**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.

This solution is very suitable for the stakeholder and appropriate for their needs as it allows the Suket to easily create, simulate and run battles, giving him invaluable experience and key learning scenarios that cannot be created anywhere else. This is an innovative solution that does not exist anywhere else on the market that is custom built perfect to his requirements, perfectly fulfilling his requirements.

## 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.

I aim to incorporate these guides ability to be crucial starting/ jumping off points for new players, giving them basic game knowledge allowing them to avoid the first learning stages of the game.

***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.

I aim to include these programs ability to provide the user with the most effective current strategies available, by allowing user to view a catalogue of their previous battles so they can view what strategies are good/ tested at the time and also to provide information while ingame, by allowing the program to be run and used alongside 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.

I aim to incorporate coaching’s ability to be specific and useful to your scenario, as you will be able to recreate the situation in your program. I also will include in my program the ability to be useful for all skill levels, for new players to get a basic idea of the combat and for professional players to quickly iterate and test numerous teams and matchups. I aim to avoid making it inaccessible to most players by providing my service/ program for free and to be accessible by being not intensive to run.

## 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

Finally, pattern recognition will be key in my project, allowing me to encapsulate behaviour of similar objects into classes. For example, rather than creating a unique variable for each Champion, I can capture all the required data and behaviour into a class, allowing me to code key methods onto the class, avoid redundant and repeated code and ensure that my final code is readable and understandable. Pattern recognition will be able to be used on lots of areas in my project, such as Champions, SummonedChampions, status effects, boards and more.

I also will utilise polymorphism in my project, giving common attributes/ traits/ methods to classes to allow objects to interact and interface with each other in similar fashions. For instance, implementing display methods on my objects, to allow them to be printed to a console by calling a method common across all classes. Utilising polymorphism allows the program to avoid redundant code distinguishing between similar objects to perform a well-defined/ common action.

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:

* 512MB (spare) RAM.
  + This is enough RAM to safely allow the program to run without running into memory issues, which may cause unforeseen crashes or hinder/ slow the program with the use of virtual memory. It is a relatively small amount that most users should be able to spare, especially if running alongside Team-Fight Tactics which only requires 1 gigabyte.
* 1GB (available) local storage.
  + This should be enough spare space to contain not only the final, compiled program, but also hold the database and give enough room for the database to grow to hold many battles/ outcomes without the user worrying about running out of space.
* 1.5 Giga-hertz processor.
  + This is a fairly, low end processor for modern standards, that should be more than sufficient to run the program and allow the program to simulate the battle in an acceptable period of time and allow the user to interact with the user interface quickly and without lag.
* Capable of running Windows 8 (or newer), Linux or MacOS.
  + These are the main operating systems available on modern computers and so I will add compatibility for these to improve accessibility and allow the vast majority of users to access the program. Adding support for other, less known computes would give diminishing returns (with so few people using them) with a very large time investment as I essentially would have to rewrite all my code/ porting software to allow for it to be ran on them.
* A GPU on par or more powerful than:
  + NVidia: GeForce 560
  + AMD: Radeon HD 6950
  + Intel: Intel UHD
  + These are very old/ low end GPUs, that should be sufficient to simulate the user interface of the program. If the user can use web browsers such as Google or likewise with their GPU, then it will be enough to run this program easily.
* These requirements are also all less than or equal to the requirements to run Team-Fight Tactics, so if the user cannot run this program, they cannot run Team-Fight Tactics (and so do not really require the program).

The software required for the program is just one of the Windows, Linux or MacOS operating systems, as I will compile down my code into an executable file, no additional/ external software will be required to run my final program, improving the accessibility of the program and improving ease of use.

## 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 (i.e. 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 frontend. 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, using the computational method of pattern recognition, 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 window 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.
  + When selecting from the list of past results to display, the user will be able to see what units were on each team, so they can select the correct one.
  + When the user selects a unit/ item to change, the stat input boxes will autofill.
* 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.
  + You can queue up multiple, different battles at once to do one after another.

## 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.

Grid Testing:

As I will touch upon later, I will implement an algorithm for detecting whether a location/ position is valid with the pseudocode:

FUNCTION valid\_location(location : Location) -> bool:

RETURN location.x >= 0

AND location.x < 10

AND location.y >= 0

AND location.y < 8

AND 2 - (location.y / 2) < location.x

AND 10 - (location.y / 2) > location.x

ENDFUNCTION

Which I will test with the table below.

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Num | X: | Y: | Valid Pos: | x >= 0 | x < 10 | y >= 0 | y < 8 | 2 - (y / 2) < x | 10 – (y / 2) > x | Output |
| 1 | 3 | 0 | Yes | T | T | T | T | T | T | True |
| 2 | 2 | 0 | No | T | T | T | T | F | T | False |
| 3 | 7 | 7 | Yes | T | T | T | T | T | T | True |
| 4 | 7 | 8 | No | T | T | T | F | T | F | False |
| 5 | 4 | 3 | Yes | T | T | T | T | T | T | True |
| 6 | 0 | 7 | Yes | T | T | T | T | T | T | True |

The tested positions in the grid are shown below:



As you can see the algorithm works accurately to discern whether positions are in the grid or not, ensuring that pathfinding is accurate and projectiles or champions cannot go outside the grid.

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.

Justifications for decisions can be seen in **bold.**

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

AND self.x < 10

AND self.y >= 0

AND self.y < 8

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

AND 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 multiplier.

**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 AND 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.

**As covered in the take\_turn function for SummonedChampion’s, I decided that the pathfinding for the units should be very simplistic, simply choosing the available cell that minimises the distance between it and its target. My reasoning behind this is that, rather than implementing a processing expensive algorithm like A\* or dijkstra’s, as the program needs to run very efficiently and quickly, that I would go for simplicity instead, avoiding calculating the most optimal path for each champion, potentially every 10 ticks, which could severely slow the program.**

## 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.

## Validation:

There will need to be validation of the classes and any data received, especially from the frontend where the user may mistakenly or maliciously enter invalid inputs.

There is some validation that is required to be built into the classes/ used on the backend, an example of this is Location’s check\_valid method, that checks if self/ the location is a valid location in the 8 by 8 hexagon grid. This will be used in the simulation, to check that if a pathfinding cell is inside the grid, i.e. the champion can move there, or to see if a projectile has gone outside the grid and needs to be removed.

The bulk of the validation will be on the frontend, however, to ensure that the user does not enter invalid data maliciously or accidentally.

There will be inputs for:

* Changing unit stats:
  + Changing the attack damage, health, attack speed, armour, magic resist, mana cost, range and starting mana of the champion.
  + We will have to ensure that all the inputs are valid, so all with values above 0 and not unreasonable large, so as not to fit in their stored datatype. For instance, starting mana should not reasonably have a value of 9999999, and so will be disregarded.
  + We can ensure validation by having a “submit” button that when clicked submits the values to the ipc to be changed. Before, however, it should go through each value, ensuring it is in the valid range. If the values are not in the range, the data/ request isn’t sent to the ipc and backend, and an error is displayed on the frontend.
* Changing item stats:
  + Changing the attack damage, ability power, armour, attack speed, current mana, crit rate, crit damage, dodge chance, health, magic resist, omnivamp and range of the item.
  + We will have to ensure that all the inputs are valid, so all with values above (or equal to) 0 and not unreasonable large, so as not to fit in their stored datatype. For instance, dodge chance should not be above 100 and we should not have a negative attack damage item.
  + We can ensure validation by having a “submit” button that when clicked submits the values to the ipc to be changed. Before, however, it should go through each value, ensuring it is in the valid range. If the values are not in the range, the data/ request isn’t sent to the ipc and backend, and an error is displayed on the frontend.
* Starting a battle:
  + There will be inputs to control the time unit and the number of ticks until a draw of the board.
  + Neither of these values should be negative, as we cannot go backwards in time or instantly draw, they should also both be whole numbers, as we cannot simulate half a tick. We can ensure they are both whole numbers by not allowing the user to input a . into the box, signifying a decimal place.
  + The number of ticks until a draw should be above a certain value (10,000), to avoid premature draws, it should be below a much larger value (such as one million), to ensure that if the user wants, they can get a very accurate/ drawn out battle, ensuring it would, if left, go on forever.
  + The time unit should be below 1000, as with a time unit that large or larger, there will be many inaccuracies and simulation issues, as it is not intended to hold/ simulate a time unit so large at once. Trying to simulate 10 seconds at once will result in issues with movement, damage and status effects.
  + We can ensure validation by having a “submit” button that when clicked submits the values to the ipc to be changed. Before, however, it should go through each value, ensuring it is in the valid range. If the values are not in the range, the data/ request isn’t sent to the ipc and backend, and an error is displayed on the frontend.
* Playing/ stepping through a battle:
  + There will be two inputs, one to control the speed at which you play through a simulation and another one that controls how far you jump forward when you press the jump forward button.
  + Neither of these values should be negative, as we cannot move backwards in time (one of the limitations mentioned).
  + Other than that, both of these values can be any size, as if they are larger than the time till draw, we will just simulate until that time, so there will be no issues with that.
  + We can ensure validation by checking when the button is pressed to play the battle or jump forward, that these values are valid.
* With all of these inputs, no alphanumeric characters (such as ABC etc), should be inputted, as all these inputs are numeric. We can avoid this by not allowing the user to input alphanumeric characters into the input box, if they press the keys they simply do not show up in the input, this can be achieved by specifying the input box as a “number” which will result in JavaScript not allowing certain characters into the input box.

## 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.

## Prototyping

I will utilise an iterative development strategy to be able to adapt to client requirements as the project progresses and test prototypes as I advance further into the project, allowing me to ensure that I am fufilling my client and success criteria. This methodology also allows me to receive feedback as I work on the project and adjust accordingly, helping me better fulfill mine and my client’s aim for the project.

I will utilise 3 prototypes as that is the required amount to successfully see and adapt to progress and changes. For each prototype, I will have a number of criteria to fulfill and test data to be used during the iterative development phase to ensure the prototype is up to standard.

While prototyping and testing for accuracy in simulations, I will have a variety of boards to test with a already known outcome to compare to the outcome of my simulations. All units will be placed on their teams side in the bottom left or top right corner to provide consistency. All boards will be tested with a time unit of 10 and ticks till draw of 10000. For prototype 1 and 2 I will create a number of boards to test with a recorded outcome, which I will display belong along with the associated prototype.

These will give a measurable way to test the accuracy of the board, ensuring I meet the criteria set out for the prototype and program overall. Small tweaks to functionality and programming may be done if the accuracy is not sufficient for the prototype to ensure I meet the standard set.

### Prototype 1:

For prototype one, I will aim to have a program that can simulate some basic parts of TFT, such as simple combat between a few different champions on a board. The SummonedChampion’s implemented will be a simplified version of themselves for initial development and testing. The code will not crash too frequently and be accurate to an acceptable degree. The user will be able to see the outcome of the battle in a CLI (command line interface). This will be an early development stage, with the only way of creating boards to be alter the code, but will be a valuable demonstration to my client.

Requirements:

* Basic implementation of Champion, PlacedChampion, SummonedChampion and Board classes.
* Ability to simulate battles:
  + Build battles/ boards manually in code.
  + Output results to a command line interface.
  + With some accuracy:
    - Provided boards/ battle outcomes correct a minimum of 50% of the time.
  + With less than a 10% crash chance per battle:
    - I will simulate 1000 random boards and count the number of crashes.

Testing for prototype one will involve a robustness test, where the program will run a large number of random boards and record the number of crashes/ closures of the program and an accuracy test, where the program will simulate a number of boards multiple times and record its accuracy in simulating that board against the recorded, correct result. I will get the recorded, correct result by creating the same board (with the associated champions) in the actual TFT game and measuring the result.

1S = one star, 2S = two star, 3S = three star.

ID of champion to the associated champion:

0 : Lulu

1 : Aatrox

2 : Ezreal

3 : Ziggs

So 1S0 = one star Lulu, 3S1 = 3 star aatrox.

The boards tested will

|  |  |  |
| --- | --- | --- |
| Player 1 Champions | Player 2 Champions | Recorded Winner |
| 3S1 | 1S0 | 3S1 |
| 1S0, 1S0 | 2S1 | 1S0, 1S0 |
| 1S2 | 1S3 | 1S3 |
| 1S0, 1S1, 1S2, 1S3 | 3S3, 3S1 | 1S0, 1S1, 1S2, 1S3 |
| 2S2 | 3S3 | 3S3 |
| 3S1, 3S1 | 3S3, 3S3, 3S1 | 3S3, 3S3, 3S1 |

### Prototype 2:

For the second prototype, I aim to have fully implemeneted the simulator behind the program. While still utilising a CLI, I aim for it to be able to accurately and efficiently simulate battles between a variety of boards, with items and selected champions fully implemented. The program should rarely if ever crash across a variety of boards and should run at an acceptable speed. This stage of the program is still poor in terms of usability, with boards still only being able to be created in the code, but will be accurate and fully fledged in terms of simulation.

Requirements:

* Full implementation of all simulator-related classes, including Champion, Item, SummonedChampion etc.
* Ability to simulate battles:
  + Build battles/ boards manually in code.
  + Output results to a command line interface.
  + With a high degree of accuracy:
    - Provided boards/ battle outcomes correct a minimum of 95% of the time.
  + With less than a 0.5% crash chance per battle

Testing in prototype 2 will once again involve a robustness test, where random boards will be simulated numerous times to calculate a crash rate.

This has the same shorthand for champions as the previous table (3S1 = 3 star aatrox), but also now a champion may have [] on the end with item ids in them, specifying those items are placed on the champion.

|  |  |  |
| --- | --- | --- |
| Player 1 Champions | Player 2 Champions | Recorded Outcome |
| 3S1[3, 12, 55] | 1S0[16, 27, 36] | Player 1 Winner |
| 1S0[33, 67], 1S0[25] | 2S1 | Player 1 Winner |
| 1S2 | 1S3 | Player 2 Winner |
| 1S0, 1S1[48], 1S2, 1S3[44] | 3S3, 3S1[67] | Player 2 Winner |
| 2S2 | 3S3 | Player 2 Winner |
| 3S1, 3S1[88] | 3S3, 3S3[56], 3S1 | Player 2 Winner |

### Prototype 3:

Finally, this prototype should build on top of the previous prototypes, still being able to accurate simulate battles, however, this prototype should feature the user interface and database capabilities, allowing the user to interact with the program through a graphical user interface as well as save/ store battles and unit changes. This prototype will allow the client easy control over the simulator created in previous prototypes, allowing the client to fully utilise the program.

Requirements:

* Full implementation of simulation from previous prototypes
* Implementation of a graphical user interface:
  + Allow the user full control of the program, creating battles, viewing the outcome of the battle, saving the result along with altering and saving changes to unit and item stats.
    - Have a navigation page where the user can click links to move around the program.
    - Have a board page where the user can drag and drop items and units onto a board, and start a battle with a button.
    - Have a battle page where the user can view the battle they created and get full information on the battle, for example clicking a unit and seeing its health/ attack damage etc.
    - Have a unit/ item stat page where the user can retrieve unit and item stats, alter them, then save them back to the database.
    - Have a previous results page where the user can view previous results and the boards that led to that outcome.
* Implementation of a database:
  + To store units, items and result.
  + Provide a way for the database to retrieve and alter entries in database.

Here are some testing points for prototype 3:

|  |  |
| --- | --- |
| **Test:** | Expected Outcome: |
| **Program** |  |
| Should be resizable | The program window should be resizable within reason like you can with any other program. |
| **Non-Navigation Pages:** |  |
| Visible link to navigation page on every page. | There should be a link to the navigation page on every page, clicking it should take you to the navigation page. |
| **Navigation Page:** |  |
| Visible link to every page. | There should be a link to each other page in the program, clicking on it should take you to said page. |
| **Board Page:** |  |
| On load | Should fetch a list of champions and items and create a drag and drop component for each of them, displaying them on a list on the left of the screen. |
| On load : create grid | On load, the page should create an 8 by 8 interactive hexagon grid. |
| Drag and drop a champion from the left. | Dragging a champion from the left side of the page and dropping it on one of the hexagons should place the champion in that hex. |
| Drag and drop a champion from the left onto occupied cell | Overwrite the champion in that cell with the new one. |
| Drag and drop an item from the left onto a hexagon with a champion | Update the item array with the new item that you added |
| Drag and drop an item from the left onto a hexagon with a champion with 3 items. | Replace the first item in the items array with the new item. |
| Drag and drop an item onto a hexagon without a champion. | No action |
| Clicking on an occupied hexagon | A tooltip should pop up. If the tooltip is already visible elsewhere, it should move and update itself to the new hexagon. |
| The tooltip | The tooltip should have information on the champion such as its type and any items it has. |
| Tooltip buttons | The tooltip should have buttons that allow the user to change the team of the unit, change its star level or delete it. |
| Clicking on an unoccupied hexagon | Should have no effect or hide the currently displayed tooltip. |
| Inputs | On the right side of the screen, there should be inputs allowing the user to specify the time unit and ticks till draw of the board. |
| Start battle button | On the right side of the screen, there should be a start battle button.  If all inputs are valid the page should send the board to the backend to be stored/ simulated.  If any of the inputs are blank or invalid (negative time unit, non-numerical input etc), then the page should show an error and not submit. |
| **Battle Page** |  |
| On load | Should fetch the stored board and display it on a grid. |
| On press occupied cell | Should display the unit details on the left side of the page |
| On press play | Every second should ask the backend to simulate the set number of ticks specificed by the associated input and then retrieve and display the new board |
| On press simulate X ticks | Should simulate X ticks and retrieve/ display new board. |
| **Change Unit and Item Stats** |  |
| On load | On load the page should fetch a list of all the units and items. |
| Input boxes | There should be a list of inputs lining up to all the adjustable stats of a unit or item |
| ID Dropdown | There should be a dropdown list of all the ids of units or items. Upon selecting one of them, it should autofill the inputs with the saved stats of the unit or item. |
| Submit unit/ item | The page should check the input boxes have valid data in them (in correct range, not empty etc), if so it should submit the unit/ item to the IPC to be altered. |
| **Previous Results Page** |  |
| On load | The page should fetch a list of past results and display buttons for each of them on the right side of the screen |
| Press button to load previous result | A board should be created with the same PlacedChampions as on the board, showing the initial state of the board. |
| On press occupied hexagon | Should load unit details on the left side of the screen. |

## Post-development Testing:

Once the application is finished, I will do post-development testing to ensure that my project fits my initial success criteria I laid out in the analysis stage. The tests I will perform to ensure that my program fits the initial success criteria are:

|  |  |
| --- | --- |
| Test | Success Criteria |
| The user can create a board through only drag and drop components and simple click mechanics | 1 |
| The simulation/ program will be efficient and quick, being able to simulate a complex board (as defined earlier as board which contains more than 3 units on each team, each with at least 1 item) in under 2 seconds. | 3 |
| The user will be able to change unit and item stats easily through the UI. | 4 |
| The user will be able to view previous boards and their result | 5 |
| The program will still meet or exceed all the criteria set out in previous prototypes. |  |

I will also test the usability features that I listed in the analysis stage which were with the following tests:

|  |  |
| --- | --- |
| **Test/ Requirement** | **Usability Criteria** |
| Text will be distinctly defined, being a clearly different colour to its background | Clear UI |
| Hexagons on grid will be clearly defined with outlines | Clear UI |
| Buttons will have clear outlines | Clear UI |
| The program will be resizable | Clear UI |
| Units and items will be draggable | Ease of Use |
| Clickable hexagon cells to show unit details | Ease of Use |
| Modifiable speed of simulation playback | Easy of Use |
| Checked input fields to prevent invalid inputs | Robust and Efficient Program |
| Low number of crashes from simulation (less than 0.5%) | Robust and Efficient Program |
| Efficient program to avoid anoyance at lag.  (Complex battles in less than 2 seconds) | Robust and Efficient Program |

# Developing the Coded Solution:

## Prototype 1:

To begin prototype 1, I import some key types and functions that I will require later on in the program:

use rand::{Rng};

use std::collections::VecDeque;

The rng import will provide functionality required to generate random numbers for critical strikes and dodging, VecDeque provides the double ended vector functionality required later.

Next I defined the champion struct:

///Champion (struct):<br />.

///Stores the basic information surrounding a champion<br />

struct Champion

{

    ///Champion ID<br />

    ///same as index in CHAMPIONS

    id : u8,

    ///Healthpoints for each Star Level

    hp : f32;

    ///Starting mana

    sm : u16,

    ///Ability Mana Cost

    mc : u16,

    ///Base Armor Value

    ar : f32,

    ///Base Magic Resist Value

    mr : f32,

    ///Auto Attack Damage for each Star Level

    ad : f32,

    ///Attack Speed in Attacks per Second

    aS : f32,

    ///Auto attack range

    ra : u8,

}

This allows me to create some basic champions with their respective base stats, allowing me to mimic champions in the real teamfight tactics.

Following this I defined the CHAMPIONs constant:

const CHAMPIONS : [Champion ; 4] = [Champion{id : 0, hp : 650.0, sm : 70, mc : 140, ar : 0.25, mr : 0.25, ad : 40.0, aS : 0.6, ra : 2}, //Support

                                    Champion{id : 1, hp : 800.0, sm : 50, mc : 100, ar : 0.45, mr : 0.45, ad : 75.0, aS : 0.7, ra : 1}, //Bruiser

                                    Champion{id : 2, hp : 700.0, sm : 35, mc : 100, ar : 0.25, mr : 0.25, ad : 65.0, aS : 0.7, ra : 3}, //AD Ranged

                                     Champion{id : 2, hp : 700.0, sm : 35, mc : 150, ar : 0.25, mr : 0.25, ad : 50.0, aS : 0.6, ra : 3} //AP Ranged

                                    ];

A four long constant array of Champion. As I know before runtime what I want my Champion’s to be, I can define it as a constant, letting the compiler hardcode it so accessing it is faster. My four long array of champions includes all the champions that I wish to implement in my version of TFT, representing Lulu (support), Aatrox (bruiser), Ezreal (Ad range) and Ziggs (ap ranged), which gives a nice range of abilities and damage types to ensure my simulation has a wide coverage of champions.

I move on to create the PlacedChampion struct, as described in the design phase:

struct PlacedChampion

{

    ///id given at instantiation

    id : usize,

    ///star level of champion

    star : usize,

    ///items

    items : [u8 ; 3],

    ///location on board

    location : [i8; 2]

}

As this is an early iteration of my program, I decided against creating the location class so early, to focus more on other more important aspects of the program. All other aspects are the same as described in design, resulting in a simplified class that can hold enough information to represent the champion on a board.

To finish up the champion class group, I define SummonedChampion:

///Struct for champion placed on board in a battle

struct SummonedChampion

{

    ///array of p, q coordinates, r can be calculated with r = -p - q

    location : [i8 ; 2],

    ///progress of movement before new square, goes up to 10 then moves

    movementProgress : [i8 ; 2],

    ///health

    health : f32,

    ///current mana

    cm : u16,

    ///dodge chance in %

    dc : u8,

    ///crit rate in %

    cr : u8,

    ///crit damage

    critD : f32,

    ///ability mana cost

    mc : u16,

    ///armor

    ar : f32,

    ///magic resist

    mr : f32,

    ///attack damage

    ad : f32,

    ///attacks per second/ attack speed

    aS : f32,

    ///auto attack range

    ra : u8,

    ///id

    id : usize,

    ///cooldown before target chance

    targetCountDown : i8,

    ///cooldown before auto attacking again

    autoAttackDelay : i16,

    ///attack speed modifier from items and effects

    attackSpeedModifier : f32,

    ///id of target

    target : usize,

    ///pathfinding target cell

    targetCells : [i8 ; 2],

    items : [u8 ; 3],

    ///ability power

    ap : f32,

    ///vec of status effects

    se : Vec<StatusEffect>,

    ///generate mana delay (can't generate mana 1 secomnd after casting ability)

    gMD : i16,

    ///star level

    starLevel : usize,

    ///incoming DMG modifier

    incomingDMGModifier : f32,

    ///starting HP

    initialHP : f32,

    ///can be targeted or not

    targetable : bool,

    ///needs to shed negative status effects

    shed : u8,

    ///vec of all shields

    shields : Vec<Shield>,

    ///whether zapped from ionic spark

    zap : bool,

    ///whether zenith banished

    banish : bool,

    ///titan's resolve stacks

    titansResolveStack : u8,

    ///omnivamp (% of healing from damage done)

    omnivamp : f32,

}

This holds all the attributes that will be required when simulating combat, such as targetable which is a bool for whether it can be targeted or if the champ is hidden by some effect like edge of night, or target which holds the id of the target champ.

This prototype can simulate matches through the board class defined below:

struct Board

{

    ///Vec of player 1's champs

    p1Champions : VecDeque<SummonedChampion>,

    ///Vec of player 2's champs

    p2Champions : VecDeque<SummonedChampion>,

    ///Time unit for board in centiseconds (1/100 of a second)

    timeUnit : i8,

    ///movement amount per tick, is calculated by const / time unit

    movementAmount : i8,

}

To simulate matches, first create a new board:

impl Board

{

    fn new(p1PlacedChamps : &VecDeque<PlacedChampion>, p2PlacedChamps : &VecDeque<PlacedChampion>, timeUnit : i8) -> Board

    {

        let mut p1Champions = VecDeque::new();

        let mut p2Champions = VecDeque::new();

        for (i, p1Champion) in p1PlacedChamps.iter().enumerate()//(!O) converts placed champions to summoned champions

        {

            p1Champions.push\_back(SummonedChampion::new(&p1Champion, i));//converts into summoned champ

        }

        for (i, p2Champion) in p2PlacedChamps.iter().enumerate()

        {

            p2Champions.push\_back(SummonedChampion::new(&p2Champion, i));//converts into summoned champ

        }

        Board{p1Champions : p1Champions,

              p2Champions : p2Champions,

              timeUnit : timeUnit,

              movementAmount : 10 \* timeUnit, //(!O)

            }//creates new board

    }

}

The new function takes in the two opposing players PlacedChampions as a reference, as hard-copying the vector would be processor expensive and unnecessary, as well as a time unit. This means that the board can simulate with different time units, opting for more precision with a low time unit or a faster runtime with a high one (where time unit is the length of time simulated in a tick). It creates an empty VecDeque, to hold each players SummonedChampion. Then, for each list of PlacedChampion the board has been given, it turns it into an iterable to get access to the enumerate method, then converting the PlacedChampion to a SummonedChampion and giving it a unique (for its team) id correlating to its index, before returning the new Board object. It also features the proportional relationship between the time unit and the movement amount, where a higher time unit results in a higher movement amount (distance moved in a single tick).

The new method for SummonedChampion is:

impl SummonedChampion

{

    ///converts PlacedChampion into SummonChampion

    fn new(placedChampion : &PlacedChampion, id : usize) -> SummonedChampion

    {

        let ofChampion = &CHAMPIONS[placedChampion.id];//get champ info

        SummonedChampion { location: [placedChampion.location[0], placedChampion.location[1]], //create summoned champ with all details

                           movementProgress : [0, 0],

                           health: ofChampion.hp,

                           initialHP : 0.0,

                           cm: ofChampion.sm, //update current mana to starting mana

                           dc: 0,

                           cr : 25,

                           critD : 0.3,

                           mc: ofChampion.mc,

                           ar: ofChampion.ar,

                           mr: ofChampion.mr,

                           ad: ofChampion.ad,

                           aS: ofChampion.aS,

                           ra: ofChampion.ra \* 2,//because distanceBetweenPoints returns value twice as large

                           id : id,

                           targetCountDown : 0,

                           autoAttackDelay : 0,

                           attackSpeedModifier : 1.0,

                           target : 255,

                           targetCells : [-1, -1], //(!O)

                           aID: ofChampion.aID,

                           items: placedChampion.items,

                           ap : 1.0,

                           se : Vec::new(),

                           gMD : 0,

                           starLevel : placedChampion.star,

                           incomingDMGModifier : 1.0,

                           targetable : true,

                           shed : 0,

                           shields : Vec::new(),

                           zap : false,

                           banish : false

                           titansResolveStack : 0,

                           omnivamp : 0.0,

                        }

    }

}

It fetches a reference to the associated champion from the CHAMPIONS constant and sets the values from that champion to the associated fields, defaulting the other values. This is all the functionality required to create a board object in prototype 1, so we can now move onto simulating a battle.

impl Board

{

fn StartBattle(mut self : Board) -> i8

    {

I first define the method, taking in a mutable board and returning an i8 (the winner of the battle).

for i in 0..self.p1Champions.len()//(!O), (!D) slam item mid round?

        {

            for item in self.p1Champions[i].items//gives item effects

            {

                GiveItemEffect(item, &mut self.p1Champions, &mut self.p2Champions, i);

            }

        }

        for i in 0..self.p2Champions.len()//repeat but for p2 Champions

        {

            for item in self.p2Champions[i].items

            {

                GiveItemEffect(item, &mut self.p2Champions, &mut self.p1Champions, i);

            }

        }

Then, all of the champions in both of the teams, I give the associated item effects. Because item effects can not only give buffs to the champion the item was placed on, but also the team/ enemy team (such as Zeke’s Herald id 13 which gives an attack speed buff to adjacent teammates), GiveItemEffect needs to take in not only the champion which has the item, but also its teammates and its opponents. For this reason, I cannot directly iterate through self.p1Champions/ self.p2Champions. The justification why it would not work is as follows, I need a mutable reference to self.p1Champions in the give item effect function, to allow the board to alter and give buffs to said champions, if I were to directly iterate through self.p1Champions, I would need a second mutable reference to the variable, which breaks Rust’s borrow checker rules which states that only one mutable reference can exist at at time. I can, however, directly iterate through self.p1Champions[i].items, as the list contains u8 values, which due to their simple nature can be inexpensively copied, avoiding having to make references of it. To avoid the double mutable reference issue, I simply iterate through the indexes in the variable, and giving the index/ current champion of which items are being applied to the function. The give item effect function is defined as:

///GiveItemEffect : (func)<br />

///Gives an item effect to a champion<br />

fn GiveItemEffect(item : u8, friendlyChampions : &mut VecDeque<SummonedChampion>, enemyChampions : &mut VecDeque<SummonedChampion>, selfIndex : usize)

{

    match item

    {

        0 => (),

        1  => friendlyChampions[selfIndex].ad += 10.0, //BF Sword

        2  => friendlyChampions[selfIndex].ap += 0.1, //Needlessly Large Rod

        3 => friendlyChampions[selfIndex].health += 150.0, //Giants Belt

        4 => friendlyChampions[selfIndex].ar += 0.2, //Chain Vest

        5 => friendlyChampions[selfIndex].mr += 0.2,//Negatron Cloak

        6 => friendlyChampions[selfIndex].attackSpeedModifier \*= 1.1,//Recurve Bow

        7 => {friendlyChampions[selfIndex].cr += 5; friendlyChampions[selfIndex].dc += 10},//Sparring Glove

        8 => friendlyChampions[selfIndex].cm += 15,//Tear of the Goddess

        11 => friendlyChampions[selfIndex].ad += [15.0, 30.0, 45.0][friendlyChampions[selfIndex].starLevel],

        12 => {friendlyChampions[selfIndex].ad += 10.0; friendlyChampions[selfIndex].ap += 0.1},

        13 => {friendlyChampions[selfIndex].ad += 10.0; friendlyChampions[selfIndex].health += 150.0;

              let thisLocation = friendlyChampions[selfIndex].location;

              for friendlyChamp in friendlyChampions

              {

                if friendlyChamp.location[1] == thisLocation[1] && DistanceBetweenPoints(friendlyChamp.location, thisLocation) < 3//checks units are adjacent and on same row

                {

                    friendlyChamp.attackSpeedModifier \*= 1.3;//increases attack speed

                }

              }

              },

        14 => {friendlyChampions[selfIndex].ad += 10.0; friendlyChampions[selfIndex].ar += 0.2;

               friendlyChampions[selfIndex].se.push(StatusEffect { duration: Some(0), statusType: StatusType::EdgeOfNight(), ..Default::default()})},//gives edge of night buff

        15 => {friendlyChampions[selfIndex].ad += 10.0; friendlyChampions[selfIndex].mr += 0.2;

               friendlyChampions[selfIndex].se.push(StatusEffect { duration: Some(0), statusType: StatusType::Bloodthirster(), ..Default::default()});//gives bloodthirster buff

               friendlyChampions[selfIndex].omnivamp += 0.25;

            },

        16 => {friendlyChampions[selfIndex].ad += 10.0; friendlyChampions[selfIndex].attackSpeedModifier \*= 0.1},//

        17 => {friendlyChampions[selfIndex].ad += 10.0; friendlyChampions[selfIndex].cr += 225; friendlyChampions[selfIndex].critD += 0.1},//(!D)?

        18 => {friendlyChampions[selfIndex].ad += 10.0; friendlyChampions[selfIndex].cm += 15},//

        19 => {friendlyChampions[selfIndex].ad += 10.0;},//(!U)

        22 => {friendlyChampions[selfIndex].ap += 0.75},

        23 => {friendlyChampions[selfIndex].ap += 0.40; friendlyChampions[selfIndex].health += 150.0}//

        24 => {friendlyChampions[selfIndex].ap += 0.1; friendlyChampions[selfIndex].ar += 0.2;//Gives locket shield

                let shieldAmount = [300.0, 350.0, 400.0][friendlyChampions[selfIndex].starLevel];

                let thisLocation = friendlyChampions[selfIndex].location;

                for friendlyChamp in friendlyChampions//iterates through friendly champs

                {

                    if friendlyChamp.location[1] == thisLocation[1] && DistanceBetweenPoints(friendlyChamp.location, thisLocation) < 5 //(!D) gives shield to those within 2 cells and same row

                    {

                        friendlyChamp.shields.push(Shield{duration : 1500, size : shieldAmount, ..Default::default()});//gives shield

                    }

                }

        },

        25 => {friendlyChampions[selfIndex].ap += 0.1; friendlyChampions[selfIndex].mr += 0.2;},//

        26 => {friendlyChampions[selfIndex].ap += 0.1; friendlyChampions[selfIndex].attackSpeedModifier \*= 0.1},//

        27 => {friendlyChampions[selfIndex].ap += 0.5; friendlyChampions[selfIndex].cr += 15; friendlyChampions[selfIndex].critD += 0.4}// //(!D) does bonus ability damage include from components? //

        28 => {friendlyChampions[selfIndex].ap += 0.1; friendlyChampions[selfIndex].cm += 15; friendlyChampions[selfIndex].se.push(StatusEffect { duration: Some(500), statusType: StatusType::ArchangelStaff(0.2), ..Default::default() })}

        29 => {friendlyChampions[selfIndex].ap += 0.1; },//add next trait

        \_ => println!("Unimplemented Item"),

    }

}

The function makes use of a match statement, to match an item id to its associated effect. Due to using a simple data type as the match value (u8) and the match branches are consecutive and incrementing, the compiler can use a jump-table to optimise finding the correct branch down to a constant time operation, justifying the use of a match statement in this scenario for efficiency and clarity.

Most of the code is simply indexing friendlyChampions at the index of the champion that has the item placed on it and increasing a stat, which I will not explain. There are some, such as id 28, where a status effect is given to the champion to help replicate the effect of the item.

Then finally, 13 and 24 give bonuses to allied champions placed on the same row. For these, I iterate through friendly champions, checking if the y value of the location matches (to correspond to same row) and then checking that the champion is in range of the effect. As the distance returned by the distance between points function is double the actual distance (a distance of 1 hex has a “distance” of 2, as two of three of the x, y and z values will have changed), the distance that I check for is twice as large. For example, for a distance of 2 cells for id 24, I check that the distance between point is lower than 5, which corresponds to a distance of 2.5 hexes.

Both the effects also apply both to allies and the champion the item is on, so despite the fact that friendlychampions includes the champion the item is on, there is no need to check what champion the effect is being applied to.

The full list of item effects can be viewed here:

0 : Null

1  : B.F Sword (+10 Attack Damage)

2  : Needlessly Large Rod (+10 Ability Power)

3  : Giants Belt (+150 health)

4  : Chain Vest (+20 Armor)

5  : Negatron Cloak (+20 Magic Resist)

6  : Recurve Bow (+10% Attack Speed)

7  : \*Sparring Gloves\* (+5% Crit Chance, +10% Dodge Chance)

8  : Tear of the Goddess (+15 Mana)

9  : Spatula

11 : Deathblade (+40, +70, +100 Attack Damage - Star Level Dependent)

12 : \*Hextech Gunblade\* (Dealing Magic and True Damage heals the owner and lowest health ally for 25% of the damage)

13 : Zekes Herald (Grants 30% bonus attack speed to the holder and 2 adjacent allies in same row)

14 : Edge of Night (At 50% health - once per combat - the holder briefly becomes untargetable and sheds negative effects. Then they gain 30% attack speed)

15 : Bloodthirster (Damage dealt heals holder for 25%. Once per combat at 40% Health, gain a 25% maximum health shield for up to 5 seconds)

16 : Giant Slayer (Abilities and attacks deal 25% more damage, increased to 50% if the holder has over 2200 maximum health)

17 : Infinity Edge (+10 Attack Damage, +75% Crit Chance, +10% Crit Damage, Converts every 1% excess critical strike chance into 1% bonus critical strike damage)

18 : Spear of Shojin (✓) (Basic attacks restore an additional 8 mana on-attack)

22 : Rabadons Deathcap (+75 Ability Power)

23 : Morellonomicon (+30 Ability Power, magic or true damage from an ability burns the holders target, dealing 25% of the targets maximum health as trude damage over 10 seconds and applying grevious wounds for the duration)

24 : Locket of the Iron Solari (At the start of combat, the wearer and all allies within 2 hexes in the same row gain a 300 / 350 / 400 health shield for 15 seconds - star level dependent)

25 : Ionic Spark (Enemies within 3 hexes have their magic resistance reduced by 50% (does not stack). When enemies within 3 hexes cast their ability, they are dealt 250% of their maximum mana as magic damage)

26 : Guinsoos Rageblade (Basic attacks grant 6% bonus attack speed for the rest of combat, stacks with no upper limit)

27 : \*Jeweled Gauntlet\* (+15% Crit Chance, +40% Crit Damage, +10 Ability Power, The holders magic adn true damage from abilities can critically strike)

28 : Archangels Staff (Grants the wearer 20 ability power every 5 seconds)

33 : Warmogs Armor (+1000 Health)

34 : Sunfire Cape (+400 Health. At the start of combat and every 2 seconds thereafter, applies a 10% maximum health burn as true damage over 10 seconds and applying grevious wounds for the duration)

35 : Zephyr (At the start of combat, banishes for 5 seconds the unit that mirrors the wielders placement on the other side of the board. Pierces through CC immunity effects)

36 : ZZ Rot Portal (At the start of combat, the wearer taunts enemies within 4 hexes. When the wearer dies, a Voidspawn arises, taunting nearby enemies. Summoned units can spawn Voidspawns at 25% effectiveness)

37 : \*Banshees Claw\* (+15% Dodge Chance, +150 Health, At the beginning of each round, the holder and allies within 1 hex in the same row gain a shield that blocks the first enemy ability, up to 600 damage)

38 : Redemption (Every 5 seconds, the wearer radiates an aura to allies within 1 hex, healing them for 12% missing health. Affected allies take 25% reduced damage from AOE attacks for  seconds)

44 : Bramble Vest (+60 Armor. Negatves 75% bonus damage from critical hits. On being hit by an attack, deal 75 / 100 / 150 magic damage to all nearby enemies (once every 2.5 seconds))

45 : Gargoyle Stoneplate (+18 Armor and Magic Resist for each enemy targeting the holder)

46 : \*Titans Resolve\* (Gain 2 attack damage and ability power when attacking or taking damage. After stacking 25 times, gain 25 armor and magic resist and stop stacking)

47 : \*Shroud of Stillness\* (Shoot a beam that delays the first cast of affected enemies by 35%)

48 : Frozen Heart (Reduce the attack speed of enemies within 2 hexes by 25%)

55 : Dragons Claw (+120 Magic Resist, every 2 seconds, regenerate 1.2% maximum health for each enemy targeting the holder. If holder is a dragon, increase all bonuses and effects by 20%)

56 : \*Runaans Hurricane\* (+10 Atttack Damage, attacks fire a bolt at a nearby enemy, dealing 70% of the holders attack damage as physical damage)

57 : \*Quicksilver\* (+20% attack speed. Immune to crowd control for 15 secnds)

58 : Chalice of Power (+30 Ability Power to holder and 2 adjacent allies on same row)

66 : Rapid Firecannon (+50% attack speed and +1 attack range, attacks cannot miss)

67 : \*Last Whisper\* (Dealing physical damage reduces the targets armor by 50% for 5 seconds, does not stack)

68 : Statikk Shiv (+15% attack speed, every 3rd attack shocks enemies for 70 magic damage and reduces their magic resist by 50% for 5 seconds)

77 : \*Thiefs Gloves\* (Each round equip 2 random items, improve with player level, you cannot equip other items)

78 : \*Hand of Justice\* (+15 attack damage, +15% ability power. Attacks and abilities heal for 15% of damage dealt. Each round randomly increase 1 effect by 30%)

88 : Blue Buff (+20 Starting Mana. Gain 20 mana after casting an ability)

Although as outlined in design, prototype 1 will not feature all implementations for items.

After the board has given all the item effects, it needs to set the initial health of the champions:

for p1Champ in &mut self.p1Champions//set all initial health to correct value

        {

            p1Champ.initialHP = p1Champ.health;

        }

        for p2Champ in &mut self.p2Champions

        {

            p2Champ.initialHP = p2Champ.health;

        }

This ensures that the initial health is increased alongside the actual health when increased due to, for instance, a health increasing item. Initial health acts as a cap to a champions healthpoints, ensuring it cannot health to infinity so to speak.

The loop takes a mutable reference of self.p1Champions so it has mutable access to each champion, rather than just an immutable reference.

The final setup involves creating vectors to hold projectiles:

let mut p1Projectiles : Vec<Projectile> = Vec::new();//instantiate projectiles vecs

        let mut p2Projectiles : Vec<Projectile> = Vec::new();

This creates two new vectors to hold any projectiles made by champions on either team.

With this, I can begin the main game loop of the battle.

while self.p1Champions.len() > 0 && self.p2Champions.len() > 0//take turns while there are champions alive

        {

I begin by creating a while loop that runs while both teams have champions remaining

for champCount in 0..self.p1Champions.len()//take turn for all p1Champs

            {

                let thisChamp = self.p1Champions.pop\_front().unwrap();

                let alive = thisChamp.takeTurn(&mut self.p1Champions, &mut self.p2Champions, self.timeUnit, self.movementAmount, &mut p1Projectiles);

                if alive{

                    self.p1Champions.push\_back(thisChamp)

                }

            }

Here, I create a loop that iterates for each champ on player 1s team. It pops a champion from the front of the list and takes it turn. If the champ is alive at the end of the turn, it pushes the champ to the back of the list.

Once again, it avoids the reoccurring issue of two multiple copies of thisChamp. If thisChamp is not removed from p1Champions, there would be two multiple copies of it, the reference to thisChamp and the original in the vecdeque.

By utilizing a double ended vector, I can ensure that popping from the front of the list and pushing to the back is an O(1) operation for each, rather than using a normal vector where popping from the front would be an O(n) operation as every element would have to be moved forwards.

I then repeat this for player two:

for champCount in 0..self.p2Champions.len()//take turn for all p1Champs

            {

                let thisChamp = self.p2Champions.pop\_front().unwrap();

                let alive = thisChamp.takeTurn(&mut self.p2Champions, &mut self.p1Champions, self.timeUnit, self.movementAmount, &mut p2Projectiles);

                if alive{

                    self.p2Champions.push\_back(thisChamp)

                }

            }

The take turn function performs a turn for a SummonedChampion :

impl SummonedChampion {

    ///simulates a tick/ turn for a champion<br />

    ///friendlyChampions[selfIndex] : this champion<br />

    ///friendlyChampionsLocations : location of all friend champs (array of positions), for pathfinding<br />

    ///enemyChampions : all enemy champions, for targetting<br />

    ///timeUnit : time unit of a frame, in centiseconds<br />

    ///movementAmount : precalculated movement distance for 1 frame<br />

    fn takeTurn(&mut self, friendlyChampions : &mut VecDeque<SummonedChampion>, enemyChampions : &mut VecDeque<SummonedChampion>,timeUnit : i8, movementAmount : i8, projectiles : &mut Vec<Projectile>) -> bool

    {

The take turn method for summonedchampion takes in a mutable reference to self, a mutable reference to friendly and enemy champions, the time unit of the board, movement amount and the projectiles of the team it is on.

self.targetCountDown -= timeUnit;//Reduce cooldown to check target/ find new target

        self.autoAttackDelay -= timeUnit as i16;

        self.gMD -= timeUnit as i16;

The function begins by reducing the target count down (delay until new target), the auto attack delay and the gain mana delay by the time unit, accounting for the time passed in the tick.

After this, I create a new block for simulating the status effects:

{

            let mut statusEffects = self.se;

            self.se = Vec::new();

            let mut stun = ShouldStun { stun: 0 };

            statusEffects.retain\_mut(|x| x.performStatus(self, friendlyChampions, enemyChampions, timeUnit, &mut stun));

            if self.health <= 0.0 { return false }

            self.se.extend(statusEffects);

            if self.shed == 1 { self.shed = 2; }

            else { self.shed = 0; }

            self.shields.retain\_mut(|x| x.updateShield(timeUnit));

            if stun.stun == 1 { return true }

        }

I move the status effects from the champion into a vector and replace it with a new, empty vector. I then create a ShouldStun object. ShouldStun is a simple class that holds a u8:

///ShouldStun<br />

///Simple struct to pass by reference to record whether stunned.<br />

struct ShouldStun

{

    ///Records whether champ is stunned. 0 = not stunned, 1 = stunned, 2 = locked (cannot be stunned)

    stun : u8,

}

I then call retain\_mut on status effect.

Status effect is a class that holds information/ details about a status afflicting a summoned champion:

///StatusEffect (struct)<br />:

///Stores a status type and a duration

struct StatusEffect

{

    ///Duration of status effect in centiseconds

    duration : Option<i16

    ///Whether the status effect has been applied

    applied : bool,

    ///Stores status type

    statusType : StatusType,

    ///Whether is negative for shred

    isNegative : bool,

}

It holds the enum StatusType, which describes the type of effect itself. It has a duration, which can either be a set length or indefinite (None), a bool to store whether the status effect has been applied and another one describing whether the status is a negative effect. If the status is negative, it will be cleansed when the champion “sheds” from the Edge of Night item effect.

Retain mut is a function that works on a vector/ vector like object, it goes through each element, calling a function that returns a bool. If the function returns true, the object is kept in the list, otherwise it is removed. It differs from retain by yielding a mutable reference to the object in the list, rather than an immutable one, allowing attributes like duration remaining to be updated on the status effect:

///Status Type (enum):<br />

///Holds information about what the status does

#[derive(PartialEq)]

enum StatusType

{

    ///Attack Speed Buff:<br />

    ///(bool : whether the buff has been applied, f32 : actual modifier)

    AttackSpeedBuff(f32),

    ///Increase Damage Taken:<br />

    ///(bool : whether the buff has been applied, i32 : actual modifier in % (so 120 = 120% or 20% increase))

    IncreaseDamageTaken(f32),

    ///Stun

    Stun(),

    ///Grevious Wounds:<br />

    ///Reduces healing by 50%

    GreviousWounds(),

    ///Gives edge of night buff<br />:

    EdgeOfNight(),

    ///Whether the target is targetable

    ///bool : Whether the buff has been applied

    Untargetable(),

    ///Bloodthirster shield at 40%

    Bloodthirster(),

    ///Morellonomicon Burn:<br />

    ///(f32 : damage per tick, f32 : damage to do, i16 : time til next tick)

    MorellonomiconBurn(f32, f32, i16),

    ///Ionic spark effect:<br />

    ///Reduces MR by 50%<br />

    IonicSparkEffect(),//maybe discrepencies? awkward cuz only lasts 1 frame?

    ///Archangel Staff:<br />

    ///(bool : applied. f32 : ap increase)

    ArchangelStaff(f32),

    ///Zephyr Item:<br />

    ///(bool : applied, i16 : banish duration)

    Zephyr(i16),

    ///Banished:<br />

    ///(bool : applied)

    Banished(),

    ///Taunted:<br />

    ///(usize : ID of taunter)

    Taunted(usize),

    ///Redemption:<br />

    ///(bool : applied)

    RedemptionGive(),

    ///Gargoyles Item Effect:<br />

    ///(u8: How many were targeting previous frame)

    Gargoyles(u8),

    ///Titans Resolve Item Effect:<br />

    ///(u8: Number of stacks previous frame)

    TitansResolve(u8),

    ///Shroud of Stillness Item Effect:<br />

    ///Immediately removed/ used at start of game

    ShroudOfStillness(),

    ///Protectors Vow Item Effect:<br />

    ProtectorsVow(),

    ///Dragon Claw Heal Item Effect:<br />

    DragonClawHeal(),

    ///Immune of CC Effect:<br />

    CrowdControlImmune(),

    ///Last Whisper Armor Shred Effect:<br />

    ///(bool : applied)

    LastWhisperShred(),

    ///Shreds Magic Resist Effect:<br />

    ///(bool : applied, f32 : multiplyer/ effect)

    ShredMagicResist(f32),

    ///None

    NoEffect()

}

I derive partial equal on statusType to allow comparison operators to be used on StatusTypes.

Perform status is a method on status effect which simulates the status.

impl StatusEffect {

    fn performStatus(&mut self, affectedChampion : &mut SummonedChampion, friendlyChampions : &mut VecDeque<SummonedChampion>, enemyChampions : &mut VecDeque<SummonedChampion>, timeUnit : i8, stun : &mut ShouldStun) -> bool {

It takes in self, the affected champion, a mutable reference to both teams champions, the time unit and a mutable reference to the should stun object as arguments, returning a bool for retain\_mut.

if self.duration.is\_some()

        {

            let mut nDuration = self.duration.unwrap().checked\_sub(timeUnit.into()).unwrap\_or(0); //unwrap duration and do checked subtraction

It begins by checking if the status has a duration, if so, it unwraps the duration and performs checked\_subtraction with time\_unit. Checked subtraction ensures that the i16 duration doesn’t underflow, replacing the value with 0 instead if it does.

if affectedChampion.shed == 2 && self.isNegative {//if shed and self is negative

                nDuration = 0;//set duration to 0

            }

Afterwards it checks if the affectedChampion is shedding and if the status is negative, if so it sets the duration to 0.

It goes on to check if the duration is zero, if so, it attempts to undo the effect given.

if nDuration <= 0 {

                match self.statusType//undo status effect/ remove effect. some effects aren't actually removed but just reinitialise

                {

                    StatusType::AttackSpeedBuff(modifier) => {

                        affectedChampion.attackSpeedModifier /= modifier

                    }

                    StatusType::IncreaseDamageTaken(modifier) => {

                        affectedChampion.incomingDMGModifier /= modifier

                    }

                    StatusType::Untargetable() => {

                        affectedChampion.targetable = true

                    }

StatusType::IonicSparkEffect() => {

                        affectedChampion.mr \*= 2.0;

                        affectedChampion.zap = false

                    }

StatusType::Banished() => {

                        affectedChampion.banish = false

                    }

StatusType::LastWhisperShred() => {

                        affectedChampion.ar \*= 2.0

                    }

Most of the status effects simply revert their effect, such as the attack speed buff or increase damage taken effects.

StatusType::MorellonomiconBurn(dmgPerTick, dmgToDo, timeTillNextTick) => {

                        if affectedChampion.shed == 2

                        {

                            return false;

                        }

                        if dmgPerTick > dmgToDo

                        {

                            affectedChampion.health -= dmgToDo;

                        }

                        else

                        {

                            nDuration = timeTillNextTick;

affectedChampion.health -= dmgPerTick;

                            self.statusType = StatusType::MorellonomiconBurn(dmgPerTick, dmgToDo - dmgPerTick, timeTillNextTick);

                        }

                    }

The morellonomicon burn is a damage over time effect, dealing 10% of the champion’s health overtime, as such, the morellonomicon effect takes advantage of duration, rather than removing itself when duration is zero, if there is still damage to do, it will do a “tick” of damage (the amount it does in one go) and “revive” the status by increasing the duration above zero to the time until the next “tick” of damage. If there is less damage to do then is done in a tick, the affected champion loses the remaining health from the burn before the status finally removes itself. By utilising the duration as the time until the next tick of damage, rather than the time until the status is removed, I can avoid having to add a separate variable to keep track of how much time is left until damage has to be done again and add behaviour that reduces this count each tick elsewhere as well. This does mean, however, that when shedding negative effects, morellonomicon would not be affected, as if there was still damage to do it would simply revive itself, so it checks to see if that is the case first.

                    StatusType::ArchangelStaff(apAmount) => {

                        nDuration = 500;

                        affectedChampion.ap += apAmount;

                    }

The ArchangelStaff effect (the effect given by the archangel staff) acts similarly. As the item archangel staff gives an ap increase every 5 seconds, the effect instead gives the boost each time the duration of the effect runs out, than increases the duration up to when the next effect should be given.

This is a reoccurring pattern that I use for numerous other status effects as well, such as RedemptionGive and Gargoyles.

                    StatusType::RedemptionGive() => {

                        nDuration = 100;//increase duration

                        let thisLocation = affectedChampion.location;

                        for friendlyChamp in friendlyChampions.iter\_mut()

                        {

                            if DistanceBetweenPoints(thisLocation, friendlyChamp.location) < 3

                            {

                                friendlyChamp.heal((friendlyChamp.initialHP - friendlyChamp.health) \* 0.12)//discrepency check at multitarget damage time for redemption heal for reduction

                            }

                        }

                    }

Redemption give is a status that heals all adjacent champs every second. To accomplish it, it takes an mutable iterator of friendlyChampions (as the heal method requires a mutable reference to self) and checks if each champion is within a distance of “3” (or 1.5 hexs) and if so, applies the healing.

