**TFTBot Project:**

# Analysis.

The Problem.

Team-Fight Tactics, often abbreviated as T F T, is an auto-chess, strategy game, created by Riot Games and running on the engine same engine that powers the popular multiplayer online battle arena League of Legend. TFT is an incredibly complex, strategic game that requires a vast amount of game knowledge and experience. Getting skilled at TFT is a large time commitment and overall a slow process. One large hinderance is the lack a training or test mode resulting in the fact that the only way to improve at TFT is to play TFT. There is not a “test mode” for instance that allows you to create a team composition and battle it against a different one and so if you wanted to get experience with a certain team composition or item, you would have to hope you get the opportunity to play the team composition/ item, which is no guarantee, meaning you could waste a large amount of time unnecessarily trying to learn about a rare interaction. If Riot Games were to implement a custom or a test game mode, lots of time could be saved, leading to shortened learning time and quicker improvement at the video game. It would reduce the time commitment that new players need to improve at the game, allowing them to enjoy the game more than if they were left in the dark, as well as allowing experienced users to test out ideas without spending lots of time trying to get the opportunity to do so.

The Stakeholders and Solution.

Suket Arya is an avid TFT player. Within TFT, what Suket particularly enjoys is brainstorming and creating new ideas for team compositions and strategies to test out against other, more popular compositions. However, with his enrolment into university, he no longer has the time to spend multiple games attempting to get a favourable position to be able to test out a team composition, and then the numerous games after that to fine tune the strategy into something completely viable.

My solution is to create a program that can simulate a board/ battle between two team compositions accurately. Following on from this, through a UI, I aim to give the user the ability to place down units and teams, giving the opportunity to users to test out ideas and learn certain matchups, without having to try to recreate the situation in an actual game, which could be a very timely investment. Furthermore, it would give more effective, instant feedback and allow slight tweaking of a board so users could see what they could have done better, rerunning a similar battle multiple times to show them what they should have changed to win that battle in the future, allowing for faster learning and improvement for new players.

This tool aims to be useful for all TFT players, both those looking to scratch the itch of team building and strategic thinking, as well as those looking to improve quicker than their counterparts. Moreover, this tool will be useful at all levels of play, even for pro players who will enjoy the extra freedom it provides.

Existing Solutions:

There are no existing programs that allow for the simulation of TFT battles, however, as this aims to be an educational tool, I’ll compare this solution to what other educational tools out there for TFT.

***Community Guides:***

As with any community, there is lots of content surrounding educating and improving at the video game. Whilst these can be very useful and helpful, especially for a beginner with zero previous experience with the game, the quality of content within the guides can vary widely, from incredibly useful to downright misinformation and harmful to anyone trying to learn how to play the game. When trying to improve through community guides any user has to be careful to ensure the guide is positively reviewed or guarantee themselves that the content within the guide is accurate (which is near impossible for newer players).

Moreover, there is an upper bound for where community guides can continue to help you. Experienced and high level players will learn near to nothing from guides aimed at newer players and there becomes a level where community guides no longer cover a high enough skill level.

Finally, as the game progresses, any guide has to update itself or find itself becoming irrelevant as the new content invalidates the information in the older guide, users have to be careful to check that any content was posted fairly recently or has been updated.

Positives:

* Can be a great starting point and can help beginners avoid simple pitfalls they often fall into.
* Can help pass on guidance from more experienced players to newer ones.

Negatives:

* Can be outdated or low quality
  + Guides must be constantly updated or else they risk giving outdated, invalid or entirely useless advice about the game.
  + The user also must be able to discern the quality of the guide, or there have to be good mechanisms in place for reviewing guides, otherwise new players can be given bad advice without knowing better, severely hindering their ability to improve.
* May not cover an issue or skill that a player needs to learn.
* Only useful for new players or semi advanced players

Examples:

* Mobalytics:
  + Graphical user interface, text, application, email

    Description automatically generated
  + “Mobalytics” offers a starting tutorial/ guide for beginners, helpful so for your first few games you aren’t completely thrown in the deep end with no help, but beyond that provides zero guidance for more experienced players.
* Mobafire Community-Made Guide:
  + Text

    Description automatically generated
  + More in-depth community made guide available on “Mobafire”, offers some good tips, but is now very outdated, at almost two years old. However, it is still one of the top options that comes up if you search for a TFT guide, so newer players could fall victim thinking it was still solid, relevant advice.

***Meta-advice Programs:***

There are programs out there that run alongside TFT and offer guidance on currently strong teams and strategies in TFT. They provide overlays that show up in game and give you a list of compositions to choose from and then information on each composition. They can also contain other pieces of information such as what component items combine into and what these items do. These programs are useful for new players or more experienced ones who do not have time to keep up with the meta (stands for most effective tactics available, a community consensus on what teams are strong at the moment). More casual players generally find these types of programs very useful.

However, they only really provide information available from a google search in a more convenient and accessible place, one that isn’t constantly changing either, so more experienced players do not really have a use for them. They do not really teach the user anything either and sometimes mindlessly following what the program tells the user to do without thinking why can harm or halt their improvement.

Positives:

* Provides advice directly in the game
* Helps player stay up to date with the latest and most effective strategies known to the community at the time.

Negatives:

* Not adaptive
  + Provides no adaptive or specific advice to certain situations, simple informs you what strategies are popular at the time, no matter how viable that strategy is for you in that specific scenario.
* Only useful for new players or players completely out of touch with the meta (most effective tactics available).
* Can slow improvement.

Examples:

* Mobalytics App:
  + 
  + Mobalytics has an app you can install that offers in game advice as shown above. It allows you to select a team composition and gives you advice on where to place units, what items to go and what characters to get.
  + However, liked stated as you progress further, it gets less and less useful.
  + Can actively harm players if it is not the right opportunity to go a certain team composition you selected, choosing to ignore it meaning the program offers nothing in value, whereas forcing the composition when you shouldn’t (either because the user does not know better or otherwise) will harm your chances to win the game.

***Coaching:***

Coaching is always an effective option when looking to improve at the game. Regardless of skill level, as long as you don’t find yourself at the very very top of the rankings, there will always be someone to coach you.

Coaching is undoubtedly the best method to improve at TFT, offering direct and insightful advice which can vastly speed up the rate at which you improve. They can highlight what you are doing wrong and give you tips and knowledge which would have taken you hundreds of hours to learn. It cannot be overstated how useful coaches can be when trying to improve. This, however, is the reason why they can be so inaccessible. Unless you know someone personally good enough and willing to coach you, who also has a talent for teaching people, you will have to pay someone for the privilege, which can be incredibly expensive. A bad coach will also not provide much use, so users have to be careful to get a highly regarded coach to avoid wasting their money and time.

Positives:

* Provides useful, accurate advice.
* Specific to your situation, can tell you the optimum move in any scenario.
* Can provide helpful tips and knowledge.
* Helpful at every skill level

Negatives:

* Not accessible to everyone.
* Requires a large time commitment from the better player.
* Can buy paid coaching which can be very expensive.

Examples:

* A screenshot of a computer

  Description automatically generated with medium confidence
* Live coaching is an effective way to rank up from the very top of top players, and is effective at higher ranks when other methods offer less and less benefit as you go up in rank.
* The downside is that it is very expensive, having to spend lots of money for even a relatively short period of time coaching unless the user personally knows someone willing to coach them.

## Need for Computational Methods.

The problem at hand is uniquely suited to being solved by a computational methodology. With my project, we can use decomposition, breaking a large problem down into many smaller parts, to split up the project in numerous ways. Units, status effects and items within the game can be represented with classes and a board class could run iteratively until the battle is over. Moreover, my program is suited for abstraction. The original TFT always utilises abstraction, however, in order to complete my program, I will need to recreate/ simulate certain parts of TFT, but by utilising abstraction, I can ignore certain aspects of the game that are irrelevant to my needs, such as the item shop and in-depth graphics they offer. I can also utilise abstraction to create a simplified graph/ breakdown of my project:

Diagram

Description automatically generated

Features.

There are numerous features I consider key to the program, such as:

* Creating teams and battles through a user interface.
* Simulating battles rapidly and efficiently.
* Pausing a battle or going through it tick by tick.
* Looking at the state of the battle through the user interface.
* Should not require internet access and be cross platform.

The ability to create certain teams and battles is key to the core aim of being able to test certain strategies efficiently and without the hassle of a normal TFT game, a user interface is required to allow the user to iterate rapidly over multiple team compositions, meaning they can quickly test numerous teams, as Suket likes to rapidly tweak teams constantly to find the best composition.

Suket only brought a low processing power laptop to university, so battles should be simulated rapidly and efficiently so that even users on computers like his can use the program without issues. The code should aim to be as efficient as possible and in a fast-programming language to accommodate for this.

