UFR de mathématique et d'informatique
Université de Strasbourg

Runestones Devs' Gems Hunt

Final Report

Ahmet Berk Ates, Ugo Berton, Solène Besancon, Anna Callet,

Louise-Anne Corbé, Hassan El Sahily, Maxime Zingraff

Integrator Project – University of Strasbourg, 2025

Project management: Maxime Zingraff
Supervisor: Jason Golda, MSc

Table of Contents

Introduction

The game we've developed, called **Gems Hunt**, is a first-person shooter inspired by Pokémon Snap and Wii Sports Shooting Range.
It's a **3D, multiplayer, real-time game**. The aim of the game is simple: to **collect as many points**, and therefore gems, as possible, in order to become the most famous gems hunter. The game takes place in a mine, into which we descend by elevator.

Our game is aimed at all age groups, but particularly at teenagers. That's why we've adopted a cartoonish aesthetic.

Nevertheless, we want the game to be dynamic and have **its own atmosphere**. The game is therefore quite dark, lit by a few torches, and dynamic music adds to the adrenalin. A number of obstacles (bats and beams) appear along the way, complicating the player's task.

We'd like to thank Mr Golda for his support throughout the project and the freedom he gave us, as well as the entire teaching team, and in particular Mr Cateloin for his availability when it was necessary to reboot the servers.

Technical Aspects

I. Project Presentation

A) Rules and Gameplay

As soon as a game is launched, **all players appear simultaneously** on the wagon (maximum 4 per game). A countdown begins, giving players 10 seconds to prepare. Players have a first-person view.

Once the countdown has reached 0, **the wagon begins to move** along the tracks. As it progresses, gems appear at player height (figure 1). These gems come in different colors and shapes (to suit color-blind players too), and represent different values.

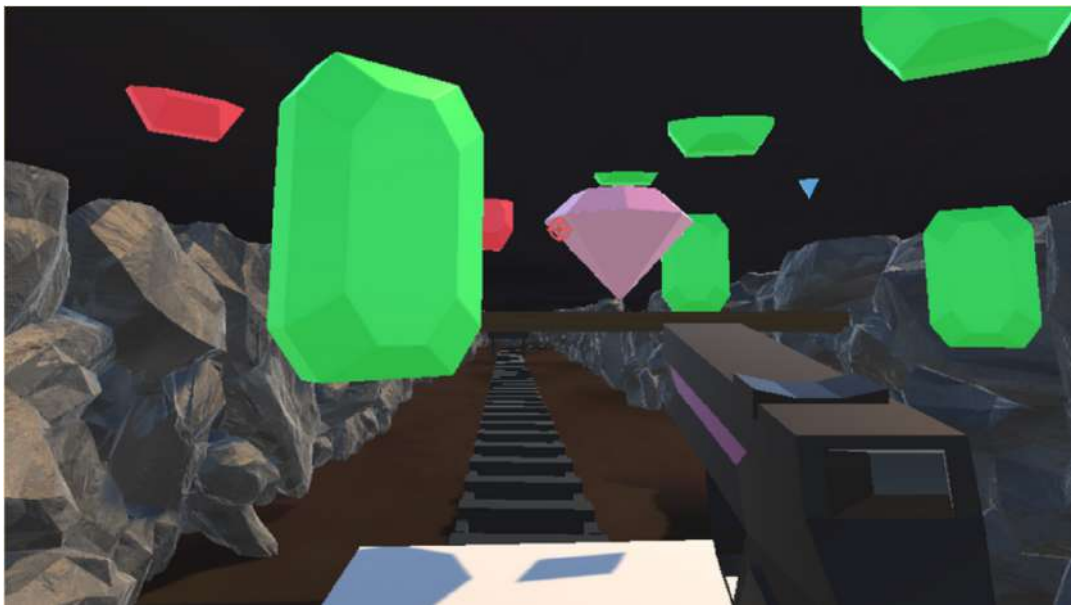

Figure 1: In-game gameplay

When a player succeeds in destroying a gem, their point counter increases by the corresponding value. The aim of the game is **to reach the end of the level** with **more points than your competitors**. It is not possible to shoot or push competitors.

When an obstacle hits the player (bats or wood beams), **they can't shoot for a few seconds** (figure 2). They can still move and pivot, but no longer shoot.

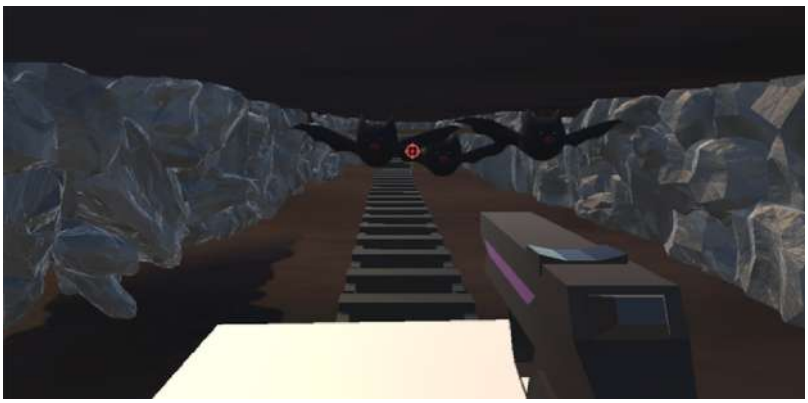

Figure 2: Bats obstacle

B) Game Modes

Normally, two game modes are available:

- play with friends, using their usernames or lobby code;
- a game mode with random opponents.

These two game modes were developed and tested in **alpha version**, but could not be implemented in the current version of the game. This alpha version therefore includes a game mode selection window, as well as a lobby to wait for all players to be ready before starting the game. The current game features **a single room**, based on port number. By default, without port modification, only one game is possible at a time (figure 3).

Figure 3: Welcome & login screen

C) User Account

Several features are linked to the **Gems Hunt user account**, which in future can be used for other games developed by Runestones Devs.

1 Account Creation and Management

Before you can use our user account, you need to create it. To do this, you'll need to **fill in your personal details**, your e-mail address and a password (figure 4).

Figure 4: Create account form

If you've forgotten your password, simply enter your e-mail address, and **you'll receive an e-mail** from `info@runestonesdevs.com`. This e-mail contains a verification code valid for 5 minutes, which you can then enter in the dedicated window to reset your password.

A number of **security features** are provided for account creation and password resets, to prevent SQL injections and insecure passwords.

2 Managing Friends

Once your account has been created, you can **manage your friends.** You can send requests, accept or reject requests (figure 5), or delete friends.

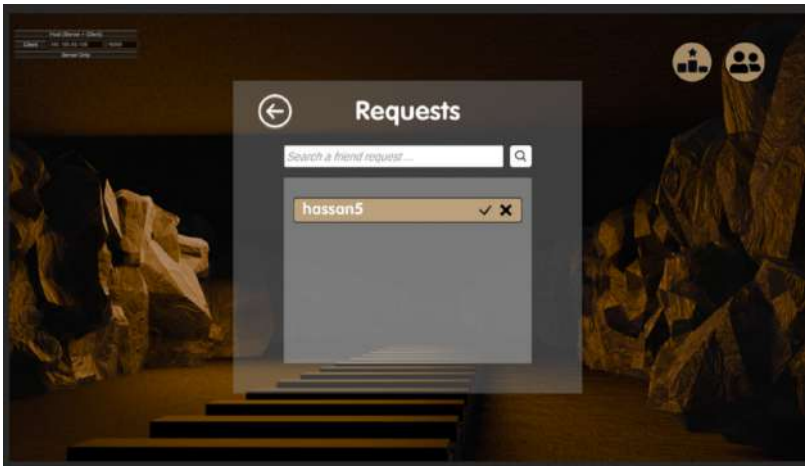

Figure 5: Incoming friend request

For the moment, some additional friend-related functionalities mentioned in V.H) are not implemented in the game.

3 In-game Scores and Leaderboard

As each user is identified by his or her Gems Hunt account, scores can be tracked in-game. A regularly refreshed **leaderboard** identifies the **top three players**, which will be recalled at the end of the game.

These scores are then added to the player's previous scores, which can be **consulted between games**. An overall leaderboard summarizes the top three scores from all entries.
At the time the report is written, the link with the API and the database is not operational but the whole infrastructure is working.

D) Complete Game Play

The complete course of a game can be summarized by the algorithm in figure 6.

