and is responsible for creating the graphical tile component, updating it on changes and the overall geometry of the playing field. It’s most important members is the variable hexagonTile and the methods determineDirection, determineTileDistance, distanceToTile, fillNeighbors and update.

hexagonTile is a variable that hosts a Polyline that is created based on the x and y inputs when a new Tile is created. The mouse input on the playing field is defined to this variable with .onMouseEntered, .onMouseClicked and .onMouseExited.

determineDirection is a method that takes another tile as an input and determines which direction S, SW, SE, N, NW, NE best represents the direction from itself.

determineTileDistance is a method that counts steps between tiles. It takes another tile as an input and uses the determineDirection method.

distanceToTile is a method that calculates the distance between tiles centre points in a normal 2-axis coordinate system. It is used for determining whether a tile is a neighboring tile.

fillNeighbors fills the Map neighbors with the neighboring tiles along with their direction as key.

Update is a method that refreshes all tiles and makes their graphical up to date.

**Troop class**

The troop class takes care of the the troops and their fighting mechanisms. Its most important members are the methods takeDamage and move.

takeDamage is the method that takes care of the fighting mechanism. It will also add a Label to the event log which gives information about how it played out.

Move is a method that determines whether a troop can move to another tile and what should happen when it does so based on what that tile contains.

**Building class**

The building class is simply the class that creates the headquarter and has no real other uses except for when being created adds the building to the tile it is created on.

## Algorithms

The programme uses a few different methods together to create the path finding algorithm for the programme. These methods are located in the Tile class and work together as follows. When the playing field has been generated each tile on the field will first recognize its neighbor tiles using the distanceToTile function it will then Map the neighbor tiles using the determineDirection function so that the direction in string form becomes the key and the neighbor tile the value. Now the setup for the determineTileDistance function which is the function that finds the shortest path is done.
The determineTileDistance function takes as an input another tile that it calculates the distance to, it then sets that tile to be the endTile, itself to be the stepTile and a distanceCounter to 0. It will then using a while loop simulate the path by determining which direction best describes where the endTile is, setting the stepTile to be the neighboring tile of the old stepTile in the direction of the endTile, it then adds 1 to the distanceCounter and finished when the stepTile is the same as the endTile.

The AI algorithm is a sum of a lot of different methods specific to the AI class under Player that all come together when the act method is called each time it’s the AIs turn. When act is called it calls the determineStrategy method which in turn calls the enemyIntel method. The enemyIntel method gathers information about the user players and its own troop situation on the playing field. It gathers information such as whether the enemy or itself has a troop closer to its own headquarter, whether it has a troop closer to the enemy headquarter than the enemy, whether it considers the enemy to be in a defensive or offensive position based on where its troops are located and the overall strength of the users as well as its own troops on the field measured in total currentHP, it will then Map this information and return it.

determineStrategy then makes a decision based on this intel what it thinks is the best strategy to use, for example if it has higher overall strength on the field it will choose Attack as long as it’s own headquarter isn’t in immediate danger and it doesn’t have an opportunity to win the game. If it has a troop that is closer to the enemy headquarter than any of the enemies troops it will choose conquer.

The strategy that the determineStrategy method returns will then be matched using a match case structure to trigger the actual method for the strategy.

Unfortunately I ran out of time with this project so the only really different strategies are attack and conquer. Defend and turtle are both the same as the attack strategy.

The attackStrategy will first determine if it has enough troops to attack, if not it will build troops using the method buildTroops. If it has it will assign targets to each of its troops based on which of the enemies troops are closest to that particular troop and spend all its energy hunting them down and if it reaches the target it will fight it to the death.

The conquer strategy will identify which of the AIs troops is closest to the enemy headquarter and spend all its energy moving that troop towards the headquarter to conquer it.

**Data Structures**

I'm using mostly maps and mutable buffers in this program to store information. I've used Maps when I need a way to get a certain member and buffers where this isn't as important as I find them quite flexible and easy to work with. I like to use maps because I like the fact that I can determine what the logic behind the keys should be so they feel natural to use.

**Files**

The program doesn't use any external files or create any new ones since I didn't have time to create a settings file, game state saving feature or more fancy graphical components.

**Testing**

I feel like I could have done a better job with creating tests for the programme and have mostly used print statements to see where things go wrong during the course of the development. I've also tried to make the code fairly robust so that there aren't very many risk factors such as infinite while loops. I've used quite a lot of while loops and prevented infinite loops by also setting a failsafe variable to the loops.

**The programmes flaws and errors**

Most of the flaws in the programme are due to time constraints such as there not being more different strategies that the AI uses, another big flaw is that the programmes GUI elements will not resize if you try to change the size of the window.

An error that is unlikely but can occur is if you manage to get a troop to the tile that the AI wants to use to deploy troops this will lead to the AI trying to move your troop which it obviously can't do and will lead to an error.

A flaw with the GUI is that when a troop has just been deployed you need to select another tile after and then click on the troop again to select it.

**Best and worst parts**

I think the best parts of the programme is the hexagon playing field, and the algorithms.
The hexagon playing field because it is quite a bit more challenging to create than just a normal square grid but provides benefits such as moving in all directions being at a distance of 1 compared to a square field where moving diagonally would either require 2 steps or a geometrical distance of sqrt(2).

The AI and pathfinding algorithm because I think they work quite well except for the missing different strategies. The AI however is fairly challenging to play against and can surprise you.

Worst parts are I think the fact that you can't resize, there not being a save feature, lack of more AI strategies and missing game features that would make the game more fun.
These are all down to running out of time. The resizing would perhaps have been good to get on right from the start and not try to start to fix at the end.

**Exceptions to the plan and time table**

In the end the game lacked a lot of the features I planned on implementing which would make the game more fun such as obstacle tiles, different kind of troops and different kind of buildings.

The project was also delayed due to a lot of work on other courses so the original time table wasn't really followed very well.

**Summary**

All in all I'm fairly happy with the end result all though it has significantly less game features than I was planning on. I could also have done a better job planning the classes and their relations, in this project I've mostly linked them all together so that I had access to all parts of the programme from any part, this is nice when you're programming but results in a bit messier result where certain methods can be in places where they don't belong as much.

If I would start the project again I would spend more time on the planning so that I could follow a more defined path when working on the project.

**Sources**

I’ve mostly used the scalafx documentation
https://www.javadoc.io/doc/org.scalafx/scalafx_2.11/8.0.192-R14/scalafx/package.html
Various stackoverflow questions and answers as well as various youtube videos as sources.