The program should allow the user to pause the battle or iterate through it tick by tick/ at a slower rate so the user can understand what is occurring in the battle on a deeper level without a result just being broadcast to them, simultaneously the battle should be visible through the user interface so the user doesn’t have to go through the additional effort of visualising the battle in their head, simplifying the process and ensuring they understand what is occurring.

Finally, the program should not require internet access, so users can still strategize and brainstorm while they would not usually be able to access TFT. Internet connection at Suket Arya’s accommodation as university is unstable at best as well, so the program should not require internet access to ensure he can access the program at all times. Moreover, while Suket’s laptop is a Windows device, his home computer is a Mac, so my solution has to be cross platform to allow him to access the program on either device.

Limitations.

There are a few limitations of the program and my development of the program. For starters, I have limited time and resources to pour into the project, which is accentuated by a lack of long term experience in programming and Rust in particular. The desired and full implementation of planned features may not be possible due to time constraints, especially as a set amount of time will have to be dedicated to working my way out of bugs/ pitfalls that more experienced programmers have seen before and know how to avoid.

A lack of money and resources will also holds the project back, as with more money a graphics designer could be hired to create the user interface, something I do not have much experience with. Any user interface I create would pale in comparison to something created by a professional and the presentation may risk driving away users who expect higher quality.

TFT itself is developed by an entire team of experienced developers and so perfectly replicating the game and keeping it constantly updated may be near impossible due to the sheer amount of time it would take, even if my abstraction means I can ignore aspects such as animations and graphics. Finetuning the exact stats of a champion or the interactions between numerous abilities will also be too timely, especially when it can be very hard to recreate certain situations in games to see what occurs, so there may be some discrepancies between TFT and the simulation.

While users will be able to edit/ add their own units with specific stats (so users can assist in keeping the simulation accurate), unique abilities for each character will not be able to be implemented by the any users, due to the fact that abilities are more often than not very complex, requiring multiple lines of code (that users would not be able to write or insert into the program) rather than a simple variable change, so users will have to use one of the abilities provided when editing units.

Unlike other educational methods that give direct feedback or advice, users of this program will be required to assess what went well or poorly in a battle, meaning it may be less useful for newer players, although the program will be useful for new players to get a feel for battles and units in TFT. Moreover, another limitation is that my program won’t cover parts of the game such as the economy or carousel so users will have go elsewhere to educate themselves on those sections of the game.

Finally, while the TFT simulation will be able to move forward through a battle at any rate, as it will not store previous ticks/ iterations in the game loop, moving backwards through the loop to rewatch things will not be possible, which may be an issue if users want to check how something occurred or for searching for bugs.

Hardware.

(Minimum) hardware requirements for the finalised program will be low:

* 1GB (spare) RAM.
* 1GB (available) local storage.
* 1.5 Ghz processor.
* Capable of running Windows 8 (or newer), Linux or MacOS.
* A GPU on par or more powerful than:
  + NVidia: GeForce 560
  + AMD: Radeon HD 6950
  + Intel: Intel UHD

Software required for the program is one of the Windows, Linux or MacOS operating systems.

Requirements:

These are the requirements for the program to be considered a success:

Simulation Requirements:

* All items from TFT implemented.
* Minimum of 3 units from TFT implemented.
* All unit abilities implemented.
* All associated status effects implemented.
* Alterable tick speed of simulation.
* Ability to move forward through simulation at variable speed.
* Accurate simulation result.
* Efficient simulation code.

The reason why I aim to implement all the items yet only 3 of the units is due to how the game is updated. Every set, a new cast of fifty or so units are released, each with their own abilities, however, items are updated and changed much less frequently, by first implementing items, I can avoid wasting time implementing units that will have changed by the time I finish the program.

In order for the simulation to be accurate, the associated status effects that items give have to of course be implemented as well.

The next two requirements, “alterable tick speed of simulation” and “ability to move forward through simulation at variable speed” sound identical/ very similar. To clarify what I mean, “alterable tick speed of simulation” means that the period of time that corresponds to a single tick or frame in the simulation should be changeable, so the user should be able to choose whether 10 milliseconds or 1 millisecond occurs in one tick. Of course, a lower tick speed results in a more accurate simulation, but requires more ticks to be simulated, resulting in a performance trade off which allows for users to get decently fast if slightly more inaccurate simulations even on poorer performance.

“Ability to move through simulation at variable speed” means that the user should be able to specify how fast ticks should occur, obviously there is an upper bound on how fast they can be simulated which is how fast the computer can simulate them, but the user should, if they so choose, be able to step through the simulation tick by tick to get a deeper understanding of what is occurring in the battle.

The aim of this program is to be a useful and educational tool for improving at TFT, meaning that the simulations of the battle must aim to be correct, being able to correctly assert the outcome of a battle to a reasonable level of accuracy. Of course, without knowledge of the code behind TFT, perfect recreation is infeasible, but an acceptable level of accuracy should be attainable. By “simulation” I mean the program should be able to accurately recreate the game mechanics of TFT, such as the placement of units on a board, their movement and the damage they deal to each other and the user of their items and abilities, as well as the affect all of these have on the final outcome of the battle.

Finally, the code/ program behind the simulation must aim to be as fast and efficient as possible, not only to allow user on low hardware computers to utilise the software without buffering and annoyance, but also to give more opportunities and use cases for the future. Theoretically, once the simulation has an acceptable level of accuracy and speed, a machine learning program could run on the simulation to learn how to play TFT, which in itself would have infinitely many use cases, but would require a vast amount of data and thus very fast simulations.

Database Requirements:

* Store and alter unit base stats.
* Store and alter item base stats.
* Store outcomes of previous simulations and what team compositions lead to that outcome.

These requirements allow the database to store and alter unit and item stats, avoiding hardcoding them into the program. By avoiding hardcoding them into the program and instead storing them in the database, it allows other parts of the program to interact with and alter the data during runtime, resulting in the user being able to alter or update these stats either to quench a thirst to change the simulation a certain way, or to update the simulation to more accurately represent the game (after a TFT game patch for instance) after the program has been distributed/ sent out.

Storing previous simulation outcomes means the user can avoid re-simulating battles or trying to remember how a certain battle occurred, they can simply look back instead. However, even with previous knowledge of the battle setup, the outcome that occurs is not guaranteed due to the many random elements within the game, meaning exact re-simulation is not possible.

Interface Requirements:

Frontend-Backend:

* Allow the frontend to send the backend details of a TFT board (units placed, time unit etc)
* Allow the frontend to send a command to start a battle.
* Allow the frontend to send commands to change unit/ item stats.
* Allow the backend to send the state of the battle to the frontend.
* Allow the backend to send the outcome of a battle to the frontend.

Backend-Database Interface Requirements:

* Allow the backend to request unit and item information.
* Allow the backend to alter unit and item information.
* Allow the backend to store simulation outcomes.
* Allow the backend to request simulation outcomes.

These interface requirements guarantee that different parts of the program can communicate with each other effectively, especially consider the program will be written partially in different programming languages, meaning that clear and concise communication between them will be very necessary. These requirements ensure that all the data and commands needed to be transmitted between different sections of the program can be.

User Interface Requirements:

* Allow the user to drag and drop units and items onto a TFT board.
* Allow the user to alter board settings such as the time unit (time per tick).
* Allow the user to start a simulation.
* Allow the user to display/ watch the simulation at a pace of their choosing.
* Allow the user to store/ view previous battles and their outcomes.
* Allow the user to alter unit and item base stats persistently (remains changed after program restart).

The user interface should allow the user to interface with the program and mean that the user can utilise the software in all the ways the software aims to be utilised. These requirements ensure the user is able to create boards, view the simulation occurring and then store the outcome, along with alter unit and item stats as they please.