The heal method on SummonedChampion is defined as:

    fn heal(&mut self, mut healingAmount : f32) {

        for statusEffect in &self.se//checks for grevious wounds

        {

            if statusEffect.statusType == StatusType::GreviousWounds()

            {

                healingAmount /= 2.0;//halves healing

                break;

            }

        }

        self.health += healingAmount;

        if self.health > self.initialHP

        {

            self.health = self.initialHP//makes sure to limit it to initial HP, so no healing to infinity

        }

    }

It takes in a mutable reference to self (so it can alter the health) and a mutable healing amount. As there is a status effect, grievous wounds, which halves healing, it first has to check through each status it has, checking its status type to see if it is grievous wounds, which if it is, halves the healing (why the healing amount is also mutable). Afterwards, it applies the healing, then ensures that health doesn’t go above the initialHP, decreasing it if so.

                    StatusType::Gargoyles(oldNumTargeting) => {

                        nDuration = 100;//increase duration

                        let mut numTargeting : u8 = 0;

                        let ourID = affectedChampion.id;

                        for enemyChamp in enemyChampions//get number of people targeting

                        {

                            if enemyChamp.target == ourID

                            {

                                numTargeting += 1;

                            }

                        }

                        let difference : f32 = (numTargeting - oldNumTargeting) as f32;//get change

                        affectedChampion.ar += 0.18 \* difference;

                        affectedChampion.mr += 0.18 \* difference;

                        self.statusType = StatusType::Gargoyles(numTargeting);

                    }

The gargoyles status effect is the status effect associated with the item Gargoyles Plating, which gives a defensive bonuses depending on the number of enemies targeting it. It updates itself every second, making use of the duration hitting 0 and increasing again that other status effects use. It first initialises a numTargeting variable and then iterates through each enemy champion, checking if its target is our id and increasing the count if so. Then it compares it against the stored, oldNumTargeting from the previous tick of this status, before using the difference to give or remove armor and magic resistance, before updating the effect with how many were targeting it this tick. By keeping track of the previous number targeting, it can add or remove bonuses depending on just the change and avoids giving too much or not removing anything.

Next we have dragon claw:

StatusType::DragonClawHeal() => {

                        nDuration = 200;//reset status effect

                        let mut numTargeting : f32 = 0.0;

                        let ourID = affectedChampion.id;

                        for enemyChamp in enemyChampions

                        {

                            if enemyChamp.target == ourID

                            {

                                numTargeting += 1.0;

                            }

                        }

                        let healingAmount = affectedChampion.initialHP \* 0.012 \* numTargeting;

                        affectedChampion.heal(healingAmount);

                    }

Which gives a heal depending on the number of enemies targeting it, once again it iterates through each enemy to check its target, before applying the correct healing amount.

StatusType::EdgeOfNight() => {

                        if affectedChampion.health <= (affectedChampion.initialHP / 2.0)

                        {

                            affectedChampion.se.push(StatusEffect {duration : Some(50), statusType : StatusType::Untargetable(), ..Default::default()});

                            affectedChampion.se.push(StatusEffect { duration: None, statusType: StatusType::AttackSpeedBuff(1.3), ..Default::default()});

                            affectedChampion.shed = 1;

                        }

                        else

                        {

                            return true

                        }

                    }

The edge of night effect for the edge of night item gives buffs when the champion is below half health. It tracks this by comparing its current health to half of the initial health of the champion.

If the champion is below half health, it gives the buff, an untargetable effect, attack speed boost and a shed of negative effects, otherwise the status returns true, exiting the retain\_mut function with the value true, keeping the effect in the vector. It does not, however, increase the duration, ensuring that the health check will repeat next tick.

StatusType::Bloodthirster() => {

                        if affectedChampion.health <= (0.4 \* affectedChampion.initialHP)

                        {

                            let quarterHP = affectedChampion.initialHP / 4.0;

                            affectedChampion.shields.push(Shield{duration : 500, size : quarterHP, ..Default::default()});

                        }

                        else

                        {

                            return true

                        }

                    }

Bloodthirster behaves similar to Edge of Night, giving a shield at 40% health. If the champion is below 40% health, it gives a shield equal to 25% of the champion’s health.

StatusType::Zephyr(banishDuration) => {

                        let oppositeLocation = [affectedChampion.location[1], affectedChampion.location[0]];//(!D) probs not opposite

                        let mut smallestDistance : i8 = 99;

                        let mut smallestDistanceID : usize = 0;

                        for (i , enemyChampion) in enemyChampions.iter().enumerate()

                        {

                            let distance = DistanceBetweenPoints(oppositeLocation, enemyChampion.location);

                            if distance < smallestDistance

                            {

                                smallestDistance = distance;

                                smallestDistanceID = i;

                                if distance == 0

                                {

                                    break;

                                }

                            }

                        }

                        enemyChampions[smallestDistanceID].se.push(StatusEffect{ duration: Some(banishDuration), statusType: StatusType::Banished(), ..Default::default() });

                    }

Zephyr is an effect/ item that “banishes” the enemy champion closest to the opposite position of itself on the board. It first calculates the opposite location, swapping around the x and y values.

It goes on to create a smallest distance variable, initialising it to a large value, and a smallest distance id, setting it to 0. It then iterates through each enemy champ, recording the distance between it and the location. If the distance is smaller than the smallest distance, it updates the smallest distance variable and id. It also checks to see if the distance is 0, in which case it can break earlier as there is no smaller distance.

Then, it gives the banished effect to the enemy closest to the location.

StatusType::Taunted(tauntID) => {

                        if findChampionIndexFromID(enemyChampions, tauntID).is\_some()

                        {

                            affectedChampion.target = tauntID;

                            affectedChampion.targetCountDown = 100;

                            nDuration = 20;

                        }

                    }

Taunted forces the target to attack the tauntee, the effect checks to see if the champion is still alive and if so, sets the target to the tauntee, sets the target cooldown to 100 so it doesn’t look for a new target soon, and sets the duration of the effect to 20, so it will be checked again soon.

findChampionIndexFromId is a utility function which tries to find the index of a champion in a vector from its id:

fn findChampionIndexFromID(champions : &VecDeque<SummonedChampion>, id : usize) -> Option<usize> //(!D) swap this out for check targetable as well

{

    for champ in champions {

        if champ.id == id {

            return Some(id);

        }

    }

    None

}

It returns an Option, as there is no guarantee the id is in the list.

It iterates through each of the given champions, checking if the id matches and returning it if so, otherwise it returns None at the end of the loop, as the id was not found in the list. This is similar to findIndexFromIdTargetable:

fn findChampionIndexFromIDTargetable(champions : &VecDeque<SummonedChampion>, id : usize) -> Option<usize>

{

    for champ in champions {

        if champ.id == id {

            if champ.targetable && !champ.banish {

                return Some(id)

            }

            return None

        }

    }

    None

Which functions the same, but also checks if the champion is targetable (targetable and not banished).

After reversing the effect/ simulating its continuance, the function checks to see if the nDuration is zero:

if nDuration > 0

                {

                    self.duration = Some(nDuration);

                }

                else

                {

                    return false

                }

If the duration is above zero, it changes the duration of the status, otherwise it returns false for retain\_mut, removing the status effect from the vector.

All of that code was just for if the duration was zero, now it moves on to check if the status has been “applied” :

if ! self.applied

        {

            self.applied = true;

Applied tracks whether the status has been applied already (so an attack speed buff doesn’t get given every tick), if it hasn’t, it sets applied to true then gives the associated effect.

match self.statusType

            {

                StatusType::AttackSpeedBuff(modifier) => {

                    affectedChampion.attackSpeedModifier \*= modifier;

                }

                StatusType::Stun() => {

                    self.applied = false;

                    if stun.stun == 0//has to check stun.stun == 0 as if stun.stun == 2 it is immune

                    {

                        stun.stun = 1;

                    }

                }

                StatusType::IncreaseDamageTaken(modifier) => {

                    affectedChampion.incomingDMGModifier \*= modifier;

                }

                StatusType::Untargetable() => {

                    affectedChampion.targetable = false

                }

                StatusType::IonicSparkEffect() => {

                    affectedChampion.mr /= 2.0;

                    affectedChampion.zap = true

                }

                StatusType::Banished() => {

                    affectedChampion.banish = true

                }

                StatusType::LastWhisperShred() => {

                    affectedChampion.ar /= 2.0;

                }

                StatusType::CrowdControlImmune() => {

                    self.applied = false;

                    stun.stun = 2;

                }

                \_ => ()

            }

Most of these are fairly self explanatory, stun sets applied to false so the stun also gets given next tick. It can’t use the duration tick of increasing the duration every time to apply an effect every tick or every so often as, unlike morellonomicon that removes itself once all the damage has been done, there is no other factor that describes whether stun should end other than the duration.

Stun also checks to see if stun.stun is equal to 0, setting it to 1 if so, this is because, as you can see in StatusType::CrowdControlImmune (ie stun immune) it sets stun.stun to 2, signaling that it is currently immune to stuns and other crowd control effects.

true }

At the end of the perform status function, it returns true, to signal that the effect should be kept in the list, as it was not removed earlier when it checked the duration.

With that out the way we have the rest of the block now:

if self.health <= 0.0 { return false }

            self.se.extend(statusEffects);

            if self.shed == 1 { self.shed = 2; }

            else { self.shed = 0; }

            self.shields.retain\_mut(|x| x.updateShield(timeUnit));

            if stun.stun == 1 { return true }

It checks if after the status effects, its health is now 0, from say a morellonomicon burn, returning false if so.

It then extends self.se by the status effect list that just had retain\_mut on it. Although self.se was set to an empty vector, I cannot do self.se = statusEffects, as some effects such as Edge of Night give more status effects, which will have been added to self.se. It then updates the shed value, to 2 if it is 1, or 0 otherwise.

It does this because when in Edge of Night it sets self.shed to 1, if it started removing negative effects right away afterwards, then it would only remove negative status effects that came after Edge of Night in the vector, so some may be missed. To work around this, it waits one tick (updating self.shed to 2) then removing the negative effects, ensuring it gets them all.

Afterwards it updates any shields on the champion, with retain\_mut.

struct Shield {

    ///duration of shield

    duration : i16,

    ///number of damage blocked

    size : f32,

    ///Optional choice for whether it only blocks a certain type

    blocksType : Option<DamageType>,

    ///Whether it pops after receiving any damage

    pop : bool,

}

Shield is a simple struct, holding a duration, a size, a type of damage it blocks and a “pop” bool, which tracks whether it should be “popped” after any damage (so even if the damage isn’t enough to kill the shield, it still removes itself).

impl Shield {

    fn updateShield(&mut self, timeUnit : i8) -> bool { //updates self

        self.duration -= timeUnit as i16; //(!O)

        return self.duration > 0 && self.size > 0.0

    }

}

Update shield is a simple function that reduces the duration by the time unit and returns whether it should be kept, if the duration is above zero and the size (amount of damage it can block) is above zero.

Afterwards, it checks if the champion is stunned, and if so returns True, to end the champion’s turn.

At this point, the end of this block:

{

            let mut statusEffects = self.se;

            self.se = Vec::new();

            let mut stun = ShouldStun { stun: 0 };

            statusEffects.retain\_mut(|x| x.performStatus(self, friendlyChampions, enemyChampions, timeUnit, &mut stun));

            if self.health <= 0.0 { return false }

            self.se.extend(statusEffects);

            if self.shed == 1 { self.shed = 2; }

            else { self.shed = 0; }

            self.shields.retain\_mut(|x| x.updateShield(timeUnit));

            if stun.stun == 1 { return true }

        }

Status effects have been handled for the turn. The reasoning to put this in an indented block is due to rust’s memory rules, by putting it in a block, then closing the block, any local variables defined within the block will be dropped, such as statusEffects. This ensures that memory/ data isn’t held longer than necessary, improving the efficiency of the program.

Afterwards it checks if self is banished, ending the turn if so:

if self.banish { return true }

Afterwards, the champ tries to locate its target:

let mut indexStore : Option<usize> = None;

        let mut needNewTargetCell : bool = false;//Bool to store whether new path is needed

        if self.targetCountDown >= 0 //if already has target and doesnt want to change targets

        {

            indexStore = findChampionIndexFromIDTargetable(enemyChampions, self.target)

        }

let mut distanceToTarget : i8 = i8::MAX;

It initialises an index store to None and a variable to keep track of whether it needcs a new target cell to false. Finally, it initiliases a distance to target cell with a value of i8::MAX.

If the target cooldown is above zero, and thus doesn’t want to switch targets, it tries to find the index of the target. As findChampionIndexFromIdTargetable returns an option, if the target isn’t found it just returns None.

if indexStore.is\_none()

        {

Then if it needs a new target or it wasn’t found

self.targetCountDown = 100;//reset target cooldown

            self.target = 0;//reset target

            let mut distance; //cache to store distance between enemy and location

            needNewTargetCell = true; //tells us to recalculate pathfinding later

We reset the target cooldown, set the target to 0 (in case we fail to find one) and initialises a distance variable.

It also sets needNewTargetCell to true so we know we need to find a new pathfinding target cell.

for (i, enemyChampion) in enemyChampions.iter().enumerate()

            {

                if !enemyChampion.targetable || enemyChampion.banish { continue; }

                distance = DistanceBetweenPoints(enemyChampion.location, self.location); //calculate distance

                if distance < distanceToTarget {//if distance to current enemy champion in loop is lower than distance to current target

                    self.target = enemyChampion.id; //change target

                    distanceToTarget = distance; //updating distance to new lower value

                    indexStore = Some(i);

                }

            }

            if indexStore.is\_none(){

                return true;

            }

We then iterate through each champ and its index (enumerate) and check if the champion is targetable. If so it calculates the distance between it and the potential target and if its lower than the distanceToTarget value, it updates the target with the new potential one, as well as the index store. This selects the closest targetable champion as the target.

Once the loop has finished, we check if the indexStore is still none, if so then there is no targetable champion available and we should end the turn, because if there was any targetable champion, the distance between it and our champion would have been lower than i8::MAX.

else {

            distanceToTarget = DistanceBetweenPoints(self.location, enemyChampions[indexStore.unwrap()].location);

        }

We then have an else block in case indexStore was not none, setting distance to target to the distance between it and the target we already have.

let index = indexStore.unwrap();

We then unwrap the index, removing the “option”. We know the indexStore will not be None as we would have ended our turn if so.

if distanceToTarget <= self.ra as i8//if target in range

        {

We then check if our target is in range.

if self.autoAttackDelay <= 0//if autoattack ready

            {

If so we check if our auto attack delay is equal to or below zero, signifiying our auto attack is ready.

self.autoAttackDelay = max((100.0 / (self.aS \* self.attackSpeedModifier)) as i16, 20); //calculating auto attack delay

We set the auto attack delay to either 100.0/ (self.aS \* self.attackSpeedModifier) or 20, as there is a limit in TFT of 5 auto attacks per second (or 100 centiseconds).

if self.items.contains(&26) { self.attackSpeedModifier \*= 1.06 }

We then check if we have item id 26, Guinsoo’s Rageblade, which every time you auto increases attack speed by 6%.

if self.gMD <= 0

                {

                    self.cm += 10;

                    if self.items.contains(&18) { self.cm += 8; }

                }

If we can gain mana, increase self.mana by 10 for auto attacking, by another 8 if we have spear of shojin.

We then check for a dodge

if enemyChampions[index].dc <= 0 || enemyChampions[index].dc < rand::thread\_rng().gen\_range(0..100) {

If the target has no dodge chance, or if their dodge chance is smaller than a random number from 0-100 (chance element) then the attack goes through/ hits.

self.dealDamage(friendlyChampions, &mut enemyChampions[index], self.ad, DamageType::Physical(), false);

We then deal damage to the enemy champion.

Deal damage takes self, friendly champions, a target, the damage amount and its type:

fn dealDamage(&mut self, friendlyChampions : &mut VecDeque<SummonedChampion>, target : &mut SummonedChampion, damageAmount : f32, damageType : DamageType, \_isSplash : bool){

We first increase the damage amount by the target’s incoming damage modifier:

let mut damage : f32 = damageAmount \* target.incomingDMGModifier;

Then initialise canCrit and critDamage:

let mut canCrit = false;

        let mut critD = self.critD;

match damageType{

            DamageType::Physical() => {

                canCrit = true;

                damage /= 1.0 + target.ar;

                if self.items.contains(&67){ //apply armor shred from last whisper

                    let mut alreadyHasShred = false;

                    for statusEffect in &target.se//check if they already have armor shred

                    {

                        if StatusType::LastWhisperShred() == statusEffect.statusType

                        {

                            alreadyHasShred = true;

                            break;

                        }

                    }

                    if ! alreadyHasShred//if they don't, give it

                    {

                        target.se.push(StatusEffect{duration : Some(500), statusType : StatusType::LastWhisperShred(), isNegative : true, ..Default::default()})

                    }

                }

                if self.cr > 100 && self.items.contains(&17){ //give extra crit damage from infinity edge

                        critD += (self.cr - 100) as f32

                }

            }

            DamageType::Magical() => {

                canCrit = self.items.contains(&27);

                damage /= 1.0 + target.mr;

            }

            DamageType::True() => {

                canCrit = self.items.contains(&27);

            }

        }

We then check the damage type, if its physical, we set can crit to true, reduce the damage by the enemy armor, apply last whisper shred to the target if we have the item and they don’t already have the effect, before increasing our crit damage by extra critical strike chance if we have infinity edge.

If the damage type is magical, canCrit depends on whether we have jeweled gauntlet, so we can self.items, and reduce the damage by the target’s magic resist.

If its true, there is no damage reduction, but we set can crit to whether we have jewled jauntlet again.

if canCrit && self.cr > rand::thread\_rng().gen\_range(0..100) {

            let mut extraDamage = damage \* critD;

            if target.items.contains(&44) { //reduce dmg if target has bramble vest

                extraDamage /= 4.0;

            }

            damage += extraDamage

        }

Then, if we can crit and we get the roll for it, we create an extra damage variable which is our damage \* crit damage. If the target contains bramble vest, we reduce the extra damage, before adding it onto our damage variable.

if self.items.contains(&16) {//give bonus giant's slayer attack dmg

            if target.initialHP >= 2200.0 { damage \*= 1.45 }

            else { damage \*= 1.2 }

        }

Here we give the additional giant slayer damage, increasing by 45% if their initial hp is above 2.2k, otherwise just 20%.

if damageType != DamageType::Physical() { //give gunblade and morellos

            if self.items.contains(&12) { //give gunblade healing

                    let healing = damage / 4.0;//calculate healing

                    self.heal(healing);//heal self

                    let lowestHPChamp = friendlyChampions.iter\_mut().reduce(|x, y| if x.health < y.health {x} else {y}); //get lowest HP ally

                    if lowestHPChamp.is\_some(){ //if there are any allies

                        lowestHPChamp.unwrap().heal(healing)

                    }

                }

            if self.items.contains(&23) { //if self has morellos give morellos effect

                target.se.push(StatusEffect { duration: Some(1000), statusType: StatusType::GreviousWounds(), isNegative: true, ..Default::default()});

                let dmgToDo = target.initialHP / 4.0;

                target.se.push(StatusEffect { duration: Some(100), statusType: StatusType::MorellonomiconBurn(dmgToDo / 10.0, dmgToDo, 100), isNegative : true, ..Default::default()})//discrepency unsure whether burn just reapplies itself

            }

        }

Then, if the damage type is magical or true, we check whether we should give gunblade healing or morellonomicon damage.

If we have gunblade (12), we get 25% of the damage as healing and heal ourselves. Then we find the lowest health ally. We turn friendlychampions into an mutable iterable then “reduce” it. Reducing a vector goes through the vector, comparing two elements in the list one by one in a defined way. One of the elements continues onwards to be compared to the next element, on and on until there is one left, thus “reducing” it. In this case, we compare health, yielding the one with the lower health. It returns an option, as there is a chance friendly champions is empty, so we check if there is a champ to heal, before also applying the healing to that.

If we have morellos (23), we give the target grievous wounds for 10 seconds and the Morellonomicon burn for 25% of its hp.

Back to the rest of the function, we then heal by our omnivamp count:

self.heal(damage \* self.omnivamp); //give omnivamp healing

Before reducing the damage we deal to the health of the target by any shields it has:

for shield in &mut target.shields {//reduce damage due to shields

            if damageType == shield.blocksType.unwrap\_or(damageType) {//if shield is of correct dmg type (or doesn't specify)

                if damage > shield.size//if damage greater than shield

                {

                    damage -= shield.size;//reduce dmg but remove shield

                    shield.size = 0.0;

                    shield.duration = 0;

                }

                else {

                    shield.size -= damage;//reduce shield size

                    damage = 0.0;//set dmg to 0

                    if shield.pop//if shield has pop

                    {

                        shield.size = 0.0;//remove shield

                        shield.duration = 0;

                    }

                    break;

                }

            }

        }

We iterate through each target shield, checking if the damage type it blocks is equal to the type we are dealing (or if it blocks all of them ie None). If the damage is bigger than the shield, we reduce the damage by the shield size and set the shield size and duration to 0, to be removed next tick.

Else, we reduce the shield size by the damage, set damage to 0 and if the shield pops upon any damage, also remove the shield.

If all the damage has been done, we break.

We give both of the champions involved their titans resolve stacks, ensuring it doesn’t go above the cap of 25:

self.titansResolveStack = min(self.titansResolveStack + 1, 25);//add titan's resolve stacks

        target.titansResolveStack = min(target.titansResolveStack + 1, 25); //give enemy titan's resolve stacks

Then finally deal the damage:

target.health -= damage;

With that we have covered the deal damage method.

else

        {

We’re now in the path of if the target is out of range of our champion.

if needNewTargetCell || self.location == self.targetCells //if need to update pathfinding or at pathfinding target

            {

We now check if we need a new target cell, whether we changed target earlier so needNewTargetCell was set to true, or if we have arrived at our destination.

self.targetCells = self.location; //setting target cells to location so if it does not find a target this frame will try to do it again

We set target cells to our location, so if we fail to find a new target cell we just do not move.

let mut lowestDistance : i8 = 100; //setting lowestDistance to high value

                let mut newPosition : [i8 ; 2];

We initialise a mutable lowestDistance var with a high value, and an empty new positon variable.

for possibleMove in [[0, -1], [1, -1], [1, 0], [-1, 0], [-1, 1], [0, 1]] //for every possible move

                {

                    newPosition = [self.location[0] + possibleMove[0], self.location[1] + possibleMove[1]];

                    distanceToTarget = DistanceBetweenPoints(newPosition, enemyChampions[index].location);

                    if distanceToTarget < lowestDistance

                    {

                        let mut failed = false;

                        if ! InGridHexagon(newPosition)

                        {

                            continue;

                        }

                        for friendlyChampionLocation in friendlyChampions.iter()

                        {

                            if friendlyChampionLocation.location[0] == newPosition[0] && friendlyChampionLocation.location[1] == newPosition[1]

                            {

                                failed = true;

                                break

                            }

                        }

                        if failed

                        {

                            continue;

                        }

                        println!("Debug : Found a Target Cell");

                        lowestDistance = distanceToTarget;

                        self.targetCells = newPosition;

                    }

                }

We then iterate through all possible moves, create the new position and then calculate the position between the new position and the target’s position.

If its lower than the lowest distance so far, we check if the position is in the grid and if so, then check that there is not currently a friendly champion in that position, failing if so and moving onto the next possible location.

If there are no possible moves, the target cell will still be on its current location, so it will not move.

If the distance is lower and it’s a valid position, we update lowest distance and the target cells. This simple pathfinding ensures that we avoid expensive calculations as used in the A\* algorithm and similar, when it is not entirely necessary.

Now we have a target pathfinding cell, or we already had one, we can calculate the movement.

self.movementProgress[0] += movementAmount \* sign(self.targetCells[0] - self.location[0]);

self.movementProgress[1] += movementAmount \* sign(self.targetCells[1] - self.location[1]);

We add to movement progress the movement amount multiplied by sign of the distance between the target cell and our location.

fn sign(num : i8) -> i8

{

    if num == 0

    {

        0

    }

    else if num > 0

    {

        1

    }

    else

    {

        -1

    }

}

Sign is a simple function that returns 1 if the number is bigger than 0, 0 if it is 0, or -1 otherwise. It is used in this scenario to calculate the direction it needs to travel in, multiplying the direction by the magnitude of movement achieves the desired effect of movement in the correct direction.

if self.movementProgress[0].abs() >= 10

            {

                self.location[0] += sign(self.movementProgress[0]);

                self.movementProgress[0] = 0;

            }

            if self.movementProgress[1].abs() >= 10

            {

                self.location[1] += sign(self.movementProgress[1]);

                self.movementProgress[1] = 0;

            }

We then check if the absolute value of movement progress is bigger than or equal to 10, in which case it is ready to move. We move in the correct direction and reset our movement progress, doing this for both x and y values.

if self.items.contains(&25)

        {

            let thisLocation = self.location;

            for enemyChamp in enemyChampions.iter\_mut()

            {

                if DistanceBetweenPoints(thisLocation, enemyChamp.location) < 7//discrepency check distance between points returns value twice as large?

                {

                    enemyChamp.se.push(StatusEffect { duration: Some((timeUnit + 1).into()), statusType: StatusType::IonicSparkEffect(), isNegative: true, ..Default::default()});

                }

            }

        }

Next, we apply an item effect, ionic spark, if the champion has the item, we take a mutable iteratable of enemy champions and check if each enemy champ is within a distance of 7 (or 3.5 hexes), if so, we push the ionic spark effect upon them.

if self.cm >= self.mc

        {

            if self.zap

            {

                self.health -= (self.mc as f32) \* 2.5;

            }

            self.cm = 0;

            self.gMD = 100;

            self.castAbility(friendlyChampions, enemyChampions, projectiles)

        }

        true

We end the function by returning true, indicating the champion is still alive, but before that, we check if we have enough mana to cast our ability.

If we do, and we are suffering from the ionic spark effect/ debuff, we lose some health from a zap, we set our current mana to 0 and gain mana delay to 100 or 1 second, we then cast our ability.

fn castAbility(&mut self, friendlyChampions : &mut VecDeque<SummonedChampion>, enemyChampions : &mut VecDeque<SummonedChampion>, projectiles : &mut Vec<Projectile>)

    {

        match self.of\_champ\_id {

Cast ability takes a mutable reference to self, friendlyChampions, enemyChampions and to self’s team’s projectiles.

Depending on the id of the champion, the ability differs.

0 => { //support ability

For 0, lulu/ the support’s ability, we have an ability that stuns enemies and gives attack speed bonuses to allies. It applies to the closest 3/4/5 champions on the board (increases with star level).

let mut champion\_distances: Vec<(i8, &mut SummonedChampion, bool)> =

                    friendlyChampions

                        .iter\_mut()

                        .map(|x| (DistanceBetweenPoints(self.location, x.location), x, true))

                        .collect(); //create a vector of distances from the SummonedChamp, SummonedChamp and bool representing its team (true = same team, false = other team)

                champion\_distances.extend(

                    enemyChampions

                        .iter\_mut()

                        .map(|x| (DistanceBetweenPoints(self.location, x.location), x, false)),

                ); //extend with enemy champions

To achieve this effect, we create a a champion\_distances vector, a vector which holds a tuple containing the distance between self and the champion, a mutable reference to the summonedchampion, and a bool of whether the champion is on the same team as the ability caster.

We create a mutable iterable of friendlyChampions and map each summoned champion into a tuple of the distance between points, x/ itself and a bool saying true. We then collect this iterable into the vector.

We then extend this vector by the enemy champions, repeating the process turning it into a mutable iterable and then mapping the values.

let star\_level = self.star\_level; //gets current star level

let number\_affected: usize = [3, 4, 5][star\_level]; //how many champions it can hit/ effect

We then get the number affected by the effect via indexing an array with the star level.

champion\_distances.sort\_unstable\_by\_key(|a| a.0);

We then call the rust in built method for vectors sort unstable by key, sorting the vector by the first item in each tuple (the player distance). Sort unstable by key makes use of quicksort to provide efficient sorting for the vector.

let mut i = 0; //count number already affected

                let ap = self.ap; //get ability power

for (\_, champ, on\_team) in champion\_distances //iterate through play distances

                {

                    if i >= number\_affected { //already affected all it can

                        break;

                    }

                    if on\_team

                    //if friendly champ

                    {

                        //give allies attack speed for 5 seconds

                        champ.se.push(StatusEffect {

                            duration: Some(500),

                            statusType: StatusType::AttackSpeedBuff(1.7 \* ap),

                            ..Default::default()

                        });

                    } else {

                        //enemy champ

                        //stun enemies for 1.5 seconds and increase damage for 20%

                        champ.se.push(StatusEffect {

                            duration: Some(150),

                            statusType: StatusType::Stun(),

                            isNegative: true,

                            ..Default::default()

                        });

                        champ.se.push(StatusEffect {

                            duration: Some(150),

                            statusType: StatusType::IncreaseDamageTaken(1.2 \* ap),

                            isNegative: true,

                            ..Default::default()

                        });

                    }

                    i += 1; //add 1 to count of hit enemies

                }

We define a variable i to count the number of times we’ve applied the effect and begin looping through the champion distance vector. We check if we’ve applied the effect enough already and then check if we’re on the correct team or not. If we are we give an attack speed buff, otherwise we apply a stun and increase the damage taken. We then increase i by one.

if i < number\_affected

                //give self effect if there aren't enough champs to hit

                {

                    self.se.push(StatusEffect {

                        duration: Some(500),

                        statusType: StatusType::AttackSpeedBuff(1.7 \* ap),

                        ..Default::default()

                    });

                }

At the end of the ability we check if have ability casts left, if so we cast it on self.

As self has been removed from friendly champions, we do not have to worry about the ability being applied twice.

Having to define a separate variable i to count the number of ability casts feels obtuse yet it is necessary. We cannot enumerate directly through the list like we normally would in rust as we need to check the count after the loop is over and we cannot loop through the vector a set number of times (the number of ability casts) as there may not be enough champions in the vector.

Then we move onto the next ability, the bruiser/ aatrox.

  1 => { //bruiser ability

                let star\_level = self.starLevel; //get star level

                let target\_index =

                    findChampionIndexFromID(enemyChampions, self.target).unwrap(); //get target object or the first in vec

                self.heal((300.0 + 50.0 \* star\_level as f32) \* self.ap); //heals

                //deals damage

                self.dealDamage(

                    friendlyChampions,

                    &mut enemyChampions[target\_index],

                    (25.0 \* star\_level as f32) \* 4.0 \* self.ad,

                    DamageType::Physical(),

                    false,

                )

            }

We locate our target, which we know will be valid as our turn has just passed and if we were without a target our turn would have ended earlier. We heal by 300 + a bonus from star level and ability power, before dealing damage to our target.

2 => {

                let target = findChampionIndexFromID(enemyChampions, self.target).unwrap\_or(0); //get target

                let target\_location = enemyChampions[target].location;

                let damage: f32 = self.ad \* 3.0 \* (self.starLevel as f32); //calculate damage

                projectiles.push(Projectile::new(

                    self.location,

                    Option::Some(target\_location),

                    self.target,

                    damage,

                    DamageType::Physical(),

                    0.0,

                    5,

                    self.id,

                )) //create new projectile

            }

            3 => {

                //fetches target index

                let target = findChampionIndexFromID(enemyChampions, self.target).unwrap\_or(0); //(!D) Can strike from out of range

                                                                                                     //gets their location

                let target\_location = enemyChampions[target].location;

                //calculates damage

                let damage: f32 = 250.0 \* self.ap \* (self.starLevel as f32);

                //adds projectile to vec

                projectiles.push(Projectile::new(

                    self.location,

                    Option::Some(target\_location),

                    self.target,

                    damage,

                    DamageType::Magical(),

                    damage / 3.0,

                    3,

                    self.id,

                ))

            }

Abilities 2 and 3 are very similar. They find their target, calculate damage and then push a new projectile to the projectiles vector.

Once we have casted our ability, the turn for the champion is over.

p1Projectiles.retain\_mut(|f| f.SimulateTick(&mut self.p2Champions, &mut self.p1Champions));

            p2Projectiles.retain\_mut(|f| f.SimulateTick(&mut self.p1Champions, &mut self.p2Champions));//simulate projectile ticks

To finish up the while loop, we use the retain\_mut pattern once again to simulate the ticks for the projectiles.

///Projectile struct

struct Projectile

{

    ///location of projectile

    location : [i8 ; 2],

    ///location progress

    locationProgress : [i8 ; 2],

    ///target location

    targetLocation : Option<[i8 ; 2]>,

    ///enemy Champion index

    targetID : usize,

    ///projectile damage

    damage : f32,

    ///projectile damage type

    damageType : DamageType,

    ///amount of splash damage

    splashDamage : f32,

    ///speed of projectile

    speed : i8,

    ///id of shooter (so can give item effects etc)

    shooterID : usize,

}

The projectile class holds a location and location progress. It also holds an option for a target location and also a target id. If a target location is given, it will path towards that location, otherwise it will path towards the location of the target given by target id.

impl Projectile

{

    ///Simulates a single tick of a projectile

    fn SimulateTick(self : &mut Projectile, possibleTargets : &mut VecDeque<SummonedChampion>, friendlyChampions : &mut VecDeque<SummonedChampion>) -> bool

    {

Simulate tick takes self, a list of enemy champions and friendly champions.

let targetLocation = match self.targetLocation //discrepency only checks after move to theoretically could phase through someone

        {

            Some(location) => {location}, //gets target location

            None => {let outLocation = findChampionIndexFromID(&possibleTargets, self.targetID);//gets location of target champion

                match outLocation

                {

                    Some(index) => possibleTargets[index].location,

                    None => [-1, -1],

                }

        }};

We initialise a variable target location, depending on the value of self.targetLocation.

If there is a target location, we set it to that, otherwise we find the target from its id. If we can find it, we return its location, otherwise we set the location to -1 -1.

if targetLocation[0] == -1//not found, remove projectile

        {

            return false

        }

If the location is -1, we return false, removing the projectile, as its target is dead.

self.locationProgress[0] += self.speed \* sign(targetLocation[0] - self.location[0]);

        self.locationProgress[1] += self.speed \* sign(targetLocation[1] - self.location[1]);//add location progress

        if self.locationProgress[0].abs() >= 10//if above 10, move

        {

            self.location[0] += sign(self.locationProgress[0]);

        }

        if self.locationProgress[1].abs() >= 10

        {

            self.location[1] += sign(self.locationProgress[1]);

        }

We then add location progress and check if we should move.

if ! InGridHexagon(self.location)//if out of grid, remove

        {

            return false;

        }

Then, we check if the projectile is still in the grid, if not, we remove it, as it will not hit anything/ factor into the board, so should be removed to avoid wasting processing power.

for possibleTarget in possibleTargets.iter\_mut()//iterate through all possible collisions

        {

            if self.location == possibleTarget.location//has a hit

            {

                possibleTarget.health -= self.damage;

                if self.splashDamage > 0.0//if there is splash damage

                {

                    for possibleSecondaryTarget in possibleTargets.iter\_mut()//iterate through possible splash hits

                    {

                        if DistanceBetweenPoints(self.location, possibleSecondaryTarget.location) < 3

                        {

                            possibleSecondaryTarget.health -= self.splashDamage;

                        }

                    }

                }

                return false//remove self

            }

        }

        true

    }

Afterwards, the projectile looks for any collisions. It iterates through each enemy champion, if the location matches, it deals the projectile damage.

It then checks to see if the projectile has any splash damage, if so it iterates through each target again, looking for any adjacent champs and if so dealing additional damage.

Then it returns false to remove itself from the vector.

If there was not a collision, it returns true to stay.

With that we have covered the entirety of the page loop, in 30 short pages.

Once the game loop is over, the board returns the winner of the battle:

if self.p1Champions.len() == 0//check winner and get champ information

        {

            println!("Debug : Player 2 Won");

            for champion in &self.p2Champions

            {

                println!("Champ Remaining ID,  Health : {0} {1}", champion.id, champion.health)

            }

            return 2;

        }

        else

        {

            println!("Debug : Player 1 Won");

            for champion in &self.p1Champions

            {

                println!("Champ Remaining ID,  Health : {0} {1}", champion.id, champion.health)

            }

            return 1;

        }

It checks for which team has no champions remaining and returns the other champion, with some helpful debugging info along with it.

We can create boards in prototype as follows:

let playerOneChamps : VecDeque<PlacedChampion> = vecPlacedChampion{id : 0, star : 0, items : [0, 0, 0], location : [3, 0]}];

    let playerTwoChamps : VecDeque<PlacedChampion> = vec![PlacedChampion{id : 1, star : 0, items : [0, 0, 0], location : [6, 7]}];

    let board : Board = Board::new(&playerOneChamps, &playerTwoChamps, 10);

And then simulate the battle with board.startBattle().

Now, in order to test prototype 1, I need to be able to test a wide variety of boards to look for errors/ fufill the prototype requirements. In order to do this, I can create code that generates random boards and then test them to measure performance and error handling.

impl Board

{

fn generate\_random\_board(timeUnit : i8) -> Board {

        let num\_p1\_champs: usize = rand::thread\_rng().gen\_range(1..6);

        let num\_p2\_champs: usize = rand::thread\_rng().gen\_range(1..6);

        let p1\_champions: VecDeque<PlacedChampion> = (0..num\_p1\_champs)

            .map(|\_ : usize| PlacedChampion::generate\_random\_champ(false))

            .collect();

        let p2\_champions: VecDeque<PlacedChampion> = (num\_p1\_champs

            ..num\_p1\_champs + num\_p2\_champs)

            .map(|\_ : usize| PlacedChampion::generate\_random\_champ(true))

            .collect();

        Board::new(&p1\_champions, &p2\_champions, timeUnit)

    }

}

This method generates a random board, given only a time unit to be run on.

We generate a random number of champions for player 1 and two, then create double ended vectors to hold them.

Then for the range 0 to number of champions, we map each number (which ends up unused), into a random PlacedChampion, given a bool signifying its team, we then collect this vector into p1 and p2 champions and return a new board.

Generate random placed champion works similarly

impl PlacedChampion {

    fn generate\_random\_champ(team : bool) -> PlacedChampion {

        let id = rand::thread\_rng().gen\_range(0..4);

        let star = rand::thread\_rng().gen\_range(0..3);

        let mut items : [u8 ; 3] = [rand::thread\_rng().gen\_range(0..30), rand::thread\_rng().gen\_range(0..30), rand::thread\_rng().gen\_range(0..30)];

        let location = match team {

            true => [rand::thread\_rng().gen\_range(3..6), 0],

            false => [rand::thread\_rng().gen\_range(0..6), 6],

        };

        PlacedChampion { id, star, items, location}

    }

}

We give it a random id, from 0 to 3, a random star level from 0 to 2, random items from the range and then a random location depending on the team, ensuring that teams spawn opposite sides of thet board to each other.

We then return the new Placed Champion.

This all ensures that we can generate random boards for testing purposes for prototype 1.

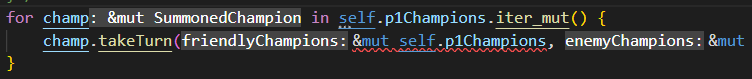
### Errors and Solutions:

Here are some of the errors I ran into developing prototype 1 and how I solved them:

**Borrow self.p1Champions as mutable twice (borrow error):**

![Text

Description automatically generated



Borrowing self.p1Champions first as a mutable to iterate through it, then in a second place in the champ.takeTurn returns an error as its being declared as mutable in two places.

Solution:

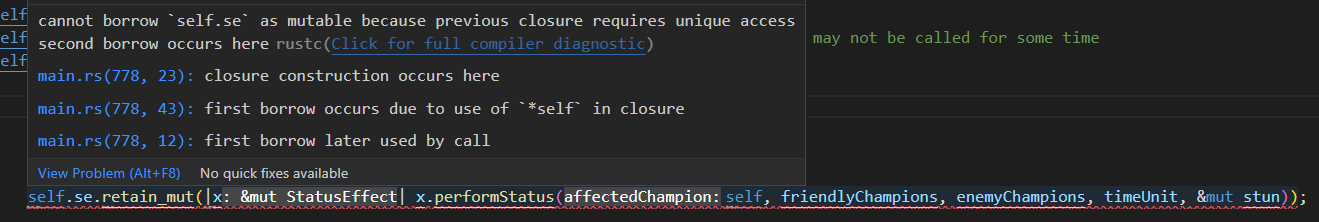
Text

Description automatically generated

Iterate through the list once, popping the front champion then pushing to the back once done. By removing thisChamp from the list, and iterating via indexes, you can avoid borrowing the two mutably in two places.

**Cannot borrow self.se because previous closure requires unique access (borrow error):**





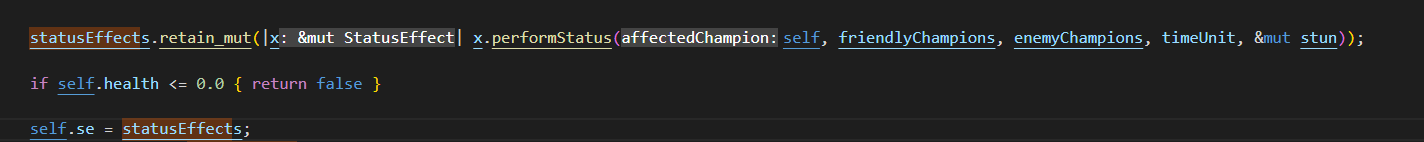
Calling retain\_mut on self.se directly involves having self borrowed mutably in two places, in the retain\_mut and in the perform\_status.

Solution:



Clone self.se and call retain\_mut on that.

**Effects given in x.perform\_status not staying (logic error):**



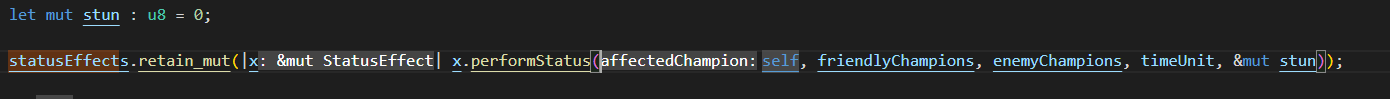
In status\_effect.perform\_status, when there is an additional status effect to be given, such as with edge of night, it gives it directly to self.se, but then when I set self.se = statusEffects to put the statuses back, it overwrites what was in self.se, removing the new status effects.

Solution:



Extend self.se by status effects instead, so we keep the new effects.

**Passing in stun directly as u8 (logic error):**



As stun is a u8, even though I pass it in as a mutable reference rather than by value, the rust compiler still simply copies the value into the function. Passing it in by value rather than reference means that any changes that occur in the function will only affect the function, so the value remains unchanged.

Solution:

Graphical user interface, text

Description automatically generated

Create a simple struct that holds a u8, so the rust compiler will not copy by value.

**Shed effect (logic error):**

If shed is a simple true or false value, true being shed and false being no shed, then on the turn that shed occurs, it will only shed negative effects that occur afterwards in the list. Then, afterwards when shed is reset, it will never get to shed the negative effects earlier.

Solution:  
Have shed be a value from 0-2, 0 being no shed, 1 being shed next turn, 2 being shed this turn, helping get the entirety of the negative effects.

**Targetable bool (logic error):**

As targetable is a bool, if two effects gives untargetable and one ends before the other, when the first status ends, it will change the champ to targetable, pre-emptively becoming targetable and hence undoing the effect of the second status.

Solution:



Have untargetable be a u8, counting the number of untargetable effects currently affecting it, 0 = targetable, anything above being untargetable, when a status begins, 1 is added, when it ends, 1 is removed, ensuring that the status doesn’t end pre-emptively.

**Movement progress jumping over 10 (logic error):**

Text

Description automatically generated

I originally checked if movement progress was equal to 10, but in scenarios when time unit was higher and thus there was more movement than 1 in a tick, it had the potential to jump over 10, thus preventing any movement.

Solution:

Graphical user interface, application

Description automatically generated

Check for movement progress bigger than or equal to 10.

**Name error:**



hp isn’t a field on SummonedChampion, its called health

Solution:  
Correct the variable to health instead of hp.

**Duration underflow error (underflow error):**



Underflow from i16 subtraction as the status isn’t removed for sometime, causing an unexpected crash. The same error and solution was applied to taret countdown, auto attack delay etc.

Solution:



Checked subtraction using the checked\_sub function, unwrapping into 0 in case of underflow.

### Test Table:

I laid out a number of tests I had to perform in prototype 1, here are the outcomes of the test.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Test | Expected Outcome | Runs | Accuracy | Comment |
| Robustness Test | <10% crash rate | 10000 | 9277 | Acceptable crash rate for prototype 1, 723 crashes in 10000 runs is slightly below the sub 10% crash rate requirement I set. Most “crashes” were not actually crashes but the battle lasting indefinitely, as there was no checking for draws, so I had to pre-emptively end them. |
| 3S1 vs 1S0 board | 3S1 winner | 100 | 85% |  |
| 1S0, 1S0 vs 2S1 | 1S0, 1S0 winner | 100 | 62% |  |
| 1S2 vs 1S3 | 1S3 winner | 100 | 100% |  |
| 1S0, 1S1, 1S2, 1S3 vs 3S3, 3S1 | 1S0, 1S1, 1S2, 1S3 winner | 100 | 52% | This battle featured a lot of draws/ indefinite battles. |
| 2S2 vs 3S3 | 3S3 winner | 100 | 88% |  |
| 3S1, 3S1 vs 3S3, 3S3, 3S1 | 3S3, 3S3, 3S1 winner | 100 | 92% |  |

### Review of Prototype:

Prototype 1 featured acceptable robustness for this early stage in development, with issues mainly due to the lack of a draw outcome in the case of a indefinite battle, which will have to be implemented in prototype 2.

The boards had fairly good accuracy, excluding the very complex battle with 6 champions which was barely above the requirement of 52%, so more work will need to be done to improve that. There are quite a few issue still existing in prototype 1, such as projectiles dealing splash damage to the champion it hits as well, as well as the initial damage from the projectile hit itself.

Projectiles hit also don’t account for resistances at the moment, all of which probably have a part to play in the discrepancies in accuracy.

Prototype 1 fulfills all the requirements laid out in design, with basic implentations of the required classes SummonedChampion, PlacedChampion and Champion. It also features the ability to simulate battles and outcome the result to a command line interface.



After consulting with my client, Suket Arya, he believed that for future prototypes, I should have more clarity about what is occurring in battles, ie the ability to have updates on what is happening mid battle, rather than just the final outcome, as well as a quicker way to create battles.

## Prototype 2:

Prototype 2 will build upon prototype 1, finishing implementations of items and classes as laid out in the design section. This will also be the final version of the simulator, with all future developments on the project be relating to the user interface and database. As this is the final version of the simulator, all code for this prototype is available to view in the appendix for this project, under the “simulator” folder (with slight alterations to be able to let it iteract with the rest of the project).

I will mostly be covering key changes and improvements from prototype 1. Most of the logic from prototype 1 has stayed, with optimisations and more succinct ways of completing the same action being implemented. The code is now more robust in nature and has fully implemented things left out of prototype 1 as well.

The code is now also entirely modular in nature, with the entire project being split into numerous files.

Graphical user interface, text, application

Description automatically generated

This allows me to encapsulate behavior and quickly locate areas that I need to work on, as well as identify the location of errors.

Moreover, scattered throughout the code, you may see info!(“…”) or warn!(“…”) blocks, which is used for debugging/ logging information, which helps me speed up debugging.

The biggest change from prototype 1 is location, which was originally laid out in design.

pub struct Location {

    ///x position

    pub x: i8,

    ///y position

    pub y: i8,

}

The location struct simply holds an x and y value, i8 as locations can be negative (even if not in the grid), but it does not need to hold large values, hence the i8.

What location does have is a lot of handy methods to avoid repeating redundant code.

fn calculate\_z(&self) -> i8 {

        -self.x - self.y

    }

The first of this is calculate\_z, which simply returns the z value of self.

///calculates the distance between two points

    pub fn distance\_between\_points(&self, other\_pos: &Location) -> i8 {

        (self.x - other\_pos.x).abs() //x distance

            + (self.y - other\_pos.y).abs() //y distance

            + (self.calculate\_z() - other\_pos.calculate\_z()).abs() //calcalates z then distance between them

    }

The next is the distance between two locations, which gets the absolute value of the x difference, y difference and z difference added together.

///subtracts two positions, returning the new location

    pub fn sub\_positions(pos\_one: &Location, pos\_two: &Location) -> Location {

        Location {

            x: pos\_one.x - pos\_two.x,

            y: pos\_one.y - pos\_two.y,

        }

    }

    ///returns a new location, of a location added to a array two long

    pub fn add\_position\_vec(pos\_one: &Location, pos\_two: [i8; 2]) -> Location {

        Location {

            x: pos\_one.x + pos\_two[0],

            y: pos\_one.y + pos\_two[1],

        }

    }

We have a subtract two positions, returning a new position, and adding an array of i8s to a position, returning a new location.

///returns a bool of whether the position is valid/ in the grid

    pub fn check\_valid(&self) -> bool {

           self.x >= 0

        && self.x < 10

        && self.y >= 0

        && self.y < 8

        && 2 - (self.y / 2) < self.x

        && 10 - (self.y / 2) >= self.x //last two lines account for slanting border/ x value

    }

We have a method to check whether a position is valid, with logic covered in the design section to ensure that the location is valid.

///generates a random position given a team (what side of the board to generate on)

    pub fn generate\_random\_position\_team(team: bool) -> Location {

        let y: i8 = if team {

            rand::thread\_rng().gen\_range(0..4) //generates random y

        } else {

            rand::thread\_rng().gen\_range(4..8)

        };

        let low = 2 - (y / 2) + 1; //calculate max and min values for x as they are dependant on y

        let high = 10 - (y / 2);

        let x: i8 = rand::thread\_rng().gen\_range(low..high);

        Location { x, y }

    }

We have a method that generates a random position given a team, we generate a y position depending on the team, so the teams are divided into the two sides of the board.

Then, we calculate the bounds for the x value depending on the y value and return the location.

///given a vector of summonedchamps, return the closest to self

    pub fn get\_closest\_to\_location<'a>(

        &self,

        champion\_list: &'a mut VecDeque<SummonedChampion>,

    ) -> Option<&'a mut SummonedChampion> {

        //iterates through the champion\_list, reducing it by comparing the distance to this location (reducing is comparing two sequential elements in the list and keeping the smaller one/ one that fits the bounds and repeating for the entire list, until you have only one)

        champion\_list.iter\_mut().reduce(|x, y| {

            if x.location.distance\_between\_points(self) < y.location.distance\_between\_points(self) {

                x

            } else {

                y

            }

        })

    }

Takes a vector of champions and returns the closest to self location.

It turns the vector into an iter mutable and reduces the iterable by the distance from this point, returning the one with the lowest distance.

pub fn get\_closest\_to\_location\_targetable<'a>(

        &self,

        enemy\_champions: &'a mut VecDeque<SummonedChampion>,

    ) -> Option<&'a mut SummonedChampion> {

        enemy\_champions.iter\_mut() //turn vector into iterator

            .filter(|x| x.get\_is\_targetable()) //filters through the iterator, not yielding any elements that are not targetable

            .reduce(|x, y| { //reduce the values by comparing the distance from self

            if x.location.distance\_between\_points(self) < y.location.distance\_between\_points(self) {

                return x;

            }

            y

        })

    }

This does the same as get closest to point, but filters the vector by only those who are targetable first.

pub fn get\_closest\_to\_location\_targetable\_index<'a>(

        &self,

        enemy\_champions: &'a mut VecDeque<SummonedChampion>,

    ) -> Option<(usize, &'a mut SummonedChampion)> {

        enemy\_champions

            .iter\_mut() //turn into iterator

            .enumerate() //get the indexes of the champions, BEFORE the filter so that indexes are valid/ accurate to given vector

            .filter(|(\_, x)| x.get\_is\_targetable()) //remove any champions that are not targetable

            .reduce(|(i, x), (j, y)| { //reduce by distance to point

                if x.location.distance\_between\_points(self)

                    < y.location.distance\_between\_points(self)

                {

                    return (i, x);

                }

                (j, y)

            })

    }

Another slight variation, this also gets the index of the closest targetable champion.

///given a distance, generates a filter to be used with .filter

    pub fn get\_within\_distance(

        self,

        distance: i8,

    ) -> impl for<'a> Fn(&&mut SummonedChampion) -> bool {

        generate\_filter(FilterType::DistanceFilter(distance, self)) //utilises the generate filter function

    }

This takes a location and returns a function that takes in a champion and returns a bool, so a method that returns a function. This can then be used with the .filter() method on an iterator to filter the iterator.

enum FilterType {

    ///i8 : Distance to check

    ///Location : Other Location

    DistanceFilter(i8, Location),

}

///generates a filter based on a given filter type

fn generate\_filter(filter: FilterType) -> impl for<'a> Fn(&&mut SummonedChampion) -> bool {

    match filter {

        FilterType::DistanceFilter(dis, location) => {

            move |n: &&mut SummonedChampion| n.location.distance\_between\_points(&location) < dis

        } //returns a function that moves the given SummonedChampion into the enclosure, and returns a bool depending on whether the distance between a point and the summonedchampion location is low enough

    }

}

Filter type is a enum for all the different possible filter types, at the moment only the distance filter exists.

Generate filter is a function that returns another function, it matches what filter type you gave it and then returns the function.

For FilterType DistanceFilter, it returns an enclosure that moves a mutable reference to a SummonedChampion into the enclosure, then returns whether the distance between points is small enough.

The enclosure can be called at any time when you give it a SummonedChampion and a distance, so can be used with the filter method.

Those are all the methods on Location.

pub struct PlacedChampion {

    ///id of the associated champion

    id: usize,

    ///star level of champion

    star: usize,

    ///items

    items: [u8; 3],

    ///location on board

    location: Location,

    ///option for team

    team : Option<u8>

}

PlacedChampion and SummonedChampion now hold locations instead of an array.