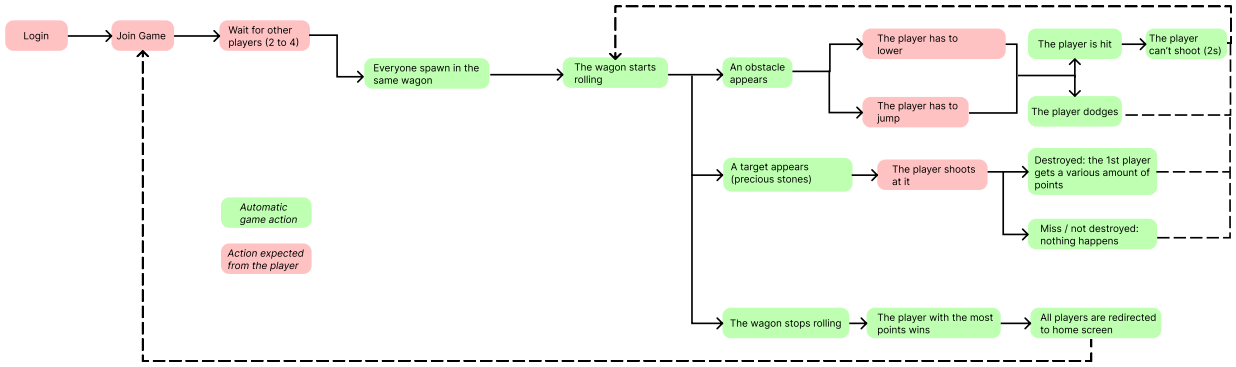

Figure 6: Game session algorithm

E) Music and Sound Design

To make the game more enjoyable and contribute to its cold, special atmosphere, we've created a **sound environment**.
This starts with the sound effects, such as when the gems are destroyed.

We also spent time choosing the music. The music we chose **forces concentration**, and **motivates the player** to finish first.

F) Game Platforms

Our main platform remains the computer, through a **downloadable "heavy" desktop client** available for **several operating systems**: Windows, Linux systems, and macOS.
However, the installation of a heavy client may hinder potential players who don't have the authorization to install applications, the required internet connection, or who simply wish to play on computers that don't belong to them.

We have therefore **developed a Web version**, identical to the "heavy client" version and accessible at the following address: https://mai-projet-integrateur.u-strasbg.fr/vmProjetIntegrateurgrp6-1/. On this site, you can also download the heavy client and find out more about the game and the Runestones Devs studio. This website is **totally responsive on any device**.

II. Technical Choices

A variety of technologies were used, **depending on the needs** of each sub-team.
As a reminder, in the initial specifications, we plan to use the following technologies: **Blender** for modeling, **Unity** for the game engine and game logic, **Mirror** for networking, **Microsoft's .NET** framework for global server management, **MySQL** for the database, and **WebSocket** for communications between database and clients. For the web version, we planned to use **WebGL**.

Interactions between technologies and their environment is modelized in figure 7.

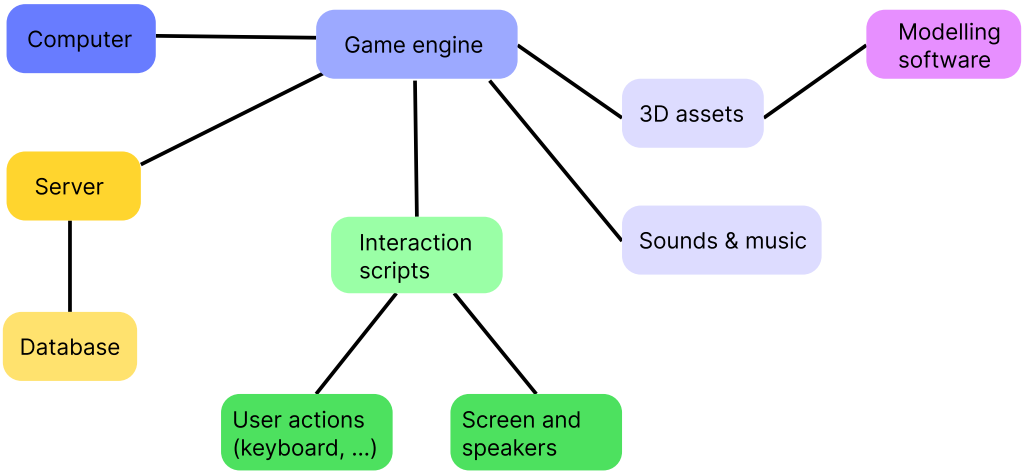

Figure 7: Infrastructure

A) Modeling

We made good use of Blender for modeling. Indeed, it's a **well-known platform** offering a wide range of features. In particular, it's the platform we used at university to learn how to model objects, so we didn't have to spend a lot of time relearning on another platform.
Blender also enables us to **create skeletons** and thus **animate our models**, which we did for the player and the bats, for example.

We didn't encounter any particular problems with this technology, and would **make the same choice** again in the future.
The only drawback was that students in the "network" specialty didn't have the time to learn this skill, and so it fell mainly on the "image" specialty.

To model the mine, we opted for a clever approach based on our "cartoon" style. We modelled just a **few walls** with muliple rocks, with little detail so that they weren't easily distinguishable. Then, these walls were **randomly assembled** and **copied many times**, to create the mine walls that **look different at each location**, when in fact they're not.

Some assets suffered import problems in Unity, which have now been resolved.

B) Game Engine and Logic

Game engine technology is **key**, as it is the link between the various elements and other technologies. As indicated in the specifications, we chose Unity.

It's a game engine that's easy to get to grips with, and offers **easy debugging and testing** of programs. It's a solution **used in the video game industry**, and also contains a lot of documentation and tutorials on the Internet.

All our team members have had the opportunity to work with Unity, to varying degrees. The advantage of Unity is that it integrates **numerous build possibilities** from the same project, generally without compatibility concerns.

We were able to generate server builds (Linux Server) in **headless mode**, i.e. without graphics; **client builds** (Windows, macOS, and Linux); and **web builds**. These features work well, except for the WebGL compiler, which had to be adapted to be compatible with our server, and which remains very slow to create compressed and optimized builds (around 1h30).

C) Networking the Game

To ensure that our game could be played in a remote multiplayer environment, we had to implement a component that **links to the server** so that the movements of one player are clearly visible to the others. In our specifications, we had identified Mirror Networking, a Unity-based solution.
This solution is recognized as the one to use for small projects. What's more, the technical side of network management is handled **almost automatically**. We therefore chose this solution, which has enabled both "network" and "image" team members to work with it.

A further advantage is the availability of documentation, which is quite abundant, and the **active Mirror community**.
The alternative mentioned in the specifications, Fish-Net, offers better performance and more customization. Nevertheless, we preferred to keep things simple and waste as little time as possible on research and documentation.

Mirror **quickly reached its limits**. As soon as we put the game on the network for the first time, with a maximum of four players, Mirror **created new bugs** for us, particularly on the wagon. As for matchmaking, Mirror doesn't really parallelize games, but rather manages instances that are visible or invisible to other players in other games. Mirror accepts a **maximum of 200 players**, which can quickly become a limitation when scaling the game.

Nevertheless, Mirror **brings security**, thanks to anti-cheating features such as player movement consistency control: if a malicious player asks to be teleported upstream, Mirror compares with the last known positions and **does not authorize such a move**.

We would have liked to see **a solution somewhere between Fish-Net and Mirror**, i.e. simple to use but effective and well-designed for matchmaking, but to our knowledge no such solution exists for Unity.

D) Managing Database Server Requests

To communicate with the database, we planned to use WebSocket in combination with Microsoft's .NET framework. Several team members were familiar with these frameworks, which are also recognized as an efficient solution widely used in the video game industry.

Right from the start of our research, we realized that using **WebSocket wouldn't necessar-**

ily be useful in our case. In fact, **.NET offers all the features** needed to implement an API and communicate with it directly from the client.

We communicate with our .NET API via **HTTPS requests**, which guarantees secure communications.
We're **happy with this choice** of technology, which remains simple to implement for simple uses like ours, but which can become much more technical and advanced if necessary in future game updates. Thanks to Microsoft's dominant position in this field, a great deal of documentation is available, making it easy to learn the framework.

Initially not included in the specifications, we have implemented **an SMTP server** which, in conjunction with the API, sends password reset e-mails.

E) Database Management

For the management of our database, we opted for efficiency and simplicity with a **SQL database** and a MySQL database management system. This is close to Oracle, currently being studied, and easy to deploy on the server.

We're happy with our choice, which justifies MySQL's position in the market, which is p**erfectly suited** to modest databases like ours.
In particular, we used **triggers** to close inactive sessions in the session table after 2 hours of inactivity.

F) Web Porting

As mentioned in the specifications, we used WebGL to port our game to the web.
WebGL is recognized as **one of the most powerful solutions** for our type of use, and is **easy to deploy** as long as browsers are **compatible**.

We didn't have to make any in-depth modifications to the server to accommodate WebGL builds, apart from **some configuration scripts**. However, our server does not natively manage build compression/decompression, which must be handled by an additional tool added at the time of the Unity build. We also used **Multiplex Transport**. Currently, the university's server doen't allow us to deploy the game.
Although WebGL debugging is complicated, when there are no problems it's a smooth and functional solution on most machines.

III. Architecture and Interactions

Our game is an online multiplayer game, based on a **centralized two-server architecture**.

A) Servers

We're using two servers hosted by the faculty, which are two Ubuntu virtual machines (`vmProjet Integrateurgrp6-0` and `vmProjetIntegrateurgrp6-1`). We've chosen to distribute our various services over these two servers because:

- In the event of a server crash, some services are **still available**
- The load is more **evenly distributed**
- Hacking into one server **doesn't compromise** what's on the second.

