TÉCNICO LISBOA

My Army: Strategy Game Engine

Alexandre Pedro Gomes Freitas

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisor: Prof. Pedro Alexandre Simões dos Santos

Examination Committee

Chairperson: Prof. Paolo Romano
Supervisor: Prof. Pedro Alexandre Simões dos Santos
Member of the Committee: Prof. António Manuel Ferreira Rito da Silva

November 2015

Acknowledgments

I would like to thank everyone that participated in this project. Special thanks to Pedro Santos, for his suggestions, advices and feedback which were crucial for this thesis, to Luís Ribeiro and João Morais, for their technical support and help during the development stage and finally, to my family, for their love and moral support.

Resumo

O presente documento aborda todo o processo necessário para se refazer o atual motor de jogo do "*My Army*", para isso foi efetuada uma análise detalhada de forma a se verificar o estado do mesmo, o funcionamento do respetivo simulador de batalhas, bem como uma análise entre a linguagem do atual simulador e a nova linguagem selecionada para se compreender a capacidade de cada uma destas solucionar os problemas existentes. De seguida, apresenta-se uma possível solução para se resolverem estes problemas, tendo em conta toda a verificação anteriormente efetuada.

Após várias semanas de desenvolvimento e de correção de problemas, a nova versão do simulador ficou concluída, porém ainda foi alvo de uma nova análise para se inserir paralelismo de modo a melhorar o seu desempenho.

Após várias tentativas, chegou-se à solução paralela que oferecia maior segurança e com uma melhoria significativa no desempenho.

Por fim, todas as versões do simulador de batalhas foram testadas utilizando-se vários exemplos de batalhas retirados do jogo para se realizar uma análise comparativa e assim demonstrar-se o desempenho de cada uma delas.

Palavras-chave: motor de jogo, batalhas, problemas de desempenho, simulador, agentes

Abstract

This document explains all the process used to remake the game engine of the "*My Army*" game. The process starts with a detailed review to the game engine to understand how it works and it was also made a review between the current language and the new language to understand the capacity of each language to solve the existing issues. Then, a solution was presented to solve the existing issues, according to the previous reviews.

After weeks of development and solving issues, a new version of the simulator was complete. The next step was to analyze this version to insert parallelism to improve the simulator performance.

After several attempts, a parallel solution was reached which offered a better performance and consistent battle outputs.

Finally, all the battle simulator versions were tested using different battle examples from the game in order to make a comparative analysis showing the performance from each version.

Keywords: game engine, battles, performance issues, simulator, agents

Contents

Acknowledgments . . . iii
Resumo . . . v
Abstract . . . vii
List of Figures . . . xi
List of Abbreviations . . . xiii

1 Introduction **1**
1.1 Motivation . . . 1
1.2 Objectives . . . 2
1.3 Contribution . . . 2
1.4 Outline . . . 2

2 Related Work **3**
2.1 Game Engine Analysis . . . 3
2.1.1 Preparation & Positioning . . . 4
2.1.2 Battle Processing . . . 5
2.1.3 Quadtree . . . 7
2.1.4 Logging Battle Results . . . 9
2.1.5 Game Engine Issues . . . 9
2.2 Programming Languages Analysis . . . 10
2.2.1 JavaScript . . . 10
2.2.2 Go . . . 10
2.2.3 Differences Between Languages . . . 11
2.2.4 Parallelism *vs* Concurrency . . . 14

3 Algorithm Analysis and Proposed Solution **19**
3.1 Profiling . . . 19
3.2 Game Architecture . . . 21
3.3 Parallelism . . . 22
3.4 Conclusions . . . 23

4 Development **25**
4.1 Issues . . . 30
4.2 Differences between versions . . . 31
4.3 Parallelism Implementation . . . 35

5 Tests **37**
5.1 Battle Tests . . . 37
5.1.1 Controlled Battle Tests . . . 38
5.1.2 Random Battle Tests . . . 40
5.2 Tests Results . . . 43
5.2.1 Controled Battle Tests Results . . . 43
5.2.2 Random Battle Tests Results . . . 44
5.3 Go Profiling . . . 47
5.4 Discussion . . . 48

6 Conclusions and Future Work **49**

Bibliography **51**

List of Figures

2.1 Different components of tactics [18, p. 30]. 4
2.2 Example of a tactic with two lines of combat. [18, p. 31]. 5
2.3 *Battle Viewer* [18, p. 50]. 6
2.4 Division of the battlefield by four times. 8
2.5 Battle Contingent (BC)'s vertexes placement with quadtree division. 8
2.6 Quadtree with four levels. 9

3.1 Performance profile from the JavaScript battle simulator. 19
3.2 Current game software architecture [18, p. 45]. 21
3.3 Illustration of the problem found in movement action. 23

4.1 The image on the top shows the positioning of all BCs in the battlefield generated by JavaScript while the image on the bottom shows the positioning of all BCs in the battlefield generated by Go. 32
4.2 The image on the left shows the last step of the battle in Go while the image on the right shows the last step of the battle in JavaScript. 34

5.1 This battle starts with 8 BCs on the blue side and 8 BCs on the red side making a total of 16 BCs. 38
5.2 This battle starts with 16 BCs on the blue side and 16 BCs on the red side making a total of 32 BCs. 38
5.3 This battle starts with 32 BCs on the blue side and 32 BCs on the red side making a total of 64 BCs. 39
5.4 This battle starts with 64 BCs on the blue side and 64 BCs on the red side making a total of 128 BCs. 39
5.5 This battle starts with 120 BCs on the blue side and 120 BCs on the red side making a total of 240 BCs. 39
5.6 Both images belong to the battle with id 2507. This battle starts with 221 BCs on the blue side and 51 BCs on the red side making a total of 272 BCs. 40
5.7 Both images belong to the battle with id 7696. This battle starts with 7 BCs on the blue side and 10 BCs on the red side making a total of 17 BCs. 40

5.8 Both images belong to the battle with id 7697. This battle starts with 4 BCs on the blue side and 2 BCs on the red side making a total of 6 BCs. 40
5.9 Both images belong to the battle with id 7698. This battle starts with 48 BCs on the blue side and 27 BCs on the red side making a total of 72 BCs. 41
5.10 Both images belong to the battle with id 7699. This battle starts with 49 BCs on the blue side and 320 BCs on the red side making a total of 369 BCs. 41
5.11 Both images belong to the battle with id 7700. This battle starts with 49 BCs on the blue side and 53 BCs on the red side making a total of 102 BCs. 41
5.12 Both images belong to the battle with id 7701. This battle starts with 33 BCs on the blue side and 15 BCs on the red side making a total of 48 BCs. 42
5.13 Both images belong to the battle with id 7702. This battle starts with 49 BCs on the blue side and 15 BCs on the red side making a total of 64 BCs. 42
5.14 Graphic showing the performance from Go and Go-Parallel versions for the same battle but with different numbers of BCs. 43
5.15 Graphic showing the average time spent, in seconds, from each battle step in Go and Go-Parallel versions using the same battle tests as above. 44
5.16 Graphics showing the performance from JavaScript, Go and Go-Parallel versions in short and long duration battles. 45
5.17 Graphic showing the average time spent, in seconds, from each battle step in JavaScript, Go and Go-Parallel versions using the same battle tests as above. 46
5.18 Performance profile from the Go battle simulator. 47

List of Abbreviations

BC Battle Contingent

Chapter 1

Introduction

1.1 Motivation

Every game created till today started as a simple idea and to turn this simple idea into reality is not as easy as it sounds because it requires to go through a long and time-consuming process to get the right resources and also to refine as much as possible this idea. Following the desire to have his own game, there was a person who had the idea to create a new strategy game, whose name was “My Army”, in which the players could own and manage an army somewhat similar to what a general would do in real life.

Since this person could not manage to create this game by himself he gathered a couple of people to help him out. With the right people gathered, the first step was to study the market and possible competitors out there allowing to select the best platform for this game and also to get as much visibility as possible.

Eventually, it was decided to make this game available to social networks and to develop a prototype using JavaScript. With the prototype developed, a bunch of tests were made in order to find and solve potential issues.

Everything looked fine and the next obvious step was to open the game to the public, but it did not go that far because the game was not prepared yet since it was found a performance issue in one of the modules. This module was an agent-based battle simulator responsible for the battle management where each battle can have a large number of agents. Beyond the JavaScript version there was also an incomplete version developed in Go.

The main challenges of this thesis were to study the existing version of this module, to complete the unfinished version developed in Go and to solve the performance issue by using a new high perfor-

mance programming language and using a parallelism approach.

1.2 Objectives

The goals of this thesis are:

- Study the original version of the battle simulator
- Improve the performance of the battle simulator through code optimization and parallelism using a high performance language to make the battle processing faster, preferably less than one minute;
- Run different tests in order to create a performance comparison between all the versions of the battle simulator.

1.3 Contribution

The contribution of this thesis is:

- A single-threaded version and a multi-threaded version of the battle simulator, both developed in a new programming language and with better performance than the previous version. Both versions with code optimizations to solve issues that were present in the previous version;

1.4 Outline

This document is divided in six chapters. Beyond this chapter, there are the following chapters:

- The Related Work chapter explains all the details about the functionality of the simulator and the existing problems with it and it will also contain a language analysis between JavaScript and Go;
- The Algorithm Analysis and Proposed Solution chapter describes the solution found to solve the existing issues and the final architecture that is going to be used;
- The Development chapter describes the development process of the new version of the simulator including some issues that occured during this process;
- The Tests chapter contains all the steps taken to evaluate the performance of each version of the battle simulator and will also contain a comparative analysis between these versions;
- Finally, the Conclusions chapter discusses about the final results along with some final thoughts.

Chapter 2

Related Work

2.1 Game Engine Analysis

“My Army” allows the players to be on a role of a general giving them the authority to own and manage an army to battle other players. The army management is done by recruting troops with different characteristics, purchasing different equipments to improve the soldiers’ performance and creating several tactics in order to achieve the game’s main goal, which is to win as many battles as possible to reach the top of the ladder.

There were two versions of this module, the first version was developed in JavaScript, being the one focused in this chapter, and it was complete and fully functional while the second version was developed in Golang, also known as Go, and it was incomplete and not functional.

The main focus of this thesis was the module dedicated to the battle simulation, which was developed in JavaScript and managed all the elements from a battle between two generals. When a general challenged another one, this module received all the information about the two generals and generated a battle composed with several agents.

The process of the battle creation by this module is divided by three stages:

1. Preparation & Positioning;
2. Battle Processing;
3. Logging Battle Results.

2.1.1 Preparation & Positioning

In this stage, the simulator creates a rectangular battlefield where each side is assigned to a general to put his army in the battlefield according to his tactics.

The tactic created by a general sets the army formation and the army behaviour, both offensive and defensive, in the battlefield. The army formation can have one or more lines of combat where each represents the alignment of a group of soldiers at the beginning of each battle. [18, p. 31].

There are three different sectors in a battlefield: a central sector, a left flank sector and a right flank sector. These three sectors have a set of rules that may vary from sector to sector. [18, p. 31].

Each sector is divided in slots where each of these contains a category and a relative power (Strong, Average, Weak).

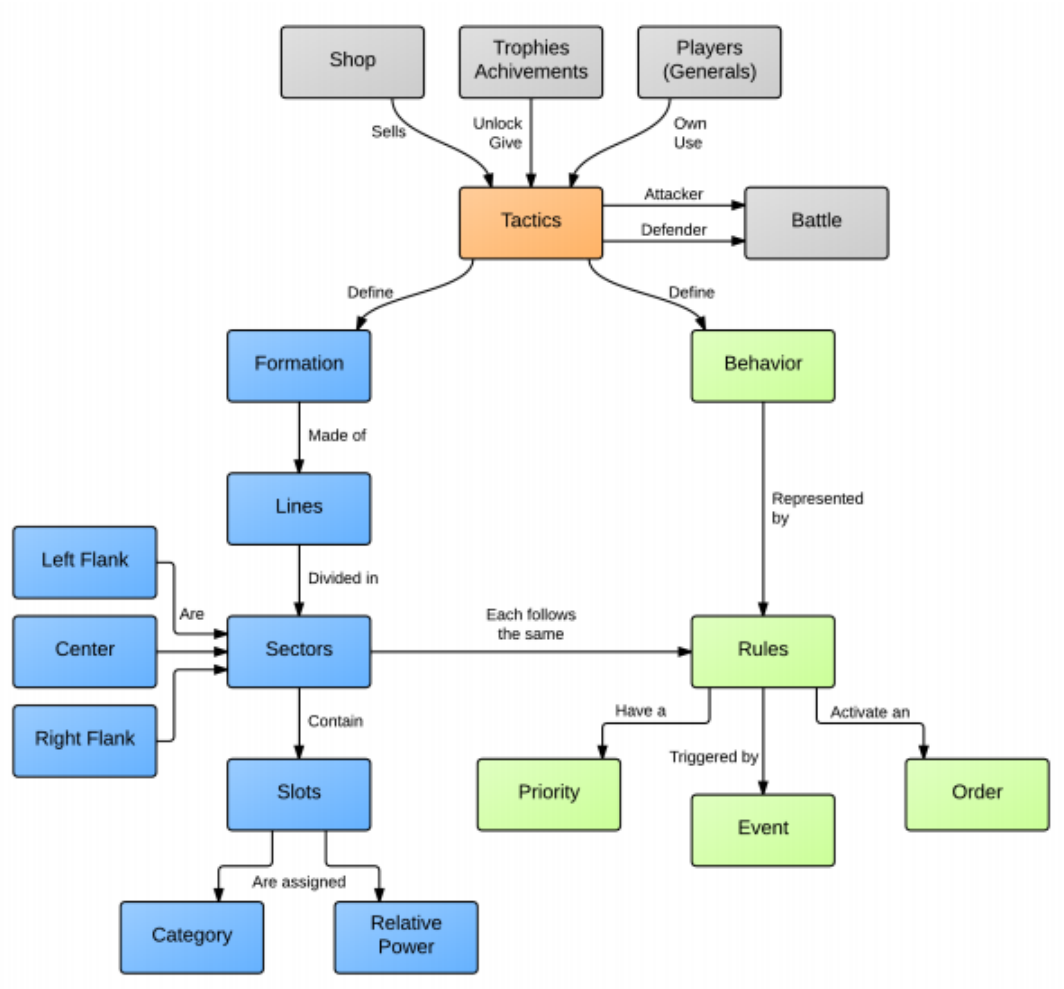


Figure 2.1: Different components of tactics [18, p. 30].

The soldiers are divided in five different categories [18, p. 32]:

- Ranged Infantry;
- Melee Infantry;
- Ranged Cavalry;
- Melee Cavalry;
- Artillery.

