Degree Final Project

Platform for massive multiplayer programming games

Computing

Héctor Ramón Jiménez

Advisor
Jordi Petit Silvestre

Facultat d'Informàtica de Barcelona
Universitat Politècnica de Catalunya

January 22, 2017

Abstract

This project states that a platform for multiplayer programming games with a high number of players and long matches is needed. It provides a design of an open-source solution to satisfy this necessity and describes its implementation using a methodology based on continuous integration. Finally, the implemented solution is evaluated, concluding that it is a good candidate for such a platform.

Contents

1 Introduction

This chapter reviews the history of programming games, states the personal motivation of the author, and details the structure of this report.

1.1 Brief history of programming games

1.1.1 Playing games while programming

In 1961, Victor Vyssotsky, a mathematician and computer scientist working at Bell Labs, along with Robert Morris Sr. and Doug McIlroy, created Darwin [24]: the **first programming game**. This game was not played by a player with a controller. Instead, Darwin could only be played by writing a computer program.

A *programming game* is a computer game where the player does not directly interact with the game. Instead, the player writes a computer program that plays the game. These computer programs are usually called artifficial intelligences (`AIs`) because they try to make intelligent decisions to win the game.

Darwin consisted of two or more small programs, written by the players, that were loaded in memory. The main goal of the game was to spread copies of your own program and find and kill the copies of other players. The game was only played for a few weeks. Then, Morris developed a program that no-one managed to defeat.

Since then, many other programming games have been created [33]. Some of them are even commercial games, like SpaceChem [34].

1.1.2 Playing with other people

With the arrival of the Internet and the World Wide Web, there was nothing stopping people from developing web platforms for multiplayer programming games.

A *multiplayer programming game* is a programming game where multiple players compete with each other to win the game. Thus, the game becomes a challenge where strategy and programming skills make the difference.

These web platforms allow players to compete with each other easily. For example, Robot Game [32] is a website where anyone can upload an `AI` written in Python and compete with other people.

1.1.3 Playing while learning

Writing `AIs` can be a really fun and rewarding experience because the game allows the players to see how their algorithms work visually, while competition motivates them to learn and improve.

It comes as no surprise, then, that programming games are being used in schools, as gamification [3], to teach students different programming techniques. For instance, an `AI` programming challenge is held every semester in the Barcelona School of Informatics (FIB) where students enrolled in the subject Data Structures and Algorithms (EDA) [2] compete with each other in a multiplayer programming game using the Jutge.org platform [29].

1.2 Personal motivation

I love videogames. Ever since my father introduced me to my first computer when I was 3 years old. I was immediately hooked. I started playing simple puzzle games, while discovering first person shooters and strategic games soon after.

Videogames were the main reason I chose to study computer science. I learned my first programming language because I wanted to open a website to share my passion about videogames. I was 10 years old back then. Programming is a really important facet of my life.

Studying computer science has made me love videogames even more. When I a see videogame now, I can try to imagine the logic behind it. I can try to picture the different algorithms involved. I imagine thousands of bits correctly aligned, flowing and changing constantly, while they follow some complex logic. For me, the fact that a videogame is able to show how its code

works visually is truly fascinating.

Programming games mix two of my passions: videogames and programming. So when this project was offered to me, I thought it was the **perfect fit**.

1.3 Report structure

This report consists of four different parts. Each one of them describes a vital phase in the development of the project:

Formulation: Identifies and analyzes the problem to solve, it specifies the scope of the project, and it shows the design of the solution.

Planning: Details the time plan and the budget to develop the project.

Implementation: Describes the development of the different components that compose the solution.

Evaluation: Describes the validation methodology, it reviews the time management and economic cost, and it discusses the sustainability and legality issues of the project.

Part I

Formulation

2 Analysis

This chapter details the problem that this project aims to solve and it decides whether there is some existing solution that may apply to solve this problem.

2.1 The requirement

The advisor of this project is one of the founders of the Jutge.org platform. Also, he is deeply involved in the `AI` programming challenge organized in the `EDA` subject in the FIB.

Currently, the contest strategy is based on a round system. All the students play at least one match per round. Players that lose a match play against other losers. At the end, the player that loses the last match is eliminated from the contest.

Clearly, this strategy has one main drawback: eliminated students stop playing. Hence, their learning process halts completely. This could be solved using a simple score system, but it would still be patching the main issue: the programming games used do not support a huge amount of players natively.

The aim of this project is to provide an easy way to develop multiplayer programming games featuring huge worlds, long matches and a massive amount of players in real-time. This will make players feel attached to the match, programs will need to adapt constantly as they play with everyone at the same time. As a consequence, this project will finally allow to improve the strategy used in the `EDA` progamming challenge.

2.2 State of the art

There are many platforms that offer `AI` programming challenges. Some examples are:

Google's AI programming challenge [27]: The university of Waterloo organized some AI programming challenges sponsored by Google during 2010-2011.

Battlecode [25]: A website that organizes an AI programming challenge every year where anyone can compete alone or in a team.

CodinGame [8]: A website that has many AI programming contests, which can help players to apply for specific jobs.

EDA competition [2]: An AI programming challenge is held every semester in the Barcelona School of Informatics (FIB) where students compete with each other using the Jutge.org platform [29].

However, all of these platforms feature multiplayer programming games with short matches played by a small number of players. As a consequence, multiple matches with different players are necessary to decide who wrote the best AI.

Given that there is no pre-made solution that fits the project's requirement, a new brand solution is necessary.

2.3 Stakeholders

Given the requirement described in section 2.1, the different users that will use the platform are considered:

Game programmers: They want to create MMPGs easily. They want to focus on programming the game logic and the game viewer, without worrying about internal aspects of the platform.

Players: They will develop AIs for a concrete game and upload them to the web platform at any time during the game match.

Viewers: They want to watch the match unfold in real-time.

Administrators: They want to control the game. They want to supervise, start, stop or pause the game, and obtain the final scores of every player.

3 Objectives

This chapter details the different objectives that the project must fullfill in order to satisfy the requirement stated in the previous chapter.

3.1 Main objective

Develop a set of components that ease the creation and the usage of massive multiplayer programming games.

3.2 Secondary objectives

Develop an abstract game engine: Any experienced programmer must be able to create new games for the platform.

Allow hot-swapping of `AIs`: Players must be able to change the code of their current `AIs` in the middle of a match.

Implement a real-time webviewer: Players must be able to watch in real-time how the game unfolds in a web browser.

Create a control panel: Administrators must be able to play, pause and stop the game and also see a ranking of the current match.

Make the infrastructure scalable and stable: The underlying infrastructure must be able to handle huge worlds and a large number of AIs without hindering performance. Moreover, the platform must be secure and fault-tolerant; `AIs` must not be able to cheat or affect the platform negatively.

Create a game example: A multiplayer programming game will be created to test and show that the platform works correctly.

4 Design

The design of the platform has to allow game developers to implement MMPGs easily. Hence, it is necessary to keep the number of components and dependencies at bay.

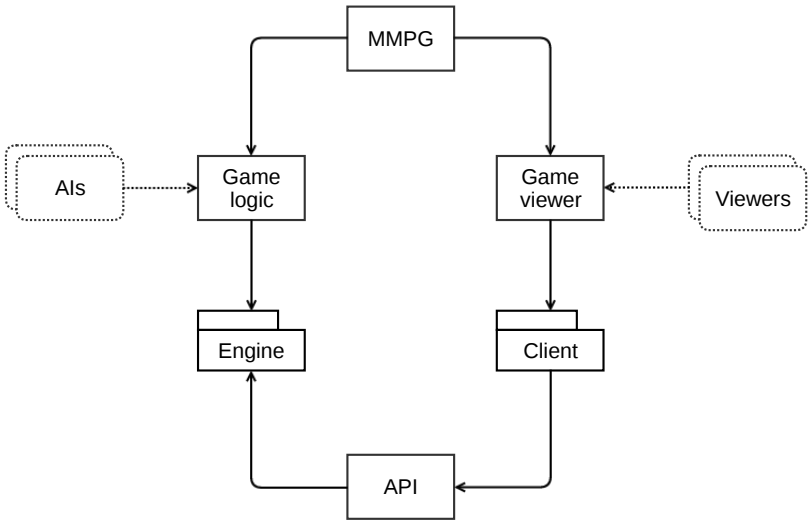

Figure 4.1: The design of an MMPG

Figure 4.1 shows a diagram of the different components that compose an MMPG and their dependencies. As it is shown, game developers only need to implement two components to create an MMPG: the game logic and the game viewer. This is quite logical, since these are the only two components that are game-specific.

Hence, a massive multiplayer programming game using the platform consists of:

Engine: A library that implements basic features needed by any MMPG. The en-

gine exposes a set of classes that must be extended to build the game logic.

`API`: A component that exposes `HTTP` endpoints that allow to interact with an underlying engine. It usually handles requests from a game viewer using the client library.

Client: A library that implements a set of useful classes to communicate with an `API` and implement game viewers.

Game logic: It includes the game world definition and the rules of the game and executes the different `AIs`. It is built on top of the engine.

Game viewer: It allows viewers to watch game matches. Also, it makes players able to upload new `AIs`. It uses the client library to connect to the `API`.

5 License

5.1 Code

The MMPG platform code and the game example will be released under The MIT License [5].

5.2 Documents

This monitoring report and the final document related with the MMPG platform hosted in https://github.com/hecrj/mmpg will be released under the Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International [1] license.

Part II

Planning

6 Time plan

6.1 Estimated project duration

The project starts the September 15th, 2015 and its deadline is the January 25th, 2016. Thus, the project will take approximately 4 months to be completed.

6.2 Tasks

This section details the different tasks and subtasks that need to be completed in order to finish the project successfully. Every task is provided with a description, an expected duration, possible complications and task dependencies.

6.2.1 Project management course

The project management course aims to help lead the project in the right direction. In this task, different parts of the project are defined: context and scope, temporal planning, and economic viability.

Expected duration: 75 hours.

Possible complications: None. The course is entirely guided and feedback is regularly provided to ensure the author can finish it properly.

Task dependencies: None.

6.2.2 Analysis and design

In this task, the project requirements are analyzed and a solution is designed to satisfy them.

Expected duration: 10 hours.

Possible complications: Some minor details might change during the implementation. It is important to focus on the *big picture* of the project.

Task dependencies: None.

6.2.3 Engine

A library that will allow game programmers to build games and wire them to the platform. The codebase will implement the features that are independent of the final game and, therefore, can be reused.

Expected duration: 70 hours

Possible complications: `AI` memory management might be difficult. The author may need to learn low-level instructions to control OS's processes.

6.2.3.1 API

A web-server that notifies its subscribers of the changes that occur in the game world.

Expected duration: 50 hours

Possible complications: The author needs to learn `Go`, which might take some time.

6.2.4 Client

A library that handles a connection with the `API`. Game programmers will be able to use it to build the real-time webviewers of their games. It will be programmed in Javascript, using `Coffeescript`.

Expected duration: 30 hours

Possible complications: The author needs to learn `Coffeescript`, which might take some time.

6.2.5 Control panel

Allows administrators to supervise the games, start, stop and pause current matches, and obtain the scores of every player.

Expected duration: 35 hours

Possible complications: None.

6.2.6 Game example

A massive multiplayer programming game will be developed to test that the underlying engine and infrastructure work as intended. This task will be splitted in other two: logic and viewer.

6.2.6.1 Logic

During this task, the main logic of the game will be developed on top of the engine. This includes the different game rules and the game world.

Expected duration: 50 hours

Possible complications: Balancing the game could be really hard.

Task dependencies: Engine.

6.2.6.2 Viewer

In this task, a web-viewer that shows the game world in real-time will be developed using the `API`.

Expected duration: 50 hours

Possible complications: Loading a huge game world might be complicated.