With this in mind, we have distributed services as follows:

- **Server 0:** Unity headless instances, Mirror, API, SMTP server
- **Server 1:** MySQL database, web server (Apache)

B) Unity & Mirror

Unity instances are built to act as servers, and therefore **centralize all connections** on port `10000` (internal) as soon as a connection to the bastion via port `16000` is received. The Mirror implementation in the Unity build then **handles synchronization** between clients and object management (players, gems, balls, scores, etc.).

C) API & Database

If the client sends an HTTPS request to the API via port `16001`, it will be processed by the API, which will **check for SQL injections** and **convert the HTTPS request** into an SQL request. Some SQL queries are performed by triggers. These include the table listing connections, with a login timestamp and a logout timestamp. However, if the player's computer suddenly shuts down (bug, power failure, etc.), the computer **doesn't have time** to send the request to close the session. For security reasons, this trigger a request and indicates that the session is closed after 2 hours of inactivity.

Our database organization is represented by figure 8.

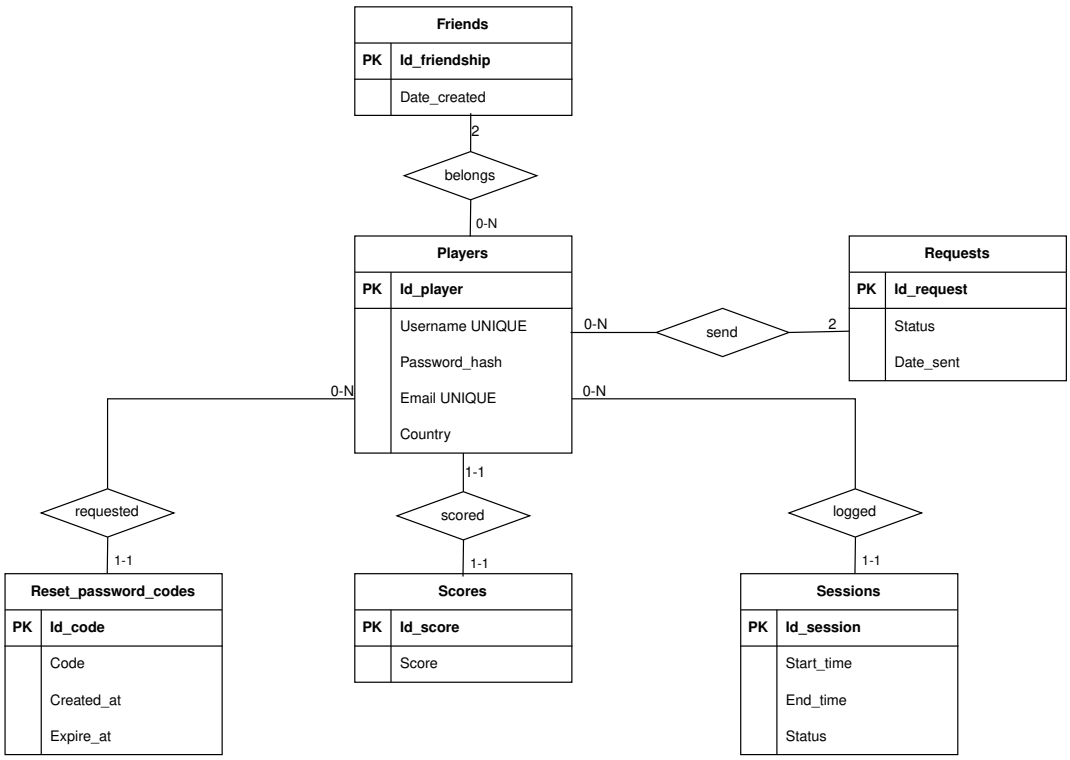

Figure 8: Database diagram

D) Web Server

We use an **Apache server**, which provides access to our website and builds via port 80, which can be accessed with a redirection performed when accessing https://mai-projet-integrateur.u-strasbg.fr/vmProjetIntegrateurgrp6-1/.
This server uses **configuration files** to integrate compression in gzip format, and to support WebGL builds generated by Unity.
However, at the time this report is written, the web version is not working on university's servers, but totally working on a localhost server. The problem is probably comming from configuration files.

E) Client

Finally, the role of the client can be determined by what was mentioned above. It **never communicates directly** with other clients, but only with Mirror **via the Unity server** for its movements; and **with the API** for connection/disconnection, password reset, friend system, score recording.

F) Global Architecture

The summary of our architecture is described in figure 9.

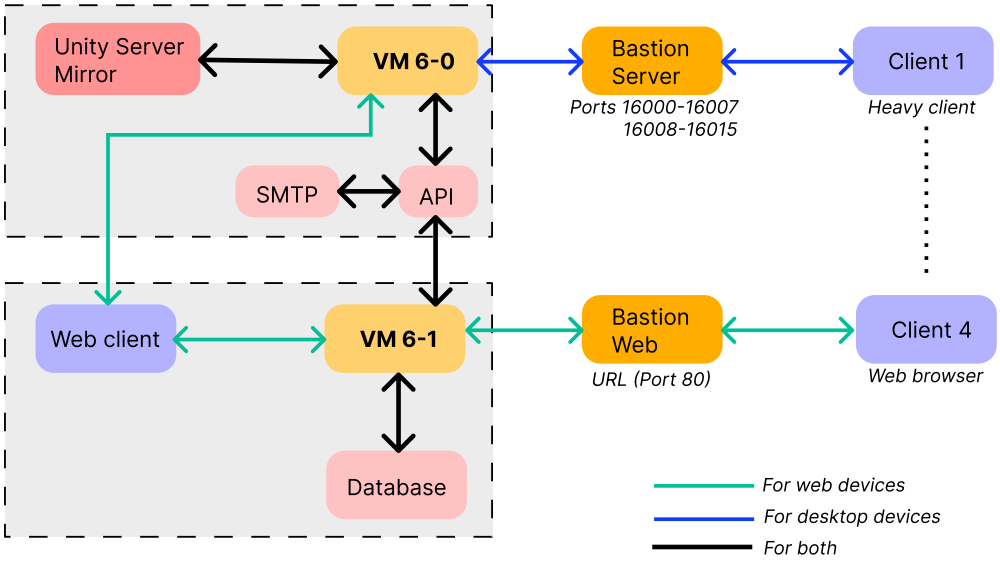

Figure 9: Project architecture

IV. Implementation Details & Challenges

This section aims to describe the chosen implementation for some interesting topics, as well as the difficulties encountered during implementation.

A) Wagon

The wagon is probably the **most complex implementation** of our game, and impacted many other implementations.
It's no easy task to move an object on which there are 4 players, who can also **move on the surface of the wagon** without falling off. Add to this the need to synchronize the movement of the wagon between all the customers.

The main problem was therefore to **keep the players on the surface** of the wagon. The player's prefab was a problem for us, so we replaced it with a **rigid body**, which posed less of a problem.
To prevent the player from falling, we installed **walls around and above the wagon**. But by positioning themselves in a corner and jumping in a particular way, players **could still fall off** the wagon.
As we reworked these walls and the wagon prefab, we kept getting errors. To counter this, we've implemented a **teleportation system**, which ensures that the **player remains in the wagon**, and even if they does fall, they'll be repatriated to the wagon.

Once the wagon was playable in online multiplayer and deployed on the server, we encountered **visual problems**: the wagon appeared rather jerky, as did the opponents. Following a code simplification, **this problem has been resolved**, and the wagon is now fully playable.

B) Online Multiplayer

We're using the **KCP**[1] protocol, an ARQ *(Automatic Repeat reQuest)* protocol that provides a degree of connection reliability (like the TCP[2] protocol, with **much lower latency** thanks to the use of UDP[3]. This protocol uses $10\% - 20\%$ **more bandwidth** than UDP because of its header, but offers $30\% - 40\%$ faster transmission than TCP[4]. So it's a **good compromise** between secure communications and low latency.

As mentioned in II.B), integrating Mirror into the game wasn't easy. In particular, the way we built our project didn't allow us to easily integrate the NetworkManager and thus network the game's elements.
It was therefore necessary to **practically recreate the project**, making each game element compatible with the NetworkManager, which took longer than expected and created inexplicable bugs such as the wagon(IV.A)). In future, it may be more appropriate to **design the game with network dependencies** in mind, and therefore **to include the NetworkManager** as soon as each object is created.

[1] Linwei. "Skywind3000/Kcp." C, May 13, 2025. https://github.com/skywind3000/kcp.
[2] "Transmission Control Protocol." Request for Comments. Internet Engineering Task Force, September 1981. https://doi.org/10.17487/RFC0793.
[3] "User Datagram Protocol." Request for Comments. Internet Engineering Task Force, August 1980. https://doi.org/10.17487/RFC0768.
[4] Ibid, https://github.com/skywind3000/kcp/blob/master/README.en.md

C) Matchmaking and Lobby

Matchmaking was **a key feature** in the specification, but we didn't have time to implement it in the main version of the game, so it remained in alpha.
Mirror is not designed to implement matchmaking mechanics natively. For this purpose, we have relied on an implementation proposed by Jared Brandjes[5], which brings together a community and therefore makes it possible to find assistance.

This implementation uses a **single instance** of Unity server, which handles all requests within the 200-player limit imposed by Mirror. These players are then divided into different lobbies (figure 10) depending on the game mode chosen:

- join a public game at random;
- join a public or private game using a lobby code;
- create a game (public or private)

Each lobby has **a unique code**.