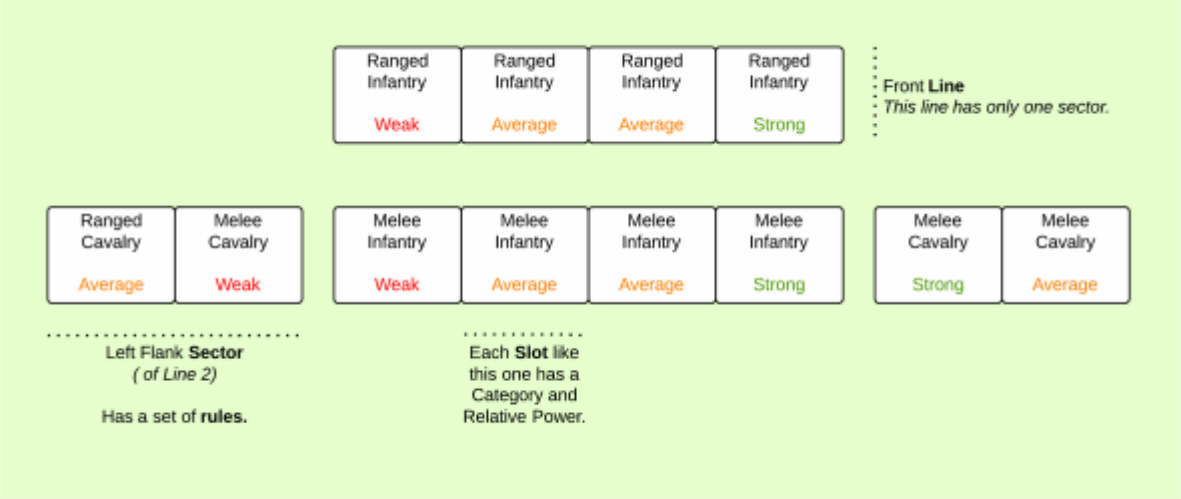


Figure 2.2: Example of a tactic with two lines of combat. [18, p. 31].

In general, the melee soldiers are specialized in close combat while the ranged soldiers are specialized in ranged combat using ranged weapons and avoiding at all costs any kind of close combat. On the other hand, there is the artillery which uses high damage equipments but they have low mobility or no mobility at all.

For strategic reasons, it is possible for the player to let an empty slot in the tactics in order to have a better control of the contingents positioning in the battlefield and it also grants the creation of unique tactics in the game.

2.1.2 Battle Processing

The simulator uses all the previous information about the tactic and the player soldiers to divide the soldiers in BCs and put them in their initial position according to the player's tactics where each BC is represented by a rectangle whose size varies with the number of soldiers in it and this means that if the number of soldiers decreases the size of the rectangle decreases as well. The rectangles cannot overlap other rectangles at any time during the battle except when an allied BC is running from the battlefield.

Each BC is an autonomous agent with a *Finite State Machine* that is responsible to decide what type of behaviour the agent must have along the battle. These agents have different possible behaviours such as moving towards an enemy or a spot in the battlefield, being able to rotate when needed, selecting a weapon to use depending on the distance to the enemy, selecting new targets to attack and helping allies in danger. Each of these actions can be slightly different between agents because it depends on the actual given order. They are also considered as ominiscient agents, this means that they have full vision of the battlefield, knowing all the enemies positions. [18, p. 49].

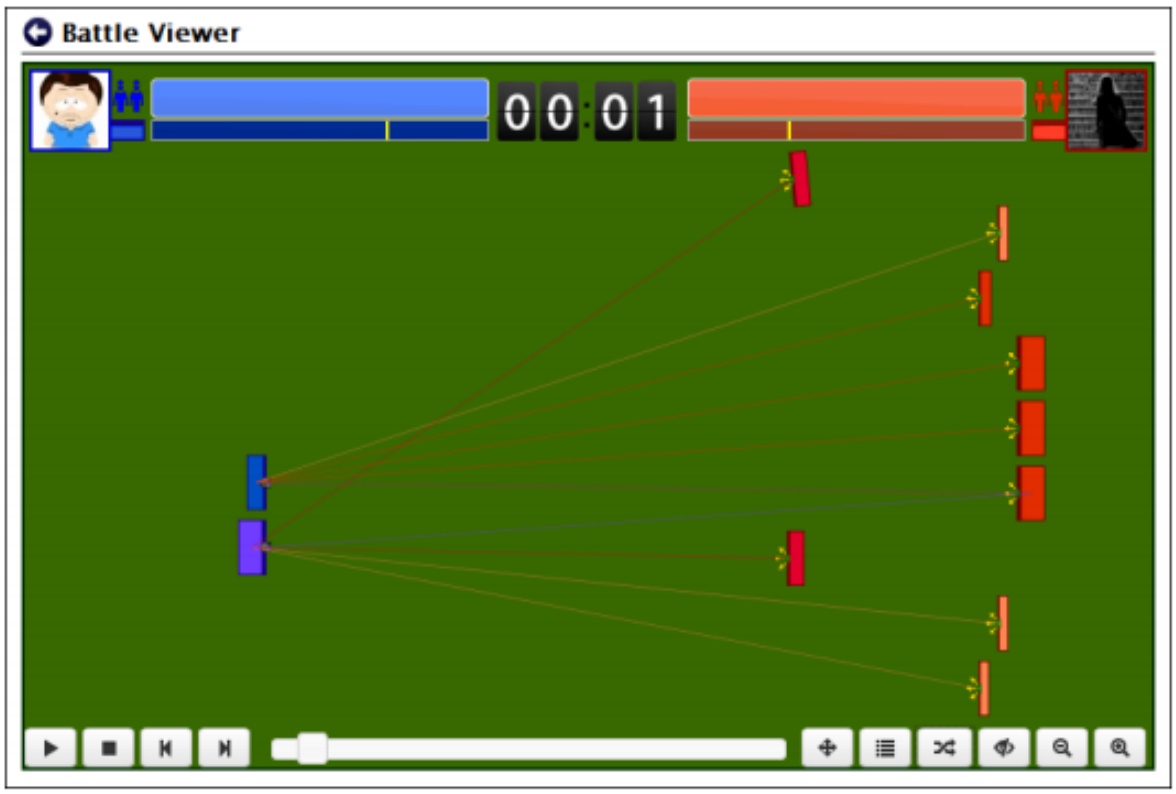


Figure 2.3: *Battle Viewer* [18, p. 50].

These contingents have different attributes that are modified by equipments and each category has different attributes values. The attributes are the following [18, p. 50]:

- Cohesion - represents the organization level of the BCs and decrements whenever a BC takes damage;
- Armor - reduces the decremented value to the Cohesion each time an attack is made and also reduces the number of casualties;
- Speed - represents the contingent's mobility. If the Speed value is high it means the contingent is agile;
- Shock Damage - represents the damage done after colliding with an enemy BC;
- Melee Damage - represents the damage done in close combat battles;
- Ranged Damage - represents the damage done in long distance battles;

Beyond these attributes, each contingent has an orders list that is established during the creation of the tactics and during the battle which will affect its behaviour throughout the battle. The contingents will always try to fulfill their current order even though they are in a bad spot.

This orders list has different types of orders that lead to simple behaviours, such as melee attack, ranged attack, defend, pursuit, and complex behaviours, such as flank attacks, rear attacks and skirmish attacks. These behaviours increase the level of complexity of the battles which also provide more diversity in terms of tactics and battles, making the game more interesting for players. [18, p. 88].

2.1.3 Quadtree

Throughout a battle the BCs may collide against each other being necessary to check multiple times the existence of collisions and instead of making a collision detection for each BC in the battlefield, a Quadtree is used which is very efficient for collision detection in a bidimensional space.

The Quadtree divides recursively a bidimensional space into four equal quadrants. In terms of code, a Quadtree is represented by a tree structure where the root of the tree represents the region to divide while the other nodes represents a quadrant [27].

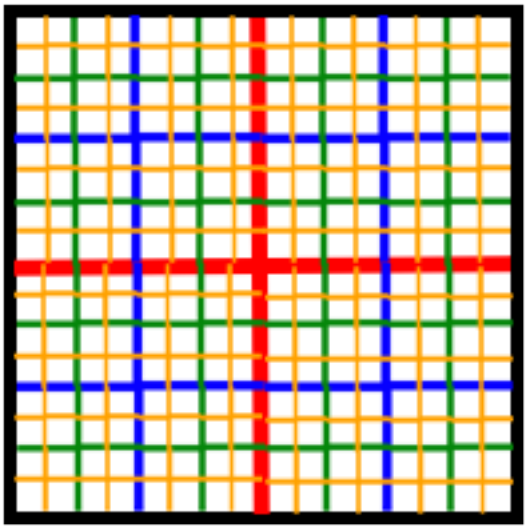


Figure 2.4: Division of the battlefield by four times.

The Quadtree implemented in the simulator has a maximum depth of four which creates a tree with five node levels and each parent node generates four child nodes where each child node represents a quadrant. At the maximum depth, there are only leaf nodes, which are nodes with no childs, where each node will contain the BCs information from a quadrant.

To check if a BC belongs to a certain quadrant, the four vertexes of the rectangle that represents the BC are used and if at least one of them are in the quadrant then the tree will continue expanding until reaching the leaf nodes where the BC information will be kept.

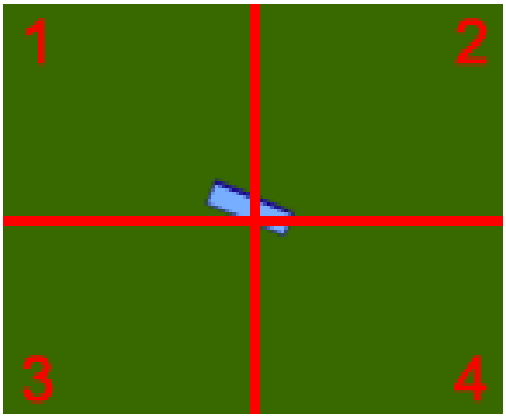


Figure 2.5: BC's vertexes placement with quadtree division.

The image above is an example of the situation explained in the previous paragraph where the BC is in three quadrants, two vertexes in quadrant 1 and one vertex in quadrant 2 and 4, which means that its information will be kept in these quadrants.

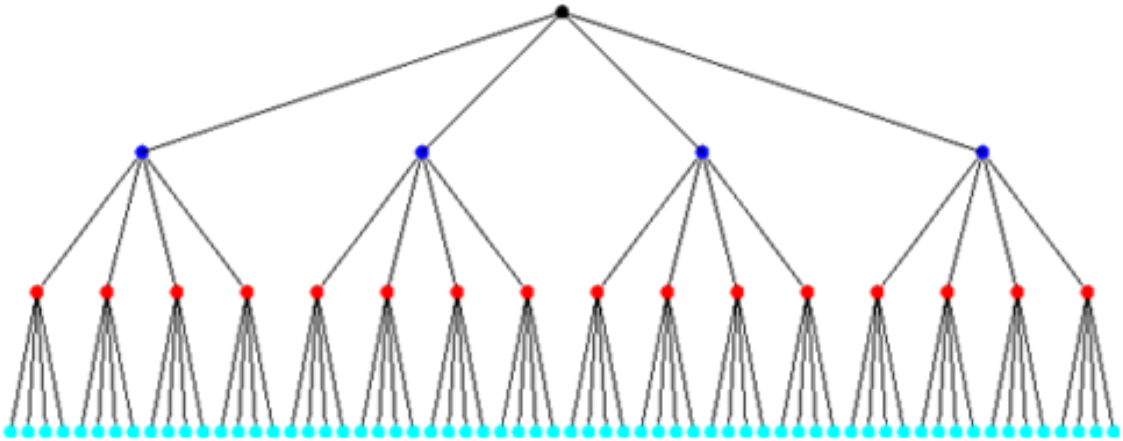


Figure 2.6: Quadtree with four levels.

Finally, a tree is generated where only the leaf nodes contain a BCs list, where all the elements of this list are considered as neighbors of each other. This list avoids the collision detection between all the BCs in the battlefield, being only necessary to detect collisions between neighbors because these are the closest BCs in the battlefield.

When two contingents are close to each other, all the conditions are created for a clash and both are subject to damage leading to casualties. A contingent routs when its cohesion is zero and disappears when the number of soldiers is zero.

The battle ends when one of the players loses all of his army or the cohesion level of his troops are too low.

2.1.4 Logging Battle Results

During the battle, all the information is kept in a *log* which will be accessible to the players in the "History" section. This *log* works as a recording of the battle and when a player decides to watch it, a video will start showing all the events of the battle. This is possible because the *log* keeps information from every frame, allowing to replicate all the movements and behaviours of the BCs in the battlefield during the battle.

2.1.5 Game Engine Issues

The issues with the battle simulator were related to performance because when two players were battling each other there was the possibility to exist a high number of contingents and each of these was making their own decisions.

On the other hand, the battle simulator was responsible to manage more than one battle request at the same time whereby it was necessary to have the best performance possible in order to decrease as much as possible the players waiting time to get their battle results.

To solve the identified issues, it was decided to reimplement the JavaScript battle simulator in a new programming language, in this particular case the new language was Go, to achieve one of the goals and hoping for an improvement of the simulator's performance.

2.2 Programming Languages Analysis

After an analysis to the game engine it is time to analyze the new language and the previous languange to identify the differences between them and also to verify if it is really worth it to conclude the incomplete version of the battle simulator developed in Go.

2.2.1 JavaScript

The previous language of the battle simulator was JavaScript which is a dynamic programming language, that is, a language that executes certain behaviours, such as adding new code, during the execution time, something that static languages can only do in compilation time. [6].

This language is often used for web browsers because it allows the usage of scripts in the client side to interact with the user, to control the web browser, to communicate asynchronously and to modify the document content that is being displayed. It is also used in the server side through the use of frameworks such as Nodejs, for game development and for desktop and mobile applications [16].

It is possible to say that the choice of the JavaScript to be the language for the simulator was good because it brings a lot of advantages for web browser oriented games but no one expected the performance issues. One of the possible reasons for these issues is the usage of a virtual machine that interprets JavaScript code which may affect the simulator's performance.

2.2.2 Go

The Go language, also known as Golang, is an open-source language that was developed and announced by Google in November 2009 [10]. This is a statically-typed language which means that the data type is verified in compilation time, which allows the early detection of errors without any performance costs [31].

The selling point of this language was the fact that parallel and concurrent programming was easy and simple to implement, making the developers life a little easier because they would not need to deal with the complexity behind concurrency and yet achieve high performance results.

The syntax of this language is identical to the C language which makes the learning and adaptation processes easier and because of this the number of developers using Go increased. When the number of developers increase the number of information through the web tends to increase as well which helps the novice developers to give their first steps using this language.

For concurrency, this language has a special system which is called *Goroutines* and it is very similar to threads. *Goroutines* are considered as a lightweight thread and to use them is like calling any common function and this is why it is easy and simple to use them. [8].

This language has the tools to solve the current performance issues in the battle simulator because it is developer friendly in terms of concurrency and parallel programming which can save some time during the development stage and at the same time it will help achieving the goals defined at the beginning.

2.2.3 Differences Between Languages

There were some differences between both languages which in some cases could create issues or even produce different outputs for the same battle. In this section, the main idea is to talk about their differences and try to understand why they existed.