Task dependencies: `API`.

6.2.7 Testing and polishing

In this task, the platform will be tested under heavy load to ensure its stability and scalability.

Expected duration: 40 hours

Possible complications: Components might need to change if some unexpected bottleneck is detected.

Task dependencies: Game example.

6.2.8 Project memory

During this task, a document explaining how the project was developed will be written.

Expected duration: 40 hours

Possible complications: None.

Task dependencies: Engine. It will be written in different parts, after each task is finished.

6.2.9 Oral presentation

Once the project memory is finished, the author will prepare the final oral presentation.

Expected duration: 10 hours

Possible complications: Live examples must be well-prepared beforehand.

Task dependencies: Project memory.

6.3 Time table

Figure 6.1 summarizes the duration of every task described in the previous section. The total duration of the project is expected to be 460 hours. The author will need to work $\frac{460\ hour}{16\ week} \simeq 29\, hour/week$ to finish the project before the deadline, which seems reasonable.

Task	**Expected duration (h)**
Project management course	75
Analysis and design	10
Engine	
Learning	5
Implementation	50
Testing	14
Integration	1
Client	
Learning	10
Implementation	25
Testing	10
Integration	5
API	
Learning	5
Implementation	20
Testing	3
Integration	2
Control panel	
Learning	5
Implementation	20
Testing	5
Integration	5
Game example	
Logic	50
Viewer	50
Testing and polishing	40
Project memory	40
Oral presentation	10
Total	460

Figure 6.1: Planning time table

6.4 Timeline

Figure 6.2 shows the expected timeline of the project, taking into consideration task dependencies.

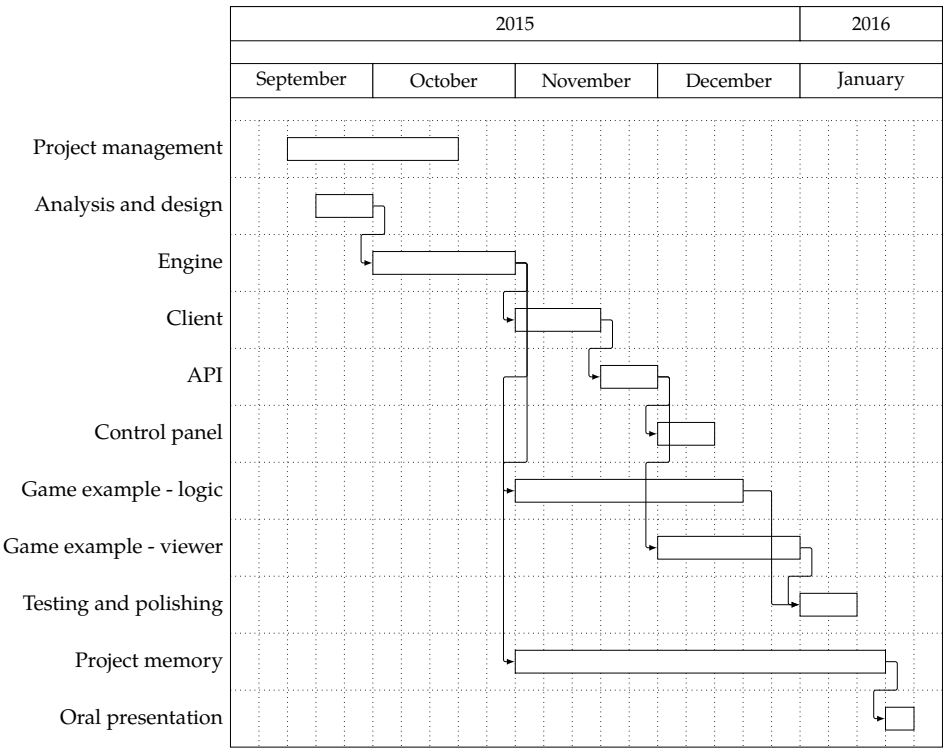

Figure 6.2: Planning timeline

7 Budget

7.1 Hardware resources

The project will be developed using a personal desktop computer and a laptop. Also, a monitor, a keyboard and a mouse are needed to use the desktop computer. There is no other hardware needed.

Hardware	Cost (€)	Useful life (years)	Amortized cost (€)
Desktop computer	2600.00	4	34.13
Personal laptop	1000.00	4	13.13
Monitor Acer XB270HU	750.00	4	9.85
Mouse Corsair M60	60.00	4	0.79
Keyboard Corsair K70 RGB	170.00	4	2.23
Total			60.13

Table 7.1: Hardware budget

7.2 Software resources

The software to develop the project is: Ubuntu, Sublime Text, a web-browser, `CLion`, LaTeX, `Makefile`, `git`, `evince`, `HAL`, C++, `Go`, Javascript, `Coffeescript`, and `WebGL`.

However, all that software can be used for free. Table 7.2 shows the licenses of most of the software needed.

Mozilla Firefox includes a Javascript engine and `evince` is included in Ubuntu. Also, the `HAL` programming language is owned by the author of the project. `Jetbrains` allows students to use `CLion` for free[1] and `Sublime Text` can be used without registration with no limitations[2]. Therefore, there are no

[1] https://www.jetbrains.com/student/
[2] http://www.sublimetext.com/2

software costs.

Software	License
Ubuntu	http://www.ubuntu.com/about/about-ubuntu/our-philosophy
LaTeX	https://latex-project.org/lppl/
git	https://git-scm.com/about/free-and-open-source
C++	https://gcc.gnu.org/onlinedocs/libstdc++/manual/license.html
Go	https://golang.org/project/
Mozilla Firefox	https://www.mozilla.org/en-US/foundation/licensing/
Coffeescript	https://github.com/jashkenas/coffeescript/blob/master/LICENSE
WebGL	https://www.khronos.org/legal/license/

Table 7.2: Software licenses

7.3 Human resources

Table 7.3 the expected salary per project role. Table 7.4 shows the expected cost of the human resources according to project roles and their respective tasks.

Role	Payment (€/ h)
Project manager	35.00
Software engineer	40.00
Software developer	30.00

Table 7.3: Salary per role

Role	Task	Time (h)	Cost (€)
Project manager	Project management course	75	2625.00
	Project memory	40	1400.00
	Oral presentation	10	350.00
Software engineer	Analysis and design	10	400.00
Software developer	Engine	70	2100.00
	API	50	1500.00
	Client	30	900.00
	Control panel	35	1050.00
	Game example	100	3000.00
	Testing and polishing	40	1200.00
Total		460.00	14 525.00

Table 7.4: Human resources budget

7.4 Other resources

7.4.1 Electricity

Electricity will be needed to power the hardware. Table 7.5 shows the power consumption, the estimated time of usage, and the cost for every piece of hardware that needs an external source of power, assuming 0.10 €/ kWh in Spain [20].

Hardware	Consumption (W)	Time of usage (h)	Cost (€)
Desktop computer	400	400	16.00
Laptop	100	50	0.50
Monitor Acer XB270HU	30	400	1.20
Total			17.70

Table 7.5: Electricity budget

7.4.2 Internet connection

An Internet connection will be necessary to perform all the tasks. The author will use its personal internet connection most of the time, which costs 38€/month ≃ 0.05€/h. It is expected to use the internet connection during the 30% of the total project's duration. Thus, the estimated budget for the internet connection is 460h · 0.05€/h · 0.3 = 7.28€.

7.5 Total

Table 7.6 shows the total budget needed to develop the project. The 10% of the total cost is added to face any unforeseen contingencies.

Resource	Total cost (€)
Hardware	60.13
Software	0.00
Human	14 525.00
Electricity	17.70
Internet	7.28
Subtotal	14 610.11
Contingency (10%)	1461.01
Total	16 071.12

Table 7.6: Total budget

Part III

Implementation

8 Methodology

As seen in chapter 4, the platform to be implemented is composed of different components that communicate with each other. Developing a single piece of software is a difficult task, but it becomes even harder when this piece of software communicates with other software, especially when both are being developed at the same time; changes made in the public interface of some component can break other components.

Therefore, components will break. Developers are humans, and humans make mistakes. Thus, in order to be productive, it is necessary to detect when a component breaks as soon as possible. That way, components can be fixed before more changes are made and fixing it becomes a nightmare.

In order to ensure that all the components of the platform keep working during the entire implementation process, and also detect and fix issues as soon as possible, the project was developed using an iterative methodology based on continuous integration [9]. This methodology consisted in:

1. Develop a simple prototype of the platform.
2. Run the prototype in some external server.
3. When a change is made to any component:
   (a) Apply and integrate the change in the server.
   (b) Test that the prototype keeps working as expected.

This methodology has a lot of benefits. Not only it allows to detect mistakes as soon as they are made, but it also provides a fully working prototype of the platform that can be shown to anyone. Thus, the development progress can be shown transparently. Stakeholders can try the software while it is being developed and give feedback. In this case, the prototype was available to the advisor of the project to assess and follow any progress closely.

9 The first prototype

The first step of the implementation process was to build a simple prototype of the platform.

This first prototype must implement the most basic feature of the platform and it also has to lay the foundations for extending it easily. Therefore, it was decided that the first prototype should feature the first working versions of:

1. The engine, running one simple player and notifying its actions to any subscribers.
2. The `API`, subscribed to the engine and notifying its actions to clients.
3. The client library, subscribed to the `API` and drawing the player actions in a web-browser.

It is important to note that the engine and the client are libraries that can be used to build game logic and game viewers, respectively. They cannot be executed as a stand-alone component. However, game-specific code was included temporarily in this first prototype, which avoided unnecessary complexity[1].

The entire code of the `MMPG` platform can be found under the `mmpg` GitHub organization: https://github.com/mmpg.

9.1 The first API

9.1.1 Programming language

There are many programming languages that allow to create HTTP APIs, like Pyhton, `Ruby`, `Elixir`... However, the language chosen to implement the `API` was the `Go` programming language because it includes native libraries to build concurrent `REST` APIs and it is simpler, faster and easier-to-deploy than the alternatives.

[1] i.e. creating game logic and game viewer components

9.1.2 The subscriber hub

The `API` needs to keep track of the different clients subscribed to it, and it also needs to be able to send data to them.

A subscriber hub was developed using the native concurrency of Go. Basically, the hub runs an infinite loop in the background that smartly monitors the connected clients; sending data or closing connections accordingly.

9.1.3 The events endpoint

A simple HTTP endpoint was created: `/events`. This endpoint accepted WebSocket connections and delegated them to the hub.

After that, the `API` was ready to send data to game viewers in real-time.

9.2 The first client

9.2.1 Programming language

The client code needs to be executed by web-browsers, as game viewers are web-based. Thus, it has to be written in `Javascript`, as it is supported by web-browsers natively.

9.2.2 API subscription

A WebSocket was connected to the `/events` endpoint, printing any received data in the `Javascript` console.

9.3 The first engine

9.3.1 Programming language

The engine needs to be fast, as the game logic will be built on top of it, and it also needs to access low-level operative system operations, so it can limit how player programs are executed.

The most well-known programming languages that satisfiy these requirements are `C` and `C++`. However, `C` is lacking the capacity to represent abstractions and interfaces easily. Hence, `C++` is the perfect alternative to implement

the engine, as it is efficient, object-oriented and it has access to the C `POSIX` API, which allows to talk directly to `UNIX`-based operative systems.

9.3.2 The architecture

When designing the first architecture of the engine, it was important to take its requirements into account:

1. The engine must not allow `AIs` to cheat.
2. The engine must be scalable.
3. `AIs` may be hot-swapped during a match.

Thus, if the game logic and all the `AIs` were executed in the same process:

1. `AIs` could access the memory related with the game logic or other `AIs`.
2. `AIs` must be executed in the same machine.
3. Updating an `AI` would require a rebuild of the entire game logic and other `AIs`, and a restart of the process.