Success Criteria:

1. The user can, through a UI, place down units and items onto the board and then run the simulation at a pace of their own choosing.
   1. The centre of the program will be a “board” which displays all the current cells on the board, all placed units and their items. Before the simulation is run, users will be able to click on the units to change their items/ level. If the unit is placed in the wrong position, there will be a button to “move” the unit to allow it to be placed elsewhere.
   2. At least 3 units and most (more than half) of the current items will be implemented, along with associated status effects.
   3. On the left of the board will be a list of all current units and items, and will have the ability to drag and drop units and items onto the board.
   4. It will not let you drag an item onto a cell that does not contain a unit.
   5. There will be options on the right side of the board that allows the user to change certain settings, such as the time unit or the number of ticks before a battle is registered as a draw.
   6. There will be a “start battle” button, clicking it will begin the simulation. It will be paused on the first frame. Clicking on units now display their health, status effects, stats etc. On the right side of the screen, instead of board settings there will be simulation settings, there will be a play, pause and skip buttons, you can specify at how many ticks per second you want the simulation to play at, skip a certain number of ticks ahead or go forward by a single tick. The battle will not have been pre-simulated, as the battle is being simulated as the user progresses forward, it will not know when the battle will end, so if the user skips past the end, it lets the user know and displays the outcome and the final board. The user will not be able to go backwards through the battle for reasons explained in the limitations section. Projectiles will also be displayed on the board during the battle.
   7. During the simulation, the UI board will only update a few times a second (when battle is not paused). If the user plays the battle at a high enough tick rate, the UI display will not attempt to keep up to avoid performance problems and will display every so often or when the user pauses/ the battle is over.
   8. After the simulation, you will be able to view the final board (on the tick it ended) and there will be a button to save the outcome and initial board layout.
   9. This should allow the user to, with simple controls, layout a team and watch a battle, ensuring they can rapidly iterate through many different team compositions and matchups. It will allow the user to view all relevant information about a battle and move through the battle as slowly as they want, while avoiding lagging the frontend if the backend is simulating ticks too fast.
2. The simulation will have an acceptable level of accuracy to the game TFT
   1. To be more definitive, the program should be able to successfully state the winner of a battle a minimum of 70% of the time.
   2. A minimum level of accuracy like this is required so the program can actually function as designed and allow the user to get a good grasp on battles within TFT.
3. The program should be able to simulate 3 complex battles in under 10 seconds on a low specification computer.
   1. I will test this on my low-performance laptop which is not far above the minimum hardware requirements to run the software.
   2. I define a “complex” battle as a board which contains more than 3 units on each team, each with at least 1 item.
   3. This success criteria ensures that the program will run efficiently and at an acceptable speed regardless of the hardware it is run on, meaning that Suket can be confident it will work on his laptop.
4. Through the UI, the user will be able to change unit and item stats.
   1. There will be a separate page (accessible through a button) which lists all current units and items. Clicking on a unit or item will display its current stats and allow you to change them. There will be a button located at the bottom that allows you to “save” your changes, pressing it will change the unit’s or item’s stats in the database.
   2. This allows the user to easily update unit and item stats. Giving them the power to alter units grants them the ability to either update the stats themselves in line with game updates or with their own desires.
5. Through the UI, the user will be able to view previous simulations.
   1. There will be a separate page (accessible through a button) which lists all previous simulations, their outcomes and the board state before the battle begun.
   2. Viewing previous simulations allows users to avoid rerunning battles pointlessly and to have a simple and conveniently placed location to view battle outcomes. It allows users to build up a catalogue of battles which they could look through in game to see what is best against certain team compositions or similar.
   3. There will not be option to “rewatch” the battle as it occurred, as due to the innate randomness within TFT, it would be simulated differently to the initial run.

# Design.

Objectives.

By looking back at my success criteria and requirements, I can create a list of objectives to aim for when designing my program:

* The user interface must be intuitive and easy to use, in order to avoid annoyance and allow all users to easily get to grips with the program.
  + The user interface must feature drag and drop mechanisms for all items and units, to allow the user to quickly create and iterate through different teams and battles.
  + The user interface must contain inputs and buttons to allow for watching battles and setting up the board, to give the user the control and ability to see what occurs in a battle.
    - These controls include pausing, playing, stepping forward and more.
  + The user interface must enable the user to save and view simulations and edit stats, so the user can build up a catalogue of battles to look over in their TFT games to relearn matchups.
* The simulation must have an acceptable level of accuracy, to ensure that users get helpful and relevant information and knowledge to use in their TFT games.
  + The simulation must have consistent and accurate movement to TFT, to ensure the simulation is accurate and to not increase the strength of ranged or melee units respectively.
    - Slower movement speed increases the strength of ranged units as they can attack uncontested for longer, with the opposite true for melee units.
  + The simulation must have accurate damage calculations, to ensure that, for instance, armour or ability power is not stronger in the simulation than real TFT.
  + Unit pathfinding in the simulation must be simple, but as accurate as possible (to the game), to avoid slowdowns from calculation heavy pathfinding and allow the program to run quickly and efficiently.
* The program must have the ability to store, retrieve and alter the results of past simulations, unit stats and item stats.
* As the program will utilise different languages and technologies for different parts of the program (frontend, backend, database), the program must be able to effectively communicate between the different components of itself.
* The program will be able to be utilised as an educational tool for TFT players and a tool for them.
* The program should have a resizable resolution (within reason) to allow the end-user to use the software on a variety of monitors and hardware.

The colour scheme for the program will be dark and grey, generally using white text and black or purple outlines for clarity.

Breaking down the Problem.

I can break down the problem/ task of creating my program into smaller, easier steps using decomposition. Moreover, I can recall my success criteria and requirements to ensure that I am on path for creating the program successfully and that what I have made so far ticks each box.

Diagram

Description automatically generatedI can refer to my decomposition diagram:

Referring to the decomposition diagram, I have broken down the TFT simulator program into a number of modules and smaller problems.

Below I will further decompose large/ complex sections of my diagram.

Battle Simulation:

Once the backend has received the signal to start the battle, and of course the board itself, the first action it has to perform is to setup the board, which I can break down further.

Setting up the board involves:

* Unit Setup:
  + Unit setup is the biggest step. The board has to create two empty vectors to hold the two teams’ units. It must be vectors to allow for their size to vary as units die.
  + The board must also create a dead champions vector to hold champions who have died (as any projectiles they have created must still deal the correct amount of damage and so require the unit details).
  + Then, to fill the two vectors, it must instantiate the unit details it received from the frontend to the correct struct and in the right location, giving each unit a unique id along with this.
  + Following this, each of their item effects has to be granted, giving the units their accompanying stat buffs or status effects.
  + Finally, their initial health must be set to the correct value.
* Projectile Setup:
  + Setup two empty vectors to hold any projectiles the two teams create. It must be vectors to allow for the size to vary as projectiles are added or removed as the game goes on.

Once the board has been successfully setup, the game loop begins until the battle has finished.

The game loop involves:

* Checking for the end of the battle:
  + If either team has zero alive champions remaining (champion vectors are empty), the battle should end and the winner declared.
* Simulating projectiles:
  + The board should iterate through each projectile and for each projectile:
    - Perform its movement action.
    - Check for collisions with any champions (on the opposing team)
      * If there are any collisions, it should deal damage to the champion, this requires finding the unit that fired the projectile as well, so the correct amount of damage is dealt.
      * If the projectile has splash damage, damage should also be dealt to any adjacent enemy champions.
    - Check if the projectile is out of bounds:
      * If the projectile is out of bounds, it will no longer have an opportunity to hit any champion and so should be deleted, to avoid unnecessary computation.
* Simulating units:
  + The board should iterate through every unit and for each unit simulate its turn. Its turn involves:
    - Checking to see if it has zero health or below, in which case it should remove itself from the vector.
    - Performing all status effects currently ailing the champion, checking to see if it has died after performing these statuses.
    - If it is still alive, it should try to move towards or attack its target:
      * If the unit does not currently have a target, it should locate the nearest enemy champion that is targetable.
      * The unit should check the distance to its target.
        + If the target is in range, it should auto attack the target, checking to see if it dodges or dealing the associated damage and giving itself the associated mana.
        + If the target is not in range, it should move to its pathfinding target cell:

If the unit does not have a current pathfinding target cell, it should find the first available cell that reduces the distance to the target by the most.

Once the unit has a pathfinding target cell, it should add any movement progress then check to see if it has enough progress to move into the next cell.

* + - Once it has moved or attacked, it should check to see if it has enough mana to cast its ability.
      * If it has enough mana to cast its ability, it should, performing the ability associated with the champion.

After simulating all of this, the game loop should begin again on the next tick, repeating these steps until one of the teams has won. However, the board should also keep track of how many ticks it has performed, incrementing it by one each time it performs the game loop, if the tick count goes above a certain value, the battle should finish and be declared a draw, to avoid any never-ending battles that are simulated indefinitely.

Frontend - UI:

The UI has two different jobs depending on whether a simulation is currently occurring.

If a battle has not begun, the UI should allow the user to setup the board, giving users the ability to drag and drop items and units onto the board in certain locations:

Board with units denoted with characters. Clear cell outline to help those with sight impairments distinguish between cells.

Drag and drop items and units

Start battle and configuration buttons/ inputs.

Popup from pressing Unit C, showing heal, items and star level, all configurable.

The UI should also allow the user to navigate to a different page to allow the user to edit unit stats or view the results of previous simulations.

If the battle has begun the UI should:

* Have buttons controlling navigation through the battle. The battle should begin paused, with buttons to play, step forward 1 stick, step forward x ticks, and change how many ticks are played per second.
* If the battle is currently playing, or the user has stepped forward some ticks and needs the backend to simulate some ticks:
  + The frontend should send a command to the backend asking it to perform x ticks.
  + Once these ticks have been performed, then the backend should return the state of the board to the frontend so the user can view the battle.
  + The frontend should be able to interpret the board it has received and display it correctly, allowing the user to view the units status/ health as well so they can get a deeper understanding of the battle.
  + There will be a limit to the rate at which the UI requests updates from the backend, to avoid lagging the frontend by trying to too rapidly update the UI/ display of the board.



