Third Project Report

Version: 1

Date: 18/03/2026

Members : Damien Hasenfratz, Ruben Desbonnet-Ramdenee, Lilou Tardif

Game : Control+Sea

Summary

Introduction

- Group organisation
- Initial ideas
- Actual game
- Reasoning for changes

Project Chronology

Evolution

- Player Mechanisms
- Networking and Multiplayer Improvements
- UI, Graphics(concept art) and Implementation of Graphics
- Website

Conclusion

- What have we learned?
- What we could have done better and possible ameliorations?

Introduction

Group organisation

Name	Role
Damien Hasenfratz	Enemy AI, Networking, Project, UI
Ruben Desbonnet-Ramdenee	Core, Gameplay mechanics, Sound effects
Lilou Tardif	Assets, Sound effects, Storyline, UI

Initial idea

A crew of pirates is searching for treasure. The only problem is that it is on a remote island, they therefore have to travel the seas in order to reach it. They start their journey on a small boat which will be upgraded while they sail. In order to upgrade their boat they will have to win some resources. There are two ways to achieve this, either by finding an island where a treasure chest may have been hidden or by winning a battle against other pirates. Indeed the crew will encounter other pirates while they sail, if they choose to fight, the crew will have to win to gain a treasure chest, or else their boat and characters will damaged. The treasure chests can contain resources to improve the boat such as wood or cotton or some other elements such as coins or rubies; rhum, and sugar cane in order to heal the wounded characters in the battle. The boat could also be damaged because of disaster events such as a storm. In the end the crew will eventually arrive to the coveted island on their magnificent ship.

Actual Game

Our game is still a pirate theme based game. But instead of sailing and encountering islands or boats, the game starts directly with the battle where the enemies are on board. The characters can change their weapons during the battle which is interesting in order to have different ranges of attacks. Moreover

they are equipped with shields which reduce the damage dealt by enemy attacks. A battle is composed of different waves of attacks, when all of these waves are won the crew wins a treasure. This game is therefore more of an arcadey battle rather than rogue-like compared to our initial ideas.

Reasoning for changes

We had several issues to take care of which led us to have to lower the expectations for our game. First of all, we lost a member of the crew, which inevitably caused us to have to rethink our game; what we could do as only a trio, leading to lost time and major game changes. Second of all, as it is our first year in EPITA, we had to accommodate to the lifestyle that it gives, notably all the work that comes with it. Our top priority was our studies and we wanted to first focus on TPs and exams rather than the game, even though we had been working on it since November and regularly throughout the rest of the year. That's why after the second "soutenance" we decided to take into account our teacher's advice, to make a simpler but playable and enjoyable game. Even though our actual game is far from our initial idea, it is a good base in order to continue our game as we envisioned in the future.

Project chronology

▼ **November**

Multiple team meetings helped us lay out everyone's ideas into something more coherent. Our main ideas were:

- *pirates*
- ships
- sea
- treasure

▼ **December → January**

Prototyping physics and movement. Phase where we started to learn the workings of Ursina and Python in a game-oriented context.

▼ **January → February**

Primitive multiplayer and particle effects.

Weapons

▼ **February → February (late)**

Weapon / damage syncing

Interpolation

Sprites and texturing

Map models

▼ **February (late) → March (mid)**

Health bars, handling deaths

Basic UI elements (joining, death)

AI enemies

Weapon syncing

▼ **March (mid) → Deadline**

Lobby system

Weapon switching

Shields

Matches and waves

Music

More sprite textures

Player mechanisms (Ruben)

Physics and movement

Panda3D and Ursina has built-in solutions for physics objects. However Ursina’s version was poorly optimised for movement, and directly using Panda3D was deemed too complicated for the scope of our project. As a result, it was decided to make our own implementation of a rough simulation of movement physics.

They are all contained within the Player class which handles input, movement, and collisions for the local player. Remote players similarly have their own instance of the local player which dictates where they can go, therefore a complete synchronization of the map is expected within all running clients.

Players are able to: (QWERTY)

- Jump (space)
- Move left / right (a / d)
- Slam (Ctrl)
- Super jump (Ctrl, hit ground, space immediately)

Movement is a core part of our game and is a major fun factor.

Combat

Weaponry recap

Players have 3 categories of weaponry at their disposal as well as a defensive shield. Each category of weapons have their own variants with different cooldowns, damage, etc..

Weapons being a large part of our game, they were the logical thing to add next. So far, there are 3 types of weapons: Bows, Daggers, and Guns.