One difference that was detected was in the sorting algorithms used by both languages, so it was necessary to make an analysis to these algorithms since they may produce different sort outputs for the same input array, particularly if there were repeated elements in the array.

In Go, the only way to sort an array is to use a sorting interface by invoking the function *Sort*. This function uses a sorting algorithm known as *Quicksort*, but this algorithm is an optimized version to execute as fast as possible so this means there is a chance to use a *Quicksort*, a *Heapsort* or an *Insertionsort* depending on the array [28].

The *Sort* function requires the definition of the function *Len*, which is used to return the size of the array, the function *Swap*, which tells how to swap elements in an array, and the function *Less*, that is used to compare two elements of the array.

```
func quickSort(data Interface, a, b, maxDepth int) {
    for b-a > 7 {
        if maxDepth == 0 {
                heapSort(data, a, b)
                return
        }
        maxDepth--
        mlo, mhi := doPivot(data, a, b)
        // Avoiding recursion on the larger subproblem
        //  guarantees a stack depth of at most lg(b-a).
        if mlo-a < b-mhi {
                quickSort(data, a, mlo, maxDepth)
                a = mhi  // i.e., quickSort(data, mhi, b)
        } else {
                quickSort(data, mhi, b, maxDepth)
                b = mlo  // i.e., quickSort(data, a, mlo)
        }
    }
    if b-a > 1 {
        insertionSort(data, a, b)
    }
}
```

The code above is the *Quicksort* implementation that is in the sorting library of Go and as it was said before there is a chance to use either a *Heapsort* or an *Insertiosort*, instead of using a *Quicksort* [30].

The *Quicksort* used by Go, starts with the pivot selection which calculates the median of three, that is the median of the first, the middle and the last element of the array, being one of these selected as a pivot and inserted at the beginning of the array.

Having a chosen pivot then it is time to start the sorting process where the elements that are equal or less than the pivot inserted in the first indexes of the array while the greater elements inserted at the remaining indexes. Finally, after a cycle step, the pivot goes to its final position in the array.

JavaScript, much like Go, uses the function *Sort* in the sorting algorithm, but in this case the algorithm used depends on the data type of the array. If the data type is numeric it uses a variation of the *Quicksort* algorithm, known as *Introsort*, if the data type is non-numeric and contiguous it uses a *Mergesort*, if this algorithm is not available then it will use the *Introsort* and finally, if the data type is different from the above it uses the *Selectionsort* or in some cases an AVL tree [17].

Since the comparisons made between elements in the battle simulator are from numeric nature this means that both languages use the *Quicksort* but each language has their own implementation of this algorithm because if an array has multiple repeated elements the output is different.

One of the possible causes for this situation must lie in the pivot selection used by the *Quicksort* because there is multiple ways to select a pivot. The pivot selecton can be a random element of the array, the element in the middle of the array or a median of three. Any of the previous methods avoid the worst case behaviour. Another reason that is also plausible is the use of different thresholds from both algorithms.

```
Before Sorting

[[0,0,2,1] [3,5,0,0] [3,5,0,1] [3,5,1,0] [3,5,1,1]
 [3,5,1,2] [3,5,1,3] [3,5,2,0] [3,5,2,1] [4,5,2,0]
 [4,5,2,1]]

After Sorting
```

JavaScript

```
[[0,0,2,1] [4,5,2,1] [3,5,0,1] [3,5,1,0] [3,5,1,1]
 [3,5,0,0] [3,5,1,3] [3,5,2,0] [3,5,2,1] [4,5,2,0]
 [3,5,1,2]]
```

Go

```
[[0 0 2 1] [3 5 0 0] [3 5 0 1] [3 5 1 0] [3 5 1 1]
 [3 5 1 3] [3 5 2 0] [3 5 2 1] [4 5 2 0] [4 5 2 1]
 [3 5 1 2]]
```

Another difference that is worth talking about is the fact that in JavaScript there is no need to declare the type of variables and the type of function arguments while in Go it is required to declare their type, being this procedure used by the most known programming languages. Below it is represented the same function for both languages to show the previous difference.

JavaScript

```
Battle.process_step = function(bc) {
    bc.update_target();
    bc.move();
    bc.combat();
    bc.update_status();
}
```

Go

```
func (b *Battle) process_step(bc *BattleContingent) {
    bc.update_target(b)
    bc.move(b)
    bc.combat(b)
    bc.update_status(b)
}
```

With this example it is possible to conclude that besides the existing difference previously talked, these functions are identical.

2.2.4 Parallelism *vs* Concurrency

Since the issues with the battle simulator were related to the performance it is important to talk about parallelism and concurrency. Both are often used to improve the performance of any program through the execution of multiple processes at the same time instead of using only one.

Concurrency and Parallelism creates a lot of confusion because both are described as multiple processes executed at the same time. The difference between them is that parallelism makes use of multiples cpus at the same time while concurrency uses multiple processes with only one cpu where each process has a short period of time to execute.

In programming, the model used to implement this kind of behaviour is called *Multithreading* which uses multiple threads with a sequence of instructions that are executed in a given interval of time [35].

In the battle simulator what would solve the performance issues is to parallelize the current code so it is necessary to give an explanation on how the parallelism works on both languages.

Go has its own scheduler that is responsible to manage the goroutines that are used in parallelism and concurrency. The creation of the scheduler was necessary because the goroutines use less information than the threads used by the operative system and also the scheduler must make informed decisions, something that the operative system scheduler cannot do. An example of an informed decisions is the Go scheduler can only run the garbage collector when all the goroutines are stopped and the memory is in a consistent state [34].

A goroutine is a lightweight version of a thread and, as it was said before, they are managed by the Go scheduler. The goroutines are executed in the same address space and the access to shared memory must be synchronized, this can be made using sync libraries that comes with the language or using channels.

A channel creates a connection where it is possible to send or receive information using the operand "<-" which is used to synchronize the goroutines without using the traditional locks system.

```
func main() {
        nCPU := 2
        runtime.GOMAXPROCS(nCPU)

        t1 := time.Now()
        array := make([]int, NUMBER)
        sumTotal := 0

        c := make(chan int, nCPU)

        fillArray(0, NUMBER, array)

        for i := 0; i < nCPU; i++ {
                go sumArray(i*NUMBER/nCPU, (i+1)*NUMBER/nCPU, array, c)
        }

        doSomething()

        for i := 0; i < nCPU; i++ {
                sumTotal += <-c
        }

        fmt.Printf("Sum Total: %d\n", sumTotal)
```

```
        tf := time.Since(t1)
        fmt.Printf("Total Time: %v\n", tf)
}
```

The example above creates an array with size NUMBER and then adds all the elements of the array. This example shows the use of parallel programming through the use of goroutines that are instantiated with the keyword ”go” and the use of channels that are created using slices.

To add parallelism it is required to say the number of cpus which represents the number of threads that it is going to be used during the runtime by the *GOMAXPROCS* function. In order to parallelize the *sumArray* function the array is split by the number of cpus it is going to be used. So each goroutine are responsible to execute the *sumArray*of a fragment of the original array which will be faster than using the *sumArray* function sequentially to add all the elements in the array.

On the other hand, the *doSomething* function is concurrently executing at the same time as the *sumArray*.

This program will only finish when all the goroutines send the final result of each fragment of the array.

JavaScript has no support for parallelism by itself but nowadays exist libraries for Nodejs that try to make the parallelism in this language a dream come true. The most known library is the Parallel.js which makes use of Web Workers. Web Workers can be used in both web browsers and Nodejs and execute tasks in background.

In general, everytime a Web Worker is created a new thread is also created where it will run some code from a different file and it is also possible to communicate between other Web Workers and the main thread [33]. In a way the Web Workers are very similar with the goroutines and channels synergy, but it has a high level of complexity which makes them difficult to use.

The Parallel.js tries to reduce the level of complexity of the Web Workers by using high level functions.

```
var p = new Parallel([40, 41, 42]),
    log = function () { console.log(arguments); };

// One gotcha: anonymous functions cannot be serialized
// If you want to do recursion, make sure the function
// is named appropriately
```

```
function fib(n) {
  return n < 2 ? 1 : fib(n - 1) + fib(n - 2);
};

p.map(fib).then(log);
```

The example above is a Fibonacci function using Parallel.js taken from the Parallel.js official site [24]. To use parallelism in Fibonacci function it must use a constructor named *Parallel* which receives the data to parallelize as argument, in this example he data is an array, then it can receive three optional arguments, such as the *maxWorkers* that represents the number of cpus to use and the default value is four, , the *evalPath* that represents the path to the file if using this library with Nodejs and finally, the *synchronous* that controls the usage of synchronous timeouts in case there is no Web Workers available and the default value is true [24].

In this case, the *map* function executes in parallel the *fib* function for each element of the array printing the final result.

This library is not available for all the versions of web browsers and it is client-side only which can be considered as a disadvantage.

Unfortunately, there are no libraries for Node.js which offers parallelism with such a simple interface for server-side web applications but there is still possible to get parallelism using other libraries or modules such as cluster.

Since the Node.js operates in a single thread, the cluster module takes advantage of multi-core systems to launch a cluster of Node.js processes. The cluster module supports two methods of distributing incoming connections, the first method is the round-robin approach where the master process listens on a port, accepts new connections and distributes them accross the workers and the second method is where the master process creates the listen socket and sends it to interested workers [3].

To use this module effectively it is necessary to understand its interface and a lot of parallelism definitions which can be time consuming.

Chapter 3

Algorithm Analysis and Proposed Solution

3.1 Profiling

Since the main issue with the battle simulator was related to its performance the first thing to do was to create a performance profile in order to better understand the reason behind this issue and also to identify possible causes. Profilers use a wide variety of tools to collect data and are extremely important for understanding the program behaviour.

Self	Total	Function
43.27%	43.27%	GenRectangle.line_intersection
6.62%	52.81%	GenRectangle.line_intersects
6.42%	7.47%	BattleContingent.smallest_distances
4.41%	59.52%	BattleContingent.target_weight
3.96%	4.01%	GenRectangle.coordinates
3.69%	18.37%	BattleContingent.minimum_distance
3.63%	3.63%	(garbage collector)
2.12%	4.94%	Point.linedistance
1.84%	1.84%	Point.subtractPoint
1.58%	2.02%	Battle.qt_node_fetch
1.51%	1.52%	Point.normsquared
1.40%	2.04%	Point.norm
1.14%	3.53%	BattleContingent.collided
1.06%	1.06%	(anonymous function)
1.06%	1.06%	(program)
1.04%	1.45%	Battle.qt_intersect
0.94%	1.07%	BattleContingent.set_if_changed
0.82%	2.61%	Point.angle
0.82%	13.58%	BattleContingent.calculate_boids_forces
0.72%	2.82%	BattleContingent.populate_zoc
0.72%	1.62%	GenRectangle.point_in
0.70%	0.70%	GenRectangle.edge_side
0.69%	1.70%	Battle.qt_node_walk
0.63%	0.63%	Point.add_point
0.54%	0.54%	Point.dmultiply
0.51%	0.51%	Point.equals

Figure 3.1: Performance profile from the JavaScript battle simulator.

The image above shows the results of the performance profile for the battle simulator code showing the time spent in each function during runtime. This image contains a table with three columns, the first column is the self time and it represents the time, in percentage, actually spent inside a certain function, the second column is the total time and it represents the time, in percentage, between the first call to the function until the function returns or ends and finally, the third column is the function column and it shows the name of the function evaluated.

To analyze our performance issue, the main focus is to look for the functions with higher self time because these are the ones probably creating the performance issue and looking at the image the function with the highest self time is the *line_intersection*.

```
GenRectangle.line_intersection = function(startA, endA, startB, endB) {
  var ax = endA.x - startA.x,
  ay = endA.y - startA.y,

  bx = endB.x - startB.x,
  by = endB.y - startB.y,

  abx = startB.x - startA.x,
  aby = startB.y - startA.y,

  determinant = -ax*by  + ay*bx ;

  if (determinant == 0) return false;

  var d1 = (ax * aby - ay * abx) / determinant;
  var d2 = (-by * abx + bx * aby) / determinant;

  return d1 >= 0 && d1 <= 1 && d2 >= 0 && d2 <= 1;
}
```

Looking at the function code it is possible to see that it is a fairly simple function that makes some mathematical calculations and as the name tells the main goal of this function is to verify if a line intersects or not. This function is called in the *target_weight* that calculates the weight of an enemy BC according to its distance and threat to the current BC. The enemy BC with the highest weight is selected as target.

3.2 Game Architecture

The software game architecture is showed in the image below and it did not suffer any kind of modifications throughout the thesis because the main issue was related to only one module of this architecture. The proposed modifications to this module were the change of the programming language, code optimization and the insertion of parallelism.

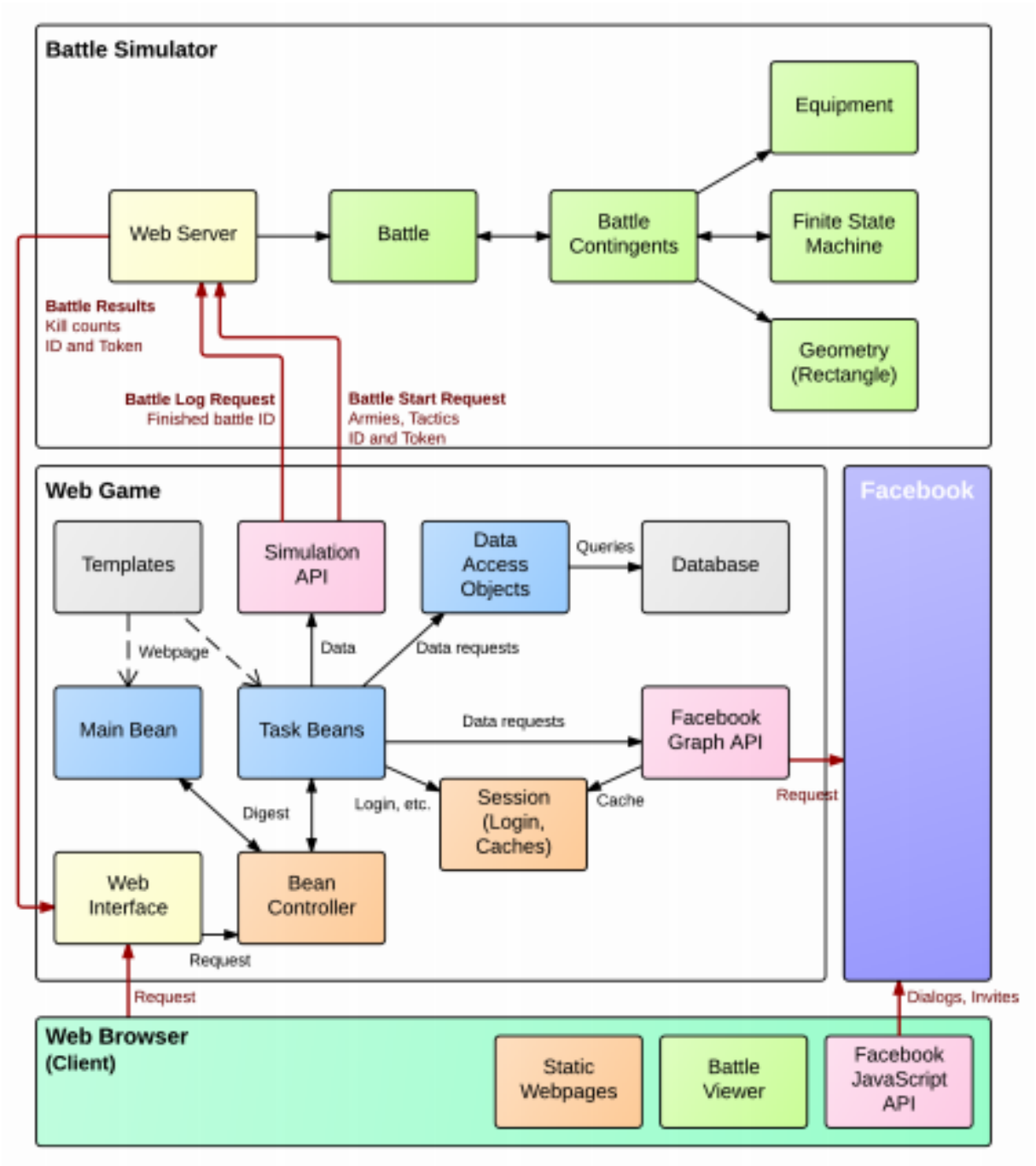


Figure 3.2: Current game software architecture [18, p. 45].