///Struct for champion placed on board in a battle

#[derive(Debug, Clone, Serialize)]

pub struct SummonedChampion {

    ///location

    pub location: Location,

    ///id of associated champion/ placed champion

    of\_champ\_id : usize,

    ///progress of movement before new square, goes up to 10 then moves

    movement\_progress: [i8; 2],

    ///health

    health: f32,

    ///current mana

    cm: i16,

    ///dodge chance in %

    dc: u8,

    ///crit rate in %

    cr: u8,

    ///crit damage

    crit\_damage: f32,

    ///ability mana cost

    mc: i16,

    ///armor

    ar: f32,

    ///magic resist

    mr: f32,

    ///attack damage

    ad: f32,

    ///attacks per second/ attack speed

    attack\_speed: f32,

    ///auto attack range

    ra: i8,

    ///unique id

    id: usize,

    ///cooldown before target chance

    target\_cooldown: i8,

    ///cooldown before auto attacking again

    auto\_attack\_delay: i16,

    ///attack speed modifier from items and effects

    attack\_speed\_modifier: f32,

    ///id of target

    target: usize,

    ///pathfinding target cell

    target\_cells: Location,

    ///Stores all the item IDs the champion is holding.<br />

    ///\*\*Item IDS:\*\*<br />

    ///0 : Null<br />

    ///1  : B.F Sword (+10 Attack Damage)<br />

    ///2  : Needlessly Large Rod (+10 Ability Power)<br />

    ///3  : Giants Belt (+150 health)<br />

    ///4  : Chain Vest (+20 Armor)<br />

    ///5  : Negatron Cloak (+20 Magic Resist)<br />

    ///6  : Recurve Bow (+10% Attack Speed)<br />

    ///7  : \*Sparring Gloves\* (+5% Crit Chance, +10% Dodge Chance)<br />

    ///8  : Tear of the Goddess (+15 Mana)<br />

    ///9  : Spatula<br />

    ///11 : Deathblade (+40, +70, +100 Attack Damage - Star Level Dependent)<br />

    ///12 : \*Hextech Gunblade\* (Dealing Magic and True Damage heals the owner and lowest health ally for 25% of the damage)<br />

    ///13 : Zekes Herald (Grants 30% bonus attack speed to the holder and 2 adjacent allies in same row)<br />

    ///14 : Edge of Night (At 50% health - once per combat - the holder briefly becomes untargetable and sheds negative effects. Then they gain 30% attack speed)<br />

    ///15 : Bloodthirster (Damage dealt heals holder for 25%. Once per combat at 40% Health, gain a 25% maximum health shield for up to 5 seconds)<br />

    ///16 : Giant Slayer (Abilities and attacks deal 25% more damage, increased to 50% if the holder has over 2200 maximum health)<br />

    ///17 : Infinity Edge (+10 Attack Damage, +225% Crit Chance, +10% Crit Damage, Converts every 1% excess critical strike chance into 1% bonus critical strike damage)<br />

    ///18 : Spear of Shojin (Basic attacks restore an additional 8 mana on-attack)<br />

    ///19 : Shimmerscale Emblem (Wearer becomes a shimmerscale, cannot equip on a shimmersclae)<br />

    ///22 : Rabadons Deathcap (+975 Ability Power)<br />

    ///23 : Morellonomicon (+30 Ability Power, magic or true damage from an ability burns the holders target, dealing 25% of the targets maximum health as trude damage over 10 seconds and applying grevious wounds for the duration)<br />

    ///24 : Locket of the Iron Solari (At the start of combat, the wearer and all allies within 2 hexes in the same row gain a 300 / 350 / 400 health shield for 15 seconds - star level dependent)<br />

    ///25 : Ionic Spark (Enemies within 3 hexes have their magic resistance reduced by 50% (does not stack). When enemies within 3 hexes cast their ability, they are dealt 250% of their maximum mana as magic damage)<br />

    ///26 : Guinsoos Rageblade (Basic attacks grant 6% bonus attack speed for the rest of combat, stacks with no upper limit)<br />

    ///27 : \*Jeweled Gauntlet\* (+15% Crit Chance, +40% Crit Damage, +10 Ability Power, The holders magic adn true damage from abilities can critically strike)<br />

    ///28 : Archangels Staff (Grants the wearer 20 ability power every 5 seconds)<br />

    ///29 : Dragonmancer Emblem (Wearer becomes an dragonmancer, cannot equip on an dragonmancer)<br />

    ///33 : Warmogs Armor (+1000 Health)<br />

    ///34 : Sunfire Cape (+400 Health. At the start of combat and every 2 seconds thereafter, applies a 10% maximum health burn as true damage over 10 seconds and applying grevious wounds for the duration)<br />

    ///35 : Zephyr (At the start of combat, banishes for 5 seconds the unit that mirrors the wielders placement on the other side of the board. Pierces through CC immunity effects)<br />

    ///36 : ZZ Rot Portal (At the start of combat, the wearer taunts enemies within 4 hexes. When the wearer dies, a Voidspawn arises, taunting nearby enemies. Summoned units can spawn Voidspawns at 25% effectiveness)<br />

    ///37 : \*Banshees Claw\* (+15% Dodge Chance, +150 Health, At the beginning of each round, the holder and allies within 1 hex in the same row gain a shield that blocks the first enemy ability, up to 600 damage)<br />

    ///38 : Redemption (Every 5 seconds, the wearer radiates an aura to allies within 1 hex, healing them for 12% missing health. Affected allies take 25% reduced damage from AOE attacks for  seconds)<br />

    ///39 : Guardian Emblem (Wearer becomes a guardian, cannot equip on a guardian)<br />

    ///44 : Bramble Vest (+60 Armor. Negatves 75% bonus damage from critical hits. On being hit by an attack, deal 75 / 100 / 150 magic damage to all nearby enemies (once every 2.5 seconds))<br />

    ///45 : Gargoyle Stoneplate (+18 Armor and Magic Resist for each enemy targeting the holder)<br />

    ///46 : \*Titans Resolve\* (Gain 2 attack damage and ability power when attacking or taking damage. After stacking 25 times, gain 25 armor and magic resist and stop stacking)<br />

    ///47 : \*Shroud of Stillness\* (Shoot a beam that delays the first cast of affected enemies by 35%)<br />

    ///48 : Frozen Heart (Reduce the attack speed of enemies within 2 hexes by 25%)<br />

    ///49 : Cavalier Emblem (Wearer becomes a cavalier, cannot equip on a cavalier)<br />

    ///55 : Dragons Claw (+120 Magic Resist, every 2 seconds, regenerate 1.2% maximum health for each enemy targeting the holder. If holder is a dragon, increase all bonuses and effects by 20%)<br />

    ///56 : \*Runaans Hurricane\* (+10 Atttack Damage, attacks fire a bolt at a nearby enemy, dealing 70% of the holders attack damage as physical damage)<br />

    ///57 : \*Quicksilver\* (+20% attack speed. Immune to crowd control for 15 secnds)<br />

    ///58 : Chalice of Power (+30 Ability Power to holder and 2 adjacent allies on same row)<br />

    ///59 : Mirage Emblem (Wearer becomes a mirage, cannot equip on a mirage)<br />

    ///66 : Rapid Firecannon (+50% attack speed and +1 attack range, attacks cannot miss)<br />

    ///67 : \*Last Whisper\* (Dealing physical damage reduces the targets armor by 50% for 5 seconds, does not stack)<br />

    ///68 : Statikk Shiv (+15% attack speed, every 3rd attack shocks enemies for 70 magic damage and reduces their magic resist by 50% for 5 seconds)<br />

    ///69 : Ragewing Emblem (Wearer becomes a ragewing, cannot equip on a ragewing)<br />

    ///77 : \*Thiefs Gloves\* (Each round equip 2 random items, improve with player level, you cannot equip other items)<br />

    ///78 : \*Hand of Justice\* (+15 attack damage, +15% ability power. Attacks and abilities heal for 15% of damage dealt. Each round randomly increase 1 effect by 30%)<br />

    ///79 : \*Assassin Emblem\* (Wearer becomes an assassin, cannot equip on an assassin)<br />

    ///88 : Blue Buff (+20 Starting Mana. Gain 20 mana after casting an ability)<br />

    ///89 : Mage Emblem (Wearer becomes a mage, cannot equip on a mage)<br />

    ///99 : Tacticians Crown (Increase board unit size by 1)<br />

    items: [u8; 3],

    ///ability power

    ap: f32,

    ///vec of status effects

    se: Vec<StatusEffect>,

    ///generate mana delay (can't generate mana 1 secomnd after casting ability)

    gain\_mana\_delay: i16,

    ///star level

    star\_level: usize,

    ///incoming DMG modifier

    incoming\_damage\_modifier: f32,

    ///starting HP

    initial\_hp: f32,

    ///can be targeted or not

    targetable: bool,

    ///needs to shed negative status effects

    shed: u8,

    ///vec of all shields

    shields: Vec<Shield>,

    ///whether zapped from ionic spark

    zap: bool,

    ///whether zenith banished

    banish: bool,

    ///titan's resolve stacks

    titans\_resolve\_stacks: u8,

    ///omnivamp (% of healing from damage done)

    omnivamp: f32,

    ///shiv attack count

    shiv\_attack\_count: u8,

}

SummonedChampion now has a setup method that gives it all the item effects and the initial stats of the Champion it was:

pub fn setup(

        &mut self,

        friendly\_champions: &mut VecDeque<SummonedChampion>,

        enemy\_champions: &mut VecDeque<SummonedChampion>,

        champions : &[Champion],

        items : &[Item]

    ) {

        info!("Setup of Champion {}", self.id);

        {

            //sets stats to initial values specified by Champion.

            let of\_champion = &champions[self.of\_champ\_id];

            self.health = of\_champion.hp;

            self.cm = of\_champion.sm;

            self.ar = of\_champion.ar;

            self.mr = of\_champion.mr;

            self.ad = of\_champion.ad;

            self.attack\_speed = of\_champion.attack\_speed;

            self.ra = of\_champion.ra \* 2;

            self.mc = of\_champion.mc;

        }

        {

            //for each item, give item effect

            for item in self.items {

                info!("Giving item effect {}", item);

                self.give\_item\_effect(item, friendly\_champions, enemy\_champions, items)

            }

        }

        //set initial hp to current health, accounting for items etc

        self.initial\_hp = self.health;

        info!("Set HP to {}", self.health);

    }

It takes in a list of friendly and enemy champions, as well as a reference to an array of champions and items.

In the first block, it sets its stats to the correct base values of the champion. Then it gives itself all the item effects and sets its initial hp to the current health.

///item struct holding base stats and ids

pub struct Item {

    pub id : u8,

    pub health : f32,

    pub ad : f32,

    pub ap : f32,

    pub ar : f32,

    pub mr : f32,

    pub attack\_speed\_modifier : f32,

    pub ra : i8,

    pub cr : u8,

    pub dc : u8,

    pub cm : i16,

    pub omnivamp : f32,

    pub crit\_damage : f32,

}

I created an item struct, holding an id and all the stat changes it has.

fn give\_item\_effect(

        &mut self,

        item: u8,

        friendly\_champions: &mut VecDeque<SummonedChampion>,

        enemy\_champions: &mut VecDeque<SummonedChampion>,

        items : &[Item],

    ) {

        info!("giving item {}", item);

        if item == 0 { return } //no actual item

        let item\_obj = items[(item as usize) - 1]; //fetch item object

        {

            self.health += item\_obj.health;

            self.ad += item\_obj.ad;

            self.ap += item\_obj.ap;

            self.ar += item\_obj.ar;

            self.mr += item\_obj.mr;

            self.attack\_speed\_modifier \*= item\_obj.attack\_speed\_modifier;

            self.ra += item\_obj.ra;

            self.cr += item\_obj.cr;

            self.dc += item\_obj.dc;

            self.cm += item\_obj.cm;

            self.omnivamp += item\_obj.omnivamp;

            self.crit\_damage += item\_obj.crit\_damage;

            //give plain stat buffs

        }

The give item effect method takes a list of items.

If the item is 0, it is a blank item etc, so return, otherwise it fetches the item object and gives the stat increases.

Then we match to the item id and give it any additional effects it has

match item {

11 => {self.ad += item\_obj.ad \* (self.star\_level as f32)}, //increasing ad with star level

            13 => {

                for friendly\_champion in friendly\_champions

                    .iter\_mut()

                    .filter(self.location.get\_within\_distance(3))

                {

                    if friendly\_champion.location.y == self.location.y {

                        friendly\_champion.attack\_speed\_modifier \*= 1.3;

                    }

                } //attack speed buff to all allied champs adjacent on same row

            }

Deathblade gives additional ad with star level.

13 or zeke’s herald gives attack speed boost to adjacent friendly champions on the same row.

We filter friendly champions to get those within the distance and then check if we have the same y value, if so giving the boost.

14 => {

                self.se.push(StatusEffect {

                    duration: Some(0),

                    status\_type: StatusType::EdgeOfNight(),

                    ..Default::default()

                })

            } //gives edge of night buff

            15 => {

                self.se.push(StatusEffect {

                    duration: Some(0),

                    status\_type: StatusType::Bloodthirster(),

                    ..Default::default()

                }); //gives bloodthirster buff

            }

14 and 15 push their associated status effect.

24 => {

                let shield\_amount = [300.0, 350.0, 400.0][self.star\_level];

                self.shields.push(Shield {

                    duration: 1500,

                    size: shield\_amount,

                    ..Default::default()

                });

                for friendly\_champion in friendly\_champions

                    .iter\_mut()

                    .filter(self.location.get\_within\_distance(3))

                {

                    if friendly\_champion.location.y == self.location.y {

                        friendly\_champion.shields.push(Shield {

                            duration: 1500,

                            size: shield\_amount,

                            ..Default::default()

                        });

                    }

                }//gives shield to self and all adjacent allies on same row

            }

24 gives a shield to itself and those on the same row and adjacent to it.

28 => {

                self.se.push(StatusEffect {

                    duration: Some(500),

                    status\_type: StatusType::ArchangelStaff(0.2),

                    ..Default::default()

                })//gives archangel buf

            }

            34 => {

                self.se.push(StatusEffect {

                    duration: Some(0),

                    status\_type: StatusType::GiveSunfire(),

                    ..Default::default()

                }) //gives sunfire buff

            } //(!U)

            35 => {

                self.se.push(StatusEffect {

                    duration: Some(0),

                    status\_type: StatusType::Zephyr(500),

                    ..Default::default()

                })

            } //gives zephyr effect

These three give their associated effect to the champion.

36 => {

                for enemy\_champion in enemy\_champions

                    .iter\_mut()

                    .filter(self.location.get\_within\_distance(9))

                {

                    enemy\_champion.se.push(StatusEffect {

                        duration: Some(0),

                        status\_type: StatusType::Taunted(self.id),

                        is\_negative: true,

                        ..Default::default()

                    }) //taunts all enemies within range

                }

            }

This effect taunts all enemy champions within 4.5 hexes.

37 => {

                self.shields.push(Shield {

                    duration: 1500,

                    size: 600.0,

                    blocks\_type: Some(DamageType::Magical()),

                    pop: true,

                }); //gives self banshee shield

                for friendly\_champion in friendly\_champions

                    .iter\_mut()

                    .filter(self.location.get\_within\_distance(3))

                {

                    if friendly\_champion.location.y == self.location.y

                    //gives banshee's shield

                    {

                        friendly\_champion.shields.push(Shield {

                            duration: 1500,

                            size: 600.0,

                            blocks\_type: Some(DamageType::Magical()),

                            pop: true,

                        });

                    }

                }

            }

This gives a shield to itself and allied champions adjacent on the same row.

38 => {

                self.se.push(StatusEffect {

                    duration: Some(100),

                    status\_type: StatusType::RedemptionGive(),

                    ..Default::default()

                })

            } //Gives redemption effect

            45 => {

                self.se.push(StatusEffect {

                    duration: Some(0),

                    status\_type: StatusType::Gargoyles(0.0),

                    ..Default::default()

                }) //gives gargoyles effect

            }

            46 => {

                self.se.push(StatusEffect {

                    duration: Some(0),

                    status\_type: StatusType::TitansResolve(0),

                    ..Default::default()

                }) //gives titans resolve effect

            }

            47 => {

                self.se.push(StatusEffect {

                    duration: Some(0),

                    status\_type: StatusType::ShroudOfStillness(),

                    ..Default::default()

                }) //gives shroud of stillness effect

            }

            48 => {

                self.se.push(StatusEffect {

                    duration: Some(0),

                    status\_type: StatusType::ProtectorsVow(),

                    ..Default::default()

                }) //gives protectors vow effect

            }

            55 => {

                self.se.push(StatusEffect {

                    duration: Some(200),

                    status\_type: StatusType::DragonClawHeal(),

                    ..Default::default()

                }) //gives dragon claw heal

            }

            57 => {

                self.se.push(StatusEffect {

                    duration: Some(15000),

                    status\_type: StatusType::CrowdControlImmune(),

                    ..Default::default()

                }); //gives cc immunity for 15 seconds

            }

More that just give a status effect.

58 => {

                for friendly\_champion in friendly\_champions

                    .iter\_mut()

                    .filter(self.location.get\_within\_distance(3))

                {

                    if friendly\_champion.location.y == self.location.y

                    {

                        friendly\_champion.ap += 0.3;

                    }

                } //gives ap to friendly champs adjacent in same row

            }

Then, an item that gives an ap boost to all allied champions adjacent on the same row.

78 => {

                //50% chance to give bonus ad ap, 50% chance to give bonus omnivamp

                if rand::thread\_rng().gen\_range(0..100) > 50

                {

                    self.ad += 30.0;

                    self.ap += 0.3;

                    self.omnivamp += 0.15;

                } else {

                    self.ad += 15.0;

                    self.ap += 0.15;

                    self.omnivamp += 0.3;

                }

            }

Before finally, an item that gives extra ad, ap or omnivamp depending on a coinflip.

The code for give item effects now is a lot shorter due to the item obj and being able to give all stat increases in a block at the start rather than scattered throughout. Moreover, only requiring item objects during the setup means that ids can be used throughout the rest of the program, using less much less memory than the whole object.

On the status effect class, there is now an implementation for PartialEqual:

impl PartialEq for StatusEffect {

    fn eq(&self, other: &Self) -> bool { //checks whether two status effects share a status type

        self.status\_type == other.status\_type

    }

}

This allows me to use operations to compare two status effects directly via their status types.

So for example in the heal method on SummonedChampion, rather than iterating through each status effect and checking its status type directly now, I can just use the contains method for vectors:

if self.se.contains(&StatusEffect {

            status\_type: StatusType::GreviousWounds(),

            ..Default::default()

        })

Where the only field I need to define is the status type.

There have been a number of improvements to the take turn function for summonedchampion.

To increase robustness, all of the timers/ cooldowns/ delays check for underflow errors:

self.target\_cooldown = self.target\_cooldown.checked\_sub(time\_unit).unwrap\_or(-1); //Reduce cooldown to check target/ find new target

        self.auto\_attack\_delay = self

            .auto\_attack\_delay

            .checked\_sub(time\_unit as i16)

            .unwrap\_or(-1); //Risks going out of bounds as auto attack value may not be called for some time, so checked subtraction

        self.gain\_mana\_delay = self

            .gain\_mana\_delay

            .checked\_sub(time\_unit as i16)

            .unwrap\_or(-1);

Unwrapping into -1 if it fails. As these values are only checked to see if they are < 0, there is no difference if the value is -1 or -100.

{

            info!("Simulating status effects");

            let mut status\_effects = take(&mut self.se); //takes status effect vec

            let mut stun = Stun { stun: 0 }; // setting stun to 0

            status\_effects.retain\_mut(|x| {

                self.perform\_status(x, friendly\_champions, enemy\_champions, time\_unit, &mut stun)

            }); //perform status for each status effect

            if self.health <= 0.0 {

                info!("Health below zero from status effect, removing");

                //died from status effect

                return false;

            }

            self.se.extend(status\_effects); //extend se by status effects

            //do NOT set self.se = status\_effects, as new status may have been added

            self.update\_shed(); //updates shed

            self.shields.retain\_mut(|x| x.update\_shield(time\_unit)); //updates all shields

            if stun.stun == 1 {

                info!("Is stunned");

                //stunned

                return true;

            }

        }

Unlike in the previous status block, which cloned self.se into the new status effect variable, I’ve improved it via using the std::mem::take method, which rather than cloning the entire vector which is expensive, simply moves the memory to a different variable, replacing it with an empty vector. This improves the performance of the operation by a vast margin.

Like how it now removes the SummonedChampion taking its turn from its vector, we now also remove the target object to simply certain areas of code.

let mut need\_new\_target\_cell: bool = false; //bool to store whether new path is needed

            let mut target\_object: Option<SummonedChampion> = None;

if self.target\_cooldown >= 0 {

                info!("Cooldown above zero, trying to find target {}", self.target);

                //if already has target and doesnt want to change targets

                if let Some(index) =

                    find\_champion\_index\_from\_id\_targetable(enemy\_champions, self.target) //try to find target

                {

                    target\_object = enemy\_champions.swap\_remove\_back(index);

                    //target found

                    info!("Target found? : {}", target\_object.is\_some());

                }

            }

Like in prototype 1, here we create a need new target cell bool and a target\_object which holds an option of a SummonedChampion.

We check if we already have a target and if we can find it, and if so we remove it from the vector, using swap\_remove\_back as it is an O(1) operation (that removes the item at index n and moves the back item to that position), this avoids the O(n) remove operation as it does not have to move all the items in the list that come after it.

if target\_object.is\_none() {

                //target\_object not found

                info!("Could not find target or need new target");

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

                need\_new\_target\_cell = true; //tells us to recalculate pathfinding later

                let mut index: Option<usize> = None;

                if let Some((i, champ)) = self

                    .location

                    .get\_closest\_to\_location\_targetable\_index(enemy\_champions) {

                    index = Some(i);//get closest to location that is targetable

                }

                if index.is\_none() {

                    //no targetable champions, ending turn

                    info!("No targetable champions, ending turn");

                    return true;

                }

                target\_object = enemy\_champions.swap\_remove\_back(index.unwrap());

            }

Afterwards, if we haven’t found our target, we look for a new one, resetting our cooldown. We use the get closest to location targetable method on our location to fetch an index.

If there isn’t one, there is not a targetable champion, so we end our turn, otherwise, we fetch the target object from the vector using the swap back remove.

let mut target\_object: SummonedChampion = target\_object.unwrap();

We unwrap our target, getting the SummonedChampion underneath. We know it definitely is an object otherwise our turn would have ended.

self.target = target\_object.id; //set target

            let distance\_to\_target = self

                .location

                .distance\_between\_points(&target\_object.location); //get distance to target

We update our target and fetch the distance to it.

if distance\_to\_target <= self.ra {

                //if target in range

                info!("Target in range, attacking or reducing auto attack cooldown");

                info!("Auto Attack Delay Remaining {0}", self.auto\_attack\_delay);

                if self.auto\_attack\_delay <= 0

Then, we check if our target is in range, and if our auto attack is ready.

self.auto\_attack\_delay =

                        ((100.0 / (self.attack\_speed \* self.attack\_speed\_modifier)) as i16).max(20); //calculating auto attack delay

                    info!("Auto attack delay set to {}", self.auto\_attack\_delay);

                    if self.items.contains(&26) {

                        self.attack\_speed\_modifier \*= 1.06;

                        info!("Increasing speed with Rageblade")

                    }

                    if self.gain\_mana\_delay <= 0 {

                        //if can gain mana

                        self.cm += 10;

                        info!("Gaining mana");

                        if self.items.contains(&18) {

                            self.cm += 8;

                            info!("Additional mana from shojin");

                        }

                        info!("Current mana {}", self.cm);

                    }

If so, we set our auto attack delay to the calculated value, or 20, whatever is larger as the maximum number of auto attacks is 5 per second.

We check if we have guinsoo’s rageblade, increasing our attack speed modifier if so, and then check if we can gain mana, gaining 10 or 18 if we do.

self.shiv\_attack\_count += 1; //increase shiv count

                    if self.items.contains(&68) && self.shiv\_attack\_count == 3 {

Here, we increase the shiv attack count and check if we have the static shiv item. The stattik shiv item, on every 4th auto attack, shoots a burst of lightning that hits 4 enemies.

So if this is the 4th auto attack and we have stattik shiv:

self.deal\_damage(

                            friendly\_champions,

                            &mut target\_object,

                            50.0,

                            DamageType::Magical(),

                            false,

                        );

                        target\_object.se.push(StatusEffect {

                            duration: Some(500),

                            status\_type: StatusType::ShredMagicResist(2.0),

                            is\_negative: true,

                            ..Default::default()

                        });

We deal the magic damage to the opponent, as well as the magic resist shred that it also applies.

for (i, enemy\_champ) in enemy\_champions.iter\_mut().enumerate() {

                            self.deal\_damage(

                                friendly\_champions,

                                enemy\_champ,

                                50.0,

                                DamageType::Magical(),

                                false,

                            );

                            enemy\_champ.se.push(StatusEffect {

                                duration: Some(500),

                                status\_type: StatusType::ShredMagicResist(2.0),

                                is\_negative: true,

                                ..Default::default()

                            });

                            if i > 2 {

                                break;

                            }

                        }

                    }

We then deal the damage and push the effect to a maximum of three enemy champions, breaking if we go over the limit.

After the block, we apply the remainder operator onto self.shiv attack count

self.shiv\_attack\_count %= 3; //make sure shiv count sticks between 0 to 3

if self.items.contains(&56) {

                        //doing runaan's second auto attack

                        info!("Has runaan's, performing second auto");

                        warn!("Runaan's can be dodged, treated like normal auto");

                        let closest\_other\_enemy = self

                            .location

                            .get\_closest\_to\_location\_targetable(enemy\_champions); //fetch closest other champ

                        if let Some(target) = closest\_other\_enemy {

                            self.deal\_damage(

                                friendly\_champions,

                                target,

                                self.ad \* 0.7,

                                DamageType::Physical(),

                                false,

                            ) //do damage

                        }

                    }

Then we check if we have runaan’s hurricane, which deals a second auto attack.

We try to find the nearest other targetable champion and if there is one, we deal the auto attack damage.

if target\_object.dc == 0

                        || target\_object.dc < rand::thread\_rng().gen\_range(0..100)

                        || self.items.contains(&66)

                    //calculating whether to dodge

                    {

                        //no dodge

                        info!("Not dodged");

                        self.deal\_damage( //deal auto damage

                            friendly\_champions,

                            &mut target\_object,

                            self.ad,

                            DamageType::Physical(),

                            false,

                        );

                        info!("Enemy Champion Health is {0}", target\_object.health);

                    } else {

                        info!("Dodged Attack");

                    }

We then do the normal auto attack, after checking for a dodge (or if we have rapidfire cannon, which mean our auto attacks can’t miss).

enemy\_champions.push\_back(target\_object);

We then push our target back.

else {

                info!("Not in Range");

                if need\_new\_target\_cell || self.location == self.target\_cells {

Now we go to the else block where our target is not in range.

We check if we need a new target cell or if we’ve arrived at it.

self.target\_cells = self.location; //setting target cells to location so if it does not find a target this frame will try to do it again

                    let mut lowest\_distance: i8 = i8::MAX; //setting lowestDistance to high value

                    let mut new\_position;

                    for possible\_move in [[0, -1], [1, -1], [1, 0], [-1, 0], [-1, 1], [0, 1]]

                    //for every possible move

                    {

We set out target cell to our location, so we don’t move if we can’t find a possible move, we initialise lowest distance to a high value and new position to an unset value.

We then iterate through all possible moves.

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

                        let distance\_to\_target = target\_object

                            .location

                            .distance\_between\_points(&new\_position);

                        if distance\_to\_target < lowest\_distance {

                            if (!new\_position.check\_valid())

                                || friendly\_champions

                                    .iter()

                                    .any(|f| f.location == new\_position)

                            {

                                continue;

                            }

                            //if distance lower, position valid and there is no other friendly champion in cell

                            info!("Found target cell {}", new\_position);

                            lowest\_distance = distance\_to\_target;//set new target cell

                            self.target\_cells = new\_position;

                        }

                    }

We calculate our new position, adding the possible move vec to our location. We then calculate the distance to our target and check if its smaller.

If the distance is smaller, we check if the position is valid or if there are any collisions.

We check if there are any collisions by turning friendly\_champions into an iterable, then seeing if any of their positions match. If there is a collision, we go to the next possible move, otherwise we update the lowest distance and target cells.

After we have found our target cell (or if we had it already) we move:

self.movement\_progress[0] +=

                    movement\_amount \* sign(self.target\_cells.x - self.location.x); //add movement progress

                info!(

                    "Position ({0:?}) -- Movement Progress ({1:?})",

                    self.location, self.movement\_progress

                );

                if self.movement\_progress[0].abs() >= 10 {

                    //move self

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

                    self.movement\_progress[0] = 0;

                }

                self.movement\_progress[1] +=

                    movement\_amount \* sign(self.target\_cells.y - self.location.y);

                if self.movement\_progress[1].abs() >= 10 {

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

                    self.movement\_progress[1] = 0;

                }

                enemy\_champions.push\_back(target\_object); //return target

We add our movement progress to the two directions, using the sign function to calculate a direction and distance. As sign returns 0 if the argument it is given is 0, we do not move if target cell is our location, avoiding collisions.

Once we have done our movement, we push back our target object to the enemy champions vector.

if self.cm >= self.mc {

            //cast ability

            info!("Enough mana casting ability");

            if self.zap {

                //zap from ionic spakr

                info!("Zap");

                self.health -= (self.mc as f32) \* 2.5;

            }

            self.cm = 0;

            if self.items.contains(&88) {

                info!("Bluebuff");

                self.cm = 20;

            }

            self.gain\_mana\_delay = 100;

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

        }

        true

We then cast our ability if we have enough mana and reset our mana to 0 or 20.

Most of the logic for take turn is the same as prototype 1, but with new items implemented and lots of improvements in the code, ensuring less crashes and faster processing.

However, there is no changes to the cast ability method from prototype 1.

There are a few status effects that require counting the number of enemys targeting this champion, such as Gargoyles status effect, so I made a method encapsulating the behaviour.

pub fn get\_num\_targeting(&self, enemy\_champions: &VecDeque<SummonedChampion>) -> usize {

        enemy\_champions

            .iter()

            .filter(|p| p.target == self.id)

            .count() //counts the number of enemy champions with target equal to id

    }

It takes in a list of enemy champions, turns them into an iterable, filters out the ones that do not have us as a target, then counts them.

As an example, here is now the updated give Gargoyles effect:

StatusType::Gargoyles(old\_num\_targeting) => {

                        n\_duration = 100; //increase duration

                        let num\_targeting: f32 = self.get\_num\_targeting(enemy\_champions) as f32;

                        let difference = num\_targeting - old\_num\_targeting; //get change

                        self.ar += 0.18 \* difference;

                        self.mr += 0.18 \* difference;

                        status\_effect.status\_type = StatusType::Gargoyles(num\_targeting);

                    }

Utilising the self.get\_num\_targeting method.

/// retrieve whether the champion is targetable

    pub fn get\_is\_targetable(&self) -> bool {

        self.targetable && !self.banish

    }

We also now have a get is targetable method, returning whether the champion is able to be hit.

The game loop code on the board class has not changed much, other than now possessing a dead champions double ended vec.

if alive {

                    //if alive push to back of p1\_champs

                    self.p1\_champions.push\_back(this\_champ);

                } else {

                    //else push to dead champs

                    self.dead\_champs.push\_back(this\_champ);

                }

If a champion dies during its turn (which is when deaths are checked for) and the function returns false, to signify that, it is pushed to the list of dead champions.

This allows projectiles to be able to find the champion that created them no matter what, allowing item effects to be correctly given. For instance, if a champion with gunblade shoots a projectile, the heal from the damage still goes through to its teammates, even if the champion itself dies.

self.p1\_projectiles.retain\_mut(|f| {

                f.simulate\_tick(

                    &mut self.p2\_champions,

                    &mut self.p1\_champions,

                    &mut self.dead\_champs,

                )

            });

Simulate tick for projectiles takes in this list of dead champions for that purpose. As there is not a separate list of dead champions for p1 and p2 champions, how IDs are given to SummonedChampions had to be changed. In the past, ids were given based on its index in the list of that team’s champions. So there could be multiple summoned champions with the same id, but they were on different teams. That approach would no longer work, as there would be no way to differentiate between teams in the dead champs vector. Now, ids given for player 1 champs work the same, but for player 2 champs, the number of p1 champs is added onto the index:

let len: usize = p1\_placed\_champs.len();

        info!("Creating Champions");

        //for each champ in p1's placed champ, push a summoned champion to p1\_champions

        for (i, p1\_champ) in p1\_placed\_champs.iter().enumerate() {

            p1\_champions.push\_back(SummonedChampion::new(p1\_champ, i)); //converts into summoned champ

        }

        //repeat for player 2

        for (i, p2\_champ) in p2\_placed\_champs.iter().enumerate() {

            //adds length of p1\_placed\_champs to id to ensure they have unique id.

            p2\_champions.push\_back(SummonedChampion::new(p2\_champ, i + len)); //converts into summoned champ

        }

This ensures that all ids are unique on both teams. This code is taken from the “new” method for the board class.

So this allows simulate tick to access the shooter, which in turn gives it access to a SummonedChampion object with the “deal\_damage” method. One thing not accounted for in prototype 1 was projectiles accounting for resistances while dealing damage, with this solution, that is now fixed.

let target\_location = match self.target\_location //if self has target location, set target location to that, else get the location of the target champion.

        {

            Some(location) => {

                info!("Target {location}");

                location}, //gets target location

            None => {

                let out\_location = find\_champion\_index\_from\_id(possible\_targets, self.target\_id);//gets location of target champion

                info!("Finding location from id : {:?}", out\_location);

                match out\_location

                {

                    Some(index) => possible\_targets[index].location, //if target is still alive, return its location

                    None => Location { x: -1, y: -1 }, //set location to invalid

                }

        }};

        if target\_location.x == -1 {

            info!("Not valid location, removing");

            return false;

        } //not found, remove projectile

The start of simulate tick for projectiles is the same, if the target is a location, it fetches that location, otherwise it fetches the location of its target.

let subtracted\_distance = Location::sub\_positions(&target\_location, &self.location); //get location difference

        self.location\_progress[0] += self.speed \* sign(subtracted\_distance.x);

        self.location\_progress[1] += self.speed \* sign(subtracted\_distance.y); //add location progress

For movement, we take advantage of Location’s subtract position method, subtracting our target location with our current one, and adding that to the location progress.

if self.location\_progress[0].abs() >= 10

        //if above 10, move

        {

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

        }

        if self.location\_progress[1].abs() >= 10 {

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

        }

We then check to see if our movement progress is enough to move, and if so, move.

if !self.location.check\_valid() {

            info!("Out of grid leaving");

            return false;

        } //if out of grid, remove

We then ensure that we are still in the grid, removing ourselves from the vector if not.

After that, we check for collisions.

for possible\_target in possible\_targets.iter\_mut()

        //iterate through all possible collisions

        {

            if self.location == possible\_target.location

            //has a hit

            {

We try to locate the shooter

let mut dead = false; //stores whether need to add to dead champions or alive

                let mut shooter: SummonedChampion;

                //if shooter alive, fetch from friendly champions, else fetch from dead champions, this is because deal damage requires damage dealer to apply correct effects

                if let Some(shooter\_index) =

                    find\_champion\_index\_from\_id(friendly\_champions, self.shooter\_id)

                {

                    //finds shooter id

                    shooter = friendly\_champions.swap\_remove\_back(shooter\_index).unwrap();

                    info!("shooter alive");

                } else {

                    let dead\_champion\_index =

                        find\_champion\_index\_from\_id(dead\_champions, self.shooter\_id).unwrap(); //fetch from dead champions

                    shooter = dead\_champions.swap\_remove\_back(dead\_champion\_index).unwrap();

                    dead = true; //remember to add to dead champions later

                    info!("shooter dead")

                }

Either we find it from the alive champions or the dead ones, if it is dead, we set the variable “dead” to true, to make sure we put it back into the dead champions list later.

shooter.deal\_damage(

                    friendly\_champions,

                    possible\_target,

                    self.damage,

                    self.damage\_type,

                    false,

                ); //deals damage

We then deal the damage.

if self.splash\_damage > 0.0

                //if there is splash damage

                {

let initial\_hit = possible\_target.id;

                    info!("dealing splash");

                    for possible\_secondary\_target in possible\_targets

                        .iter\_mut()

                        .filter(self.location.get\_within\_distance(3))

                    //iterate through possible splash hits

                    {

if possible\_secondary\_target.id == initial\_hit { continue }

                        shooter.deal\_damage(

                            friendly\_champions,

                            possible\_secondary\_target,

                            self.splash\_damage,

                            self.damage\_type,

                            true,

                        ); //deal secondary dmg

                    }

                }

Then, if there’s splash damage, we check all adjacent tiles for enemy champions, dealing damage to those as well, but checking to make sure it isn’t the same as we originally hit, which would do double damage.

if !dead {

                    friendly\_champions.push\_back(shooter) //push to alive

                } else {

                    dead\_champions.push\_back(shooter) //push to dead

                }

                return false; //has exploded, so return false

We then push our shooter back into the correct respective vector.

After we have checked for collisions, we check to see if the projectile has reached its target location:

if self.target\_location.is\_some() && self.target\_location.unwrap() == self.location {

            return false;

        }

If so, we should remove ourselves, as there was not a collision there.

true //still alive

Otherwise we just return true.

Finally, one big improvement from prototype 1 was the implementation of draws, the board struct now has a field:

/// Number of ticks until the battle is declared a draw.

    ticks\_till\_draw : u32,

which must be specified when creating a new one, which is the total number of ticks the battle should last before it is declared a draw. Drawing from feedback from my client from prototype 1, I also added the functionality of, rather than simulating the entire battle at once, simulating just a chunk of it:

pub fn simulate\_battle(&mut self, ticks\_to\_simulate : Option<u32>) {

        info!("Starting Battle");

        let upper = match ticks\_to\_simulate {

            Some(cnt) => self.ticks\_till\_draw.min(self.tick\_count + cnt), //simulates battle till draw or for the tick count, whatever comes first.

            None => self.ticks\_till\_draw //if none simnulates entire battle till draw

        };

        //for each tick

        for \_ in self.tick\_count..upper {

            info!("Battle Iteration : {}", self.tick\_count);

            self.tick\_count += 1; //increment ticks

You can now give an option of ticks to simulate when simulating a battle, if you give None, it will simulate the battle until there is a winner or a draw, however, if you give it ticks to simulate, it will go to either a draw or to the number of ticks you asked it to simulate (which is the current tick count added to the number you wanted simulated), whatever comes first, to ensure that it does not go over the upper limit specified by ticks till draw.

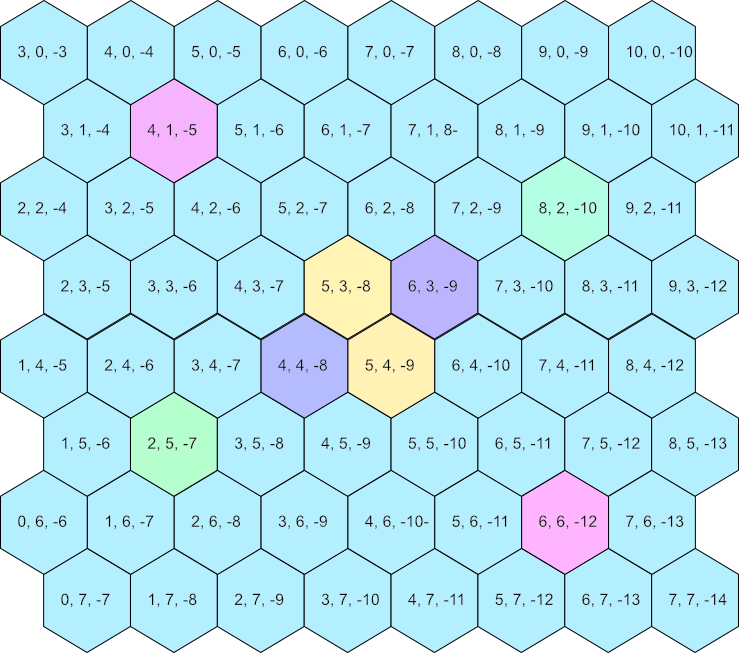
We keep track of the current tick count by the attribute self.tick\_count, and then in the game loop, we go from the current tick count to the upper limit calculated earlier, increasing our tick count by 1 each game loop.

This allows the board to be simulated in chunks and allows the user to view the status of the battle before it has finished/ half way through, like requested by my client Suket Arya in the feedback to prototype 1.

That is an overview of all the changes, improvements and additions made in prototype 2.

### Errors and Solutions:

**Zephyr Targeting (logic error):**

Zephyr targets the “opposite” cell on the board and banishes the nearest champion to that cell as shown in the colour coded example below:

In prototype 1, the logic to calculate the opposite cell was simply swapping the x and y value. A simple counterexample for this logic is the cell 6, 6, -12, as the x and y values swapped just, equal themselves, so which is obviously not the opposite cell.

**Solution:**

To calculate the opposite x and y value, we need to figure out the distance between it and one of the walls of the grid, and then the opposite cell will have that distance to the other wall on the grid, the problem is that the coordinates of the walls of the grid changes as you move through it, at least for x values.

For y values, it is fairly simple, the walls are at 0 and 7, so to calculate the opposite y value, we can do oppositeY = 7 – currentY, which works, as you can see for the examples such as 4,1, -5 and 6, 6, -12.

The coordinates of the x boundaries are dependent on the y value, they are, left : 3 – y // 2, right : 10 – y // 2.

So, to figure out the opposite x cell, we can do:

currentX – leftWall = rightWall – oppositeX

currentX – (3 – y // 2) = 10 – y // 2 – oppositeX

currentX = 10 – y // 2 – oppositeX + 3 – y // 2

currentX = 13 – 2(y//2) – oppositeX, which is equal to

currentX – 13 – y – y % 2 – oppositeX, rather than dividing by 2 and taking the floor of it, then multiplying by two, we can just take the y value and subtract its remainder when divided by 2, ie subtracting 1 if it is an odd number.

Which will work for the examples given above and gives us the following code:

let opposite\_location = Location {

                            x: 13 - self.location.y - (self.location % 2) - self.location.x,

                            y: 7 - self.location.y,

                        }; //gets opposite location

                        opposite\_location

                            .get\_closest\_to\_location(enemy\_champions)

                            .unwrap()

                            .se

                            .push(StatusEffect {

                                duration: Some(banish\_duration),

                                status\_type: StatusType::Banished(),

                                ..Default::default()

                            }); //banishes champ closest to that location

**Find Champion Index from ID Error (index error):**

As id comes from index, in the utility function find\_champion\_index\_from\_id

pub fn find\_champion\_index\_from\_id(

    champions: &VecDeque<SummonedChampion>,

    id: usize,

) -> Option<usize> {

I initially checked to see if the id of the champion at index id was correct:

if champions[id].equal\_id(id) {

        info!("found from index");

        return Some(id);

    }

Where equal id is a simple function to compare ids:

pub fn equal\_id(&self, id: usize) -> bool {

        self.id == id //checks if id equal

    }

The problem with this is that it did not check the bounds of the vector, and so would have numerous index/ out of bounds errors when champions died, reducing the size of the vector. To get around it, I initially checked if id was smaller than the length of champions:

if id < champions.len() && champions[id].equal\_id(id) {

        info!("found from index");

        return Some(id);

    }

Making the function:

pub fn find\_champion\_index\_from\_id(

    champions: &VecDeque<SummonedChampion>,

    id: usize,

) -> Option<usize> {

    info!("finding champ from id");

if champions[id].equal\_id(id) {

        info!("found from index");

        return Some(id);

    }

    //else checks every champion for the id

    for (i, champ) in champions.iter().enumerate() {

        if champ.equal\_id(id) {

            info!("found from id");

            return Some(i);

        }

    }

    None

}

However, after the change to how ids are assigned, with champions in team 2 having their ids extended/ increased by the length of p1 champions, as the optimisation would only work for team 1 champions, I deemed it unnecessary and removed it, leaving the function as:

pub fn find\_champion\_index\_from\_id(

    champions: &VecDeque<SummonedChampion>,

    id: usize,

) -> Option<usize> {

    info!("finding champ from id");

    //else checks every champion for the id

    for (i, champ) in champions.iter().enumerate() {

        if champ.equal\_id(id) {

            info!("found from id");

            return Some(i);

        }

    }

    None

}

I made the same alterations to find\_champion\_index\_from\_id\_targetable.

**Projectile not removing itself at target location (logic error):**

When a projectile moves to a target location, if it doesn’t have a collision, it will just stick in the target location, because as the difference between the target location and its current location is 0, sign of 0 will return 0, ensuring there is no movement, this result in the projectile standing there waiting for something to enter the tile, hogging resources and not acting as intended.

**Solution:**

After checking for collisions, if it has reached its target location, it should return false, removing itself from the vector/ deleting itself.

if self.target\_location.is\_some() && self.target\_location.unwrap() == self.location {

            return false;

        }

        true //still alive

**Splash damage not accounting for resistances:**

When dealing damage projectiles in prototype 1 didn’t account for resistances.

**Solutions:**

To fix this, I added the dead champions list so the projectile could always find its shooter, giving it access to the deal damage method on SummonedChampion that accounts for resistances as well as items on the champion itself.

**Splash damage does double damage (logic error):**

Splash damage applied to the champion initially hit by the projectile, when it shouldn’t be.

**Solution:**

When checking for splash damage collisions, make sure that the id of the splash damage champion isn’t the same as the one initially hit.

if self.splash\_damage > 0.0

                //if there is splash damage

                {

                    let initial\_hit = possible\_target.id;

                    info!("dealing splash");

                    for possible\_secondary\_target in possible\_targets

                        .iter\_mut()

                        .filter(self.location.get\_within\_distance(3))

                    //iterate through possible splash hits

                    {

                        if possible\_secondary\_target.id == initial\_hit { continue }

                        shooter.deal\_damage(

                            friendly\_champions,

                            possible\_secondary\_target,

                            self.splash\_damage,

                            self.damage\_type,

                            true,

                        ); //deal secondary dmg

                    }

                }

**Checking for banish (logic error):**

I had an error where I would check:

if self.banish {

            //banished

            info!("Is banished");

            return true;

        }

Before handling status effects, this caused an issue because as self.banish is defined by a status effect, as it would banish/ end turn before it got the chance to update status effects, it would essentially be permanently banished, as the status would never update/ reduce in duration.

**Solution:**

Check for self.banish after updating status effects.

{

            info!("Simulating status effects");

            let mut status\_effects = take(&mut self.se); //takes status effect vec

            let mut stun = Stun { stun: 0 }; // setting stun to 0

            status\_effects.retain\_mut(|x| {

                self.perform\_status(x, friendly\_champions, enemy\_champions, time\_unit, &mut stun)

            }); //perform status for each status effect

            if self.health <= 0.0 {

                info!("Health below zero from status effect, removing");

                //died from status effect

                return false;

            }

            self.se.extend(status\_effects); //extend se by status effects

            //do NOT set self.se = status\_effects, as new status may have been added

            self.update\_shed(); //updates shed

            self.shields.retain\_mut(|x| x.update\_shield(time\_unit)); //updates all shields

            if stun.stun == 1 {

                info!("Is stunned");

                //stunned

                return true;

            }

        }

        if self.banish {

            //banished

            info!("Is banished");

            return true;

        }

**Get Closest to location targetable (logic error):**

Get closest to location targetable index used to have an error in how it dealt with filtering.

pub fn get\_closest\_to\_location\_targetable\_index<'a>(

        &self,

        enemy\_champions: &'a mut VecDeque<SummonedChampion>,

    ) -> Option<(usize, &'a mut SummonedChampion)> {

        enemy\_champions

            .iter\_mut() //turn into iterator

            .filter(|(\_, x)| x.get\_is\_targetable()) //remove any champions that are not targetable

            .enumerate()

            .reduce(|(i, x), (j, y)| { //reduce by distance to point

                if x.location.distance\_between\_points(self)

                    < y.location.distance\_between\_points(self)

                {

                    return (i, x);

                }

                (j, y)

            })

    }

It used to filter the champions before enumerating them, this would cause issues as the index would not be correct to its position in the vector. If a champion was the third item in the list, but the first item that was targetable, its “index” would be 0, as enumerate would only count the targetable objects.

**Solution:**

pub fn get\_closest\_to\_location\_targetable\_index<'a>(

        &self,

        enemy\_champions: &'a mut VecDeque<SummonedChampion>,

    ) -> Option<(usize, &'a mut SummonedChampion)> {

        enemy\_champions

            .iter\_mut() //turn into iterator

            .enumerate() //get the indexes of the champions, BEFORE the filter so that indexes are valid/ accurate to given vector

            .filter(|(\_, x)| x.get\_is\_targetable()) //remove any champions that are not targetable

            .reduce(|(i, x), (j, y)| { //reduce by distance to point

                if x.location.distance\_between\_points(self)

                    < y.location.distance\_between\_points(self)

                {

                    return (i, x);

                }

                (j, y)

            })

    }

Move enumerate to before filter, so it counts all of them regardless of whether it is targetable, so it returns the correct position in the list.

**Item ID != Index (logic error):**

In the give item\_effect method, it tries to get the item\_obj using the id as an index, but as there are some indexes that do not correlate to an item (such as 9, 19) it doesn’t work.

**Solution:**

Try to locate the item by id using a for loop, returning if it can’t find it:

fn give\_item\_effect(

        &mut self,

        item: u8,

        friendly\_champions: &mut VecDeque<SummonedChampion>,

        enemy\_champions: &mut VecDeque<SummonedChampion>,

        items : &[Item],

    ) {

        info!("giving item {}", item);

        if item == 0 { return } //no actual item

        let mut item\_obj = None;

        for item\_n in items {

            if item\_n.id == item {

                item\_obj = Some(item\_n);

            }

        }

        if item\_obj.is\_none() { return } //can't find item

        let item\_obj = item\_obj.unwrap();

### Test Table:

Testing specific boards was accomplished by the following code:

use super::champions::DEFAULT\_CHAMPIONS;

use super::item::DEFAULT\_ITEMS;

fn perform\_test() {

    let p1\_champions = VecDeque::from([PlacedChampion::new(3, 1, [3, 12, 55], Location { x : 3, y : 0})]);

    let p2\_champions = VecDeque::from([PlacedChampion::new(1, 0, [16, 27, 36], Location { x : 7, y : 7})]);

    let mut outcomes : [u16 ; 3] = [0, 0, 0];

    for \_ in 0..100 {

        let mut board = Board::new(&p1\_champions, &p2\_champions, &DEFAULT\_CHAMPIONS, &DEFAULT\_ITEMS, 10, 10000);

        board.simulate\_battle(None);

        outcomes[board.get\_winner() as usize] += 1;

    }

    println!("{outcomes:?}");

}

Where we import our champions and items at the top, the define a perform test function, which creates the placed champions vecdeque for player 1 and 2.

We then define an outcomes array, and initialise it to 0, 0, 0. Then, 100 times we create the new board, simulate the battle to completion or a draw and then get the winner and add 1 to that outcome, where board.get\_winner() is a simple method that returns the winner of a board:

pub fn get\_winner(&self) -> i8 {

        if self.p1\_champions.is\_empty() {

            return 2

        }

        if self.p2\_champions.is\_empty() {

            return 1

        }

        return 0

    }

If there are no player 1 champions remaining, player 2 is the winner, if there are no player 2 champions remaining, player 1 is the winner, otherwise both players have champions remaining, so the outcome is a draw.

We turn the i8 it returns to a usize so it can index the outcomes array, and add one to that specific outcome, before finally printing the final outcomes of the battle, so I can measure its accuracy.

**Validation of accuracy:**

|  |  |  |  |
| --- | --- | --- | --- |
| Player 1 Champions | Player 2 Champions | Expected Outcome | Simulated Accuracy |
| 3S1[3, 12, 55] | 1S0[16, 27, 36] | Player 1 Winner | 95% |
| 1S0[33, 67], 1S0[25] | 2S1 | Player 1 Winner | 100% |
| 1S2 | 1S3 | Player 2 Winner | 97% |
| 1S0, 1S1[48], 1S2, 1S3[44] | 3S3, 3S1[67] | Player 2 Winner | 100% |
| 2S2 | 3S3 | Player 2 Winner | 100% |
| 3S1, 3S1[88] | 3S3, 3S3[56], 3S1 | Player 2 Winner | 94% |

We have fairly high accuracies for simulations across the board, a few of the more complex boards have slightly lower accuracies, but with a mean overall accuracy of 97.67%, it is above the requirement defined in the prototype aims.

**Robustness validation:**

For the robustness test, I ran 10000 random battles, as given by the generate random board method, measuring the number of crashes from this high number of runs should give us a fairly accurate measure of the robustness of the program/ the crash rate. The high number of runs accounts for the randomness of the battles, allowing us to ignore outliers and get a fairly accurate measure of our crash rate.

pub fn perform\_test() {

    for \_ in 0..10000 {

        let mut board = Board::generate\_random\_board(10, &DEFAULT\_CHAMPIONS, &DEFAULT\_ITEMS, 10000);

        board.simulate\_battle(None);

    }

}

pub fn generate\_random\_board(time\_unit: i8, champions : &Vec<Champion>, items : &[Item], ticks\_till\_draw : u32) -> Board {

        //randomly selects the number of player 1's and 2's champions in the range 1 to 6

        let num\_p1\_champs: usize = rand::thread\_rng().gen\_range(1..6);

        let num\_p2\_champs: usize = rand::thread\_rng().gen\_range(1..6);

        //fetches all item ids

        let item\_ids : Vec<u8> = items.iter().map(|f| f.id).collect();

        //for each champ, generate a random placed champion

        let p1\_champions: VecDeque<PlacedChampion> = (0..num\_p1\_champs)

            .map(|\_ : usize| PlacedChampion::generate\_random\_champ(champions.len(), &item\_ids, false))

            .collect();

        let p2\_champions: VecDeque<PlacedChampion> = (num\_p1\_champs

            ..num\_p1\_champs + num\_p2\_champs)

            .map(|\_ : usize| PlacedChampion::generate\_random\_champ(champions.len(), &item\_ids, true))

            .collect();

        //create new board

        Board::new(&p1\_champions, &p2\_champions, champions, items, time\_unit, ticks\_till\_draw)

    }

With the generate random placed champion method:

pub fn generate\_random\_champ(id\_range : usize, valid\_items : &Vec<u8>, team : bool) -> PlacedChampion {

        //generates a random id and star level

        let id = rand::thread\_rng().gen\_range(0..id\_range);

        let star = rand::thread\_rng().gen\_range(0..3) as usize;

        //initialise item array

        let mut items : [u8 ; 3] = [0, 0, 0];

        for i in 0..rand::thread\_rng().gen\_range(0..3) {

            //choose item from valid\_items

            items[i] = \*valid\_items.choose(&mut rand::thread\_rng()).unwrap();

        }

        //generate random location

        let location = Location::generate\_random\_position\_team(team);

        //returns placed champion

        PlacedChampion { id, star, items, location, team: None }

    }

In the 10,000 runs, there was a total of 3 crashes, resulting in a 0.0003% crash rate, much below the 0.5% goal, which I am happy with, considering the wide variety of boards and items being tested using the generate random board method.

