Simulated Concurrency Programming Game

Contents

[Sprint Backlog: 2](#_Toc128749864)

[Sprint 1: Thursday 02/02/2023 – 09/02/2023 2](#_Toc128749865)

[User Requirements: 2](#_Toc128749866)

[Sprint 1 Design Choices: 2](#_Toc128749867)

[Low Fidelity Prototypes: 2](#_Toc128749868)

[Evidence of Progress: 3](#_Toc128749869)

[Sprint Summary: 5](#_Toc128749870)

[Plan for Sprint 2: 5](#_Toc128749871)

[Sprint 2: Friday 10/02/2023 - Friday 24/02/2023 6](#_Toc128749872)

[User Requirements: 6](#_Toc128749873)

[Sprint 2 Design choices: 6](#_Toc128749874)

[State Diagram: 7](#_Toc128749875)

[Class Diagram: 8](#_Toc128749876)

[Forecast implementation: 9](#_Toc128749877)

[Sprint 2 Evidence: 10](#_Toc128749878)

[Sprint Summary: 11](#_Toc128749879)

[Plan for Sprint 3: 11](#_Toc128749880)

# Sprint Backlog:

## Sprint 1: Thursday 02/02/2023 – 09/02/2023

As the first sprint of the project, one of the main focuses is getting the project set up. This involves creating a git repository, setting up a unity project and collecting some of the required packages, such as Antlr4. Another key focus for this sprint is getting the core functionality of creating, editing, and deleting scripts.

### User Requirements:

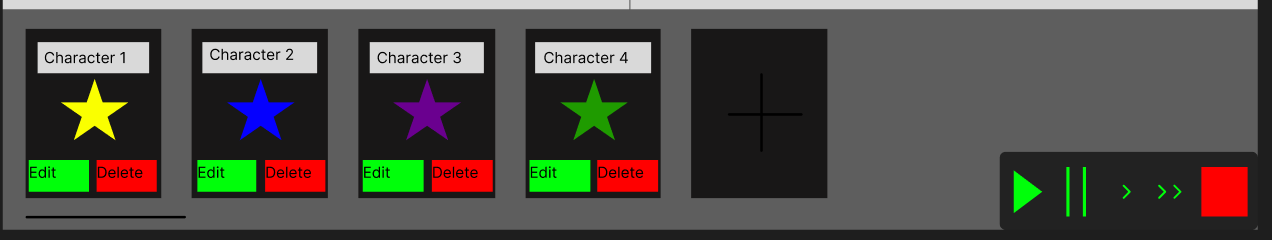
|  |  |
| --- | --- |
| **Story ID:** | **Description:** |
| 15 | As a player, I want to be able to access all options through a main menu and campaign map so that I can navigate to all features of the game |
| 1 | As a player I want to be able to create a new character script so that I can define the behaviour of the players on my team |
| 2 | As a player I want to be able to Edit/ Delete scripts so I can change that behaviour to optimise my team’s chance of victory |
| 7 | As a player, I want to be able to save my scripts so that they carry over between levels, and persist when I exit the game and restart |

Story 15 does include the concept of the campaign map, for this sprint, the menu system will be simplified to only include a sample play scene for development, a settings button (to be fleshed out later), and a ‘quit game’ option.

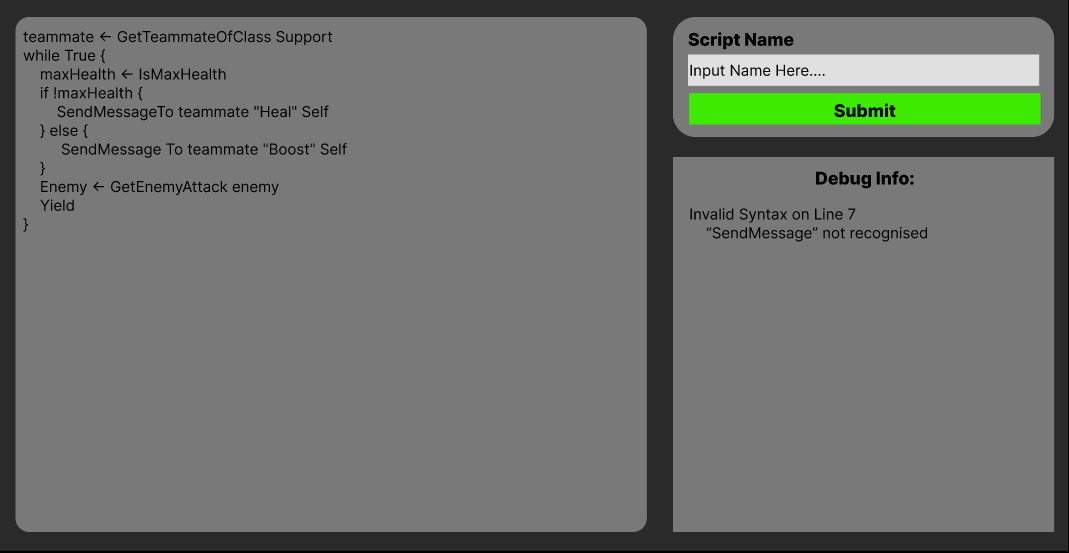
### Sprint 1 Design Choices:

Since this sprint I will be implementing the lexer and parser for the language, I have already made some design choices from the original specification. The first of which is to do with iteration. In the early development stage of the language, iteration was designed to work in 2 ways; conditional loops and non-conditional loops (e.g., loop 5 times). I have decided to remove the functionality for non-conditional loops and only have ‘while <condition>’ as a feature. This is for two reasons, firstly, non-conditional loops don’t add any expressive power to the language, and only serve to add more choices to the user. By removing them, I will make the language simpler, and therefore more user-friendly for amateur programmers, which is the main goal of the game. Secondly, for complexity reasons. The language is already surprisingly complex for a small language and removing this feature now will serve to reduce my workload in later sprints.

### Low Fidelity Prototypes:



Referring back to the low fidelity prototype presented in the progress report, the aim of this sprint is to have a functioning version of the character management UI. This includes a box for each of the characters, as well as an Edit and Delete button for each one. By clicking on the ‘+’ button at the end of the list, a new box should appear allowing the user to choose from either an existing script, or to create a new one.



Here is an example prototype for the IDE. It uses a very simple design, with a textbox for entering the code, another for entering the name of the script (which is also the name of the character on the previous screen, and a box for potential debug info. The scope of debugging information is still to be decided. The submit button is used to save and close the script editor. When the button is clicked, the script is passed through a lexer and parser, if it has any errors, then the script will not be saved, and an error will appear in the debug box. Otherwise, the script will save, and the IDE will close.

### Evidence of Progress:

This is the early prototype of the main Menu:



For enhanced user feedback, each of the buttons changes colour when highlighted and clicked. The settings page is not currently implemented as that isn’t one of the core features of the game. Both the Campaign and Skirmish buttons currently lead to the same testing development scene as no other scenes have been implemented yet. Furthermore, the title and background are subject to change, as they are only placeholders for now.



This is a screenshot of the character control bar. It allows for multiple characters, some repetitions of each other. You can also click to add a new character, after which you can select from a list of existing scripts, or create a new one, opening up the IDE. The white boxes will later have images of the characters, so you can see which is which in the battle scene above.

This is the IDE in its current form. There is a section for writing code, a place to enter the name of the script and some debug info. There are various feedback systems such as the button flashing red to show an error, or debug info appearing in the box on the right.

### Sprint Summary:

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  | | --- | | Sprint 1 Backlog: | | Setup Repo and Unity | | Install and configure Antlr4 | | Create Main Menu | | Create Script Editor | | Create Script | | Save Script | | Delete Script | | Edit Script | | Create Language Grammar | |  |

Here is the burndown chart for the first sprint. For the first few days, progress was slow, this is mostly due to the difficulty of installing and configuring Antlr4 for use with Unity, this task alone took almost 3 days and was pivotal for moving forward with the project. Once this was out of the way, the other tasks fell into place quite quickly, saving and loading files was the other main challenge to overcome, as I did not have previous experience with file IO in C#. All tasks were completed ahead of schedule, meaning the last few days could be spent cleaning up code and working on extra features not neces sarily mentioned in the spec, for example, setting up Audio functionality for later on.