In this architecture, the *Web Browser*is the mean to access the game, the *Web Game* is the module responsible to generate the HTML pages and the data maintenance while the *Battle Simulator* is the module responsible for managing all the aspects related to the battles. [18, p. 44-46].

After developing the new version of the battle simulator it is necessary to evaluate the performance results from both versions in order to see if the proposed solution was a success and if the thesis' goals were achieved.

3.3 Parallelism

With the code analysis it was possible to conclude that the best place to insert parallelism would be in the code block responsible to execute the BCs actions because it was where it made more sense and also where the code execution spent most of the time when processing battles. But it was also clear that this problem was not a fully independent because there was an order when executing the BCs actions which prevented the better use of the parallelism.

But to insert parallelism it was necessary to make some changes to the code in order to avoid synchronous issues and wrong battle outputs so one thing that was changed was the execution of the BCs actions.

The idea was to execute one action of all BCs and only then they would execute another action, instead of a BC executing all of its actions. With this approach, it was possible to split the number of BCs by the number of cpus.

With the previous idea implemented it was time to make some testing with parallelism and it started very well but there was an action that was creating some issues. With some debugging, the action that was creating all the fuss was the move action and the reason why will be explained next.

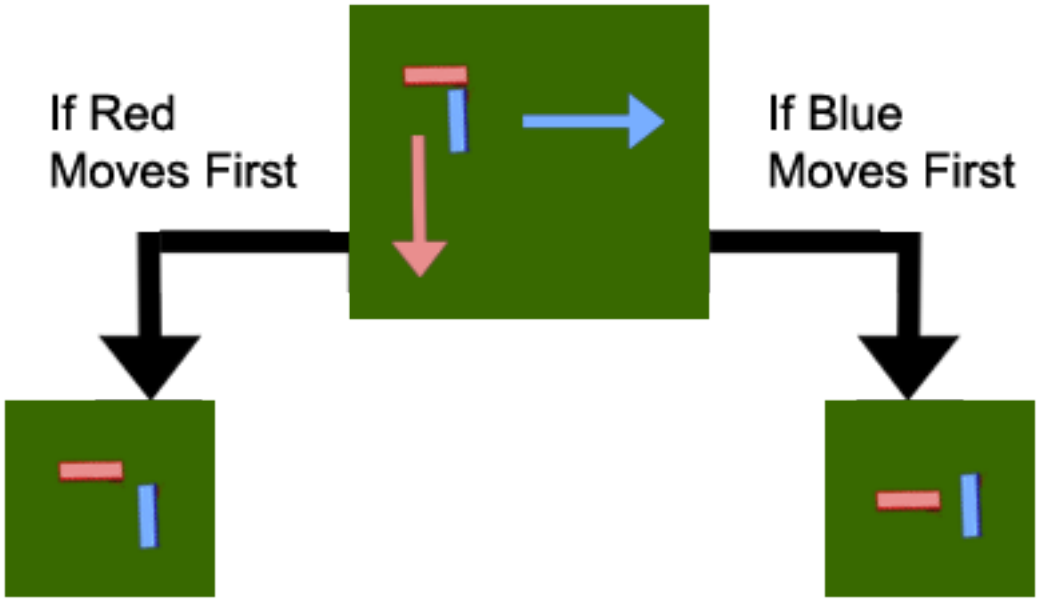


Figure 3.3: Illustration of the problem found in movement action.

The image above shows an example of the problem caused by the movement action. The blue and red arrows represents the movement direction of each BC with the same color. So if the red BC moves first, the final result will be the image on the left, otherwise the final result will be the image on the right.

The issue above can happen because of the dependency of this action. If the BCs executed the move action by id in an ascending order the result would be different than executing the move action in a descending order and this was the reason why this action could not be parallelized.

3.4 Conclusions

With all said and being the *line_intersection* function already optimized, it was decided to finish the incomplete version developed in Go and after that and, with the new language support, insert parallelism to improve the performance of the battle simulator.

Chapter 4

Development

This chapter explains the development process of the new battle simulator version in Go as well as all the issues encountered during this process, explaining their origin and how they were solved, and a brief comparison between versions.

The first step to take was to know the current state of the simulator in Go and this version was compiling without any errors but it was producing invalid battle outputs where most of the BCs were not even drawn and they would not execute any of their actions. After knowing the state of the simulator, it was time to create a plan to guarantee that the development went as smoothly as possible.

It was decided that this version must have the same functionality as the JavaScript version so the process consisted in coverting code from JavaScript to Go with some adjustments and improvements during this process. Having this in mind and the issues found, it was necessary to go through a long debugging process solving all the issues to make the Go version fully functional.

The duration of the debugging process was too long because the issues were all over the code and to guarantee that every issue was found and solved, all the code was inspected. Having the other version also helped a lot since it was possible to use it to compare results. With this process it was possible to develop a fully functional version of the battle simulator in Go.

As it was said in the previous chapters, the battle simulator needs to get from the Web Game module all the information about the players that are going to battle. Having this information, the simulator will organize it and use it throughout the battle, in particular, it will be used by the main function of the simulator.

Below is the code from main function of the simulator which is named as *Simulate*. This function can be divided in three blocks: preparation & positioning, battle processing and logging battle results.

```
Simulate(player_a, player_b) {
    /* Preparation & Positioning */
        create_new_battle(player_a, player_b);

        sort_slots(player_a);
        sort_slots(player_b);

        create_and_place_battle_contingents(player_a);
        create_and_place_battle_contingents(player_b);

    /* Battle Processing */
        prepare_log_info();

        for i = 0; i < MAX_SIMULATION_STEPS
                        && winner == 0; i++ {
                update_battlefield();
        }

        if winner == 0 && winner_decided != 0 {
                winner = winner_decided
        }

    /* Logging Battle Results */
        update_log_info();
        create_and_dump_info_to_log();

}
```

The first block is responsible for the creation of the battle between players by calling the *NewBattle* function which is going to initialize all the necessary variables for the battle. Then, it will sort the slot list for each player to guarantee that the slots with the best rank comes first in the list. Finally, a side is assigned for each player and their army positioned in each side of the battlefield.

```
/* Preparation & Positioning */
    b := NewBattle(player_a, player_b)

    sort.Sort(player_a.Plan)
    sort.Sort(player_b.Plan)
```

```
    player_a.battle_position = -1
    player_b.battle_position = 1

    b.place_side(player_a)
    b.place_side(player_b)
```

To position the troops in the battlefield it is required to make some calculations to put the troops in the right slots and also to calculate their initial position.

```
place_side(player) {
    categories = split_by_category(player_army);

    for each category {
        calculate_best_slots();

        /* Associate slots to BCs */
        place_side_category();
    }

    compute_bcs_positions(player);
}
```

The code above represents the function responsible to make the previous calculations and its named as *place_side*. This function starts to split the troops into categories, using the *split_by_category* function, and for each category it will select the best slots. After this selection, it is time to create the BCs and associate each of them to a slot through the *place_side_category* function.

```
place_side_category(plan, contingents, player) {
    /* Max size each BC can have */
    calculate_max_bc_size();

    /* Compute how many BCS a CT will be split into
    Also count total BCs */
    compute_number_of_bcs_generated_from_ct();

    /*Create all BCs without positioning them */
    create_all_bcs():
```

```
    /*Start by filling the best slots */
    with the best BCs

    fill_best_slots_with_best_bcs();

    position_bcs_in_battlefield();
}
```

The *place_side_category* function starts by counting the number of troops from a category and calculates the maximum number of troops each BC can have and this number varies with the category. The next step is to split the troops into BCs and insert them into a list. Then this list will be sorted by Power in a descending order. Finally, the BCs can be positioned in the side of the battlefield that was assigned to each player.

```
/* Battle Processing */
    player_a.results = b.prepare_results(-1)
    player_b.results = b.prepare_results(1)

    b.phase = PHASE_START

    b.add_log(b.get_state_log(true))

    winner := 0
    i := 0

    for ; i < MAX_SIMULATION_STEPS && winner == 0; i++ {
        winner = b.update()
    }

    if winner == 0 && b.winner_decided != 0 {
        winner = b.winner_decided
    }
```

After this stage is complete, the block for the battle processing will start and in terms of code is just a loop that executes a battle step, which in other words means that all the BCs in the battle will execute their actions one time. At the end of each step, the simulator checks if there is a winner or if the number of maximum steps has been reached in order to end the battle.

```
update() {
    executes_all_bcs_actions(player_a);
    executes_all_bcs_actions(player_b);

    rout_bcs();
    kill_bcs();

    update_log_info();

    verify_winner_decided();
}
```

Each BC executes four different actions following a defined order:

1. Pick an enemy BC to attack;
2. Move;
3. Attack;
4. Update internal state.

To execute the first function, the BC has to access all its neighbors and then verify which of these are the closest enemies being these inserted into a list of possible targets to attack.

In order to get all the neighbors of a BC, it was used a data structure known as Quadtree and it was previously explained.

When a BC is inserted in the battlefield it is also inserted in the Quadtree and everytime a BC updates its position it also updates its data from the Quadtree and because of this each BC can be removed or inserted multiple times in the Quadtree throughout the battle.

The next action to execute is responsible to make the movement forces calculation which are going to be applied to a BC making them to move. These forces can be rotational forces which cause a BC to rotate, moving forces which make a BC to move and BOIDS-like [5] forces which simulate the cooperation and formation between BCs. After this calculations, it is time to try to apply these forces and before doing that it is required to make some collisions verifications. If these forces are applied to a BC and it collides, then the algorithm will try to just apply rotational forces and if the BC is still colliding another BC, then it does not move in this step, otherwise the forces will be applied causing the BC to move.

The BOIDS is an algorithm that simulates the flocking behaviour of the birds and it is a good example of emergence [1]. The emergence is the process where larger entities, patterns and regularities

arise through interactions between smaller or simpler entities [9].

In the attack function, the BC will attack all the enemies that are in the weapon range. The damage that a BC can cause to another is modified by equipments, current order and current state and this damage is divided by the number of enemies set as targets. The damage dealt to the enemies may cause casualties.

```
/* Logging Battle Results */
    b.populate_results(player_a.results, -1)
    b.populate_results(player_b.results, 1)

    b.Battle_result.Winner = winner
    b.Battle_result.Steps = i

    b.Battle_result.Results_a = *player_a.results
    b.Battle_result.Results_b = *player_b.results

    b.dump_result()
```

After every BC execute their actions, all the information of a step is saved in a map and after that it verifies the existence of a winner. When a winner is determined or the maximum number of steps are reached, all the information in the map will be written into a log file which is going to be used later by the Battle Viewer. The Battle Viewer reads the log file data and creates a battle animation showing all the events of the battle to the players.

4.1 Issues

When the development process was almost finished, some issues, that were also present in the other version, were revealed and from these issues one was very problematic.

This issue occured everytime a BC was declared dead and consequently removed from the battlefield. As explained in the previous section, a step from a battle consists in executing actions for all the BCs in the battlefield and when one of these are declared dead it is immediately removed from the player's BC list which results in a reduction of the list's size. With the previous explanation it is not possible to visualize the issue, so an example was made to better understand the issue.

```
Example: Imagine a player with three BCs
         with 1, 2 and 3 as their ids.

func (b *Battle) update() int {
    var bcs_list = [1, 2, 3];

    /* Execute bcs actions */
    for i:= 0; i < len(bcs_list); i++ {
        var bc = bcs_list[i];
        execute_bcs_actions(bc);
}
```

In the example above, the BC with id 1 is the first to execute its actions, using the *execute_bcs_actions* function, and it is verified that its current state is dead, so it means that this BC must be removed from the battlefield and the loop ends. By removing a BC, the size of *bcs_list* is reduced to two and by ending the loop the variable i is incremented to one, so in this example the BC with id 2 will not execute any actions in this step something that must not happen. In order to solve this issue, the BCs with current state dead are only removed from the list at the end of a step avoiding the previous issue.

The previous issue would also happen if the current state of the BCs was rout but in this case instead of making a BC not to execute any actions in a step it makes a BC to execute its actions twice in a step. This behaviour is possible because the BCs with current state ok execute their actions first than the BCs with current state rout.

The solution was to change their state to rout after all the BC with state ok have already executed their actions. It was decided to let the double execution of the actions because it decreases the number of steps of a battle and also because it does not change the final outcome of the battle since the rout state means that the BC has already given up and it is running from the battlefield.

4.2 Differences between versions

After finishing the development of the battle simulator in Go it was possible to see that there were two major differences between this version and the JavaScript version which were visible in the battle outputs.

The two versions were producing different sorting results for the slot list and different outputs from the cosine function which affected the positioning from the BCs throughout the battle making the BCs to

have a behaviour variation and also an increase or decrease of the number of steps

The first difference created different initial positioning for the BCs, given that the slots of each player were sorted by their rank in a descending order, thus, from the best slot to the worst slot and in the beginning it was possible to have multiple slots with the same rank which led to different sorting results from both languages, despite both languages using the same sorting algorithm. This behaviour proved that even using the same sorting algorithm it can lead to different results because each language has their own optimization of this algorithm which also means that each language deals with multiple repeated elements in a different way.