Whilst not specifically mentioned in the prototype 2 testing, as this is the final version of the simulator, I thought I would test the efficiency and see if it was up to par with the expectations laid out in post-development testing of being able to simulate a complex battle in under 2 seconds on poor hardware. So I created a generate complex battle method:

pub fn generate\_complex\_random\_board(time\_unit: i8, champions : &[Champion], items : &[Item], ticks\_till\_draw : u32) -> Board {

        //randomly selects the number of player 1's and 2's champions in the range 3 to 6

        let num\_p1\_champs: usize = rand::thread\_rng().gen\_range(3..6);

        let num\_p2\_champs: usize = rand::thread\_rng().gen\_range(3..6);

        //fetches all item ids

        let item\_ids : Vec<u8> = items.iter().map(|f| f.id).collect();

        let id\_range = champions.len();

        //for each champ, generate a random placed champion

        let p1\_champions: VecDeque<PlacedChampion> = (0..num\_p1\_champs)

            .map(|\_ : usize| PlacedChampion::generate\_random\_champ(id\_range, &item\_ids, false))

            .collect();

        let p2\_champions: VecDeque<PlacedChampion> = (num\_p1\_champs

            ..num\_p1\_champs + num\_p2\_champs)

            .map(|\_ : usize| PlacedChampion::generate\_random\_champ(id\_range, &item\_ids, true))

            .collect();

        //create new board

        Board::new(&p1\_champions, &p2\_champions, champions, items, time\_unit, ticks\_till\_draw)

    }

Generating a minimum of 3 champions on each team, as defined for a complex battle and ran the simulation 1000 times on my laptop with the code:

use super::champions::DEFAULT\_CHAMPIONS;

use super::item::DEFAULT\_ITEMS;

use std::time::{Duration, Instant};

pub fn perform\_test() {

    let start = Instant::now();

    for \_ in 0..1000 {

        let mut board = Board::generate\_complex\_random\_board(10, &DEFAULT\_CHAMPIONS, &DEFAULT\_ITEMS, 10000);

        board.simulate\_battle(None);

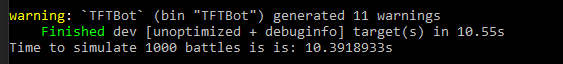
    }

    let duration = start.elapsed();

    println!("Time to simulate 1000 battles is is: {:?}", duration);

}

Which creates a variable with time now, called start, simulates the 1000 complex battles, then measures how long it takes, so we can get an average length of time.



The total execution time was 10.4 seconds, resulting in an average of 0.0104 seconds per battle, much much lower than the 2 second aim I set.

### Review of Prototype

This prototype was a large clean-up and optimisation of prototype 1, with some new implementations of things left out. Overall, I’m very happy with this prototype, which far exceeded the expectations set out in the design phase, feedback from my client for prototype 2 was good.

For prototype 2, we have full implementation of the simulator related classes, such as Champion and Item and we have battles simulated to a >95% accuracy with a <0.5% crash rate, fulfilling all the requirements laid out in design.

## Prototype 3:

Prototype 3 is the final prototype of the project, where we will implement a front-end user interface as well as a database. This prototype will make use of the Svelte frontend framework with Tauri to create a cross-platform app, as well as a SurrealDB database to store information.

We will create an ipc (inter-process communicator) for communication between the front and backend, and the backend will handle communication with the database as well, through the Store class, encapsulating all behaviour for both frontend backend communication and database communication.

The simulator for this prototype has remained untouched from prototype 2.

For prototype 3, we still have a modular layout, displayed below:

Backend:

Graphical user interface, application

Description automatically generated

The only “change” to the simulator is the mod.rs file, which replaces the main.rs file (which is no longer present as simulator is not stand alone). Mod.rs simply rexports all the files within the simulator:

/\* Exports Crate Modules \*/

pub mod board;

pub mod champions;

mod location;

mod projectiles;

mod shields;

mod status\_effects;

mod utils;

pub mod item;

pub mod perform\_tests;

Frontend:

Graphical user interface, application

Description automatically generated

The first thing I implemented in prototype 3 is a new error type:

use serde::Serialize;

///All possible errors for this project (not related to simulator)

#[derive(thiserror::Error, Debug, Serialize)]

pub enum Error {

    ///Error from SurrealDB

    #[error(transparent)]

    Surreal(#[from] surrealdb::Error),

    ///Error related to database

    #[error("Database Error: '{0}'")]

    DatabaseError(&'static str),

    ///Error retrieving Store from Ctx/ Tauri

    #[error("Failed to retrieve store")]

    StoreError,

    ///Error fetching Board from Store

    #[error("Failed to fetch board")]

    FetchBoardError,

    ///Error fetching last board from Store

    #[error("Failed to retrieve last board")]

    LastBoardError,

}

For the enum, I derive thiserror::Error, which derives the basic methods needed for the enum to be used as an error type in Rust. I also derive debug, which allows the enum to be printed to the console, as well as serialise from serde, which allows the enum to be serialised into json and sent to the frontend.

Then in the enum, I describe the error types, the first is the error type from surrealdb, allowing any errors from the database to display correctly. I then define some of my own, a different database error, that can be given a string to be displayed. I also create failure to fetch errors for the store, board and the last board id on the enum.

Then, I create a prelude file, with common definitions to be used throughout the project:

/\* Prelude \*/

//default imports of Error and Result

pub use crate::error::Error;

pub type Result<T> = core::result::Result<T, Error>;

I import the error defined in the crate (so that it is also imported in files that use the prelude) and then redefine the result struct, defining a Result of type T as a Result of type T with error Error from the crate.

A result struct is a type that contains a value or an error, by predefining the error as the crate error, I can avoid redundantly defining a Result with a type and an error, instead just with a type and an error predefined.

Afterwards, I moved onto working on the database. As laid out earlier, the database has to be able to store and retrieve unit and item stats, as well as previous board results.

We start by defining the Store class:

pub struct Store {

    ///database file

    ds: Datastore,

    ///session of database

    ses: Session,

    ///board currently being simulated

    board : Option<Board>,

    ///last board simulated

    last\_board : Option<String>

}

It holds a datastore and a session, which is an opened database file and the current database session. The class will also hold an option for a board and last\_board, which is the current board being simulated (as the frontend needs to be able to access it) as well as the id for the last board simulated and stored in the database (which allows the result/ outcome of the battle to be updated).

Later on, when building the Tauri application in the backend, we can pass in a store object to be “held” by the Tauri application, allowing the program to access it, for instance, whenever a command is called from the frontend, thus attributes that are put onto the store class can be accessed from the frontend/ backend.

Due to the nature of surrealdb databases, as well as the time spent communicating between different elements on the program, most if not all methods on the store class have to be asynchronous, to be able to handle delays/ waiting for the database etc, as well as running other lines of code alongside it.

impl Store {

    ///Creates a new store

    pub async fn new() -> Result<Self> {

        let ds = Datastore::new("file://tft\_bot\_database").await.unwrap(); //opens or creates database file

        let ses = Session::for\_db("appns", "appdb"); //creates a new session

        Ok(Store { ds, ses, board : None, last\_board : None})

    }

Here, we create a new store object, we create a new datastore object by connecting to a file called tft\_bot\_database, it will create an empty database if there is none, awaiting the interaction then unwrapping the result.

We then create a new session, with the handles “appns” and “appdb”.

Finally we create the new store object, without a board or last\_board.

We return a result of self, signifying there were no errors during the process, we can use Result without defining an error type as we have used the prelude earlier in the document:

use crate::prelude::\*;

We then define a setup method for the store:

///setups the board

    pub async fn setup(&self) -> Result<()> {

        //if there are no champions in the database

        if self.fetch\_champions\_ids().await?.is\_empty() {

            //insert default champions

            for champ in DEFAULT\_CHAMPIONS {

                match self.insert\_champion(&champ).await {

                    Ok(()) => info!("successfully inserted champ: {}", champ.id),

                    Err(e) => error!("error inserting champ: {}. {}", champ.id, e),

                }

            }

        }

        //if there are no items in the database

        if self.fetch\_items\_ids().await?.is\_empty() {

            //insert default items

            for item in DEFAULT\_ITEMS {

                match self.insert\_item(&item).await {

                    Ok(()) => info!("successfully inserted item: {}", item.id),

                    Err(e) => error!("error inserting item: {}. {}", item.id, e),

                }

            }

        }

        Ok(())

    }

Here, we fetch the champions and items stored in the current database, awaiting the outcome, and if there are none, we insert the default champions or items into the database, outputting a success or error message depending on the result.

All commands for champions and items are mimicked with slightly different sql depending on what is being queried.

///fetch a vector of all the champion ids

    pub async fn fetch\_champions\_ids(&self) -> Result<Vec<u8>> {

        let sql = "SELECT id FROM champions";

        //execute the statement, turn the result into a vector of objects, and for each one fetch the id and turn it onto a u8

        Ok(

            into\_iter\_objects(self.ds.execute(sql, &self.ses, None, false).await?)?

                .map(|f| {

                    fetch\_id(f.unwrap()).as\_int() as u8

                })

                .collect(),

        )

    }

So in the fetch champions ids method, we create a string to hold our sql. Surrealdb uses a language that is very similar to SQL but with slight discrepancies sometimes.

Then, we create our return value. We return a result, so we wrap our output in the “Ok” wrapper. To retrieve the value, we execute the statement on the datastore, passing in the sql, a reference to the session, None for variables and false for strict mode. None means that we don’t pass in any values alongside the statement, false for strict mode is irrelevant for select statements, but if it were an insertion/ create statement, if the table/ field specified in your sql statement didn’t exist, in strict mode it would return an error, but in non-strict mode, the field/ table would be created and then the value inserted, avoiding the error.

So we execute the sql, await the result and unwrap the result we get into its value. The ? operator in Rust can be used in any functions/ methods that return a Result. Using the ? operator will unwrap the result into a value, if the result contains an error, the function will close and the error will be returned.  
With our value returned from executing the sql, we pass it into the into\_iter\_objects function.

///code taken from: https://www.youtube.com/watch?v=iOyvum0D3LM

fn into\_iter\_objects(ress: Vec<Response>) -> Result<impl Iterator<Item = Result<Object>>> {

    let res = ress

        .into\_iter() //turns ress into an iterator of responses

        .next() //gets the first response (as in this project, I only make one request per statement, so there will next be any other responses)

        .map(|rp: Response| rp.result) //get the result of the response

        .transpose()?; //swap Option<Result> into Result<Option>

    match res {

        Some(Value::Array(arr)) => { //if res is an array of responses

            let it = arr.into\_iter().map(|v| match v {

                Value::Object(object) => Ok(object), //map each value into an object

                \_ => Err(Error::DatabaseError("A record was not an object")), //return error if invalid

            });

            Ok(it) //return iterator

        }

        \_ => Err(Error::DatabaseError("No records found")), //return database error

    }

}

Code taken from a youtube tutorial on the matter.

The function into iter objects takes a vector of Responses, which is always returned from a statement executed on the database. We then turn our ress into an iterable and call “next” on it, getting the first value in the vector. In surrealdb, you can execute multiple statements at once, called a transaction, as surrealdb is ACID compliant (used to describe compliance with safety standards for transactions), ensuring data security inspite of errors etc. However, in this project we will only ever use single statements in our transactions, so we always only need the first response.

Once we have our response, we map the response to its result. We then call transpose on the outcome, turning an Option<Result<T>> into Result<Option<T>>, then we unwrap the result.

We then match our res (which is the option we just created).

If the value is an array, we get the array value and turn it into an iterator, then for each object, we check its value, if it can be turned into a surrealdb object (which is a dict-like object), we wrap it in an ok value, otherwise we return an error, then we return our iterator. Our return type is a result of an iterator which iterates over values of Result<Object>.

If the value isn’t an array, we return a database error.

into\_iter\_objects(self.ds.execute(sql, &self.ses, None, false).await?)?

                .map(|f| {

                    fetch\_id(f.unwrap()).as\_int() as u8

                })

                .collect(),

Now we have an iterator of objects, we unwrap the result value on the iterator and then map each object in the iterable, we unwrap each object and pass it into the fetch\_id function, then for the value returned (which is a surrealdb VALUE object) we interpret it as an integer, then convert it into a u8, we then collect the iterator into a vector, wrap it into the Ok wrapper and return it.

The fetch id function is a small piece of utility code for some fairly obtuse and boilerplate code:

///small utility piece of code to fetch the id of the first result from a response

///<br /> fairly obtuse piece of code enclosed in a function to avoid redundant repeated code

fn fetch\_id(mut obj: Object) -> Value {

    Value::from(obj.remove("id").unwrap().record().unwrap().id)

}

from our object, we remove the “id” field (we take in obj as mutable so we can remove it) then unwrap the option it returns. We then take the value returned as a record, unwrap the option again, fetch the id and then convert the id into a surrealdb Value type, which can be interpreted as many types. (Ids/ values can be int, string, float etc).

The code for items is the same, just selecting items rather than champions in the sql:

///fetch a vector of all the item ids

    pub async fn fetch\_items\_ids(&self) -> Result<Vec<u8>> {

        let sql = "SELECT id FROM items";

        //execute the statement, turn the result into a vector of objects, and for each one fetch the id and turn it onto a u8

        Ok(

            into\_iter\_objects(self.ds.execute(sql, &self.ses, None, false).await?)?

                .map(|f| {

                    fetch\_id(f.unwrap()).as\_int() as u8

                })

                .collect(),

        )

    }

Now we move onto inserting champions:

///insert a champion into the database

    pub async fn insert\_champion(&self, champion: &Champion) -> Result<()> {

        //create the sql statement, creating a champion with id id and content data.

        let sql = format!("CREATE champions:{id} CONTENT $data", id = champion.id);

        //turn the champion into values and then store it in a BTreeMap

        let data: BTreeMap<String, Value> = champion.into\_values().into();

        let vars: BTreeMap<String, Value> = [("data".into(), data.into())].into();

        //execute on the database

        self.ds.execute(&sql, &self.ses, Some(vars), false).await?;

        Ok(())

    }

We return an empty result, signifying the success of the operation. The insert champion asynchronous method takes in a reference to self and a reference to a champion.

We create our sql statement, where we create an entry with id equal to the champion id, in the champions table, with content “data”. Using the format statement changes the string, replacing the {id} with the value given by id, which we define as champion.id, this allows us to alter our sql using a variable.

As we run the code in non-strict mode, if the champions table does not exist, then the table will be create, same goes for any fields.

We then create our data, a BTreeMap, a binary tree with key value relationships, with the key being a string and the value being the surrealdb value object. We then turn our champion into\_values and then convert it to the BTreeMap with the “into” method.

pub fn into\_values(&self) -> [(String, Value) ; 9] {

        [

            ("id".into(), self.id.into()),

            ("hp".into(), self.hp.into()),

            ("sm".into(), self.sm.into()),

            ("mc".into(), self.mc.into()),

            ("ar".into(), self.ar.into()),

            ("mr".into(), self.mr.into()),

            ("ad".into(), self.ad.into()),

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

            ("ra".into(), self.ra.into())

        ]

    }

The into value method on champion takes a reference to self and then converts the champion into a 9 long array of key value pairs. For the strings, we convert &str (a reference to a string which is what you create when you plainly write a string like we did) and then convert the attributes on the champion into surrealdb values, the code for how to convert from, say, u8 to surrealdb value is coded as a method on the surrealdb value class, so no extra work has to be done on our end. We then take our data and turn use it in another BTreeMap, this one with the field “data” and value our data variable.

We then execute our statement on the datastore, passing in our session and sql, our variables which is Some(vars) and do not run it in strict mode.

The same occurs for insert\_item:

///insert an item into the database

    pub async fn insert\_item(&self, item: &Item) -> Result<()> {

        //create the sql statement

        let sql = format!("CREATE items:{id} CONTENT $data", id = item.id);

        //turn the item into values

        let data: BTreeMap<String, Value> = item.into\_values().into();

        let vars: BTreeMap<String, Value> = [("data".into(), data.into())].into();

        self.ds.execute(&sql, &self.ses, Some(vars), false).await?;

        Ok(())

    }

impl Item {

    ///converts into string value array for insertion in database

    pub fn into\_values(&self) -> [(String, Value) ; 13] {

        [

            ("id".into(), self.id.into()),

            ("health".into(), self.health.into()),

            ("ad".into(), self.ad.into()),

            ("ap".into(), self.ap.into()),

            ("ar".into(), self.ar.into()),

            ("mr".into(), self.mr.into()),

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

            ("ra".into(), self.ra.into()),

            ("cr".into(), self.cr.into()),

            ("dc".into(), self.dc.into()),

            ("cm".into(), self.cm.into()),

            ("omnivamp".into(), self.omnivamp.into()),

            ("crit\_damage".into(), self.crit\_damage.into())

        ]

    }

}

We then have a few other methods.

Fetch champions fetches a vector of all champions in the database:

///fetch a list of all champions

    pub async fn fetch\_champions(&self) -> Result<Vec<Champion>> {

        //create the sql statement

        let sql = "SELECT \* FROM champions";

        //get the ress

        let ress = self.ds.execute(sql, &self.ses, None, false).await?;

        //turn it into an iterator of object, and for each object try to create a champion from it, then collect the iterator into a vector

        Ok(into\_iter\_objects(ress)?

            .map(|f| Champion::try\_from(f.unwrap()).unwrap())

            .collect())

    }

We select all from champions, execute the sql, await the result, then turn our result into an iterable of objects and map each object into a champion, collecting the resulting vector.

impl TryFrom<Object> for Champion {

    type Error = Error;

    fn try\_from(mut obj: Object) -> Result<Self> {

        //fetch and convert values from the object

        let ad : f32 = obj.remove("ad").unwrap().as\_float() as f32;

        let ar : f32 = obj.remove("ar").unwrap().as\_float() as f32;

        let attack\_speed : f32 = obj.remove("attack\_speed").unwrap().as\_float() as f32;

        let hp : f32 = obj.remove("hp").unwrap().as\_float() as f32;

        let id : u8 = Value::from(obj.remove("id").unwrap().record().unwrap().id).as\_int() as u8;

        let mc : i16 = obj.remove("mc").unwrap().as\_int() as i16;

        let mr : f32 = obj.remove("mr").unwrap().as\_float() as f32;

        let ra : i8 = obj.remove("ra").unwrap().as\_int() as i8;

        let sm : i16 = obj.remove("sm").unwrap().as\_int() as i16;

        //return new champ

        Ok(Champion { id, hp, sm, mc, ar, mr, ad, attack\_speed, ra})

    }

}

Try from for champion takes a mutable object as input and returns a result of type self. It treats object like a dict-like object and takes fields stored on the object, turning them into the correct type, before returning the new champion.

    ///fetch a vector of all items

    pub async fn fetch\_items(&self) -> Result<Vec<Item>> {

        let sql = "SELECT \* FROM items";

        let ress = self.ds.execute(sql, &self.ses, None, false).await?;

        //println!("{ress:?}");

        Ok(into\_iter\_objects(ress)?

            .map(|f| Item::try\_from(f.unwrap()).unwrap())

            .collect())

    }

impl TryFrom<Object> for Item {

    type Error = Error;

    ///tries to convert from a database object into an item object

    fn try\_from(mut obj: Object) -> Result<Self> {

        //fetches values from database like object

        let ad = obj.remove("ad").unwrap().as\_float() as f32;

        let ap = obj.remove("ap").unwrap().as\_float() as f32;

        let ar = obj.remove("ar").unwrap().as\_float() as f32;

        let mr = obj.remove("mr").unwrap().as\_float() as f32;

        let attack\_speed\_modifier = obj.remove("attack\_speed\_modifier").unwrap().as\_float() as f32;

        let cm = obj.remove("cm").unwrap().as\_int() as i16;

        let cr = obj.remove("cr").unwrap().as\_int() as u8;

        let crit\_damage = obj.remove("crit\_damage").unwrap().as\_float() as f32;

        let dc = obj.remove("dc").unwrap().as\_int() as u8;

        let health = obj.remove("health").unwrap().as\_float() as f32;

        let omnivamp = obj.remove("omnivamp").unwrap().as\_float() as f32;

        let ra = obj.remove("ra").unwrap().as\_int() as i8;

        let id : u8 = Value::from(obj.remove("id").unwrap().record().unwrap().id).as\_int() as u8;

        Ok(Item { id, ad, ap, attack\_speed\_modifier, health, ar, mr, ra, cr, dc, cm, omnivamp, crit\_damage})

    }

}

We can also fetch champions by id:

    ///fetch a champion from an id

    pub async fn fetch\_champion\_from\_id(&self, id: u8) -> Result<Option<Champion>> {

        let sql = &format!("SELECT \* FROM champions:{id}");

        let ress = self.ds.execute(sql, &self.ses, None, false).await?;

        //if there is an object in the result vector, try to create a champion from the object

        if let Some(obj) = into\_iter\_objects(ress)?.next() {

            return Ok(Some(Champion::try\_from(obj?)?))

        }

        Ok(None)

    }

We create our sql, where we specify the id for our query and directly access the values.

We then check if the first value in the into object iterable is an object and if so return the Champion wrapped in a result and option, otherwise we return None.

We wrap our return value in a Result and Option wrappers, to give the most information possible. If we get a result of ok but an option of None, we know a champion of that id doesn’t exist, otherwise if we get an error, we know an error occurred in the operation.

    ///fetch an item from the database by id

    pub async fn fetch\_item\_from\_id(&self, id: u8) -> Result<Option<Item>> {

        let sql = &format!("SELECT \* FROM items:{id}");

        let ress = self.ds.execute(sql, &self.ses, None, false).await?;

        //if there is an object in the result vector, try to create an item from the object

        if let Some(obj) = into\_iter\_objects(ress)?.next() {

            return Ok(Some(Item::try\_from(obj?)?))

        }

        Ok(None)

    }

We also have the option to update a champion in the database, given one as an argument:

///updates a champion's value

    pub async fn update\_champion(&self, champion : Champion) -> Result<()> {

        //create sql, update champ with id : id

        let sql = format!("UPDATE champions:{id} CONTENT $data", id = champion.id);

        //turn champion into values

        let data: BTreeMap<String, Value> = champion.into\_values().into();

        let vars: BTreeMap<String, Value> = [("data".into(), data.into())].into();

        //execute statement

        self.ds.execute(&sql, &self.ses, Some(vars), false).await?;

        Ok(())

    }

We update the value at champions:{id} with the new data, we turn our new champion into values and set that to our vars, then execute the statement, updating the champion in the database, returning an Ok to signify the operation went through without errors.

    ///updates an item's values

    pub async fn update\_item(&self, item : Item) -> Result<()> {

        let sql = format!("UPDATE items:{id} CONTENT $data", id = item.id);

        let data: BTreeMap<String, Value> = item.into\_values().into();

        let vars: BTreeMap<String, Value> = [("data".into(), data.into())].into();

        self.ds.execute(&sql, &self.ses, Some(vars), false).await?;

        Ok(())

    }

As the board field is private on a store, we have a method to set the board to a new value:

    ///takes in a board as input and sets the self.board field to it

    pub fn set\_board(&mut self, board : Board) -> Result<()> {

        self.board = Some(board);

        Ok(())

    }

It takes in a mutable reference to self and the board, and updates the board stored on self.

    ///replace the self.board value with the given value, returning the old value

    pub fn replace\_board(&mut self, mut board : Option<Board>) -> Result<Option<Board>> {

        swap(&mut self.board, &mut board);

        Ok(board)

    }

We also have one to replace the board, we take a mutable reference to self and a mutable Option of a board. We then swap the value given to us with the value currently stored, using the std::mem::swap, a rust built in function, that swaps the values of two variables with the same type, using this we can avoid expensive clone/ copy operations.

//import swap

use std::mem::swap;

We then return the old board. This method can be used to retrieve the board, swapping it with an empty value to signify the board is currently unavailable. This method is commonly used when the backend wants to simulate some ticks on the board, the board value can be retrieved, used to simulate some ticks, then swapped back into the store.

    ///clones the current board and returns it

    pub fn fetch\_board(&self) -> Result<Option<Board>> {

        Ok(self.board.as\_ref().cloned())

    }

We have another method that retrieves the board as a reference, then clones the variable and returns it, wrapping it in a result wrapper.

One of the requirements for the store/ database is that it has to be able to store previous boards, or more accurately the champions placed on the board.

///stores a board with given placed champions

    pub async fn store\_board(&mut self, p1\_champs : &VecDeque<PlacedChampion>, p2\_champs : &VecDeque<PlacedChampion>) -> Result<()> {

        //create board sql with unknown outcome

We have a store board method that takes a double ended vector of placed champions for both player 1 and 2.

//create board sql with unknown outcome

        let sql = "CREATE boards SET outcome = 0";

        let ress = self.ds.execute(sql, &self.ses, None, false).await?;

        //fetch id of new field created

        let id = fetch\_id(into\_iter\_objects(ress)?.next().unwrap()?).as\_string();

We create an empty board with an undefined outcome/ draw and then fetch the autogenerated field id generated by surrealdb, from the result.

Then for each champ in both teams:

for champ in p1\_champs {

            //turn champ into values

            let mut data: BTreeMap<String, Value> = champ.into\_values().into();

            //insert into data link to board value

            data.insert("board".into(), board\_link.clone().into());

            //insert into data team value

            data.insert("team".into(), 1.into());

            //create vars

            let vars: BTreeMap<String, Value> = [("data".into(), data.into())].into();

            //execute statement

            self.ds.execute(sql, &self.ses, Some(vars), false).await?;

        }

We take our champ and turn it into values. We then insert the board id and the team its on (so it can be correctly displayed later). We put this data into our vars, then execute the statement.

We repeat this for each placed champion on both teams:

for champ in p2\_champs {

            //repeat for player 2 champs

            let mut data: BTreeMap<String, Value> = champ.into\_values().into();

            data.insert("board".into(), board\_link.clone().into());

            data.insert("team".into(), 2.into());

            let vars: BTreeMap<String, Value> = [("data".into(), data.into())].into();

            self.ds.execute(sql, &self.ses, Some(vars), false).await?;

        }

Here is the implementation of into values for placed champion:

impl PlacedChampion {

    ///converts a placed champion into a String value array for insertion into the database

    pub fn into\_values(&self) -> [(String, Value); 7] {

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

         ("star".into(), self.star.into()),

         ("item\_0".into(), self.items[0].into()),

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

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

         ("location\_x".into(), self.location.x.into()),

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

        ]

    }

}

Nothing new, but we do split up items and locations up into their attributes as they do not have surrealdb representation into values.

Then, once we have created the board in the database, we set the last board to the id given:

 //set last board to this board id

        self.last\_board = Some(id);

        Ok(())

}

Then, we have a method for updating the outcome of the last board, this is to be used when the user has finished simulating the battle on the frontend and they want to save the result.

///updates the outcome of the last board

    pub async fn update\_outcome(&self, outcome : u8) -> Result<()> {

        if self.last\_board.is\_some() { //if there is last board

            let last\_board = self.last\_board.clone().unwrap(); //clone the string

            let sql = &format!("UPDATE boards:{last\_board} SET outcome = {outcome}"); //update the board outcome in the database with the new outcome

            self.ds.execute(sql, &self.ses, None, false).await?;

            return Ok(())

        }

        Err(Error::LastBoardError) //return last board error

    }

We try to fetch the id of the last board, if so, we clone the id and unwrap it. We have to clone it as unwrapping is modifying the variable, so we cannot use a reference.

We then create our sql to update the correct board with the correct outcome given in the arguments, before executing the sql and returning Ok.

If we cannot fetch the last board/ there isn’t one, we return a last board error.

Next, is a method to fetch a list of all the save outcomes/ boards, to display on the frontend so the user can choose a board to view.

///returns a vector of a outcome, board ID pair

    pub async fn fetch\_outcomes(&self) -> Result<Vec<(i64, String)>> {

        let sql = "SELECT \* FROM boards"; //select all from boards

        let ress = self.ds.execute(sql, &self.ses, None, false).await?;

        //execute statement, get result

        //turn ress into iterator of objects, map the objects to an outcome, id pair and return

        Ok(into\_iter\_objects(ress)?.map(|obj| {

            let mut obj = obj.unwrap();

            (obj.remove("outcome").unwrap().as\_int(), fetch\_id(obj).as\_string())

        }).collect())

    }

We select all from boards in our sql and get out result. We then map our result to an enclosure, in the enclosure we unwrap the object, remove the outcome and fetch the id and then return from the enclosure a tuple containing the outcome and the id as a string. We collect our iterable into a vector and return it wrapped in a result.

Our final method on the store is a method that returns the layout of placed champions on the board, given an id:

///fetch the board state of a board with id : ID

    pub async fn fetch\_outcome\_board(&self, id : String) -> Result<Vec<PlacedChampion>> {

        //fetch all champs from boards\_champ with board id id

        let sql = &format!("SELECT \* FROM boards\_champ WHERE board = boards:{id}");

        let ress = self.ds.execute(sql, &self.ses, None, false).await?;

        //map result iterator into a vector of placedChampions

        Ok(into\_iter\_objects(ress)?.map(|f| PlacedChampion::try\_from(f.unwrap()).unwrap()).collect())

    }

We select all from boards\_champs with the correct board id, we execute the sql and then map the iterable of objects into PlacedChampions, collecting it into a vector.

impl TryFrom<Object> for PlacedChampion {

    type Error = Error;

    fn try\_from(mut obj: Object) -> Result<Self> {

        let id = obj.remove("of\_champ").unwrap().as\_int() as usize;

        let item\_0 = obj.remove("item\_0").unwrap().as\_int() as u8;

        let item\_1 = obj.remove("item\_1").unwrap().as\_int() as u8;

        let item\_2 = obj.remove("item\_2").unwrap().as\_int() as u8;

        let star\_level = obj.remove("star").unwrap().as\_int() as usize;

        let location\_x = obj.remove("location\_x").unwrap().as\_int() as i8;

        let location\_y = obj.remove("location\_y").unwrap().as\_int() as i8;

        let team = obj.remove("team").unwrap().as\_int() as u8;

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

    }

}

Tryfrom for object and placed champion looks like this, fetching each item and location individually. Any placed champion stored in the database will always have a team specified, so we know we can fetch as “some”.

With that, we’ve covered the full functionality of the store, this fulfilles all the requirements of the store, it allows champions and items to be stored or altered. It can store board outcomes and the layout of the board itself and it autofills with default values on first setup, so fits the database requirements laid out in design.

With store covered, we can now move onto the main.rs file of my project, which is run on launch.

#[tokio::main]

async fn main() -> Result<()> {

We define the main function, as an asynchronous function and also tokio main. Tokio is a rust library which allows for asynchronous functions, methods and multithreaded behaviour, which is required for interacting with surrealdb which makes use of asynchronous functions. We make main return an empty Result, so we get access to the ? operator and also can tell if the main function finished successfully.

env::set\_var("RUST\_LOG", "error");

    env\_logger::init(); //setup logger

    info!("Program Start Up");

We start the function by setting the environment log level, then initialising the logger, to allow for debugging information.

let store = Store::new().await?; //create a new store

    if store.setup().await.is\_ok() { //if store setup ok

        let store = Arc::new(RwLock::new(store)); //create new Arc and RwLock of store for cross-thread mutability

We then await the creation of our store, unwrapping the Store object from the result with the ? operator (which we can use as this function returns a Result, even if empty) and setup the store, if the store setup goes ok, we continue with the program.

We put our store inside an RwLock inside an Arc. Arc is a type that allows for cross-thread references/ clones. If we didn’t need any interior mutability, we could just use an Arc object for simplicity, allowing the store to be cloned/ referenced whenever needed, regardless of what thread it is on. However, as we require store to have interior mutability, for methods such as update board, we cannot have just an Arc alone. An RwLock inside an Arc is a common pattern for cross thread mutability. Moreover, an RwLock is more suitable for our use case as, most commonly, we will only need read access to our store. RwLock allows for multiple readers of whatever is contained inside the lock, but only one writer at a time/ one with mutable access. This is in comparison to another option, the Mutex class, which also allows for mutable, cross-thread access, but only allows 1 reader or writer at a time. Using RwLock decreases the chance there will be wait time fetching the store, as there can be numerous readers at once, which is all we will require most of the time, improving the performance of the program.

So we’ve wrapped our store inside an Arc and RwLock to allow for cross thread, mutable access. We then call our tauri builder method:

tauri::Builder::default() //call tauri builder to create app

            .manage(store)

            .invoke\_handler(tauri::generate\_handler //give commands

                retrieve\_all\_items,

                retrieve\_all\_units,

                retrieve\_item\_from\_id,

                retrieve\_unit\_from\_id,

                retrieve\_all\_item\_ids,

                retrieve\_all\_unit\_ids,

                update\_unit,

                update\_item,

                submit\_board,

                fetch\_board,

                simulate\_x\_ticks,

                update\_outcome,

                fetch\_outcomes,

                fetch\_outcome\_board

            ])

            .run(tauri::generate\_context!())

            .expect("error while running tauri application");

        Ok(())

    }

Here, we call the tauri builder to create our application and run the main loop. This is all fairly boilerplate/ generic code to start our application, autogenerated when creating a new project (other than the specifics of our commands/ the store).

We give it a store, to manage, this allows the program to access “store”, whenever it has access to the tauri application, such as when a command from the frontend is called.

Next, we call “invoke handler” on the builder, giving it a list of commands for it control, allowing the frontend to call any of them as it pleases.

Finally we “run” the program, expecting any errors, which if so, we return the message “error while running tauri application. “, before returning Ok(()).

Earlier, if the setup failed, we return a database error,

 else {

        Err(Error::DatabaseError("Failure to Start Up")) //return database error

    }

All our commands come from our ipc:

use crate::ipc::{retrieve\_all\_items, retrieve\_all\_units, retrieve\_item\_from\_id, retrieve\_unit\_from\_id, retrieve\_all\_item\_ids, retrieve\_all\_unit\_ids, update\_unit, update\_item, submit\_board, fetch\_board, simulate\_x\_ticks, update\_outcome, fetch\_outcomes, fetch\_outcome\_board};

Another thing to note is that, for compatibility with tokio, we do not use rust’s standard library rwlock, instead using tokio’s:

use tokio::sync::{RwLock};

We do use the standard one for Arc and env:

use std::{env, sync::Arc};

use tokio::sync::RwLock;

use crate::store::Store;

mod simulator;

mod store;

mod error;

mod prelude;

mod ipc;

use crate::ipc::{retrieve\_all\_items, retrieve\_all\_units, retrieve\_item\_from\_id, retrieve\_unit\_from\_id, retrieve\_all\_item\_ids, retrieve\_all\_unit\_ids, update\_unit, update\_item, submit\_board, fetch\_board, simulate\_x\_ticks, update\_outcome, fetch\_outcomes, fetch\_outcome\_board};

use crate::prelude::\*;

use crate::simulator::perform\_tests::perform\_test;

#[macro\_use]

extern crate log;

This is the top of the main.rs file. We create modules for specific parts of our code, allowing them to be accessed throughout the entirety of our backend, we import our Store type, use all from the prelude and setup our log macros, so we can use info! Etc.

Onto the ipc (interprocess communicator), which handles communication between the frontend/ backend, but mainly allows the frontend to call commands for the backend to perform. We can start out with the imports:

use crate::prelude::\*;

use crate::simulator::champions::PlacedChampion;

use crate::simulator::{champions::Champion, item::Item, board::Board};

use crate::store::Store;

use std::collections::VecDeque;

use std::sync::{Arc};

use tokio::sync::{RwLock};

use tauri::{command, AppHandle, Manager, Wry};

We use the entirety of the prelude, some required types from the simulator such as PlacedChampion and Item, to send to the frontend, we use the store and VecDeque, so we can access the store and create a board which requires VecDeque’s, as well as Arc and Rwlock, the wrappers for the store, so we can correctly interact with it.

We also use some imports from tauri, such as command, apphandle, manager and Wry, so we can interact with the tauri application and create tauri commands.

First, we create a function to fetch the store from the connection:

///fetches a store from the connection

fn get\_store\_read\_from\_state(connection: AppHandle<Wry>) -> Result<Arc<RwLock<Store>>> {

    Ok((\*connection.state::<Arc<RwLock<Store>>>()).clone())

}

When a command is called, if the command asks for it, a “connection” will be passed in as a argument, taking the form of AppHandle<Wry>. When we asked the tauri builder to “manage” our store, it placed it on the AppHandle in the state attribute.

connection.state::<Arc<RwLock<Store>>>()

Here, we call the “state” method, which retrieves the state on the connection. As the state can take many forms and there isn’t a defined type on the class, we have to use the “turbofish syntax” :

::<Arc<RwLock<Store>>>

To tell the compiler/ program what type our state is.

It returns a reference to the Arc<RwLock<Store>>, which we dereference with the \* operator. We then clone the state, using .clone(), which actually isn’t an expensive operation as the “Arc” type simply creates a new reference to what is stored inside and returns that, we wrap it all in the Ok result and return it.

We then define our commands:

///retrieves a unit from an id

#[command]

pub async fn retrieve\_unit\_from\_id(id: u8, connection: AppHandle<Wry>) -> Result<Option<Champion>> {

    //fetch store

    if let Ok(store) = get\_store\_read\_from\_state(connection) {

        return store.read().await.fetch\_champion\_from\_id(id).await //get store read only, call fetch champion from id, await response

    }

    //return failed to fetch store

    Err(Error::StoreError)

}

We use the #[command] syntax, imported from tauri, to designate it as a command. We define it as async, as it interacts with the store and also is called from the frontend, so time will be spent communicating between the two, it public so it can be access from main.rs.

We take in two arguments, an id from the frontend and our connection, which will be provided by the application/ tauri itself.

We return a champion wrapped in an option and a result, which will be returned to the frontend. The result tells us whether the operation was a success, whereas the option will tell the frontend whether a champion exists with that id.

if let Ok(store) = get\_store\_read\_from\_state(connection) {

        return store.read().await.fetch\_champion\_from\_id(id).await //get store read only, call fetch champion from id, await response

    }

    //return failed to fetch store

    Err(Error::StoreError)

We try to get the store from the state, if the return value from get store is a success, we put it in the store variable, using the if let syntax. We then return our value, we call .read() on the store, to get read access to the RwLock, we wait the read and then await fetching the champion from id, which is what we return.

If the value returned from get store is now ok, we return the fetch store error.

///retrieve an item from an id

#[command]

pub async fn retrieve\_item\_from\_id(id: u8, connection: AppHandle<Wry>) -> Result<Option<Item>> {

    //fetch store

    if let Ok(store) = get\_store\_read\_from\_state(connection) {

        return store.read().await.fetch\_item\_from\_id(id).await //get store read only, call fetch item from id, await response

    }

    //return failed to fetch store

    Err(Error::StoreError)

}

Similar code for item.

We have this pattern repeated for most commands, we take in the connection and any arguments required, get a readable reference to the store, and then call the associated store function. Here are all the commands that follow that pattern:

///retrieve all units

#[command]

pub async fn retrieve\_all\_units(connection : AppHandle<Wry>) -> Result<Vec<Champion>> {

    if let Ok(store) = get\_store\_read\_from\_state(connection) {

        return store.read().await.fetch\_champions().await

    }

    Err(Error::StoreError)

}

///retrieve all items

#[command]

pub async fn retrieve\_all\_items(connection : AppHandle<Wry>) -> Result<Vec<Item>> {

    if let Ok(store) = get\_store\_read\_from\_state(connection) {

        return store.read().await.fetch\_items().await

    }

    Err(Error::StoreError)

}

///retrieve all unit ids

#[command]

pub async fn retrieve\_all\_unit\_ids(connection : AppHandle<Wry>) -> Result<Vec<u8>> {

    if let Ok(store) = get\_store\_read\_from\_state(connection) {

        return store.read().await.fetch\_champions\_ids().await

    }

    Err(Error::StoreError)

}

///retrieve all item ids

#[command]

pub async fn retrieve\_all\_item\_ids(connection : AppHandle<Wry>) -> Result<Vec<u8>> {

    if let Ok(store) = get\_store\_read\_from\_state(connection) {

        return store.read().await.fetch\_items\_ids().await

    }

    Err(Error::StoreError)

}

///update a unit with new values

#[command]

pub async fn update\_unit(selected\_unit : Champion, connection : AppHandle<Wry>) -> Result<()> {

    if let Ok(store) = get\_store\_read\_from\_state(connection) {

        return store.read().await.update\_champion(selected\_unit).await;

    }

    Err(Error::StoreError)

}

///update an item with new values

#[command]

pub async fn update\_item(selected\_item : Item, connection : AppHandle<Wry>) -> Result<()> {

    if let Ok(store) = get\_store\_read\_from\_state(connection) {

        return store.read().await.update\_item(selected\_item).await

    }

    Err(Error::StoreError)

}

These commands allow the frontend to manipulate the database, allowing the user to change things such as unit or item stats, or for the frontend to retrieve these things.

///take a board from the frontend and set the store to hold that board

#[command]

pub async fn submit\_board(player\_one\_champs : VecDeque<PlacedChampion>, player\_two\_champs : VecDeque<PlacedChampion>, time\_unit : i8, time\_till\_draw: u32, connection : AppHandle<Wry>) -> Result<()> {

    if let Ok(store) = get\_store\_read\_from\_state(connection) {

        let champs : Vec<Champion>;

        let items : Vec<Item>;

        {

            let store\_read = store.read().await; //get readable store

            champs = store\_read.fetch\_champions().await?; //fetch champions and items for use in Board initialisation

            items = store\_read.fetch\_items().await?;

        }

        let mut store\_write = store.write().await; //get writable store

        store\_write.store\_board(&player\_one\_champs, &player\_two\_champs).await?; //store board in database

        return store\_write.set\_board(Board::new(&player\_one\_champs, &player\_two\_champs, &champs, &items, time\_unit, time\_till\_draw)); //set board to new board

    }

    Err(Error::StoreError)

}

We then have the submit board command, this takes in the placement of the board and stores it in the database, before a battle.

We first try fetch our store, returning store fetch error if we can.

We create an empty vec of champs and items, then create a new block, in which we get a readable reference to the store, and then fetch champions and items.

After the block, the store\_read variable is dropped, to lower the number of active connections to the database during the next part, its also why we initialised champs and items to nothing outside the block, so they wouldn’t be dropped as well.

We then fetch a writeable/ mutable reference to our store and call the store\_board method, then create a new board with the details we have and set the store board to that board. Set board returns an empty result, so we can return the value it returns.

///simulate x ticks of stored board

#[command]

pub async fn simulate\_x\_ticks(num\_ticks : Option<u32>, connection : AppHandle<Wry>) -> Result<()> {

    if let Ok(store) = get\_store\_read\_from\_state(connection) {

        let board\_opt = store.write().await.replace\_board(None)?; //fetch board

        if let Some(mut board) = board\_opt { //if board exists

            board.simulate\_battle(num\_ticks); //simulate battle

            store.write().await.replace\_board(Some(board))?; //swap back board

            return Ok(())

        }

        else {

            return Err(Error::FetchBoardError) //return fetch board error

        }

    }

    Err(Error::StoreError) //return fetch store error

}

Next we have the command to simulate some ticks on the board, we take in the number of ticks to simulate and the connection, we try to fetch the store once more.

Then, we fetch a writeable reference to the store and call the replace board method on store, swapping the board currently stored on the object with an empty one. By swapping the proper board with an empty one, if someone tries to call this method in quick succession/ while the actual board is still being used, the method can simply return an error and we can avoid two simultaneous simulations/ an asynchronous race to see which board stays.

We check for this with the next line, ensuring that the board we just fetched is Some rather than None, unwrapping it with the if let syntax. If this is the case, we call the simulate battle method, entering the number of ticks given.

After this is done, we swap back the boards, replacing it with the old one that has simulated the extra ticks, we tehen return ok.

If the board we fetched was none, we return a fetch board error.

#[command]

pub async fn fetch\_board(connection : AppHandle<Wry>) -> Result<Option<Board>> {

    if let Ok(store) = get\_store\_read\_from\_state(connection) {

        return store.read().await.fetch\_board() //return board

    }

    Err(Error::StoreError)

}

#[command]

pub async fn update\_outcome(outcome : u8, connection : AppHandle<Wry>) -> Result<()> {

    if let Ok(store) = get\_store\_read\_from\_state(connection) {

        return store.read().await.update\_outcome(outcome).await //take an outcome from the frontend and update the most recently saved board with said outcome

    }

    Err(Error::StoreError)

}

#[command]

pub async fn fetch\_outcomes(connection : AppHandle<Wry>) -> Result<Vec<(i64, String)>> {

    if let Ok(store) = get\_store\_read\_from\_state(connection) {

        return store.read().await.fetch\_outcomes().await; //fetch all results/ board

    }

    Err(Error::StoreError)

}

#[command]

pub async fn fetch\_outcome\_board(id : String, connection : AppHandle<Wry>) -> Result<Vec<PlacedChampion>> {

    if let Ok(store) = get\_store\_read\_from\_state(connection) {

        return store.read().await.fetch\_outcome\_board(id).await; //give the result id of a board, fetch the board stored in database

    }

    Err(Error::StoreError)

}

We then have more methods that just fetch the store from the connection and call a method on the store, these methods allow the user to do a multitude of things, such as updating the outcome of the last board simulated, to fetching a list of all the outcomes stored in the database.

All in all, our ipc is fairly simple, as thanks to good setup and code throughout the rest of the project, all we really do in the ipc is fetch the store and then call the correct method. This is because we have encapsulated all the behaviour we need for the store or board to perform. This results in the ipc never having to directly, for example, query the database or perform a check, as its all encapsulated in other sections of the code.

With that, we have covered the entirety of the backend developments in prototype 3 and so we can move onto the frontend.

For our frontend, we use sveltekit, allowing us to have routing/ multiple pages functionality built in, for ease of use, this reduces one of the limitations I laid out in analysis of limited time.

We start out with layout.svelte:

<main class="container">

  <!--create a link to navigation that is acccessible everywhere, by placing in layout, will be present on all pages-->

  <a href="../">Navigation</a> <br />

  <slot />

</main>

The main class “container” will be present on every page by default, we create a link to “../”, which is a relative link which will take you to the parent/ base page. We title the link “navigation” so they know it takes them to the navigation page. We then put a <br /> to ensure any content starts on a new line.

Here you can see the navigation link in the top left of the page.

The “slot” element is where all the other elements on the other pages will be placed, here we place it underneath the navigation element.

Next we have the home/ base page, which we have setup as our navigation page:

<br />

<a href="board">Board</a>

<br />

<a href="change\_stats">Change Unit and Item Stats</a>

<br />

<a href="past\_results">Previous Results</a>

<!--navigation page is base page, with links to all other pages available-->

We have links to all other pages in the program, with their associated name, as shown below:

![Text

Description automatically generated with medium confidence

We can look at the change stats page next:

Diagram

Description automatically generated with medium confidence

On this page, we display a list of unit and item stats and offer the user the chance to change them. The frontend, of course, does not have this list of unit and items, so we have to fetch them from the backend.

This is an asynchronous action, so I start out by using the svelte await block:

{#await selected\_item}

    <div>Loading...</div>

{:then}

Graphical user interface

Description automatically generated with medium confidence

Which displays loading while we are waiting for the fetch from the backend.

<script>

    // @ts-nocheck

    import { invoke } from "@tauri-apps/api/tauri"

In a script block, we import invoke from the tauri api, which allows us to invoke back end commands.

We then create an asynchronous function to fetch the units and items:

async function fetch\_champs() {

        champs\_list = await invoke("retrieve\_all\_units")

        return champs\_list[0]

    }

    async function fetch\_items() {

        items\_list = await invoke("retrieve\_all\_items")

        return items\_list[0]

    } //fetch champs and items

    let champs\_list = []

    let items\_list = []

We can then fetch these champs and items:

let selected\_unit = fetch\_champs() //initialise selected unit and item

    let selected\_item = fetch\_items()

Once we have fetched our lists, we need to split the page into two.

<div class="row">

    <div class="column"> <!--split page into two-->

</div>

    <div class="column">

    </div>

</div>

{/await}

We can create a row and two columns, then in the style tag:

<style>

    /\* create flex box and set each column to 50% of page \*/

    .row {

        display: flex;

    }

    .column {

        flex: 50%;

    }

</style>

We set row to be a flex box, with each column taking up 50% of the width of the box.

In the first column, we can have unit stats:

<h1>Change Unit Stats</h1>

We have some text saying change unit stats, and then we can create a form:

<form on:submit|preventDefault={handle\_submit\_update\_champ}> <!--create form-->

Using a form we can ensure that each input box is filled before submission, as well as have a method to handle the submit.

We can then define our inputs:

<label>ID</label>

            <select bind:value = {selected\_unit} required>

                {#each champs\_list as champ}

                    <option value = {champ}>

                        {champ.id}

                    </option>

                {/each}

            </select>

We start out with a label for the input below, being the id input. We then have a select dropdown, where we bind our value to the selected unit variable.

We use a svelte for loop to iterate through each champ in champs\_list, and then create a dropdown option where the value is the champ, and the text display is the champ id.

A picture containing table

Description automatically generated

If you select one, it changes the value of selected unit to the selected dropdown item.

<label>AD</label> <!--create inputs for all required values-->

            <input type= "number" bind:value="{selected\_unit.ad}" step = 0.01 required><br>

            <label>HP</label>

            <input type="number" bind:value="{selected\_unit.hp}" step = 0.01 required><br>

            <label>Attack Speed</label>

            <input type="number" bind:value="{selected\_unit.attack\_speed}" step = 0.01 required><br>

            <label>Ar</label>

            <input type="number" bind:value="{selected\_unit.ar}" step = 0.01 required><br>

            <label>Mr</label>

            <input type="number" bind:value="{selected\_unit.mr}" step = 0.01 required><br>

            <label>Mc</label>

            <input type="number" bind:value="{selected\_unit.mc}" required><br>

            <label>Ra</label>

            <input type="number" bind:value="{selected\_unit.ra}" required><br>

            <label>Sm</label>

            <input type="number" bind:value="{selected\_unit.sm}" required><br>

We then have a bunch of labels and inputs, for each of the changeable values of a champ.

We bind these values to the appropriate attribute on the selected unit object. This has two effects, when a selected unit is selected, the value in the input will automatically update to the initial value of the stat, as shown:

Starts off empty:

A picture containing table

Description automatically generated

Then autofills when you select an id:

Table

Description automatically generated

As all the values in the input are updated. This gives the user a baseline for what to expect/ what it currently is. Another thing to note about the inputs is that each of them have “type = number”, this means that the input box won’t allow any non-numeric values in the boxes and won’t let the user type them. Plus, by default, the step for number inputs is 1. This means that the minimum difference between values is 1, so the numbers can only be integers. When I specify step = 0.01, it allows values up to two decimal places within the input box. This is part of our validation for the inputs, ensuring the user cannot enter invalid values into the boxes, if any invalid inputs are entered, they either won’t be allowed or an error will be shown stating what is wrong. This part of the validation is mainly handled by SvelteKit/ Javascript automatically.

Finally, we have a submit button on the bottom, of type submit, so when the button is pressed the function given on the “on form submit” is executed:

<button type = "submit">Submit</button>

If you try to submit a value that is invalid, such as a decimal with two many places, the form automatically won’t let you and the function is not called:

A picture containing diagram

Description automatically generated

On the bottom, we have some text:

<h1 style="opacity : {opacity\_champ\_error}">Invalid values for some or all variables, please try again.</h1> <!--create error that is hidden most of the time-->

We set the opacity in the style, to the opacity champ error variable, which means the opacity of the text will be updated every time the variable changes value.

let opacity\_champ\_error = 0 //set errors for champ and items opacity to 0

    let opacity\_item\_error = 0

By default this value is 0, so the text is hidden.

In the handle submit update champ for the form:

async function handle\_submit\_update\_champ (e) {

        if (check\_valid\_champ(selected\_unit)) { //if the selected unit is valid, update the unit

We have a function that checks whether the champion is valid

await invoke("update\_unit", {selectedUnit : selected\_unit})

            fetch\_champs() //fetch new champ

            opacity\_champ\_error = 0 //hide error

If it is, we invoke the update unit command from the ipc, and pass in the argument “selectedUnit” with the value selected\_unit. We then fetch champs again to update/ reset all the champions and hide the opacity champ error (if it was shown). If “check\_valid\_champ”, returned false, we show the champ error:

else {

            opacity\_champ\_error = 100 //show champ error

        }

A picture containing text

Description automatically generated

The check valid champ function checks that the value of the champs are reasonable/ will not cause any errors:

Text

Description automatically generated

If ensures that none of the values are below zero, which would cause errors, or if they are above 9999 or 20. If the value on the backend is an f32, we ensure that the value given on the frontend is below 9999, as that can fit, apart from for attack speed, which would not be reasonable if it was above 20. We ensure range is below 20 as the value is a u8, so cannot hold large values and is fairly useless to increase above certain values, as the board is not that large.

If it fails any of the conditions, it returns false, otherwise it returns true. This is the other part of validation for our inputs, ensuring the numbers given to us in the input boxes are in the correct range.