Figure 10: Waiting lobby

Then, all players are in the same lobby, and the **display is managed according to the game they are in**. In concrete terms, players in game 1 will only see those in game 1, and not other players.
The same applies to the game scene: the server manages a single scene, and display differentiation is managed according to the game code.

We've **finalized the entire UI** for selecting rooms, **as well as the lobby**, but we haven't been able to make the link with the game scene.
The problem encountered is similar to that of multiplayer networking with Mirror: practically **everything has to be recreated** in the game scene, so that all objects specific to each game (i.e. all objects except the scenery) are not visible to players outside the game.

Given that matchmaking is a feature that needs to be finalized before progress can be made on other issues, it represents a sticking point in the game's development. Had the game's design been geared towards the **matchmaking philosophy from the outset**, implementation would have caused fewer problems.

[5] https://gitlab.com/4t0m1c

D) HTTPS Certification of API Requests

For security reasons, it's essential to **communicate securely** with the database via the API. We therefore wanted to communicate over HTTPS rather than over HTTP.
To achieve this, we need to **encrypt our messages**, using a public and a private key.

To ensure that these keys cannot be falsified, and thus avoid "Man in the Middle"[6] attacks, the public key used must be **recognized by the computer**. It's not enough to simply enter it into the system: the key must be validated by a certification authority.
Generally speaking, you have to pay to have your key recognized by a **certification authority**, or you can use free solutions such as Let's Encrypt[7]. However, these free solutions only apply to domain names, and it is not possible with these tools to certify communications addressed directly to an IP address.

We therefore had to **create our own root certification authority**, with various programs to generate public and private keys signed by the key of this root certification authority (figure 11).

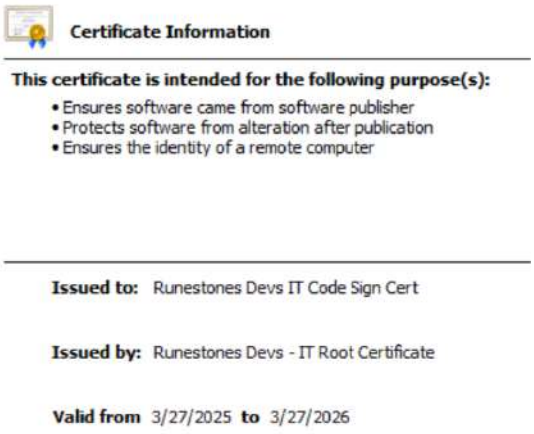

Figure 11: Certificate information window

However, this certification authority **is not registered by any device**, and it is therefore necessary to **manually add it** to the root certification authority stores of each device.

E) Problems accessing Virtual Machines

We often encountered **problems accessing the two VMs** to which we had access. In fact, these VMs are not very efficient, and therefore **need to be rebooted frequently** to kill all parasitic processes. As we couldn't reboot or reset them ourselves, we wasted a lot of time asking our teachers to do it for us.

What's more, this lack of performance is **variable**: depending on the time of day or the day of the week, performance fluctuates. The theory is that it also depends on what other groups are doing with their VMs, or what other programs are running on the server hosting those VMs. Indeed, we often had better performance late at night or early in the morning, while it

[6] F. Callegati, W. Cerroni and M. Ramilli, "Man-in-the-Middle Attack to the HTTPS Protocol," in IEEE Security & Privacy, vol. 7, no. 1, pp. 78-81, Jan.-Feb. 2009, doi: 10.1109/MSP.2009.12

[7] https://letsencrypt.org/

decreased in the afternoon, to the point where we were often disconnected from the VMs.

The fact that the VMs have to be accessed by **being connected to the university network**, either physically via eduroam / osiris or via the university VPN (vpn.unistra.fr) makes the job more complex.

F) Performance Tests

We conducted **performance tests** to ensure that our application runs efficiently under expected workloads and does not introduce significant delays or excessive resource consumption. These tests are essential to **detect potential bottlenecks**, assess system responsiveness, and verify scalability, especially in scenarios involving multiple users or complex interactions.

To carry out the tests, we simulated various usage conditions using automated tools (load testing frameworks, figure 12). We measured key metrics such as **memory usage, CPU load, and disk usage**. This allowed us to identify performance-critical areas and optimize them before deployment. Our goal was to guarantee a smooth user experience and avoid performance degradation, particularly in high-load environments.

To do this, we ran a game with **four players**.
We can see that resource usage **remains stable**, except for CPU usage, which is hi**gh at the start** of the game, and then spikes according to game events (massive appearance of gems, etc.)
RAM usage remains stable, and **decreases at the end** of the game, when all players return to the menu.

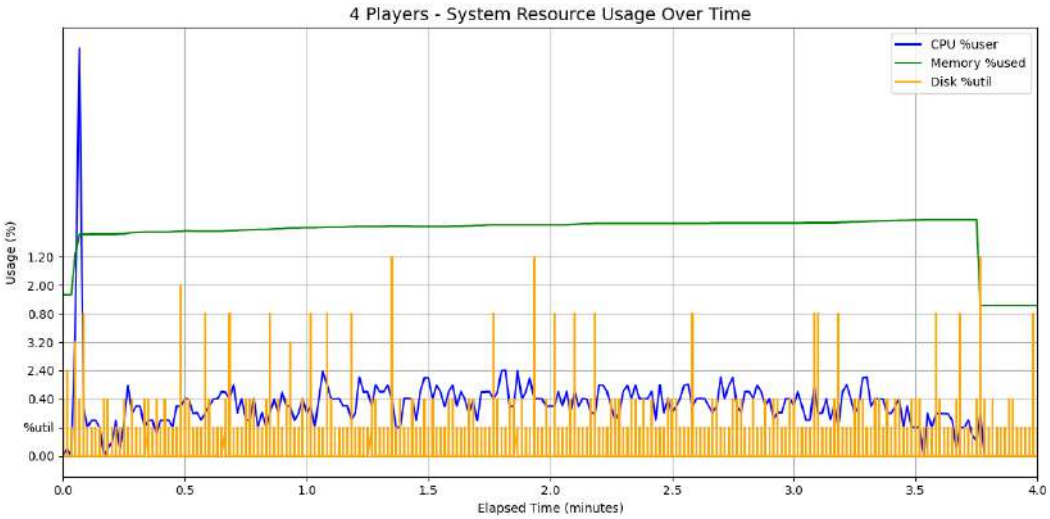

Figure 12: Performance graph

V. Prospects for Improvement

A) Requirements Document Comparison

Almost all the features mentioned in the requirements document can be found in the game. Given that we each spent between 150 and 200 hours on the project, **we couldn't do more in the time available without sacrificing other school subjects**.

Matchmaking and the lobby (IV.C)), which have a special status, were **developed in alpha version** but not implemented in the game.
Another feature that was not developed, but which was marked as a "development option" in the requirements document, is a game mode allowing **one wagon per person**. This game mode would have made it possible to have one track (and therefore one wagon) per player, with moments when these tracks disperse, and moments when they recross. It would probably take **40 hours** to develop this game mode.

What's still missing is the **transparency of players when they collide**. This doesn't happen very often, as there's little movement in the car, and it's more a case of shooting at targets. But when collisions do occur, nothing in particular happens. Developing a ghost mode in the event of a collision would surely take around **20 hours**.

B) Branch Discrepancies

One area where we could improve the organization of our code is **branch management**. Indeed, we quickly created numerous branches, in order to dispatch tasks and avoid interfering with others. Although this was a good practice, these branches began to remain **open for far too long**, and thus to diverge sharply from the main and/or other branches.
During merges, it sometimes took **several days** to resolve all conflicts and debug.

We should have created **smaller branches**, created for just a few days, to ensure that there weren't too many divergences.
Although some people were more affected than others, on average, better organization of the branches could have saved **20 hours per person**.

C) Matchmaking

As mentioned in section IV.C), matchmaking could unfortunately not be implemented in the final game and remained in alpha version, as it would have been necessary to recreate almost all objects to make them **visible** or **invisible** depending on the lobby, except for the scenery.

We estimate that **70 hours** would be needed to finalize matchmaking and implement it in the game.

D) VM Management on our Own Server

Although this represents an additional cost, it would have been simpler to **manage VMs on our own server**. This would have solved two problems:

- access and performance problems would be better controlled and could probably be re-

solved more easily. Estimated average gain: **3h / person**;

- the certificate used to establish an HTTPS connection could have been verified by a solution such as Let's Encrypt (which requires the purchase of a domain name, but which we already own), and therefore **verified by the root certification authorities** present **by default** on the vast majority of devices, making our game more accessible to the greatest number of people.

  Likewise, we wouldn't need to be on university premises to launch a game from the heavy client. Estimated deployment time: **5 hours**.

E) Further Customization

This is not a priority, but we would have liked to inc**lude more customization** options for players.

- **Skins**: it would have been interesting to offer more options in the choice of skins, such as different accessories or morphologies.
- **Game theme customization**: dark, light, etc.
- **Accessibility options**: menu font zoom, color-blind mode, game translated into more languages.

Development of these features is estimated to take **30 hours**.