### Plan for Sprint 2:

Referring back to the Gannt chart in the progress report, my next steps in development are to develop the scheduling algorithm and execution model. This is quite a large task, so I’m going to allocate 2 weeks to the next sprint. Below is the sprint 2 backlog and burndown chart setup.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  | | --- | | Sprint 2 Backlog: | | Design and create thread scheduling algorithm | | Connect Lexer and Parser to IDE | | Build AST from scripts on play | | Set character class | | Create execution model | | Integrate interpreter into scripts | |  |

## Sprint 2: Friday 10/02/2023 - Friday 24/02/2023

### User Requirements:

|  |  |
| --- | --- |
| **Story ID:** | **Description:** |
| 2 (cont.) | As a player I want to be able to Edit/ Delete scripts so I can change that behaviour to optimise my team’s chance of victory |
| 3 | As a player I want to be able to choose the class that each character has which will provide certain buffs or de-buffs to their abilities in game |

### Sprint 2 Design choices:

The main design choices I have come across with this sprint is the cost with which to assign each instruction. For now, I have decided to use dummy values, as it’s difficult to assess how effective each cost is when most gameplay is not actually implemented yet. This was discussed in the progress report, and I stated that these values are iterative and will likely change many times over the development cycle. The next decision is a more major one, and it's to do with 2 unique interactions which can occur with thread scheduling.

#### Thread scheduler discussion:

Most real world schedulers are deterministic at a low level, however, due to the chaotic nature of concurrency and multithreading with so many programs on one machine, they have a chaotic output which can appear non-deterministic. I want my scheduler to represent this to the user to help them understand the unpredictable nature of multithreading. To achieve this, I am going to introduce a random element into the scheduler, so that levels don’t always play exactly the same way.

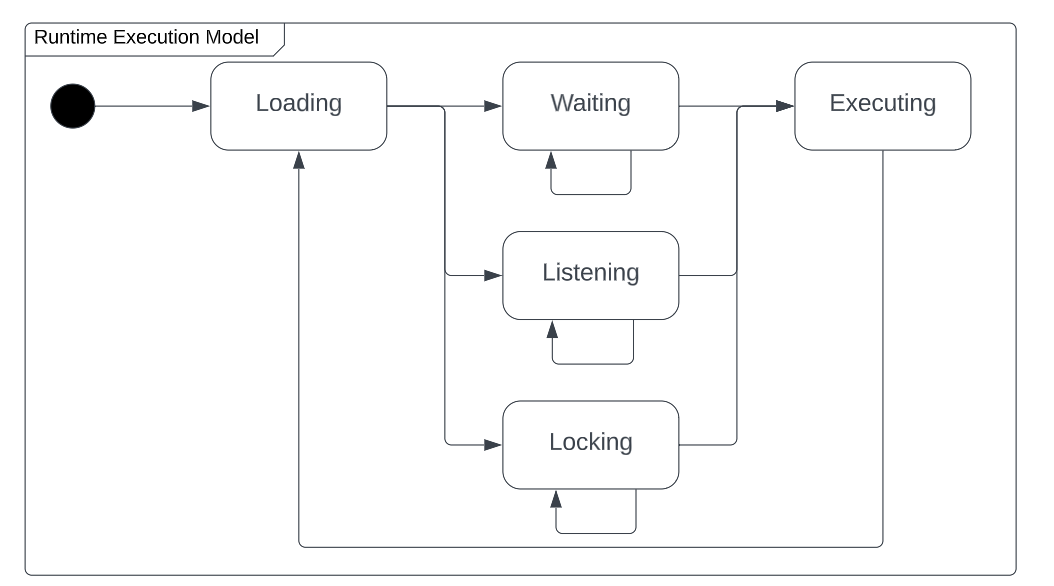
The other non-traditional aspect of my thread scheduler is the fact that it shows a forecast of upcoming threads. This present some design issues: how accurate should the forecast be? And what should happen if a thread yields or dies when it’s already been queued for execution. In real world applications, no forecast is created, so this isn’t an issue. For my application, the forecast is an essential UI element which provides helpful information and helps the user understand the concept of multithreading.