- Bow → *shoots physics-based projectiles with an initial velocity, they are destroyed on hit with the world or a player*
- Dagger → *stabs a given distance*
- Guns → *hitscans with a max distance, or cut short by an obstacle*

A combat item permanently points to the mouse, rotating around the player, allowing for mobile, high-intensity combat, and easy-to-learn / difficult-to-master shooting. Shooting is achieved through a left click, although each weapon variant has its own cooldown duration for balancing reasons.

Each player always has a shield available in-game; deployable by pressing ‘2’ or holding right click for quick-draw. The shield has a permeability value; meaning it does not absorb the full damage value from a projectile, only a set significant percentage.

Player-to-player damage is always on for more chaotic fun!

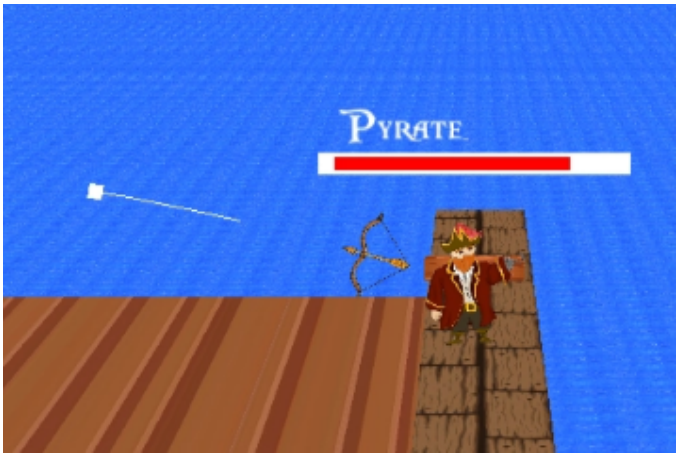
Weapon classes

A base *Swinging* class handles all mouse-following related maths.

Pointing towards the cursor was not simple. In Panda3D, the cursor is positioned in the viewport space, so has a position from (-1,-1) to (1,1) with (0,0) as the middle of the viewport. However we want to get the position of the mouse in the 3D world space, but the mouse can have an infinite number of positions in said world space. But using our camera, we can narrow down the possible positions to an infinite line in the world space based on our camera's perspective. Finally, with a plane at z=0 (the weapons z-axis position), we can find the point where the cursor's infinite line and plane intersect, giving us a workable coordinate. Afterwards, simply converting the computed point into its cartesian form can give us an angle to put the weapon at with a fixed radius r.

Placeholder weapon texture pointing towards the player’s cursor

Weapons stay fixed to players, rotating around them, giving a fluid, satisfying, yet challenging combat experience. This mechanic was chosen to encourage players to move around to dodge projectiles, whilst still being able to attack enemies.



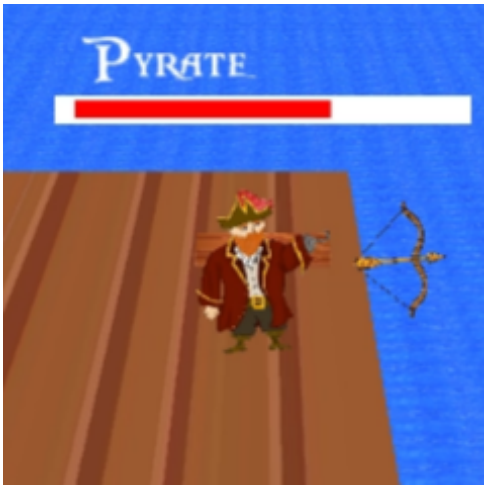
A Pyrate shooting a projectile

Bows produce projectiles which act as an independent entity, having their own self-contained physics (velocity, gravity, air resistance), and executing damage logic on hits. For visual synchronisation purposes, remote player's projectiles are *ghosts* of the original projectile to a local player; they execute trigger no external procedures.

A Pyrate hitting a Totalitarian Octarian with a sword

Swords were included to encourage more risky tactics where players can go have to go up close to enemies, but deal much more damage.

A Pyrate deploying its shield



A Pyrate with its shield sheethed

Shields were implemented as a ‘holdable’ just like weapons, meaning each player can only have the shield *or* the weapon out at the same time. We chose to implement this using a modular “hand” approach where each player has a “hand” with a certain number of slots for holdables. Keybinds “sheeth” and “unsheeth” holdables accordingly. In the current state of our game these slots

are the weapon and shield slots, but it would be easy to add a third holdable slot such as consumables, hence this approach.