F) Asset Animations

Staying with the aesthetic aspect, we would have liked to **perfect the animations** of certain assets:

- Wagon: we could have made the wagon more realistic by implementing tremors and small tilts on the longitudinal axis;
- Player: the player's movements still lack realism. This would require the creation of a more **complete skeleton** on Blender, and the refinement of skeleton element relationships according to rotations.

These animations would take around **20 hours** to complete.

G) Web Version

Although our web version is functional and playable, there are still **adjustments to be made** to the front end and backend.

- Front-end: **improve the visual aspect** of the showcase site, and include more content. For the WebGL version, it might be appropriate to adapt the game, while maintaining cross-platform compatibility, to suit the slightly different target audience;
- Back-end: make the WebGL build **more efficient**, by adding native gzip decompression on the Apache server;
- Make sure that the **university server** is **correclty configured** to run the build.

This would require around **30 hours**.

H) Friend System Features

One of the first things we'd like to improve is the **friend system**. In fact, the entire infrastructure is developed and functional, but not yet fully exploited. This is due to the late completion of this system. We'd like to add :

- the ability to join friends' games/lobbies;
- send messages to friends, create group chats;
- share profiles with other friends;
- add a status (online, inactive, offline).

All these features would require around **50 hours**.

Project Management

VI. Project Management Tools

A) Requirements Document

All our work has been based primarily on the **requirements document**. In fact, this document was worked on with the team, and therefore integrates the functionalities that were decided collectively. As project manager, it's a document I came back to often, to make sure we were on the right track and that no key functionality was left out, which worked out rather well.

Although some adjustments were made (V.A)), the end result is **pretty close** to what had been planned in the requirements document.

It might have been useful to **detail the tasks** in greater detail, as this might have enabled us to better anticipate heavy tasks such as matchmaking, so that they could be taken into account **right from the start** of development.

B) Versioning Tool

We used Git to version our code. In order to share it, we used the university's GitLab server to collaborate on the code.

To avoid all projects ending up in the same place, we organized our Git as follows:

- A "**Runestones Devs**" group brings together all the different Git repositories.
- The "**Gem Hunter**" Git[8] integrates all Unity and Blender developments linked to the heavy client (desktop), as well as files linked to the server build. It includes scenes, assets, scripts, configuration files, prefabs, etc.
- The "**Gem Hunter - Server 0**" Git[9] gathers, as its name indicates, the files present on Server 0. It is located in the `/home/ubuntu` directory of `vmProjetIntegrateurgrp6-0`, and therefore contains the Unity server builds, as indicated in III.A)
- The "**Gem Hunter - Server 1**" Git[10] contains the MySQL database, located in the `/var/lib/mysql` directory.
- The "**Gem Hunter - Web server**" Git[11] contains the files making up our website, i.e.

[8] https://git.unistra.fr/runestones-devs/gem-hunter
[9] https://git.unistra.fr/runestones-devs/gem-hunter-server0
[10] https://git.unistra.fr/runestones-devs/gem-hunter-server1
[11] https://git.unistra.fr/runestones-devs/gem-hunter-web-server

the files in the `/var/www/html` directory, including our game's showcase site, the various WebGL builds, and the heavy client installation files, depending on the desired platform.

To homogenize our code and **limit conflicts**, we created the following Git and GitLab usage policy with tech lead Solène (see appendix **??**)

C) Definition of Done

The implementation of the Git policy enabled us to homogenize our code, but some tasks in the backlog were marked as "completed" even though bugs remained during certain actions, or certain changes led to security problems.

To remedy this, Mr Golda suggested a wide range of tools. I then suggested that the group set up a "**Definition of Done**" (DoD).
This definition, which remains general to adapt to all tasks, **establishes the main principles** and helps **avoid major disruptions** to the project.

Obviously, the DoD **complements the specification** created for each task in the backlog.

- Merge with main branch or other intermediate branch
- Code reviewed
- Acceptance criteria met
- Functionality documented if not trivial
- Interaction between pre-existing objects not affected
- Functional tests
- No impact on game multiplayer
- No major impact on server load
- Communication security guarantee

D) Gantt Chart

For such a long project, it's essential to plan the tasks to make sure you have time to do them all.
I therefore relied heavily on our project's **Gantt chart**, managed and edited using the Gantt Project application.

The latter was **updated frequently**, to take into account delays on certain tasks, add new ones, and update the buffer time (often to reduce it, but on a few occasions to increase it).
A lot of **buffer time** had been planned, as this was our first project of this scale, so we probably misjudged the time needed to finish each task.

Each time the Gantt was updated, the team was of course **kept informed** and **consulted** before implementation, and their feedback was taken into account to ensure that the deadlines were closer to reality.
You can consult the Gantt Chart in appendix B..

E) Timesheets

I used my Excel skills to develop timesheets that were as **ergonomic** as possible.
Apart from a few bugs that were quickly resolved or caused by misuse, these timesheets proved their worth and enabled **efficient time tracking** throughout the project.

Team members simply had to enter the start time, end time and a justification. A box explains the instructions:

- Add your hours with this format: `XX:XX`
- After each work session, add one of the following justifications:
  - GitLab commit link
  - Documentation read
  - Meeting name & short summary
  - Other justification if necessary

Some cells are **protected**, to prevent team members from modifying them: these are the cells that perform **sums**, to prevent them from being inadvertently modified.
In addition, timesheets are **private**: only the tutor and I have access to everyone else's timesheets, to **avoid peer pressure** on team members.

Timesheets include **various statistics**: a daily sum, a monthly sum and a global sum. The most useful tool is the **tracking curve** (figure 13): this includes the sum per week, as well as a plot of the cumulative sum. Linear "ideal" scenarios are modeled: a 150h scenario, and a 200h scenario, which are the minimum and optimistic limits.

To ensure stable progress, it is advisable to remain at or above the minimum scenario at all times.

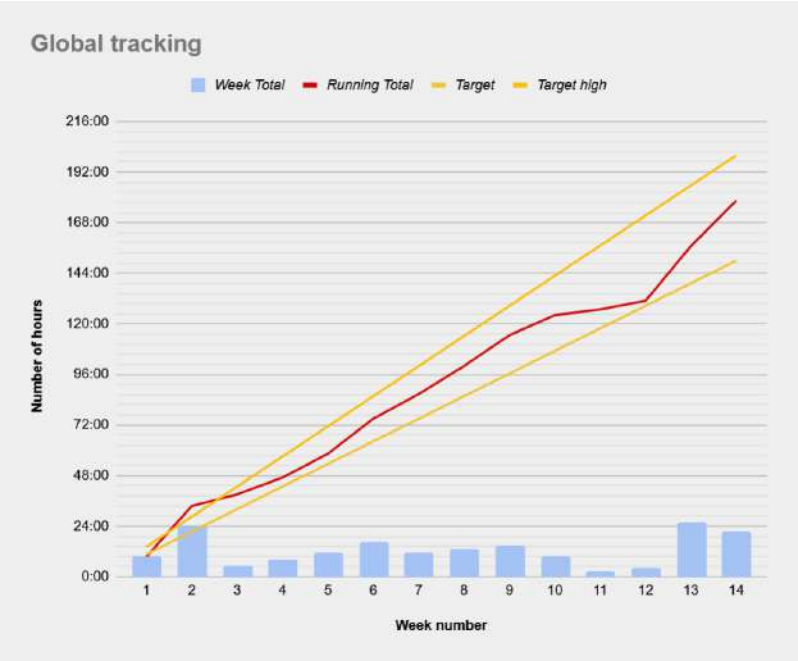

Figure 13: Timesheet charts

F) Backlog & Availabilities

After discussion with the tutor, we set up a **backlog** (figure 14). This document **groups together** the various tasks, which have different statuses: to specify, to do, doing, review, done.
A **person is assigned to each task**, which contains a start date and two reviewers. A hyperlink details the task: task number, duration, title, specifications, prerequisites and acceptance criteria.

Here are the filling instructions:

- Don't create new tasks directly: send a message to the project manager for the 1st draft to be created.
- If you have any suggestion regarding an existing task (regardless of its status), please send a message to the project manager and we will refine them together.
- For "to-do" tasks, estimate the time needed and assign the task to yourself.
- Once you are in charge of a task, you have to update its status!
- If you see a task with no reviewer, feel free to put your name!

ID (Tasks)	Name	Status	Start date	In charge	Review 1	Review 2	Comment
13	Implement Mirror in the test scene	DONE		Louise-Anne			
14	Fix client script in online test scene	DONE	14-Mar-2025	Louise-Anne			
15	Write lobby script	TO DO					Not urgent
16	Parallelisation research	DOING	19-Mar-2025	Louise-Anne			
17	Authentification	DONE	17-Mar-2025	Hassan			
18	Modelisation: Player	DONE	19-Mar-2025	Anna	Ahmet		(mesh + rigging, no

Figure 14: Backlog extract

34 tasks were created, but this tool **showed its limitations** at the end of the project, where many small, elementary tasks didn't necessarily require detailed specifications. We therefore stopped using it at the end of the project to **move forward more efficiently**.