Board with units denoted with characters. Clear cell outline to help those with sight impairments distinguish between cells.

Play, pause, step forward and play options all available.

Unit C selected, showing their stats and details.

Database:

The database should be in communication with the backend and should appropriately respond to any requests it receives. There are a few requests it may receive:

* Retrieve list of unit ids:
  + If this is requested, it should return the unit ids of all units in the backend, allowing the backend to request further details on all of them.
* Retrieve unit stats:
  + If this is requested, the database should return the unit stats/ details correlating to the associated id it received with the request.
* Retrieve list of item ids:
  + If this is requested, it should return the item ids of all item in the backend, allowing the backend to request further details on all of them.
* Retrieve item stats:
  + If this is requested, the database should return the item stats/ details correlating to the associated id it received with the request.
* Alter unit stats:
  + If this is requested, the database should change the unit stats to the stats it received along with the request, for the unit that is associated with the id it received in the request.
* Alter item stats:
  + If this is requested, the database should change the item stats to the stats it received along with the request, for the item that is associated with the id it received in the request.
* Retrieve list of ids of previous simulations:
  + If requested, the database should return a list of the ids of all previous simulations, allowing the backend to query each of them.
* Retrieve details of previous simulation:
  + If requested, the database should return the details of the simulation associated with the id received in the request.
* Add new simulation:
  + If requested, the database should add a new entry to the database with details of the simulation it received.

The database should always assume that any data it receives is accurate and in a valid format, with the backend doing the job of validating any requests/ data it receives for the database from the frontend (ie ensuring that the new simulation has all the details it needs).

Languages:

To develop my program, I will be utilising a library known as Tauri. Tauri is a toolkit for developing applications. All programs written with Tauri can be compiled to Windows, Linux and Mac, allowing me to fulfil my requirement of cross platform development. Moreover, developing for a single platform with Tauri doesn’t take any more or less time than for all available platforms, so I can help lessen the that limitation and avoid time constraints due to developing for multiple platforms.

Tauri allows you to develop with Rust on the backend and with Javascript on the backend. Rust is suitable for my backend as it is an incredibly fast, compiled language, matching up with C and C++ in terms of performance. This will help with my requirement of my program running quickly and efficiently, even on low specification computers. As good performance was a necessity, choosing a programming language for this restricted me to the few choices of Rust, C and C++, all of which I have little experience with. This also makes Rust more preferable, with its compiler renowned for its helpful error messages and pre-compile checking, helping me avoid errors and time spent researching vague issues.

Rust also features a strict type system and an advanced borrow checking/ memory safety system. These all help catch and fix bugs earlier in the development cycle, helping me avoid time debugging and ensures that the final program is more robust and less prone to crashing, meaning that the tool can avoid crashing at crucial times such as in a game and save the user’s time by avoiding rerunning simulations.

To build the UI, I will be utilising Javascript, specifically through the Sveltekit framework. Sveltekit is useful in a number of ways. Sveltekit provides inbuilt tools for routing and page navigation, allowing me to easily build the different pages I need for the program (main page, previous simulations page, edit unit stats page).

Sveltekit also works in a modular fashion, utilising components to avoid rewriting code for the UI, this means that for repeated patterns in my UI, such as the units on the board and each entry of a previous simulation, I can make a component for the pattern and avoid rewriting code, also making the code less bloated.

Finally, Sveltekit compiles down to pure Javascript, resulting in much faster runtimes than other frameworks, something the UI will need to keep up with the Rust backend.

To round out my program, I will use SurrealDB for my database, an SQL like language that interfaces well with Rust and offers many quality of life features that mean that development with the database will be easier.

Usability Features:

My program will feature a number of usability features with the aim of making the program as easy to use as possible. These features include:

* Clear UI:
  + All text will be white on a black background, or vice versa, or another two contrasting colours on top of each other.
  + The different hexagons on the board/ grid will feature a distinctive outline to highlight the different hexagons.
  + Buttons will have a clear outline that contrasts the background to ensure they stand out.
  + All of these features will give clarity to the UI and improve visibility, helping normal users but especially those suffering from visual impairments or colour-blindness.
  + All buttons will be clearly placed on either side of the screen, with the board consistently in the middle. This ensures that buttons will never obstruct or obscure the board and helps the user always know where the UI is.
  + The UI will be resizable (within reason), allowing the program to be used on a variety of different monitors and hardware without issues.
* Ease of Use:
  + Units and items will be drag and drop.
    - This avoids clunky inputs where the user has to type in the location they want, it speeds up team building
  + In battles, unit health/ details will be retrievable by clicking directly on them.
    - Allows users to quickly find the details of the unit they want, ensures the information is put in an obvious and convenient place, helps the user avoid selecting the wrong unit, possible if in drop down for example.
    - The unit selected will also be made exceptionally clear, with the unit name and type displayed in large text to avoid confusion.
  + There will be UI allowing the user to watch through a battle at any pace, allowing them to view any details or information they may want.
* Robust and Efficient Program:
  + The program will be coded in a high performance language with efficiency in mind, allowing users to use the program on low-end machines or perform innumerable simulations in a second, avoiding annoyance at slow speeds. The program requires it be robust to avoid crashing when hundreds of simulations must be performed in a short period, frequent crashes would slow down the process immeasurably.
  + There will be a number of user input fields where the user can input data, from the time unit to changing unit stats. All fields will be checked to ensure the data in them is valid before any calculations or processing is done with them, for example, checking and giving a warning to the user if they inputted a character or a float instead of an integer for the time unit, for example. It allows the user to feel safer and helps them know they will not break something by accident by tinkering with settings designed to be changed.

Algorithms and Classes:

There are a number of classes and algorithms that will be crucial for development of the program. Classes encapsulate key data and methods into a single location and helps remove redundant and repeated code. Algorithms are equally important, describing the sequence of actions required to solve a certain problem.

Grid:

The grid and coordinate system will work as shown below. A picture containing clock

Description automatically generated



One benefit of using a coordinate system like this is that z can be calculated with z = -x – y, allowing the storage to be more efficient and save memory, allowing the program to run faster.

Classes:

As most of the functionality of the program will be encapsulated in classes and methods, most if not all of the key algorithms will be in class methods. Below is an overview of all of the key classes in my program.

class Location:

Description:

Class that describes the location of a PlacedChampion or SummonedChampion.

Location describes a place on a hexagon grid, however, two dimensional coordinates

cannot fully/ accurately describe position on graph, so additional "z" value needed, however

z (which is used when calculating distance) can be calculated with z = -x - y therefore only

x and y positions need to be saved and z can be calculated.

Most methods public as Location's as usually attached to a separate class that wants to utilise methods of Location.

Justification:

By creating a class for Location the class can hold many key methods to help reduce redundant and repeated code, providing methods for

common actions such as checking the distance between two points. Also allows for different classes to hold a Location attribute and therefore

have easier interaction between them.

Attributes:

-(pub) x : i8,

Stores the x value.

i8 as positions can be negative yet does not need to store large values.

-(pub) y : i8,

Stores the y value.

i8 as positions can be negative yet does not need to store large values.

Methods:

calculate\_z:

Calculates the z value of self.

Psuedocode:

FUNCTION calculate\_z(self) -> i8:

RETURN -self.x - self.y

ENDFUNCTION

(pub) distance\_between\_points:

Calculates the distance between two locations. For a distance of 1 hex, returns 2, for this reason the range of all Champions are doubled when converted into SummonedChampion.

Psuedocode: (ABS = absolute value)

FUNCTION distance\_between\_points(self, other\_pos : Location) -> i8:

RETURN ABS(self.x - other\_pos.x) + ABS(self.y - other\_pos.y) + ABS(self.calculate\_z() - other.calculate\_z())

ENDFUNCTION

(pub) sub\_positions:

Subtracts two positions, returns a new position.

Psuedocode:

FUNCTION sub\_positions(pos\_one : Location, pos\_two : Location) -> Location:

RETURN Location { x : pos\_one.x - pos\_two.x, y : pos\_one.y - pos\_two.y }

ENDFUNCTION

(pub) add\_position\_vec:

Adds an array to a Location, returning a new location.

Psuedocode:

FUNCTION add\_position\_vec(location : Location, pos\_array = [i8 ; 2]) -> Location:

RETURN Location {x : location.x + pos\_array[0], y : location.y + pos\_array[1]}

ENDFUNCTION

(pub) check\_valid:

Checks if the location is a valid position in the grid

Psuedocode:

FUNCTION check\_valid(self) -> bool:

RETURN self.x >= 0

&& self.x < 10

&& self.y >= 0

&& self.y < 8

&& 2 - (self.y / 2) < self.x

&& 10 - (self.y / 2) > self.x

ENDFUNCTION

(pub) get\_closest\_to\_location:

Takes a list of locations or class with a location and returns the closest to self position.