Weapon selection wheel deployed

When the weapon selection wheel is in use, the current weapon is hidden for clarity. We found that not doing so would cause the current weapon to keep shooting and moving when we try to select a new weapon.

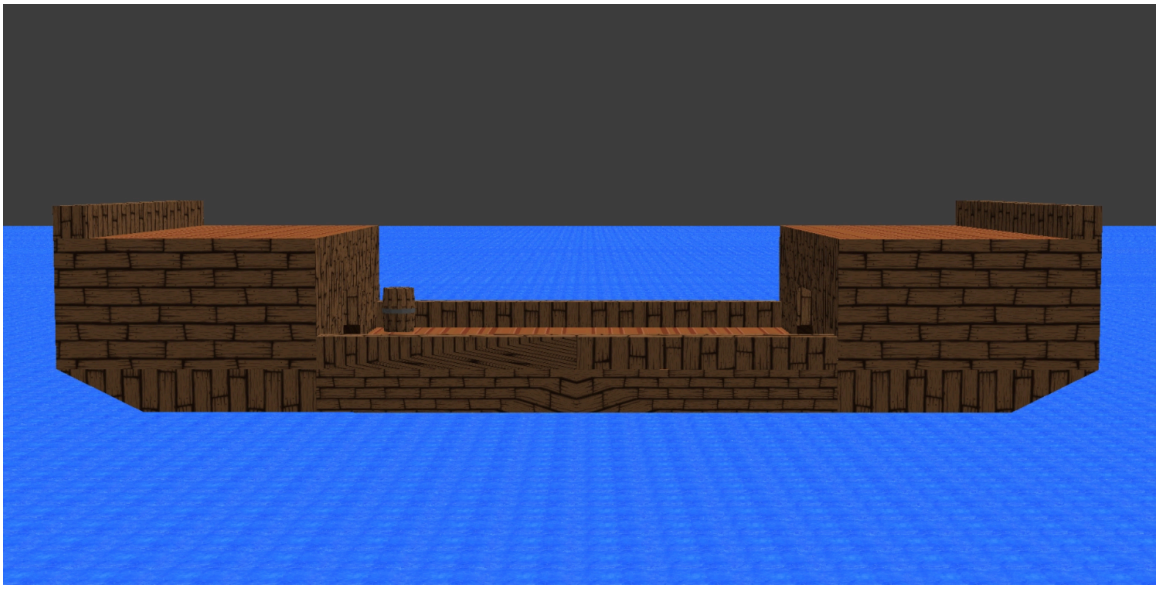
Changing weapon only changes the object in the weapon slot.

Data-driven weapon / shield definitions

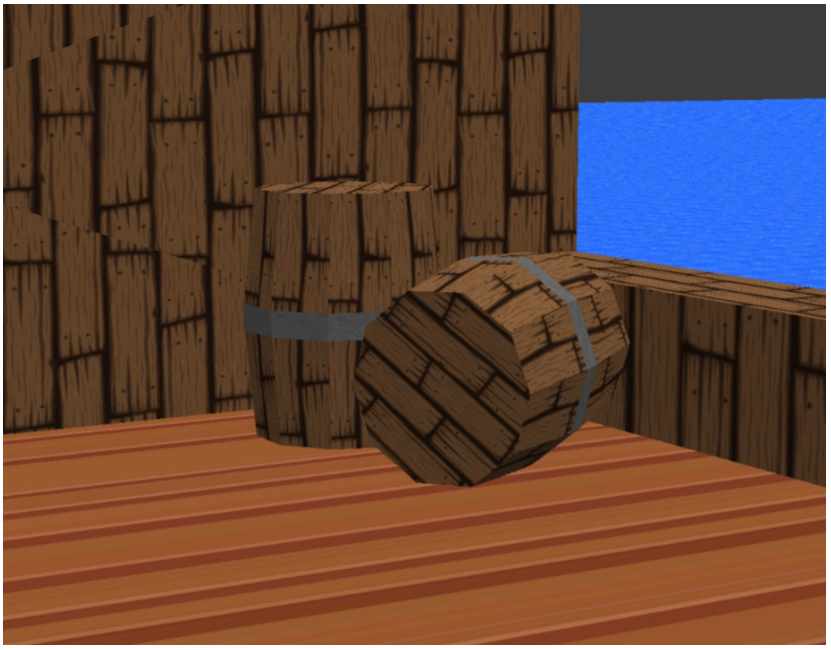
It was deemed most appropriate to use JSON to define each weapon variant's attributes and generic model info for each weapon class. This approach gave the most modular code and made it very easy to implement multiplayer weapon switching syncing as only the weapon class had to be synced, the rest of the data was already present on each client.