We have a very similar layout for the item column, on the right side of the screen:

<div class="column">

        <h1>Change Item Stats</h1>

        <form on:submit|preventDefault={handle\_submit\_update\_item}> <!--create form for item stats-->

            <label>ID</label><!--create inputs for all required values-->

            <select bind:value = {selected\_item} required>

                {#each items\_list as item}

                    <option value = {item}>

                        {item.id}

                    </option>

                {/each}

            </select>

            <br>

            <label>AD</label>

            <input type= "number" bind:value="{selected\_item.ad}" step = 0.01 required><br>

            <label>AP</label>

            <input type= "number" bind:value="{selected\_item.ap}" step = 0.01 required><br>

            <label>Ar</label>

            <input type= "number" bind:value="{selected\_item.ar}" step = 0.01 required><br>

            <label>Attack\_speed\_modifier</label>

            <input type= "number" bind:value="{selected\_item.attack\_speed\_modifier}" step = 0.01 required><br>

            <label>CM</label>

            <input type= "number" bind:value="{selected\_item.cm}" required><br>

            <label>Cr</label>

            <input type= "number" bind:value="{selected\_item.cr}" required><br>

            <label>crit\_damage</label>

            <input type= "number" bind:value="{selected\_item.crit\_damage}" step = 0.01 required><br>

            <label>Dc</label>

            <input type= "number" bind:value="{selected\_item.dc}" required><br>

            <label>Health</label>

            <input type= "number" bind:value="{selected\_item.health}" step = 0.01 required><br>

            <label>mr</label>

            <input type= "number" bind:value="{selected\_item.mr}" step = 0.01 required><br>

            <label>omnivamp</label>

            <input type= "number" bind:value="{selected\_item.omnivamp}" step = 0.01 required><br>

            <label>Range</label>

            <input type= "number" bind:value="{selected\_item.ra}" required><br>

            <button type = "submit">Submit</button>

            <h1 style="opacity : {opacity\_item\_error}">Invalid values for some or all variables, please try again.</h1> <!--create error that is hidden most of the time-->

        </form>

    </div>

We have a dropdown to select the item, a large number of inputs to allow the user to change the item stats and then a submit button, with an error text hidden on the bottom. The handle submit function is very similar as well:

async function handle\_submit\_update\_item (e) {

        if (check\_valid\_item(selected\_item)) { //if item is valid

            await invoke("update\_item", {selectedItem : selected\_item}) //update item

            fetch\_items() //fetch new item

            opacity\_item\_error = 0 //hide item erorr

        }

        else {

            opacity\_item\_error = 100 //show item error

        }

    }

Check if the item is valid, invoke the ipc command if so, fetch items again and hide the error message, otherwise show the item error text.

function check\_valid\_item(item) { //make sure item values are reasonable/ will not cause errors

        if (item.ad < 0 || item.ad > 9999) { return false }

        if (item.ap < 0 || item.ap > 9999) { return false }

        if (item.health < 0 || item.health > 9999) { return false }

        if (item.ar < 0 || item.ar > 9999) { return false }

        if (item.mr < 0 || item.mr > 9999) { return false }

        if (item.attack\_speed\_modifier < 0 || item.attack\_speed\_modifier > 20) { return false }

        if (item.cm < 0 || item.cm > 9999) { return false }

        if (item.cr < 0 || item.cr > 100) { return false }

        if (item.ra < 0 || item.ra > 20) { return false }

        if (item.dc < 0 || item.dc > 100) { return false }

        if (item.omnivamp < 0 || item.omnivamp > 9999) { return false}

        if (item.crit\_damage < 0 || item.crit\_damage > 9999) { return false }

        return true

    }

Check valid items ensures the values are above 0, reasonable and will not cause any out of bounds errors on the backend.

In total, this allows the user to change unit and item stats:

Table

Description automatically generated with medium confidence

Next, we can move onto our past results page.

A picture containing diagram

Description automatically generated

Here, we need 3 columns, one for the grid, one for the stats and one to show previous results. We can use the same flex box trick to accomplish this:

.row {

        display: flex;

    }

    .column1\_noselect {

        flex: 20%;

        background-color: grey;

        user-select: none;

    }

    .column3 {

        flex: 10%;

        background-color: grey;

        display:grid;

    }

    .column2 {

        flex : 70%;

        background-color: aliceblue;

    }

In the style tag. We can start out with the hexagon grid, the middle column.

<div class="column2">

        {#each grid as row, i}

            {#if i % 2 == 1}

                <HexagonIndent></HexagonIndent> <!--indent every other line-->

            {/if}

            {#each row as hex, j} <!-- create hexagon grid-->

                <Hexagon champion = {grid[i][j]} on:click = {() => hex\_click(i, j)}></Hexagon>

            {/each}

            <div class = "hex-row"></div>

        {/each}

    </div>

In the second column, we iterate through each row in the grid, also fetching i, which is the index/ count of the loop.

Grid is an 8x8, 2 dimensional array:

let grid = create\_grid() //create empty grid

function create\_grid() { //create empty grid

        let grid = new Array(8)

        for(var i = 0; i < grid.length; i++) {

            grid[i] = new Array(8)

        }

        return grid

    }

That starts out empty.

{#each grid as row, i}

            {#if i % 2 == 1}

                <HexagonIndent></HexagonIndent> <!--indent every other line-->

            {/if}

            {#each row as hex, j} <!-- create hexagon grid-->

                <Hexagon champion = {grid[i][j]} on:click = {() => hex\_click(i, j)}></Hexagon>

            {/each}

            <div class = "hex-row"></div>

        {/each}

So we fetch each row in the grid, then, if the index is odd (so for every other row), we create a “HexagonIndent” component. This is a simple component that indents every other row, so you get the correct grid effect.

Diagram

Description automatically generated with low confidence

It is an invisible element that simply indents the row.

We then iterate through each hexagon on the row:

{#each row as hex, j} <!-- create hexagon grid-->

                <Hexagon champion = {grid[i][j]} on:click = {() => hex\_click(i, j)}></Hexagon>

            {/each}

We create a hexagon component, we pass in “champion” with the value of its location on the grid and we bind the on:click event for the hexagon to the function “hex\_click” with the arguments i, j, its location in the list.

<div class = "hex">

    <div class = "top"></div>

    <!--create a button with background color : color, changes dependant on team

        set button to listen to on:click and on:mouseup, parent componenent can define behaviour

        if there is a champion, show the champion id

    -->

    <button on:click on:mouseup style = "background-color: {color}">

        {#if champion}

            {champion.placed\_id}

        {/if}

    </button>

    <div class = "bottom"></div>

</div>

In the hexagon component, we create the hex, with the classes hex, top and bottom, which with this css:

.hex {

        float: left;

        margin-left: 3px;

        margin-bottom: -26px; /\* sets float to left, sets margin\*/

    }

    .top {

        width: 0;

        border-bottom: 30px solid #6C6;

        border-left: 52px solid transparent;

        border-right: 52px solid transparent;

        z-index: 2;

    }

    button {

        width: 104px;

        height: 60px;

        background: #6C6;

        z-index: 2;

    }

    .bottom {

        width: 0;

        border-top: 30px solid #6C6;

        border-left: 52px solid transparent;

        border-right: 52px solid transparent;

        z-index: 2;

    }

Creates a hexagon shape, the css for the hexagon was taken from:

<http://jtauber.github.io/articles/css-hexagon.html>

We also create the style for the button, with the correct width, height and background color, as well as the correct z-level, which ensures it displays on top.

<button on:click on:mouseup style = "background-color: {color}">

        {#if champion}

            {champion.placed\_id}

        {/if}

    </button>

We create our button, defining listeners for the on:click and on:mouseup effect. We do not provide functions that occur when these events happen, this allows the parent component to define this for us then, like how we do in the parent page we define the on:click for hexagon as hex\_click.

We then have a svelte if statement, if the champion we have been given isn’t none, we show the champion placed id.

<script>

    //@ts-nocheck

    //export champion to parent component to set

    export let champion;

In the script tag we export champion, letting our parent component define it.

let color = "aquamarine"

    //if champion exists and team is 1

    $: if (champion && champion.team) {

        color = "#ffa895"

    }

    else {

        color = "#49ff2d"

    }

</script>

We also define our variable colour, for the style “background-color : {color}” on the button.

We default it to aquamarine, we then change it depending on if the champion exists and if it has a team, so the hexagon is visibly different depending on the team of the champion.

The $: syntax makes the statement reactive, whenever champion is changed/ updated, that code will rerun, updating the colour to the correct team.

Hexagon indent just features the left side of the hexagon:

<div class = "hex">

    <div class = "top"></div>

    <div class = "middle"></div>

    <div class = "bottom"></div>

</div>

<!--just an indent so every other hexagon row is correctly indented, invisible-->

<style>

    .hex {

        float: left;

        margin-left: 3px;

        margin-bottom: -26px;

    }

    .top {

        width: 0;

        border-bottom: 30px solid transparent;

        border-left: 50px solid transparent;

    }

</style>

Then, at the bottom of the each row statement:

<div class = "hex-row"></div>

We have a hex-row that simply ensures that each row starts on a new line.

function hex\_click(i, j) { //update selected champ

        selected\_champ = grid[i][j]

    }

Hex click is a simple function that updates the selected champion to whatever cell was clicked. We can then cover the left column:

<div class="column1\_noselect">

        {#if selected\_champ} <!--if there is selected champ, show details-->

            <h1>Champ: {selected\_champ.placed\_id}</h1>

            <h1>Type: {selected\_champ.id}</h1>

            <h1>Star Level: {selected\_champ.star}</h1>

            <h1>Items: {selected\_champ.items}</h1>

        {/if}

    </div>

If there is a selected champion, we simple display some stats of the champion in text. As this is a grid before a battle, we have placed champions that we are displaying, so we just have an id, type, star level and items to display. We initialise selected champ to an empty value:

let selected\_champ;

Finally we have the right column, where we display the past battles to show:

<div class="column3">

        <h1>Previous Results</h1>

        <h1>Click for more information</h1>

        {#await outcomes} <!--wait for outcomes from backend-->

            <h1>Fetching Results</h1>

        {:then outcomes} <!--display outcomes, button for each to view board-->

            {#each outcomes as outcome}

                {#if outcome[0] != 0}

                    <button on:click = {() => view\_battle(outcome[1])}>Match : {outcome[1]}, Winner : {outcome[0]}</button>

                {:else}

                    <button on:click = {() => view\_battle(outcome[1])}>Match : {outcome[1]}, Draw</button>

                {/if}

                <br>

            {/each}

        {/await}

    </div>

We start out with some h1 divs to inform the user this is the previous results column. As we have to fetch the outcomes from the backend, we use the await block, telling the user we are fetching outcomes. We fetch the previous outcomes with the associated ipc command:

let outcomes = fetch\_previous\_result() //fetch previous outcomes

//fetch list of previous results from ipc

    async function fetch\_previous\_result() {

        return await invoke("fetch\_outcomes")

    }

When we have our outcomes, we iterate through each one as outcome. The outcomes are returned to us as a tuple being : (match\_winner, match\_id), so we first check if the outcome wasn’t a draw. If not, we create a button, with the on\_click event being the view battle function with the argument being the board id, with the button text displaying the match id and also the match winner.

If the match winner is 0, ie a draw, we display a button with the text draw rather than winner.

Our view battle function takes in the battle id as an argument:

//fetch specific board

    async function view\_battle(battle\_id) {

        let outcome = await invoke("fetch\_outcome\_board", {id : battle\_id}) //fetch board

        for(let i = 0; i < outcome.length; i++) {

            outcome[i].team -= 1 //reduce team by 1, as they are stored as 1 larger than they should be in database

            outcome[i].placed\_id = generate\_id() //generate id

            grid[outcome[i].location.x][outcome[i].location.y] = outcome[i] //update grid with new champ

        }

    }

This fetches the specific details of a board to display on our grid.

We fetch the outcome using the fetch\_outcome\_board ipc command, which returns us a list of placed champions.

We then iterate through the placed champion with a standard for loop, initialising the index as 0 and increasing it every loop with the ++ operator until it reaches the length of outcome.

For each one, we decrease the team by 1 (as in the database its stored as 1, 2 whereas here hexagon expects team 0, 1), generate an id for it and then update the grid cell at its location, setting it to this placed champion.

Generate id is a function that tries to generate a unique letter id from a number. You can think of it the same way as rows are named in excel.

function generate\_id() {

        num\_placed\_champs += 1 //add 1 to placed champs

We have a “num\_placed\_champs” variable, that increases each time the function is called, which ensures that each champion gets a unique id.

let num\_placed\_champs = -1

We initialise num\_placed\_champs in the script tag with the value -1, so as the num\_placed\_champs variable is increased at the stat of the generate\_id function, its value will be 0 for the rest of the function.

function generate\_id() {

        num\_placed\_champs += 1 //add 1 to placed champs

        if(num\_placed\_champs == 0) { //cant log 0, return A

            return "A"

        }

Back in the generate id function, we check if num placed is 0, returning A if so, the first id.

let num\_to\_id = num\_placed\_champs

        let id\_str = "" //empty string

We create a copy of num placed champs with the num to id variable, and create an empty id string which is what we will return.

let cap = log\_base\_n(ids.length, num\_to\_id) // get numbers of characters needed

We then use a log base n function to calculate “cap”. Ids is a function with every character we intend to use in an id:

let ids = ['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L', 'M', 'N', 'O', 'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W', 'X', 'Y', 'Z']

So the letters of the alphabet here.

function log\_base\_n(base, number) {

        return Math.floor(Math.log(number) / Math.log(base));

    }

Javascript only have functionality for logging with a base of e, but we can take advantage of the fact that if we log both the number and the base and divide the two, it is equivalent to logging the number with a base of base. We then round down our result.

Cap is the number of letters that our id will be made up of, as with each additional letter the size of the number we can represent increases exponentially with our base.

for(let i = 0; i < cap + 1; i++) {

We then calculate each individual letter in the id

if(num\_to\_id == 0){ //cant log 0, add A to string

                id\_str += "A"

                break

            }

As we can’t log 0, we check if the remaining amount is 0, add A to the string, then break from the loop.

let log\_n = log\_base\_n(ids.length, num\_to\_id)

            let divisor = Math.pow(ids.length, log\_n)

            let id\_index = Math.floor(num\_to\_id / divisor)

            id\_str += ids[id\_index] //add new char

            num\_to\_id -= divisor \* id\_index

We then calculate the largest letter we can have in that slot. Finding the current log of num\_to\_id, calculating the divisor (as log\_base\_n is rounded down, we will have a smaller number). We then divide our num to id with our divisor, to find the index of this letter, before subtracting the divisor \* id\_index from the num\_to\_id.

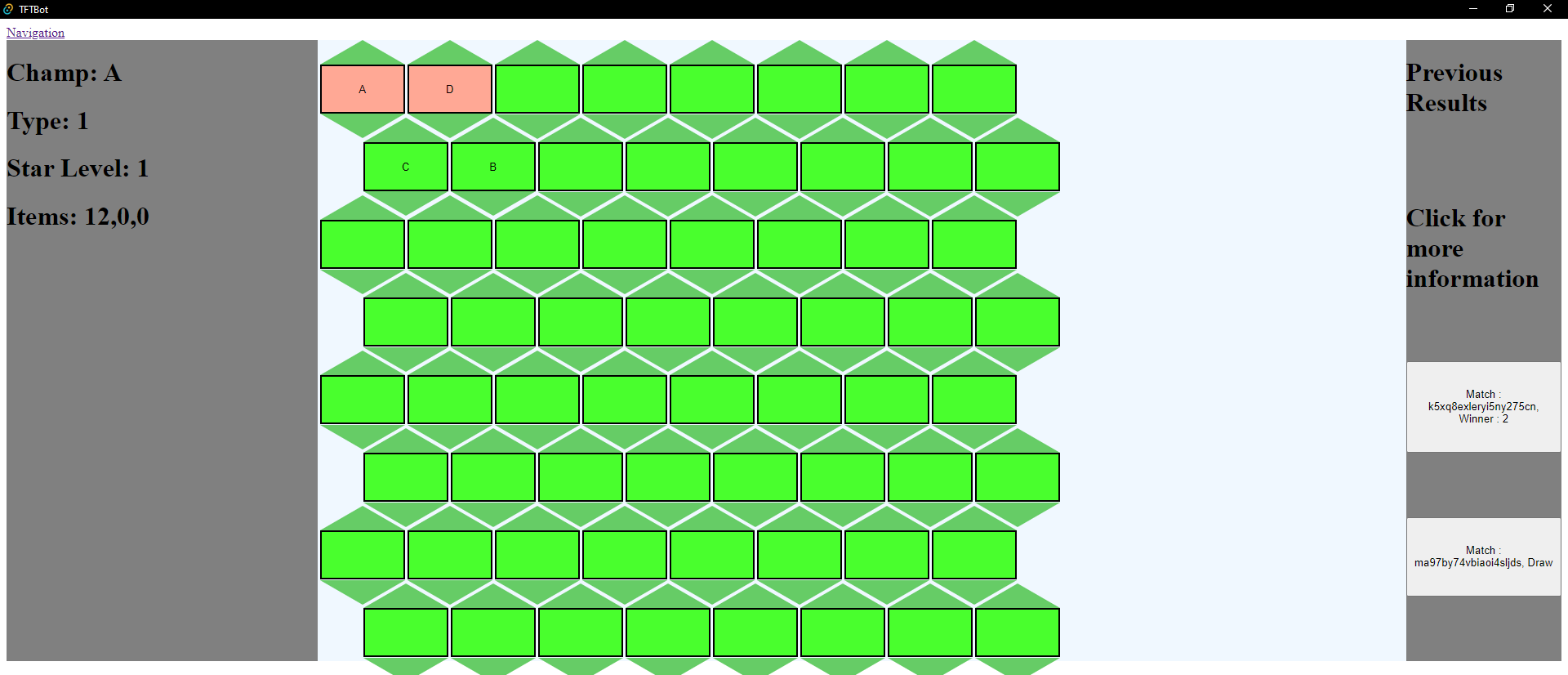
Once we’ve finished the loop, we return the new id:

}

        return id\_str //return ID

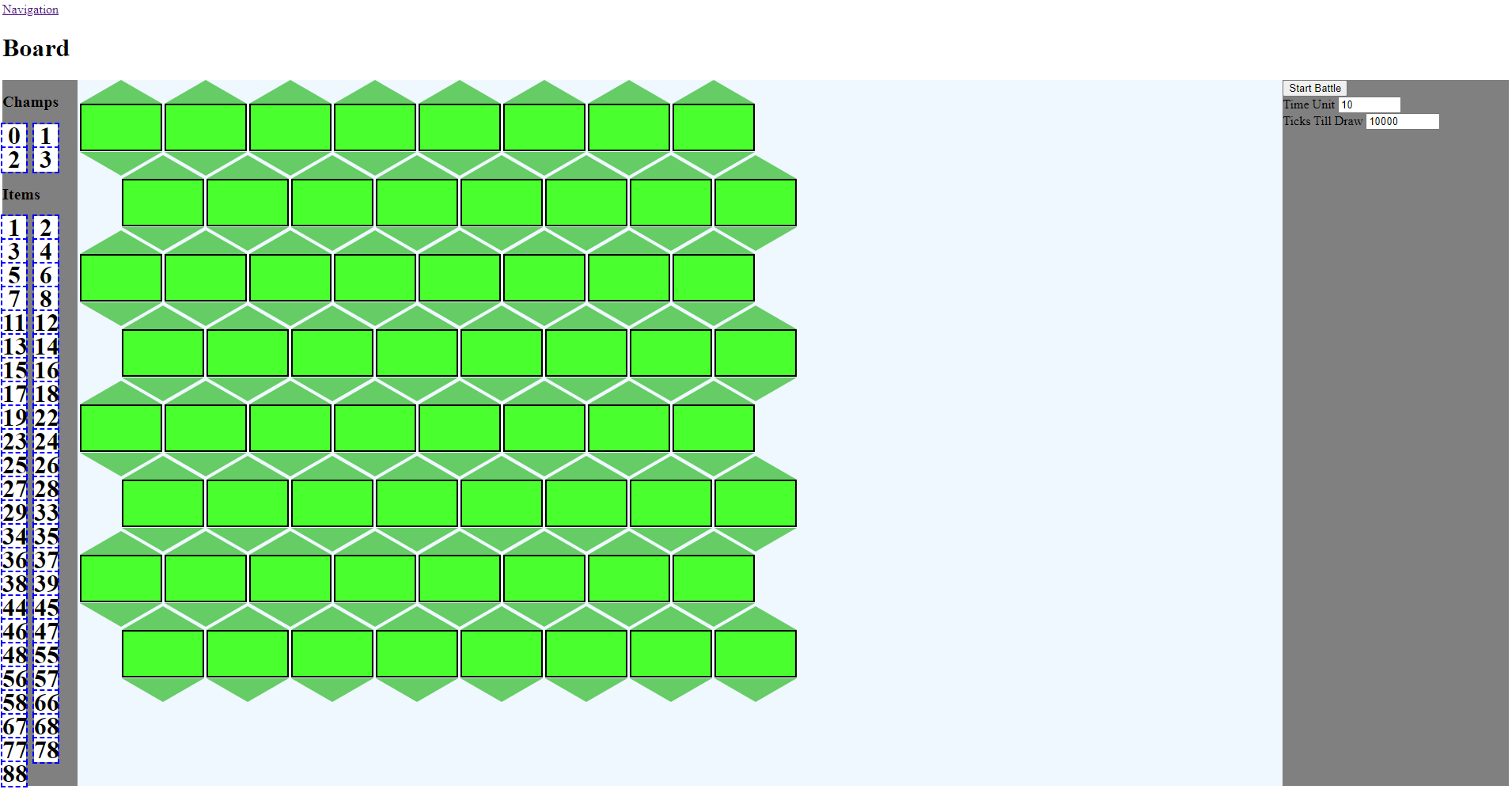
    }

With this we can correctly display past results. Our final result, when viewing a battle, looks like this:



Correctly displaying the initial state of the board.

We can then move onto the board page, on this page, the user is allowed to create the initial setup of a board:



We make use of the flex box and three separate columns once more:

<style>

    /\* create flex box for 3 column split \*/

    .row {

        display: flex;

    }

    /\*split column 1 into 2 columns

    give right space to each flex column

    \*/

    .column1\_noselect {

        flex: 5%;

        background-color: grey;

        display:grid;

        grid-template-columns: 40px 40px;

        user-select: none; /\* make sure user select disable to user doesnt "select"/ highlight a bunch of the drag and drop when moving \*/

    }

    .column3 {

        flex: 15%;

        background-color: grey;

        display:grid;

    }

    .column2 {

        flex : 80%;

        background-color: aliceblue;

    }

    .hex-row {

        clear: left;

    }

Our grid is much the same as with past results:

<div class="column2">

        {#each grid as row, i} <!--create hexagon grid-->

            {#if i % 2 == 1}

                <HexagonIndent></HexagonIndent>

            {/if}

            {#each row as hex, j} <!-- set on click to hex click location, on mouse up to on mouse up with location-->

                <Hexagon champion = {grid[i][j]} on:click = {() => hex\_click(i, j)} on:mouseup = {() => on\_mouse\_up(i, j)}></Hexagon>

            {/each}

            <div class = "hex-row"></div>

        {/each}

    </div>

The left side of our screen is a list of draggable list of items and champions, as laid down in design.

<div class="column1\_noselect"> <!--left column: create drag and drop for each chmap and item-->

        <h2 style = "font-size: 20px">Champs</h2><br>

        {#each champs\_list as champ} <!--bound on mouse down to on mouse down function with selected champ-->

            <UnitItemDragDrop champ\_or\_item = {champ} on:mousedown = {(e) => on\_mouse\_down(champ, e)}></UnitItemDragDrop>

        {/each}

        <h2 style = "font-size: 20px">Items</h2><br>

        {#each items\_list as item}

            <UnitItemDragDrop champ\_or\_item = {item} on:mousedown = {(e) => on\_mouse\_down(item, e)}></UnitItemDragDrop>

        {/each}

    </div>

We specify “user-select : none;” in the style for column1 so the user won’t select the entire left side of the screen while dragging a unit or item:

A picture containing diagram

Description automatically generated

Which looks poor and disrupts dragging the next item.

Next, in the column, we create the text to display champs and items, to divide the two, and create drag and drop components for each unit/ item:

<h2 style = "font-size: 20px">Champs</h2><br>

        {#each champs\_list as champ} <!--bound on mouse down to on mouse down function with selected champ-->

            <UnitItemDragDrop champ\_or\_item = {champ} on:mousedown = {(e) => on\_mouse\_down(champ, e)}></UnitItemDragDrop>

        {/each}

        <h2 style = "font-size: 20px">Items</h2><br>

        {#each items\_list as item}

            <UnitItemDragDrop champ\_or\_item = {item} on:mousedown = {(e) => on\_mouse\_down(item, e)}></UnitItemDragDrop>

        {/each}

We bind mouse\_down to the “on\_mouse\_down” function.

Champ and item lists are fetched from the ipc/ backend:

let champs\_list = []

    let items\_list = [] //initialise champ and item list

fetch\_champs() //fetch champs and items

    fetch\_items()

    //fetch champs and items func, invoke ipc method

    async function fetch\_champs() {

        champs\_list = await invoke("retrieve\_all\_units")

        selected\_champ\_or\_item = champs\_list[0]

    }

    async function fetch\_items() {

        items\_list = await invoke("retrieve\_all\_items")

    }

Let’s look at the unit item drag drop component. We start out in the script tag, exporting the champ or item variable to let the parent component control the value of the variable:

<script>

    // @ts-nocheck

    export let champ\_or\_item;

</script>

The rest of the component is fairly simple:

<!--listen to mousedown event, export behaviour to parent component, show champ or item id-->

<div on:mousedown>

    <h1>{champ\_or\_item.id}</h1>

</div>

<style>

    /\*white on black styling for clear display/ reading. outline for more clarity \*/

    div {

        outline: 2px dashed blue;

        background-color: white;

        width: 30px;

        height: 30px;

    }

    h1 {

        color: black;

        text-align: center;

        margin-top: -4px;

    }

</style>

We create a div with a mousedown listener (not bound to anything so the parent can define it), with the text of the champ or item id. By binding our on mouse down listen to the hexagon component (which is what we use to drop the champion/ item onto the board, we can validate/ ensure that the user only places champions inside the grid, ensuring that dropping one outside the area will not have an effect, ensuring no champions can be placed outside the grid).

In the style tag, we give a background to the div and set the text to black on a white background. We use margin-top: -4px to centre the text correctly.

However, the unit item drag drop component isn’t actually what is dragged. The on mouse down function that is called when the component is clicked on:

<UnitItemDragDrop champ\_or\_item = {champ} on:mousedown = {(e) => on\_mouse\_down(champ, e)}></UnitItemDragDrop>

Does this:

function on\_mouse\_down(champ, e) {

        //on mouse down of drag and drop

        selected\_champ\_or\_item = structuredClone(champ) //clone champ so as not to change by reference the original

        id = champ.id //update id

        show\_drag = 100 //show the drag object

        left = e.clientX + 5 //set position to mouse position + 5

        top = e.clientY + 5

    }

We update the selected champ or item variable, with a clone of the champion attached to the component (so any updates made on the board to that champ do not affect the one in the list). We set an id variable to the champ id, show\_drag to 100, and the variables left and top to “e.clientX + 5” etc, in this case, e is the mouse so we are setting left and top to the mouse position.

We have a separate div in the html, which is actually what we drag:

<div class = "drag\_container" style = "position: absolute; left: {left}px; top: {top}px; opacity: {show\_drag}"> <!--drag container, what shows when user is dragging a champ or item-->

    <h1>{id}</h1>

</div>

It is a box, with text id. We position it absolutely to the values of left and top, with the opacity of show\_drag.

The style for the drag container is :

.drag\_container {

        border: solid 1px gray;

        background-color: aliceblue;

    }

A simple box with a border. We then have the on mouse move function:

function on\_mouse\_move(e) {

        if (show\_drag == 100) { //if dragging something

            left += e.movementX; //add mouse movement

            top += e.movementY;

        }

    }

Which if we are showing a drag, moves the box along with the mouse movement, we set a base listener to this on the page, not to any component:

<svelte:window on:mousemove={on\_mouse\_move} />

Finally, on the hexagon, we also define a listener for on mouse up:

<Hexagon champion = {grid[i][j]} on:click = {() => hex\_click(i, j)} on:mouseup = {() => on\_mouse\_up(i, j)}></Hexagon>

Which calls the on mouse up function with the correct location of the hex.

//on mouse up on cell

    function on\_mouse\_up(i, j) {

        if (show\_drag == 100) { //hide drag

            show\_drag = 0

The on mouse up function takes in a location in the grid, it checks if show\_drag is 100 (ie we are dragging something), resetting it if so

if (selected\_champ\_or\_item.attack\_speed\_modifier) { //check if has attack\_speed\_modifier field, if soit is an item

We then check if there is the attribute “attack\_speed\_modifier” on the selected drag drop. If it has that attribute, then it is an item:

if (grid[i][j]) { //if there is a champion in that position

                    if (grid[i][j].items[0] == 0) { //update items array to first empty space, or 1st slot

                        grid[i][j].items[0] = selected\_champ\_or\_item.id

                    }

                    else if (grid[i][j].items[1] == 0) {

                        grid[i][j].items[1] = selected\_champ\_or\_item.id

                    }

                    else if (grid[i][j].items[2] == 0) {

                        grid[i][j].items[2] = selected\_champ\_or\_item.id

                    }

                    else {

                        grid[i][j].items[0] = selected\_champ\_or\_item.id

                    }

                }

                update\_show(i, j) //update the show, so shows a new item added

            }

So we check if there is a champion at that location in the grid, if so, we update the first empty item slot on the champion with the new one, or replaces the 1st item if there are no empty slots. We then call the update\_show function.

else {

                selected\_champ\_or\_item.items = new Array(0, 0, 0) //set items to empty array

                selected\_champ\_or\_item.star\_level = 1 //set star level to 1

                selected\_champ\_or\_item.placed\_id = generate\_id() //generate new id

                selected\_champ\_or\_item.team = false; //set team to team 1

                grid[i][j] = selected\_champ\_or\_item //update grid

            }

If the selected object is a champion, we replace whatever is at that location in the grid with our new champion, creating an empty item array, initialising the star level to 1, and generating a placed id (the function is the same as on the past results page). We also by default set the team to false, or 0, before updating that location in the grid.

What we do not have at the moment, is a way to view the champion placed at a grid location, change its team, delete it or increase its star level.

We create another div, only shown if the “show” variable has a value:

{#if show} <!--if show champ info-->

    <div style = "position: absolute; top: {pos[1]}px; left: {pos[0]}px" class = "info\_bar"> <!--show champ info at location-->

        <h1>Champ: {show.placed\_id}</h1>

        <h1>Type: {show.id}</h1>

        <h1>Star Level: {show.star\_level}</h1>

        <h1>Items: {show.items}</h1>

        <button on:click = {delete\_click}>Delete Unit</button>

        <button on:click = {increase\_star\_level}>Increase Star Level</button>

        <button on:click = {change\_team}>Change Team</button>

    </div>

{/if}

We position it relative to “pos”.

let pos = [0, 0];

let show\_pos = [0, 0] //set the pos being shown to 0, 0

We also initialise both pos and show pos to [0, 0]. Pos is the location (in pixels/ on the window) of the popup, whereas show pos is the grid location of the hex being shown.

Now we can look at the hex click function:

function hex\_click(i, j) {

        champ = grid[i][j] //set champ to clicked hexagon

        show\_pos = [i, j] //set show pos to position

        if (champ == show) { //if already shown, hide

            show = null

        }

        else {

            show = champ //update show

            pos = [80 + 130 \* j, 200 + 80 \* i]; //change pos depending on position in grid

        }

    }

We update champ to the value of the location at i, j, which is the champ being displayed. We set show pos to the grid position. Next, we check if champ is the same as show, this means we’ve clicked on the same hex twice in a row, so we should hide the currently shown champ, so we set show to null.

If it is not the same, we should update the champ being shown, we update it and set the position of the show box relative to its position in the grid, allowing the popup to be placed close to the cell.

We have some buttons on the popup, that do fairly self explanatory things.

Delete click deletes the champion at the location being shown:

function delete\_click() { //delete champ at grid location

        grid[show\_pos[0]][show\_pos[1]] = null; //set both grid and show to null

        show = null;

    }

And resets show.

function change\_team() {

        grid[show\_pos[0]][show\_pos[1]].team = !grid[show\_pos[0]][show\_pos[1]].team //swap team

    }

Change team swaps the team of the location.

function increase\_star\_level() {

        grid[show\_pos[0]][show\_pos[1]].star\_level += 1 //increase star level

        if (grid[show\_pos[0]][show\_pos[1]].star\_level > 3) { //make sure it loops around after 3

            grid[show\_pos[0]][show\_pos[1]].star\_level = 1

        }

        update\_show(show\_pos[0], show\_pos[1]) //call update show

    }

Increase star level increases the star level of the pos, but makes sure it doesn’t go above 3.

Update show is a function that is called whenever we make a visible change to the cell being shown, updating the popup to the accurate value:

function update\_show(i, j) {

        //if the cell that has been updated is the same as the cell currently being shown

        if (show\_pos[0] == i && show\_pos[1] == j && show) {

            show = grid[i][j] //update the show variable

        }

    }

It checks if we are showing something and if the cell updated is the one being shown, if so, it updates show to the grid that has been updated.

Now we can get onto the right column.

This column allows us to submit the board.

<div class="column3">

        <form on:submit={handle\_submit}> <!--create a form, with on submit = handle submit-->

            <button type = "submit">Start Battle</button><br>

            <label>Time Unit</label>

            <input type = "number" min = 1 max = 1000 bind:value = {time\_unit} required><br> <!-- create an input with min 1 max 1000 and type number, so only numeric inputs allow, also require for submission-->

            <label>Ticks Till Draw</label>

            <input type = "number" min = 100 max = 100000 bind:value = {time\_till\_draw} required> <!--create an input with similar checks to ensure value is valid-->

        </form>

    </div>

We create a form, to allow us to ensure that all values are present. We have input boxes for the time unit and ticks till draw, allowing the user to customise them. We use the “required” tag to make it so that the form cannot submit without them. We also define them as number inputs, so no non numeric characters can be added. We finally define “min” and “max” for them as well, to ensure that the values are reasonable, for example, we cannot have a negative time unit.

We also have labels for all the inputs so the user can tell what is what.

We have a submit button, which we bind to the handle\_submit function.

async function handle\_submit() { //send board to backend

        let player\_one\_champs = []

        let player\_two\_champs = []

        for (let i = 0; i < grid.length; i++) { //for each grid cell, if a champ is placed there, create a placed champ from the information and push to respective team

            for (let j = 0; j < grid[i].length; j++) {

                if (grid[i][j]) {

                    let placed\_champ = new PlacedChamp(grid[i][j].id, grid[i][j].items, grid[i][j].star\_level, new Location(i, j))

                    if (grid[i][j].team) {

                        player\_one\_champs.push(placed\_champ)

                    }

                    else {

                        player\_two\_champs.push(placed\_champ)

                    }

                }

            }

        } //submit board

        await invoke("submit\_board", {"playerOneChamps": player\_one\_champs, "playerTwoChamps": player\_two\_champs, "timeUnit": time\_unit, "timeTillDraw" : time\_till\_draw})

        window.location.href = "/board/battle" //move to battle page

    }

The handle submit function will only be called if the form is valid.

If it is, we iterate through each hex in the grid, using two for loops to go through each row and then each hex in the column. If there is a champion at that grid location, we create a new placed champion and add it to the correct team.

class PlacedChamp { //create placed champ class to send to backend

        constructor(id, items, star\_level, location) {

            this.id = id

            this.items = items

            this.star = star\_level

            this.location = location

            this.team = null

        }

    }

    class Location { //create location class to send to backend

        constructor(x, y) {

            this.x = x

            this.y = y

        }

    }

We create a simple constructor for each class on the frontend, so the backend gets the correct class/ objects it is expecting.

We then invoke the submit board method, and move to the battle page.

A screenshot of a computer

Description automatically generated with medium confidence

We have the same grid as before, same flex box:

<div class="column2"> <!--mid column-->

        {#each grid as row, i} <!--for each row and index in grid, put indent if odd row-->

            {#if i % 2 == 1}

                <HexagonIndent></HexagonIndent>

            {/if}

            {#each row as hex, j} <!-- create hexagon with champ at location i j-->

                <Hexagon champion = {grid[i][j]} on:click = {() => hex\_click(i, j)}></Hexagon>

            {/each}

            <div class = "hex-row"></div> <!-- create new row-->

        {/each}

    </div>

Hex click on this page changes the selected champion:

//set the selected champ to the champ at location

    function hex\_click(i, j) {

        selected\_champ = grid[i][j]

    }

Which is the champion whose details are shown on the left.

On page load, we fetch the board currently stored:

async function fetch\_board() {

        create\_grid() //creates a grid

        //fetches the board from backend

        board = await invoke("fetch\_board")

        //if one side has no champions, declare winner

        if (board.p1\_champions.length == 0 || board.p2\_champions.length == 0) {

            show\_over = true

            //show winner then hide after 2 seconds

            await new Promise(r => setTimeout(r, 2000));

            show\_over = false

        }

        //for champ in p1 champs

        for (let champ\_index in board.p1\_champions) {

            //get the location

            let location = board.p1\_champions[champ\_index].location

            board.p1\_champions[champ\_index].team = true //set the team to true/ team 1

            board.p1\_champions[champ\_index].placed\_id = generate\_id(board.p1\_champions[champ\_index].id) //generate an id for it

            grid[location.x][location.y] = board.p1\_champions[champ\_index] //set the grid positon at location to the new champ

        }

        //repeat for player 2 champs

        for (let champ\_index in board.p2\_champions) {

            let location = board.p2\_champions[champ\_index].location

            board.p2\_champions[champ\_index].team = false

            board.p2\_champions[champ\_index].placed\_id = generate\_id(board.p2\_champions[champ\_index].id)

            grid[location.x][location.y] = board.p2\_champions[champ\_index]

        }

    }

We set the grid to an empty grid, then fetch the board through the ipc.

We check if the board we have just fetched is over, ie there is a team without any alive champions.

If so we display a message saying the battle is over, wait for 2 seconds, then hide it again, so they can view the final battle board:

{#if show\_over} <!--show battle not over-->

    <div class = "show\_over">Battle Over!!</div>

{/if}

Show over is a simple div with the text battle over, with some css to position it in the middle of the screen with a big font:

.show\_over {

        position: absolute;

        top: 50%;

        left: 50%;

        background-color: grey;

        font-size: 60px;

    }

We then move to place all the champions on the board:

//for champ in p1 champs

        for (let champ\_index in board.p1\_champions) {

            //get the location

            let location = board.p1\_champions[champ\_index].location

            board.p1\_champions[champ\_index].team = true //set the team to true/ team 1

            board.p1\_champions[champ\_index].placed\_id = generate\_id(board.p1\_champions[champ\_index].id) //generate an id for it

            grid[location.x][location.y] = board.p1\_champions[champ\_index] //set the grid positon at location to the new champ

        }

        //repeat for player 2 champs

        for (let champ\_index in board.p2\_champions) {

            let location = board.p2\_champions[champ\_index].location

            board.p2\_champions[champ\_index].team = false

            board.p2\_champions[champ\_index].placed\_id = generate\_id(board.p2\_champions[champ\_index].id)

            grid[location.x][location.y] = board.p2\_champions[champ\_index]

        }

    }

We iterate through each index in the player 1 champions. We get their location, set their team, generate a display id for them, using their unique id from the backend, then place it in the correct location. We repeat this for player 2 champions, correctly displaying everything on the grid.

On the left, we have a column that displays all the details about the champion:

<div class="column1\_noselect"> <!--left column-->

        {#if selected\_champ} <!--if selected champ, display all stats on left side-->

            <title>Unit {selected\_champ.placed\_id}</title>

            <h1>Type: {selected\_champ.of\_champ\_id}</h1>

            <h1>Location: {selected\_champ.location.x} {selected\_champ.location.y}</h1>

            <h1>Movement Progress: {selected\_champ.movement\_progress}</h1>

            <h1>Target: {generate\_id(selected\_champ.target)}</h1>

            <h1>Status Effects: {selected\_champ.se.length}</h1>

            <h1>AP: {selected\_champ.ap.toFixed(2)}</h1>

            <h1>AD: {selected\_champ.ad.toFixed(2)}</h1>

            <h1>AR: {selected\_champ.ar.toFixed(2)}</h1>

            <h1>Attack Speed: {selected\_champ.attack\_speed.toFixed(2)}</h1>

            <h1>Attack Speed Mod: {selected\_champ.attack\_speed\_modifier.toFixed(2)}</h1>

            <h1>Auto Attack Delay: {selected\_champ.auto\_attack\_delay}</h1>

            <h1>Banished?: {selected\_champ.banish}</h1>

            <h1>Current Mana: {selected\_champ.cm}</h1>

            <h1>Crit Rate: {selected\_champ.cr}</h1>

            <h1>Crit Damage: {selected\_champ.crit\_damage.toFixed(2)}</h1>

            <h1>Dodge Chance: {selected\_champ.dc}</h1>

            <h1>Inc Damage Modifier: {selected\_champ.incoming\_damage\_modifier.toFixed(2)}</h1>

            <h1>HP: {selected\_champ.health.toFixed(2)}</h1>

            <h1>Initial HP: {selected\_champ.initial\_hp.toFixed(2)}</h1>

            <h1>Items: {selected\_champ.items}</h1>

            <h1>Mana Cost: {selected\_champ.mc}</h1>

            <h1>Shed: {selected\_champ.shed}</h1>

            <h1>Shields: {selected\_champ.shields.length}</h1>

            <h1>Target Cells: {selected\_champ.target\_cells.x} {selected\_champ.target\_cells.y}</h1>

            <h1>Target Cooldown: {selected\_champ.target\_cooldown}</h1>

            <h1>Targetable: {selected\_champ.targetable}</h1>

            <h1>Titan's Resolve Stacks: {selected\_champ.titans\_resolve\_stacks}</h1>

            <h1>Zap: {selected\_champ.zap}</h1>

        {/if}

    </div>

If there is a champion selected, it shows everything the user could want to see, ensuring they get the maximum accuracy possible. We round some values where necessary to 2 decimal places.

Finally, we have the right column, that controls the play of the battle:

<div class="column3"> <!--right column-->

        <button on:click = {play\_at}>Play</button> <!--inputs and buttons to control playback-->

        <button on:click = {pause}>Pause</button>

        <button on:click = {jump\_forward}>Jump Forward</button>

        <label>Play at</label>

        <input type = "number" min = 1 max = 20 bind:value = {play\_at\_ticks}>

        <label>Jump Forward</label>

        <input type = "number" min = 1 max = 20000 bind:value = {jump\_ticks\_num}>

        <button on:click = {save\_battle}>Save Result</button>

        <h1 style = "opacity: {show\_battle\_over}">Battle not over</h1>

    </div>

It has inputs, to change the speed at which the battle is played or the number of ticks to jump by.

We have a button, at the bottom, to save the result of the battle:

    async function save\_battle() {

        if (board.numTicks != board.ticks\_till\_draw  && (board.p1\_champions.length > 0 && board.p2\_champions.length > 0) ) { //if battle isn't over

            show\_battle\_over = 100 //show battle not over text

            await new Promise(r => setTimeout(r, 500));

            show\_battle\_over = 0

            return //return

        }

        let outcome = 0 //default outcome for draw

        if (board.p1\_champions.length > 0 && board.p2\_champions.length > 0) {

            outcome = 0 //set as draw

        }

        else if (board.p1\_champions.length > 0) {

            outcome = 1// set winner to p1

        }

        else {

            outcome = 2 //set winner to p2

        }

        invoke("update\_outcome", {outcome}) //update outcome

    }

Before we save the outcome, we check if the battle isn’t over. If the board isn’t at the draw tick count and both teams have champions left, we show the text stating the battle isn’t over yet, wait half a second, then hide it, returning early.

If the battle is over, we find out the winner, if both teams have champions left, we declare it a draw, otherwise we give it to the one with champions left, we then invoke the update outcome command.

Next, we have the play, pause and jump forward buttons:

<button on:click = {play\_at}>Play</button> <!--inputs and buttons to control playback-->

        <button on:click = {pause}>Pause</button>

        <button on:click = {jump\_forward}>Jump Forward</button>

Jump forward is the simplest:

async function jump\_forward() { //simulate x ticks then fetch board

        if (play) { return }

        await invoke("simulate\_x\_ticks", {numTicks : jump\_ticks\_num})

        await fetch\_board()

    }

Play is a variable that keeps track of whether we have “play” active right now, ticking forwards the board at a reasonable rate, if so, we shouldn’t jump forwards, as there is a big chance the board is busy/ being simulated right now, so we return early.

Otherwise, we simulate the ticks and then fetch the new board. The pause function is also simple, just setting play to false:

//pauses the playthrough

    function pause() {

        play = false;

    }

The play button is more complex

//plays the simulation

    async function play\_at() {

        if (play) { return } //if already playing, return

        play = true //set play to true

        while (play) { //while play

            let start = Date.now()

            await invoke("simulate\_x\_ticks", {numTicks : play\_at\_ticks})

            await fetch\_board() //simulate ticks

            let time\_taken = Date.now() - start //calculate time taken to simulate ticks and calculate how long to wait for another second

            if (time\_taken < 1000) {

                await new Promise(r => setTimeout(r, 1000 - time\_taken)); //wait until a second passes

            } //repeat until paused

        }

    }

If we are already playing, we return early, otherwise, we set play to true, to make sure no other play loops begin.

Then we have a while loop, while play. There is no place in the loop that play gets set to false, however, it isn’t an infinite loop, as due to the fact this is an asynchronous function and runs alongside other actions that can occur, the pause function can be called, which will also break us out of the play loop.

In the play loop, we get the start time, then simulate the play at ticks and fetch the new board.

Then we calculate the time that it took for us to do that and if the time taken was less than a second, we wait the remaining time in the second. This ensures that we will never wait for less than a second for a tick and allows the rate at which the board is updated is consistent, regardless of how long it takes us to simulate the ticks.

With that, we have covered the entirety of additions in prototype 3.

### Errors and Solutions:

**On click needs arguments (logic error):**

Simply binding:

<Hexagon on:click = {hex\_click}></Hexagon>

Doesn’t work as no arguments are given to the hex click function, meaning we have no idea where the click occurred.

**Solution:**

Using the

on:click = {() => hex\_click(i, j)}

syntax solves our problem, allowing us to pass in arguments.

**No champion.team (TypeError):**

Text

Description automatically generated

Using the code:

$: if (champion.team) {

        color = "#ffa895"

    }

    else {

        color = "#49ff2d"

    }

Fails to render any hexagons without a champion placed on them, as champion does not exist, so it doesn’t have a team attribute.

Graphical user interface

Description automatically generated with low confidence

**Solution:**

Check for both champion and champion.team:

 $: if (champion && champion.team) {

        color = "#ffa895"

    }

    else {

        color = "#49ff2d"

    }

So if champion is null, it simply returns the default color.

**Champion being placed down with item (logic error):**

When the user drags a champion onto the board, if the same champion has been dragged onto the board and given an item, the new dragged champion will also have the item.

selected\_champ\_or\_item = champ

This is because this sets selected champion to be a reference of champ, so when the selected champ is placed and altered, the original also gets altered permanently.

**Solution:**

Make a clone of the champion before dragging it:

selected\_champ\_or\_item = structuredClone(champ) //clone champ so as not to change by reference the original

**Mouse can’t click on hexagon (logic error):**

The mouse can’t be let up on a hexagon cell, as the drag and drop box is in the way of the cursor

A picture containing text, clipart

Description automatically generated

**Solution:**

Change:

left = e.clientX

        top = e.clientY

to:

left = e.clientX + 5 //set position to mouse position + 5

        top = e.clientY + 5

This slightly displaces the box, so it is no longer directly on the cursor.

**Can’t save a draw (logic error):**

The save battle function doesn’t allow a user to save a draw, as there are still champions left on one team.

**Solution:**

Check if the board ticks are equal to the board ticks till draw, so change:

if (board.p1\_champions.length > 0 && board.p2\_champions.length > 0) { //if battle isn't over

            show\_battle\_over = 100 //show battle not over text

            await new Promise(r => setTimeout(r, 500));

            show\_battle\_over = 0

            return //return

        }

To

if (board.numTicks != board.ticks\_till\_draw  && (board.p1\_champions.length > 0 && board.p2\_champions.length > 0) ) { //if battle isn't over

            show\_battle\_over = 100 //show battle not over text

            await new Promise(r => setTimeout(r, 500));

            show\_battle\_over = 0

            return //return

        }

To allow the user to save battles that ended in a draw.

**Store cannot be accessed mutably (reference error):**

Storing store in an Arc does not allow us mutable access to the store, so we cannot update the board stored:

Text

Description automatically generated

**Solution:**

Put store inside an RwLock, allowing us to request mutable access/ write access to the board, as well as read access.

### Test Table

|  |  |  |
| --- | --- | --- |
| **Test:** | Expected Outcome: | Outcome |
| **Program** |  |  |
| Should be resizable | The program window should be resizable within reason like you can with any other program. | Yes program window is resizable. |
| **Non-Navigation Pages:** |  |  |
| Visible link to navigation page on every page. | There should be a link to the navigation page on every page, clicking it should take you to the navigation page. | Yes link to navigation available on all pages. |
| **Navigation Page:** |  |  |
| Visible link to every page. | There should be a link to each other page in the program, clicking on it should take you to said page. | Yes link to all pages on navigation page. |
| **Board Page:** |  |  |
| On load | Should fetch a list of champions and items and create a drag and drop component for each of them, displaying them on a list on the left of the screen. | Yes list of item and champs on left side of screen |
| On load : create grid | On load, the page should create an 8 by 8 interactive hexagon grid. | Yes 8 by 8 interactive grid created on load. |
| Drag and drop a champion from the left | Dragging a champion from the left side of the page and dropping it on one of the hexagons should place the champion in that hex. | You can drag champions/ items onto the board. |
| Drag and drop a champion from the left onto occupied cell | Overwrite the champion in that cell with the new one. | Yes, it overwrites the previous one |
| Drag and drop an item from the left onto a hexagon with a champion | Update the item array with the new item that you added | Yes, it updates the array with the new one. |
| Drag and drop an item from the left onto a hexagon with a champion with 3 items. | Replace the first item in the items array with the new item. | Yes, with 3 items the first is updated when given a new one. |
| Drag and drop an item onto a hexagon without a champion. | No action | No action occurs. |
| Clicking on an occupied hexagon | A tooltip should pop up. If the tooltip is already visible elsewhere, it should move and update itself to the new hexagon. | Yes, a tooltip appears |
| The tooltip | The tooltip should have information on the champion such as its type and any items it has. | Tooltip has these. |
| Tooltip buttons | The tooltip should have buttons that allow the user to change the team of the unit, change its star level or delete it. | Tooltip has all of these. |
| Clicking on an unoccupied hexagon | Should have no effect or hide the currently displayed tooltip. | Hides the currently displayed tooltip |
| Inputs | On the right side of the screen, there should be inputs allowing the user to specify the time unit and ticks till draw of the board. | Yes |
| Start battle button | On the right side of the screen, there should be a start battle button.  If all inputs are valid the page should send the board to the backend to be stored/ simulated.  If any of the inputs are blank or invalid (negative time unit, non-numerical input etc), then the page should show an error and not submit. | All inputs are validated to ensure they hold reasonable/ valid values. |
| **Battle Page** |  |  |
| On load | Should fetch the stored board and display it on a grid. | Yes the stored board is retrieved and shown. |
| On press occupied cell | Should display the unit details on the left side of the page | Yes, unit details are shown on left. |
| On press play | Every second should ask the backend to simulate the set number of ticks specificed by the associated input and then retrieve and display the new board | Yes, every second the board is updated and retrieved, while play is pressed. |
| On press simulate X ticks | Should simulate X ticks and retrieve/ display new board. | Yes, simulates the provided number of ticks, returning the new board, but only if play isn’t pressed. |
| **Change Unit and Item Stats** |  |  |
| On load | On load the page should fetch a list of all the units and items. | List of units and items are fetched. |
| Input boxes | There should be a list of inputs lining up to all the adjustable stats of a unit or item | Yes, list of inputs for champs and items for all adjustable stats. |
| ID Dropdown | There should be a dropdown list of all the ids of units or items. Upon selecting one of them, it should autofill the inputs with the saved stats of the unit or item. | Dropdown is available and autofills all input boxes. |
| Submit unit/ item | The page should check the input boxes have valid data in them (in correct range, not empty etc), if so it should submit the unit/ item to the IPC to be altered. | Values are checked to ensure they are valid by the frontend. |
| **Previous Results Page** |  |  |
| On load | The page should fetch a list of past results and display buttons for each of them on the right side of the screen | A list is correctly created with the match ids and their outcome. |
| Press button to load previous result | A board should be created with the same PlacedChampions as on the board, showing the initial state of the board. | The initial state of the board is displayed |
| On press occupied hexagon | Should load unit details on the left side of the screen. | With unit details on the left on click. |

### Validation Table:

|  |  |  |
| --- | --- | --- |
| To Validate: | Outcome | Proof: |
| User cannot drop unit not on cell |  |  |
| Dropping item on empty cell has no effect |  |  |
| User dropping unit between two cells snaps to one |  |  |
| --input validation-- |  |  |
|  |  |  |

### Review of Prototype

In summary of the prototype, I am overall happy with the finished product and final version of my project. It has fulfilled all the prototype requirements and accomplished its aim of providing a way for users to interact with the simulator through a graphical user interface, as well as providing a way for the user to alter units and items within the interface. My client, Suket, is happy with the product, finally able to easily simulate battles without having to do it through code, and is also pleased with the usability features provided.

# Evaluation:

## Post-Development Testing:

**The user can create a board through only drag and drop components and simple click mechanics:**

Yes, it is possible for the user to create a board through drag and drop mechanics, dragging in champions and items from the left, then clicking start battle button. This ensures that the program is easy to access, making it accessible to people of all experience with computers, including those whose only game they play is TFT and who have no other experience with computers. It also makes creating boards quick, important for those in a game who need to learn how they can improve their position, an example mentioned in analysis.

Diagram

Description automatically generated

A picture containing diagram

Description automatically generated

**The simulation/ program will be efficient and quick, being able to simulate a complex board (as defined earlier as board which contains more than 3 units on each team, each with at least 1 item) in under 2 seconds:**

This was tested at the end of prototype 2, but will be resimulated through the user interface for this prototype.

<https://youtu.be/YPmkbWzp44E>

As shown, the battle occurs almost instantly, even being more complex than the definition of a complex battle and going to 100,000 ticks.