In other words, it would be really unsafe, inefficient and it would couple all the `AIs` and the game logic together.

Therefore, the architecture of the engine consisted of 3 types of processes:

Master process: It represented the game-world server. The master process listened to requests coming from players and updated the game world accordingly. There was only one master process per runtime.

Worker process: It represented a pool of players. A worker process executed a set of players and managed them. There could be multiple workers per runtime.

Player process: It represented a player program. A player process read the game world from the master process and performed requests to change the game world.

Processes comunicated with each other using low-latency sockets. Thus, different workers could be executed in different machines to achieve better performance.

Figure 9.1 shows the hierarchy of an engine runtime with N workers and $M = \sum_{i=0}^{N} M_i$ players.

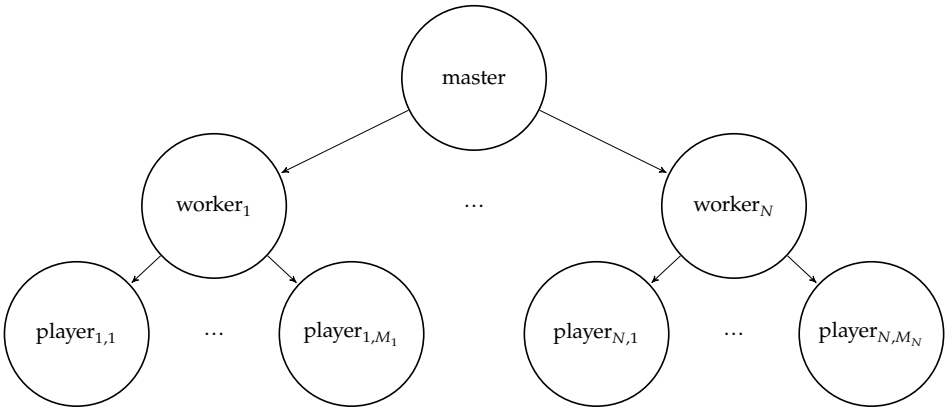

Figure 9.1: Hierarchy of engine processes

9.3.3 Inter-process communication

As it was explained previously, the engine featured a decoupled architecture. Different processes were executed and communicated with each other using sockets. However, implementing inter-process communication from scratch would have been a real challenge by itself, and it was not the subject of this project. This is where a messaging technology came in.

The most well-known messaging technologies out there are: `RabbitMQ`, `ZeroMQ` and `ActiveMQ`.

`RabbitMQ` implements a broker architecture, which means that messages are queued on a central node before being sent to its destination. This architecture is totally unnecessary for the engine, as we want to decouple components, and it would also add some latency.

`ActiveMQ` can be used with a peer-to-peer architecture but, when compared with `ZeroMQ`, it is a high-level library. Thus, controlling the type of communication or socket behaviour is not easy with `ActiveMQ`.

On the other hand, `ZeroMQ` [17] is an embeddable networking library that implements low-latency socket communication. It manages low-level communication, while providing a flexible and easy-to-use interface. Also, `ZeroMQ` has bindings available for the most well-known programming languages.

Thus, `ZeroMQ` was the library chosen to implement all the inter-process communication.

9.3.4 The first master process

The first master process consisted of two basic components:

The notifier: It sent any given message through a `ZeroMQ` socket.

The server: It listened requests made by player processes and notified them through the notifier.

This components allowed the master process to act as a game server, while notifying the actions of the players to any subscribers.

9.4 Putting it all together

Once the initial engine was finished, a simple `AI` was developed to simulate a player moving while following a square shape. This way, it would be clear if the prototype worked correctly or not, as the web-viewer would need to show this movement.

Then, the `API` was connected to the engine notifier and the received messages were directly delegated to the `API` hub, which in turn were sent to any open clients.

Finally, once the `Javascript` console started showing the received events, a simple scene using `Three.js` was created in the client. This scene featured a cube representing the player, whose position was updated with every received event. Figure 9.2 shows the observed result. The prototype was finished.

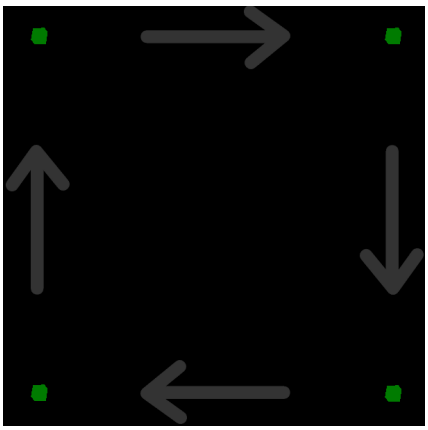

Figure 9.2: A cube that was moved by the player program

9.5 Summary

In this chapter, it is been described how the first prototype of the platform was built. A prototype that implemented the most basic features of every component:

1. The engine compiled and run players, and notified their actions to subscribers.
2. The `API` subscribed to the engine and delegated the received events to all the connected clients.
3. The client was able to connect to the `API` and receive events from the engine.

The foundations of the platform were ready.

10 Continuous integration

Once the first prototype was finished, it was time to set up the integration server, as described in chapter 8.

10.1 The dedicated server

First things first, an integration server to run the prototype was needed.

Initially, a server was requested to the project advisor. However, the university policy did not allow professors to give root access to students. Root access was necessary because many different services might need to be installed during the development of the platform[1]. Therefore, it was decided to use an external cloud-host provider.

There are many providers that present cloud-hosting solutions[2]. At the end, Linode [22] was the provider chosen, as the project developer already had experience with it and the cheapest solution[3] was enough to run the integration server.

A Linode 1GB [22] running `Ubuntu` was booted up to execute an instance of the platform.

10.2 The integration service

The next step consisted in setting up an integration service.

Integration services keep track of some code repositories and can be configured to build and test the code when a change happens. Additionally, most of them can be configured to deploy changes when a build succeeds.

[1] `ZeroMQ`, for example
[2] AWS [18], DigitalOcean [19], etc.
[3] This cost was covered with the contingency budget.

As it happened with cloud-hosting providers, there are many different integration services[4]. Fortunately, most of them are free to use for open-source projects, like this one. At the end, Semaphore [30] was the chosen integration service. This decision was based in the experience the project developer had with the service, as it also happened with the cloud-hosting provider.

10.3 The integration process

Semaphore was configured to trigger a build when any component changed. If a build succeeded, then Semaphore deployed the changes to the Linode, where the changes were applied.

Changes were applied in the Linode using bare `git` repositories and script hooks. Basically, when some changes were pushed to the repository, a script was executed that built and restarted the component. These scripts were located in the repository itself, so they could be updated easily with the same process.

Figure 10.1 shows the build process of the engine, while Figure 10.2 shows its deployment process.

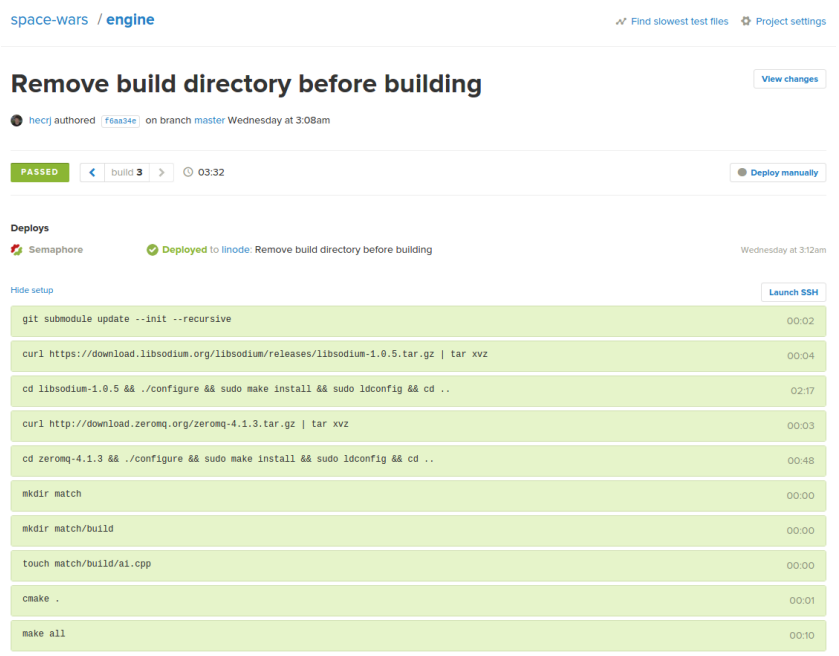

Figure 10.1: Semaphore build process

[4]TravisCI [14], CircleCI [7], etc.

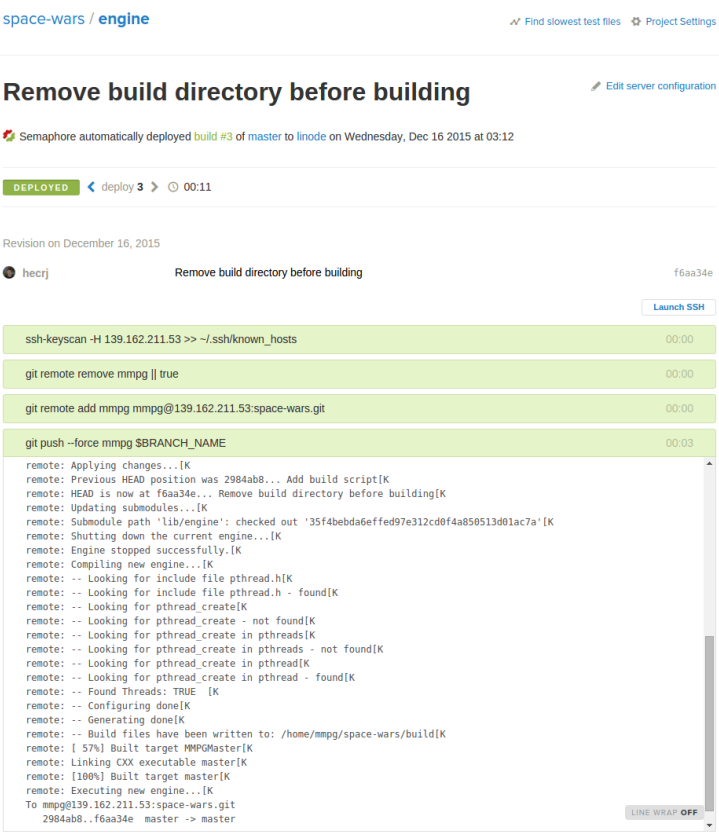

Figure 10.2: Semaphore deployment process

11 The log system

Matches of MMPGs are expected to last days or weeks, and while allowing viewers to watch the current state of the match is mandatory, AIs will keep playing while players sleep. Therefore, players will miss parts of the match. For this reason, it is important to allow viewers to replay the past of the match. This feature is especially interesting, as it allows players to detect long-term strategies and learn from mistakes without the need to watch the match constantly.

This chapter details how the replay feature was added to the platform by creating the log system.

11.1 Event sourcing

Event sourcing [12] is a technique that consists in saving the state changes of an application as a sequence of events. Hence, instead of saving the current state of the application, its entire history is saved. This technique has one main benefit: it allows to reconstruct the state of the application at any given time.

Event sourcing can be applied to the MMPG platform. A match can be seen as a sequence of immutable events:

Player actions: Triggered by the different AIs.

World updates: Triggered by the master process to update the game world.

Thus, it would be possible to recreate the match state at any given time, if such sequence were available, by applying all the events from the start until that point in time. But, as it was said before, matches can be quite long. Therefore, processing all the events might become really time consuming.

An efficient way to solve this problem consists in saving a snapshot of the match periodically. That is, a big event containing the current state of the match. As a consequence, it is possible to start from the closest snapshot in order to recreate the state of the match at a given time.