Finally, this document contains an availability sheet (figure 15. This allows each team member to indicate his or her availability, and thus **plan task allocation more efficiently**. Different statuses are possible:

- Green: Fully available
- Yellow: Max. 1h/day (exams, etc.)
- Red: Emergencies only
- Black: Not available at all

The instructions are as follows:

- Please indicate here your availabilities througout the project, using these colors.
- Please try to have maximum 12 days of red/black
- Always add a justification

Figure 15: Availabilities extract

G) Other Tools

1 Collaborative Tools

Daily communication is via **Discord**, which is used by everyone and is common in university projects. This is the preferred solution for team members, and has therefore been chosen. In an emergency, it is possible to communicate by telephone. We have created numerous channels, some general (`announcements`, `general`, `banter`) and others specific to each type of task (`web`, `access-vm`, `network-database`, etc.)

Finally, we use **Google Drive** for project management, while Git is used for code management (VI.B)).

2 Reports

Reports were created for each meeting: team meeting, individual meeting, or with the tutor. This allows us to keep track of each interaction and decision, and thus avoid forgetting them. The reports of team meetings are **freely accessible** by each team member, who can suggest modifications in the event of misinterpretation.

VII. Team Management

A) Common Times

1 Regular Meetings

Every Monday, we got together for regular work meetings. These were mainly dedicated to group meetings and decision-making. This was also the time for **meetings with the tutor**.

These meetings enabled all team members to **get an overview** of what each other had done during the week, and for everyone to get an overall view of the project. It was at this point that we decided on the **week's tasks**, and their distribution by team. After that, the **distribution of tasks within the team** was mostly done autonomously.

2 Working Times

After a few weeks' work, we realized that **4 hours** of shared time **was not enough**.
Indeed, many tasks are interdependent, and it is therefore necessary to work together on them. This is entirely possible at a distance, but requires a lot of interactions by message, and therefore a long delay before we get the necessary answers.
What's more, some team members prefer to work with others to motivate each other and move forward together.

As a result, we set up these **work sessions**, which we held practically every week (figure 16).
Some lasted just 2 hours, while others spanned the whole day.
This initiative was beneficial to the group's dynamics, and these times generally led to **significant progress**. Although we weren't always at full strength, everyone was able to participate **at least 2 times**.

Figure 16: Runestones Devs team after a common work time

3 Informal Times

At the end of these working sessions, there were usually **informal moments**, important for team cohesion. This was not usually planned at the start of the working time, but came about as a result of the proposal to motivate everyone to finish up for lunch or a night out.

Although the project is not mentioned too much during these informal times (which were not

included in the timesheets), they do **contribute to team cohesion** and the general motivation of the group.
In particular, on May 1st, which was a public holiday, we had two big work sessions, each lasting the whole day. These two days were interspersed with an informal time when the other group was also invited, so that we had enough **energy** to complete the two days.

B) Individual Follow-up

One of my priorities as project manager is to provide **individual follow-up** for each team member. To achieve this, we quickly set up several follow-up mechanisms.

The main form of follow-up is **weekly individual calls**. These usually last around 15 minutes, and are designed to provide detailed information on each team member's progress.
The idea came from the observation that during the weekly Monday meetings, it was not necessary for each team member to know the detailed progress of each task. Nevertheless, this is something I need to know as project manager, in order to **assist each team member** in more detail with the **organization of their tasks**, and above all to avoid blockages.
If I see that a person is staying too long on a task, it may be necessary to cut it up, assign several people to it, review its scope, or review its integration in the Gantt.

These calls took place almost every week, thanks to the establishment of a fixed time slot. This has enabled us to **detect bottlenecks earlier**. Several feedbacks show that they are also **appreciated by the team**.

Otherwise, individual follow-up takes place via **Discord**, through private messages or thematic channels. I also regularly check task progress via the backlog, timesheet and commits. In case of doubt, we often **communicate by message**, or by call if the conditions are right.

C) Team Organization

What is special about our team is that several people are multidisciplinary and therefore relevant to several subjects.

That's why we have **several development teams**, and why **each person** can be a **member of several development teams**. The workload per team is therefore not equitable.
Regarding the distribution of work: the project manager **assigns a task to a development team**, which then **distributes the task internally** according to :

1. the skills and interests of each member,
2. the number of tasks each member has completed in the other teams to which they belong.

Once the task has been completed by the member in charge, the other team members are responsible for validating the code before it goes into production.

The development teams are as follows:

- **Modelling.** Responsible for modelling assets (scenery, characters, objects, etc.) and any animations.
- **Game logic.** Responsible for implementing the game algorithms in the game engine and managing the interactions between elements (player, targets, obstacles, etc.), as well as

managing the man-machine interface, menus, etc. This team prepares the data to be sent to the network team.

- **Adaptation.** This team manages the adaptation of the game to the Web version and ensures that the game's functionality is preserved. It is a task force made up of members of the game logic team and the network team.
- **Network and server.** Is responsible for receiving and securing the data sent by the game logic team using appropriate protocols. It then validates this data, synchronises it and liaises with the DB team. It also manages the configuration of the server(s), their security, and their resilience to a surge in load.
- **Databases.** This team manages access to the database, securing the data and sending it to clients for display.

The allocation is the following:

- **Project management:** Maxime Zingraff
- **Modelling:** Ugo Berton *(lead)*, Ahmet Berk Ates, Solène Besancon, Anna Callet *(support)*
- **Game logic:** Solène Besancon *(lead)*, Anna Callet, Ahmet Berk Ates, Ugo Berton, Louise-Anne Corbé *(support)*, Hassan El Sahily *(support)*
- **Network and server:** Louise-Anne Corbé *(lead)*, Hassan El Sahily, Solène Besancon *(support)*, Maxime Zingraff *(adviser)*
- **Database:** Hassan El Sahily *(lead)*, Ahmet Berk Ates, Louise-Anne Corbé *(support)*, Maxime Zingraff *(adviser)*
- **Web adaptation:** Ahmet Berk Ates *(lead)*, Solène Besancon *(lead)*.

D) Role of the Project Manager

Our general view of the project manager's role is that of coordinator, planner and mediator. The project manager wrote a charter, which was then validated by the team members. Here are the main tasks that were set:

- **Functional description of tasks**: the project manager ensures that the project is broken down into precise functional tasks, and writes a description of what is expected. As the team members are the experts in their field, they take care of the technical description of the task as well as estimating its duration.
- **Evolution of task allocation**: the project manager manages the overall progress of the project, and ensures the overall anticipation of tasks so that the project is finished on time. Using the functional description of the tasks and their estimated time, he updates the Gantt chart and ensures that the milestones set are met. He then allocates the tasks to the development teams responsible, who share out the work according to their affinities and availability. He also manages the monitoring tools, such as Discord and GitLab. Once the task has been completed, it can be validated by the other members of the development team and then implemented.

- **Human relations**: in the event of interpersonal problems within the team, the project manager ensures that the working atmosphere is healthy and does his utmost to resolve conflicts. If this doesn't work, he reports the problem to the tutor. Team members undertake to report any human problems they encounter.

- **Keeping track of hours and availability**: team members undertake to fill in the time tracking sheet honestly and accurately. The project manager monitors the members' sheets and ensures that they are consistent with the justifications. In the event of an abnormal gap or work overload, he will discuss the matter with the person concerned to ensure that the distribution of work is appropriate, and if necessary re-evaluate the distribution of tasks. In the event of unavailability, team members must inform the project manager. Similarly, if the project manager is unavailable, he will give notice and the second in command will take over.

- **Organising team meetings**: the project manager is responsible for organising team meetings. Before the meeting, he sends out the agenda, or if there is no agenda, explains the purpose of the meeting (if the meeting is dedicated to a specific topic), and takes care of the moderation and note-taking. No later than one week after the meeting, minutes are sent out to ensure that what is said is consistent with what was said during the meeting. Meetings can be held on Mondays or at other times. In addition, development team meetings can be held if necessary.

- **Individual monitoring of team members**: every week, if possible, the project manager has a quick meeting with each member of the team to review their work. This gives a clearer picture of how the project is progressing, avoids unnecessary bottlenecks and enables the allocation of tasks to be updated.

- **Link with the tutor**: he is the only person who interacts directly with the tutor. He must therefore collect questions from the team and pass them on, as well as disseminating general information given by the tutor. He undertakes not to misrepresent what the team has to say, as they trust him to represent their interests.

- **The project manager's job is not to code**, but rather to help out if it's within his or her remit. This allows them to take a step back from the code, and therefore to unblock situations that may be caused by the choice of an unsuitable data structure or algorithm. This position also makes it possible to make the link with the solution developed by another team, and to anticipate blockages.

VIII. Handling Unexpected Events

A) Delays

1 On the Project

The biggest source of unforeseen events on this project was delays on various tasks.
Up until April, we were encountering **relatively few delays** on ongoing tasks. For some development teams, we were actually ahead of schedule.

However, this rapid progress in March created another invisible problem, which we only identified later: the **multiplication of branches** (V.B). Indeed, at the time of the planned presentation to the tutor, we were planning to merge our existing progress into a presentable game. However, it was at this point that we realized how much our branches had **diverged** from each other and from the main one.

I then set up an action plan, with a precise order of merge and deadlines to be met. Having never had this kind of experience, I underestimated certain merges, which **took longer than expected** due to conflicts.