(pub) get\_closest\_to\_location\_targetable:

Takes a list of SummonedChampion's and returns the closest to self location that is targetable.

(pub) get\_closest\_to\_location\_targetable\_index:

Takes a list of SummonedChampion's and returns the closest to self location that is targetable, along with its index in the list.

(pub) get\_within\_distance):

Takes a list of SummonedChampion's and returns a list of all SummonedChampions within provided distance.

class Champion:

Description:

Class that describes the base stats of a champion.

Attributes:

-(pub) id : u8,

ID of the champion.

Public so store can access id.

u8 as no need to be negative and not many champions so no need for high value

-hp : f32,

Base healthpoints of the champion.

f32 so operations can be done involving division and percentages without losing precision.

-sm : i16,

Starting mana for the champion.

i16 so it can be used in calculations with current mana (attribute on SummonedChampion) without conversions.

-mc : i16,

Mana cost for the champion.

i16 so it can be used in calculations with current mana (attribute on SummonedChampion) without conversions.

-ar : f32,

Base armour of the champion.

f32 so it can be used in calculations with hp without conversions and to allow it to act like a percentage.

-mr : f32,

Base magic resist of the champion.

f32 so it can be used in calculations with hp without conversions and to allow it to act like a percentage.

-ad : f32,

Base attack damage of the champion.

f32 so it can be used in calculations with hp without conversions and to allow it to act like a percentage.

-attack\_speed: f32,

Base attack speed of the champion.

f32 so can store decimal point attack speeds (attacks per second), without needing to divide by 100.

-ra : i8,

Range of the champion.

i8 so it can be used in calculations with the Location class.

Methods:

from\_object:

Converts from an object collected from the database into the Champion class.

Required so users can retrieve altered Champion's from the database, allowing alterations they have made to persistent across app runs.

Pseudocode:

FUNCTION from\_object(obj) -> Champion:

let ad = obj.get("ad") //retrieves ad from dictionary like object "obj"

let ar = obj.get("ar")

let attack\_speed = obj.get("attack\_speed")

let hp = obj.get("hp")

let id = obj.get("id")

let mc = obj.get("mc")

let mr = obj.get("mr")

let ra = obj.get("ra")

let sm = obj.get("sm")

RETURN Champion { id, hp, sm, mc, ar, mr, ad, attack\_speed, ra }

ENDFUNCTION

into\_values:

Converts a Champion into an array of key value pairs for storage in the database.

Required so users can save altered Champion's from the database, allowing alterations they have made to persistent across app runs.

Psuedocode:

FUNCTION into\_values(self) -> [(String, Value) ; 9]:

RETURN [

("id", self.id),

("hp", self.hp),

("sm", self.sm),

("mc", self.mc),

("ar", self.ar),

("mr", self.mr),

("ad", self.ad),

("attack\_speed", self.attack\_speed),

("ra", self.ra)

]

ENDFUNCTION

Constants:

There is one constant involving the Champion class:

DEFAULT\_CHAMPIONS:

Array of champions : [Champion ; 4]

If there are no champions stored in the database (common after first time initialisation of the database)

then DEFAULT\_CHAMPIONS will be used to fill the database with the default champions.

Value:

[

Champion {

id: 0,

hp: 1100.0,

sm: 70,

mc: 140,

ar: 0.25,

mr: 0.25,

ad: 70.0,

attack\_speed: 0.6,

ra: 2,

},

Champion {

id: 1,

hp: 1400.0,

sm: 50,

mc: 100,

ar: 0.45,

mr: 0.45,

ad: 100.0,

attack\_speed: 0.7,

ra: 1,

},

Champion {

id: 2,

hp: 1200.0,

sm: 35,

mc: 100,

ar: 0.25,

mr: 0.25,

ad: 120.0,

attack\_speed: 0.7,

ra: 3,

},

Champion {

id: 3,

hp: 1200.0,

sm: 35,

mc: 150,

ar: 0.25,

mr: 0.25,

ad: 60.0,

attack\_speed: 0.6,

ra: 3,

}

]

class PlacedChampion:

Description:

Class to be used to represent a champion placed on a board during setup. Holds a reference to Champion and is converted into SummonedChampion during a battle.

Justification:

Due to the existence of the champion class, the PlacedChampion class can be simple, only holding an id correlating to a champion.

This means the new class only needs to hold values that make it unique to other PlacedChampion's of the same type, such as its location or items, reducing the quantity of memory used by the program.

Attributes:

-id : usize,

Id of the PlacedChampion, correlates to the ID of a Champion.

Usize so the id can directly index arrays to retrieve the associated champion.

-star : usize,

Star level of the PlacedChampion.

Usize so the star level can directly index arrays where values change depending on the star level.

-items : [u8; 3],

Ids of the items it holds, correlates to the ID and index of an item.

Can hold 3 items so array of length three.

u8 as that is the datatype of the id of items.

-location : Location,

Location of the PlacedChampion.

-team : Option<u8>,

Team of the PlacedChampion.

This attribute is only used when storing and retrieving PlacedChampion's from the database.

When this class is used in the backend, the two separate teams of PlacedChampion's are stored in two

lists, but when PlacedChampion's are stored in the backend, they are stored together and thus need a record

of what team they are on so the board can be reconstructed accurately.

u8 to allow for easy storage, option so the value can be None when not needed/ unused.

Methods:

from\_object:

Converts from an object collected from the database into the PlacedChampion class.

Psuedocode:

FUNCTION try\_from(obj) -> PlacedChampion:

let id = obj.get("of\_champ") //retrieves id from dictionary like object "obj"

let item\_0 = obj.get("item\_0")

let item\_1 = obj.get("item\_1")

let item\_2 = obj.get("item\_2")

let star\_level = obj.get("star")

let location\_x = obj.get("location\_x")

let location\_y = obj.get("location\_y")

let team = obj.get("team")

RETURN PlacedChampion { id, items : [item\_0, item\_1, item\_2], star : star\_level, location : Location { location\_x, location\_y }, team : Some(team) }

ENDFUNCTION

into\_values:

Converts a PlacedChampion into an array of key value pairs for storage in the database.

Psuedocode:

FUNCTION into\_values(self) -> [(String, Value) ; 7]:

RETURN [

("of\_champ", self.id),

("star", self.star),

("item\_0", self.items[0]), //need to separate items as arrays cannot be stored in database.

("item\_1", self.items[1]),

("item\_2", self.items[2]),

("location\_x", self.location.x), //need to separate x and y as Location cannot be stored in database.

("location\_y", self.location.y)

]

ENDFUNCTION

class SummonedChampion:

Description:

Class that represents a champion on a board during a battle. Constructed from a SummonedChampion.

Justification:

Unlike PlacedChampion that only has to hold the id of a champion and its location (essentially), SummonedChampion needs to hold all the stats/ fields that are required for the simulation as there will be plenty of ways to alter the stats of a SummonedChampion from the base stats on the Champion. For example, placing an item on a SummonedChampion will change the stats of it, that is why the class must hold the require fields so they can be altered. As SummonedChampion's are only required while simulating the battle, while creating or viewing a board PlacedChampions are sufficient as they use much less memory.

Attributes:

-(pub) location : Location,

Location of the SummonedChampion

-of\_champ\_id : usize,

Id of the PlacedChampion/ Champion it was constructed from.

-movement\_progress : [i8 ; 2],

Progress towards moving towards a new cell.

i8 to allow it to be used in calculations with Location without conversions.

-health: f32,

Current health of the SummonedChampion.

f32 to allow for calculations with percentages and division while holding precision.

-cm : i16,

Current mana.

i16 as current mana can be negative (if given debuff from certain item).

-dc : u8,

Dodge chance.

Used as a percentage.

u8 as it cannot be negative or larger than 100, so u8 to save space.

-cr : u8,

Critical strike rate.

Used as a percentage.

u8 as it cannot be negative.

-crit\_damage : f32,

Critical strike damage multiplyer.

f32 to allow it to store decimal increases and be used in calculations with health without conversions, which would slow program.

-mc : i16,

Mana cost of ability.

i16 to allow it to be used in calculations with current mana without conversions.

-ar : f32,

Armour of SummonedChampion.

f32 to allow it to be treated like percentage and be used in calculations with health without conversions, which would slow program.

-mr : f32,

Magic resist of SummonedChampion.

f32 to allow it to be treated like percentage and be used in calculations with health without conversions, which would slow program.

-ad : f32

Attack damage of SummonedChampion.

f32 to allow it to hold decimal values and be used in calculations with health without conversions, which would slow program.

-attack\_speed : f32

Base attack speed from SummonedChampion.

f32 to allow it to store decimal values of attacks per second.

-ra : i8,

Basic attack range of the SummonedChampion, double the value of the range on the associated Champion.

i8 to allow it to be used in calculations with location without conversions, which would slow program.

-id : usize,

Unique id of the SummonedChampion to allow the object to be uniquely identified.