The event sourcing technique with the snapshot strategy was used to implement replays of a match efficiently in the `MMPG` platform.

11.2 Logging events

The first step to implement the replay feature was to log events in the engine.

The log system saved every player action. Additionally, a snapshot was also saved every second. These snapshots allowed to recreate the state of the match easily and also allowed viewers to synchronize its state.

Internally, the log system consisted of multiple log files, where every file represented a 5-minute interval, allowing the retrieval of events for a particular interval of time.

11.3 Delivering log files

While the notifier of the engine master process implemented in the first prototype is well suited for publishing events to subscribers, it is not able to receive data from them. For this reason, a new component was implemented in the master process: the engine app-server.

The app-server was able to receive requests from another application and reply accordingly. This component creates a channel that allows external applications to interact directly with the engine. The first version of this component only was able to handle log requests.

A log request was represented by a string of characters: the `LOG` keyword followed by a space and then the UNIX time of the log to be retrieved. For example:

```
LOG 1451606400
```

This request would ask for the log file that contains events created at `01/01/2016 @ 00:00 (UTC)`.

Once the engine was able to reply to log requests, a new endpoint `/log` was created in the `API` component. This endpoint accepted a `time` parameter, which was used to request the correct log file to the engine. Then, the log file was sent back to the client that requested it.

Finally, the client library was updated to enable log requests in viewers.

11.4 Compressing log files

Log files had a repetitive structure, given that multiple events of the same type were stored in it. Hence, they were really compressible. Moreover, log files were directly delivered to web-browsers through the `API`, and most modern web-browsers are able to decompress `gzip` natively. For this reason, log files were compressed using `gzip`, reducing both disk and bandwidth usage.

11.5 Controlling time

Finally, a widget was created in the viewer to control the game time. This widget allowed to start, pause, rewind and forward the game timer. Figure 11.1 shows the design of this widget.

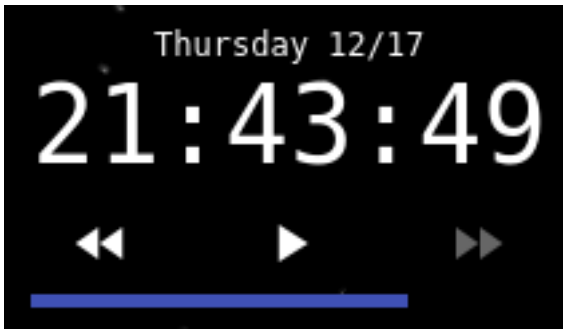

Figure 11.1: Game time widget

The game time widget performed log requests under the scenes when necessary. Then, it created a buffer of events from the log and started consuming them at the same rate they were created. When the buffer begun to empty, the next log file was requested to refill the buffer.

11.6 Summary

In this chapter, the replay feature was implemented. The implementation process featured:

1. Saving player actions and snapshots in log files.
2. Creating an app-server in the engine to handle log requests.
3. Creating a `/log` endpoint in the `API` to allow clients to request log files.

4. Compressing log files using `gzip`.
5. Developing a game time widget in the viewer to control game time.

12 Deployment of AI

The next most basic feature that the platform needed to support was the hot-swapping of `AIs`. Players needed to be able to change their `AIs` during a match.

However, it was also necessary to provide some security, so players could not change the `AIs` of other players. Hence, a method to authenticate players was needed.

This chapter details the implementation of the authentication system and the deployment system.

12.1 Recognizing players

The first step consisted in making the engine acknowledge the different players by some identifier. When the engine was booted up, it read a file containing a list of user identifiers. Then, each one of these identifiers was internally linked with a match player.

Afterwards, a new type of request handler was added to the app-server developed in the previous chapter: `PLAYER_EXISTS`. This request consisted of the `PLAYER_EXISTS` keyword followed by a space, and then a user identifier. For example:

```
PLAYER_EXISTS hector.ramon@est.fib.upc.edu
```

In this example, the user identifier is an e-mail. But it is important to note that the engine was not limited to e-mails, it was able to work with any type of string.

When the engine received a `PLAYER_EXISTS` request, it checked if the given identifier was linked to some player in the current match and replied `YES` or `NO` accordingly.

12.2 Authenticating players

The authentication system was implemented in the `API` component. Mainly because authentication solved an environment limitation of the component that exposed the engine: the `API`. Hence, the engine did not need to know how to authenticate users, the `API` did.

A new endpoint `/auth` was created in the `API`. This endpoint accepted two types of requests:

`POST`: It validated player credentials:

1. Accepted an identifier and a password.
2. Checked whether the identifier and the password were valid using a credentials validator.
3. Returned a `403 Forbidden` error if the credentials were not valid.
4. Checked if the player existed in the engine using the `PLAYER_EXISTS` request.
5. Returned a `403 Forbidden` error if the player did not exist.
6. Returned a digitally-signed authentication token for the given identifier.

`GET`: It renewed authentication tokens

1. Accepted an authentication token.
2. Checked if the authentication token was valid.
3. Returned a `400 Bad Request` error if the authentication token was not valid.
4. Returned a new authentication token.

The authentication tokens were implemented using `JSON WebTokens` [4]. This tokens have an expìration date, hence the necessity to renew them.

A credentials validator is a simple `Go` function that tells whether the given identifier and password are valid or not. A credential validator can be injected in the `API`. Hence, anyone can implement its own credentials validator and customize the authentication system.

Finally, a credentials validator for the Jutge.org platform was developed using an endpoint provided by the project advisor. Thus, to authenticate properly it was necessary to provide a valid e-mail and password of a Jutge.org account.

12.3 Logging in players

Once the authentication system was implemented, the client library was updated to add support to the new `/auth` endpoint. Also, a login form was designed in the viewer to allow players to login. Figure 12.1 shows how this login form looks.

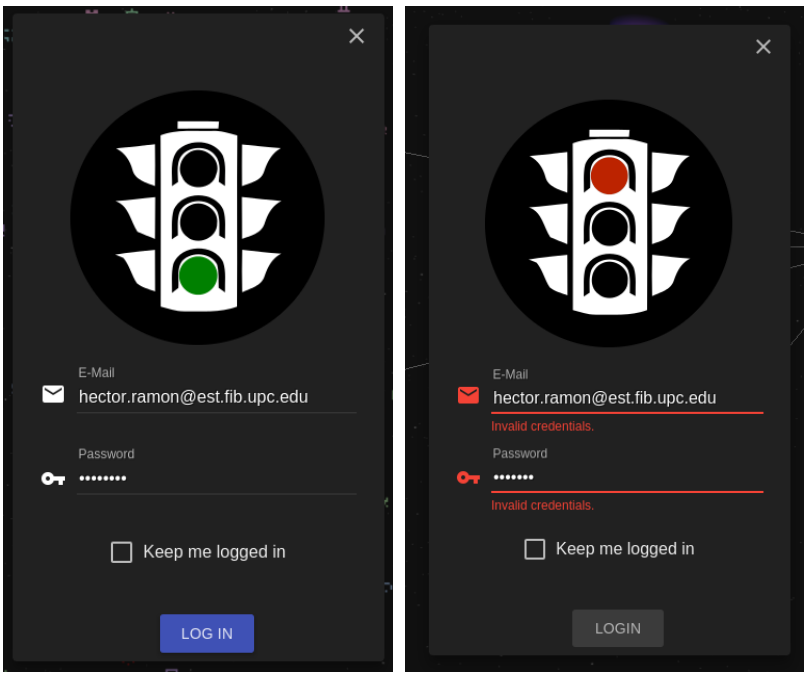

Figure 12.1: Login form. Valid (left). Invalid (right)

When a login succeeded the returned authentication token was saved in the local storage of the browser and renewed when the viewer was reaccessed, or every 30 minutes if it was kept open.

12.4 Hot-swapping AI

Once, players were able to login, the deployment of new `AIs` could be implemented.

Firstly, a new request handler in the app-server was needed: `DEPLOY_PLAYER`. This type of request had two parameters: the player identifier and the code of the new `AI` encoded in `base64`. After receiving a `DEPLOY_PLAYER` request, the engine tried to compile the code, assigned the new executable to the player

with the given identifier, and restarted its player process. In case the compilation failed, then the compilation error was returned.

Secondly, a new `API` endpoint `/player` was implemented to expose the new engine functionality and allow authenticated players to change their `AIs`. This endpoint was pretty simple:

1. Accepted an authentication token and an uploaded file.
2. Checked whether the authentication token was valid or not.
3. Returned a `401 Unauthorized` error if the authentication token was invalid.
4. Encoded the uploaded file in `base64`.
5. Issued a `DEPLOY_PLAYER` request to the engine using the identifier in the authentication token and the encoded file as parameters.
6. Returned a descriptive error if the deploymet failed.
7. Returned a `200 OK` response.

As it is been shown multiple times now. There is an `API` endpoint for almost every request that the engine app-server can handle. Hence, the `API` exposes the engine with an access layer on top of it.

12.5 Deploying new AI

Finally, the client was updated to support the new `API` endpoint and a deployment form was created in the viewer. This form was designed having in mind that the deployment process should be fun, so players feel encouraged to improve its code. Figure 12.2 shows how the deployment form looks.

Also, a player menu was designed to welcome authenticated players. From this menu, players can access the deployment form and they can also logout. Figure 12.3 shows how the player menu looks.

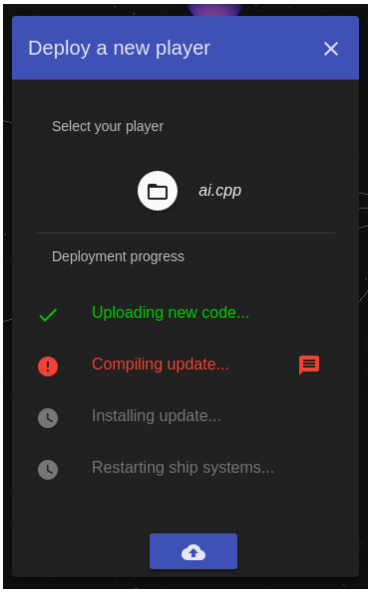

Figure 12.2: Player deployment with a compilation error

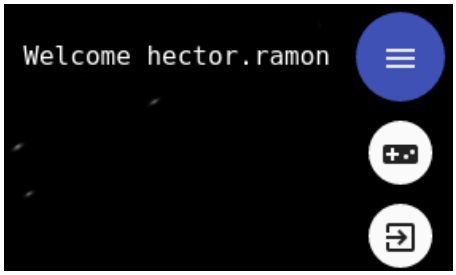

Figure 12.3: Player menu

12.6 Summary

This chapter detailed the implementation of the `AI` hot-swapping feature, which needed an authentication system first. Summarizing, the most important implemented components are:

1. The `PLAYER_EXISTS` and `DEPLOY_PLAYER` request handlers in the engine app-server
2. The `/auth` and `/player` endpoint in the `API`
3. The login form, player menu, and deployment form in the viewer.

13 The control panel

Given that an authentication system was available, it was then possible to create a control panel to allow administrators control the current match.

13.1 Recognizing administrators

At this point, the platform did not know anything about administrators, only players.

An administrator is a user that has privilege to control the current match. Given that administrators were not used in the game logic, it did not make sense to add them in the engine. Hence, the concept was added in the `API`, as a way to control access to some engine operations.

The `API` was changed to read a file when booting up that contained a list of user identifiers that should be considered administrators. Then, the `/auth` endpoint was changed to allow administrators login even if they did not have an associated player in the engine.

13.2 Assuming control

New types of request handlers were added to the engine app-server that allowed to pause and play the match: `PLAY_MATCH` and `PAUSE_MATCH`, respectively.