For other merges, I knew what to expect, so they were **better anticipated**.
As a result, we have, for example, **reduced the divergence of our branches**, with branches that are open for less time and where the main is pulled frequently.
The exception is the matchmaking branch. Initially conceived as an experimental branch to understand the mechanics, it became the branch implementing matchmaking. As a result, the merge became very cumbersome, to the point where I realized that it would take dozens of hours to complete.

At this point, in consultation with Louise-Anne, who was in charge of development, I took the initiative of **stopping the merge** to focus on finishing the project. This proposal was approved by the group, and we moved on.
Although this was a complicated experience for the team, it's important to know when to stop when a task becomes too burdensome. It was not possible to restrict the scope, allocate more human resources or more time, so this was **the only lever available**.
Thanks to the buffer time provided by the Gantt (VI.D)), we were still able to finish the other functionalities on time, with a limited increase in pace.

In future, to limit these delays, it would be relevant to have **more points dedicated to the organization of branches** and discrepancies between functionalities.

2 On Timesheets

Some team members were for a while under the minimum curve of the 150h target (VI.E)). This was communicated to the tutor.
For the people concerned, this was a point that was discussed regularly. Systematically, **an action plan was agreed** with the person concerned, in order to set short- and medium-term **quantified targets** (e.g. 20 hours by next Monday).

These hour recovery plans have worked, as all team members are now **above the 150h mark**.
Some are closer to 200h: it's not a question of a poor distribution of tasks, but **a desire for**

these people to keep moving forward with the project, while others prefer to focus on other school projects, which is perfectly understandable.

B) Human Issues

Our team was fairly close-knit, and we encountered **very few human problems**.
The main problems were linked to our student status, and therefore to the fact that most of the work had to be carried out **asynchronously** and often remotely. This can easily lead to **conflicts**, due to **misunderstandings** of poorly drafted messages.

What's more, this organization can lead to stress, as there are fewer interactions and therefore **less visibility** on the project.
This is why we have set up work sessions (VII.A)**2**).
Informal times were necessary to **maintain a good atmosphere** and a sense of togetherness (VII.A)**3**).

On several occasions, certain team members had personal delays that were not communicated, and therefore not anticipated. Some could have been better anticipated, such as during **university vacations** or **exam periods**.

In future, it may be appropriate to adapt the initial Gantt and the requirements document to take account of these external calendar constraints.
Apart from that, we fortunately encountered very few human problems.

C) Organizing a Demonstration

The public demonstration was a way for us to **limit unforeseen events** and risks, by identifying potential new bugs so that they could be corrected.
The demo was therefore placed **as soon as the main functionalities had been completed**, so as to have time to implement the feedback.

It was a success, with the presence of computer science students but also participants from outside the faculty (figure 17), who gave us **interesting feedback** that was implemented, thus avoiding the need to incorporate these errors into the final game.

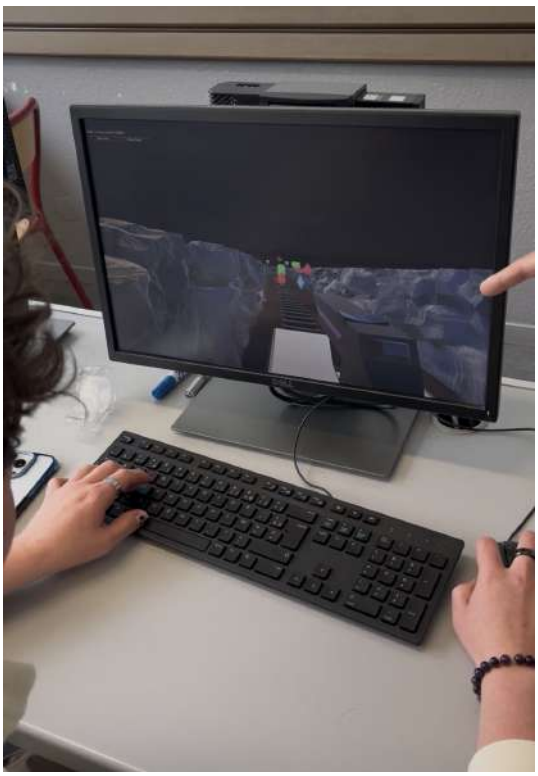

Figure 17: People playing during the demonstration

D) Interactions

1 With the Tutor

Interactions with the tutor enabled me to be **guided** on certain human or organizational issues. The Monday meetings were always busy (see tutor meeting reports), and helped **reassure the team** about the overall direction of the project. As I had little experience of this kind of project, these interactions were invaluable.

As project manager, I **felt free**: if there was a problem, with timing for example, Mr Golda simply suggested **different options**, and it was then up to me to choose the one that seemed most suitable, or to create a hybrid one.

2 With other Project Coordinators

I had several interactions with Antoine Gautheron, project coordinator of the 4B group. These interactions enabled us to **compare our practices**, and to **gain insights** into how to manage problems, particularly human ones. These interactions also enabled us to share best practices, and **mutually improve our project management methods**.

IX. Individual work allocation

By alphabetical order on the last name

A) Ahmet

Ahmet was quite **versatile**, and was able to touch on many subjects.

At the start of the project, he spent a lot of time **designing the wagon** and debugging it. He tried out a number of strategies to prevent the player from falling off the wagon, and in the end it worked. Ahmet also spent time on **modeling**, and in particular modeled the entire Level 1 scenery using the technique described in II.A). He also modeled the wagon.

He returned briefly to the wagon to add the **teleportation mechanism**, and finished the project on the web. In particular, he took care of the entire frontend, adapting the game to the **web format**.

Complicated bugs to explain delayed his work, such as errors in new WebGL compilations at the end of the project.

Between these major tasks, Ahmet was a **resource person for Unity**, and was often on hand to help with debugging, such as merge conflicts or reviews.

B) Ugo

Ugo's tasks were very versatile. His work began with the creation of various **test scenes** to learn certain elements of the game, including the creation of a **first player model** and its movements. He continued with the player, implementing **interactions with obstacles**.

After that, Ugo focused a lot **on the UI** in preparation for the two UI exams linked to the project. He contributed to the Figma and produced the first version of many screens (login, game launch, etc.), even if some had to be taken over by other people.

Finally, he took care of the **creation of the matchmaking lobby** (IV.C)), although unfortunately it remained in alpha version with the abandonment of matchmaking.

The end of the project was marked by the creation of the **user manual**, virtually on his own.

C) Solène

As **tech lead**, Solène was on all fronts. Right from the start, she took charge of the game's networking. Thanks to her knowledge of .NET, she also **led the research** on this subject.

The majority of her project was therefore dedicated to Mirror. She took charge of the game's first online multiplayer, which practically required **starting from scratch**, as the NetworkManager (IV.B)) had to be implemented.

As tech lead, she wrote the first version of the **Git policy** (appendix A.). Then, during the

merge build-up, she did a lot of merge reviews that prevented her from developing.

The end of the project was devoted to **resolving bugs** on the wagon. Players kept falling over despite the walls, so she had to revise the player and wagon prefabs. She also contributed to **modeling assets** such as the bat, managing the endgame, and helping other team members like Louise-Anne with matchmaking.

D) Anna

Anna focused **mainly on Unity** for the first half of the project, with various tasks carried out mostly on her own. For example, she worked on the design of gems and the in-game **leaderboard** (I.C)**3**). This enabled us to grasp an essential game mechanic: being **able to share an object** *(the leaderboard)* with other players, and ensuring that everyone could contribute to it *(increase their score)* according to their actions. After numerous authority problems, the leaderboard was implemented.

Anna also worked on **modeling several assets**, such as the player and the gems. She then animated the gems, added scripts (score, etc.) and managed the random gems generation algorithm.

Finally, at the end of the project, she worked on a number of **separate tasks**.
In particular, she took charge of **communication**, creating the visuals (see communication document), **adding music and sound effects**, and carrying out performance tests.

E) Louise-Anne

For some time, Louise-Anne has been **documenting Mirror**, the use of VMs, and the installation of Unity server, which has enabled her to be a resource person in these areas.

For the online multiplayer version of the game, she worked partly on her own: she created an **entire test scene**, which ran on a network. Then all we had to do was apply the mechanisms we'd learned to our game. A week was needed for **debugging**, but this method proved its worth.

This is what we wanted to reproduce for **matchmaking**: based on an existing philosophy (IV.C)), she created a functional test matchmaking environment, before implementing it in the game. This phase was extremely **time-consuming**, and would have taken even longer to finalize, unfortunately.

Meanwhile, Louise-Anne also **contributed to the UI**, correcting early versions of Ugo to make them consistent with the graphic charter.

F) Hassan

Hassan focused mainly on the **backend** of this project. Right from the start, he was in charge of creating the databse, and the .NET API. The API required a lot of work, in particular to **make communications secure**. The HTTPS connection required a certificate (IV.D)), and this took longer than expected.

Hassan managed **the database** almost autonomously, including the links between tables, the creation of triggers and procedures, and the **reception of API requests**.

He also worked on **authentication** and **account creation**, with several protections and security features (password criteria, non-redundancy of users, etc.)