To prevent slowness due to disk access, the JSON files are loaded as monolithic Python objects (dicts).

Models



Custom ship_1 model



Barrel models

Some custom models were made for the map.

They follow a low-poly, repeated texture format as so to keep a consistent ‘vibe’ for the game; using hyper-realistic textures would have not matched the style of our sprites.

They communicate the overall theme rather well, making up a decent fighting arena.

Of course with more time and contributors, we could have prettied up the ship more and made variations.

Preloading textures on game launch instead of on a needed-basis has helped significantly reduce lag spikes.

Networking and Multiplayer Improvements (Damien)

Networking Recap

The multiplayer architecture is based on a client-server model implemented using the Ursina Engine networking and RPC system.

In this architecture:

- The **server** acts as the authoritative source of truth
- The **clients** are responsible for rendering, local input handling, and sending gameplay information to the server

This separation is essential to maintain a consistent multiplayer environment. Critical gameplay data such as match state, enemy health, wave progression, and combat events are validated and synchronized through the server instead of relying on local client states.

Communication uses **TCP** combined with **Remote Procedure Calls (RPCs)**.

RPCs greatly simplify multiplayer programming because remote functions can be called almost like local functions. Ursina automatically handles:

- Serialization of arguments
- Data transmission over the network
- Deserialization on the receiving side
- Execution of the remote function

This abstraction allowed the networking layer to remain relatively clean while still supporting a large amount of synchronized gameplay logic.

Network Ticks

To improve synchronization consistency, the networking system uses a fixed tick rate on both the client and server sides.

Without fixed ticks, network updates would depend on the rendering frame rate, which can vary significantly between machines. This would result in:

- Irregular packet sending
- Different update rates between players
- Less stable multiplayer synchronization

To avoid this, networking operations are executed at fixed intervals independently from rendering FPS.

At each network tick:

- Clients send movement and gameplay updates
- The server processes events and updates authoritative state
- Synchronization snapshots are broadcast to connected clients

This ensures more predictable and deterministic multiplayer behavior.

Interpolation

Because synchronization updates are only received at each network tick, directly applying received positions would create visible stuttering and abrupt movement.

To solve this, interpolation was implemented for remote entities.

For every synchronized player or AI entity, the client stores:

- The previous received state
- The latest received state

During rendering, positions are interpolated between these two states depending on the progression between ticks.

This creates smoother movement and significantly improves the visual quality of multiplayer gameplay.

Match Flow and Lobby System

Lobby Phase

One of the major improvements during this iteration was the introduction of a proper multiplayer lobby system.

Previously, players could directly enter gameplay immediately after connecting to the server. While functional, this approach created several synchronization issues:

- Players spawning before all entities were registered
- UI inconsistencies between clients
- Players entering the match before everyone was ready
- Match state race conditions during startup

To solve this, a dedicated **LOBBY** phase was introduced.

The lobby now acts as a synchronization and preparation state before gameplay begins.

During this phase, the server manages:

- Connected players
- Player registration
- Ready status
- Match start validation

Each player can join the session and wait until all participants are synchronized correctly before gameplay starts.

The server continuously broadcasts lobby information to all clients, including:

- Number of connected players
- Ready players
- Match status

This guarantees that every client shares the same understanding of the current multiplayer state.

Match Lifecycle

The overall multiplayer flow was reorganized around explicit gameplay states.

The game now transitions through several synchronized phases:

- **LOBBY** → Waiting and preparation phase
- **IN_GAME** → Active gameplay and wave progression
- **MATCH_LOST** → End-of-game transition before returning to lobby

These transitions are fully synchronized through dedicated RPC events such as:

- match_started
- wave_started
- match_lost

When one of these events is received, clients update:

- Entity visibility
- UI/HUD state
- Input handling
- Gameplay activation/deactivation

This solved previous issues where some clients could remain in outdated states after transitions.

For example:

- Some players previously kept gameplay UI visible after losing
- Some entities remained active after returning to the lobby
- Certain transitions could occur at slightly different times on different machines

The explicit lifecycle system made multiplayer flow significantly more deterministic and stable.

Wave System

Data-Driven Wave Configuration

A complete wave progression system was implemented to structure gameplay.

Instead of hardcoding enemy spawning values directly inside scripts, wave parameters are written in a dedicated JSON configuration file (waves.json).