Important to provide a good user experience for the user, avoiding long wait times and periods of delay, especially when simulating lots of battles in a short period of time, or in a time important scenario such as during a game.

This was one of the requirements laid out in analysis.

**The user will be able to change unit and item stats easily through the UI:**

The user can change unit and item stats through the ui on the change unit and item stats page, which allows the user more flexibility to experiment within the confines of TFT. It also provides a way for the user to update the program/ improve its accuracy after updates and before I get the chance to do it myself, allowing the user to take matters into their own hands.

**The user will be able to view previous boards and their result:**

The user can view past results through the past results page, allowing users to build up a catalogue of results that they can refer back to during matches and giving them the ability to learn specific matchups and store that knowledge for later use.

**The program will still meet or exceed all the criteria set out in previous prototypes:**

The simulator is unchanged from prototype 2, so all previous criteria will still succeed, and all the requirements for prototype 3 was fulfilled.

## Usability Features Testing:

**Text clearly defined:**

Links are blue and underlined, on a white background:

A picture containing graphical user interface

Description automatically generated

And normal text is commonly if not always black on a white background:

Table

Description automatically generated

Graphical user interface, text, application

Description automatically generated

Or a bold black on a light grey:

Text

Description automatically generated

This allows users to clearly and easily read text, especially useful for those with visual impairments so they are no disadvantaged using this application.

**Hexagons on grid with an outline:**

A picture containing athletic game, sport

Description automatically generated

Hexagons have a clear outline, allowing the users to clearly distinguish between hexes, avoiding confusion and aiding those with visual impairments.

**Buttons Clear Outline:**

A picture containing application

Description automatically generated

Buttons have a clear outline, giving the users more clarity on where the buttons are, avoiding confusion within the program.

**Program will be resizable:**Yes, the program is resizable. This allows the user to run the program on a wide variety of monitor sizes, allowing access on most devices that can run TFT, improving the accessibility of the program.

**Units and Items will be draggable:**

Yes, units and items are draggable, this provides a very easy way to create battles quickly and efficiently, vastly improving user experience over a method that requires multiple clicks or manually typing in hex coordinates.

**Clickable Hexagon cells:**

Yes, all hexagons are interactable for extra detail, allowing the user extra information on whatever occupies the cell. This means that the user can investigate battles more in depth, providing more valuable learning opportunities for them and giving them an idea of what occurs behind the scenes, much easier than searching through a list of champions to find the correct one.

**Modifiable speed of simulation playback:**Yes, you can simulate at a certain tick rate per second, or jump forward, this means users can watch the battle at their own pace, slowly, to get a solid understanding of the battle when they are first learning, or quickly to view the outcome or broad strokes of the battle. The user can also pause the battle whenever they like, allowing them to step away as needed and does not require them to give the program their undivided attention for however long.

**Checked input fields:**All input fields are checked, from the unit change stats to the submit board ones, this prevents user errors from breaking the program and increases the robustness of the program.

**Efficient and robust program:**

The program can simulate complex battles incredibly quickly, as demonstrated in the testing for prototype 2, this allows the user to run the program on most hardware, even with low processing power, improving accessibility, whilst also avoiding user annoyance and disatistfaction from low speeds. This has been successfully accomplished as shown in the demonstration/ prototype 2 test.

When the user selects a unit/ item to change, the stat input boxes will autofill:

While true, that when the user selects a unit or item, the input boxes autofill, due to floating point errors from the database, which alters values slightly from 0.4 to 0.3999 etc, and due to the fact that the input boxes do not accept values with such precision/ as many decimal places, they often require altering to the rounded value manually by the user, which can be annoying, so I will mark as partially complete.

To solve this issue, all values could be rounded whether required or not, to the decimal precision allowed, meaning that users can more quickly alter stats and without as much hassle.

You can queue up multiple, different battles at once to do one after another.

This ended up being a hassle for users, as scheduling battles still required users to build the board before the battle, not proving to be much faster. The UI for scheduling the battles was also clunky and got in the way therefore not improving usability and instead being an obstruction.

If in the future, users want to simulate a similar battle over and over again for testing purposes, the frontend could save the last board created and recreate it automatically when the user returns to the page. However, this could also prove a hassle if users want a different board, as they would have to manually delete all units. Either we could implement alongside it a clear board button, or we could have a list of recently created boards or templates that the user could alter/ create and then use, allowing quick iteration of similar boards.

When selecting from the list of past results to display, the user will be able to see what units were on each team, so they can select the correct one.

This feature was not implemented as there was not room on the button. The buttons were too large and the text ended up being too small to read, especially with lots of results saved. It ended up taking required space from the board which lowered clarity and ended up having the opposite effect, as users could not read the buttons anyway, so it was removed.

To work around this issue, small icons could be created, which would be more graphically pleasing and clear, giving a clearer idea for the user. However, as the units do not have items currently, creating the graphics for them would be a large ask/ time requirement.

All usability features implemented can be seen in the demonstration below.

## Demonstration:

A demonstration of the program is visible here:

<https://youtu.be/4T5fxnH0Z64>

demonstrating key features of the program that is hard to represent in an image format.

## Success Criteria Evaluation:

For any “proofs” with video evidence, time stamps are provided to the relevant section of the video.

* 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.
  + 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.
    - Board is placed in the middle of the page
    - Displays hexagons on the board, placed units and their items.
    - Users can click on units to change their items and star levels (although usersdo not change their items by clicking on the unit, rather dragging and dropping them). <https://youtu.be/a1Fo72gkjgI?t=47>
    - There is not a move button, as I deemed it an unnecessary complication, using time I did not have available, instead this functionality can be achieved through the delete button and replacing the unit.
    - In future development, this feature could be implemented through clicking and holding down click on a champion, allowing it then to be dragged to a different cell.
    - Proof:
      * <https://youtu.be/a1Fo72gkjgI?t=2>
  + At least 3 units and most (more than half) of the current items will be implemented, along with associated status effects.
    - 4 units are implemented, all current TFT items are implemented, along with associated status effects.
    - Proof:
      * Appendix, <https://youtu.be/a1Fo72gkjgI?t=2>
  + 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.
    - Proof:
      * <https://youtu.be/a1Fo72gkjgI?t=2>
  + It will not let you drag an item onto a cell that does not contain a unit.
    - Proof:
      * <https://youtu.be/a1Fo72gkjgI?t=13>
  + 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.
    - Proof:
      * Graphical user interface, text

        Description automatically generated
      * <https://youtu.be/a1Fo72gkjgI?t=27>
  + 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.
    - Start battle begins the simulation
    - It is paused on first frame. <https://youtu.be/a1Fo72gkjgI?t=55>
    - Clicking on units display health, stats effects and other stats. <https://youtu.be/a1Fo72gkjgI?t=58>
    - On the right side of the screen simulation settings such as play/ pause buttons are displayed. <https://youtu.be/a1Fo72gkjgI?t=104>
    - Battle is not pre-simulated.
    - If users reaches end, they are informed. <https://youtu.be/a1Fo72gkjgI?t=93>
    - Projectiles will also be displayed on the board during the battle:
      * Projectiles are not displayed on the board, due to issues with collisions displaying a projectile where a champion is also currently placed in that position (and they are on the same team, so the projectile doesn’t explode). Due to this and time constraints, I decided against keeping this success criteria.
      * In future developments, I could develop a system that displays a projectile if the cell is uncontested, but if it is contested, it hides the projectile, but clicking on the cell should show the projectile details under the champion details.
  + 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.
    - The board updates every second, rather than a few times every second. I chose to do this to minimise the number of trips the board has to make between the frontend and backend, which is an expensive/ time consuming operation, which involves pausing the current simulation and copying the board. It is more effective instead to do a large amount of simulation at once then pause and send back the board, then repeatedly/ frequently doing a little simulation and sending it back. <https://youtu.be/a1Fo72gkjgI?t=58>
    - In future development if users wanted this feature to be implemented, work could be done to only broadcast the changes made to a board during a tick, reducing the time taken for a board to be sent through the ipc and allowing faster/ more rapid updates.
  + 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.
    - All boards are automatically saved and there is a save results button that is available to press once the battle is done.
    - Proof:
      * <https://youtu.be/a1Fo72gkjgI?t=112>
* The simulation will have an acceptable level of accuracy to the game TFT
  + To be more definitive, the program should be able to successfully state the winner of a battle a minimum of 70% of the time.
  + Proof:
    - [Prototype 2 test table](#Prototype2TestTable).
* The program should be able to simulate 3 complex battles in under 10 seconds on a low specification computer.
  + Proof:
    - [Prototype 2 test table](#Prototype2TestTable).
* Through the UI, the user will be able to change unit and item stats.
  + 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.
  + Proof:
    - <https://youtu.be/a1Fo72gkjgI?t=136>
* Through the UI, the user will be able to view previous simulations.
  + 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.
    - Proof:
    - <https://youtu.be/a1Fo72gkjgI?t=120>
  + 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.

# Maintenance:

Maintenance for the project will be hard, as it has to keep up with the development and updates to TFT, a tall task for a game with a full development team. The compartmentalised/ encapsulated nature of the program will mean that updating specific sections will be very easy and provides a way to separate debugging certain sections from each other. Any improvements to the user interface, database or simulator can be separated from one another.

Moreover, with little change, the simulator can be separated from the rest of the program, providing a way for other uses to add their own interface to the program, or improve it/ implement more features on their own.

Future updates could be done to implement more champions into the game, or traits and augments, improving the accuracy and utility of the program.

# Appendix:

## Frontend/ Javascript Svelte File structure:

Text

Description automatically generated

**Hexagon.svelte:**

<script>

    //@ts-nocheck

    //export champion to parent component to set

    export let champion;

    let color = "aquamarine"

    //if champion exists and team is 1

    $: if (champion && champion.team) {

        color = "#ffa895"

    }

    else {

        color = "#49ff2d"

    }

</script>

<div class = "hex">

    <div class = "top"></div>

    <!--create a button with background color : color, changes dependant on team

        set button to listen to on:click and on:mouseup, parent componenent can define behaviour

        if there is a champion, show the champion id

    -->

    <button on:click on:mouseup style = "background-color: {color}">

        {#if champion}

            {champion.placed\_id}

        {/if}

    </button>

    <div class = "bottom"></div>

</div>

<!--

    hexagon shape css taken from:

    https://stackoverflow.com/questions/17896791/hexagon-shape-with-css3

-->

<style>

    .hex {

        float: left;

        margin-left: 3px;

        margin-bottom: -26px; /\* sets float to left, sets margin\*/

    }

    .top {

        width: 0;

        border-bottom: 30px solid #6C6;

        border-left: 52px solid transparent;

        border-right: 52px solid transparent;

        z-index: 2;

    }

    button {

        width: 104px;

        height: 60px;

        background: #6C6;

        z-index: 2;

    }

    .bottom {

        width: 0;

        border-top: 30px solid #6C6;

        border-left: 52px solid transparent;

        border-right: 52px solid transparent;

        z-index: 2;

    }

</style>

**HexagonIndent.svelte:**

<div class = "hex">

    <div class = "top"></div>

    <div class = "middle"></div>

    <div class = "bottom"></div>

</div>

<!--just an indent so every other hexagon row is correctly indented, invisible-->

<style>

    .hex {

        float: left;

        margin-left: 3px;

        margin-bottom: -26px;

    }

    .top {

        width: 0;

        border-bottom: 30px solid transparent;

        border-left: 50px solid transparent;

    }

</style>

**UnitItemDragDrop.svelte:**

<script>

    // @ts-nocheck

    export let champ\_or\_item;

</script>

<!--listen to mousedown event, export behaviour to parent component, show champ or item id-->

<div on:mousedown>

    <h1>{champ\_or\_item.id}</h1>

</div>

<style>

    /\*white on black styling for clear display/ reading. outline for more clarity \*/

    div {

        outline: 2px dashed blue;

        background-color: white;

        width: 30px;

        height: 30px;

    }

    h1 {

        color: black;

        text-align: center;

        margin-top: -4px;

    }

</style>

**Battle : +page.svelte:**

<script>

    // @ts-nocheck

    //import hexagon and hexagon indent

    import Hexagon from "$lib/Hexagon.svelte";

    import HexagonIndent from "$lib/HexagonIndent.svelte";

    import { invoke } from "@tauri-apps/api/tauri"

    let play = false; //let currently playing be false

    //alphabet for id list

    let ids = ['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L', 'M', 'N', 'O', 'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W', 'X', 'Y', 'Z']

    //creates an empty grid, 2d array 8x8

    function create\_grid() {

        grid = new Array(8) //creates empty 8 by 8 array

        for(var i = 0; i < grid.length; i++) {

            grid[i] = new Array(8) //for i in grid, set i to an array 8 long

        }

    }

    //logs number with base N

    function log\_base\_n(base, number) {

        return Math.floor(Math.log(number) / Math.log(base));

    }

    //generates an id given a num

    function generate\_id(num\_to\_id) {

        if(num\_to\_id == 0) { //return A

            return "A"

        }

        if(num\_to\_id == 255) {

            return "None"

        }

        let id\_str = "" //initalise string to empty

        let cap = log\_base\_n(ids.length, num\_to\_id) //get number of characters

        for(let i = 0; i < cap + 1; i++) { //for each character

            if(num\_to\_id == 0){

                id\_str += "A" //add A as you cannot logarithm 0

                break

            }

            //get log of num\_to\_id with ids.length, rounded down

            let log\_n = log\_base\_n(ids.length, num\_to\_id)

            //get divisor

            let divisor = Math.pow(ids.length, log\_n)

            //get id from divisor

            let id\_index = Math.floor(num\_to\_id / divisor)

            id\_str += ids[id\_index]

            //minus divisor

            num\_to\_id -= divisor \* id\_index

        }

        //return new string

        return id\_str

    }

    //set the selected champ to the champ at location

    function hex\_click(i, j) {

        selected\_champ = grid[i][j]

    }

    async function fetch\_board() {

        create\_grid() //creates a grid

        //fetches the board from backend

        board = await invoke("fetch\_board")

        //if one side has no champions, declare winner

        if (board.p1\_champions.length == 0 || board.p2\_champions.length == 0 || board.ticks\_till\_draw == board.tick\_count) {

            show\_over = true

            //show winner then hide after 2 seconds

            await new Promise(r => setTimeout(r, 2000));

            show\_over = false

        }

        //for champ in p1 champs

        for (let champ\_index in board.p1\_champions) {

            //get the location

            let location = board.p1\_champions[champ\_index].location

            board.p1\_champions[champ\_index].team = true //set the team to true/ team 1

            board.p1\_champions[champ\_index].placed\_id = generate\_id(board.p1\_champions[champ\_index].id) //generate an id for it

            grid[location.x][location.y] = board.p1\_champions[champ\_index] //set the grid positon at location to the new champ

        }

        //repeat for player 2 champs

        for (let champ\_index in board.p2\_champions) {

            let location = board.p2\_champions[champ\_index].location

            board.p2\_champions[champ\_index].team = false

            board.p2\_champions[champ\_index].placed\_id = generate\_id(board.p2\_champions[champ\_index].id)

            grid[location.x][location.y] = board.p2\_champions[champ\_index]

        }

    }

    //pauses the playthrough

    function pause() {

        play = false;

    }

    //plays the simulation

    async function play\_at() {

        if (play) { return } //if already playing, return

        play = true //set play to true

        while (play) { //while play

            let start = Date.now()

            await invoke("simulate\_x\_ticks", {numTicks : play\_at\_ticks})

            await fetch\_board() //simulate ticks

            let time\_taken = Date.now() - start //calculate time taken to simulate ticks and calculate how long to wait for another second

            if (time\_taken < 1000) {

                await new Promise(r => setTimeout(r, 1000 - time\_taken)); //wait until a second passes

            } //repeat until paused

        }

    }

    async function jump\_forward() { //simulate x ticks then fetch board

        if (play) { return }

        await invoke("simulate\_x\_ticks", {numTicks : jump\_ticks\_num})

        await fetch\_board()

    }

    async function save\_battle() {

        if (board.tick\_count != board.ticks\_till\_draw  && (board.p1\_champions.length > 0 && board.p2\_champions.length > 0) ) { //if battle isn't over

            show\_battle\_over = 100 //show battle not over text

            await new Promise(r => setTimeout(r, 500));

            show\_battle\_over = 0

            return //return

        }

        let outcome = 0 //default outcome for draw

        if (board.p1\_champions.length > 0 && board.p2\_champions.length > 0) {

            outcome = 0 //set as draw

        }

        else if (board.p1\_champions.length > 0) {

            outcome = 1// set winner to p1

        }

        else {

            outcome = 2 //set winner to p2

        }

        invoke("update\_outcome", {outcome}) //update outcome

    }

    let grid; //initialise empty grid

    let play\_at\_ticks = 5; //set play at ticks to 5

    let jump\_ticks\_num = 100; //set jump ticks to 100

    let board; //initialise empty board

    let selected\_champ; //initialise empty selected champ

    let show\_battle\_over = 0 //set show battle over and show over to false

    let show\_over = false;

    fetch\_board() //fetch board

</script>

<div class="row">

    <div class="column1\_noselect"> <!--left column-->

        {#if selected\_champ} <!--if selected champ, display all stats on left side-->

            <title>Unit {selected\_champ.placed\_id}</title>

            <h1>Type: {selected\_champ.of\_champ\_id}</h1>

            <h1>Location: {selected\_champ.location.x} {selected\_champ.location.y}</h1>

            <h1>Movement Progress: {selected\_champ.movement\_progress}</h1>

            <h1>Target: {generate\_id(selected\_champ.target)}</h1>

            <h1>Status Effects: {selected\_champ.se.length}</h1>

            <h1>AP: {selected\_champ.ap.toFixed(2)}</h1>

            <h1>AD: {selected\_champ.ad.toFixed(2)}</h1>

            <h1>AR: {selected\_champ.ar.toFixed(2)}</h1>

            <h1>Attack Speed: {selected\_champ.attack\_speed.toFixed(2)}</h1>

            <h1>Attack Speed Mod: {selected\_champ.attack\_speed\_modifier.toFixed(2)}</h1>

            <h1>Auto Attack Delay: {selected\_champ.auto\_attack\_delay}</h1>

            <h1>Banished?: {selected\_champ.banish}</h1>

            <h1>Current Mana: {selected\_champ.cm}</h1>

            <h1>Crit Rate: {selected\_champ.cr}</h1>

            <h1>Crit Damage: {selected\_champ.crit\_damage.toFixed(2)}</h1>

            <h1>Dodge Chance: {selected\_champ.dc}</h1>

            <h1>Inc Damage Modifier: {selected\_champ.incoming\_damage\_modifier.toFixed(2)}</h1>

            <h1>HP: {selected\_champ.health.toFixed(2)}</h1>

            <h1>Initial HP: {selected\_champ.initial\_hp.toFixed(2)}</h1>

            <h1>Items: {selected\_champ.items}</h1>

            <h1>Mana Cost: {selected\_champ.mc}</h1>

            <h1>Shed: {selected\_champ.shed}</h1>

            <h1>Shields: {selected\_champ.shields.length}</h1>

            <h1>Target Cells: {selected\_champ.target\_cells.x} {selected\_champ.target\_cells.y}</h1>

            <h1>Target Cooldown: {selected\_champ.target\_cooldown}</h1>

            <h1>Targetable: {selected\_champ.targetable}</h1>

            <h1>Titan's Resolve Stacks: {selected\_champ.titans\_resolve\_stacks}</h1>

            <h1>Zap: {selected\_champ.zap}</h1>

        {/if}

    </div>

    <div class="column2"> <!--mid column-->

        {#each grid as row, i} <!--for each row and index in grid, put indent if odd row-->

            {#if i % 2 == 1}

                <HexagonIndent></HexagonIndent>

            {/if}

            {#each row as hex, j} <!-- create hexagon with champ at location i j-->

                <Hexagon champion = {grid[i][j]} on:click = {() => hex\_click(i, j)}></Hexagon>

            {/each}

            <div class = "hex-row"></div> <!-- create new row-->

        {/each}

    </div>

    <div class="column3"> <!--right column-->

        <button on:click = {play\_at}>Play</button> <!--inputs and buttons to control playback-->

        <button on:click = {pause}>Pause</button>

        <button on:click = {jump\_forward}>Jump Forward</button>

        <label>Play at</label>

        <input type = "number" min = 1 max = 20 bind:value = {play\_at\_ticks}>

        <label>Jump Forward</label>

        <input type = "number" min = 1 max = 20000 bind:value = {jump\_ticks\_num}>

        <button on:click = {save\_battle}>Save Result</button>

        <h1 style = "opacity: {show\_battle\_over}">Battle not over</h1>

    </div>

</div>

{#if show\_over} <!--show battle not over-->

    <div class = "show\_over">Battle Over!!</div>

{/if}

<style>

    .row { /\* flex display\*/

        display: flex;

    }

    /\* set column flexs to correct width \*/

    .column1\_noselect {

        flex: 20%;

        background-color: grey;

        user-select: none;

    }

    .column3 {

        flex: 5%;

        background-color: grey;

        display:grid;

    }

    .column2 {

        flex : 75%;

        background-color: aliceblue;

    }

    .hex-row {

        clear: left;

    }

    h1 {

        color: black;

        padding-bottom: 0px;

        margin-bottom: 0px;

        font-size: 20px;

        line-height: 10px;

    }

    .show\_over {

        position: absolute;

        top: 50%;

        left: 50%;

        background-color: grey;

        font-size: 60px;

    }

</style>

**Board : +page.svelte:**

<script>

    // @ts-nocheck

    //component imports

    import UnitItemDragDrop from "$lib/UnitItemDragDrop.svelte";

    import Hexagon from "$lib/Hexagon.svelte";

    import HexagonIndent from "$lib/HexagonIndent.svelte";

    import { invoke } from "@tauri-apps/api/tauri"

    //all ids

    let ids = ['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L', 'M', 'N', 'O', 'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W', 'X', 'Y', 'Z']

    //set default values for time unit and time till draw

    let time\_unit = 10

    let time\_till\_draw = 100000

    //fetch champs and items func, invoke ipc method

    async function fetch\_champs() {

        champs\_list = await invoke("retrieve\_all\_units")

        selected\_champ\_or\_item = champs\_list[0]

    }

    async function fetch\_items() {

        items\_list = await invoke("retrieve\_all\_items")

    }

    function increase\_star\_level() {

        grid[show\_pos[0]][show\_pos[1]].star\_level += 1 //increase star level

        if (grid[show\_pos[0]][show\_pos[1]].star\_level > 3) { //make sure it loops around after 3

            grid[show\_pos[0]][show\_pos[1]].star\_level = 1

        }

        update\_show(show\_pos[0], show\_pos[1]) //call update show

    }

    function change\_team() {

        grid[show\_pos[0]][show\_pos[1]].team = !grid[show\_pos[0]][show\_pos[1]].team //swap team

    }

    function hex\_click(i, j) {

        champ = grid[i][j] //set champ to clicked hexagon

        show\_pos = [i, j] //set show pos to position

        if (champ == show) { //if already shown, hide

            show = null

        }

        else {

            show = champ //update show

            pos = [80 + 130 \* j, 200 + 80 \* i]; //change pos depending on position in grid

        }

    }

    //creates empty grid, 2d array 8x8

    function create\_grid() {

        let grid = new Array(8)

        for(var i = 0; i < grid.length; i++) {

            grid[i] = new Array(8)

        }

        return grid

    }

    function on\_mouse\_down(champ, e) {

        //on mouse down of drag and drop

        selected\_champ\_or\_item = structuredClone(champ) //clone champ so as not to change by reference the original

        id = champ.id //update id

        show\_drag = 100 //show the drag object

        left = e.clientX + 5 //set position to mouse position + 5

        top = e.clientY + 5

    }

    //on mouse up on cell

    function on\_mouse\_up(i, j) {

        if (show\_drag == 100) { //hide drag

            show\_drag = 0

            if (selected\_champ\_or\_item.attack\_speed\_modifier) { //check if has attack\_speed\_modifier field, if soit is an item

                if (grid[i][j]) { //if there is a champion in that position

                    if (grid[i][j].items[0] == 0) { //update items array to first empty space, or 1st slot

                        grid[i][j].items[0] = selected\_champ\_or\_item.id

                    }

                    else if (grid[i][j].items[1] == 0) {

                        grid[i][j].items[1] = selected\_champ\_or\_item.id

                    }

                    else if (grid[i][j].items[2] == 0) {

                        grid[i][j].items[2] = selected\_champ\_or\_item.id

                    }

                    else {

                        grid[i][j].items[0] = selected\_champ\_or\_item.id

                    }

                }

                update\_show(i, j) //update the show, so shows a new item added

            }

            else {

                selected\_champ\_or\_item.items = new Array(0, 0, 0) //set items to empty array

                selected\_champ\_or\_item.star\_level = 1 //set star level to 1

                selected\_champ\_or\_item.placed\_id = generate\_id() //generate new id

                selected\_champ\_or\_item.team = false; //set team to team 1

                grid[i][j] = selected\_champ\_or\_item //update grid

            }

        }

    }

    function on\_mouse\_move(e) {

        if (show\_drag == 100) { //if dragging something

            left += e.movementX; //add mouse movement

            top += e.movementY;

        }

    }

    function delete\_click() { //delete champ at grid location

        grid[show\_pos[0]][show\_pos[1]] = null; //set both grid and show to null

        show = null;

    }

    function update\_show(i, j) {

        //if the cell that has been updated is the same as the cell currently being shown

        if (show\_pos[0] == i && show\_pos[1] == j && show) {

            show = grid[i][j] //update the show variable

        }

    }

    function log\_base\_n(base, number) {

        return Math.floor(Math.log(number) / Math.log(base));

    }

    function generate\_id() {

        num\_placed\_champs += 1 //add a new placed champ

        if(num\_placed\_champs == 0) { //can't log 0, so return A

            return "A"

        }

        let num\_to\_id = num\_placed\_champs

        let id\_str = "" //empty string

        let cap = log\_base\_n(ids.length, num\_to\_id) //get number of cells

        for(let i = 0; i < cap + 1; i++) {

            if(num\_to\_id == 0){

                id\_str += "A" //cant log 0, so get add A

                break

            }

            let log\_n = log\_base\_n(ids.length, num\_to\_id) //get log of num\_to\_id by ids.length rounded down

            let divisor = Math.pow(ids.length, log\_n) //calculate divisor

            let id\_index = Math.floor(num\_to\_id / divisor) //get index

            id\_str += ids[id\_index] //add char to string

            num\_to\_id -= divisor \* id\_index //remove num from num to id

        }

        return id\_str

    }

    let show\_pos = [0, 0] //set the pos being shown to 0, 0

    let id = 0 //set id to 0

    let grid = create\_grid() //create the grid

    let champs\_list = []

    let items\_list = [] //initialise champ and item list

    let show; //initialise show

    let num\_placed\_champs = -1 //set num placed champs to -1

    let show\_drag = 0;

    let pos = [0, 0];

    let selected\_champ\_or\_item;

    let left = 0;

    let top = 0;

    fetch\_champs() //fetch champs and items

    fetch\_items()

    class PlacedChamp { //create placed champ class to send to backend

        constructor(id, items, star\_level, location) {

            this.id = id

            this.items = items

            this.star = star\_level - 1

            this.location = location

            this.team = null

        }

    }

    class Location { //create location class to send to backend

        constructor(x, y) {

            this.x = x

            this.y = y

        }

    }

    async function handle\_submit() { //send board to backend

        let player\_one\_champs = []

        let player\_two\_champs = []

        for (let i = 0; i < grid.length; i++) { //for each grid cell, if a champ is placed there, create a placed champ from the information and push to respective team

            for (let j = 0; j < grid[i].length; j++) {

                if (grid[i][j]) {

                    let placed\_champ = new PlacedChamp(grid[i][j].id, grid[i][j].items, grid[i][j].star\_level, new Location(i, j))

                    if (grid[i][j].team) {

                        player\_one\_champs.push(placed\_champ)

                    }

                    else {

                        player\_two\_champs.push(placed\_champ)

                    }

                }

            }

        } //submit board

        await invoke("submit\_board", {"playerOneChamps": player\_one\_champs, "playerTwoChamps": player\_two\_champs, "timeUnit": time\_unit, "timeTillDraw" : time\_till\_draw})

        window.location.href = "/board/battle" //move to battle page

    }

    let champ = champs\_list[0]

</script>

<svelte:window on:mousemove={on\_mouse\_move} />

<h1>Board</h1>

<div class="row">

    <div class="column1\_noselect"> <!--left column: create drag and drop for each chmap and item-->

        <h2 style = "font-size: 20px">Champs</h2><br>

        {#each champs\_list as champ} <!--bound on mouse down to on mouse down function with selected champ-->

            <UnitItemDragDrop champ\_or\_item = {champ} on:mousedown = {(e) => on\_mouse\_down(champ, e)}></UnitItemDragDrop>

        {/each}

        <h2 style = "font-size: 20px">Items</h2><br>

        {#each items\_list as item}

            <UnitItemDragDrop champ\_or\_item = {item} on:mousedown = {(e) => on\_mouse\_down(item, e)}></UnitItemDragDrop>

        {/each}

    </div>

    <div class="column2">

        {#each grid as row, i} <!--create hexagon grid-->

            {#if i % 2 == 1}

                <HexagonIndent></HexagonIndent>

            {/if}

            {#each row as hex, j} <!-- set on click to hex click location, on mouse up to on mouse up with location-->

                <Hexagon champion = {grid[i][j]} on:click = {() => hex\_click(i, j)} on:mouseup = {() => on\_mouse\_up(i, j)}></Hexagon>

            {/each}

            <div class = "hex-row"></div>

        {/each}

    </div>

    <div class="column3">

        <form on:submit={handle\_submit}> <!--create a form, with on submit = handle submit-->

            <button type = "submit">Start Battle</button><br>

            <label>Time Unit</label>

            <input type = "number" min = 1 max = 1000 bind:value = {time\_unit} required><br> <!-- create an input with min 1 max 1000 and type number, so only numeric inputs allow, also require for submission-->

            <label>Ticks Till Draw</label>

            <input type = "number" min = 100 max = 100000 bind:value = {time\_till\_draw} required> <!--create an input with similar checks to ensure value is valid-->

        </form>

    </div>

</div>

<div class = "drag\_container" style = "position: absolute; left: {left}px; top: {top}px; opacity: {show\_drag}"> <!--drag container, what shows when user is dragging a champ or item-->

    <h1>{id}</h1>

</div>

{#if show} <!--if show champ info-->

    <div style = "position: absolute; top: {pos[1]}px; left: {pos[0]}px" class = "info\_bar"> <!--show champ info at location-->

        <h1>Champ: {show.placed\_id}</h1>

        <h1>Type: {show.id}</h1>

        <h1>Star Level: {show.star\_level}</h1>

        <h1>Items: {show.items}</h1>

        <button on:click = {delete\_click}>Delete Unit</button>

        <button on:click = {increase\_star\_level}>Increase Star Level</button>

        <button on:click = {change\_team}>Change Team</button>

    </div>

{/if}

<style>

    /\* create flex box for 3 column split \*/

    .row {

        display: flex;

    }

    /\*split column 1 into 2 columns

    give right space to each flex column

    \*/

    .column1\_noselect {

        flex: 5%;

        background-color: grey;

        display:grid;

        grid-template-columns: 40px 40px;

        user-select: none; /\* make sure user select disable to user doesnt "select"/ highlight a bunch of the drag and drop when moving \*/

    }

    .column3 {

        flex: 15%;

        background-color: grey;

        display:grid;

    }

    .column2 {

        flex : 80%;

        background-color: aliceblue;

    }

    .hex-row {

        clear: left;

    }

    .info\_bar {

        width: 140px;

        height: 160px;

        font-size: 10px;

        background-color: aliceblue;

        z-index: 3;

    }

    .drag\_container {

        border: solid 1px gray;

        background-color: aliceblue;

    }

</style>

**Change stats : +page.svelte**

<script>

    // @ts-nocheck

    import { invoke } from "@tauri-apps/api/tauri"

    async function fetch\_champs() {

        champs\_list = await invoke("retrieve\_all\_units")

        return champs\_list[0]

    }

    async function fetch\_items() {

        items\_list = await invoke("retrieve\_all\_items")

        return items\_list[0]

    } //fetch champs and items

    let champs\_list = []

    let items\_list = []

    let opacity\_champ\_error = 0 //set errors for champ and items opacity to 0

    let opacity\_item\_error = 0

    //initialise champs and items list and fetch them

    let selected\_unit = fetch\_champs() //initialise selected unit and item

    let selected\_item = fetch\_items()

    async function handle\_submit\_update\_champ (e) {

        if (check\_valid\_champ(selected\_unit)) { //if the selected unit is valid, update the unit

            await invoke("update\_unit", {selectedUnit : selected\_unit})

            fetch\_champs() //fetch new champ

            opacity\_champ\_error = 0 //hide error

        }

        else {

            opacity\_champ\_error = 100 //show champ error

        }

    }

    function check\_valid\_champ(champ) { //makes sure champ values are reasonable/ will not cause errors

        if (champ.ad < 0 || champ.ad > 9999) { return false }

        if (champ.hp <= 0 || champ.health > 9999) { return false }

        if (champ.attack\_speed < 0 || champ.attack\_speed > 20) { return false }

        if (champ.ar <= 0 || champ.ar > 9999) { return false }

        if (champ.mr <= 0 || champ.mr > 9999) { return false }

        if (champ.mc <= 0 || champ.mc > 9999) { return false }

        if (champ.ra <= 0 || champ.ra > 20) { return false }

        if (champ.sm <= 0 || champ.sm > 9999) { return false }

        return true

    }

    function check\_valid\_item(item) { //make sure item values are reasonable/ will not cause errors

        if (item.ad < 0 || item.ad > 9999) { return false }

        if (item.ap < 0 || item.ap > 9999) { return false }

        if (item.health < 0 || item.health > 9999) { return false }

        if (item.ar < 0 || item.ar > 9999) { return false }

        if (item.mr < 0 || item.mr > 9999) { return false }

        if (item.attack\_speed\_modifier < 0 || item.attack\_speed\_modifier > 20) { return false }

        if (item.cm < 0 || item.cm > 9999) { return false }

        if (item.cr < 0 || item.cr > 100) { return false }

        if (item.ra < 0 || item.ra > 20) { return false }

        if (item.dc < 0 || item.dc > 100) { return false }

        if (item.omnivamp < 0 || item.omnivamp > 9999) { return false}

        if (item.crit\_damage < 0 || item.crit\_damage > 9999) { return false }

        return true

    }

    async function handle\_submit\_update\_item (e) {

        if (check\_valid\_item(selected\_item)) { //if item is valid

            await invoke("update\_item", {selectedItem : selected\_item}) //update item

            fetch\_items() //fetch new item

            opacity\_item\_error = 0 //hide item erorr

        }

        else {

            opacity\_item\_error = 100 //show item error

        }

    }

</script>

{#await selected\_item}

    <div>Loading...</div>

{:then}

<div class="row">

    <div class="column"> <!--split page into two-->

        <h1>Change Unit Stats</h1>

        <form on:submit|preventDefault={handle\_submit\_update\_champ}> <!--create form-->

            <label>ID</label>

            <select bind:value = {selected\_unit} required>

                {#each champs\_list as champ}

                    <option value = {champ}>

                        {champ.id}

                    </option>

                {/each}

            </select>

            <br>

            <label>AD</label> <!--create inputs for all required values-->

            <input type= "number" bind:value="{selected\_unit.ad}" step = 0.01 required><br>

            <label>HP</label>

            <input type="number" bind:value="{selected\_unit.hp}" step = 0.01 required><br>

            <label>Attack Speed</label>

            <input type="number" bind:value="{selected\_unit.attack\_speed}" step = 0.01 required><br>

            <label>Ar</label>

            <input type="number" bind:value="{selected\_unit.ar}" step = 0.01 required><br>

            <label>Mr</label>

            <input type="number" bind:value="{selected\_unit.mr}" step = 0.01 required><br>

            <label>Mc</label>

            <input type="number" bind:value="{selected\_unit.mc}" required><br>

            <label>Ra</label>

            <input type="number" bind:value="{selected\_unit.ra}" required><br>

            <label>Sm</label>

            <input type="number" bind:value="{selected\_unit.sm}" required><br>

            <button type = "submit">Submit</button>

            <h1 style="opacity : {opacity\_champ\_error}">Invalid values for some or all variables, please try again.</h1> <!--create error that is hidden most of the time-->

        </form>

    </div>

    <div class="column">

        <h1>Change Item Stats</h1>

        <form on:submit|preventDefault={handle\_submit\_update\_item}> <!--create form for item stats-->

            <label>ID</label><!--create inputs for all required values-->

            <select bind:value = {selected\_item} required>

                {#each items\_list as item}

                    <option value = {item}>

                        {item.id}

                    </option>

                {/each}

            </select>

            <br>

            <label>AD</label>

            <input type= "number" bind:value="{selected\_item.ad}" step = 0.01 required><br>

            <label>AP</label>

            <input type= "number" bind:value="{selected\_item.ap}" step = 0.01 required><br>

            <label>Ar</label>

            <input type= "number" bind:value="{selected\_item.ar}" step = 0.01 required><br>

            <label>Attack\_speed\_modifier</label>

            <input type= "number" bind:value="{selected\_item.attack\_speed\_modifier}" step = 0.01 required><br>

            <label>CM</label>

            <input type= "number" bind:value="{selected\_item.cm}" required><br>

            <label>Cr</label>

            <input type= "number" bind:value="{selected\_item.cr}" required><br>

            <label>crit\_damage</label>

            <input type= "number" bind:value="{selected\_item.crit\_damage}" step = 0.01 required><br>

            <label>Dc</label>

            <input type= "number" bind:value="{selected\_item.dc}" required><br>

            <label>Health</label>

            <input type= "number" bind:value="{selected\_item.health}" step = 0.01 required><br>

            <label>mr</label>

            <input type= "number" bind:value="{selected\_item.mr}" step = 0.01 required><br>

            <label>omnivamp</label>

            <input type= "number" bind:value="{selected\_item.omnivamp}" step = 0.01 required><br>

            <label>Range</label>

            <input type= "number" bind:value="{selected\_item.ra}"required><br>

            <button type = "submit">Submit</button>

            <h1 style="opacity : {opacity\_item\_error}">Invalid values for some or all variables, please try again.</h1> <!--create error that is hidden most of the time-->

        </form>

    </div>

</div>

{/await}

<style>

    /\* create flex box and set each column to 50% of page \*/

    .row {

        display: flex;

    }

    .column {

        flex: 50%;

    }

</style>

**Past Results: +page.svelte**

<script>

    // @ts-nocheck

    //import required components

    import { invoke } from "@tauri-apps/api/tauri";

    import Hexagon from "$lib/Hexagon.svelte";

    import HexagonIndent from "$lib/HexagonIndent.svelte";

    //create array of ids

    let ids = ['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L', 'M', 'N', 'O', 'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W', 'X', 'Y', 'Z']

    function create\_grid() { //create empty grid

        let grid = new Array(8)

        for(var i = 0; i < grid.length; i++) {

            grid[i] = new Array(8)

        }

        return grid

    }

    //fetch list of previous results from ipc

    async function fetch\_previous\_result() {

        return await invoke("fetch\_outcomes")

    }

    //fetch specific board

    async function view\_battle(battle\_id) {

        let outcome = await invoke("fetch\_outcome\_board", {id : battle\_id}) //fetch board

        for(let i = 0; i < outcome.length; i++) {

            outcome[i].team -= 1 //reduce team by 1, as they are stored as 1 larger than they should be in database

            outcome[i].placed\_id = generate\_id() //generate id

            grid[outcome[i].location.x][outcome[i].location.y] = outcome[i] //update grid with new champ

        }

    }

    function log\_base\_n(base, number) {

        return Math.floor(Math.log(number) / Math.log(base));

    }

    function generate\_id() {

        num\_placed\_champs += 1 //add 1 to placed champs

        if(num\_placed\_champs == 0) { //cant log 0, return A

            return "A"

        }

        let num\_to\_id = num\_placed\_champs

        let id\_str = "" //empty string

        let cap = log\_base\_n(ids.length, num\_to\_id) // get numbers of characters needed

        for(let i = 0; i < cap + 1; i++) {

            if(num\_to\_id == 0){ //cant log 0, add A to string

                id\_str += "A"

                break

            }

            let log\_n = log\_base\_n(ids.length, num\_to\_id)

            let divisor = Math.pow(ids.length, log\_n)

            let id\_index = Math.floor(num\_to\_id / divisor)

            id\_str += ids[id\_index] //add new char

            num\_to\_id -= divisor \* id\_index

        }

        return id\_str //return ID

    }

    let num\_placed\_champs = -1

    function hex\_click(i, j) { //update selected champ

        selected\_champ = grid[i][j]

    }

    let outcomes = fetch\_previous\_result() //fetch previous outcomes

    let grid = create\_grid() //create empty grid

    let selected\_champ;

</script>

<div class="row">

    <div class="column1\_noselect">

        {#if selected\_champ} <!--if there is selected champ, show details-->

            <h1>Champ: {selected\_champ.placed\_id}</h1>

            <h1>Type: {selected\_champ.id}</h1>

            <h1>Star Level: {selected\_champ.star}</h1>

            <h1>Items: {selected\_champ.items}</h1>

        {/if}

    </div>

    <div class="column2">

        {#each grid as row, i}

            {#if i % 2 == 1}

                <HexagonIndent></HexagonIndent> <!--indent every other line-->

            {/if}

            {#each row as hex, j} <!-- create hexagon grid-->

                <Hexagon champion = {grid[i][j]} on:click = {() => hex\_click(i, j)}></Hexagon>

            {/each}

            <div class = "hex-row"></div>

        {/each}

    </div>

    <div class="column3">

        <h1>Previous Results</h1>

        <h1>Click for more information</h1>

        {#await outcomes} <!--wait for outcomes from backend-->

            <h1>Fetching Results</h1>

        {:then outcomes} <!--display outcomes, button for each to view board-->

            {#each outcomes as outcome}

                {#if outcome[0] != 0}

                    <button on:click = {() => view\_battle(outcome[1])}>Match : {outcome[1]}, Winner : {outcome[0]}</button>

                {:else}

                    <button on:click = {() => view\_battle(outcome[1])}>Match : {outcome[1]}, Draw</button>

                {/if}

                <br>

            {/each}

        {/await}

    </div>

</div>

<style>

    .row {

        display: flex;

    }

    .column1\_noselect {

        flex: 20%;

        background-color: grey;

        user-select: none;

    }

    .column3 {

        flex: 10%;

        background-color: grey;

        display:grid;

    }

    .column2 {

        flex : 70%;

        background-color: aliceblue;

    }

    .hex-row {

        clear: left;

    }

</style>

**+layout.svelte:**

<script>

  import "../app.css";

</script>

<main class="container">

  <!--create a link to navigation that is acccessible everywhere, by placing in layout, will be present on all pages-->

  <a href="../">Navigation</a> <br />

  <slot />

</main>

**+page.svelte:**

<br />

<a href="board">Board</a>

<br />

<a href="change\_stats">Change Unit and Item Stats</a>

<br />

<a href="past\_results">Previous Results</a>

<!--navigation page is base page, with links to all other pages available-->

## Backend/ Rust Tauri File structure:

Text

Description automatically generated

**Board.rs:**

// Import Serialize from serde so Board can be sent to frontend.

use serde::Serialize;

// Import the necessary types from other modules.

use super::champions::{PlacedChampion, SummonedChampion, Champion};

use super::item::Item;

use super::projectiles::Projectile;

// Import format, VecDeque and Rng.

use core::fmt;

use std::collections::VecDeque;

use rand::Rng;

/// Constant for movement amount.

const MOVEMENT\_AMOUNT\_CONST: i8 = 10;

/// Board Struct:<br/>

/// Simulates Battles

#[derive(Clone, Serialize)]

pub struct Board {

    /// A vector deque (double ended vector) of player 1's summoned champions.

    p1\_champions: VecDeque<SummonedChampion>,

    /// A vector deque of player 2's summoned champions.

    p2\_champions: VecDeque<SummonedChampion>,

    /// Time unit for board in centiseconds (1/100 of a second).

    time\_unit: i8,

    /// Movement amount per tick, is calculated by const \* time unit.

    movement\_amount: i8,

    /// Number of ticks until the battle is declared a draw.

    ticks\_till\_draw : u32,

    /// The current count/ number of ticks that has passed.

    tick\_count : u32,

    /// A vector of player 1's projectiles.

    p1\_projectiles: Vec<Projectile>,

    /// A vector of player 2's projectiles.

    p2\_projectiles: Vec<Projectile>,

    /// A vector deque of the dead summoned champions.

    dead\_champs: VecDeque<SummonedChampion>

}

impl Board {

    ///Creates a new board

    pub fn new (p1\_placed\_champs: &VecDeque<PlacedChampion>, p2\_placed\_champs: &VecDeque<PlacedChampion>, champions : &[Champion], items : &[Item], time\_unit: i8, ticks\_till\_draw : u32) -> Board {

        info!("New Board");

        //creates empty vecdeque's for player 1s and 2s champs.

        let mut p1\_champions = VecDeque::new();

        let mut p2\_champions = VecDeque::new();

        //gets number of p1's champions.

        let len: usize = p1\_placed\_champs.len();

        info!("Creating Champions");

        //for each champ in p1's placed champ, push a summoned champion to p1\_champions

        for (i, p1\_champ) in p1\_placed\_champs.iter().enumerate() {

            p1\_champions.push\_back(SummonedChampion::new(p1\_champ, i)); //converts into summoned champ

        }

        //repeat for player 2

        for (i, p2\_champ) in p2\_placed\_champs.iter().enumerate() {

            //adds length of p1\_placed\_champs to id to ensure they have unique id.

            p2\_champions.push\_back(SummonedChampion::new(p2\_champ, i + len)); //converts into summoned champ

        }

        //pop each champ from the p1\_champion, set up the champ and then push back.

        for \_ in 0..p1\_champions.len() {

            let mut champ = p1\_champions.pop\_front().unwrap();

            champ.setup(&mut p1\_champions, &mut p2\_champions, champions, items);

            p1\_champions.push\_back(champ);

        }

        //repeat for player 2

        for \_ in 0..p2\_champions.len() {

            let mut champ = p2\_champions.pop\_front().unwrap();

            champ.setup(&mut p2\_champions, &mut p1\_champions, champions, items);

            p2\_champions.push\_back(champ);

        }

        info!("Champions Created");

        //return new board

        Board {

            p1\_champions,

            p2\_champions,

            time\_unit,

            movement\_amount: MOVEMENT\_AMOUNT\_CONST \* time\_unit / 10, //(!O)

            ticks\_till\_draw,

            tick\_count: 0,

            p1\_projectiles: Vec::new(),

            p2\_projectiles: Vec::new(),

            dead\_champs: VecDeque::new(),

        }

    }

    ///Generates a random new board

    pub fn generate\_random\_board(time\_unit: i8, champions : &[Champion], items : &[Item], ticks\_till\_draw : u32) -> Board {

        //randomly selects the number of player 1's and 2's champions in the range 1 to 6

        let num\_p1\_champs: usize = rand::thread\_rng().gen\_range(1..6);

        let num\_p2\_champs: usize = rand::thread\_rng().gen\_range(1..6);

        //fetches all item ids

        let item\_ids : Vec<u8> = items.iter().map(|f| f.id).collect();

        let id\_range = champions.len();

        //for each champ, generate a random placed champion

        let p1\_champions: VecDeque<PlacedChampion> = (0..num\_p1\_champs)

            .map(|\_ : usize| PlacedChampion::generate\_random\_champ(id\_range, &item\_ids, false))

            .collect();

        let p2\_champions: VecDeque<PlacedChampion> = (num\_p1\_champs

            ..num\_p1\_champs + num\_p2\_champs)

            .map(|\_ : usize| PlacedChampion::generate\_random\_champ(id\_range, &item\_ids, true))

            .collect();

        //create new board

        Board::new(&p1\_champions, &p2\_champions, champions, items, time\_unit, ticks\_till\_draw)

    }

    pub fn generate\_complex\_random\_board(time\_unit: i8, champions : &[Champion], items : &[Item], ticks\_till\_draw : u32) -> Board {

        //randomly selects the number of player 1's and 2's champions in the range 3 to 6

        let num\_p1\_champs: usize = rand::thread\_rng().gen\_range(3..6);

        let num\_p2\_champs: usize = rand::thread\_rng().gen\_range(3..6);

        //fetches all item ids

        let item\_ids : Vec<u8> = items.iter().map(|f| f.id).collect();

        let id\_range = champions.len();

        //for each champ, generate a random placed champion

        let p1\_champions: VecDeque<PlacedChampion> = (0..num\_p1\_champs)

            .map(|\_ : usize| PlacedChampion::generate\_random\_champ(id\_range, &item\_ids, false))

            .collect();

        let p2\_champions: VecDeque<PlacedChampion> = (num\_p1\_champs

            ..num\_p1\_champs + num\_p2\_champs)

            .map(|\_ : usize| PlacedChampion::generate\_random\_champ(id\_range, &item\_ids, true))

            .collect();

        //create new board

        Board::new(&p1\_champions, &p2\_champions, champions, items, time\_unit, ticks\_till\_draw)

    }

    ///simulates a battle from self, with an option input of ticks to simulate.

    pub fn simulate\_battle(&mut self, ticks\_to\_simulate : Option<u32>) {

        info!("Starting Battle");

        let upper = match ticks\_to\_simulate {

            Some(cnt) => self.ticks\_till\_draw.min(self.tick\_count + cnt), //simulates battle till draw or for the tick count, whatever comes first.