These new handlers were exposed through the `API` in the new endpoints: `/control/play` and `/control/stop`. Obviously, these endpoints checked that the authenticated user requesting the action was an administrator. If not, a `403 Forbidden` error was returned.

Finally, a simple control panel was added in the viewer to allow administrators control the match.

14 The first game world

Before going any further, it was convenient to make a quick review of the current state of the platform. At this point, the platform was able to:

1. Notify game events in real-time
2. Replay the past of the current match
3. Hotswap `AIs` during a match

These features alone, fullfilled 2 of the 6 secondary objectives detailed in section 3.2, specifically:

1. Allow hot-swapping of `AIs`
2. Implement a real-time webviewer

Therefore, there were still four secondary objectives to fullfill. However, one of these objectives talked about stability and scalability, features that could only be validated at the end of the project (see chapter 17).

For this reason, it was decided that the next implemented features aimed to fullfill the objectives:

1. Develop an abstract game engine
2. Create a game example

These two objectives could be fullfilled both at the same time by creating a game example without adding any game-specific code in the engine. To achieve that, a set of abstract classes were created in the engine to build game logic on top of them:

`Game`: Used to configure general aspects of the game and as a factory of game `Worlds` and `Actions`.

`World`: Used to implement game update logic and world serialization.

`AI`: Used to build the interface that players will have available when imple-

menting AIs.

Action: Used to implement action logic and serialization.

These classes allowed game developers to build MMPGs while extending the engine code, not changing it.

14.1 Choosing a game: Space Wars

First, it was necessary to choose the game to implement. The best candidates were games with simple rules, but that also seemed easy to extend. Hence, development could start with the implementation of the original game rules while new ideas to increase the game size and make it massive could be added later.

At the end, it was chosen to implement a game inspired by Galcon [15], which also inspired Planet Wars [28]. Planet Wars was a game used in the AI programming challenge (see section 2.2) sponsored by Google during 2010. The game rules are simple:

1. The game world consists in a system with a determined number of planets.
2. Every player owns one planet before the match starts.
3. During the match, every owned planet generates space ships in a rate that is proportional to the radius of the planet.
4. Ships can be sent from planet to planet.
5. A player can conquer other planets by sending a greater number of ships than the planet holds.
6. The objective of the match is to destroy other players and/or conquer the major number of planets.

These rules make an interesting strategic game that is quite convenient for a programming game.

The code of the game example can be found in the space-wars-game GitHub organization: https://github.com/space-wars-game.

14.2 Generating a planetary system

The first step to implement the game rules was to generate a game world: a planetary system.

In order to make every match different, it was decided to generate the game world procedurally using a pseudo-random generator[1].

A simplified model of a realistic planetary system was taken. A planetary system in the game had one sun and a determined number of planets. A planet was assumed to move describing an ellipse with the sun as its center. Thus, assuming the position of the sun was $(0, 0)$, then the coordinates x and y of the planet were

$$x = a\,cos\theta$$

$$y = b\,sin\theta$$

where a and b were the major and minor axes of the ellipse, respectively, and θ was the eccentric anomaly.

Algorithm 14.1 shows the algorithm used to generate planetary systems procedurally in the game. Basically, the algorithm generated planets outwards from the sun, while ensuring their orbits did not collide, and positioned them on a random point of its orbit.

Algorithm 14.1 Procedural generation of planetary systems

```
sun ← Sun(in_range(Sun::MIN_RADIUS, Sun::MAX_RADIUS))
num_planets ← in_range(System::MIN_SIZE, System::MAX_SIZE)
planets ← list of num_planets planets
previous ← sun
for p in planets do
  p_radius ← in_range(Planet::MIN_RADIUS, Planet::MAX_RADIUS)
  dist_x ← in_range(Planet::MIN_DIST_X, Planet::MAX_DIST_X)
  dist_y ← in_range(Planet::MIN_DIST_Y, Planet::MAX_DIST_Y)
  p_a ← previous_a + dist_x + p_radius + previous_radius
  p_b ← previous_b + dist_y + p_radius + previous_radius
  p_θ ← in_range(0, 2π)
  previous ← p
end for
```

[1] Mersenne Twister [23]

14.3 Connecting planets

At this point, it was possible to generate a planetary system as the game world. However, it was decided that a planet could not send ships to any other planet, but only to planets that were connected to it, unlike the original Galcon [15]. As a consequence, the game became more interesting, encouraging players to learn path-finding techniques.

Hence, it was necessary to also generate a connected graph of planets. Algorithm 14.2 shows the algorithm used to generate the planetary connections procedurally.

Algorithm 14.2 Procedural generation of planetary connections

Require: p is a list of planets
  $relay \leftarrow p[in\_range(0, p_{length})]$
  $connected \leftarrow$ queue of planets
  $disconnected \leftarrow p - relay$
  push $relay$ to $connected$
  while $disconnected$ is not empty **do**
    $edge \leftarrow$ front of $connected$
    pop front of $connected$
    $max\_connections \leftarrow min(\text{Planet::MAX\_CONNECTIONS}, disconnected_{length})$
    $num\_connections \leftarrow in\_range(\text{Planet::MIN\_CONNECTIONS}, max\_connections)$

    for i in $(0, num\_connections)$ **do**
      $c \leftarrow disconnected[in\_range(0, disconnected_{length})]$
      add c to $edge_{connections}$
      add $edge$ to $c_{connections}$
      $disconnected \leftarrow disconnected - c$
      push c to $connected$
    end for
  end while

14.4 Rendering the planetary system

Finally, the viewer was upgraded to render the generated planetary system and its connections.

Basically, OpenGL spheres were used to draw the sun and the planets, while connections were represented using lines. Additionally, a light was added in the center of the sun to give some depth to the scene.

Figure 14.1 shows a planetary system rendered by the viewer.

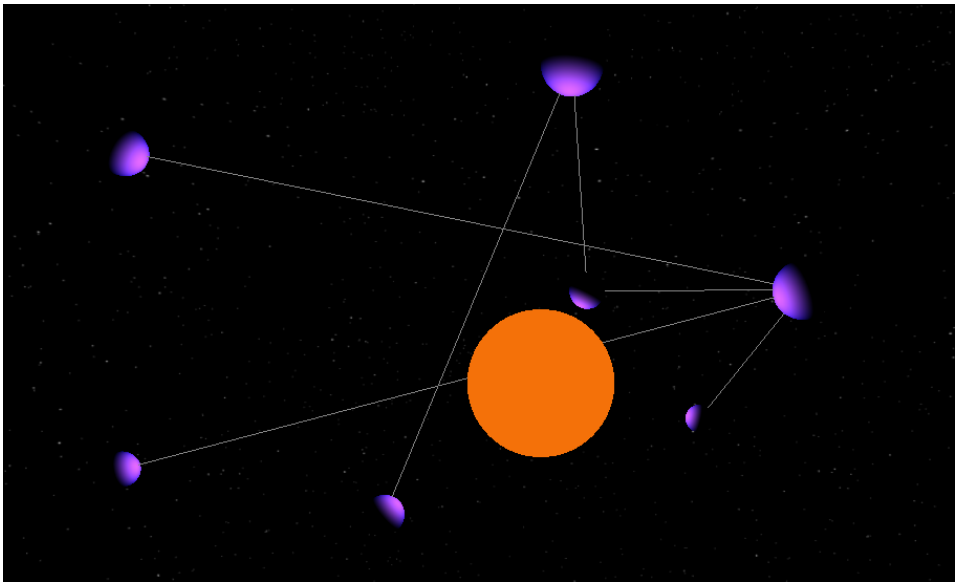

Figure 14.1: Planetary system rendered by the viewer

14.5 Summary

This chapter detailed how the implementation of the game example started:

1. It was chosen to create a game inspired by Galcon [15].
2. A simple algorithm was implemented to procedurally generate the game world.
3. The viewer was upgraded to render the generated game world.

15 Fleets and planets

At this point, a planetary system was being generated as the game world. Thus, it was possible to start developing the main game mechanics: ship generation and fleet movement.

15.1 Generating ships in conquered planets

The ship generation mechanic states that conquered planets generate ships at a rate proportional to the radius of the planet. Hence, the bigger the planet, the more ships it will generate during a specific timeframe.

The nature of this mechanic is periodical: conquered planets need to be updated quite frequently. Games can easily implement this type of mechanics because their entire logic is executed in a game loop.

A game loop is the central code of a game. It is a loop that is executed constantly to update the state of the game accordingly. A good game loop computes the time between iterations and has a fixed timestep, which ensures that the update logic always advances the state of the game in the same amount.

Algorithm 15.1 shows an example of a game loop with a fixed timestep. This type of game loop avoids differences in game speed between different hardware, while it also avoids passing a big timestep to the update logic, which could break game calculations[1].

At the end, the engine was upgraded to support a game loop with a fixed timestep. The timestep could be configured by extending the `Game` class described in chapter 14. Then, the corresponding update logic was implemented in the game logic to generate ships in conquered planets as described by the game mechanic.

[1] like collision detection, movement, etc.

Algorithm 15.1 Game loop with a fixed timestep

```
accum ← 0
time_current ← current_time()
while not quit do
  time_new ← current_time()
  time_frame ← time_new − time_current
  time_current ← time_new
  accum ← accum + time_frame
  while accum ≥ timestep do
    update_state(timestep)
    accum ← accum − timestep
  end while
end while
```

15.2 Moving fleets between planets

The fleet movement mechanic describes how players can send ships from a planet to another.

Basically, it was implemented assuming that a fleet trip is defined by:

- Planet origin
- Planet destination
- Fleet owner
- Number of ships
- Travel time

The travel time was proportional to the euclidean distance between the planet origin o and the planet destination d, assuming that every fleet moved at the same velocity v:

$$t_{time} = \frac{\sqrt{(d_x - o_x)^2 + (d_y - o_y)^2}}{v}$$

The velocity value was constant in the game, but it could be easily changed to experiment with different values.

Then, in the update logic, every on-going fleet trip was updated properly, applying changes to the destination planets when a trip finished. Algorithm 15.2 and 15.3 give a basic idea of how the trip and planet update logics were implemented, respectively.

Algorithm 15.2 Fleet trip update logic

```
for t in current_trips do
  t_time ← t_time − timestep
  if t_time ≤ 0 then
    update_planet(t_destination, t)
    current_trips ← current_trips − t
  end if
end for
```

Algorithm 15.3 Planet update logic

```
Require: p is a planet
Require: t is a fleet trip
if not p_owner then
  p_owner ← t_owner
end if
if p_owner = t_owner then
  p_ships ← p_ships + t_ships
else
  if p_ships ≥ t_ships then
    p_ships ← p_ships − t_ships
  else
    p_owner ← t_owner
    p_ships ← t_ships − p_ships
  end if
end if
```

15.3 Rendering fleets and planet information

Finally, the viewer was upgraded to render fleets moving between planets. Octahedrons were used to represent fleets, scaling them proportionally to the number of ships. Fleets movement was animated using the planet origin, planet destination and travel time.

Additionally, the viewer was modified to show planet information: number of ships and owner. These elements were projected on top of the `3D` viewer using `HTML` elements.