Each wave can define:

- Enemy count
- Enemy health
- Enemy damage
- Enemy position

Balancing difficulty only requires editing configuration files rather than modifying gameplay code.

Server-Authoritative Wave Management

Wave progression is entirely controlled by the server.

The server is responsible for:

- Determining when a wave starts
- Computing the number of enemies to spawn
- Applying wave-specific difficulty scaling
- Broadcasting wave information to clients

Clients only receive wave events and update local visuals/UI accordingly.

This prevents clients from diverging regarding the current progression state.

Benefits of the Wave System

This architecture provides several important advantages:

- Easier balancing during presentations and testing
- Consistent progression across all clients
- Cleaner separation between gameplay logic and balancing data
- Better scalability for future content additions

AI Enemies and Synchronization

Hybrid AI Ownership Architecture

One of the largest networking improvements was the design of AI synchronization.

During the previous presentation, enemy AI was only executed locally on the client side without any real multiplayer synchronization. This approach was sufficient for demonstrating gameplay mechanics, but it was not compatible with a proper online multiplayer environment because each client would simulate enemies independently.

For this final iteration, the AI system was integrated into the online multiplayer architecture.

Initially, the objective was to create a fully server-authoritative AI system. This approach appeared to be the most logical solution because enemies are shared between all connected players. In this model:

- The server would generate all enemies
- The server would execute all AI behavior
- Clients would only receive synchronized snapshots

While this approach is theoretically cleaner and more secure, it quickly revealed important limitations.

The main issue is that the server does not have direct knowledge of the full physical world state on each client. AI behavior in our game heavily depends on local environment interactions such as:

- Detecting walls
- Jumping over obstacles
- Reacting to terrain geometry
- Collision-based movement

For example, an enemy may need to jump when encountering a wall detected through local boxcasts or raycasts. Reproducing all these physics interactions entirely server-side would require a much more complex world simulation architecture.

As a result, a hybrid ownership solution was implemented.

The system now uses a hybrid ownership model:

- The server remains authoritative for AI identity and critical attributes
- One client temporarily owns movement simulation for an AI entity
- Other clients instantiate synchronized “ghost” entities

When an enemy is spawned:

1. The server creates AI metadata
2. The server assigns an owner client
3. The owner client creates a real local AIEnemy
4. Other clients create synchronized ghost representations
5. Periodic updates are distributed through update_ai

In practice:

- The owner client executes AI behavior locally
- The server validates and distributes important gameplay state
- Other clients only reproduce synchronized movement and state updates

This architecture significantly reduces server-side simulation complexity while still maintaining multiplayer consistency.

It also allows AI behavior to remain reactive to the local environment, especially for movement and obstacle handling, without requiring the server to simulate the entire physical world in detail.

Server-Authoritative AI Health

Enemy health is now fully authoritative on the server side.

The process works as follows:

1. A client detects a combat interaction
2. A damage event is generated locally
3. The event is sent to the server
4. The server validates and applies the damage

5. Updated health values are replicated back to clients

This prevents situations where enemies appear dead for some players but alive for others.

AI Combat Improvements

Enemy combat behavior was also expanded.

AI enemies can now:

- Use bow-based attacks
- Spawn projectiles
- Respect cooldown systems
- Dynamically switch between combat states

The AI combat system now shares more logic with the player combat system, creating more coherent gameplay interactions.

This improved both gameplay quality and maintainability.

HUD and User Interface Improvements

HUD Design

The HUD was designed to provide clearer and more useful gameplay information during multiplayer sessions.

The HUD displays:

- Current wave number
- Remaining enemy count
- Alive players / total players
- Team average HP
- Enemy average HP

The objective was to provide a compact but information-rich interface suitable for cooperative gameplay.

Dynamic HUD Updates

HUD elements are now updated dynamically based on gameplay and networking events.

The interface refreshes automatically when:

- Players join or disconnect
- AI enemies spawn or die
- Damage is applied
- Waves change
- Match states transition

This prevents stale information from remaining visible and improves overall gameplay clarity.

UI, Graphics and Implementation of Graphics (Lilou)

Choice of player’s sprite

The game now features an option to choose the player’s character. It has three different options(as the crew is composed of three pirates), which means that some players can or will have the same character when playing but as the camera implemented follows our own character it won’t be problematic for the game.

The three different characters are in fact buttons that:

- are white if not selected
- gold and bigger when selected

When selecting a character, it clicks a button which will then give the right name to the variable taking care of the sprite, which will be used later on in the game. As a default setting the first character is automatically selected (before it can be changed by the player).