            None => self.ticks\_till\_draw //if none simnulates entire battle till draw

        };

        //for each tick

        for \_ in self.tick\_count..upper

        {

            info!("Battle Iteration : {}", self.tick\_count);

            self.tick\_count += 1; //increment ticks

            info!("Taking Champion Turns");

            for \_champ\_count in 0..self.p1\_champions.len() {

                //take turn for all p1Champs

                //pop front champ

                let mut this\_champ = self.p1\_champions.pop\_front().unwrap();

                //take turn for champ

                let alive = this\_champ.take\_turn(

                    &mut self.p1\_champions,

                    &mut self.p2\_champions,

                    self.time\_unit,

                    self.movement\_amount,

                    &mut self.p1\_projectiles

                );

                if alive {

                    //if alive push to back of p1\_champs

                    self.p1\_champions.push\_back(this\_champ);

                } else {

                    //else push to dead champs

                    self.dead\_champs.push\_back(this\_champ);

                }

            }

            //repeat for p2 champions

            for \_champ\_count in 0..self.p2\_champions.len() {

                let mut this\_champ = self.p2\_champions.pop\_front().unwrap();

                let alive = this\_champ.take\_turn(

                    &mut self.p2\_champions,

                    &mut self.p1\_champions,

                    self.time\_unit,

                    self.movement\_amount,

                    &mut self.p2\_projectiles,

                );

                if alive {

                    self.p2\_champions.push\_back(this\_champ);

                } else {

                    self.dead\_champs.push\_back(this\_champ);

                }

            }

            info!("Simulating Projectiles");

            //simulate tick for each p1 projectile

            self.p1\_projectiles.retain\_mut(|f| {

                f.simulate\_tick(

                    &mut self.p2\_champions,

                    &mut self.p1\_champions,

                    &mut self.dead\_champs,

                )

            });

            //simulate tick for each p2 projectile

            self.p2\_projectiles.retain\_mut(|f| {

                f.simulate\_tick(

                    &mut self.p1\_champions,

                    &mut self.p2\_champions,

                    &mut self.dead\_champs,

                )

            });

            info!("End of Turn");

            //check for a winner

            if self.p1\_champions.is\_empty() {

                info!("Player 2 Wins");

                break; //break if there is a winner

            } else if self.p2\_champions.is\_empty() {

                info!("Player 1 Wins");

                break;

            }

        }

    }

    pub fn get\_winner(&self) -> i8 {

        if self.p1\_champions.is\_empty() {

            return 2

        }

        if self.p2\_champions.is\_empty() {

            return 1

        }

        return 0

    }

}

///implements display for board

impl std::fmt::Display for Board {

    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {

        write!(

            f,

            "p1: {:?},\np2: {:?}",

            self.p1\_champions, self.p2\_champions

        )

    }

}

**Champions.rs:**

// Import required types from other modules

use super::location::Location;

use super::projectiles::Projectile;

use super::shields::Shield;

use super::status\_effects::{StatusEffect, StatusType, Stun};

use super::utils::{find\_champion\_index\_from\_id, find\_champion\_index\_from\_id\_targetable, sign};

use super::item::Item;

use crate::prelude::\*;

// Import functionality from core

use core::fmt;

use rand::Rng;

use rand::seq::SliceRandom;

use std::collections::VecDeque;

use std::mem::take;

// Import surrealdb Value object and serde Serialize.

use surrealdb::sql::{Value, Object};

use serde::{Serialize, Deserialize};

///Champion struct<br />

///Holds basic information about the champion

#[derive(Debug, Serialize, Deserialize)]

pub struct Champion {

///index in champions array

pub id: u8,

///healthpoints

hp: f32,

///starting mana

sm: i16,

///ability mana cost

mc: i16,

///base armor value

ar: f32,

///Base Magic Resist Value

mr: f32,

///attack damage

ad: f32,

///attack speed (attacks per second)

attack\_speed: f32,

///attack range

ra: i8,

}

///Converts from a surrealdb Object to a champion

impl TryFrom<Object> for Champion {

type Error = Error;

fn try\_from(mut obj: Object) -> Result<Self> {

//fetch and convert values from the object

let ad : f32 = obj.remove("ad").unwrap().as\_float() as f32;

let ar : f32 = obj.remove("ar").unwrap().as\_float() as f32;

let attack\_speed : f32 = obj.remove("attack\_speed").unwrap().as\_float() as f32;

let hp : f32 = obj.remove("hp").unwrap().as\_float() as f32;

let id : u8 = Value::from(obj.remove("id").unwrap().record().unwrap().id).as\_int() as u8;

let mc : i16 = obj.remove("mc").unwrap().as\_int() as i16;

let mr : f32 = obj.remove("mr").unwrap().as\_float() as f32;

let ra : i8 = obj.remove("ra").unwrap().as\_int() as i8;

let sm : i16 = obj.remove("sm").unwrap().as\_int() as i16;

//return new champ

Ok(Champion { id, hp, sm, mc, ar, mr, ad, attack\_speed, ra})

}

}

/// Default values for champs

impl Default for Champion {

fn default() -> Self {

Champion { id: 0, hp: 0.0, sm: 0, mc: 0, ar: 0.0, mr: 0.0, ad: 0.0, attack\_speed: 0.0, ra: 0 }

}

}

impl Champion {

///Converts from a champion into String Value array for insertion in database

pub fn into\_values(&self) -> [(String, Value) ; 9] {

[

("id".into(), self.id.into()),

("hp".into(), self.hp.into()),

("sm".into(), self.sm.into()),

("mc".into(), self.mc.into()),

("ar".into(), self.ar.into()),

("mr".into(), self.mr.into()),

("ad".into(), self.ad.into()),

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

("ra".into(), self.ra.into())

]

}

}

/// Default champions array

pub const DEFAULT\_CHAMPIONS: [Champion; 4] = [

//Support

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,

},

//Bruiser

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,

},

//AD Ranged

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,

},

//AP Ranged

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,

},

];

///Enum for the 3 damage types Physical, Magical and True

#[derive(PartialEq, Eq, Clone, Copy, Debug, Serialize)]

pub enum DamageType {

Physical(),

Magical(),

#[allow(dead\_code)]

True(),

}

///PlacedChampion:<br />

///Stores information about a champion's location and status on a board (as well as ID of actual champion)

#[derive(Deserialize, Debug, Serialize)]

pub struct PlacedChampion {

///id of the associated champion

id: usize,

///star level of champion

star: usize,

///items

items: [u8; 3],

///location on board

location: Location,

///option for team

team : Option<u8>

}

///Converts from a surrealdb object from the database to a PlacedChampion

impl TryFrom<Object> for PlacedChampion {

type Error = Error;

fn try\_from(mut obj: Object) -> Result<Self> {

let id = obj.remove("of\_champ").unwrap().as\_int() as usize;

let item\_0 = obj.remove("item\_0").unwrap().as\_int() as u8;

let item\_1 = obj.remove("item\_1").unwrap().as\_int() as u8;

let item\_2 = obj.remove("item\_2").unwrap().as\_int() as u8;

let star\_level = obj.remove("star").unwrap().as\_int() as usize;

let location\_x = obj.remove("location\_x").unwrap().as\_int() as i8;

let location\_y = obj.remove("location\_y").unwrap().as\_int() as i8;

let team = obj.remove("team").unwrap().as\_int() as u8;

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

}

}

impl PlacedChampion {

///converts a placed champion into a String value array for insertion into the database

pub fn into\_values(&self) -> [(String, Value); 7] {

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

("star".into(), self.star.into()),

("item\_0".into(), self.items[0].into()),

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

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

("location\_x".into(), self.location.x.into()),

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

]

}

///generates a random placed champion

pub fn generate\_random\_champ(id\_range : usize, valid\_items : &Vec<u8>, team : bool) -> PlacedChampion {

//generates a random id and star level

let id = rand::thread\_rng().gen\_range(0..id\_range);

let star = rand::thread\_rng().gen\_range(0..3) as usize;

//initialise item array

let mut items : [u8 ; 3] = [0, 0, 0];

for i in 0..rand::thread\_rng().gen\_range(0..3) {

//choose item from valid\_items

items[i] = \*valid\_items.choose(&mut rand::thread\_rng()).unwrap();

}

//generate random location

let location = Location::generate\_random\_position\_team(team);

//returns placed champion

PlacedChampion { id, star, items, location, team: None }

}

pub fn new(id : usize, star : usize, items : [u8 ; 3], location : Location) -> PlacedChampion {

PlacedChampion { id, star, items, location, team : None}

}

}

///Struct for champion placed on board in a battle

#[derive(Debug, Clone, Serialize)]

pub struct SummonedChampion {

///location

pub location: Location,

///id of associated champion/ placed champion

of\_champ\_id : usize,

///progress of movement before new square, goes up to 10 then moves

movement\_progress: [i8; 2],

///health

health: f32,

///current mana

cm: i16,

///dodge chance in %

dc: u8,

///crit rate in %

cr: u8,

///crit damage

crit\_damage: f32,

///ability mana cost

mc: i16,

///armor

ar: f32,

///magic resist

mr: f32,

///attack damage

ad: f32,

///attacks per second/ attack speed

attack\_speed: f32,

///auto attack range

ra: i8,

///unique id

pub id: usize,

///cooldown before target chance

target\_cooldown: i8,

///cooldown before auto attacking again

auto\_attack\_delay: i16,

///attack speed modifier from items and effects

attack\_speed\_modifier: f32,

///id of target

target: usize,

///pathfinding target cell

target\_cells: Location,

///Stores all the item IDs the champion is holding.<br />

///\*\*Item IDS:\*\*<br />

///0 : Null<br />

///1 : B.F Sword (+10 Attack Damage)<br />

///2 : Needlessly Large Rod (+10 Ability Power)<br />

///3 : Giants Belt (+150 health)<br />

///4 : Chain Vest (+20 Armor)<br />

///5 : Negatron Cloak (+20 Magic Resist)<br />

///6 : Recurve Bow (+10% Attack Speed)<br />

///7 : \*Sparring Gloves\* (+5% Crit Chance, +10% Dodge Chance)<br />

///8 : Tear of the Goddess (+15 Mana)<br />

///9 : Spatula<br />

///11 : Deathblade (+40, +70, +100 Attack Damage - Star Level Dependent)<br />

///12 : \*Hextech Gunblade\* (Dealing Magic and True Damage heals the owner and lowest health ally for 25% of the damage)<br />

///13 : Zekes Herald (Grants 30% bonus attack speed to the holder and 2 adjacent allies in same row)<br />

///14 : Edge of Night (At 50% health - once per combat - the holder briefly becomes untargetable and sheds negative effects. Then they gain 30% attack speed)<br />

///15 : Bloodthirster (Damage dealt heals holder for 25%. Once per combat at 40% Health, gain a 25% maximum health shield for up to 5 seconds)<br />

///16 : Giant Slayer (Abilities and attacks deal 25% more damage, increased to 50% if the holder has over 2200 maximum health)<br />

///17 : Infinity Edge (+10 Attack Damage, +225% Crit Chance, +10% Crit Damage, Converts every 1% excess critical strike chance into 1% bonus critical strike damage)<br />

///18 : Spear of Shojin (Basic attacks restore an additional 8 mana on-attack)<br />

///19 : Shimmerscale Emblem (Wearer becomes a shimmerscale, cannot equip on a shimmersclae)<br />

///22 : Rabadons Deathcap (+975 Ability Power)<br />

///23 : Morellonomicon (+30 Ability Power, magic or true damage from an ability burns the holders target, dealing 25% of the targets maximum health as trude damage over 10 seconds and applying grevious wounds for the duration)<br />

///24 : Locket of the Iron Solari (At the start of combat, the wearer and all allies within 2 hexes in the same row gain a 300 / 350 / 400 health shield for 15 seconds - star level dependent)<br />

///25 : Ionic Spark (Enemies within 3 hexes have their magic resistance reduced by 50% (does not stack). When enemies within 3 hexes cast their ability, they are dealt 250% of their maximum mana as magic damage)<br />

///26 : Guinsoos Rageblade (Basic attacks grant 6% bonus attack speed for the rest of combat, stacks with no upper limit)<br />

///27 : \*Jeweled Gauntlet\* (+15% Crit Chance, +40% Crit Damage, +10 Ability Power, The holders magic adn true damage from abilities can critically strike)<br />

///28 : Archangels Staff (Grants the wearer 20 ability power every 5 seconds)<br />

///29 : Dragonmancer Emblem (Wearer becomes an dragonmancer, cannot equip on an dragonmancer)<br />

///33 : Warmogs Armor (+1000 Health)<br />

///34 : Sunfire Cape (+400 Health. At the start of combat and every 2 seconds thereafter, applies a 10% maximum health burn as true damage over 10 seconds and applying grevious wounds for the duration)<br />

///35 : Zephyr (At the start of combat, banishes for 5 seconds the unit that mirrors the wielders placement on the other side of the board. Pierces through CC immunity effects)<br />

///36 : ZZ Rot Portal (At the start of combat, the wearer taunts enemies within 4 hexes. When the wearer dies, a Voidspawn arises, taunting nearby enemies. Summoned units can spawn Voidspawns at 25% effectiveness)<br />

///37 : \*Banshees Claw\* (+15% Dodge Chance, +150 Health, At the beginning of each round, the holder and allies within 1 hex in the same row gain a shield that blocks the first enemy ability, up to 600 damage)<br />

///38 : Redemption (Every 5 seconds, the wearer radiates an aura to allies within 1 hex, healing them for 12% missing health. Affected allies take 25% reduced damage from AOE attacks for seconds)<br />

///39 : Guardian Emblem (Wearer becomes a guardian, cannot equip on a guardian)<br />

///44 : Bramble Vest (+60 Armor. Negatves 75% bonus damage from critical hits. On being hit by an attack, deal 75 / 100 / 150 magic damage to all nearby enemies (once every 2.5 seconds))<br />

///45 : Gargoyle Stoneplate (+18 Armor and Magic Resist for each enemy targeting the holder)<br />

///46 : \*Titans Resolve\* (Gain 2 attack damage and ability power when attacking or taking damage. After stacking 25 times, gain 25 armor and magic resist and stop stacking)<br />

///47 : \*Shroud of Stillness\* (Shoot a beam that delays the first cast of affected enemies by 35%)<br />

///48 : Frozen Heart (Reduce the attack speed of enemies within 2 hexes by 25%)<br />

///49 : Cavalier Emblem (Wearer becomes a cavalier, cannot equip on a cavalier)<br />

///55 : Dragons Claw (+120 Magic Resist, every 2 seconds, regenerate 1.2% maximum health for each enemy targeting the holder. If holder is a dragon, increase all bonuses and effects by 20%)<br />

///56 : \*Runaans Hurricane\* (+10 Atttack Damage, attacks fire a bolt at a nearby enemy, dealing 70% of the holders attack damage as physical damage)<br />

///57 : \*Quicksilver\* (+20% attack speed. Immune to crowd control for 15 secnds)<br />

///58 : Chalice of Power (+30 Ability Power to holder and 2 adjacent allies on same row)<br />

///59 : Mirage Emblem (Wearer becomes a mirage, cannot equip on a mirage)<br />

///66 : Rapid Firecannon (+50% attack speed and +1 attack range, attacks cannot miss)<br />

///67 : \*Last Whisper\* (Dealing physical damage reduces the targets armor by 50% for 5 seconds, does not stack)<br />

///68 : Statikk Shiv (+15% attack speed, every 3rd attack shocks enemies for 70 magic damage and reduces their magic resist by 50% for 5 seconds)<br />

///69 : Ragewing Emblem (Wearer becomes a ragewing, cannot equip on a ragewing)<br />

///77 : \*Thiefs Gloves\* (Each round equip 2 random items, improve with player level, you cannot equip other items)<br />

///78 : \*Hand of Justice\* (+15 attack damage, +15% ability power. Attacks and abilities heal for 15% of damage dealt. Each round randomly increase 1 effect by 30%)<br />

///79 : \*Assassin Emblem\* (Wearer becomes an assassin, cannot equip on an assassin)<br />

///88 : Blue Buff (+20 Starting Mana. Gain 20 mana after casting an ability)<br />

///89 : Mage Emblem (Wearer becomes a mage, cannot equip on a mage)<br />

///99 : Tacticians Crown (Increase board unit size by 1)<br />

items: [u8; 3],

///ability power

ap: f32,

///vec of status effects

se: Vec<StatusEffect>,

///generate mana delay (can't generate mana 1 secomnd after casting ability)

gain\_mana\_delay: i16,

///star level

star\_level: usize,

///incoming DMG modifier

incoming\_damage\_modifier: f32,

///starting HP

initial\_hp: f32,

///can be targeted or not

targetable: bool,

///needs to shed negative status effects

shed: u8,

///vec of all shields

shields: Vec<Shield>,

///whether zapped from ionic spark

zap: bool,

///whether zenith banished

banish: bool,

///titan's resolve stacks

titans\_resolve\_stacks: u8,

///omnivamp (% of healing from damage done)

omnivamp: f32,

///shiv attack count

shiv\_attack\_count: u8,

}

impl SummonedChampion {

///converts PlacedChampion into SummonChampion

pub fn new(placed\_champion: &PlacedChampion, id: usize) -> SummonedChampion {

info!(

"New Summoned Champion ID : {} Champion ID : {}",

id, placed\_champion.id

);

//create summoned champ with all details

SummonedChampion {

location: placed\_champion.location,

of\_champ\_id: placed\_champion.id,

movement\_progress: [0, 0],

initial\_hp: 0.0,

dc: 0,

cr: 25,

crit\_damage: 0.3,

id,

target\_cooldown: 0,

auto\_attack\_delay: 0,

attack\_speed\_modifier: 1.0,

target: 255,

target\_cells: Location { x: -1, y: -1 }, //(!O)

items: placed\_champion.items,

ap: 1.0,

se: Vec::new(),

gain\_mana\_delay: 0,

star\_level : placed\_champion.star,

incoming\_damage\_modifier: 1.0,

targetable: true,

shed: 0,

shields: Vec::new(),

//sortBy : 0,

//traits : traits,

zap: false, //discrepency maybe if order of status Effects is ever affected, alternative would be to iterate through status Effects and check for ionic spark

banish: false, //discrepency with this and many others if one status effect banishing ends and another is still going on etc.

titans\_resolve\_stacks: 0,

omnivamp: 0.0,

shiv\_attack\_count: 0,

..Default::default()

}

}

///setup champion

pub fn setup(

&mut self,

friendly\_champions: &mut VecDeque<SummonedChampion>,

enemy\_champions: &mut VecDeque<SummonedChampion>,

champions : &[Champion],

items : &[Item]

) {

info!("Setup of Champion {}", self.id);

{

//sets stats to initial values specified by Champion.

let of\_champion = &champions[self.of\_champ\_id];

self.health = of\_champion.hp;

self.cm = of\_champion.sm;

self.ar = of\_champion.ar;

self.mr = of\_champion.mr;

self.ad = of\_champion.ad;

self.attack\_speed = of\_champion.attack\_speed;

self.ra = of\_champion.ra \* 2;

self.mc = of\_champion.mc;

}

{

//for each item, give item effect

for item in self.items {

info!("Giving item effect {}", item);

self.give\_item\_effect(item, friendly\_champions, enemy\_champions, items)

}

}

//set initial hp to current health, accounting for items etc

self.initial\_hp = self.health;

info!("Set HP to {}", self.health);

}

///heal

fn heal(&mut self, mut healing\_amount: f32) {

info!("{self} - Healing");

//check for grevious wounds

if self.se.contains(&StatusEffect {

status\_type: StatusType::GreviousWounds(),

..Default::default()

}) {

info!(

"Has Grevious Wounds cutting halving healing before: {}",

healing\_amount

);

healing\_amount /= 2.0; //cut healing in half

info!("After {}", healing\_amount);

}

self.health = self.initial\_hp.min(self.health + healing\_amount); //make sure health doesn't go higher than initial amount

info!("{self}");

}

///takes the turn for a summonedchamp, returns whether the champion is alive at end of turn

pub fn take\_turn(

&mut self,

friendly\_champions: &mut VecDeque<SummonedChampion>,

enemy\_champions: &mut VecDeque<SummonedChampion>,

time\_unit: i8,

movement\_amount: i8,

projectiles: &mut Vec<Projectile>,

) -> bool {

info!("Taking turn for {self}");

if self.health <= 0.0 {

info!("Health below zero, removing self");

return false;

//champ is dead

}

self.target\_cooldown = self.target\_cooldown.checked\_sub(time\_unit).unwrap\_or(-1); //Reduce cooldown to check target/ find new target

self.auto\_attack\_delay = self

.auto\_attack\_delay

.checked\_sub(time\_unit as i16)

.unwrap\_or(-1); //Risks going out of bounds as auto attack value may not be called for some time, so checked subtraction

self.gain\_mana\_delay = self

.gain\_mana\_delay

.checked\_sub(time\_unit as i16)

.unwrap\_or(-1);

{

info!("Simulating status effects");

let mut status\_effects = take(&mut self.se); //takes status effect vec

let mut stun = Stun { stun: 0 }; // setting stun to 0

status\_effects.retain\_mut(|x| {

self.perform\_status(x, friendly\_champions, enemy\_champions, time\_unit, &mut stun)

}); //perform status for each status effect

if self.health <= 0.0 {

info!("Health below zero from status effect, removing");

//died from status effect

return false;

}

self.se.extend(status\_effects); //extend se by status effects

//do NOT set self.se = status\_effects, as new status may have been added

self.update\_shed(); //updates shed

self.shields.retain\_mut(|x| x.update\_shield(time\_unit)); //updates all shields

if stun.stun == 1 {

info!("Is stunned");

//stunned

return true;

}

}

if self.banish {

//banished

info!("Is banished");

return true;

}

{

info!("Calculating auto attack or movement");

let mut need\_new\_target\_cell: bool = false; //bool to store whether new path is needed

let mut target\_object: Option<SummonedChampion> = None;

if self.target\_cooldown >= 0 {

info!("Cooldown above zero, trying to find target {}", self.target);

//if already has target and doesnt want to change targets

if let Some(index) =

find\_champion\_index\_from\_id\_targetable(enemy\_champions, self.target) //try to find target

{

target\_object = enemy\_champions.swap\_remove\_back(index);

//target found

info!("Target found? : {}", target\_object.is\_some());

}

}

if target\_object.is\_none() {

//target\_object not found

info!("Could not find target or need new target");

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

need\_new\_target\_cell = true; //tells us to recalculate pathfinding later

let mut index: Option<usize> = None;

if let Some((i, champ)) = self

.location

.get\_closest\_to\_location\_targetable\_index(enemy\_champions) {

index = Some(i);//get closest to location that is targetable

}

if index.is\_none() {

//no targetable champions, ending turn

info!("No targetable champions, ending turn");

return true;

}

target\_object = enemy\_champions.swap\_remove\_back(index.unwrap());

}

let mut target\_object: SummonedChampion = target\_object.unwrap();

info!("Target is {target\_object}");

self.target = target\_object.id; //set target

let distance\_to\_target = self

.location

.distance\_between\_points(&target\_object.location); //get distance to target

info!("Distance to target {distance\_to\_target}");

if distance\_to\_target <= self.ra {

//if target in range

info!("Target in range, attacking or reducing auto attack cooldown");

info!("Auto Attack Delay Remaining {0}", self.auto\_attack\_delay);

if self.auto\_attack\_delay <= 0

//if autoattack ready

{

info!("Ready to attack");

/\*

self.aS = attacks per 1 second

self.autoAttackDelay = time in 1/10 of second until next attack

self.attackSpeedIncrease = percentage increase in attack speed

autoAttackDelay (seconds) = 1 / (attackSpeed \* attackSpeedMod)

autoAttackDelay (centiseconds) = 100 / (attackSpeed \* attackSpeedMod)

\*/

info!(

"as: {}, mod: {}",

self.attack\_speed, self.attack\_speed\_modifier

);

self.auto\_attack\_delay =

((100.0 / (self.attack\_speed \* self.attack\_speed\_modifier)) as i16).max(20); //calculating auto attack delay

info!("Auto attack delay set to {}", self.auto\_attack\_delay);

if self.items.contains(&26) {

self.attack\_speed\_modifier \*= 1.06;

info!("Increasing speed with Rageblade")

}

if self.gain\_mana\_delay <= 0 {

//if can gain mana

self.cm += 10;

info!("Gaining mana");

if self.items.contains(&18) {

self.cm += 8;

info!("Additional mana from shojin");

}

info!("Current mana {}", self.cm);

}

self.shiv\_attack\_count += 1; //increase shiv count

if self.items.contains(&68) && self.shiv\_attack\_count == 3 {

info!("Has shiv and on third auto, applying affect");

//shiv effect

self.deal\_damage(

friendly\_champions,

&mut target\_object,

50.0,

DamageType::Magical(),

false,

);

target\_object.se.push(StatusEffect {

duration: Some(500),

status\_type: StatusType::ShredMagicResist(2.0),

is\_negative: true,

..Default::default()

});

//for other champs affected by shiv effect

for (i, enemy\_champ) in enemy\_champions.iter\_mut().enumerate() {

self.deal\_damage(

friendly\_champions,

enemy\_champ,

50.0,

DamageType::Magical(),

false,

);

enemy\_champ.se.push(StatusEffect {

duration: Some(500),

status\_type: StatusType::ShredMagicResist(2.0),

is\_negative: true,

..Default::default()

});

if i > 2 {

break;

}

}

}

self.shiv\_attack\_count %= 3; //make sure shiv count sticks between 0 to 3

if self.items.contains(&56) {

//doing runaan's second auto attack

info!("Has runaan's, performing second auto");

warn!("Runaan's can be dodged, treated like normal auto");

let closest\_other\_enemy = self

.location

.get\_closest\_to\_location\_targetable(enemy\_champions); //fetch closest other champ

if let Some(target) = closest\_other\_enemy {

self.deal\_damage(

friendly\_champions,

target,

self.ad \* 0.7,

DamageType::Physical(),

false,

) //do damage

}

}

info!("Auto attacking");

if target\_object.dc == 0

|| target\_object.dc < rand::thread\_rng().gen\_range(0..100)

|| self.items.contains(&66)

//calculating whether to dodge

{

//no dodge

info!("Not dodged");

self.deal\_damage( //deal auto damage

friendly\_champions,

&mut target\_object,

self.ad,

DamageType::Physical(),

false,

);

info!("Enemy Champion Health is {0}", target\_object.health);

} else {

info!("Dodged Attack");

}

}

//return target

enemy\_champions.push\_back(target\_object);

} else {

info!("Not in Range");

if need\_new\_target\_cell || self.location == self.target\_cells {

//checks if need new pathfinding target

info!("Need Target Cell");

self.target\_cells = self.location; //setting target cells to location so if it does not find a target this frame will try to do it again

let mut lowest\_distance: i8 = i8::MAX; //setting lowestDistance to high value

let mut new\_position;

for possible\_move in [[0, -1], [1, -1], [1, 0], [-1, 0], [-1, 1], [0, 1]]

//for every possible move

{

//calculate distance

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

let distance\_to\_target = target\_object

.location

.distance\_between\_points(&new\_position);

if distance\_to\_target < lowest\_distance {

if (!new\_position.check\_valid())

|| friendly\_champions

.iter()

.any(|f| f.location == new\_position)

{

continue;

}

//if distance lower, position valid and there is no other friendly champion in cell

info!("Found target cell {}", new\_position);

lowest\_distance = distance\_to\_target;//set new target cell

self.target\_cells = new\_position;

}

}

}

info!("Moving to Target Cell {}", self.target\_cells);

self.movement\_progress[0] +=

movement\_amount \* sign(self.target\_cells.x - self.location.x); //add movement progress

info!(

"Position ({0:?}) -- Movement Progress ({1:?})",

self.location, self.movement\_progress

);

if self.movement\_progress[0].abs() >= 10 {

//move self

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

self.movement\_progress[0] = 0;

}

self.movement\_progress[1] +=

movement\_amount \* sign(self.target\_cells.y - self.location.y);

if self.movement\_progress[1].abs() >= 10 {

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

self.movement\_progress[1] = 0;

}

enemy\_champions.push\_back(target\_object); //return target

}

}

if self.items.contains(&25) {

//giving ionic spark effect

info!("giving ionic spark effect");

for champ in enemy\_champions

.iter\_mut()

.filter(self.location.get\_within\_distance(7))

{

//give effect to all enemy champs within range

info!("Push to {champ}");

champ.se.push(StatusEffect {

duration: Some((time\_unit + 1).into()),

status\_type: StatusType::IonicSparkEffect(),

is\_negative: true,

..Default::default()

})

}

}

if self.cm >= self.mc {

//cast ability

info!("Enough mana casting ability");

if self.zap {

//zap from ionic spakr

info!("Zap");

self.health -= (self.mc as f32) \* 2.5;

}

self.cm = 0;

if self.items.contains(&88) {

info!("Bluebuff");

self.cm = 20;

}

self.gain\_mana\_delay = 100;

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

}

true

}

///deals damage to a target, requires a source Summoned Champion

pub fn deal\_damage(

&mut self,

friendly\_champions: &mut VecDeque<SummonedChampion>,

target: &mut SummonedChampion,

damage\_amount: f32,

damage\_type: DamageType,

\_is\_splash: bool,

) {

info!("Dealing {damage\_amount} from {self} to {target}");

//calculate base damage amount

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

info!("Increased to {damage}");

let can\_crit; //initialise can\_crit bool

let mut crit\_damage = self.crit\_damage; //initialise crit damage variable

match damage\_type {

DamageType::Physical() => {

//can crit true, decrease damage by armor

can\_crit = true;

damage /= 1.0 + target.ar;

if self.items.contains(&67) {

//apply armor shred from last whisper

if !target.se.contains(&StatusEffect {

status\_type: StatusType::LastWhisperShred(),

..Default::default()

}) {

//give last whisper shred if target does not have it

target.se.push(StatusEffect {

duration: Some(500),

status\_type: StatusType::LastWhisperShred(),

is\_negative: true,

..Default::default()

})

}

}

if self.cr > 100 && self.items.contains(&17) {

//give extra crit damage from infinity edge

crit\_damage += (self.cr - 100) as f32

}

}

DamageType::Magical() => {

can\_crit = self.items.contains(&27);

damage /= 1.0 + target.mr;//decrease damage by mr

//can crit depends on jeweled gauntlet

}

DamageType::True() => {

can\_crit = self.items.contains(&27); //can crit depends on jeweled

}

}

if can\_crit && self.cr > rand::thread\_rng().gen\_range(0..100) { //check for crit

info!("Crit");

let mut additional\_crit\_damage = damage \* crit\_damage;

if target.items.contains(&44) {

//reduce dmg if target has bramble vest

additional\_crit\_damage /= 4.0;

}

damage += additional\_crit\_damage //increase dmg by crit dmg

}

if self.items.contains(&16) {

//give bonus giant's slayer attack dmg

if target.initial\_hp >= 2200.0 {

damage \*= 1.45

} else {

damage \*= 1.2

}

}

if damage\_type != DamageType::Physical() {

//give gunblade and morellos

if self.items.contains(&12) {

//give gunblade healing

let healing = damage / 4.0; //calculate healing

self.heal(healing); //heal self

if let Some(lowest\_hp\_champ) =

friendly\_champions

.iter\_mut()

.reduce(|x, y| if x.health < y.health { x } else { y })

{

//get lowest HP ally

//if there are any allies

lowest\_hp\_champ.heal(healing)

}

}

if self.items.contains(&23) {

//if self has morellos give morellos effect

target.se.push(StatusEffect {

duration: Some(1000),

status\_type: StatusType::GreviousWounds(),

is\_negative: true,

..Default::default()

});

let damage\_to\_do = target.initial\_hp / 4.0;

target.se.push(StatusEffect {

duration: Some(100),

status\_type: StatusType::MorellonomiconBurn(

damage\_to\_do / 10.0,

damage\_to\_do,

100,

),

is\_negative: true,

..Default::default()

})

}

}

self.heal(damage \* self.omnivamp); //give omnivamp healing

info!("Accounting for shield");

for shield in &mut target.shields {

//reduce damage due to shields

info!("shield : {}", shield.size);

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

info!("damage : {}", damage);

if damage <= 0.0 {

break;

//no damage left, stop checking shields

}

}

self.titans\_resolve\_stacks = 25.min(self.titans\_resolve\_stacks + 1); //add titan's resolve stacks

target.titans\_resolve\_stacks = 25.min(target.titans\_resolve\_stacks + 1); //give enemy titan's resolve stacks

target.health -= damage; //deal damage

if target.gain\_mana\_delay <= 0 {

// give mana is delay off

target.cm = target

.cm

.checked\_add((0.7 \* damage) as i16)

.unwrap\_or(target.mc);

//give bonus mana

}

}

///cast ability

fn cast\_ability(

&mut self,

friendly\_champions: &mut VecDeque<SummonedChampion>,

enemy\_champions: &mut VecDeque<SummonedChampion>,

projectiles: &mut Vec<Projectile>,

) {

info!("casting ability");

//match id to its selected ability

match self.of\_champ\_id {

0 => { //support ability

let mut player\_distances: Vec<(i8, &mut SummonedChampion, bool)> =

friendly\_champions

.iter\_mut()

.map(|x| (self.location.distance\_between\_points(&x.location), x, true))

.collect(); //create a vector of distances from the SummonedChamp, SummonedChamp and bool representing its team (true = same team, false = other team)

player\_distances.extend(

enemy\_champions

.iter\_mut()

.map(|x| (self.location.distance\_between\_points(&x.location), x, false)),

); //extend with enemy champions

let star\_level = self.star\_level; //gets current star level

player\_distances.sort\_unstable\_by\_key(|a| a.0); //sorts the player distances so the smallest distances are first

let number\_affected: usize = [3, 4, 5][star\_level]; //how many champions it can hit/ effect

let mut i = 0; //count number already affected

let ap = self.ap; //get ability power

for (\_, champ, on\_team) in player\_distances //iterate through play distances

{

if i >= number\_affected { //already affected all it can

break;

}

if on\_team

//if friendly champ

{

//give allies attack speed for 5 seconds

champ.se.push(StatusEffect {

duration: Some(500),

status\_type: StatusType::AttackSpeedBuff(1.7 \* ap),

..Default::default()

});

} else {

//enemy champ

//stun enemies for 1.5 seconds and increase damage for 20%

champ.se.push(StatusEffect {

duration: Some(150),

status\_type: StatusType::Stun(),

is\_negative: true,

..Default::default()

});

champ.se.push(StatusEffect {

duration: Some(150),

status\_type: StatusType::IncreaseDamageTaken(1.2 \* ap),

is\_negative: true,

..Default::default()

});

}

i += 1; //add 1 to count of hit enemies

}

if i < number\_affected

//give self effect if there aren't enough champs to hit

{

self.se.push(StatusEffect {

duration: Some(500),

status\_type: StatusType::AttackSpeedBuff(1.7 \* ap),

..Default::default()

});

}

}

1 => { //bruiser ability

let star\_level = self.star\_level; //get star level

let target\_index =

find\_champion\_index\_from\_id(enemy\_champions, self.target).unwrap\_or(0); //get target object or the first in vec

self.heal((300.0 + 50.0 \* star\_level as f32) \* self.ap); //heals

//deals damage

self.deal\_damage(

friendly\_champions,

&mut enemy\_champions[target\_index],

(25.0 \* star\_level as f32) \* 4.0 \* self.ad,

DamageType::Physical(),

false,

)

}

2 => {

let target = find\_champion\_index\_from\_id(enemy\_champions, self.target).unwrap\_or(0); //get target

let target\_location = enemy\_champions[target].location;

let damage: f32 = self.ad \* 3.0 \* (self.star\_level as f32); //calculate damage

projectiles.push(Projectile::new(

self.location,

Option::Some(target\_location),

self.target,

damage,

DamageType::Physical(),

0.0,

5,

self.id,

)) //create new projectile

}

3 => {

//fetches target index

let target = find\_champion\_index\_from\_id(enemy\_champions, self.target).unwrap\_or(0); //(!D) Can strike from out of range

//gets their location

let target\_location = enemy\_champions[target].location;

//calculates damage

let damage: f32 = 250.0 \* self.ap \* (self.star\_level as f32);

//adds projectile to vec

projectiles.push(Projectile::new(

self.location,

Option::Some(target\_location),

self.target,

damage,

DamageType::Magical(),

damage / 3.0,

3,

self.id,

))

}

\_ => println!("Unimplemented {}", self.id),

}

}

/// get number of enemy champions targetting

pub fn get\_num\_targeting(&self, enemy\_champions: &VecDeque<SummonedChampion>) -> usize {

enemy\_champions

.iter()

.filter(|p| p.target == self.id)

.count() //counts the number of enemy champions with target equal to id

}

/// retrieve whether the champion is targetable

pub fn get\_is\_targetable(&self) -> bool {

self.targetable && !self.banish

}

///Gives an item effect to a champion<br />

///\*\*Item IDS:\*\*<br />

///0 : Null<br />1 : B.F Sword (+10 Attack Damage)<br />2 : Needlessly Large Rod (+10 Ability Power)<br />3 : Giants Belt (+150 health)<br />4 : Chain Vest (+20 Armor)<br />5 : Negatron Cloak (+20 Magic Resist)<br />6 : Recurve Bow (+10% Attack Speed)<br />7 : \*Sparring Gloves\* (+5% Crit Chance, +10% Dodge Chance)<br />8 : Tear of the Goddess (+15 Mana)<br />9 : Spatula<br />11 : Deathblade (+40, +70, +100 Attack Damage - Star Level Dependent)<br /> 12 : \*Hextech Gunblade\* (Dealing Magic and True Damage heals the owner and lowest health ally for 25% of the damage)<br />13 : Zekes Herald (Grants 30% bonus attack speed to the holder and 2 adjacent allies in same row)<br />14 : Edge of Night (At 50% health - once per combat - the holder briefly becomes untargetable and sheds negative effects. Then they gain 30% attack speed)<br />15 : Bloodthirster (Damage dealt heals holder for 25%. Once per combat at 40% Health, gain a 25% maximum health shield for up to 5 seconds)<br />16 : Giant Slayer (Abilities and attacks deal 25% more damage, increased to 50% if the holder has over 2200 maximum health)<br />17 : Infinity Edge (+10 Attack Damage, +75% Crit Chance, +10% Crit Damage, Converts every 1% excess critical strike chance into 1% bonus critical strike damage)<br />18 : Spear of Shojin (✓) (Basic attacks restore an additional 8 mana on-attack)<br />19 : Shimmerscale Emblem (Wearer becomes a shimmerscale, cannot equip on a shimmersclae)<br />22 : Rabadons Deathcap (+75 Ability Power)<br />23 : Morellonomicon (+30 Ability Power, magic or true damage from an ability burns the holders target, dealing 25% of the targets maximum health as trude damage over 10 seconds and applying grevious wounds for the duration)<br />24 : Locket of the Iron Solari (At the start of combat, the wearer and all allies within 2 hexes in the same row gain a 300 / 350 / 400 health shield for 15 seconds - star level dependent)<br />25 : Ionic Spark (Enemies within 3 hexes have their magic resistance reduced by 50% (does not stack). When enemies within 3 hexes cast their ability, they are dealt 250% of their maximum mana as magic damage)<br />26 : Guinsoos Rageblade (Basic attacks grant 6% bonus attack speed for the rest of combat, stacks with no upper limit)<br />27 : \*Jeweled Gauntlet\* (+15% Crit Chance, +40% Crit Damage, +10 Ability Power, The holders magic adn true damage from abilities can critically strike)<br />28 : Archangels Staff (Grants the wearer 20 ability power every 5 seconds)<br />29 : Dragonmancer Emblem (Wearer becomes an dragonmancer, cannot equip on an dragonmancer)<br />33 : Warmogs Armor (+1000 Health)<br />34 : Sunfire Cape (+400 Health. At the start of combat and every 2 seconds thereafter, applies a 10% maximum health burn as true damage over 10 seconds and applying grevious wounds for the duration)<br />35 : Zephyr (At the start of combat, banishes for 5 seconds the unit that mirrors the wielders placement on the other side of the board. Pierces through CC immunity effects)<br />36 : ZZ Rot Portal (At the start of combat, the wearer taunts enemies within 4 hexes. When the wearer dies, a Voidspawn arises, taunting nearby enemies. Summoned units can spawn Voidspawns at 25% effectiveness)<br />37 : \*Banshees Claw\* (+15% Dodge Chance, +150 Health, At the beginning of each round, the holder and allies within 1 hex in the same row gain a shield that blocks the first enemy ability, up to 600 damage)<br />38 : Redemption (Every 5 seconds, the wearer radiates an aura to allies within 1 hex, healing them for 12% missing health. Affected allies take 25% reduced damage from AOE attacks for seconds)<br />39 : Guardian Emblem (Wearer becomes a guardian, cannot equip on a guardian)<br />44 : Bramble Vest (+60 Armor. Negatves 75% bonus damage from critical hits. On being hit by an attack, deal 75 / 100 / 150 magic damage to all nearby enemies (once every 2.5 seconds))<br />45 : Gargoyle Stoneplate (+18 Armor and Magic Resist for each enemy targeting the holder)<br />46 : \*Titans Resolve\* (Gain 2 attack damage and ability power when attacking or taking damage. After stacking 25 times, gain 25 armor and magic resist and stop stacking)<br />47 : \*Shroud of Stillness\* (Shoot a beam that delays the first cast of affected enemies by 35%)<br />48 : Frozen Heart (Reduce the attack speed of enemies within 2 hexes by 25%)<br />49 : Cavalier Emblem (Wearer becomes a cavalier, cannot equip on a cavalier)<br />55 : Dragons Claw (+120 Magic Resist, every 2 seconds, regenerate 1.2% maximum health for each enemy targeting the holder. If holder is a dragon, increase all bonuses and effects by 20%)<br />56 : \*Runaans Hurricane\* (+10 Atttack Damage, attacks fire a bolt at a nearby enemy, dealing 70% of the holders attack damage as physical damage)<br />57 : \*Quicksilver\* (+20% attack speed. Immune to crowd control for 15 secnds)<br />58 : Chalice of Power (+30 Ability Power to holder and 2 adjacent allies on same row)<br />59 : Mirage Emblem (Wearer becomes a mirage, cannot equip on a mirage)<br />66 : Rapid Firecannon (+50% attack speed and +1 attack range, attacks cannot miss)<br />67 : \*Last Whisper\* (Dealing physical damage reduces the targets armor by 50% for 5 seconds, does not stack)<br />68 : Statikk Shiv (+15% attack speed, every 3rd attack shocks enemies for 70 magic damage and reduces their magic resist by 50% for 5 seconds)<br />69 : Ragewing Emblem (Wearer becomes a ragewing, cannot equip on a ragewing)<br />77 : \*Thiefs Gloves\* (Each round equip 2 random items, improve with player level, you cannot equip other items)<br />78 : \*Hand of Justice\* (+15 attack damage, +15% ability power. Attacks and abilities heal for 15% of damage dealt. Each round randomly increase 1 effect by 30%)<br />79 : \*Assassin Emblem\* (Wearer becomes an assassin, cannot equip on an assassin)<br />88 : Blue Buff (+20 Starting Mana. Gain 20 mana after casting an ability)<br />89 : Mage Emblem (Wearer becomes a mage, cannot equip on a mage)<br />99 : Tacticians Crown (Increase board unit size by 1)<br />

fn give\_item\_effect(

&mut self,

item: u8,

friendly\_champions: &mut VecDeque<SummonedChampion>,

enemy\_champions: &mut VecDeque<SummonedChampion>,

items : &[Item],

) {

info!("giving item {}", item);

if item == 0 { return } //no actual item

let mut item\_obj = None;

for item\_n in items {

if item\_n.id == item {

item\_obj = Some(item\_n);

}

}

if item\_obj.is\_none() { return } //can't find item

let item\_obj = item\_obj.unwrap();

{

self.health += item\_obj.health;

self.ad += item\_obj.ad;

self.ap += item\_obj.ap;

self.ar += item\_obj.ar;

self.mr += item\_obj.mr;

self.attack\_speed\_modifier \*= item\_obj.attack\_speed\_modifier;

self.ra += item\_obj.ra;

self.cr += item\_obj.cr;

self.dc += item\_obj.dc;

self.cm += item\_obj.cm;

self.omnivamp += item\_obj.omnivamp;

self.crit\_damage += item\_obj.crit\_damage;

//give plain stat buffs

}

//give item specific effects

match item {

11 => {self.ad += item\_obj.ad \* (self.star\_level as f32)}, //increasing ad with star level

13 => {

for friendly\_champion in friendly\_champions

.iter\_mut()

.filter(self.location.get\_within\_distance(3))

{

if friendly\_champion.location.y == self.location.y {

friendly\_champion.attack\_speed\_modifier \*= 1.3;

}

} //attack speed buff to all allied champs adjacent on same row

}

14 => {

self.se.push(StatusEffect {

duration: Some(0),

status\_type: StatusType::EdgeOfNight(),

..Default::default()

})

} //gives edge of night buff

15 => {

self.se.push(StatusEffect {

duration: Some(0),

status\_type: StatusType::Bloodthirster(),

..Default::default()

}); //gives bloodthirster buff

}

24 => {

let shield\_amount = [300.0, 350.0, 400.0][self.star\_level];

self.shields.push(Shield {

duration: 1500,

size: shield\_amount,

..Default::default()

});

for friendly\_champion in friendly\_champions

.iter\_mut()

.filter(self.location.get\_within\_distance(3))

{

if friendly\_champion.location.y == self.location.y {

friendly\_champion.shields.push(Shield {

duration: 1500,

size: shield\_amount,

..Default::default()

});

}

}//gives shield to self and all adjacent allies on same row

}

28 => {

self.se.push(StatusEffect {

duration: Some(500),

status\_type: StatusType::ArchangelStaff(0.2),

..Default::default()

})//gives archangel buf

}

34 => {

self.se.push(StatusEffect {

duration: Some(0),

status\_type: StatusType::GiveSunfire(),

..Default::default()

}) //gives sunfire buff

} //(!U)

35 => {

self.se.push(StatusEffect {

duration: Some(0),

status\_type: StatusType::Zephyr(500),

..Default::default()

})

} //gives zephyr effect

36 => {

for enemy\_champion in enemy\_champions

.iter\_mut()

.filter(self.location.get\_within\_distance(9))

{

enemy\_champion.se.push(StatusEffect {

duration: Some(0),

status\_type: StatusType::Taunted(self.id),

is\_negative: true,

..Default::default()

}) //taunts all enemies within range

}

}

37 => {

self.shields.push(Shield {

duration: 1500,

size: 600.0,

blocks\_type: Some(DamageType::Magical()),

pop: true,

}); //gives self banshee shield

for friendly\_champion in friendly\_champions

.iter\_mut()

.filter(self.location.get\_within\_distance(3))

{

if friendly\_champion.location.y == self.location.y

//gives banshee's shield

{

friendly\_champion.shields.push(Shield {

duration: 1500,

size: 600.0,

blocks\_type: Some(DamageType::Magical()),

pop: true,

});

}

}

}

38 => {

self.se.push(StatusEffect {

duration: Some(100),

status\_type: StatusType::RedemptionGive(),

..Default::default()

})

} //Gives redemption effect

45 => {

self.se.push(StatusEffect {

duration: Some(0),

status\_type: StatusType::Gargoyles(0.0),

..Default::default()

}) //gives gargoyles effect

}

46 => {

self.se.push(StatusEffect {

duration: Some(0),

status\_type: StatusType::TitansResolve(0),

..Default::default()

}) //gives titans resolve effect

}

47 => {

self.se.push(StatusEffect {

duration: Some(0),

status\_type: StatusType::ShroudOfStillness(),

..Default::default()

}) //gives shroud of stillness effect

}

48 => {

self.se.push(StatusEffect {

duration: Some(0),

status\_type: StatusType::ProtectorsVow(),

..Default::default()

}) //gives protectors vow effect

}

55 => {

self.se.push(StatusEffect {

duration: Some(200),

status\_type: StatusType::DragonClawHeal(),

..Default::default()

}) //gives dragon claw heal

}

57 => {

self.se.push(StatusEffect {

duration: Some(15000),

status\_type: StatusType::CrowdControlImmune(),

..Default::default()

}); //gives cc immunity for 15 seconds

}

58 => {

for friendly\_champion in friendly\_champions

.iter\_mut()

.filter(self.location.get\_within\_distance(3))

{

if friendly\_champion.location.y == self.location.y

{

friendly\_champion.ap += 0.3;

}

} //gives ap to friendly champs adjacent in same row

}

78 => {

//50% chance to give bonus ad ap, 50% chance to give bonus omnivamp

if rand::thread\_rng().gen\_range(0..100) > 50

{

self.ad += 30.0;

self.ap += 0.3;

self.omnivamp += 0.15;

} else {

self.ad += 15.0;

self.ap += 0.15;

self.omnivamp += 0.3;

}

}

\_ => (),

}

}

///checks if id equal

pub fn equal\_id(&self, id: usize) -> bool {

self.id == id //checks if id equal

}

///checks if shed == 2

pub fn is\_shed(&self) -> bool {

self.shed == 2 //checks if shed

}

///updates shed

fn update\_shed(&mut self) {

//if shed is 1, set to 2, else set to 0

if self.shed == 1 {

self.shed = 2;

} else {

self.shed = 0;

}

}

///performs a status effect, returning true if status effect should be kept or false if removed

fn perform\_status(

&mut self,

status\_effect: &mut StatusEffect,

friendly\_champions: &mut VecDeque<SummonedChampion>,

enemy\_champions: &mut VecDeque<SummonedChampion>,

time\_unit: i8,

stun: &mut Stun,

) -> bool {

info!("Performing status");

//if status has duration

if status\_effect.duration.is\_some() {

let mut n\_duration = status\_effect

.duration

.unwrap()

.checked\_sub(time\_unit.into())

.unwrap\_or(0); //unwrap duration and do checked subtraction

if self.is\_shed() && status\_effect.is\_negative {

n\_duration = 0;

} //if shed and negative set duration to 0

if n\_duration <= 0 {

match status\_effect.status\_type {

//undo status effect/ remove effect. some effects aren't actually removed but just reinitialise

StatusType::AttackSpeedBuff(modifier) => self.attack\_speed\_modifier /= modifier,

StatusType::IncreaseDamageTaken(modifier) => {

self.incoming\_damage\_modifier /= modifier

}

StatusType::Untargetable() => {

self.targetable = true //(!D) if have 2 untargetable effects this will untarget too early

}

StatusType::MorellonomiconBurn(dmg\_per\_tick, dmg\_to\_do, time\_next\_tick) => {

if self.is\_shed() {

return false; //remove if shed

}

if dmg\_per\_tick > dmg\_to\_do {

self.health -= dmg\_to\_do; //deal remainiong damage

} else {

n\_duration = time\_next\_tick; //set duration to next tick time

self.health -= dmg\_per\_tick;

status\_effect.status\_type = StatusType::MorellonomiconBurn(

dmg\_per\_tick,

dmg\_to\_do - dmg\_per\_tick,

time\_next\_tick,

); //update status

}

}

StatusType::IonicSparkEffect() => {

self.mr \*= 2.0; //(!D) Possible discrepency

self.zap = false

}

StatusType::ArchangelStaff(ap\_amount) => {

n\_duration = 500;

self.ap += ap\_amount;

}

StatusType::Banished() => self.banish = false,

StatusType::RedemptionGive() => {

n\_duration = 100; //increase duration

for champ in friendly\_champions

.iter\_mut()

.filter(self.location.get\_within\_distance(3))

{

champ.heal((champ.initial\_hp - champ.health) \* 0.12)

//discrepency check at multitarget damage time for redemption heal for reduction

}

self.heal((self.initial\_hp - self.health) \* 0.12); //gives healing to self and all adjacent

}

StatusType::Gargoyles(old\_num\_targeting) => {

n\_duration = 100; //increase duration

let num\_targeting: f32 = self.get\_num\_targeting(enemy\_champions) as f32;

let difference = num\_targeting - old\_num\_targeting; //get change

self.ar += 0.18 \* difference;

self.mr += 0.18 \* difference;

status\_effect.status\_type = StatusType::Gargoyles(num\_targeting);

}

StatusType::ShroudOfStillness() => {

for champ in enemy\_champions

.iter\_mut()

.filter(|x| x.location.x == self.location.x)

{

champ.cm -= (7 \* champ.mc) / 20;

} //mana debuff to all in row

}

StatusType::DragonClawHeal() => {

n\_duration = 200; //reset status effect

let num\_targeting: f32 = self.get\_num\_targeting(enemy\_champions) as f32;

self.heal(self.initial\_hp \* 0.012 \* num\_targeting);

}

StatusType::LastWhisperShred() => {

self.ar \*= 2.0

}

StatusType::GiveSunfire() => {

//(!U)

n\_duration = 300;

for champ in enemy\_champions

.iter\_mut()

.filter(self.location.get\_within\_distance(3))

{

let dmg = champ.initial\_hp / 20.0;

champ.se.push(StatusEffect {

duration: Some(100),

status\_type: StatusType::MorellonomiconBurn(dmg, dmg / 3.0, 100),

..Default::default()

})

}

}

StatusType::EdgeOfNight() => {

//if health below half

if self.health <= (self.initial\_hp / 2.0) {

self.se.push(StatusEffect {

duration: Some(50),

status\_type: StatusType::Untargetable(),

..Default::default()

}); //optimisation at every ..Default::default() with instead isNegative : false

self.se.push(StatusEffect {

duration: None,

status\_type: StatusType::AttackSpeedBuff(1.3),

..Default::default()

}); //(!D) technically attack speed buff comes into play after untargetable wears off

self.shed = 1; //push effect and shred

} else {

return true; //do not remove

}

}

StatusType::Bloodthirster() => {

//if health below 40%

if self.health <= (0.4 \* self.initial\_hp) {

self.shields.push(Shield {

duration: 500,

size: self.initial\_hp / 4.0,

..Default::default()

}); //push shield

} else {

return true; //do not remove

}

}

StatusType::Zephyr(banish\_duration) => {

let opposite\_location = Location {

x: 13 - self.location.y - (self.location.y % 2) - self.location.x,

y: 7 - self.location.y,

}; //gets opposite location

opposite\_location

.get\_closest\_to\_location(enemy\_champions)

.unwrap()

.se

.push(StatusEffect {

duration: Some(banish\_duration),

status\_type: StatusType::Banished(),

..Default::default()

}); //banishes champ closest to that location

}

StatusType::Taunted(taunt\_id) => {

//if taunter still alive

if find\_champion\_index\_from\_id(enemy\_champions, taunt\_id).is\_some() {

self.target = taunt\_id;

self.target\_cooldown = 100;

n\_duration = 20; //set target to taunter

}

}

StatusType::TitansResolve(old\_stack\_num) => {

if old\_stack\_num != 25 { //if old stacks not maxed out

let difference: f32 =

(self.titans\_resolve\_stacks - old\_stack\_num).into();

self.ad += 2.0 \* difference; //gives bonus from difference

self.ap += 0.02 \* difference;

if self.titans\_resolve\_stacks == 25 {

self.ar += 0.25;

self.mr += 0.25;

} //give bonus if maxed out

status\_effect.status\_type =

StatusType::TitansResolve(self.titans\_resolve\_stacks); //update to new stack count

}

return true;

}

StatusType::ProtectorsVow() => {

//if health below half

if self.health <= (self.initial\_hp / 2.0) {

self.mr += 0.25; //give protectors vow effect

self.ar += 0.25;

self.shields.push(Shield {

duration: 500,

size: self.initial\_hp / 4.0,

..Default::default()

})

} else {

return true;

}

}

\_ => (),

}

//if duration updated

if n\_duration > 0 {

status\_effect.duration = Some(n\_duration); //do not remove

} else {

return false; //remove

}

}

}

//if status is not applied yet

if !status\_effect.applied {

status\_effect.applied = true; //is now

//match status type and give associated effect

match status\_effect.status\_type {

StatusType::AttackSpeedBuff(modifier) => {

self.attack\_speed\_modifier \*= modifier;

}

StatusType::Stun() => {

status\_effect.applied = false;

if stun.stun == 0 {

stun.stun = 1;

} //has to check stun.stun == 0 as if stun.stun == 2 it is immune

}

StatusType::IncreaseDamageTaken(modifier) => {

self.incoming\_damage\_modifier \*= modifier;

}

StatusType::Assassin() => {

if self.location.y >= 4 {

self.location.y = 0;

} else {

self.location.y = 7;

}

return false;

}

StatusType::Untargetable() => self.targetable = false,

StatusType::IonicSparkEffect() => {

self.mr /= 2.0;

self.zap = true

}

StatusType::Banished() => self.banish = true,

StatusType::LastWhisperShred() => {

self.ar /= 2.0;

}

StatusType::CrowdControlImmune() => {

status\_effect.applied = false;

stun.stun = 2;

}

\_ => (),

}

}

true

}

///sets id

pub fn \_set\_id(&mut self, id: usize) {

self.id = id //sets id

}

}

impl Default for SummonedChampion {

//default values for summoned champion

fn default() -> Self {

SummonedChampion {

location: Location {

..Default::default()

},

of\_champ\_id: 0,

movement\_progress: [0, 0],

health: 0.0,

cm: 0,

dc: 0,

cr: 0,

crit\_damage: 0.0,

mc: 0,

ar: 0.0,

mr: 0.0,

ad: 0.0,

attack\_speed: 0.0,

ra: 0,

id: 0,

target\_cooldown: 0,

auto\_attack\_delay: 0,

attack\_speed\_modifier: 0.0,

target: 0,

target\_cells: Location { x: 0, y: 0 },

items: [0, 0, 0],

ap: 0.0,

se: Vec::new(),

gain\_mana\_delay: 0,

star\_level: 0,

incoming\_damage\_modifier: 0.0,

initial\_hp: 0.0,

targetable: false,

shed: 0,

shields: Vec::new(),

zap: false,

banish: false,

titans\_resolve\_stacks: 0,

omnivamp: 0.0,

shiv\_attack\_count: 0,

}

}

}

impl fmt::Display for SummonedChampion {

fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {

write!(f, "{}: {}", self.id, self.health)

}

}

**Item.rs:**

/// Import serde and surrealdb for frontend-backend and database communication

use serde::{Serialize, Deserialize};

use surrealdb::sql::{Object, Value};

use crate::prelude::\*;

///default items

pub const DEFAULT\_ITEMS : [Item ; 47] = [

    Item {id : 1, ad : 10.0, health : 0.0, ap : 0.0, ar : 0.0, mr : 0.0, ra : 0, attack\_speed\_modifier : 1.0, cr : 0, dc : 0, cm : 0, omnivamp : 0.0, crit\_damage : 0.0},