A simple `AI` was implemented and two players were confronted with each other to test these mechanics and the viewer together. Figure 15.1 shows the code of the developed `AI`, while Figure 15.2 shows an example of a match played by two players using this `AI`.

```
#include "ai.hpp"

using namespace space_wars;

class SimpleAI : public space_wars::AI {
 public:
  SimpleAI() {

  }

  void Play() {
    ScanUniverse();

    int me_ = me();

    for(int planet_id : owned_planets(me_)) {
      const Planet& owned = planet(planet_id);

      for(int connection_id : owned.connections) {
        const Planet& connection = planet(connection_id);

        if(connection.owner != me_ and owned.ships / 2 > connection.ships) {
          SendFleet(planet_id, connection_id, owned.ships / 2);
          return;
        }
      }
    }
  }
};

RegisterAI(SimpleAI)
```

Figure 15.1: A simple `AI` that tries to conquer all the planets

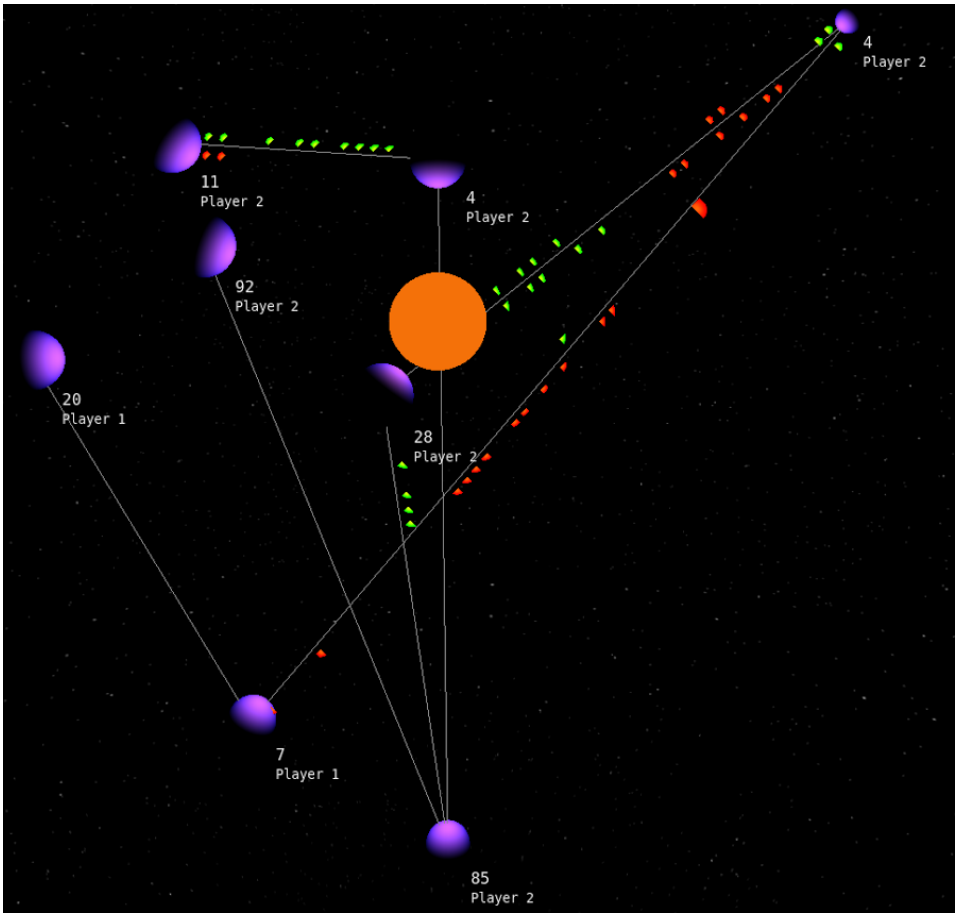

Figure 15.2: The viewer showing fleets and planet information

15.4 Summary

In this chapter, the basic mechanics of the game example were developed. The implementation process consisted in:

1. Adding game loop support to the engine
2. Implementing the ship generation logic
3. Implementing the fleet movement logic
4. Upgrading the viewer to show fleets and planet information

16 A whole galaxy

At this point, the integration server was running a game similar to Galcon: a small number of players could compete to win control over the generated planetary system. However, it was necessary to increase the game size in order to add support for a high amount of players.

16.1 Generating multiple planetary systems

At this moment, the game world was a planetary system. Therefore, the logical step to increase the game size was to think about a galaxy.

The game size was increased by generating a number of planetary systems proportional to the amount of players of the match. Different systems were connected using an algorithm similar to the one used to connect planets (described in chapter 15). A new type of celestial body was added to each system that allowed ships to travel to its connected systems: the relay.

In order to simplify the galaxy generation, it was decided that the system position would not affect travel time between systems. Specifically, travelling from one relay to another was instantaneous. This way, a galaxy could be simplified to an ordered set of systems, with no need to positionate the systems correctly inside the game logic.

16.2 Separating world structure from dynamic data

Once the new world generation was ready, a new prototype was deployed to the integration server. This prototype generated 300 systems as the game world.

After deploying the prototype, the CPU usage in the server increased from 1-2% to 20%. While an increase in the CPU usage was expected, it was not expected to be that high. Hence, an analysis was performed to detect any

possible bottlenecks.

The main detected symptom was that the `API` was notifying 200 kbits of events per second, when the bandwidth needed was 10-12 kbps before deploying. The events that changed after the deployment were the snapshots (see chapter 11). The game world increased its size by 300 times, thus the snapshots were approximately 300 times bigger. These snapshots were also sent to players quite often to keep their copies of game worlds updated.

Given that the snapshot generation turned expensive, an optimization was needed. Most of the generated game world was static, it did not change: a system stayed with the same structure the entire match. The only data that could potentially change was the number of ships and the owner of every planet. Thus, there was no reason to constantly generate snapshots where most of their data was static. Instead, static data and dynamic data could be separated.

Therefore, the optimization consisted in generating a world structure snapshot only once, and notify it to players and viewers only once. Then, snapshots containing dynamic data were generated and notified periodically, as before.

The optimization reduced the CPU usage from 20% to 3% consistently and the bandwidth from 200kbps to 20kbps. Figure 16.1 shows the CPU usage drop after the optimization was deployed.

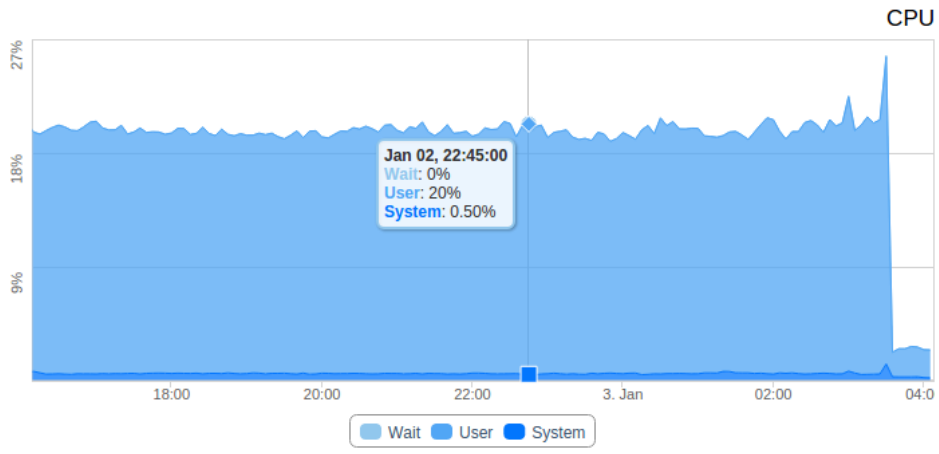

Figure 16.1: CPU usage drop after separating world structure from dynamic data

16.3 Rendering a galaxy

After the optimization, the game logic was mostly finished. Space Wars was able to scale its size with the number of players efficiently. Hence, the last task left to finish the game was to upgrade the game viewer so it could show the generated galaxy.

The idea was to show a galaxy map. Basically, the galaxy map would show a general view of all the game world. In other words, it would show every system in a clear way. However, as it was said before, the galaxy is just an ordered set of systems. A system did not have a particular position in the game logic. As a consequence, the viewer had the responsibility to turn this ordered set into a galaxy.

Now, galaxies come in different shapes [31], but the most familiar of them is the spiral shape. Our galaxy, the Milky Way has an spiral shape, as shown by Figure 16.2.

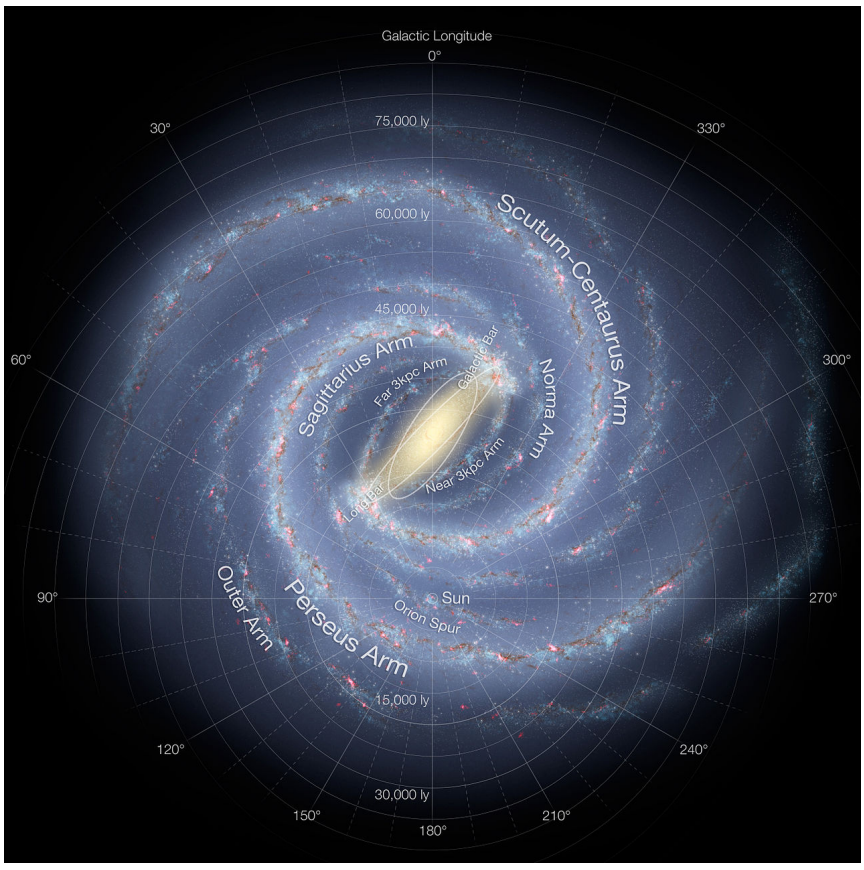

Figure 16.2: Artist's conception of the spiral structure of the Milky Way [16]

Hence, in order to simulate an spiral galaxy, it was necessary to study the spiral equations [21] first.

Basically, the coordinates of the points in a spiral are defined by

$$x(t) = at\,cos(t)$$
$$y(t) = at\,sin(t)$$

where t is an angle and a is a constant that controls the spiral density.

However, spiral galaxies can have multiple arms. This means that they can be formed by multiple spirals with different rotations. A spiral can be easily rotated by an angle θ:

$$x(t) = at\,cos(t + \theta)$$
$$y(t) = at\,sin(t + \theta)$$

Using these last equations, it was relatively easy to build a simple algorithm that assigned a position (x, y) to every planetary system of the game world, while generating a galaxy with a number of arms proportional to the number of systems. Algorithm 16.1 shows the algorithm used to generate the galaxy map.

Algorithm 16.1 Galaxy map generation

Require: *systems* is a list of systems
Require: α is the initial angle
Require: $arm_{systems}$ is the amount of systems per arm
  $\theta \leftarrow 0$
  $t \leftarrow \alpha$
  $\theta_{inc} \leftarrow 2\pi \frac{arm_{systems}}{systems_{length}}$
  $t_{inc} \leftarrow \frac{2\pi}{arm_{systems}}$
  for s in *systems* **do**
    $s_x \leftarrow a \cdot t \cdot cos(t + \theta)$
    $s_y \leftarrow a \cdot t \cdot sin(t + \theta)$
    $t \leftarrow t + t_{inc}$
    if $t \geq 2\pi + \alpha$ **then**
      $t \leftarrow \alpha$
      $\theta \leftarrow \theta + \theta_{inc}$
    end if
  end for

Additionally, an ideal way to represent players uniquely was needed. Given that the number of players could be really high, using colors would have been a bad idea, as some players would share shades of the same color. Hence, `GitHub`-based `Identicons` [13] were used to identify different players. The `Identicons` are generated using player identifiers, thus players can keep the same `Identicon` in different matches.