-target\_cooldown : i8,

Variable that holds the length of time (in centiseconds) before it should update its current target (to the nearest located enemy champion).

i8 to reduce the chance of underflow error when subtracting the time unit from it.

-auto\_attack\_delay : i16,

Holds the length of time (in centiseconds) before the SummonedChampion can auto attack again. Calculated by 100 / (attack\_speed \* attack\_speed\_modifier).

-attack\_speed\_modifier : f32,

Attack speed modifier from items and abilities.

f32 to hold percentage/ decimal changes and to allow it to be used in calculations with attack speed without conversions.

-target : usize,

The id of the SummonedChampion that this SummonedChampion is currently targeting.

-target\_cells : Location,

The stored target cell for pathfinding, attempts to move towards this spot.

-items : [u8 ; 3],

Array of the ids of the items the SummonedChampion is currently holding

-ap : f32,

Ability power, a measure of how effective abilities are/ how much damage they do etc.

f32 to allow it to hold decimal values and be used in calculations with health without conversion.

-se : Vec<StatusEffect>,

A list of all the status effects currently affecting the SummonedChampion.

List/ vector as it needs to be able to grow and shrink in size

-gain\_mana\_delay : i16,

Delay until SummonedChampion can start generating mana again, to ensure consistency with actual TFT (champions cannot generate mana 1 second after casting ability)

-star\_level : usize,

Star level of the SummonedChampion

usize to allow it to index arrays for values that change depending on star level.

-incoming\_damage\_modifier : f32,

A percentage increase or decrease to incoming damage, altered by some abilities.

f32 to allow it to hold percentage/ decimal values accurately and be used in calculations with health without conversion.

-initial\_hp : f32,

Attribute that stores the initial healthpoints of the SummonedChampion, ensuring the SummonedChampion doesn't overheal and allows the value to be retrieved for effects that activate at half health etc.

-targetable : bool,

Stores whether the SummonedChampion can be targeted by abilities and auto attacks.

bool as it is a true or false value

-shed : u8,

Whether the SummonedChampion is in the process of shedding negative status effects, can be 0, 1 or 2 measuring process of shed effect.

u8 as it cannot be negative or store large values, allowing less data usage.

-shields : Vec<Shield>,

List of shields on the SummonedChampion

List/ vector as it needs to be able to grow and shrink in size

-zap : bool,

Stores whether currently "zapped" by ionic spark.

bool as it is true or false value.

-banish : bool,

Whether the SummonedChampion is currently "banished".

bool as it is true or false value

-titans\_resolve\_stacks : u8,

Stores the current stacks from titan's resolve item.

u8 as it cannot be negative and only needs to go up to 25.

-omnivamp : f32,

Stores omnivamp (healing from damage)

f32 so it can hold percentage values and so it can be used in calculations with damage without conversions.

-shiv\_attack\_count: u8,

Stores the current attack count for stattik shiv item.

u8 as it only needs to contain the values 0 to 4.

Methods:

new:

Creates a new SummonedChampion from a placed\_champion and an id.

setup:

Gives all item stat increases and initialises SummonedChampion stats to base stats given by its respective Champion.

heal:

Provides healing to the SummonedChampion

Psuedocode:

FUNCTION heal(self, healing\_amount : f32):

IF self.se.contains(GreviousWounds):

healing\_amount /= 2

ENDIF

self.health = MIN(self.health + healing\_amount, self.initial\_hp)

ENDFUNCTION

(pub) take\_turn:

Takes the turn of the SummonedChampion, returns a boolean describing if the SummonedChampion is still alive.

False = dead, true = alive

Psuedocode: (Option is a type that contains a value which can be something or nothing)

FUNCTION take\_turn(self, friendly\_champions : Vec<SummonedChampion>, enemy\_champions : Vec<SummonedChampion>, time\_unit : i8, movement\_amount : i8, projectiles : Vec<Projectile>) -> bool:

IF self.health <= 0:

RETURN False

ENDIF

self.target\_cooldown -= time\_unit

self.auto\_attack\_delay -= time\_unit

self.gain\_mana\_delay -= time\_unit

IF self.banish:

RETURN True //is banished, end turn

ENDIF

let stun : BOOLEAN = false

FOR status\_effect in self.se:

IF status\_effect == stun:

stun = True

ENDIF

self.perform\_status(status\_effect, friendly\_champions, enemy\_champions, time\_unit)

ENDFOR

IF self.health <= 0:

return False //killed by status effect

ENDIF

FOR shield in self.shields:

shield.update\_shield(time\_unit) //reduce remaining duration of shield

IF shield.duration <= 0:

self.shields.remove(shield)

ENDIF

ENDFOR

IF stun:

return True //stunned, end turn

ENDIF

let need\_new\_target\_cell : BOOLEAN = false //assumes doesn't need new pathfinding target cell

let target\_object : Option<SummonedChampion> = None //creates variable to store a target object, but sets to None by default.

IF self.target\_cooldown >= 0: //doesn't want to find new target.

FOR enemy\_champ in enemy\_champions:

IF enemy\_champ.id == self.target:

target\_object = Some(enemy\_champ)

break

ENDIF

ENDFOR

IF target\_object == None: //couldn't find target or need new target.

self.target\_cooldown = 100 //reset target cooldown

need\_new\_target\_cell = true //reset pathfinding

target\_object = Some(self.location.get\_closest\_to\_location\_targetable(enemy\_champions))

ENDIF

let target\_object : SummonedChampion = target\_object.unwrap() //remove the Option from summonedchampion.

let distance\_to\_get : i8 = self.location.distance\_between\_points(target\_object.location)

IF distance\_to\_target <= self.ra: //target in range

IF self.auto\_attack\_delay <= 0: //auto-attack ready

self.auto\_attack\_delay = 100 / (self.attack\_speed \* self.attack\_speed\_modifier)

IF self.gain\_mana\_delay <= 0: //can gain mana

self.cm += 10

ENDIF

IF target\_object.dc < GENERATE\_RANDOM\_NUMBER(0 to 100): //checking for a dodge

self.deal\_damage(friendly\_champions, target\_object, self.ad, PhysicalDamage)

ENDIF

ENDIF

ELSE: //target not in range

IF need\_new\_target\_cell || self.location == self.target\_cells: //find new target cell

self.target\_cells = self.location //if can't find new target cell, then don't pathfind anywhere

let lowest\_distance : i8 = i8::MAX;

let new\_position : Location;

FOR possible\_move in [[0, -1], [1, -1], [1, 0], [-1, 0], [-1, 1], [0, 1]]:

new\_position = Location::add\_position\_vec(self.location, possible\_move)

let distance\_from\_target : i8 = new\_position.distance\_between\_points(target\_object.location)

IF distance\_from\_target < lowest\_distance AND new\_position.is\_valid(): //if lower distance and a valid position

let collision : BOOLEAN = False

FOR friendly\_champ in friendly\_champions:

IF new\_position == friendly\_champ.location: //make sure we don't pathfind into collision

collision = True

BREAK

ENDIF

ENDFOR

IF collision:

CONTINUE

ENDIF

lowest\_distance = distance\_from\_target //update lowest distance and target cells

self.target\_cells = new\_position

ENDIF

ENDFOR

ENDIF

self.movement\_progress[0] += movement\_amount \* sign(self.target\_cells.x - self.location.x) //sign converts positive numbers to 1, zero to zero and negative numbers to -1, so just calculates the direction.

self.movement\_progress[1] += movement\_amount \* sign(self.target\_cells.y - self.location.y)

IF ABS(self.movement\_progress[0]) >= 10: //ready to move to new cell

self.location.x += sign(self.movement\_progress[0]) //add movement

self.movement\_progress[0] = 0 //reset movement progress

ENDIF

IF ABS(self.movement\_progress[1]) >= 10:

self.location.y += sign(self.movement\_progress[1]) //add movement

self.movement\_progress[1] = 0

ENDIF

ENDIF

IF self.cm >= self.mc: //ready to cast ability

self.cm = 0

self.gain\_mana\_delay = 100

self.cast\_ability(friendly\_champions, enemy\_champions, projectiles)

ENDIF

ENDFUNCTION

deal\_damage:

Deals damage to a enemy champion, calculating all associated effects.