Weapons’ graphics implementation

In our game, the enemies and the players have the option of three different weapons that can be selected and changed throughout the game thanks to a weapon wheel. These weapons are objects that contains their size, color, texture, etc... In Ursina in order to implement an image as the object, the texture has to be set as the weapons’ drawing.

Randomizing Enemy’s Sprite

In Ursina, as it is very close from python, there is an option to get a random value. But this random value can only be between 0 and 1. And our game has five different sprites for the enemy. Therefore a little math needed to be used, a variable contains the random value:

- if this value is greater than 0.49 then it is divided by two
- otherwise it is multiplied by 10
- as it is a float number it is converted as an integer

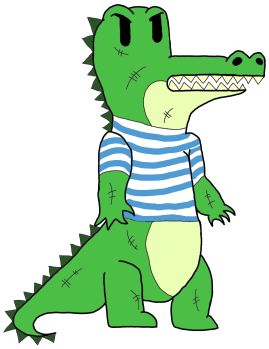
After these operations the value is now between 0 and 4. Between this range the value can therefore be used as an index for the table containing the different sprites which will be used in the texture just like the characters and the weapons.

Graphics

I first started with the captain, when drawing him i haven’t drawn in quite some time and i did everything directly on the computer. This was not a good idea as it took me a lot of time and a lot of energy. That’s why i decided to draw the characters on paper first then to trace them on the computer. This method made everything simpler and quicker especially when drawing on the computer.

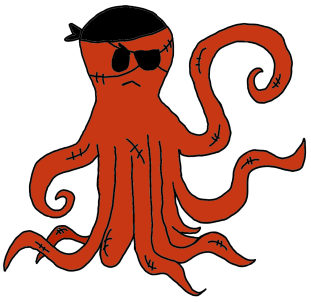
Characters' sprites

Enemies' sprites

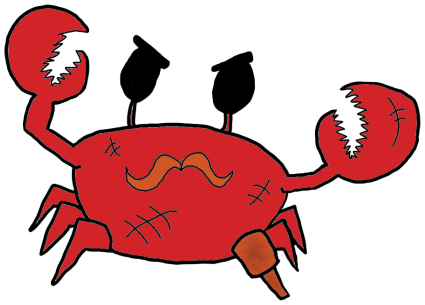


Cranky Crocodile

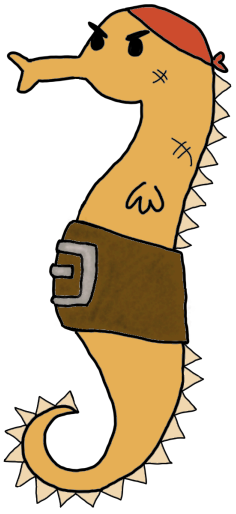
Sly Shark



Totalitarian Octarian



Crab "casse-tête"



No-trust Hippocampus

Website

To present our game more professionally and centralize all project resources, we developed a dedicated website for *Control+Sea*.

The website acts as both a presentation platform and a documentation hub for the project. It was designed using HTML and TailwindCSS in order to create a modern, responsive and visually coherent interface matching the pirate atmosphere of the game.

The site contains several sections presenting different aspects of the project:

- Introduction to the game
- Gameplay features
- Screenshots
- Development team presentation
- Download section
- Project reports and documentation
- Gameplay trailer

The visual design uses the same color palette as the game itself, mainly based around:

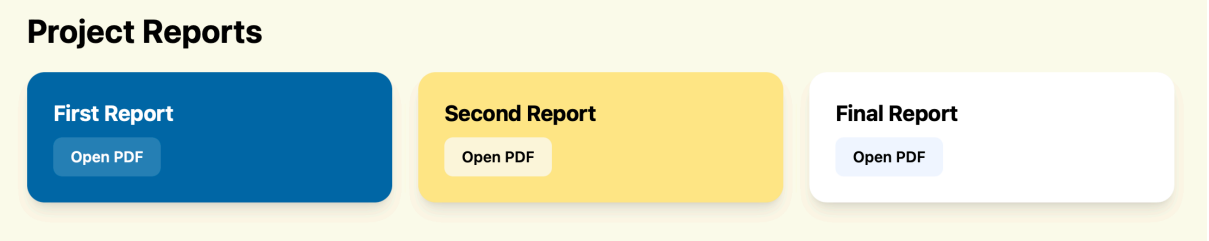
- dark blue
- amber/gold
- white and sand tones

This helped reinforce the pirate and sea-inspired aesthetic while maintaining good readability.