Finally, he set up the **friend system**, with the possibility of sending requests, accepting them, etc. Hassan worked with Ugo to create the friends UI.
To finish off the project and make the "**reset password**" (I.C)1) button functional, he set up the API to send password reset e-mails with a code, which you simply enter in the game to have your password reset.

G) Maxime

As **project manager**, the tasks accomplished are detailed in the VII. section.

As well as managing the team and the project, he also helped with certain tasks, such as **resolving the certificates** (IV.D)), or the first **WebGL tests** with Ahmet.

Appendices

A. Git Usage Policy

Convention

Git Repository

A git must be organised otherwise it's illegible and we might do some ✨mistakes✨ Here are the repo convention we'll use for our project :

Definition of Done :

- Merge with main branch or other intermediate branch
- Code reviewed
- Acceptance criteria met
- Functionality documented if not trivial
- Interaction between pre-existing objects not affected
- Functional tests
- No impact on game multiplayer
- No major impact on server load
- Communication security guaranteed

Branch :

- Each branch is named after its associated task (e.g., interaction-player-wagon).
- The branch name is written in minuscule separated by `-` as the example above
- The task performed in a branch is UNIQUE and corresponds to its title (e.g., creation-player-model is only for creating the player model, not all models).
- Each branch has assigned developers, and only they are allowed to work on it.

Merge :

- When creating your merge request, you must validate options **squash commits** and **delete source branch**
- NEVER MERGE YOUR OWN BRANCH.
- A merge request (MR) must be approved (which means carefully reviewed, verified, and tested) by two people before being merged into main.

Commit :

- All commits must be in English and start with a verb (e.g., remove SendToAllAsync() func).
- Keep them simple, short, and explicit.
- If u cite a name in the code, e.g. function profil or variable name, be careful to e-x-a-c-t-l-y copy the case (so we can easly ctrl+F)
- Make atomic commits whenever possible (e.g., don't commit at the end of the day with "end of day commit"; instead, when finishing a class structure, commit "create class Obstacle", then add functions and commit "add func CollisionTrigger()", etc.).

Bonus how to use git :

- **Somebody pushed their branch before yours in the repo**. Exemple, for the player interaction in unity, all unity team member may work on the GameManager. If someone merge their work before you in the main, you need to update *your branch* with the *current main* which change compared to when you created your branch. Only then you'll be able to merge your own work in the main. Don't panic here is what u should do :

```
git checkout my_branch
git fetch origin

git checkout main
git pull origin main

git checkout my_branch
git merge main --no-edit
```

If they are conflicts, you must solve them in your branch.

Code

It is just as essential for maintaining consistency in our project, so here are the coding conventions we will follow:

- Every class and public function must have a short description (as well as in and out parameters description for the public functions) IN ENGLISH explaining its purpose, formatted as follows:

```
/// <summary>
/// Description...
/// </summary>
```

- Variables should follow camelCase, with a special rule for private variables: they should be in _camelCase (preceded by an underscore `_`). Constant variable must be in MAJ_SNAKE_CASE.

"A picture is worth a thousand words"—you must exactly replicate the formatting in the example below (placement of parentheses and braces in functions, spacing in conditional loops, etc.). But most importantly, make your code readable (spaced out, consistent, and explicit).

```
using System.Text;
using Database.Repositories;
using Microsoft.Extensions.Logging;
using Microsoft.Extensions.Options;
using Network.Configs;
using Util;

namespace Network.WsServer
{
    /// <summary>
    /// Class that represent and manage a WebSocket server.
    /// </summary>
    public class WsServer : Server, IWsServer
    {
        private const int MAX_WAIT_FOR_CONTEXT_PROCESSOR_MS = 2000;
        protected override int MAX_WAIT_FOR_REQUEST_PROCESSOR_MS => 1000;

        private readonly ConcurrentDictionary<string, WebSocket> _connections;

        public WsServer(IOptions<WsServerSettings> settings, ITeamRepository teamRepository)
        {
            _settings = settings.Value;
            _connections = new ConcurrentDictionary<string, WebSocket>();
            _teamRepository = teamRepository;

        }

        /// <summary>
        /// Get uri of the WebSocket server.
        /// </summary>
```

```
        /// <returns>The uri of the server.</returns>
        public string GetUri() => Uri;

        /// <summary>
        /// Start the server. This method is idempotent.
        /// </summary>
        public void Start(IWsServer.ProcessWsRequestAsync processWsRequestAction,
            IWsServer.ConnectionStateChangeAsync connectionStateChangeAsyncCallback,
            int expectedNbOfPlayerConnections)
        {
            _expectedNbOfPlayerConnections = expectedNbOfPlayerConnections;
            base.Start();

            _processWsRequestAsyncAction = processWsRequestAction;
            _connectionStateChangeAsyncCallback = connectionStateChangeAsyncCallback;
        }

        /// <summary>
        /// Send a message to every connected WebSocket asynchronously.
        /// </summary>
        /// <param name="message">Message to send.</param>
        /// <param name="msgType">Type of the message.</param>
        /// <returns>A task to await the send if necessary.</returns>
        public async Task SendToAllAsync(string message, WebSocketMessageType msgType = WebSocketMessageType.Text)
        {
            byte[] msgBytes = Encoding.UTF8.GetBytes(message);
            ArraySegment<byte> buffer = new ArraySegment<byte>(msgBytes);

            List<Task> sendTasks = new List<Task>();
            sendTasks.Capacity = _connections.Count;

            foreach (KeyValuePair<string, WebSocket> entry in _connections)
            {
                WebSocket ws = entry.Value;
                sendTasks.Add(SendToOne(ws, buffer, msgType));
            }

            await Task.WhenAll(sendTasks);
        }
}
```

Bonus convention

If you have any question during the project or feel unconfident for any reason, seek help toward anybody. As stupid as a question might sound to you, we'd rather answer stupid questions than having to repare weird problems. So I must insist, feel free to ask anything to anybody if it can help you carry on the project. We must learn and have fun while doing this project ^^.

B. Gantt Chart

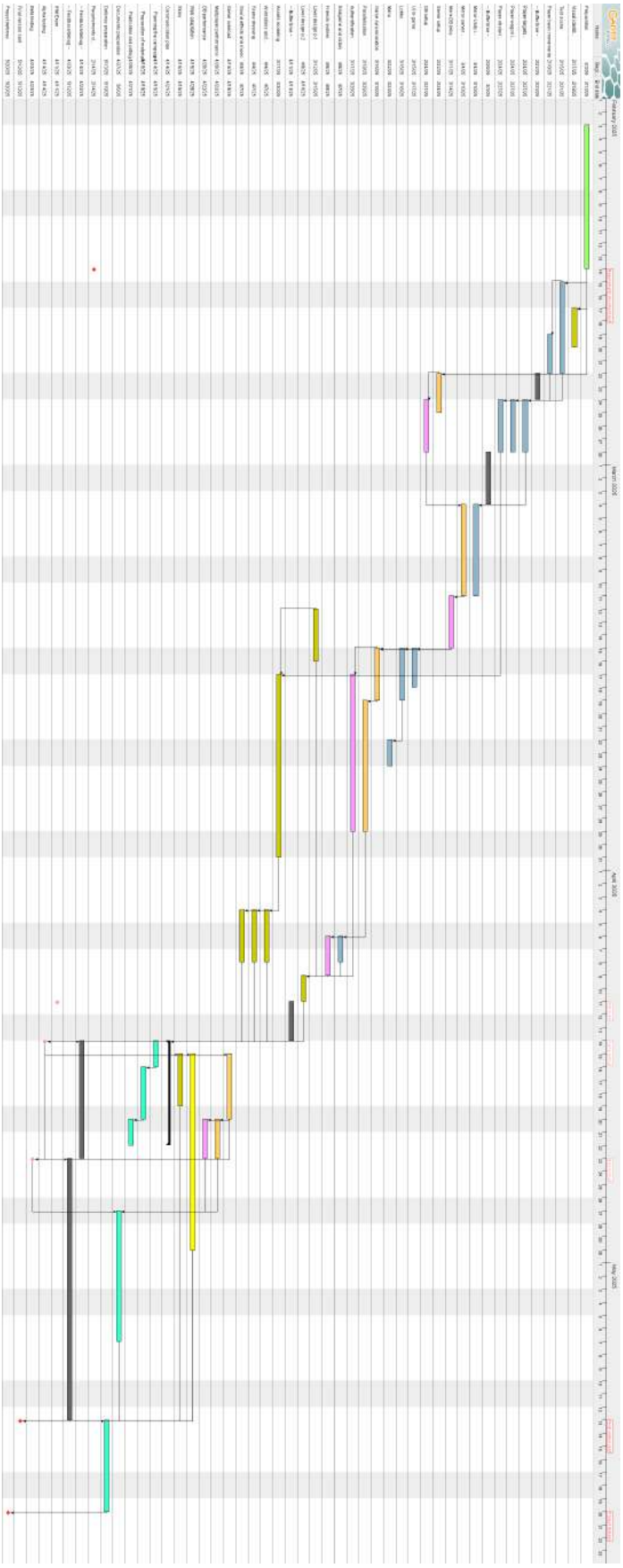

Figure 18: Gantt chart