The first area I want to discuss is what elements of randomness I implement. Firstly, I have decided to introduce randomness in the priority queue. All threads have an associated priority, rather than the scheduler always picking the highest one (this would result in all threads taking turns in the same order every time as there is only going to be a maximum of around 10 characters at once), the scheduler will instead choose a thread randomly, but weighted by its priority. Below is a demonstrated example. I am also going to randomly decide how long a thread is able to execute for. This may change later during the balancing phase of the game, but again this is a mechanism designed to teach the user about the chaotic and non-deterministic nature of multithreading. For example, if the user knows that a thread will always be able to execute for 5 time steps, they will be able to optimise their scripts with this in mind, which is a playstyle I want to avoid as programmers in most cases aren’t able to do this in real-life.

|  |  |  |
| --- | --- | --- |
| **Character ID** | **Priority** | **Chance of being picked** |
| C1 | 0 | 0 |
| C2 | 5 | 5/22 |
| C3 | 8 | 8/22 |
| C4 | 9 | 9/22 |
| **Total** | 22 | 22/22 |

The second area I want to discuss is the forecast. In this area I have 2 possible implementations. Firstly, when a character dies or yields, the forecast doesn’t change, that character is just removed from the schedule and the core will idle during the empty space left behind. This is no representative of a real-world scheduler as it reduces overall efficiency. The alternative is that I reschedule when an event occurs which disrupts the forecast. The drawback this has is that the user may not trust the forecast if it reshuffles too often, and it may not be clearly explained why it suddenly changed. However, if implemented well, it could be used to teach the player about the non-deterministic nature and how it’s impossible to give a truly accurate prediction of a program’s exact execution.

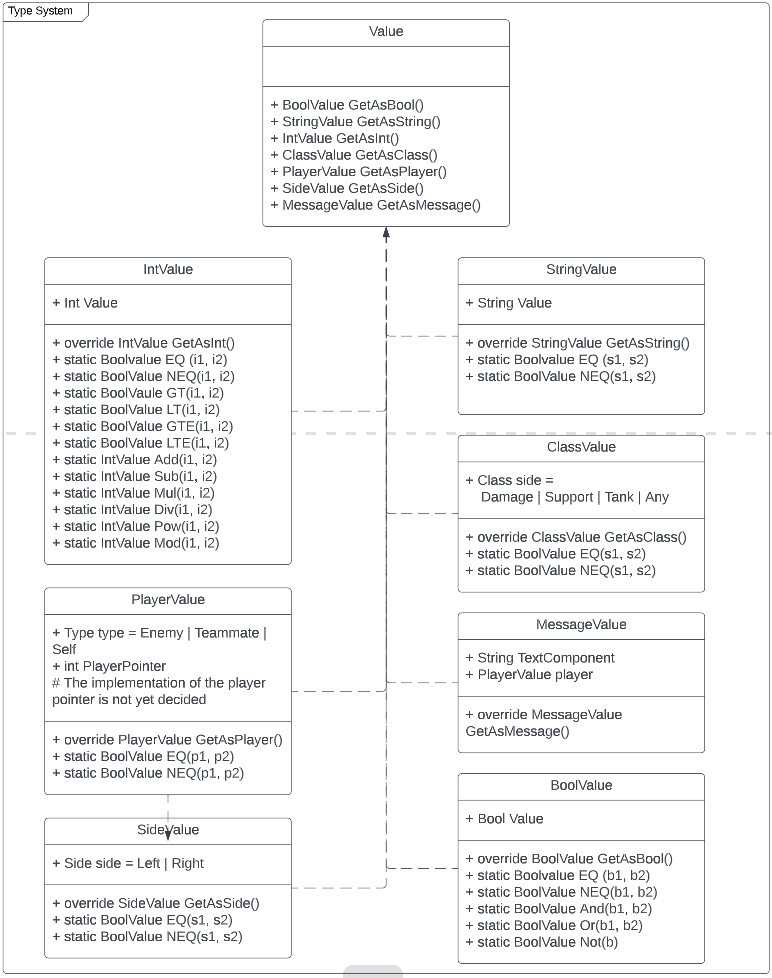
### State Diagram:



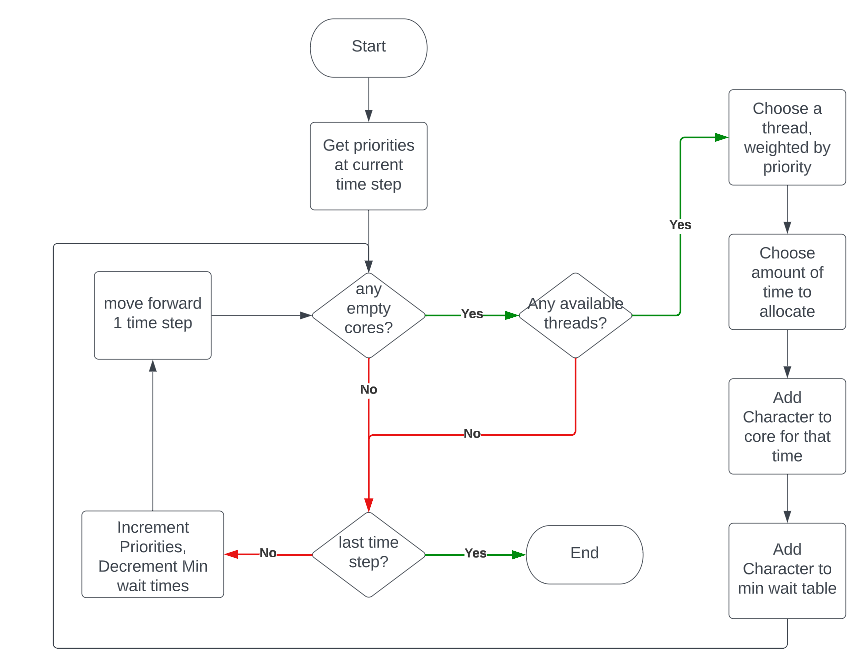
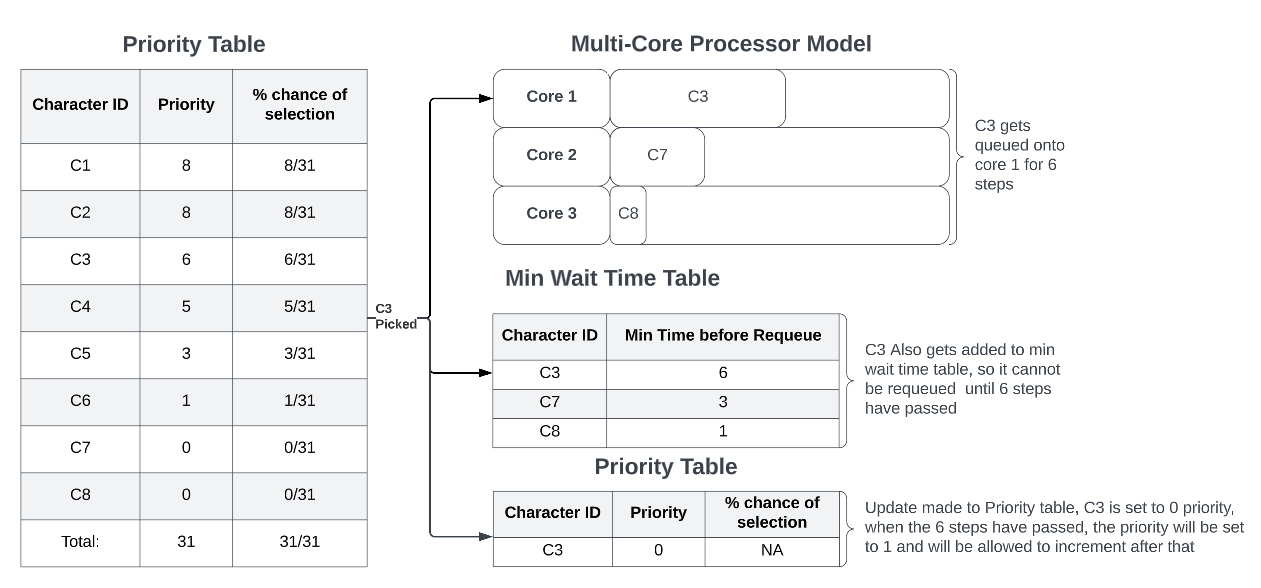
This is a finite state machine representing the runtime execution model of the language. During runtime, the machine must be able to advance at discrete time steps. On each time step, the execution model can take one action, and advance to the next state. The first state is loading the instruction. This involves calculating how many time steps it should wait during the next state, figuring out which action should be taken in the execution state, and updating the instruction stack tree to assist in the next loading state.

The next 3 states are either waiting, listening or locking. Waiting simply counts a number of time steps before moving to the executing state. Listening will wait a minimum amount of time, and then start checking the incoming messages buffer queue for new messages, once a message is found, then it will move to the executing state. Locking will also wait a minimum number of time steps, and then start trying to lock a given resource, not achieving anything until its free. Finally, Execution will perform the actual logic steps required to evaluate an instruction or take an action.