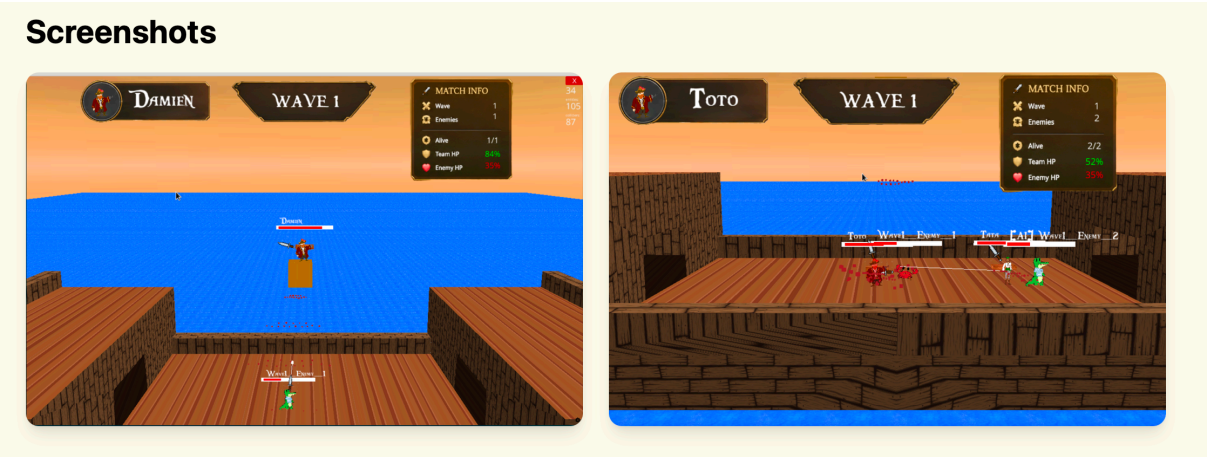
New Features

Compared to the first versions of the website, several important improvements were added.

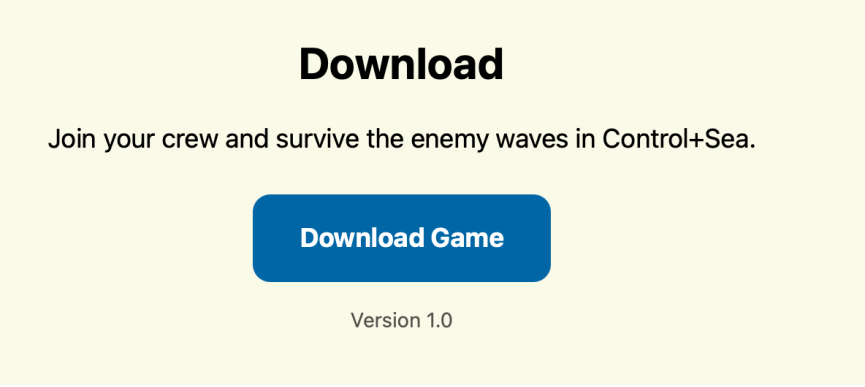
A dedicated **Project Reports** section was implemented to centralize all documentation related to the project. Users can directly open the different PDF reports from the website through interactive cards.



A **Screenshots** section was also added in order to visually present gameplay scenes from both singleplayer and multiplayer sessions. These screenshots allow visitors to quickly understand the visual style and atmosphere of the game before downloading it.



In addition, the website now includes a direct **Download** section allowing users to easily access and install the latest version of the game.



Trailer

The trailer is meant to introduce and describe briefly the game, and make a potential player want to play to our game. This trailer therefore displays the main features of our game in a pirate theme based video. It also features the music that can be found in our game, when a battle occurs, in order to hint on our actual game.

Conclusion

To conclude, even though our game does not look exactly like what we had envisioned at the start of the project, we are still genuinely proud of what we have collectively built and the time and effort we have all invested to make a fully playable game. The final result stands as a testament to our perseverance and teamwork, and reflects the many iterations, adjustments, and creative decisions we made along the way. What began as an ambitious idea gradually took shape into something we can truly call our own.

What have we learned?

During this process, we have all acquired many valuable skills that will undoubtedly prove useful in our future careers. As it is a large-scale project with a timeline covering nearly the entirety of the school year (roughly seven months) it served as a meaningful preview of the kinds of ambitious, long-term projects we will encounter later, whether during our studies or in our professional lives as engineers. Managing a project of this scale taught us the importance of structuring our work efficiently and maintaining momentum over an extended period, even when motivation fluctuated or unforeseen obstacles arose.