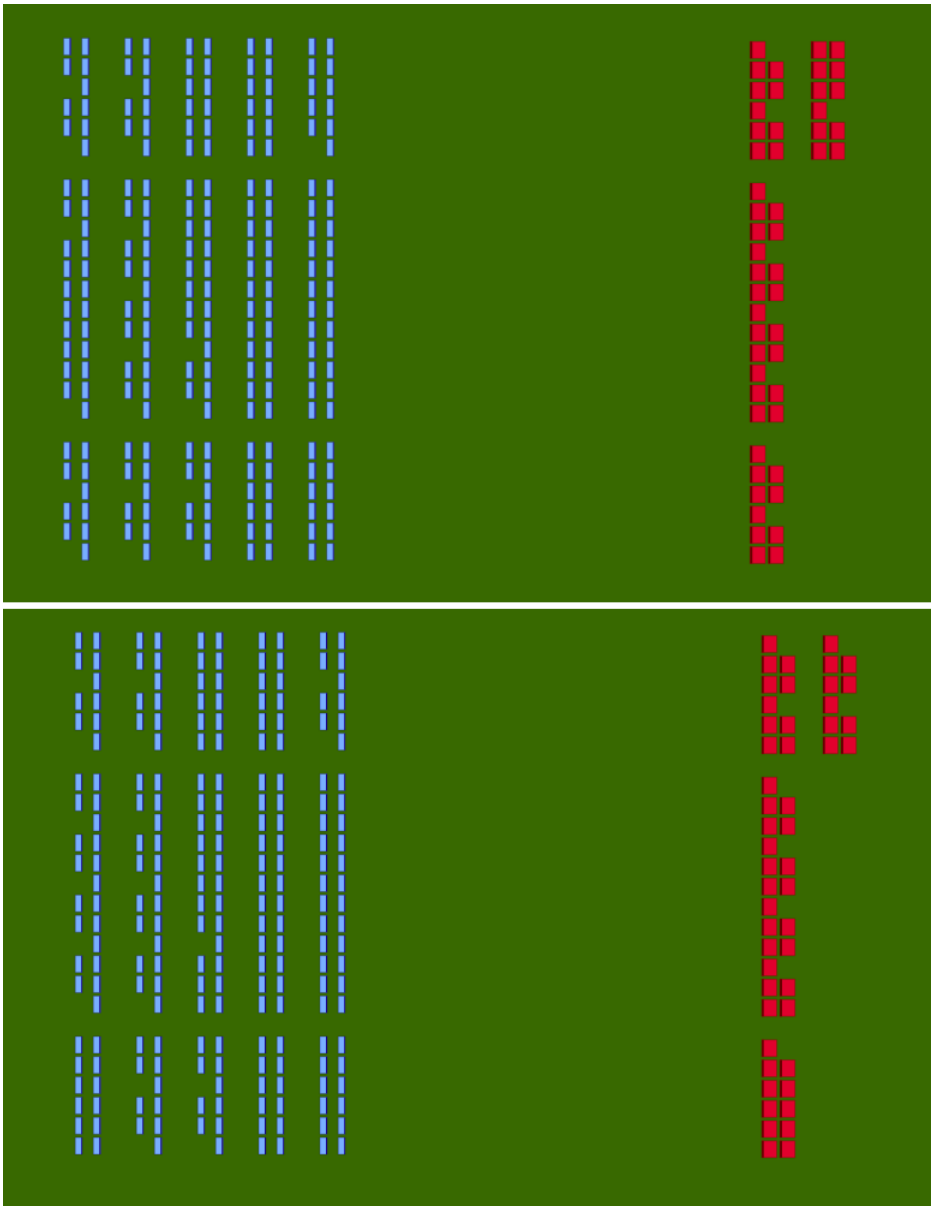


Figure 4.1: The image on the top shows the positioning of all BCs in the battlefield generated by JavaScript while the image on the bottom shows the positioning of all BCs in the battlefield generated by Go.

In the figure above, it is visible the differences of the BCs positioning caused by the previous issue. Although these differences are not very significant they can create a variation of the number of steps between the two battle simulator versions. Despite the existence of this issue, it is not very alarming since it is not visible to the players because they will never have the opportunity to compare the results from the two versions.

The other difference was related to a problem with the cosine function which also caused positioning changes and a variation of the number of steps. This issue was only visible in long duration battles having no impact in short duration battles.

```
Example: Go - math.Cos(-0.1) -> 0.9950041652780257
         JS - Math.cos(-0.1) -> 0.9950041652780258
```

The example above shows the value difference of the cosine function for the same input in both languages. To know if this issue was caused by the battle simulator versions or by the languages itself it was decided to test this same example in an online code playground. Firstly, the Go language was tested and the final result of the cosine for the same input both locally and on playground was the same. The next step was to use the previous method but this time for JavaScript and the final results were different. With this, it was possible to conclude that the cosine function in the JavaScript battle simulator version was returning different results.

After some research, a possible reason found for this behaviour is related to the cosine and sine implementation which uses a mathematical function, known as *TrigonometricInterpolation*, that uses a reverse lookup table with samples generated in C++. What happened was that this table was updated and the Nodejs, in v0.10.36 version, still uses the outdated table while the online code playground already uses the updated table and for this reason the returned values from the cosine functions from both the Node.js and the online code playground are different, being the returned value from the playground the most accurate [20].

This slight difference originated uneven battle results because there were rotational forces that were applied to the BCs whose forces used the cosine and sine functions and since each battle simulator version had a different result from cosine it produced different BC positioning throughout the battle.

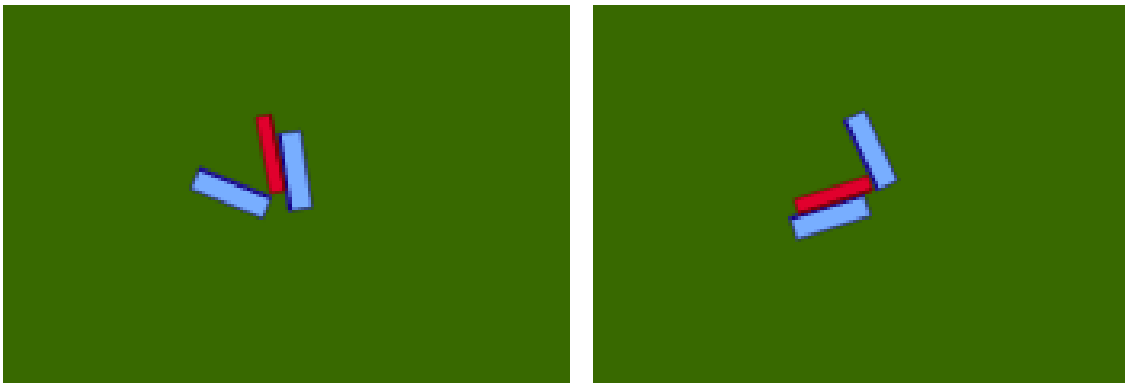


Figure 4.2: The image on the left shows the last step of the battle in Go while the image on the right shows the last step of the battle in JavaScript.

The image above shows an example of a 2 versus 1 battle situation and for the same input it is possible to see that in the last step of the battle the positioning of the BCs are not the same in both versions because of the cosine issue explained in the previous paragraph.

If the code of both versions were side by side, it was possible to see that they were very identical, but as expected there were some differences specially because of the capacity of each language.

One of the things that was changed when developing the new version was the code variables organization through the usage of structures.

One possible example is the *Player* structure which contains all the variables necessary to represent a player in a battle and, on the other hand, the usage of structures prevents the global access to these variables.

```
type Player struct {
    Plan             Planlist
    Army             []Unit
    battle_position  int
    front, rear      int

    bcs          BattleContingentList
    bcs_to_kill  BattleContingentList
    bcs_to_rout  BattleContingentList
    results      *ResultMeta
    id           string

    ok, routed, logonce int
}
```

4.3 Parallelism Implementation

After solving most of the issues with the new version in Go, the next step was to insert parallelism in order to get a performance improvement and to make the game more appealing to players, specially the new ones.

In order to insert parallelism it was necessary to create a goroutine, which is a lightweight thread, for each cpu where each of them will be managed by the Go Scheduler and to share information between goroutines it is necessary the use of channels which are used to send values from one goroutine to another.

```
Parallel: RED
Single-Threaded: GREEN

execute_actions() {
    all_update_target();

    all_move();
    all_combat();

    all_update_status();
}
```

Since there was dependency problems, the final solution for this problem was to use a mixed solution between single-threaded and parallelism in order to obtain some performance improvements and also to guarantee the correctiveness of the battle results. Probably there would be a better way to solve this issue but, once again, this solution showed good results.

Chapter 5

Tests

After finishing the development of the new version it was time to put it online and use some battle inputs to test it and also to make a comparative analysis between the versions.

During this process there were three versions of the battle simulator tested, one was the original version developed in JavaScript and the other two were developed in Go where one is identical to the JavaScript and the other is a version with parallel programming using four cpus. The test process consisted in using the battle inputs from the game and locally run these inputs between the different versions, registering their execution times.

After running all the tests, a table was created with the registered execution times from each version and with the battle information from each input. With this table it was possible to create graphics to better visualize the difference between the three versions.

5.1 Battle Tests

Before getting to the tests results, it is important to explain some battle characteristics from each battle input used. These tests were divided into two groups, the controlled battle tests and the random battle tests. In the first group, there are five battle tests where both players use the same tactic and troops and the only thing that changes from battle to battle is the number of BCs. In the second group, there are eight battle tests that were picked randomly from the game. For each battle test, there are two images that shows the initial BC positioning and the final BC positioning.

5.1.1 Controlled Battle Tests

As it was said before, in these battles both players use the same tactic and troops. The tactic used in these battles is the Simple Frontal Assault which has only one line assault with flanks and all the troops rush for a meele attack. Both sides have the same number of Militiamen, Bowmen, Skirmishers and Light Lancers. When looking at the images below, it seems that the battles were unbalanced, even though both players had the same tactics and troops. But there are some reasons that made the battles look that way, in the initial steps of a battle, the BCs have the same type of behaviour but when they collide with each other the BC positioning may be different which leads to a different behaviour also if a BC from the blue side gets blocked by another and the same BC from the red side did not, it will make them to behave differently. The other reason that makes the battles look unbalanced is when 70% of the army is dead or routed, the remaining BCs will run from the battlefield.

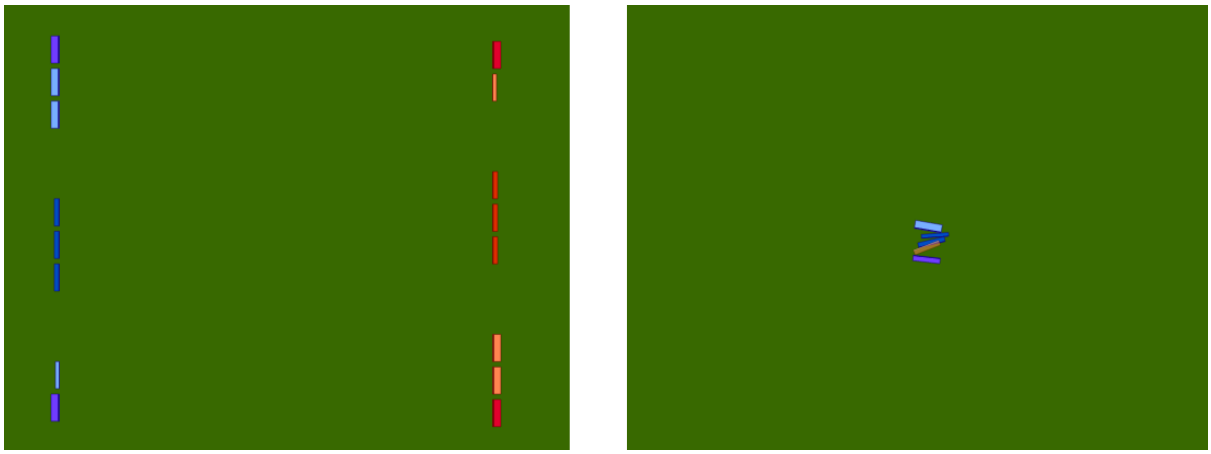


Figure 5.1: This battle starts with 8 BCs on the blue side and 8 BCs on the red side making a total of 16 BCs.

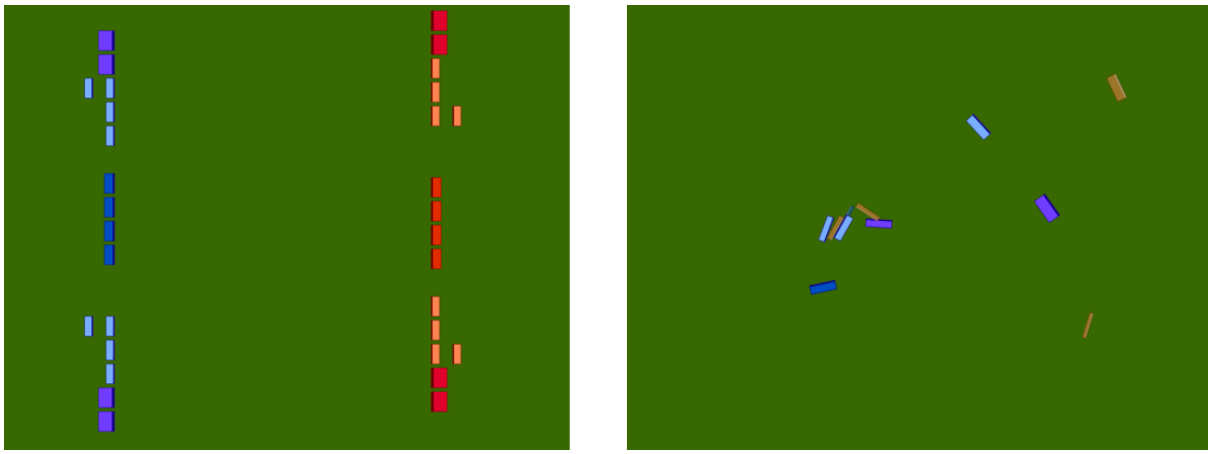


Figure 5.2: This battle starts with 16 BCs on the blue side and 16 BCs on the red side making a total of 32 BCs.

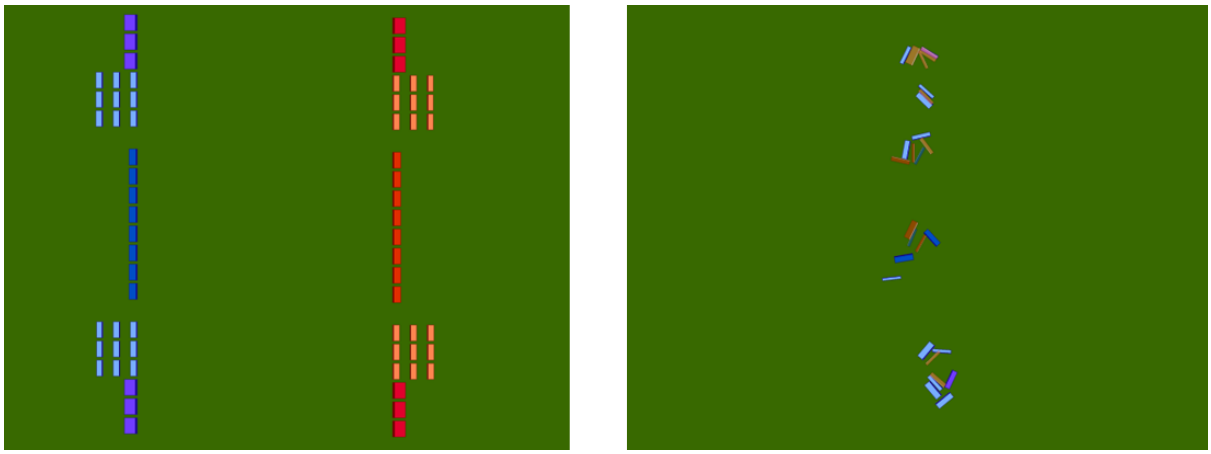


Figure 5.3: This battle starts with 32 BCs on the blue side and 32 BCs on the red side making a total of 64 BCs.