Psuedocode:

FUNCTION deal\_damage(self, friendly\_champions : Vec<SummonedChampion>, target : SummonedChampion, damage\_amount : f32, damage\_type : DamageType):

let damage : f32 = damage\_amount \* target.incoming\_damage\_modifier

let can\_crit : BOOLEAN = false //magic and true damage can only crit under certain conditions

let crit\_damage : f32 = self.crit\_damage

IF damage\_type == PhysicalDamage:

can\_crit = true

damage /= 1 + target.ar //damage reduction due to armor

ELSE:

can\_crit = self.items.contains(27) //requires a certain item to crit

IF damage\_type == MagicDamage:

damage /= 1 + target.mr //damage reduction due to mr

ENDIF

ENDIF

IF can\_crit AND self.cr > GENERATE\_RANDOM\_NUMBER(0 to 100): //if can crit and does crit

let additional\_crit\_damage : f32 = damage \* crit\_damage

IF target.items.contains(44): //reduce damage due to bramble vest item

additional\_crit\_damage /= 4

ENDIF

damage += additional\_crit\_damage

ENDIF

IF self.items.contains(16): //give bonus giant slayer damage

IF target.initial\_hp >= 2200:

damage \*= 1.45

ELSE:

damage \*= 1.2

ENDIF

ENDIF

self.heal(damage \* self.omnivamp)

IF damage\_type != PhysicalDamage AND self.items.contains(12): //give gunblade healing

let healing : f32 = damage / 4

self.heal(healing)

let lowest\_hp\_ally : SummonChampion = friendly\_champions[0]

FOR friendly\_champion in friendly\_champions: //fetch lowest HP ally

IF friendly\_champion.health < lowest\_hp\_ally.health:

lowest\_hp\_ally = friendly\_champion

ENDIF

ENDFOR

lowest\_hp\_ally.heal(healing)

ENDIF

FOR shield in target.shields:

damage = shield.handle\_damage(damage, damage\_type)

IF damage <= 0:

BREAK

ENDIF

ENDFOR

self.titans\_resolve\_stacks = MIN(25, self.titans\_resolve\_stacks + 1) //give titan resolve stack

target.titans\_resolve\_stacks = MIN(25, self.titans\_resolve\_stacks + 1) //make sure stacks do not go above max of 25

target.health -= damage //deal damage

IF target.gain\_mana\_delay <= 0: /if can gain mana

targget.cm += damage \* 0.7 //give 70% of damage to keep consistency with TFT

ENDIF

ENDFUNCTION

cast\_ability:

Casts the ability of the SummonedChampion

Arguments: self, friendly\_champions : Vec<SummonedChampion>, enemy\_champions : Vec<SummonedChampion>, projectiles : Vec<Projectile>

(pub) get\_is\_targetable:

Method that returns whether the SummonedChampion is targetable

Psuedocode:

FUNCTION get\_is\_targetable(self) -> bool:

RETURN self.targetable && NOT self.banish

ENDFUNCTION

give\_item\_effect:

Takes an item as an argument and gives its stats increases to the SummonedChampion as well as any other secondary effects.

(pub) equal\_id:

Takes in an id as input and checks if it is equal to its own id, instance of encapsulation.

(pub) is\_shred:

Checks if self.shed == 2, encapsulation.

(pub) update\_shred:

Updates self.shred

Psuedocode:

FUNCTION update\_shred(self):

IF self.shed == 1:

self.shed = 2

ELSE:

self.shed = 0

ENDIF

ENDFUNCTION

perform\_status:

Performs and updates the inputted status effect, reducing its duration and performing any effects it has.

enum DamageType:

Description:

A type of variable which can take on one of three forms.

PhysicalDamage, MagicalDamage or TrueDamage.

Correlates to the three damage types in TFT

Justification:

Required to differentiate between the damage types, by creating an enum I can create more readable and understandable code as opposed to an integer id system (where 0 is true damage, 1 is physical etc).

It also means I can ensure all damage types are accounted for when writing code and easily compare/ check damage types.

class Item:

Description:

Holds the stat increases an item gives.

All attributes are public to allow for SummonedChampion to retrieve their stats.

Attributes (pub):

-id : u8,

Stores the id for this respective item

-health : f32,

Health increase given by the item

-ad : f32,

Ad increase given by the item

-ap : f32,

Ap increase given by the item

-ar : f32,

Armor increase given by the item

-mr : f32,

Magic resist increase given by the item

-attack\_speed\_modifier : f32,

Attack speed increase given by the item

-ra : i8,

Range increase given by the item

-cr : u8,

Critical strike rate increase given by the item

-dc : u8,

Dodge chance rate increase given by the item

-cm : i16,

Current mana increase given by the item

-omnivamp : f32,

Omnivamp increase given by the item

-crit\_damage : f32,

Crit damage increase given by the item

Methods:

into\_values:

Converts the item into a key value array to insert into the database.

Required to users can save altered item stats allowing changes to be persistent.

try\_from\_object:

Trys to create an item object from an dictionary like object retrieved from the database.

Required to users can retrieve altered item stats allowing changes to be persistent.

class Projectile:

Description:

A class for a projectile, can be created by abilities.

Encapsulates the behaviour of a projectile, simplifying code and avoiding redundant code. For instance, by adding a location field to the projectile, it alows the projectile to check if it is out of bounds easily, being able to delete itself if so to avoid wastefully simulating something that will not come into play.

Attributes:

location : Location,

Current location of the projectile

location\_progress : [i8 ; 2],

Movement progress of the projectile

target\_location : Option<Location>,

Target location for the projectile, option as the projectile may aim for a SummonedChampion instead.

target\_id : usize,

Id of the SummonedChampion it is aiming for, ignored if there is a target location.

damage : f32,

Damage the projectile will do

damage\_type : DamageType,

Damage type of the projectile

splash\_damage : f32,

Splash damage of the projectile.

speed : i8,

Speed of the projectile

i8 to be used in calculations with location without conversions.

shooter\_id : usize,

Id of the shooter, required so the projectile can successfully apply any effects from the shooter.

Methods:

simulate\_tick:

Simulates a tick for a projectile

Returns a bool describing whether the projectile should be deleted after this tick or not.

Psuedocode:

FUNCTION simulate\_tick(self, possible\_targest : Vec<SummonedChampion>, friendly\_champions : Vec<SummonedChampion>, dead\_champions : Vec<SummonedChampion>) -> bool:

target\_location = None

IF self.target\_location != None:

target\_location = self.target\_location

ELSE:

FOR champ in possible\_targets:

IF champ.id == self.target\_id:

target\_location = champ.location

BREAK

ENDIF

ENDFOR

IF target\_location == None:

RETURN False //target dead so remove projectile

ENDIF

ENDIF

self.location\_progress[0] += self.speed \* sign(target\_location.x - self.location.x)

self.location\_progress[1] += self.speed \* sign(target\_location.y - self.location.y)

IF ABS(self.location\_progress[0]) >= 10:

self.location.x += sign(location\_progress[0])

ENDIF

IF ABS(self.location\_progress[1]) >= 10:

self.location.y += sign(location\_progress[1])

ENDIF

IF NOT self.location.check\_valid():

RETURN False //out of bounds, removing

ENDIF

FOR possible\_target in possible\_targets:

IF self.location == possible\_target.location:

DEAL\_DAMAGE(self.damage, possible\_target)

RETURN False

ENDIF

ENDFOR

RETURN True

ENDFUNCTION

new:

Creates a new projectile.

class Shield:

Description:

Reduces damage taken by a certain amount.

Attributes:

duration : i16,

Duration of shield in centiseconds

size : f32,

Size of shield/ quantity of damage it can take

blocks\_type : Option<DamageType>,

Type of damage it can block, option as some shields block all types of damage

pop : bool,

Whether the shield pops/ destroys itself after any damage is taken, even if damage was not enough to completely destroy shield

Methods:

update\_shield:

Reduces the duration of the shield by the length of time provided.

handle\_damage:

Takes an input of damage and its damage\_type, if the damage type is the type it blocks, reduces the damage by the size of the shield.

enum StatusType:

Description:

An enum that holds all the possible/ different status types.

class StatusEffect:

Description:

A class describing a status effect afflicting a SummonedChampion, with a duration and status type.

Attributes:

duration : Option<i16>,

Duration of the status effect

Option as some status effects can be indefinite.

applied : bool,

Tracks whether the status has been applied already.

status\_type : StatusType,

The status type of the effect.

is\_negative : bool,

Tracks whether the status effect is negative, if it is, it will be shed when shedding negative effects.

class Board:

Description:

A class containing the layout and state of a board.

Attributes:

p1\_champions : VecDeque<SummonedChampion>,

A vector/ list holding all the SummonedChampion's on player one's team.

Vector as it has to be able to grow and shrink in size as SummonedChampion die etc.

Double ended vector as I have to be able to pop and append elements to both ends of the vector.

Reasoning convered in pseudocode comments.

p2\_champions : VecDeque<SummonedChampion>,

A vector/ list holding all the SummonedChampion's on player two's team.

Vector as it has to be able to grow and shrink in size as SummonedChampion die etc.