Figure 16.3 shows an example of a galaxy rendered by the game viewer.

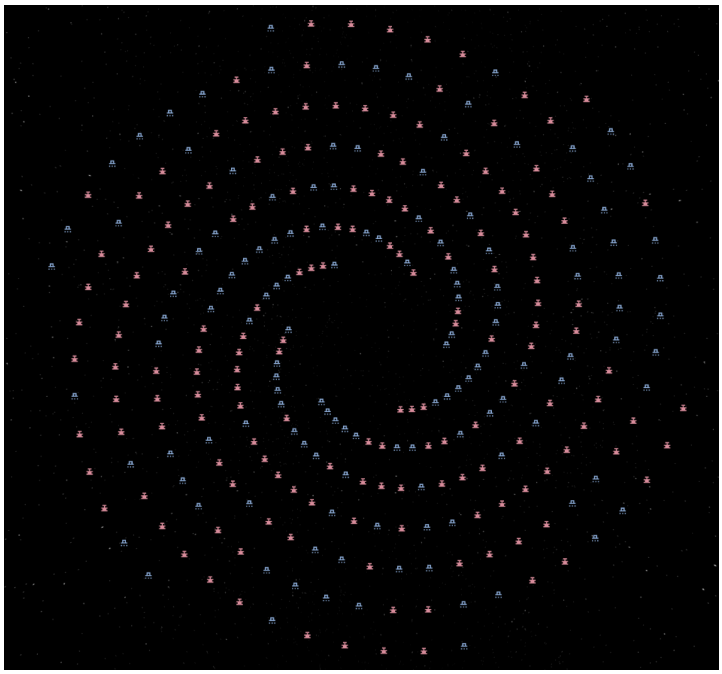

Figure 16.3: The Space Wars galaxy map showing an spiral galaxy with 6 arms and 300 systems

16.4 Summary

This chapter detailed how Space Wars was made scalable with the number of players. Basically, the implementation process consisted in:

1. Generating multiple planetary systems
2. Separating world structure from dynamic data
3. Rendering a spiral galaxy using `Identicons`

Once this was achieved, the first `MMPG` was finished.

Part IV

Evaluation

17 Validation

This chapter studies whether the project's secondary objectives described in chapter 3 have been fullfiled after the implementation process. Then, after that study, it ponders whether the solution obtained fullfills the main objective.

17.1 Secondary objectives

17.1.1 Develop an abstract game engine

During the entire implementation process, one of the main concerns has been to avoid adding any game-specific code in the engine component.

Moreover, the engine architecture is entirely decoupled from any game built on top of it.

Furthermore, the engine provides an abstraction that games can override to customize certain aspects: like world generation, game-loop timestep, etc.

Hence, the objective has been achieved.

17.1.2 Allow hot-swapping of AIs

This `AI` hot-swapping feature was implemented in chapter 12. Also, it has been tested constantly thanks to the implementation methodology described in chapter 8. Thus, the objective has been achieved.

17.1.3 Implement a real-time webviewer

The engine was able to notify game events in real-time since the first prototype implemented in chapter 9. Therefore, the objective has been achieved.

17.1.4 Create a control panel

The control panel was a feature implemented in chapter 13. Hence, the objective has been achieved.

However, it might be too simplistic, as it only allows to play and pause the match. Hence, more features will probably be needed before using it in a production environment, like monitoring players in real-time, banning players, etc. These features should be easy to implement by extending the engine, the `API` and the client.

17.1.5 Create a game example

Space Wars, the game example, was implemented during chapter 14, chapter 15, and chapter 16. Therefore, the objective has been achieved.

However, the game might need some tweaking and balancing before using it in the EDA subject. Private tournaments could be organised to test it properly.

17.1.6 Make the infrastructure scalable and stable

First, the platform was designed with scalability in mind:

1. The engine and the `API` can work from different machines.
2. Multiple engine workers can be used to distribute `AI` calculations in different machines.
3. Viewers can be statically served by a simple service like `Apache` [11] or `nginx` [26].

On the other hand, stability has ben assessed through continuous integration described in the implementation methodology in chapter 8. The platform has been working 24/7 since the first prototype was deployed, without a single crash and without any anomaly.

Figure 17.1 and Figure 17.2 show the CPU usage and disk IO in a period of 30 days in the integration server, respectively. An increase in CPU usage and disk IO can be observed when a whole galaxy was generated but it was fixed later, as it is described in chapter 16.

For these reasons, the objective has been achieved.

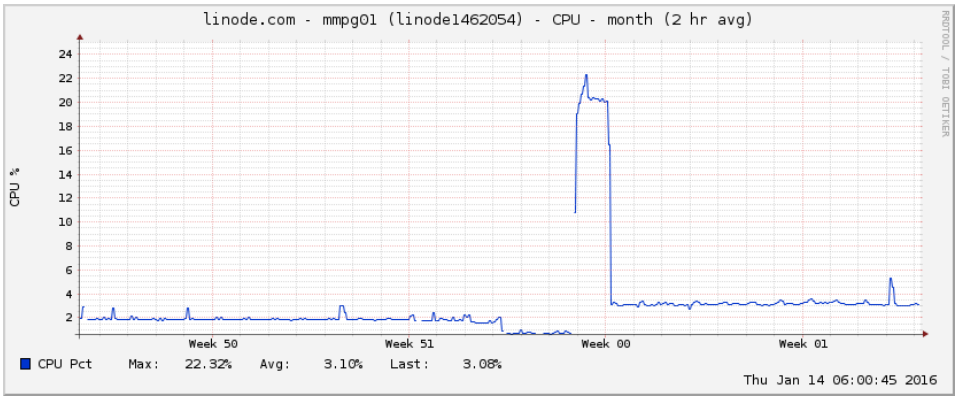

Figure 17.1: CPU usage of the platform in a 30 day period

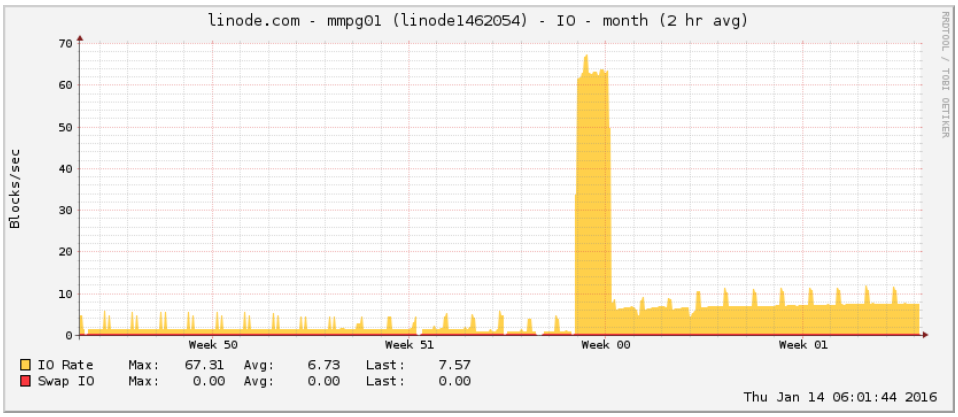

Figure 17.2: Disk IO of the platform in a 30 day period

17.2 Main objective

The main objective of the project is:

> *Develop a set of components that ease the creation and the usage of massive multiplayer programming games.*

The abstract game engine allows game developers to build the logic of an `MMPG`, while the client library allows them to build the viewer of an `MMPG`. Moreover, the engine supports `AI` hot-swapping natively, allowing players to change their programs. Also, a game example has been developed to show that the platform works: Space Wars.

However, real documentation about how to use the platform is lacking. While the game example could be used by developers to learn about the plat-

form, it is not easy to learn from code. Thus, the main objective might not be achieved completely until some documentation is written.

Once the lack of documentation is solved, it will be possible to say that the platform sets the foundations for a new type of programming games: the `MMPGs`, while providing a useful set of components to create them and use them. Thus, the main objective will be achieved.

18 Time management

This chapter details the final time arrengement of the project and compares it with the original time planning. It shows the final duration for every task in a timetable and a timeline of how tasks were performed, while it also studies the dedication given to the project.

18.1 Time table

Task	Planned duration (h)	Final duration (h)
Project management course	75	70
Analysis and design	10	20
Engine	70	100
API	50	30
Client	30	30
Control panel	35	15
Game example	100	100
Testing and polishing	40	40
Project memory	40	50
Oral presentation	10	10
Total	460	465

Figure 18.1: Time table

Figure 18.1 shows the final duration for every task of the project.

The main differences with the original time planning are:

- The engine took more time to implement than it was expected. Mainly because of the optimization performed in chapter 16.
- The `API` was implemented faster than planned. The Go programming language resulted really easy to learn and use, while the native libraries

made the entire implementation process really productive.

- The control panel was integrated in the viewer, not in the Jutge.org platform. As a result, it took less time to implement it, as learning the Jutge.org environment was not necessary.
- The project memory took more time to write than expected.

18.2 Timeline

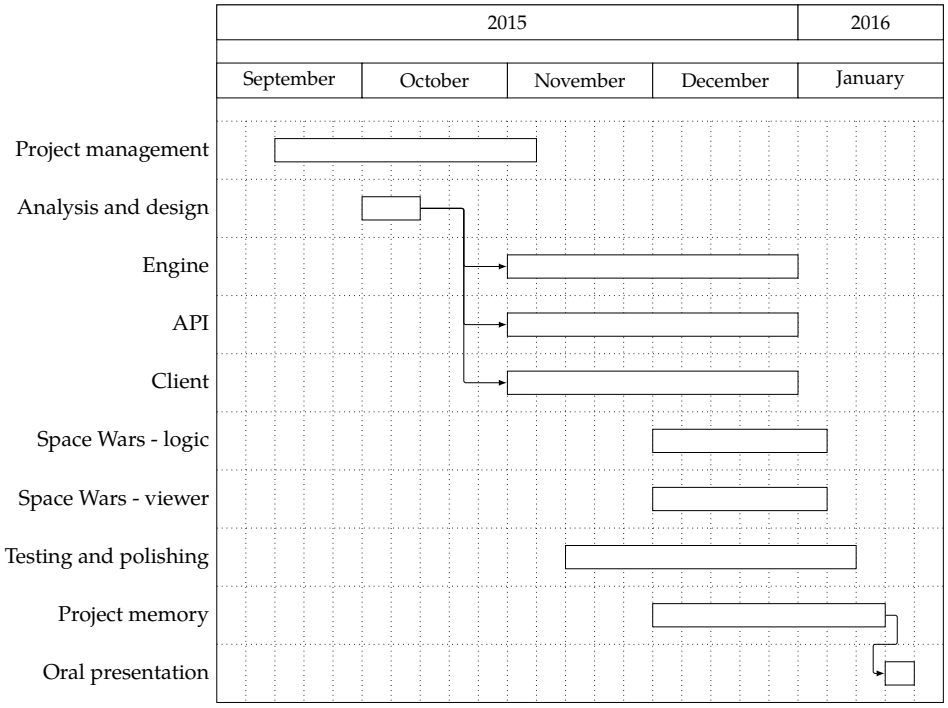

Figure 18.2: Final timeline

Figure 18.2 shows the final timeline for every task of the project.

The main difference can be observed in the testing and polishing task. As the implementation methodology used continuous integration (chapter 8), the testing and polishing of the platform has been constant since the first prototype (chapter 9). Also, the different components have been developed at the same time, in order to have a fully-functional prototype after each iteration.