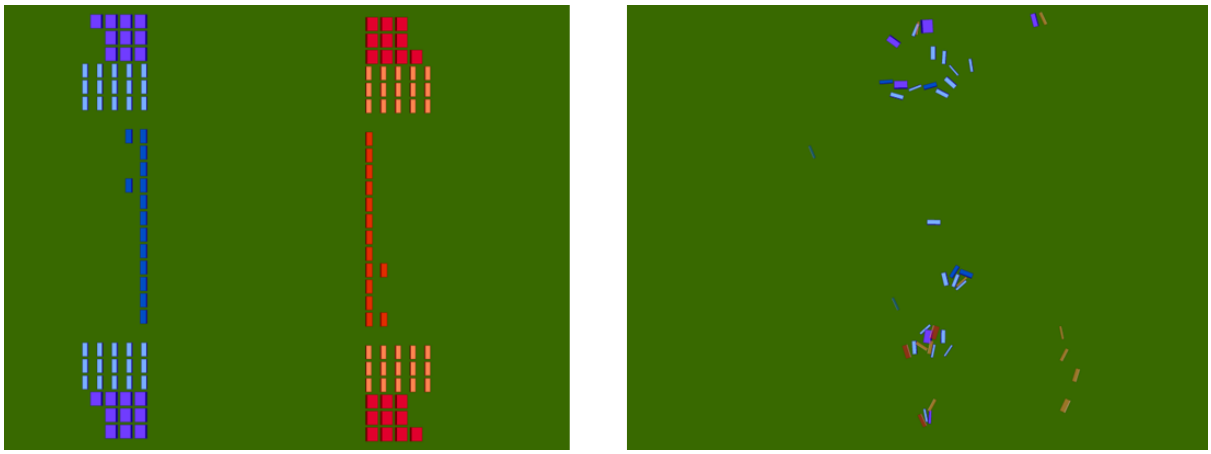


Figure 5.4: This battle starts with 64 BCs on the blue side and 64 BCs on the red side making a total of 128 BCs.

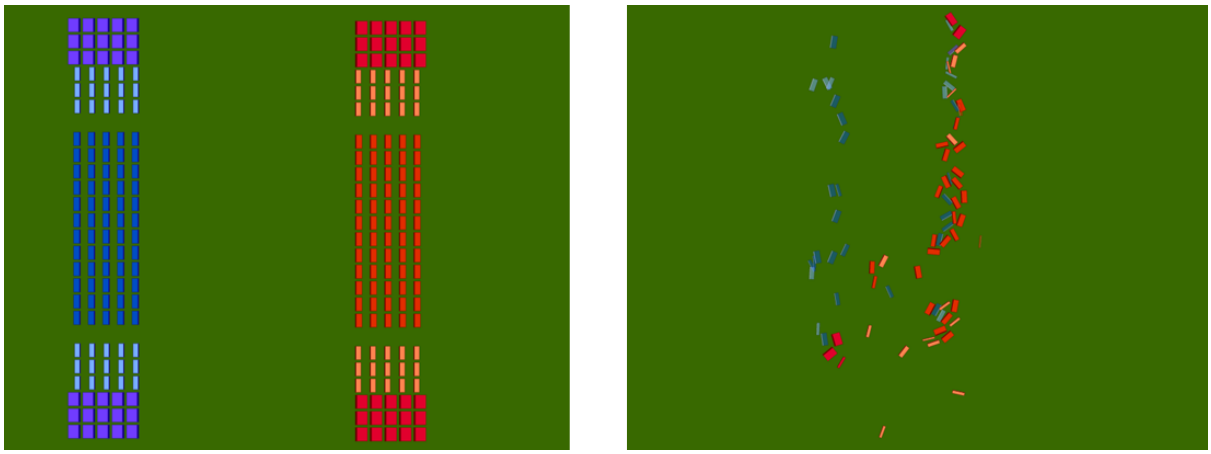


Figure 5.5: This battle starts with 120 BCs on the blue side and 120 BCs on the red side making a total of 240 BCs.

5.1.2 Random Battle Tests

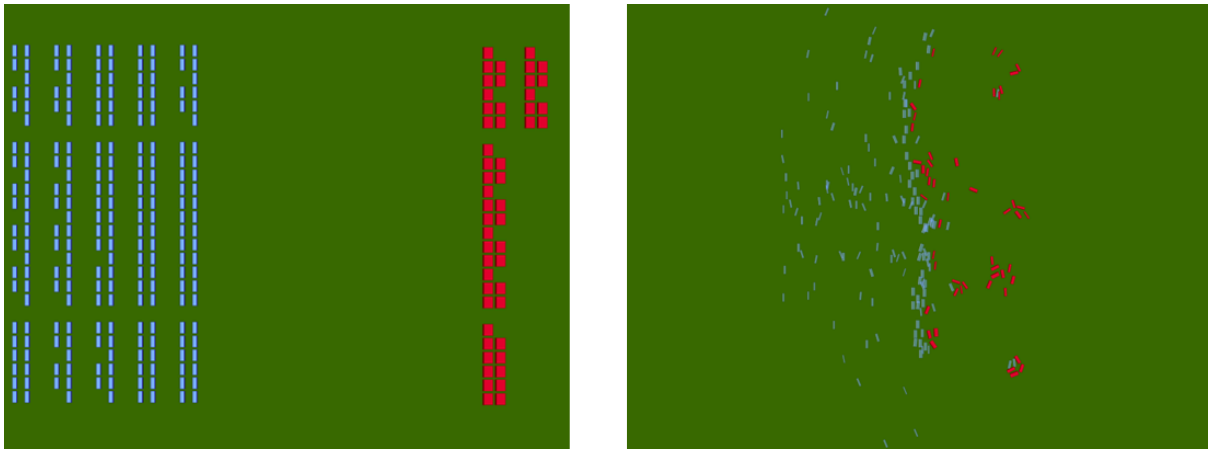


Figure 5.6: Both images belong to the battle with id 2507. This battle starts with 221 BCs on the blue side and 51 BCs on the red side making a total of 272 BCs.

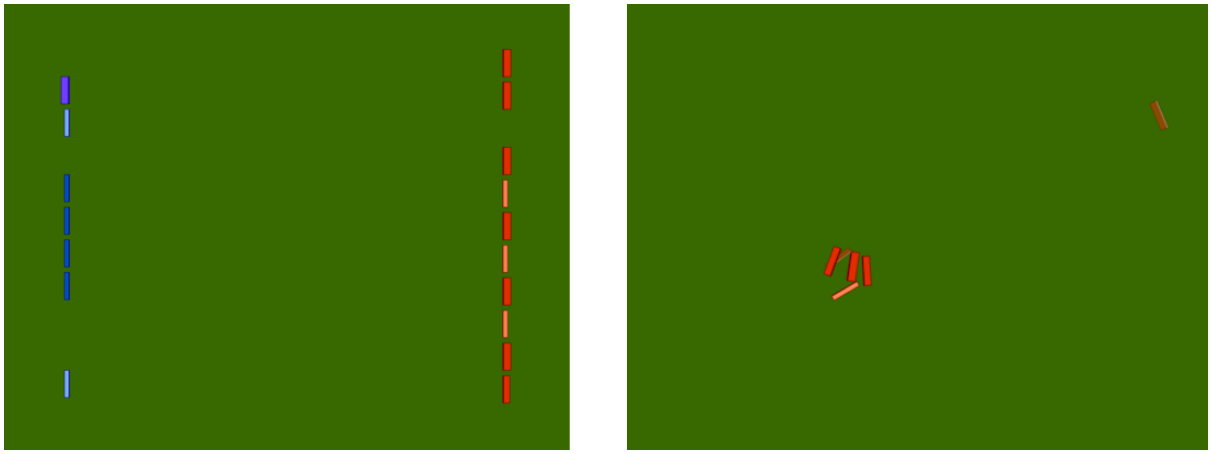


Figure 5.7: Both images belong to the battle with id 7696. This battle starts with 7 BCs on the blue side and 10 BCs on the red side making a total of 17 BCs.

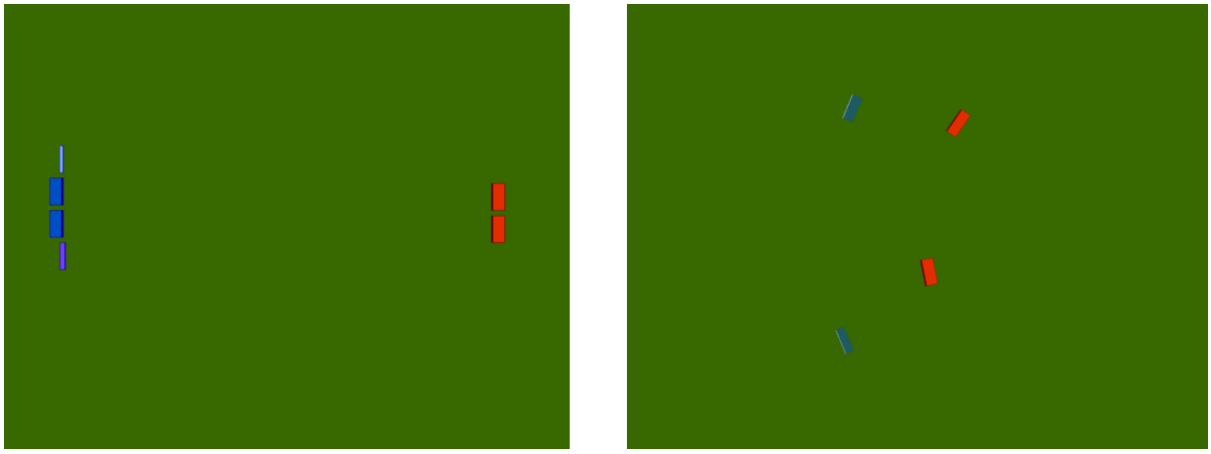


Figure 5.8: Both images belong to the battle with id 7697. This battle starts with 4 BCs on the blue side and 2 BCs on the red side making a total of 6 BCs.

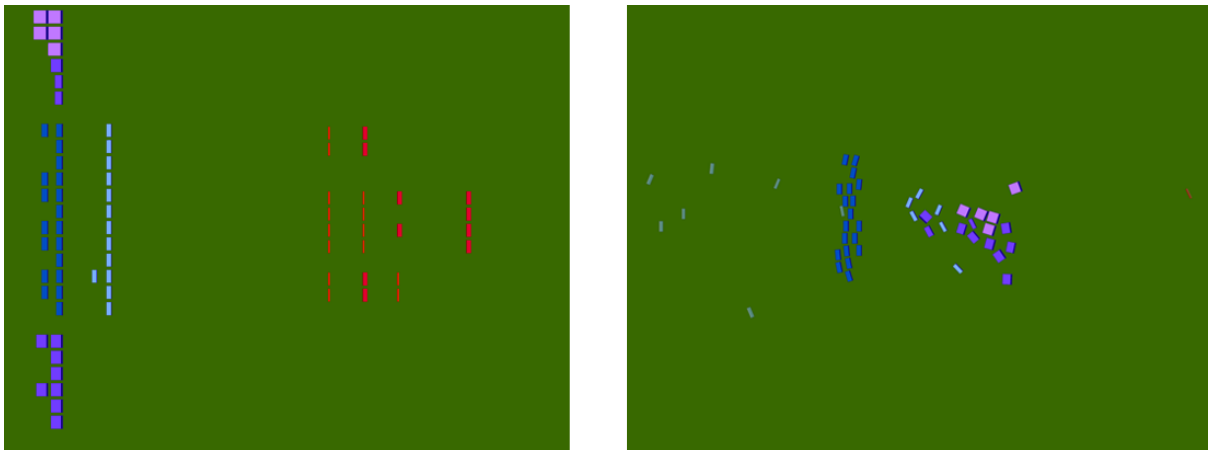


Figure 5.9: Both images belong to the battle with id 7698. This battle starts with 48 BCs on the blue side and 27 BCs on the red side making a total of 72 BCs.

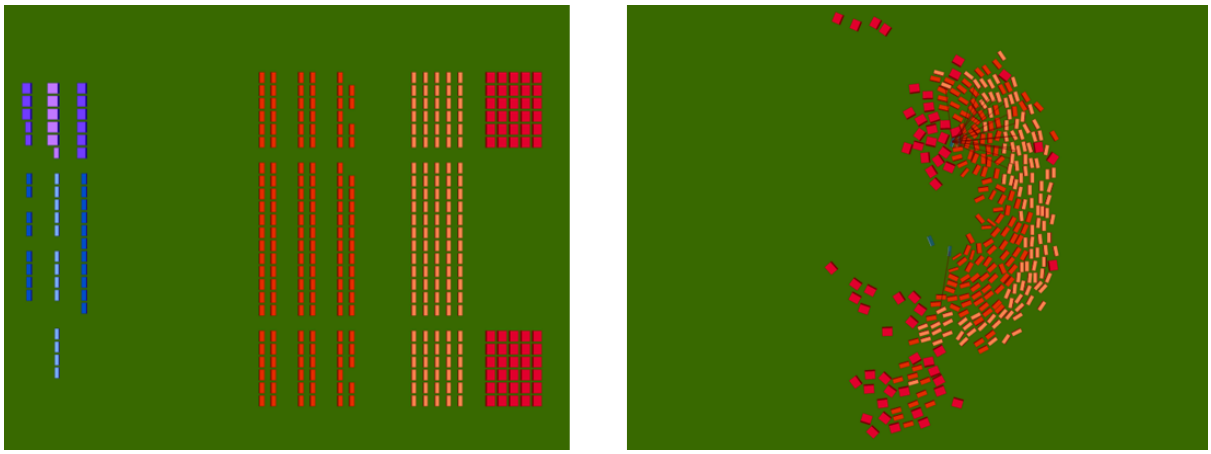


Figure 5.10: Both images belong to the battle with id 7699. This battle starts with 49 BCs on the blue side and 320 BCs on the red side making a total of 369 BCs.