This experience helped us greatly in terms of project management: learning how to organize a team, prioritize tasks according to their urgency and dependencies, set realistic deadlines, and provide regular updates to keep every member aligned and informed. We therefore significantly developed our

skills in communication, coordination, and organization. Knowing how to delegate responsibilities and trust one another was just as important as the technical work itself, and this collaborative dynamic is something we will carry forward into every future project.

Prioritizing and breaking down the different tasks into manageable components was one of our biggest challenges. It required us to maintain a clear and shared vision of the game as a whole: its mechanics, its lore, its atmosphere, and its specific genre, etc... while also being able to zoom in on individual features without losing sight of the bigger picture. Balancing creative ambition with technical feasibility was a constant exercise in judgment and adaptability.

On the technical side, it also took us some time to get comfortable with the game engine and the various game mechanics involved, especially since this was our first experience creating a game entirely from scratch, and particularly within a Python-based engine. This learning curve ultimately pushed us to deepen our understanding of object-oriented programming, applying concepts such as inheritance, and encapsulation in a concrete and meaningful context. Beyond programming paradigms, we also explored the practical application of physics and mathematics (from collision detection and movement vectors to randomizing sprites) bridging the gap between theoretical knowledge and real-world implementation.

What we could have done better and possible improvements?

Looking back at the project as a whole, we can confidently say that we gave the best we could with the resources, and time we had available, doing everything in our power to deliver a game that is both playable and enjoyable. Every member contributed meaningfully, and the final product reflects the genuine effort and dedication we poured into it throughout the year.

Of course, with the benefit of hindsight, there are a number of things we would have approached differently. Had we had more time and perhaps one or two additional group members, we would have been in a much stronger position to implement a wider range of gameplay mechanics, enrich the overall

experience, and polish the finer details that had to be set aside due to time and resource constraints. A larger team would also have allowed us to distribute the workload more evenly, reducing the pressure on individual members and enabling more parallel progress across different areas of the game.

One of the most significant challenges we faced was the unexpected departure of one of our group members. Unfortunately, we were not informed of this directly by the person themselves, but rather learned about it through the administration after we had to ask. This lack of communication made the transition even more difficult to manage, as we were not given the opportunity to prepare or reorganize proactively. In retrospect, as soon as we became aware that a member might be leaving, we should have acted more swiftly and decisively: reassessing our goals, restructuring our priorities, and shifting our direction earlier rather than continuing on the original course for longer than was sustainable.

From that point forward, we made the deliberate choice to concentrate our efforts on the battle aspect of the game, which helped us refocus and maintain a clear sense of direction. However, following the departure, we should have also reorganized our internal task distribution more thoroughly. Specifically, sharing core sections of the code among all remaining members would have allowed us to work on critical systems simultaneously, accelerating development and freeing up time to invest in additional features and refinements. Working in a more parallel and collaborative manner on the foundational code, rather than having isolated responsibilities, would have made the team more resilient and efficient.

Another area where we feel the game fell short of our original vision is animation. For instance, characters do not visually animate when moving across the boat, which is something that would have added a great deal of life and immersion to the game. This was ultimately a consequence of prioritization; with so many core mechanics to implement and limited time, animations were pushed down the list and never made it back to the top. Additionally, our graphics manager faced personal academic challenges that made it difficult for her to dedicate as much time to the game as she would have liked, which inevitably impacted the visual and animated aspects of the project.

Nevertheless, and despite all of these obstacles, the game remains an enjoyable experience. The pirate-themed assets bring a strong and cohesive visual identity to the project, giving it a distinctive atmosphere that we are proud of. The theme comes through clearly, and players can engage with the core mechanics in a way that feels coherent and fun. What we have built may not be perfect, but it is ours, shaped by collaboration, resilience, and a shared passion for seeing it through to the end. We are therefore proud of our game.

Possible improvements

As our actual game is more of a combat game and our initial idea was for our game to be more rogue like then we would maybe try to continue our game to make it look like our initial idea.

As we already have the battles we would only have to implement:

- The lore with dialogues and UI elements
- The exploration of the sea
- encounters with islands or boats

And if we want to work on it further more we could implement the upgrades functions such as:

- Upgrading weapons
- Upgrading and decorating the ship

  The most difficult parts would concern mainly the camera views and the upgrading of the boat . But as the battles are already done thanks to our game’s first version; the main idea of our game is already playable.