18.3 Dedication

Figure 18.3 shows the relative amount of contributions per day to this project.

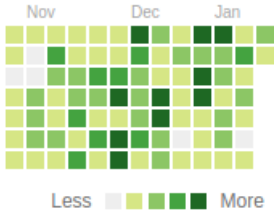

Figure 18.3: GitHub contributions to the `MMPG` project
https://github.com/hecrj

At least one contribution was made every day since the 2nd of November of 2015, with only two days being the exception. As a consequence, the project has been in constant progress; new features were added to the fully-functional prototype and were reported to the project advisor on a weekly basis.

Contribution peaks (*greenest squares*) happened when some important feature or optimization was implemented, which increased the dedication to the project during a specific timeframe.

It should be noted that Figure 18.3 does not give any type of feedback about the testing tasks performed in the platform during its implementation.

Overall, the amount of work has been distributed properly and the project has been developed at a steady pace.

19 Economic cost

19.1 Hardware resources

The project used the hardware resources specified in chapter 7 Budget. However, a Linode 1GB was needed during 3 months.

Hardware	Cost (€)
Hardware budget	60.13
Linode 1GB	27.47
Total	87.60

Table 19.1: Hardware cost

19.2 Software resources

As stated in the original planning, all the software needed to develop the project can be used for free. Table 19.2 shows additional software that was used but it was not listed in the original planning.

Software	License
`Three.js`	https://github.com/mrdoob/three.js/blob/master/LICENSE
`Identicon.js`	https://github.com/hecrj/identicon.js/blob/master/LICENSE

Table 19.2: Additional software

19.3 Human resources

Table 19.3 shows the final cost of the human resources according to the final task durations detailed in Figure 18.1 and the salary per role in Table 7.3.

Role	Task	Time (h)	Cost (€)
Project manager	Project management course	70	2450.00
	Project memory	50	1750.00
	Oral presentation	10	350.00
Software engineer	Analysis and design	20	800.00
Software developer	Engine	100	3000.00
	API	30	900.00
	Client	30	900.00
	Control panel	15	450.00
	Game example	100	3000.00
	Testing and polishing	40	1200.00
Total		465.00	14 525.00

Table 19.3: Human resources cost

19.4 Other resources

19.4.1 Electricity

Table 19.4 shows the electricity cost of the project, assuming the same cost per kWh stated in chapter 7.

Hardware	Consumption (W)	Time of usage (h)	Cost (€)
Desktop computer	400	440	17.60
Laptop	100	5	0.05
Monitor Acer XB270HU	30	440	1.32
Total			18.97

Table 19.4: Electricity cost

19.4.2 Internet connection

The project was developed using the internet connection described in chapter 7. The internet connection was used approximately during the 30% of the total project's duration, as expected. Thus, the final cost of the internet connection was $465\text{h} \cdot 0.05€/\text{h} \cdot 0.3 = 7.36€$.

19.5 Total

Table 19.5 shows the total cost of the project compared to the budget of the planning stage.

Resource	**Budget (€)**	**Total cost (€)**
Hardware	60.13	87.60
Software	0.00	0.00
Human	14 525.00	14 800.00
Electricity	17.70	18.97
Internet	7.28	7.36
Contingency (10%)	1461.01	0.00
Total	16 071.12	14 913.93

Table 19.5: Total cost

20 Sustainability

This chapter contains a basic sustainability report of the project based on the method described by Christian Felber in 'The economy of the common good' [10].

20.1 Economic analysis

The economical analysis was complete and thorough, all costs are assessed and reasonable. Resources were used efficiently, more time was spenth in the most important tasks. Moreover, the best technologies were chosen to produce the best results in the least amount of time possible. A contingency budget was added to take care of any unexpected problems.

However, the project is not aimed to be profitable, so there will be no direct economic benefit from it.

20.2 Social impact analysis

The project is directed to computer science students and teachers, and game progammers. The project will be used by students to learn different programming techniques, by teachers to evaluate these students, and by game programmers to create new content.

Moreover, it will be used in the EDA subject at the FIB. The project will make the learning process more fun, it will also reduce the amount of work for teachers, and it will make game programmers able to create entertaining games easily.

However, the project will not affect the mainstream consumer directly. It will only be relevant inside a specific community in computer science and education.

20.3 Environmental impact analysis

The project used 177kWh of electricity, which is approximately 174kg of CO^2 [6]. Also, any source of information was accessed digitally over the internet.

However, the hardware used can not be fully recycled and it will, eventually, become electronic waste with a high amount of contaminants.

21 Legality

There are no special laws that apply to this project. The entity using the platform is liable for the information inputted in it, as it is specified in the code license.

22 Conclusion

This last chapter closes the project by summarizing the document, detailing the future of the project and giving the author personal thoughts.

22.1 Summary

This document detailed the building process of a platform for MMPGs.

First, the requirement for MMPGs games was established. Then, it was shown that there was not an existing solution to satisfy it. Thus, the requirement was analyzed and a solution was designed. This solution featured a set of different components that allowed the creation of MMPGs.

Second, the development of the solution was planned. A time plan and a budget for the solution were shown.

Next, the implementation of the solution was detailed. Starting with a prototype and using continuous integration, the different features that the solution needed were developed.

Finally, the implemented solution was evaluated. The solution was checked against the objectives of the project, while also showing the time management during its implemenation and its final economic cost.

22.2 The future

The future of this project is promising. Given that the platform is released as open source, the project will be maintained by the author and, hopefully, by the community in the future.

Some of the tasks that will be performed in the near-future are:

Documentation: Different types of documentation will be written:

1. A guide for developers will explain how to start developing a game with the platform with detail.
2. A guide for contributors will help developers to contribute to the project.
3. A guide for Space Wars will teach players how to install Space Wars and develop `AIs` locally.
4. `README`s in every component will explain its responsibility in the platform.

Widget library: A library containing predefined viewer widgets will be implemented. Game developers will be able to use this widgets to implement viewer functionalities; like a login form, or the time control, for example.

Game template: A simple game will be created. This game might be used by game developers as a starting point. Forking the repository of this game will give game developers a simple fully-functional game out-of-the-box.

Game-logic testing suite: The engine will get its own testing suite, so game developers will be able test the logic of their games effortlessly.

22.3 Personal thoughts

It is difficult to express my thoughts after such a long project.

When I started, I did not imagine I would have to develop so many different components at the same time. However, I really liked the implementation methodology used in the project. Being able to deploy changes automatically and show them to anyone was motivating. Also, it surprises me how low the time spent debugging has been. Bugs were detected and fixed really fast.

I have learned a lot during this project. I did not know how to code in `Go` when I started. Right now, it is another tool in my toolbox. `Three.js` was also new to me, and I've been amazed by how powerful it is. I have also learned about the different low-level libraries to implement inter-process communication, which I had never used for anything serious before this project. Moreover, procedural generation was one of those topics that interested me but never tried. I have been constantly learning and mixing my knowledge with new concepts.

I have mixed feelings with the outcome of this project. I think it is really promising, but I would have liked to offer a more finished and better documented solution.

At the end, though, developing this project has been a really fun and re-

warding experience and I am sure it will benefit me deeply in my incoming professional career.

Bibliography

[1] Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International. http://creativecommons.org/licenses/by-nc-sa/4.0/legalcode.txt. [Online; accessed January 20, 2016].

[2] EDA, Estructures de Dades i Algorismes. http://www.cs.upc.edu/eda/. [Online; accessed September 23, 2015].

[3] Gamification. https://badgeville.com/wiki/Gamification. [Online; accessed January 20, 2016].

[4] JSON Web Tokens. https://jwt.io/. [Online; accessed December 18, 2015].

[5] The MIT License. https://opensource.org/licenses/MIT. [Online; accessed January 20, 2016].

[6] U.S. Energy Information Administration. How much carbon dioxide is produced per kilowatthour when generating electricity with fossil fuels? http://www.eia.gov/tools/faqs/faq.cfm?id=74&t=11. [Online; accessed October 18, 2015].

[7] CircleCI. Continuous Integration and Deployment - CircleCI. https://circleci.com/. [Online; accessed January 14, 2016].

[8] CodingGame. CodinGame - Programming is fun. https://www.codingame.com/. [Online; accessed January 14, 2016].

[9] Paul M. Duvall. *Continuous Integration: Improving Software Quality and Reducing Risk*. Addison-Wesley, 1st edition, 2007. ISBN: 9780321336385.

[10] C. Felber. *The economy of the common good*. Deusto SA Editions, 2011. ISBN: 9788423412808.

[11] The Apache Software Foundation. Welcome - The Apache HTTP Server Project. https://httpd.apache.org/. [Online; accessed January 14, 2016].

[12] Martin Fowler. Event Sourcing. http://martinfowler.com/eaaDev/EventSourcing.html. [Online; accessed January 10, 2016].

[13] Inc. GitHub. Identicons! https://github.com/blog/1586-identicons. [Online; accessed January 20, 2016].

[14] Travis CI GmbH. Travis - Test and Deploy Your Code with Confidence. https://travis-ci.org/. [Online; accessed January 14, 2016].

[15] Phil Hassey. Galcon. http://www.galcon.com/. [Online; accessed December 18, 2015].

[16] NASA/JPL-Caltech/ESO/R. Hurt. Artist's impression of the Milky Way. http://www.eso.org/public/images/eso1339e/. [Online; accessed January 20, 2016].

[17] iMatix Corporation. ZeroMQ. Distributed Messaging. http://zeromq.org/. [Online; accessed December 18, 2015].

[18] Amazon.com Inc. Amazon Web Services. https://aws.amazon.com. [Online; accessed January 14, 2016].

[19] DigitalOcean Inc. Simple Cloud Infrastructure. https://www.digitalocean.com/. [Online; accessed January 14, 2016].

[20] Statista Inc. Electricity prices in selected countries. http://www.statista.com/statistics/263492/electricity-prices-in-selected-countries/. [Online; accessed January 20, 2016].

[21] Jürgen Köller. Spirals. http://www.mathematische-basteleien.de/spiral.htm. [Online; accessed January 20, 2016].

[22] LLC Linode. SSD Cloud Hosting. http://www.linode.com/. [Online; accessed January 14, 2016].

[23] Makoto Matsumoto. What & how is MT? http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/ewhat-is-mt.html. [Online; accessed January 20, 2016].

[24] Douglas McIlroy. Darwin, a Game of Survival of the Fittest among Programs. http://www.cs.dartmouth.edu/~doug/darwin.pdf. [Online; accessed September 23, 2015].

[25] MIT. BattleCode - AI Programming Competition. https://www.battlecode.org/. [Online; accessed January 14, 2016].

[26] Inc. Nginx. nginx news. http://nginx.org/. [Online; accessed January 14, 2016].

[27] University of Waterloo. AI programming challenge. http://aichallenge.org/. [Online; accessed January 14, 2016].

[28] University of Waterloo. Google. Planet Wars. http://planetwars.aichallenge.org/. [Online; accessed December 18, 2015].

[29] Jordi Petit and Salvador Roura. Jutge.org. https://jutge.org. [Online; accessed September 23, 2015].

[30] Semaphore. Semaphore - Continuous Integration. https://semaphoreci.com/. [Online; accessed January 14, 2016].

[31] Cornell University. Types and Classification of Galaxies. http://www.astro.cornell.edu/academics/courses/astro201/galaxies/types.htm. [Online; accessed January 20, 2016].

[32] Peter Wen. Robot Game. https://robotgame.net/. [Online; accessed September 23, 2015].

[33] Wiki. ProgrammingGames. http://programminggames.org/. [Online; accessed September 23, 2015].

[34] Zachtronics. SpaceChem. http://www.zachtronics.com/spacechem/. [Online; accessed September 23, 2015].