    Item {id : 2, ap : 0.1, health : 0.0, ad : 0.0, ar : 0.0, mr : 0.0, ra : 0, attack\_speed\_modifier : 1.0, cr : 0, dc : 0, cm : 0, omnivamp : 0.0, crit\_damage : 0.0},

    Item {id : 3, health : 150.0, ad : 0.0, ap : 0.0, ar : 0.0, mr : 0.0, ra : 0, attack\_speed\_modifier : 1.0, cr : 0, dc : 0, cm : 0, omnivamp : 0.0, crit\_damage : 0.0},

    Item {id : 4, ar : 0.2, health : 0.0, ad : 0.0, ap : 0.0, mr : 0.0, ra : 0, attack\_speed\_modifier : 1.0, cr : 0, dc : 0, cm : 0, omnivamp : 0.0, crit\_damage : 0.0},

    Item {id : 5, mr : 0.2, health : 0.0, ad : 0.0, ap : 0.0, ar : 0.0, ra : 0, attack\_speed\_modifier : 1.0, cr : 0, dc : 0, cm : 0, omnivamp : 0.0, crit\_damage : 0.0},

    Item {id : 6, attack\_speed\_modifier : 1.1, health : 0.0, ad : 0.0, ap : 0.0, ar : 0.0, mr : 0.0, ra : 0, cr : 0, dc : 0, cm : 0, omnivamp : 0.0, crit\_damage : 0.0},

    Item {id : 7, cr : 5, dc : 10, health : 0.0, ad : 0.0, ap : 0.0, ar : 0.0, mr : 0.0, ra : 0, attack\_speed\_modifier : 1.0, cm : 0, omnivamp : 0.0, crit\_damage : 0.0},

    Item {id : 8, cm : 15, health : 0.0, ad : 0.0, ap : 0.0, ar : 0.0, mr : 0.0, ra : 0, attack\_speed\_modifier : 1.0, cr : 0, dc : 0, omnivamp : 0.0, crit\_damage : 0.0},

    Item {id : 11, ad : 15.0, health : 0.0, ap : 0.0, ar : 0.0, mr : 0.0, ra : 0, attack\_speed\_modifier : 1.0, cr : 0, dc : 0, cm : 0, omnivamp : 0.0, crit\_damage : 0.0},

    Item {id : 12, ad : 10.0, ap : 0.1, health : 0.0, ar : 0.0, mr : 0.0, ra : 0, attack\_speed\_modifier : 1.0, cr : 0, dc : 0, cm : 0, omnivamp : 0.0, crit\_damage : 0.0},

    Item {id : 13, ad : 10.0, health : 150.0, attack\_speed\_modifier : 1.3, ap : 0.0, ar : 0.0, mr : 0.0, ra : 0, cr : 0, dc : 0, cm : 0, omnivamp : 0.0, crit\_damage : 0.0},

    Item {id : 14, ad : 10.0, ar : 0.2, health : 0.0, ap : 0.0, mr : 0.0, ra : 0, attack\_speed\_modifier : 1.0, cr : 0, dc : 0, cm : 0, omnivamp : 0.0, crit\_damage : 0.0},

    Item {id : 15, ad : 10.0, mr : 0.2, omnivamp : 0.25, health : 0.0, ap : 0.0, ar : 0.0, ra : 0, attack\_speed\_modifier : 1.0, cr : 0, dc : 0, cm : 0, crit\_damage : 0.0},

    Item {id : 16, ad : 10.0, attack\_speed\_modifier : 1.1, health : 0.0, ap : 0.0, ar : 0.0, mr : 0.0, ra : 0, cr : 0, dc : 0, cm : 0, omnivamp : 0.0, crit\_damage : 0.0},

    Item {id : 17, ad : 10.0, cm : 75, crit\_damage : 0.1, health : 0.0, ap : 0.0, ar : 0.0, mr : 0.0, ra : 0, attack\_speed\_modifier : 1.0, cr : 0, dc : 0, omnivamp : 0.0},

    Item {id : 18, ad : 10.0, cm : 15, health : 0.0, ap : 0.0, ar : 0.0, mr : 0.0, ra : 0, attack\_speed\_modifier : 1.0, cr : 0, dc : 0, omnivamp : 0.0, crit\_damage : 0.0},

    Item {id : 19, ad : 10.0, health : 0.0, ap : 0.0, ar : 0.0, mr : 0.0, ra : 0, attack\_speed\_modifier : 1.0, cr : 0, dc : 0, cm : 0, omnivamp : 0.0, crit\_damage : 0.0},

    Item {id : 22, ap : 0.75, health : 0.0, ad : 0.0, ar : 0.0, mr : 0.0, ra : 0, attack\_speed\_modifier : 1.0, cr : 0, dc : 0, cm : 0, omnivamp : 0.0, crit\_damage : 0.0},

    Item {id : 23, ap : 0.4, health : 150.0, ad : 0.0, ar : 0.0, mr : 0.0, ra : 0, attack\_speed\_modifier : 1.0, cr : 0, dc : 0, cm : 0, omnivamp : 0.0, crit\_damage : 0.0},

    Item {id : 24, ap : 0.1, ar : 0.2, health : 0.0, ad : 0.0, mr : 0.0, ra : 0, attack\_speed\_modifier : 1.0, cr : 0, dc : 0, cm : 0, omnivamp : 0.0, crit\_damage : 0.0},

    Item {id : 25, ap : 0.1, mr : 0.2, health : 0.0, ad : 0.0, ar : 0.0, ra : 0, attack\_speed\_modifier : 1.0, cr : 0, dc : 0, cm : 0, omnivamp : 0.0, crit\_damage : 0.0},

    Item {id : 26, ap : 0.1, attack\_speed\_modifier : 1.1, health : 0.0, ad : 0.0, ar : 0.0, mr : 0.0, ra : 0, cr : 0, dc : 0, cm : 0, omnivamp : 0.0, crit\_damage : 0.0},

    Item {id : 27, ap : 0.5, cr : 15, crit\_damage : 0.4, health : 0.0, ad : 0.0, ar : 0.0, mr : 0.0, ra : 0, attack\_speed\_modifier : 1.0, dc : 0, cm : 0, omnivamp : 0.0},

    Item {id : 28, ap : 0.1, cm : 15, health : 0.0, ad : 0.0, ar : 0.0, mr : 0.0, ra : 0, attack\_speed\_modifier : 1.0, cr : 0, dc : 0, omnivamp : 0.0, crit\_damage : 0.0},

    Item {id : 29, ap : 0.1, health : 0.0, ad : 0.0, ar : 0.0, mr : 0.0, ra : 0, attack\_speed\_modifier : 1.0, cr : 0, dc : 0, cm : 0, omnivamp : 0.0, crit\_damage : 0.0},

    Item {id : 33, health : 1000.0, ad : 0.0, ap : 0.0, ar : 0.0, mr : 0.0, ra : 0, attack\_speed\_modifier : 1.0, cr : 0, dc : 0, cm : 0, omnivamp : 0.0, crit\_damage : 0.0},

    Item {id : 34, health : 300.0, ar : 0.2, ad : 0.0, ap : 0.0, mr : 0.0, ra : 0, attack\_speed\_modifier : 1.0, cr : 0, dc : 0, cm : 0, omnivamp : 0.0, crit\_damage : 0.0},

    Item {id : 35, health : 150.0, mr : 0.2, ad : 0.0, ap : 0.0, ar : 0.0, ra : 0, attack\_speed\_modifier : 1.0, cr : 0, dc : 0, cm : 0, omnivamp : 0.0, crit\_damage : 0.0},

    Item {id : 36, health : 150.0, attack\_speed\_modifier : 1.1, ad : 0.0, ap : 0.0, ar : 0.0, mr : 0.0, ra : 0, cr : 0, dc : 0, cm : 0, omnivamp : 0.0, crit\_damage : 0.0},

    Item {id : 37, health : 150.0, dc : 15, ad : 0.0, ap : 0.0, ar : 0.0, mr : 0.0, ra : 0, attack\_speed\_modifier : 1.0, cr : 0, cm : 0, omnivamp : 0.0, crit\_damage : 0.0},

    Item {id : 38, health : 150.0, cm : 15, ad : 0.0, ap : 0.0, ar : 0.0, mr : 0.0, ra : 0, attack\_speed\_modifier : 1.0, cr : 0, dc : 0, omnivamp : 0.0, crit\_damage : 0.0},

    Item {id : 39, health : 150.0, ad : 0.0, ap : 0.0, ar : 0.0, mr : 0.0, ra : 0, attack\_speed\_modifier : 1.0, cr : 0, dc : 0, cm : 0, omnivamp : 0.0, crit\_damage : 0.0},

    Item {id : 44, ar : 0.8, health : 0.0, ad : 0.0, ap : 0.0, mr : 0.0, ra : 0, attack\_speed\_modifier : 1.0, cr : 0, dc : 0, cm : 0, omnivamp : 0.0, crit\_damage : 0.0},

    Item {id : 45, ar : 0.2, mr : 0.2, health : 0.0, ad : 0.0, ap : 0.0, ra : 0, attack\_speed\_modifier : 1.0, cr : 0, dc : 0, cm : 0, omnivamp : 0.0, crit\_damage : 0.0},

    Item {id : 46, ar : 0.2, attack\_speed\_modifier : 1.1, health : 0.0, ad : 0.0, ap : 0.0, mr : 0.0, ra : 0, cr : 0, dc : 0, cm : 0, omnivamp : 0.0, crit\_damage : 0.0},

    Item {id : 47, ar : 0.2, dc : 15, health : 0.0, ad : 0.0, ap : 0.0, mr : 0.0, ra : 0, attack\_speed\_modifier : 1.0, cr : 0, cm : 0, omnivamp : 0.0, crit\_damage : 0.0},

    Item {id : 48, ar : 0.2, cm : 15, health : 0.0, ad : 0.0, ap : 0.0, mr : 0.0, ra : 0, attack\_speed\_modifier : 1.0, cr : 0, dc : 0, omnivamp : 0.0, crit\_damage : 0.0},

    Item {id : 55, mr : 1.2, health : 0.0, ad : 0.0, ap : 0.0, ar : 0.0, ra : 0, attack\_speed\_modifier : 1.0, cr : 0, dc : 0, cm : 0, omnivamp : 0.0, crit\_damage : 0.0},

    Item {id : 56, mr : 0.2, ad : 10.0, attack\_speed\_modifier : 1.1, health : 0.0, ap : 0.0, ar : 0.0, ra : 0, cr : 0, dc : 0, cm : 0, omnivamp : 0.0, crit\_damage : 0.0},

    Item {id : 57, mr : 0.2, dc : 15, attack\_speed\_modifier : 1.2, health : 0.0, ad : 0.0, ap : 0.0, ar : 0.0, ra : 0, cr : 0, cm : 0, omnivamp : 0.0, crit\_damage : 0.0},

    Item {id : 58, cm : 15, mr : 0.2, ap : 0.3, health : 0.0, ad : 0.0, ar : 0.0, ra : 0, attack\_speed\_modifier : 1.0, cr : 0, dc : 0, omnivamp : 0.0, crit\_damage : 0.0},

    Item {id : 66, attack\_speed\_modifier : 1.55, ra : 1, health : 0.0, ad : 0.0, ap : 0.0, ar : 0.0, mr : 0.0, cr : 0, dc : 0, cm : 0, omnivamp : 0.0, crit\_damage : 0.0},

    Item {id : 67, attack\_speed\_modifier : 1.21, cr : 15, health : 0.0, ad : 0.0, ap : 0.0, ar : 0.0, mr : 0.0, ra : 0, dc : 0, cm : 0, omnivamp : 0.0, crit\_damage : 0.0},

    Item {id : 68, attack\_speed\_modifier : 1.21, cm : 15, health : 0.0, ad : 0.0, ap : 0.0, ar : 0.0, mr : 0.0, ra : 0, cr : 0, dc : 0, omnivamp : 0.0, crit\_damage : 0.0},

    Item {id : 77, cr : 15, dc : 15, health : 0.0, ad : 0.0, ap : 0.0, ar : 0.0, mr : 0.0, ra : 0, attack\_speed\_modifier : 1.0, cm : 0, omnivamp : 0.0, crit\_damage : 0.0},

    Item {id : 78, cm : 10, cr : 15, health : 0.0, ad : 0.0, ap : 0.0, ar : 0.0, mr : 0.0, ra : 0, attack\_speed\_modifier : 1.0, dc : 0, omnivamp : 0.0, crit\_damage : 0.0},

    Item {id : 88, cm : 50, health : 0.0, ad : 0.0, ap : 0.0, ar : 0.0, mr : 0.0, ra : 0, attack\_speed\_modifier : 1.0, cr : 0, dc : 0, omnivamp : 0.0, crit\_damage : 0.0},

];

///item struct holding base stats and ids

#[derive(Clone, Copy, Serialize, Deserialize)]

pub struct Item {

    pub id : u8,

    pub health : f32,

    pub ad : f32,

    pub ap : f32,

    pub ar : f32,

    pub mr : f32,

    pub attack\_speed\_modifier : f32,

    pub ra : i8,

    pub cr : u8,

    pub dc : u8,

    pub cm : i16,

    pub omnivamp : f32,

    pub crit\_damage : f32,

}

impl Item {

    ///converts into string value array for insertion in database

    pub fn into\_values(&self) -> [(String, Value) ; 13] {

        [

            ("id".into(), self.id.into()),

            ("health".into(), self.health.into()),

            ("ad".into(), self.ad.into()),

            ("ap".into(), self.ap.into()),

            ("ar".into(), self.ar.into()),

            ("mr".into(), self.mr.into()),

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

            ("ra".into(), self.ra.into()),

            ("cr".into(), self.cr.into()),

            ("dc".into(), self.dc.into()),

            ("cm".into(), self.cm.into()),

            ("omnivamp".into(), self.omnivamp.into()),

            ("crit\_damage".into(), self.crit\_damage.into())

        ]

    }

}

impl TryFrom<Object> for Item {

    type Error = Error;

    ///tries to convert from a database object into an item object

    fn try\_from(mut obj: Object) -> Result<Self> {

        //fetches values from database like object

        let ad = obj.remove("ad").unwrap().as\_float() as f32;

        let ap = obj.remove("ap").unwrap().as\_float() as f32;

        let ar = obj.remove("ar").unwrap().as\_float() as f32;

        let mr = obj.remove("mr").unwrap().as\_float() as f32;

        let attack\_speed\_modifier = obj.remove("attack\_speed\_modifier").unwrap().as\_float() as f32;

        let cm = obj.remove("cm").unwrap().as\_int() as i16;

        let cr = obj.remove("cr").unwrap().as\_int() as u8;

        let crit\_damage = obj.remove("crit\_damage").unwrap().as\_float() as f32;

        let dc = obj.remove("dc").unwrap().as\_int() as u8;

        let health = obj.remove("health").unwrap().as\_float() as f32;

        let omnivamp = obj.remove("omnivamp").unwrap().as\_float() as f32;

        let ra = obj.remove("ra").unwrap().as\_int() as i8;

        let id : u8 = Value::from(obj.remove("id").unwrap().record().unwrap().id).as\_int() as u8;

        Ok(Item { id, ad, ap, attack\_speed\_modifier, health, ar, mr, ra, cr, dc, cm, omnivamp, crit\_damage})

    }

}

impl Default for Item {

    ///default values for item

    fn default() -> Self {

        Item {

            id : 0,

            health : 0.0,

            ad : 0.0,

            ap : 0.0,

            ar : 0.0,

            mr : 0.0,

            ra : 0,

            attack\_speed\_modifier : 1.0,

            cr : 0,

            dc : 0,

            cm : 0,

            omnivamp : 0.0,

            crit\_damage : 0.0,

        }

    }

}

**Item.rs:**

use super::champions::SummonedChampion;

use serde::{Deserialize, Serialize};

use std::collections::VecDeque;

use rand::Rng;

///Different types of filters

enum FilterType {

    ///i8 : Distance to check

    ///Location : Other Location

    DistanceFilter(i8, Location),

}

///generates a filter based on a given filter type

fn generate\_filter(filter: FilterType) -> impl for<'a> Fn(&&mut SummonedChampion) -> bool {

    match filter {

        FilterType::DistanceFilter(dis, location) => {

            move |n: &&mut SummonedChampion| n.location.distance\_between\_points(&location) < dis

        } //returns a function that moves the given SummonedChampion into the enclosure, and returns a bool depending on whether the distance between a point and the summonedchampion location is low enough

    }

}

///Location class holding x y and lots of useful methods

#[derive(Debug, Default, Clone, Copy, PartialEq, Eq, Deserialize, Serialize)]

pub struct Location {

    ///x position

    pub x: i8,

    ///y position

    pub y: i8,

}

impl Location {

    ///calculates the z value of a location

    fn calculate\_z(&self) -> i8 {

        -self.x - self.y

    }

    ///calculates the distance between two points

    pub fn distance\_between\_points(&self, other\_pos: &Location) -> i8 {

        (self.x - other\_pos.x).abs() //x distance

            + (self.y - other\_pos.y).abs() //y distance

            + (self.calculate\_z() - other\_pos.calculate\_z()).abs() //calcalates z then distance between them

    }

    ///subtracts two positions, returning the new location

    pub fn sub\_positions(pos\_one: &Location, pos\_two: &Location) -> Location {

        Location {

            x: pos\_one.x - pos\_two.x,

            y: pos\_one.y - pos\_two.y,

        }

    }

    ///returns a new location, of a location added to a array two long

    pub fn add\_position\_vec(pos\_one: &Location, pos\_two: [i8; 2]) -> Location {

        Location {

            x: pos\_one.x + pos\_two[0],

            y: pos\_one.y + pos\_two[1],

        }

    }

    ///returns a bool of whether the position is valid/ in the grid

    pub fn check\_valid(&self) -> bool {

           self.x >= 0

        && self.x < 10

        && self.y >= 0

        && self.y < 8

        && 2 - (self.y / 2) < self.x

        && 10 - (self.y / 2) >= self.x //last two lines account for slanting border/ x value

    }

    ///generates a random position given a team (what side of the board to generate on)

    pub fn generate\_random\_position\_team(team: bool) -> Location {

        let y: i8 = if team {

            rand::thread\_rng().gen\_range(0..4) //generates random y

        } else {

            rand::thread\_rng().gen\_range(4..8)

        };

        let low = 2 - (y / 2) + 1; //calculate max and min values for x as they are dependant on y

        let high = 10 - (y / 2);

        let x: i8 = rand::thread\_rng().gen\_range(low..high);

        Location { x, y }

    }

    ///given a vector of summonedchamps, return the closest to self

    pub fn get\_closest\_to\_location<'a>(

        &self,

        champion\_list: &'a mut VecDeque<SummonedChampion>,

    ) -> Option<&'a mut SummonedChampion> {

        //iterates through the champion\_list, reducing it by comparing the distance to this location (reducing is comparing two sequential elements in the list and keeping the smaller one/ one that fits the bounds and repeating for the entire list, until you have only one)

        champion\_list.iter\_mut().reduce(|x, y| {

            if x.location.distance\_between\_points(self) < y.location.distance\_between\_points(self) {

                x

            } else {

                y

            }

        })

    }

    ///given a vector of summonedchamps, return the closest to self that is targetable

    pub fn get\_closest\_to\_location\_targetable<'a>(

        &self,

        enemy\_champions: &'a mut VecDeque<SummonedChampion>,

    ) -> Option<&'a mut SummonedChampion> {

        enemy\_champions.iter\_mut() //turn vector into iterator

            .filter(|x| x.get\_is\_targetable()) //filters through the iterator, not yielding any elements that are not targetable

            .reduce(|x, y| { //reduce the values by comparing the distance from self

            if x.location.distance\_between\_points(self) < y.location.distance\_between\_points(self) {

                return x;

            }

            y

        })

    }

    ///given a vector of summonedchamps, return the closest to self that is targetable, along with its index

    pub fn get\_closest\_to\_location\_targetable\_index<'a>(

        &self,

        enemy\_champions: &'a mut VecDeque<SummonedChampion>,

    ) -> Option<(usize, &'a mut SummonedChampion)> {

        enemy\_champions

            .iter\_mut() //turn into iterator

            .enumerate() //get the indexes of the champions, BEFORE the filter so that indexes are valid/ accurate to given vector

            .filter(|(\_, x)| x.get\_is\_targetable()) //remove any champions that are not targetable

            .reduce(|(i, x), (j, y)| { //reduce by distance to point

                if x.location.distance\_between\_points(self)

                    < y.location.distance\_between\_points(self)

                {

                    return (i, x);

                }

                (j, y)

            })

    }

    ///given a distance, generates a filter to be used with .filter

    pub fn get\_within\_distance(

        self,

        distance: i8,

    ) -> impl for<'a> Fn(&&mut SummonedChampion) -> bool {

        generate\_filter(FilterType::DistanceFilter(distance, self)) //utilises the generate filter function

    }

}

impl std::fmt::Display for Location {

    fn fmt(&self, f: &mut std::fmt::Formatter<'\_>) -> std::fmt::Result {

        write!(f, "x, y, z  {}, {}, {}", self.x, self.y, self.calculate\_z())

    }

}

**Mod.rs:**

/\* Exports Crate Modules \*/

pub mod board;

pub mod champions;

mod location;

mod projectiles;

mod shields;

mod status\_effects;

mod utils;

pub mod item;

**perform\_tests.rs: (currently test for complex battle generated)**

use super::board::Board;

use super::champions::PlacedChampion;

use super::location::Location;

use std::collections::VecDeque;

use super::champions::DEFAULT\_CHAMPIONS;

use super::item::DEFAULT\_ITEMS;

use std::time::{Duration, Instant};

pub fn perform\_test() {

    let start = Instant::now();

    for \_ in 0..1000 {

        let mut board = Board::generate\_complex\_random\_board(10, &DEFAULT\_CHAMPIONS, &DEFAULT\_ITEMS, 10000);

        board.simulate\_battle(None);

    }

    let duration = start.elapsed();

    println!("Time to simulate 1000 battles is is: {:?}", duration);

}

**projectiles.rs:**

//import require types from other files

use super::{

    champions::{DamageType, SummonedChampion},

    location::Location,

    utils::{find\_champion\_index\_from\_id, sign},

};

//import serialise and vecdeque

use serde::Serialize;

use std::collections::VecDeque;

///Projectile struct

#[derive(Clone, Serialize)]

pub struct Projectile {

    ///location of projectile

    location: Location,

    ///location progress

    location\_progress: [i8; 2],

    ///target location

    target\_location: Option<Location>,

    ///enemy Champion index

    target\_id: usize,

    ///projectile damage

    damage: f32,

    ///projectile damage type

    damage\_type: DamageType,

    ///amount of splash damage

    splash\_damage: f32,

    ///speed of projectile

    speed: i8,

    ///id of shooter (so can give item effects etc)

    shooter\_id: usize,

}

impl Projectile {

    ///Simulates a single tick of a projectile

    pub fn simulate\_tick(

        self: &mut Projectile,

        possible\_targets: &mut VecDeque<SummonedChampion>,

        friendly\_champions: &mut VecDeque<SummonedChampion>,

        dead\_champions: &mut VecDeque<SummonedChampion>,

    ) -> bool {

        info!("Simulating projectile");

        let target\_location = match self.target\_location //if self has target location, set target location to that, else get the location of the target champion.

        {

            Some(location) => {

                info!("Target {location}");

                location}, //gets target location

            None => {

                let out\_location = find\_champion\_index\_from\_id(possible\_targets, self.target\_id);//gets location of target champion

                info!("Finding location from id : {:?}", out\_location);

                match out\_location

                {

                    Some(index) => possible\_targets[index].location, //if target is still alive, return its location

                    None => Location { x: -1, y: -1 }, //set location to invalid

                }

        }};

        if target\_location.x == -1 {

            info!("Not valid location, removing");

            return false;

        } //not found, remove projectile

        let subtracted\_distance = Location::sub\_positions(&target\_location, &self.location); //get location difference

        self.location\_progress[0] += self.speed \* sign(subtracted\_distance.x);

        self.location\_progress[1] += self.speed \* sign(subtracted\_distance.y); //add location progress

        if self.location\_progress[0].abs() >= 10

        //if above 10, move

        {

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

        }

        if self.location\_progress[1].abs() >= 10 {

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

        }

        if !self.location.check\_valid() {

            info!("Out of grid leaving");

            return false;

        } //if out of grid, remove

        for possible\_target in possible\_targets.iter\_mut()

        //iterate through all possible collisions

        {

            if self.location == possible\_target.location

            //has a hit

            {

                info!("has a hit");

                let mut dead = false; //stores whether need to add to dead champions or alive

                let mut shooter: SummonedChampion;

                //if shooter alive, fetch from friendly champions, else fetch from dead champions, this is because deal damage requires damage dealer to apply correct effects

                if let Some(shooter\_index) =

                    find\_champion\_index\_from\_id(friendly\_champions, self.shooter\_id)

                {

                    //finds shooter id

                    shooter = friendly\_champions.swap\_remove\_back(shooter\_index).unwrap();

                    info!("shooter alive");

                } else {

                    let dead\_champion\_index =

                        find\_champion\_index\_from\_id(dead\_champions, self.shooter\_id).unwrap(); //fetch from dead champions

                    shooter = dead\_champions.swap\_remove\_back(dead\_champion\_index).unwrap();

                    dead = true; //remember to add to dead champions later

                    info!("shooter dead")

                }

                shooter.deal\_damage(

                    friendly\_champions,

                    possible\_target,

                    self.damage,

                    self.damage\_type,

                    false,

                ); //deals damage

                if self.splash\_damage > 0.0

                //if there is splash damage

                {

                    let initial\_hit = possible\_target.id;

                    info!("dealing splash");

                    for possible\_secondary\_target in possible\_targets

                        .iter\_mut()

                        .filter(self.location.get\_within\_distance(3))

                    //iterate through possible splash hits

                    {

                        if possible\_secondary\_target.id == initial\_hit { continue }

                        shooter.deal\_damage(

                            friendly\_champions,

                            possible\_secondary\_target,

                            self.splash\_damage,

                            self.damage\_type,

                            true,

                        ); //deal secondary dmg

                    }

                }

                if !dead {

                    friendly\_champions.push\_back(shooter) //push to alive

                } else {

                    dead\_champions.push\_back(shooter) //push to dead

                }

                return false; //has exploded, so return false

            }

        }

        if self.target\_location.is\_some() && self.target\_location.unwrap() == self.location {

            return false;

        }

        true //still alive

    }

    ///Makes new projectile

    pub fn new(

        location: Location,

        target\_location: Option<Location>,

        target\_id: usize,

        damage: f32,

        damage\_type: DamageType,

        splash\_damage: f32,

        speed: i8,

        shooter\_id: usize,

    ) -> Projectile {

        Projectile {

            location,

            location\_progress: [0, 0],

            target\_location,

            target\_id,

            damage,

            damage\_type,

            splash\_damage,

            speed,

            shooter\_id,

        }

    }

}

**Shields.rs:**

use serde::Serialize;

use super::champions::DamageType;

///Implementation for Shields

#[derive(Debug, Clone, Serialize)]

pub struct Shield {

    ///duration of shield

    pub duration: i16,

    ///number of damage blocked

    pub size: f32,

    ///Optional choice for whether it only blocks a certain type

    pub blocks\_type: Option<DamageType>,

    ///Whether it pops after receiving any damage

    pub pop: bool,

}

impl Shield {

    ///updates shield, reducing duration with time unit returns bool whether should be kept or removed

    pub fn update\_shield(&mut self, time\_unit: i8) -> bool {

        //updates duration

        self.duration -= time\_unit as i16;

        info!("updating shield {} {}", self.duration, self.size);

        //returns whether duration and size is still above zero

        self.duration > 0 && self.size > 0.0

    }

    ///handles incoming damage, returning the remaining damage

    pub fn handle\_damage(&mut self, damage: f32, damage\_type: DamageType) -> f32 {

        if self.blocks\_type.is\_none() || self.blocks\_type.unwrap() == damage\_type { //if it blocks all types or the specific damage type

            let out = damage - self.size; //reduce the damage

            self.size -= damage; //reduce the size

            if self.pop { //if pop ie removes self after any damage

                self.size = 0.0;

            }

            return out.max(0.0); //returns the damage or 0, whatever larger

        }

        damage

    }

}

///Default for shield

impl Default for Shield {

    fn default() -> Shield {

        Shield {

            duration: 0,

            size: 0.0,

            blocks\_type: None,

            pop: false,

        }

    }

}

**Status\_effects.rs:**

use serde::Serialize;

///Records champion's stun state.<br />

pub struct Stun {

    ///Records whether champion is stunned:<br />

    ///0 = not stunned<br />

    ///1 = stunned<br />

    ///2 = stun immune

    pub stun: u8,

}

///Status Type (enum):<br />

///Holds information about what the status does

#[derive(PartialEq, Debug, Clone, Serialize)]

pub enum StatusType {

    ///Attack Speed Buff:<br />

    ///(bool : whether the buff has been applied, f32 : actual modifier)

    AttackSpeedBuff(f32),

    ///Increase Damage Taken:<br />

    ///(bool : whether the buff has been applied, i32 : actual modifier in % (so 120 = 120% or 20% increase))

    IncreaseDamageTaken(f32),

    ///Stun

    Stun(),

    ///Grevious Wounds:<br />

    ///Reduces healing by 50%

    GreviousWounds(),

    ///Gives edge of night buff<br />:

    EdgeOfNight(),

    ///Whether the target is targetable

    ///bool : Whether the buff has been applied

    Untargetable(),

    ///Bloodthirster shield at 40%

    Bloodthirster(),

    ///Assassin trait leap

    #[allow(dead\_code)]

    Assassin(),

    ///Morellonomicon Burn:<br />

    ///(f32 : damage per tick, f32 : damage to do, i16 : time til next tick)

    MorellonomiconBurn(f32, f32, i16),

    ///Ionic spark effect:<br />

    ///Reduces MR by 50%<br />

    IonicSparkEffect(), //maybe discrepencies? awkward cuz only lasts 1 frame?

    ///Archangel Staff:<br />

    ///(bool : applied. f32 : ap increase)

    ArchangelStaff(f32),

    ///Zephyr Item:<br />

    ///(bool : applied, i16 : banish duration)

    Zephyr(i16),

    ///Banished:<br />

    ///(bool : applied)

    Banished(),

    ///Taunted:<br />

    ///(usize : ID of taunter)

    Taunted(usize),

    ///Redemption:<br />

    ///(bool : applied)

    RedemptionGive(),

    ///Gargoyles Item Effect:<br />

    ///(f32: How many were targeting previous frame)

    Gargoyles(f32),

    ///Titans Resolve Item Effect:<br />

    ///(u8: Number of stacks previous frame)

    TitansResolve(u8),

    ///Shroud of Stillness Item Effect:<br />

    ///Immediately removed/ used at start of game

    ShroudOfStillness(),

    ///Protectors Vow Item Effect:<br />

    ProtectorsVow(),

    ///Dragon Claw Heal Item Effect:<br />

    DragonClawHeal(),

    ///Immune of CC Effect:<br />

    CrowdControlImmune(),

    ///Last Whisper Armor Shred Effect:<br />

    ///(bool : applied)

    LastWhisperShred(),

    ///Shreds Magic Resist Effect:<br />

    ///(bool : applied, f32 : multiplyer/ effect)

    ShredMagicResist(f32),

    ///Gives sunfire effect:<br />

    ///Not implemented

    GiveSunfire(),

    ///None

    NoEffect(),

}

///StatusEffect (struct)<br />:

///Stores a status type and a duration

#[derive(Debug, Clone, Serialize)]

pub struct StatusEffect {

    ///Duration of status effect in centiseconds

    pub duration: Option<i16>, //optimisation so uses Option<i16> rather than i16

    ///Whether the status effect has been applied

    pub applied: bool,

    ///Stores status type

    pub status\_type: StatusType,

    ///Whether is negative for shred

    pub is\_negative: bool,

}

///Default Status Effect Values

impl Default for StatusEffect {

    fn default() -> StatusEffect {

        StatusEffect {

            duration: None,

            applied: false,

            status\_type: StatusType::NoEffect(),

            is\_negative: false,

        }

    }

}

impl PartialEq for StatusEffect {

    fn eq(&self, other: &Self) -> bool { //checks whether two status effects share a status type

        self.status\_type == other.status\_type

    }

}

**Utils.rs:**

use super::champions::SummonedChampion;

use std::collections::VecDeque;

///findChampionIndexFromID:<br />

///champions : &Vec<SummonedChampion> - List of champions to iterate through<br />

///id : usize - ID wanted<br />

///returns : Option<usize> - Some(correct id) or None if not found

pub fn find\_champion\_index\_from\_id(

    champions: &VecDeque<SummonedChampion>,

    id: usize,

) -> Option<usize> {

    info!("finding champ from id");

    //else checks every champion for the id

    for (i, champ) in champions.iter().enumerate() {

        if champ.equal\_id(id) {

            info!("found from id");

            return Some(i);

        }

    }

    None

}

///Same as find champ index from id but also checks it is targetable/ not banished

pub fn find\_champion\_index\_from\_id\_targetable(

    champions: &VecDeque<SummonedChampion>,

    id: usize,

) -> Option<usize> {

    for (i, champ) in champions.iter().enumerate() { //iterates through all champions searching for index

        if champ.equal\_id(id) {

            //checks if targetable

            if champ.get\_is\_targetable() {

                return Some(i)

            }

            //not targetable, save time by not iterating through rest of vector

            return None

        }

    }

    //champ not found

    None

}

///0 if num is 0, 1 if num > 0, -1 if num < 0

pub fn sign(num: i8) -> i8 {

    if num == 0 {

        return 0;

    } else if num > 0 {

        return 1;

    }

    -1

}

**Error.rs:**

use serde::Serialize;

///All possible errors for this project (not related to simulator)

#[derive(thiserror::Error, Debug, Serialize)]

pub enum Error {

    ///Error from SurrealDB

    #[error(transparent)]

    Surreal(#[from] surrealdb::Error),

    ///Error related to database

    #[error("Database Error: '{0}'")]

    DatabaseError(&'static str),

    ///Error retrieving Store from Ctx/ Tauri

    #[error("Failed to retrieve store")]

    StoreError,

    ///Error fetching Board from Store

    #[error("Failed to fetch board")]

    FetchBoardError,

    ///Error fetching last board from Store

    #[error("Failed to retrieve last board")]

    LastBoardError,

}

**Ipc.rs:**

use crate::prelude::\*;

use crate::simulator::champions::PlacedChampion;

use crate::simulator::{champions::Champion, item::Item, board::Board};

use crate::store::Store;

use std::collections::VecDeque;

use std::sync::{Arc};

use tokio::sync::{RwLock};

use tauri::{command, AppHandle, Manager, Wry};

///fetches a store from the connection

fn get\_store\_read\_from\_state(connection: AppHandle<Wry>) -> Result<Arc<RwLock<Store>>> {

    Ok((\*connection.state::<Arc<RwLock<Store>>>()).clone())

}

///retrieves a unit from an id

#[command]

pub async fn retrieve\_unit\_from\_id(id: u8, connection: AppHandle<Wry>) -> Result<Option<Champion>> {

    //fetch store

    if let Ok(store) = get\_store\_read\_from\_state(connection) {

        return store.read().await.fetch\_champion\_from\_id(id).await //get store read only, call fetch champion from id, await response

    }

    //return failed to fetch store

    Err(Error::StoreError)

}

///retrieve an item from an id

#[command]

pub async fn retrieve\_item\_from\_id(id: u8, connection: AppHandle<Wry>) -> Result<Option<Item>> {

    //fetch store

    if let Ok(store) = get\_store\_read\_from\_state(connection) {

        return store.read().await.fetch\_item\_from\_id(id).await //get store read only, call fetch item from id, await response

    }

    //return failed to fetch store

    Err(Error::StoreError)

}

///retrieve all units

#[command]

pub async fn retrieve\_all\_units(connection : AppHandle<Wry>) -> Result<Vec<Champion>> {

    if let Ok(store) = get\_store\_read\_from\_state(connection) {

        return store.read().await.fetch\_champions().await

    }

    Err(Error::StoreError)

}

///retrieve all items

#[command]

pub async fn retrieve\_all\_items(connection : AppHandle<Wry>) -> Result<Vec<Item>> {

    if let Ok(store) = get\_store\_read\_from\_state(connection) {

        return store.read().await.fetch\_items().await

    }

    Err(Error::StoreError)

}

///retrieve all unit ids

#[command]

pub async fn retrieve\_all\_unit\_ids(connection : AppHandle<Wry>) -> Result<Vec<u8>> {

    if let Ok(store) = get\_store\_read\_from\_state(connection) {

        return store.read().await.fetch\_champions\_ids().await

    }

    Err(Error::StoreError)

}

///retrieve all item ids

#[command]

pub async fn retrieve\_all\_item\_ids(connection : AppHandle<Wry>) -> Result<Vec<u8>> {

    if let Ok(store) = get\_store\_read\_from\_state(connection) {

        return store.read().await.fetch\_items\_ids().await

    }

    Err(Error::StoreError)

}

///update a unit with new values

#[command]

pub async fn update\_unit(selected\_unit : Champion, connection : AppHandle<Wry>) -> Result<()> {

    if let Ok(store) = get\_store\_read\_from\_state(connection) {

        return store.read().await.update\_champion(selected\_unit).await;

    }

    Err(Error::StoreError)

}

///update an item with new values

#[command]

pub async fn update\_item(selected\_item : Item, connection : AppHandle<Wry>) -> Result<()> {

    if let Ok(store) = get\_store\_read\_from\_state(connection) {

        return store.read().await.update\_item(selected\_item).await

    }

    Err(Error::StoreError)

}

///take a board from the frontend and set the store to hold that board

#[command]

pub async fn submit\_board(player\_one\_champs : VecDeque<PlacedChampion>, player\_two\_champs : VecDeque<PlacedChampion>, time\_unit : i8, time\_till\_draw: u32, connection : AppHandle<Wry>) -> Result<()> {

    if let Ok(store) = get\_store\_read\_from\_state(connection) {

        let champs : Vec<Champion>;

        let items : Vec<Item>;

        {

            let store\_read = store.read().await; //get readable store

            champs = store\_read.fetch\_champions().await?; //fetch champions and items for use in Board initialisation

            items = store\_read.fetch\_items().await?;

        }

        let mut store\_write = store.write().await; //get writable store

        store\_write.store\_board(&player\_one\_champs, &player\_two\_champs).await?; //store board in database

        return store\_write.set\_board(Board::new(&player\_one\_champs, &player\_two\_champs, &champs, &items, time\_unit, time\_till\_draw)); //set board to new board

    }

    Err(Error::StoreError)

}

///simulate x ticks of stored board

#[command]

pub async fn simulate\_x\_ticks(num\_ticks : Option<u32>, connection : AppHandle<Wry>) -> Result<()> {

    if let Ok(store) = get\_store\_read\_from\_state(connection) {

        let board\_opt = store.write().await.replace\_board(None)?; //fetch board

        if let Some(mut board) = board\_opt { //if board exists

            board.simulate\_battle(num\_ticks); //simulate battle

            store.write().await.replace\_board(Some(board))?; //swap back board

            return Ok(())

        }

        else {

            return Err(Error::FetchBoardError) //return fetch board error

        }

    }

    Err(Error::StoreError) //return fetch store error

}

#[command]

pub async fn fetch\_board(connection : AppHandle<Wry>) -> Result<Option<Board>> {

    if let Ok(store) = get\_store\_read\_from\_state(connection) {

        return store.read().await.fetch\_board() //return board

    }

    Err(Error::StoreError)

}

#[command]

pub async fn update\_outcome(outcome : u8, connection : AppHandle<Wry>) -> Result<()> {

    if let Ok(store) = get\_store\_read\_from\_state(connection) {

        return store.read().await.update\_outcome(outcome).await //take an outcome from the frontend and update the most recently saved board with said outcome

    }

    Err(Error::StoreError)

}

#[command]

pub async fn fetch\_outcomes(connection : AppHandle<Wry>) -> Result<Vec<(i64, String)>> {

    if let Ok(store) = get\_store\_read\_from\_state(connection) {

        return store.read().await.fetch\_outcomes().await; //fetch all results/ board

    }

    Err(Error::StoreError)

}

#[command]

pub async fn fetch\_outcome\_board(id : String, connection : AppHandle<Wry>) -> Result<Vec<PlacedChampion>> {

    if let Ok(store) = get\_store\_read\_from\_state(connection) {

        return store.read().await.fetch\_outcome\_board(id).await; //give the result id of a board, fetch the board stored in database

    }

    Err(Error::StoreError)

}

**Main.rs:**

#![cfg\_attr(

    all(not(debug\_assertions), target\_os = "windows"),

    windows\_subsystem = "windows"

)]

use std::{env, sync::Arc};

use tokio::sync::RwLock;

use crate::store::Store;

mod simulator;

mod store;

mod error;

mod prelude;

mod ipc;

use crate::ipc::{retrieve\_all\_items, retrieve\_all\_units, retrieve\_item\_from\_id, retrieve\_unit\_from\_id, retrieve\_all\_item\_ids, retrieve\_all\_unit\_ids, update\_unit, update\_item, submit\_board, fetch\_board, simulate\_x\_ticks, update\_outcome, fetch\_outcomes, fetch\_outcome\_board};

use crate::prelude::\*;

use crate::simulator::perform\_tests::perform\_test;

#[macro\_use]

extern crate log;

#[tokio::main]

async fn main() -> Result<()> {

    env::set\_var("RUST\_LOG", "error");

    env\_logger::init(); //setup logger

    info!("Program Start Up");

    let store = Store::new().await?; //create a new store

    if store.setup().await.is\_ok() { //if store setup ok

        let store = Arc::new(RwLock::new(store)); //create new Arc and RwLock of store for cross-thread mutability

        tauri::Builder::default() //call tauri builder to create app

            .manage(store)

            .invoke\_handler(tauri::generate\_handler![ //give commands

                retrieve\_all\_items,

                retrieve\_all\_units,

                retrieve\_item\_from\_id,

                retrieve\_unit\_from\_id,

                retrieve\_all\_item\_ids,

                retrieve\_all\_unit\_ids,

                update\_unit,

                update\_item,

                submit\_board,

                fetch\_board,

                simulate\_x\_ticks,

                update\_outcome,

                fetch\_outcomes,

                fetch\_outcome\_board

            ])

            .run(tauri::generate\_context!())

            .expect("error while running tauri application");

        Ok(())

    }

    else {

        Err(Error::DatabaseError("Failure to Start Up")) //return database error

    }

}

**Prelude.rs:**

/\* Prelude \*/

//default imports of Error and Result

pub use crate::error::Error;

pub type Result<T> = core::result::Result<T, Error>;

**store.rs:**

//import require types

use crate::prelude::\*;

use crate::simulator::board::Board;

use crate::simulator::champions::PlacedChampion;

use crate::simulator::{

    champions::Champion, champions::DEFAULT\_CHAMPIONS, item::Item, item::DEFAULT\_ITEMS,

};

use std::collections::BTreeMap;

use surrealdb::sql::{Object, Value};

use surrealdb::{Datastore, Response, Session};

use std::collections::VecDeque;

//import swap

use std::mem::swap;

///Holds connection to database, a board and the last board simulated

pub struct Store {

    ///database file

    ds: Datastore,

    ///session of database

    ses: Session,

    ///board currently being simulated

    board : Option<Board>,

    ///last board simulated

    last\_board : Option<String>

}

impl Store {

    ///Creates a new store

    pub async fn new() -> Result<Self> {

        let ds = Datastore::new("file://tft\_bot\_database").await.unwrap(); //opens or creates database file

        let ses = Session::for\_db("appns", "appdb"); //creates a new session

        Ok(Store { ds, ses, board : None, last\_board : None})

    }

    ///setups the board

    pub async fn setup(&self) -> Result<()> {

        //if there are no champions in the database

        if self.fetch\_champions\_ids().await?.is\_empty() {

            //insert default champions

            for champ in DEFAULT\_CHAMPIONS {

                match self.insert\_champion(&champ).await {

                    Ok(()) => info!("successfully inserted champ: {}", champ.id),

                    Err(e) => error!("error inserting champ: {}. {}", champ.id, e),

                }

            }

        }

        //if there are no items in the database

        if self.fetch\_items\_ids().await?.is\_empty() {

            //insert default items

            for item in DEFAULT\_ITEMS {

                match self.insert\_item(&item).await {

                    Ok(()) => info!("successfully inserted item: {}", item.id),

                    Err(e) => error!("error inserting item: {}. {}", item.id, e),

                }

            }

        }

        Ok(())

    }

    ///insert a champion into the database

    pub async fn insert\_champion(&self, champion: &Champion) -> Result<()> {

        //create the sql statement, creating a champion with id id and content data.

        let sql = format!("CREATE champions:{id} CONTENT $data", id = champion.id);

        //turn the champion into values and then store it in a BTreeMap

        let data: BTreeMap<String, Value> = champion.into\_values().into();

        let vars: BTreeMap<String, Value> = [("data".into(), data.into())].into();

        //execute on the database

        self.ds.execute(&sql, &self.ses, Some(vars), false).await?;

        Ok(())

    }

    ///insert an item into the database

    pub async fn insert\_item(&self, item: &Item) -> Result<()> {

        //create the sql statement

        let sql = format!("CREATE items:{id} CONTENT $data", id = item.id);

        //turn the item into values

        let data: BTreeMap<String, Value> = item.into\_values().into();

        let vars: BTreeMap<String, Value> = [("data".into(), data.into())].into();

        self.ds.execute(&sql, &self.ses, Some(vars), false).await?;

        Ok(())

    }

    ///fetch a list of all champions

    pub async fn fetch\_champions(&self) -> Result<Vec<Champion>> {

        //create the sql statement

        let sql = "SELECT \* FROM champions";

        //get the ress

        let ress = self.ds.execute(sql, &self.ses, None, false).await?;

        //turn it into an iterator of object, and for each object try to create a champion from it, then collect the iterator into a vector

        Ok(into\_iter\_objects(ress)?

            .map(|f| Champion::try\_from(f.unwrap()).unwrap())

            .collect())

    }

    ///fetch a champion from an id

    pub async fn fetch\_champion\_from\_id(&self, id: u8) -> Result<Option<Champion>> {

        let sql = &format!("SELECT \* FROM champions:{id}");

        let ress = self.ds.execute(sql, &self.ses, None, false).await?;

        //if there is an object in the result vector, try to create a champion from the object

        if let Some(obj) = into\_iter\_objects(ress)?.next() {

            return Ok(Some(Champion::try\_from(obj?)?))

        }

        Ok(None)

    }

    ///fetch an item from the database by id

    pub async fn fetch\_item\_from\_id(&self, id: u8) -> Result<Option<Item>> {

        let sql = &format!("SELECT \* FROM items:{id}");

        let ress = self.ds.execute(sql, &self.ses, None, false).await?;

        //if there is an object in the result vector, try to create an item from the object

        if let Some(obj) = into\_iter\_objects(ress)?.next() {

            return Ok(Some(Item::try\_from(obj?)?))

        }

        Ok(None)

    }

    ///fetch a vector of all the champion ids

    pub async fn fetch\_champions\_ids(&self) -> Result<Vec<u8>> {

        let sql = "SELECT id FROM champions";

        //execute the statement, turn the result into a vector of objects, and for each one fetch the id and turn it onto a u8

        Ok(

            into\_iter\_objects(self.ds.execute(sql, &self.ses, None, false).await?)?

                .map(|f| {

                    fetch\_id(f.unwrap()).as\_int() as u8

                })

                .collect(),

        )

    }

    ///fetch a vector of all the item ids

    pub async fn fetch\_items\_ids(&self) -> Result<Vec<u8>> {

        let sql = "SELECT id FROM items";

        //execute the statement, turn the result into a vector of objects, and for each one fetch the id and turn it onto a u8

        Ok(

            into\_iter\_objects(self.ds.execute(sql, &self.ses, None, false).await?)?

                .map(|f| {

                    fetch\_id(f.unwrap()).as\_int() as u8

                })

                .collect(),

        )

    }

    ///fetch a vector of all items

    pub async fn fetch\_items(&self) -> Result<Vec<Item>> {

        let sql = "SELECT \* FROM items";

        let ress = self.ds.execute(sql, &self.ses, None, false).await?;

        //println!("{ress:?}");

        Ok(into\_iter\_objects(ress)?

            .map(|f| Item::try\_from(f.unwrap()).unwrap())

            .collect())

    }

    ///updates a champion's value

    pub async fn update\_champion(&self, champion : Champion) -> Result<()> {

        //create sql, update champ with id : id

        let sql = format!("UPDATE champions:{id} CONTENT $data", id = champion.id);

        //turn champion into values

        let data: BTreeMap<String, Value> = champion.into\_values().into();

        let vars: BTreeMap<String, Value> = [("data".into(), data.into())].into();

        //execute statement

        self.ds.execute(&sql, &self.ses, Some(vars), false).await?;

        Ok(())

    }

    ///updates an item's values

    pub async fn update\_item(&self, item : Item) -> Result<()> {

        let sql = format!("UPDATE items:{id} CONTENT $data", id = item.id);

        let data: BTreeMap<String, Value> = item.into\_values().into();

        let vars: BTreeMap<String, Value> = [("data".into(), data.into())].into();

        self.ds.execute(&sql, &self.ses, Some(vars), false).await?;

        Ok(())

    }

    ///takes in a board as input and sets the self.board field to it

    pub fn set\_board(&mut self, board : Board) -> Result<()> {

        self.board = Some(board);

        Ok(())

    }

    ///replace the self.board value with the given value, returning the old value

    pub fn replace\_board(&mut self, mut board : Option<Board>) -> Result<Option<Board>> {

        swap(&mut self.board, &mut board);

        Ok(board)

    }

    ///clones the current board and returns it

    pub fn fetch\_board(&self) -> Result<Option<Board>> {

        Ok(self.board.as\_ref().cloned())

    }

    ///stores a board with given placed champions

    pub async fn store\_board(&mut self, p1\_champs : &VecDeque<PlacedChampion>, p2\_champs : &VecDeque<PlacedChampion>) -> Result<()> {

        //create board sql with unknown outcome

        let sql = "CREATE boards SET outcome = 0";

        let ress = self.ds.execute(sql, &self.ses, None, false).await?;

        //fetch id of new field created

        let id = fetch\_id(into\_iter\_objects(ress)?.next().unwrap()?).as\_string();

        //create sql

        let sql = "CREATE boards\_champ CONTENT $data";

        //create link to board field

        let board\_link = format!("boards:{id}");

        //for each champ in p1\_champs

        for champ in p1\_champs {

            //turn champ into values

            let mut data: BTreeMap<String, Value> = champ.into\_values().into();

            //insert into data link to board value

            data.insert("board".into(), board\_link.clone().into());

            //insert into data team value

            data.insert("team".into(), 1.into());

            //create vars

            let vars: BTreeMap<String, Value> = [("data".into(), data.into())].into();

            //execute statement

            self.ds.execute(sql, &self.ses, Some(vars), false).await?;

        }

        for champ in p2\_champs {

            //repeat for player 2 champs

            let mut data: BTreeMap<String, Value> = champ.into\_values().into();

            data.insert("board".into(), board\_link.clone().into());

            data.insert("team".into(), 2.into());

            let vars: BTreeMap<String, Value> = [("data".into(), data.into())].into();

            self.ds.execute(sql, &self.ses, Some(vars), false).await?;

        }

        //set last board to this board id

        self.last\_board = Some(id);

        Ok(())

    }

    ///updates the outcome of the last board

    pub async fn update\_outcome(&self, outcome : u8) -> Result<()> {

        if self.last\_board.is\_some() { //if there is last board

            let last\_board = self.last\_board.clone().unwrap(); //clone the string

            let sql = &format!("UPDATE boards:{last\_board} SET outcome = {outcome}"); //update the board outcome in the database with the new outcome

            self.ds.execute(sql, &self.ses, None, false).await?;

            return Ok(())

        }

        Err(Error::LastBoardError) //return last board error

    }

    ///returns a vector of a outcome, board ID pair

    pub async fn fetch\_outcomes(&self) -> Result<Vec<(i64, String)>> {

        let sql = "SELECT \* FROM boards"; //select all from boards

        let ress = self.ds.execute(sql, &self.ses, None, false).await?;

        //execute statement, get result

        //turn ress into iterator of objects, map the objects to an outcome, id pair and return

        Ok(into\_iter\_objects(ress)?.map(|obj| {

            let mut obj = obj.unwrap();

            (obj.remove("outcome").unwrap().as\_int(), fetch\_id(obj).as\_string())

        }).collect())

    }

    ///fetch the board state of a board with id : ID

    pub async fn fetch\_outcome\_board(&self, id : String) -> Result<Vec<PlacedChampion>> {

        //fetch all champs from boards\_champ with board id id

        let sql = &format!("SELECT \* FROM boards\_champ WHERE board = boards:{id}");

        let ress = self.ds.execute(sql, &self.ses, None, false).await?;

        //map result iterator into a vector of placedChampions

        Ok(into\_iter\_objects(ress)?.map(|f| PlacedChampion::try\_from(f.unwrap()).unwrap()).collect())

    }

}

///small utility piece of code to fetch the id of the first result from a response

///<br /> fairly obtuse piece of code enclosed in a function to avoid redundant repeated code

fn fetch\_id(mut obj: Object) -> Value {

    Value::from(obj.remove("id").unwrap().record().unwrap().id)

}

///code taken from: https://www.youtube.com/watch?v=iOyvum0D3LM

fn into\_iter\_objects(ress: Vec<Response>) -> Result<impl Iterator<Item = Result<Object>>> {

    let res = ress

        .into\_iter() //turns ress into an iterator of responses

        .next() //gets the first response (as in this project, I only make one request per statement, so there will next be any other responses)

        .map(|rp: Response| rp.result) //get the result of the response

        .transpose()?; //swap Option<Result> into Result<Option>

    match res {

        Some(Value::Array(arr)) => { //if res is an array of responses

            let it = arr.into\_iter().map(|v| match v {

                Value::Object(object) => Ok(object), //map each value into an object

                \_ => Err(Error::DatabaseError("A record was not an object")), //return error if invalid

            });

            Ok(it) //return iterator

        }

        \_ => Err(Error::DatabaseError("No records found")), //return database error

    }

}