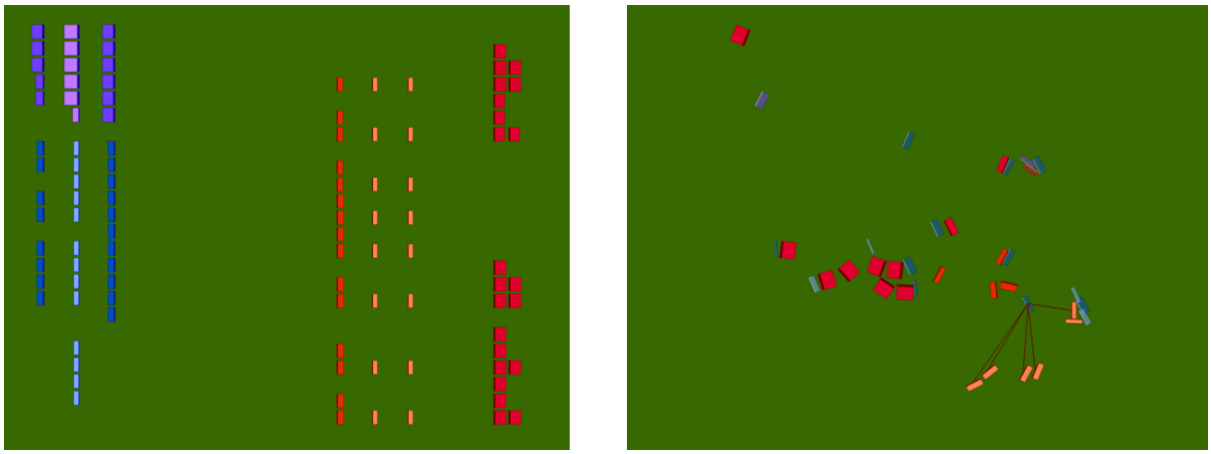


Figure 5.11: Both images belong to the battle with id 7700. This battle starts with 49 BCs on the blue side and 53 BCs on the red side making a total of 102 BCs.

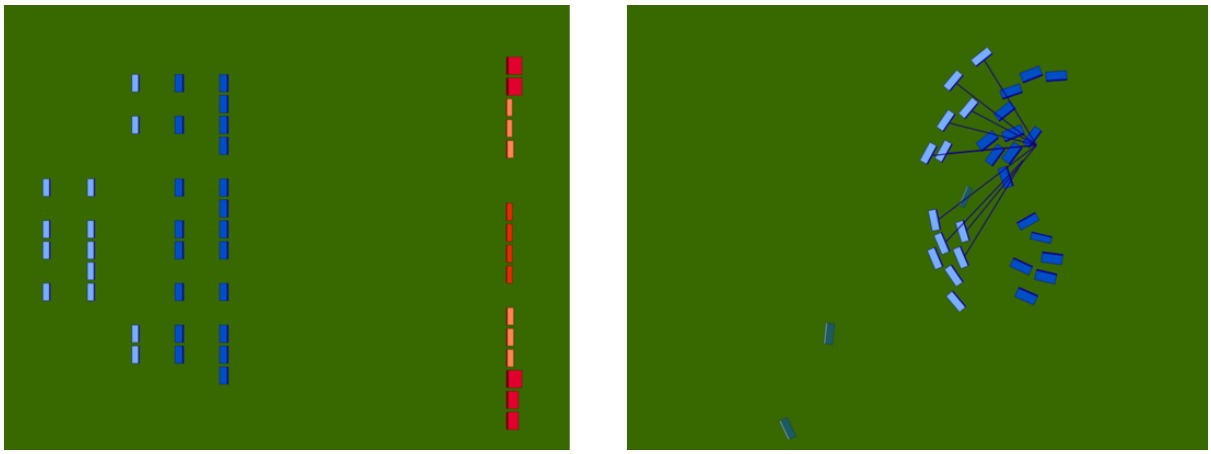


Figure 5.12: Both images belong to the battle with id 7701. This battle starts with 33 BCs on the blue side and 15 BCs on the red side making a total of 48 BCs.

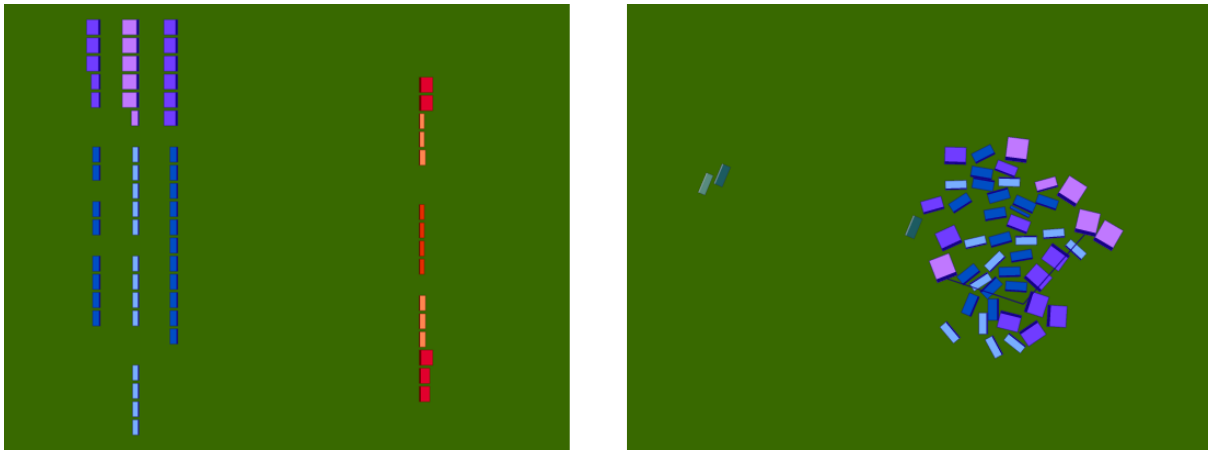


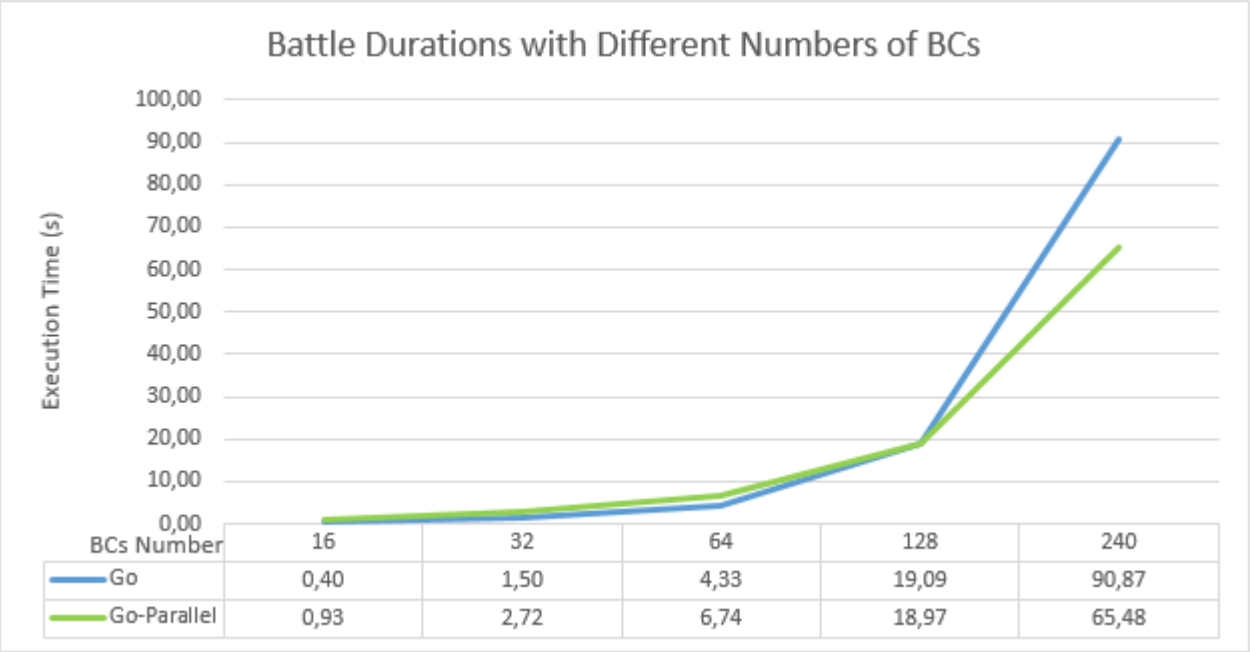
Figure 5.13: Both images belong to the battle with id 7702. This battle starts with 49 BCs on the blue side and 15 BCs on the red side making a total of 64 BCs.

5.2 Tests Results

The controlled tests were created to compare the performance of Go and Go-Parallel versions when testing battles with the same characteristics but with different numbers of BCs while the random tests were used to compare the performance of JavaScript, Go and Go-Parallel versions with different kinds of battles.

5.2.1 Controled Battle Tests Results

After running these tests, it was possible to create a graphic showing the execution times for the five battles and also showing the battle duration variation between them regarding the number of BCs.



BCs Number	16	32	64	128	240
Go	0,40	1,50	4,33	19,09	90,87
Go-Parallel	0,93	2,72	6,74	18,97	65,48

Figure 5.14: Graphic showing the performance from Go and Go-Parallel versions for the same battle but with different numbers of BCs.

The graphic above shows two important aspects, the first aspect is that for the same battle characteristics but with different numbers of BCs the execution times from both versions increases and the second aspect is that the Go version was the fastest executing battles containing less than 128 BCs while the Go-Parallel was the fastest executing battles containing 128 or more BCs. Overall, the Go version was up to 57% faster than the Go-Parallel in battles with less than 128 BCs version while the Go-Parallel version was up to 27% faster than Go version in battles with 128 BCs or more.

The first aspect showed that increasing the number of BCs it will affect directly the battle duration which is expected because when the number of BCs is increased the time to process a battle step is also

increased and consequently increases the overall battle duration.

The second aspect showed that the Go-Parallel version when dealing with short battles duration and with a short number of BCs to be slower than the Go version and this happens because of the overhead added when using with parallelism but, on the other hand, when dealing with long battles durations and large numbers of BCs the parallelism pays off.

Another graphic was created but this time using the execution times and the number of steps from each battle to calculate the average time spent from each battle step in Go and Go-Parallel versions.

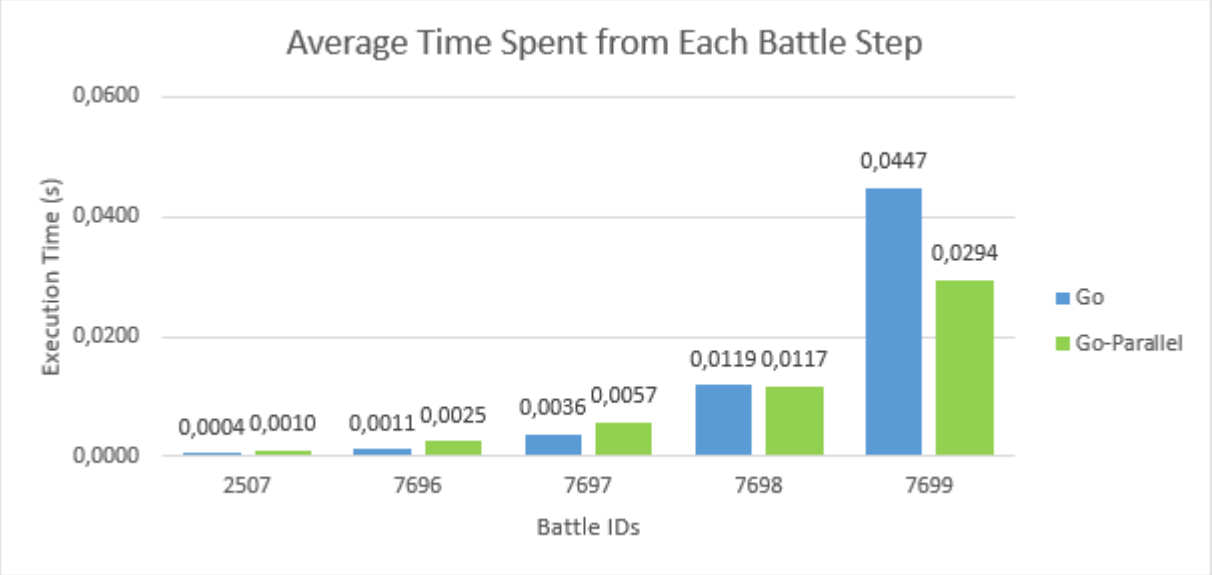


Figure 5.15: Graphic showing the average time spent, in seconds, from each battle step in Go and Go-Parallel versions using the same battle tests as above.

The graphic above shows the same result as the previous one which was expected because the number of BCs has a direct impact in the battle's execution time, in other words, if the number of BCs increases, the battle's execution time also increases.

5.2.2 Random Battle Tests Results

From the selected battle tests it was possible to create two graphics, one for short duration battles and the other for long duration battles, showing the differences of the execution times between the JavaScript, Go, Go-Parallel versions.

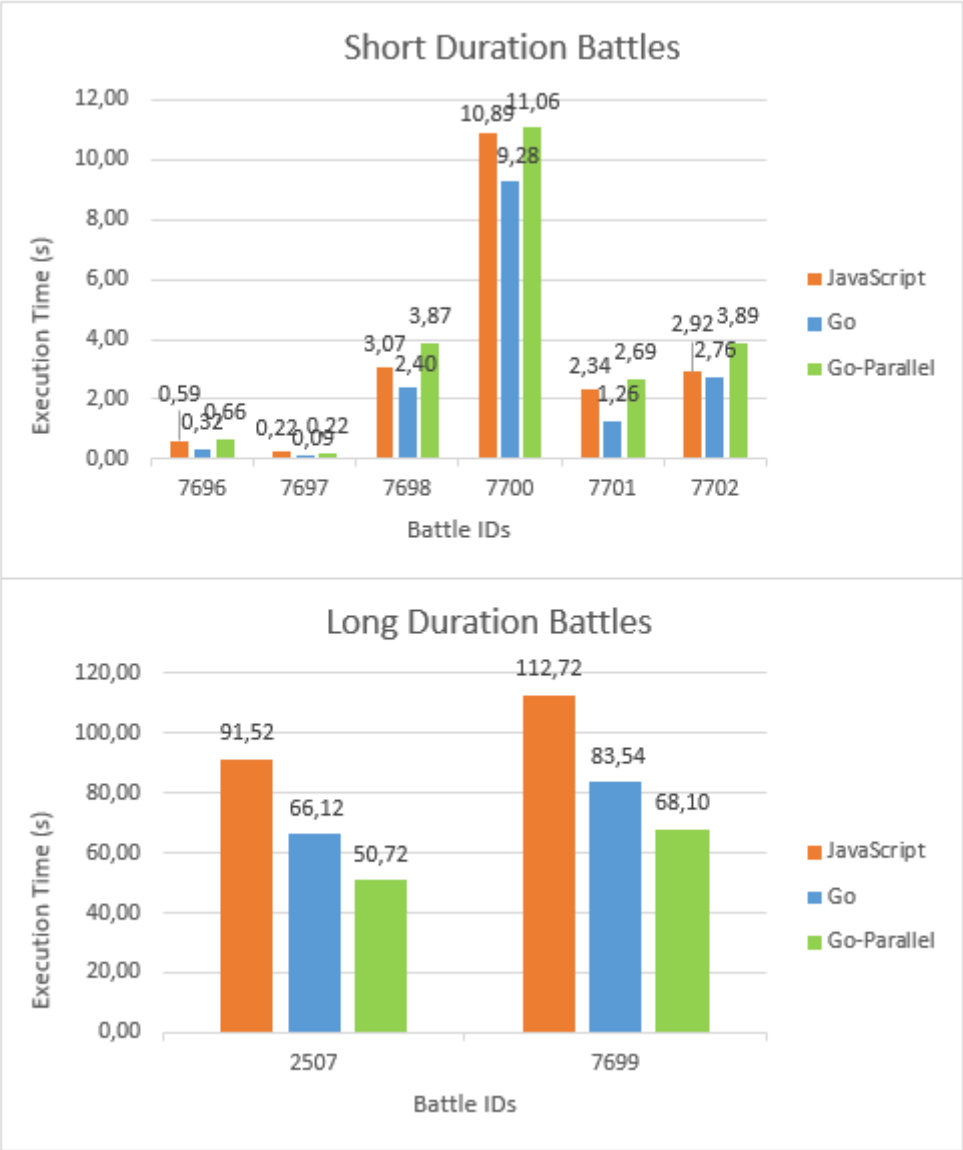


Figure 5.16: Graphics showing the performance from JavaScript, Go and Go-Parallel versions in short and long duration battles.

In the short duration battles, the graphic shows that the Go version was faster than the other versions showing a performance improvement between some milliseconds, visible in the battle 7697, and a couple of seconds, visible in the battle 7700, while the Go-Parallel version was the slowest version. In the long duration battles, the Go-Parallel version was the fastest version showing a performance improvement greater than fifteen seconds.

In both graphics, when comparing the Go and JavaScript performance it is possible to see that the Go version had the best performance in all the battle tests, with a performance improvement up to 46%. This result shows that the new language offers a better performance than the original language. The Go-Parallel version is slow in short duration battles because of the overhead added when using parallelism and the performance improvements from the parallelism are only visible in long duration battles,

with improvements up to 23% when comparing with the Go version.

The next graphic is similar to the graphic from the previous section where it uses the execution times from the battle tests and the number of steps from each battle to calculate the average time spent from each battle step in all the versions of the battle simulator.

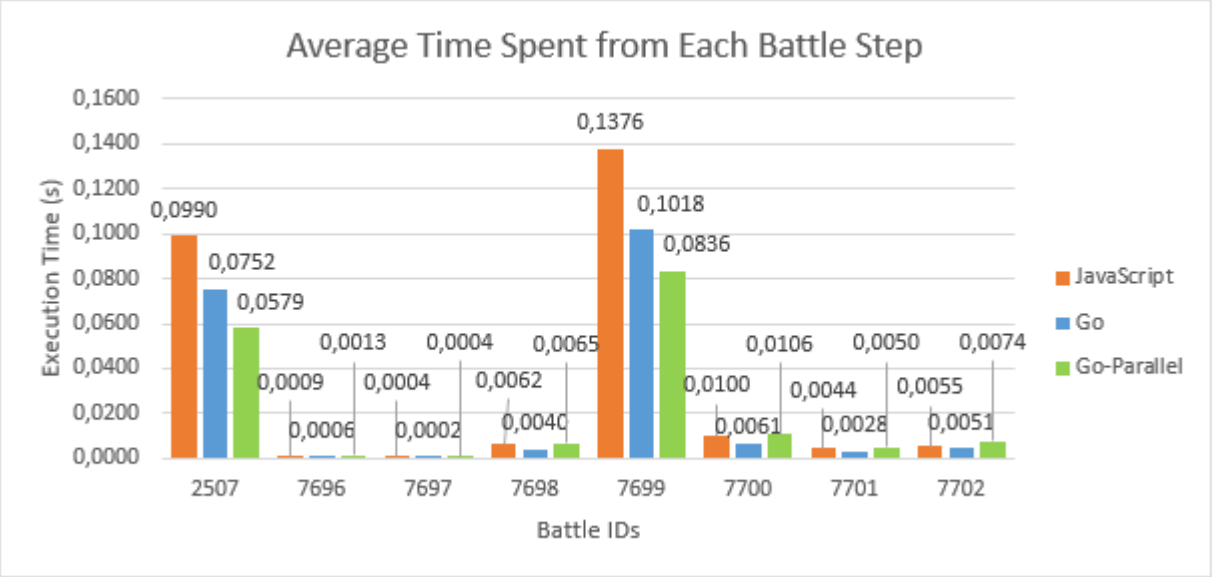


Figure 5.17: Graphic showing the average time spent, in seconds, from each battle step in JavaScript, Go and Go-Parallel versions using the same battle tests as above.

Comparing this graphic to the previous one, it is possible to see that the results are the same because, as it was said before, the execution time is dependent on the number of BCs in a battle.

5.3 Go Profiling

After analyzing the results from the tests, it is time to have a look at the Go profiling to see where is the time spent during the code execution.

```
Self   | Total    | Function
-----------------------------------------------------------------
8.7%     18.9%      runtime.mallocgc
7.9%     40.9%      models.(*BattleContingent).target_weight
4.9%      4.9%      runtime.aeshash64
4.7%      7.3%      runtime.mapaccess2_fast64
-----------------------------------------------------------------
4.6%      4.6%      sort.(*IntSlice).Less
4.5%     12.4%      hash_insert
4.4%      4.4%      runtime.memclr
4.3%      4.3%      sweepspan
-----------------------------------------------------------------
4.2%      4.2%      scanblock
3.3%      3.3%      models.(*SDistList).Swap
3.2%      4.6%      evacuate
2.9%      2.9%      runtime.markallocated
-----------------------------------------------------------------
2.2%      2.2%      models.(*SDistList).Less
2.1%      2.1%      flushptrbuf
2.0%      2.0%      models.line_intersection
1.9%      1.9%      hash_next
-----------------------------------------------------------------
1.6%     10.6%      models.smallest_distances
1.4%      9.4%      sort.doPivot
1.3%     12.0%      models.(*Point).linedistance
```

Figure 5.18: Performance profile from the Go battle simulator.

Looking at the profiling results above, it is possible to see that, unlike the JavaScript profiling, there are some Go internal functions using up some time during the code execution such as the malloc function that uses 8.7% of the execution time. Another difference between this profiling and the JavaScript profiling is the fact that the line_intersection function does not spend as much time as it would in the JavaScript version which is an unexpected result because in JavaScript this function took 43% of the execution time while in Go it only took 2%.

5.4 Discussion

Having a final look at the results, it is possible to conclude that the Go version has a better performance than the original version in JavaScript in all the tests made. Another conclusion is that the Go version outperforms the Go-Parallel version when executing short duration battles with a small number of BCs, on the other hand, the Go-Parallel has a better performance executing long duration battles with a large number of BCs.

Despite having battles over one minute even with parallel programming, overall these tests showed a good performance from the new programming language which was one the main goals from this thesis but it also shows that these versions solve some performance issues regarding the JavaScript version, but the Go-Parallel version still needs some improvements in order to execute the battles as fast as possible until it achieves the desirable performance.

Chapter 6

Conclusions and Future Work

This document tried to include all the aspects related to the battle simulator module from “My Army”, explaining all the decisions made as well as the issues found along the way.

Before starting developing, the first stage was to study the original battle simulator in order to understand how the battle simulation worked and also to detect any kind of issues or improvements. At first, this sounded like an easy and fast task to do, but in reality it took much more time than it was expected because there were a lot of hidden details that it was need to understand beyond the functionality of the simulator, but in the end everything went well.

After understanding all the details and issues related to the simulator it was time to create a solution to achieve the main goals. The final solution was divided into two steps. The first step was to change the programming language from the battle simulator from JavaScript to Go and the second step was to insert parallelism in the Go version in order to get the best performance as possible.

The development process was long and with a lot of obstacles to overtake, but with a lot of effort the Go version was completed. Later on and with some struggle, the Go-Parallel version was created. In the end, the final result was two battle simulator versions, both developed in Go, where one version contains the same functionality as the initial version of the battle simulator developed in JavaScript and another version that uses parallel programming.

With the new developed versions it was time to test them and to create a comparative analysis that helped to take some conclusions. The comparative analysis showed that the Go version had a better performance than the JavaScript in any of the battle tests used which showed that the new language could outperform the original language and it also showed that the parallelism produced worse results in short duration battles than in long duration battles.

Despite the performance improvements obtained from Go and Go-Parallel versions, there is still some

work to be done. The main problem with this game is the lack of scalability because as the user keeps playing his army tends to grow and consequently the number of BCs increases and with the current performance from the Go-Parallel is not enough to run battles below the one minute which was one of the goals of this thesis.

After all this process there are still some improvements that could be made to the parallel version of the battle simulator as future work, such as:

- **Move the parallelism to the quadtree:** Everytime a BC updates its position, it is removed from the quadtree and inserted again with the current position and this process takes some time. So a viable solution is to parallelize the current quadtree in order to make the nodes exploration faster;
- **Split the BCs list into four quadrants list:** Basically instead of running a single list of BCs from each player, the idea behind this is to create four BCs lists, one for each quadrant, and then run them in parallel. Theoretically, this should improve the performance of the parallel version;
- **Insert a maximum battle duration or limit the number of troops/BCs per battle:** Since this is a real-time game and in case none of the previous suggestions helped, then the only solution is to supress the number of troops/BCs in a battle or to insert a maximum battle duration which will avoid all the performance issues.

Finally, it is possible to conclude that most of the main goals of this thesis were fulfilled except for the one stating that battles must execute below one minute.

Bibliography

[1] Boids - Wikipedia, the free encyclopedia. Consultado em 15/06/2015. https://en.wikipedia.org/wiki/Boids

[2] Bill Lewis: Threads Primer: A Guide to Multithreaded Programming, Prentice Hall

[3] Cluster Node.js v4.1.1 Manual & Documentation. Consultado em 28/09/2015. https://nodejs.org/api/cluster.html

[4] Cormen, Thomas H.; Leiserson, Charles E., Rivest, Ronald L., Stein, Clifford (2009) [1990]. Introduction to Algorithms (3rd ed.). MIT Press and McGraw-Hill.

[5] Craig W. Reynolds. Flocks, Herds, and Schools: A Distributed Behavioural Model. http://www.red3d.com/cwr/papers/1987/SIGGRAPH87.pdf, 1987

[6] Dynamic programming language. Consultado em 30/12/2014. http://en.wikipedia.org/wiki/Dynamic_programming_language

[7] Edward A. Lee. The Problem with Threads. http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-1.pdf, 2006

[8] Effective Go - The Go Programming - Concurrency. Consultado em 21/11/2014. https://golang.org/doc/effective_go.html#concurrency

[9] Emergence - Wikipedia, the free encyclopedia. Consultado em 15/06/2015. http://en.wikipedia.org/wiki/Emergence

[10] Go (programming language). Consultado em 21/11/2014. http://en.wikipedia.org/wiki/Go_(programming_language)

[11] Going Go Programming: Concurrency, Goroutines and GOMAXPROCS. Consultado em 20/03/2015. http://www.goinggo.net/2014/01/concurrency-goroutines-and-gomaxprocs.html

[12] Goodman, Danny; Eich, Brendan (2001). JavaScript Bible. John Wiley & Sons.

[13] Google pits C++ against Java, Scala, and Go. Consultado em 11/12/2014. http://www.theregister.co.uk/2011/06/03/google_paper_on_cplusplus_java_scala_go/, 2011

[14] Google's Go Programming Language: Taking Cloud Development By Storm. Consultado em 26/11/2014. http://readwrite.com/2014/03/21/google-go-golang-programming-language-cloud-development, 2014

[15] Husted, Robert; Kuslich, JJ (1999). Server-Side JavaScript: Developing Integrated Web Applications (1st ed.). Addison-Wesley.

[16] JavaScript (programming language). Consultado em 30/12/2014. http://en.wikipedia.org/wiki/JavaScript

[17] Javascript Array.sort implementation - Stack Overflow. Consultado em 25/05/2015. http://stackoverflow.com/questions/234683/javascript-array-sort-implementation

[18] João Morais. My Army: A Game for Social Networks, 2011

[19] Marius Wolfensberger. Improving the Performance of Region Quadtrees. www.ifi.uzh.ch/dbtg/teaching/thesesarch/ReportWolfensbergerFA.pdf, 2013

[20] Nodejs cosine problem. Consultado em 4/07/2015. http://stackoverflow.com/questions/24455775/why-does-node-not-evaluate-math-tanmath-pi-2-to-infinity-but-chrome-v8-does

[21] Node.js vs Go 2014. Consultado em 26/11/2014. https://jaxbot.me/articles/node-vs-go-2014, 2014

[22] Parallel For-Loop - Go Language Patterns. Consultado em 21/03/2015. http://www.golangpatterns.info/concurrency/parallel-for-loop

[23] Parallel programming. Consultado em 29/12/2014. http://en.wikipedia.org/wiki/Parallel_programming_model

[24] Parallel.js: Parallel computing with Javascript. Consultado em 24/06/2015. http://adambom.github.io/parallel.js/

[25] Parallelism and concurrency need different tools. Consultado em 11/12/2014. http://yosefk.com/blog/parallelism-and-concurrency-need-different-tools.html, 2013

[26] Peiyi Tang. Multi-Core Parallel Programming in Go. http://www.ualr.edu/pxtang/papers/acc10.pdf

[27] Quadtrees and Octtrees. Consultado em 12/06/2015. http://http.cs.berkeley.edu/~demmel/cs267/lecture26/lecture26.html

[28] Quicksort - Wikipedia, the free encyclopedia. Consultado em 9/06/2015. https://en.wikipedia.org/wiki/Quicksort

[29] Robert Hundt. Loop Recognition in C++/Java/Go/Scala. https://days2011.scala-lang.org/sites/days2011/files/ws3-1-Hundt.pdf, 2011

[30] src/sort/sort.go - The Go Programming Language. Consultado em 9/06/2015. https://golang.org/src/sort/sort.go

[31] Static Type system. Consultado em 21/11/2014. http://en.wikipedia.org/wiki/Type_system#Static_type-checking

[32] Systems Programming with Go, Rust, and ParaSail. Consultado em 11/12/2014. http://parasail-programming-language.blogspot.pt/2013/04/

systems-programming-with-go-rust-and.html, 2013

[33] The Basics of Web Workers - HTML5 Rocks. Consultado em 24/06/2015. http://www.html5rocks.com/en/tutorials/workers/basics/

[34] The Go scheduler - Morsing's blog. Consultado em 8/05/2015. https://morsmachine.dk/go-scheduler

[35] Thread (computing). Consultado em 29/12/2014. http://en.wikipedia.org/wiki/Thread_(computing)

[36] What is JavaScript. Consultado em 30/12/2014. http://media.wiley.com/product_data/excerpt/88/07645790/0764579088.pdf

[37] Why Go Is Not Good::Will Yager. Consultado em 26/11/2014. http://yager.io/programming/go.html