Double ended vector as I have to be able to pop and append elements to both ends of the vector.

time\_unit : i8,

Time unit for the board, the length of time simulated in a single tick/ iteration, measured in centiseconds.

i8 to be used in calculations with variables that can be negative without conversions.

ticks\_till\_draw : u32,

The number of ticks until the battle is declared a draw.

u32 as it cannot be negative but has to store very large numbers.

tick\_count : u32,

A count of the number of ticks that has currently surparsed. Required so the board can avoid indefinite battles.

u32 as it has to store very large values, but cannot be negative.

p1\_projectiles : Vec<Projectile>,

A vector/ list of all the projectiles created by player one's summoned champions.

p2\_projectiles : Vec<Projectile>,

A vector/ list of all the projectiles created by player two's summoned champions.

dead\_champs : Vec<SummonedChampion>,

A vector/ list of all the dead champions from both teams, required so projectiles that outlive who shoot them can still apply the correct effects.

Methods:

new:

Creates a new board from a list of PlacedChampion for both players

simulate\_battle:

Simulates a number of ticks on this board.

Psuedocode:

FUNCTION simulate\_battle(self, ticks\_to\_simulate : Option<u32>):

let upper\_limit : u32 = self.ticks\_till\_draw //sets upper limit to ticks till draw

IF ticks\_to\_simulate.is\_some() AND self.tick\_count + ticks\_to\_simulate < self.ticks\_till\_draw:

upper\_limit = self.tick\_count + ticks\_to\_simulate //if there is input ticks to simulate and its lower than ticks till draw, update upper limit

ENDIF

FOR \_ in RANGE(self.tick\_count, upper\_limit): //for tick in simulating range

self.tick\_count += 1

//when calling take\_turn on each SummonedChampion, I have to pass in a vector containing all friendly champions, for healing and pathfinding iteractions.

//the SummonedChampion has to be outside of the vector at this stage, to avoid interacting with itself as though it was a separate champion.

//it also means I can avoid copying the champion, as it doesn't exist in two places, avoiding the expensive calculation.

//By using a double-ended vector, I can pop an element from the front of the list and append it to the back once finished,

//by doing this once for every item in the list, I can access all elements in the vector, keeping them in order all with O(1)

//operations that ensure the program run efficiently and quickly.

FOR \_champ\_count in range(0, self.p1\_champions.length()):

let current\_champ : SummonedChampion = self.p1\_champions.pop\_left()

let alive = current\_champ.take\_turn(self.p1\_champions, self.p2\_champions, self.time\_unit, self.movement\_amount, self.p1\_projectiles) //take turn

IF alive: //if still alive after turn

self.p1\_champions.push\_back(current\_champ) //push back

ELSE:

self.dead\_champions.push(current\_champ) //add to dead champions in case needed by projectile

ENDIF

ENDFOR

FOR \_champ\_count in range(0, self.p2\_champions.length()):

let current\_champ : SummonedChampion = self.p2\_champions.pop\_left()

let alive : bool = current\_champ.take\_turn(self.p2\_champions, self.p1\_champions, self.time\_unit, self.movement\_amount, self.p2\_projectiles) //take turn

IF alive: //if still alive after turn

self.p2\_champions.push\_back(current\_champ) //push back

ELSE:

self.dead\_champions.push(current\_champ) //add to dead champions in case needed by projectile

ENDIF

ENDFOR

FOR projectile in self.p1\_projectiles: //simulate tick for each projectile

projectile.simulate\_tick(self.p2\_champions, self.p1\_champions, self.dead\_champs)

ENDFOR

FOR projectile in self.p2\_projectiles:

projectile.simulate\_tick(self.p1\_champions, self.p2\_champions, self.dead\_champs)

ENDFOR

IF self.p1\_champions.length == 0 OR self.p2\_champions.length: //there is a winner, stop simulating

BREAK

ENDIF

ENDFOR

ENDFUNCTION

There will also be a number of accompanying useful utility functions I will create to assist with the development/ running of the app, helping avoiding repeating redundant code.

Functions:

find\_champion\_index\_from\_id:

Retrieves the index of a champion in a vector, returning None if it is not found.

It is a good idea to create a function for this behavior as it will be very commonly used, as, for example, SummonedChampion objects hold a target id that references another SummonedChampion's id, which will need retrieving from a vector. If the summonedchampion were to hold the index rather than id of its target, then when SummonedChampion's die and are removed, the index would be made invalidated and be focusing the wrong object, which may even lead to an out of bounds index error in some scenarios.

Psuedocode:

find\_champion\_index\_from\_id(champions : Vec<SummonedChampion>, id : usize) -> Option<usize>:

let i : usize = 0 //initialise counting variable i to 0

FOR champ in champions: //iterate through all the champions

IF champ.id == id: //check for match

RETURN Some(i)

ENDIF

i += 1

ENDFOR

RETURN None

ENDFUNCTION

find\_champion\_index\_from\_id\_targetable:

Retrieves the index of a champion in a vector, only returning it if the champion is targetable.

Psuedocode:

find\_champion\_index\_from\_id\_targetable(champions : Vec<SummonedChampion>, id : usize) -> Option<usize>:

let i : usize = 0

FOR champ in champions:

IF champ.id == id:

IF champ.get\_is\_targetable():

RETURN Some(i)

ENDIF

RETURN None //if the champ is not targetable, return None

ENDIF

i += 1

ENDFOR

RETURN None

ENDFUNCTION

-----------------

class Store:

Description:

Class that interfaces with the database

Attributes:

ds : Datastore,

Opened database file

ses : Session,

Database session to query

board : Option<Board>:

Current board being held/ simulated

Option as none may be simulated

last\_board : Option<String>:

ID of the last saved board, can be used to update/ input the outcome of the match

Option as no board may be saved.

Methods:

new:

Returns a new Store object.

setup:

Setups the database, checks to see if there are saved champions/ items in the datastore.

If there are none, it inputs the default champions or items into the database.

insert\_champion:

Inserts the inputted champion into the database

insert\_item:

Inserts the inputted item into the database

fetch\_champions:

Returns a list of all the champions in the database

fetch\_champion\_from\_id:

Returns a champion from the given id

fetch\_items:

Returns a list of all the items in the database

fetch\_item\_from\_id:

Returns an item from the given id

fetch\_champions\_ids:

Returns a list of all the ids of the champions in the database

fetch\_items\_ids:

Returns a list of all the ids of the items in the database

update\_champion:

Updates the stats in the database of the given champion

update\_item:

Updates the stats in the database of the given item

set\_board:

Sets self.board to the given value

replace\_board:

Swaps the stored board with the given board, returning the old one

fetch\_board:

Returns the stored board

store\_board:

Takes two inputs of placed champions and stores them in the database.

update\_outcome:

Updates the outcome of the self.last\_board stored in the database with the given output

fetch\_outcomes:

Returns a list of all the stored outcomes/ battle results in the database

fetch\_outcome\_board:

Fetches the initial board state of the board with the given id.

Database Design:

I will plan out the structure of my database in advance, to avoid mistakes, errors and confusion when programming the database.

Graphical user interface, application

Description automatically generatedAll tables in the database will need a unique id so they can be accessed directly. board\_champions should have a foreign key to its associated board to represent the link between the two and the one to many relationship they have.

Both items and champions should have all the basic/ alterable stats for the two classes so any modifications made in the program by the user can be saved, such as health, ad and attack speed modifier. board\_champions should contain all the information required to reconstruct the board in its initial state, including its star level, all the items it had as well as its location.

Navigation and Communication:

The overall flow of the program and communication between components will be as displayed below. Grey = store/ database, yellow = backend, green = ipc (interprocess communcation, frontend-backend communcation handler), orange = UI/ frontend.

Diagram, table

Description automatically generated

By having a navigation menu from which you can navigate to the rest of the program with, user movement around the program will be simple and easy, without the possibility of getting lost.

By having a separate page to distunguish between creating the board and watching a battle, different UI elements will be able to be present and separate interactions for the user. For instance, on the Board page, clicking on a cell with a unit present will open up a popup near your click with information on the unit, however, on the battle page, as the left side of the UI is not taken up with the drag and drop, more space can be given to display this information, and it can instead be placed on the left side where the drag and drop was, as shown in the visualisation diagrams earlier.

By having common patterns such as fetching unit stats, complexity can be reduced by having function calls performing specific duties to allow the code to be cleaner. By having an IPC communicating between the two sides of the program, complexity can be reduced on both ends by avoiding having to anticipate/ work out the features that each program has, instead leaving it to a designed component, resulting in encapsulation. Encapsulation is a tenant of object orientated programming, where the code and explanation behind features is obscured to certain parts of the program to simplify behaviour. For instance, the IPC does not need to know how the store retrieves a previous board/ result, all it needs to know is that it has to pass in an id and should expect to receive an error or board in return. This simplifies programs and helps declutter.

By storing the initial state of the board before a battle, any unintended and unlikely crashes of the program can be mitigated, allowing the user to retrieve/ check what the board was that led to the issue, allowing them to rerun the battle or avoid it in the future. It also helps in case the user forgets to save the outcome of the battle, allowing them to resimulate it.

Test data to be used during iterative development.

Data to be used in post development phase.

What each prototype will achieve.