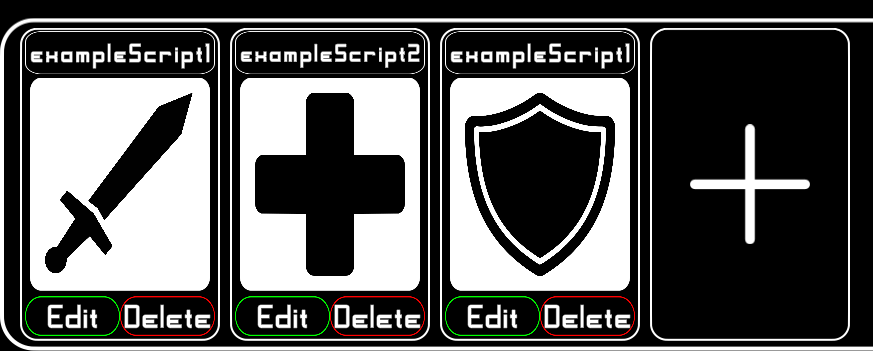
### Class Diagram:



### Forecast implementation:



### Sprint 2 Evidence:



This is now what the control bar looks like. The blank images have been replaced with images representing the characters class. Clicking on the icon will cycle through the 3 classes. A script isn’t tied to a specific class, allowing the user to write generic scripts which match any script. However, if the user makes a script for a support, but then assigns it the tank class, it will fail any support specific actions like healing teammates.

#### Thread Testing:

By running the following script, I can test the scheduler’s ability to queue multiple characters in multiple cores:

|  |
| --- |
| static void Main(string[] args)  {  // Character( int team, int num)  Character A1 = new Character(1, 1);  Character B1 = new Character(2, 1);  Character A2 = new Character(1, 2);  Character B2 = new Character(2, 2);  Character[] chars = new Character[4] { A1, B1, A2, B2 };  ThreadScheduler scheduler = new ThreadScheduler(  3, // Number of cores  2, // Minimum Time to be Queued for  5, // Maximum Time to be Queued for  10, // Forecast size  1, // Yield Boost (priority boost received when teammate yields)  1, // Passive Priority Build Rate  chars // Characters to queue  );  Console.WriteLine(scheduler);  Console.ReadLine();  } |
| Thread Scheduler:  Current Time : 0  Forecast Size : 10  NumCores : 3  Cores:  Core: , A2, A2, A2, A2, B1, B1, A2, A2, A2, A2, A1, A1, A1, A1  Core: , A1, A1, A1, A1, B2, B2, B2, B2, B2, B2, A2, A2  Core: , B2, B2, B1, B1, A1, A1, A1, A1, B1, B1, B1, B1 |

In this example, ‘A’ represents team 1, ‘B’ represents team 2. The scheduler plans for 10 steps ahead of the current time (which in this case is 0 as we have not advanced time at all). You’ll notice that the schedule actually extends beyond 10 steps, this is caused when a character is queued for multiple steps. So on the 10th step, A1 is scheduled for 4 steps, which means the forecast actually goes on to the 14th step. This is intentional, as these steps will simply extend beyond the visual range of the forecast visualiser. The 10 steps set are just the minimum required to make the visualiser doesn’t have empty space at the end.

### Sprint Summary:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  | | --- | | Sprint 2 Backlog: | | Design and create thread scheduling algorithm | | Connect Lexer and Parser to IDE | | Build AST from scripts on play | | Set character class | | Create execution model | | Integrate interpreter into scripts | |  |

As you can see, I overestimated this sprint quite heavily. I was ahead of schedule from the very start. However, several of these tasks were relatively small ones. Connecting the lexer and parser didn’t take long, neither did setting a characters class. This meant that the start of the sprint went at a rapid pace as I was focusing on these minor tasks. The major task was designing and implementing the thread scheduling algorithm. You can see where I developed this component on the burndown chart as it plateaus for several days.

### Plan for Sprint 3:

Following the same plan as the Gannt chart presented in the progress report, sprint 3 will be focussed on the scheduler representation, playing pausing and stopping the simulation. After this sprint is complete, I will be able to watch battles play out, albeit with dummy instructions for now. The main tasks for sprint 3 are to create the characters on the battle scene, play the battle and step through instructions. There should also be a representation of the queue at the top of the screen.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  | | --- | | Sprint 3 Backlog: | | Create Character objects in game | | Play simulation | | Pause, Stop, and Step Simulation | | Instruction Cost Visualisation | | Scheduler Representation | | Pause Menu | |  |

## Sprint 3:

