Project Report (Technical)

Version: 1

Date: 15/01/2026

Introduction

Description of our game

Our game is pirate theme game. A crew of pirates are 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 have be 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.

Elements included

The important elements that are required for our game are the use of some 3D elements. We haven’t choose to make a full 3D game as it would have limited the functionement of our game, we therefore chose to make a 2.5D game which blends 2D and 3D in a natural way. In order to do that we will produce our own designs and graphics. One of the requirements was to make an AI that would control a part of the game, which is what we included in the pirate enemies our crew will encounter throughout it’s journey. This AI will therefore control an other pirate crew where the level will be adapted according to the proximity of our crew to the final treasure(this means that the level of difficulty will increase as the crew continues to sail). In order to fight them back, this will require some teamwork and collaboration made possible thanks to the multiplayer mode developed by the network and the core. We want our game to

look as professional as it can be which mean that our game will have a basic user interface with menu and starting button, in-game information and player feedback.

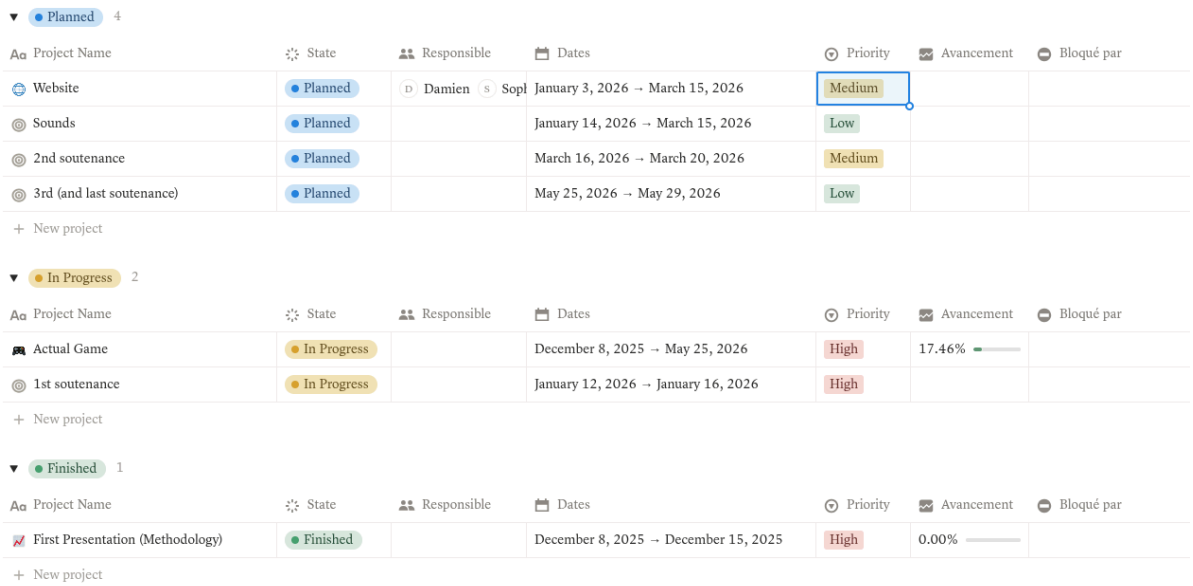
Group organisation

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

Organisation of our project

Tasks and deadlines are all written on Notion as to have a clear vision of our project and the pending tasks.

We have all the important deadlines to have a vision on our progression regarding our goals and their dates especially regarding the soutenances.



▼ Planned 4

Project Name	State	Responsible	Dates	Priority	Avancement	Bloqué par
Website	Planned	Damien Sopl	January 3, 2026 → March 15, 2026	Medium		
Sounds	Planned		January 14, 2026 → March 15, 2026	Low		
2nd soutenance	Planned		March 16, 2026 → March 20, 2026	Medium		
3rd (and last soutenance)	Planned		May 25, 2026 → May 29, 2026	Low		

+ New project

▼ In Progress 2

Project Name	State	Responsible	Dates	Priority	Avancement	Bloqué par
Actual Game	In Progress		December 8, 2025 → May 25, 2026	High	17.46%	
1st soutenance	In Progress		January 12, 2026 → January 16, 2026	High		

+ New project

▼ Finished 1

Project Name	State	Responsible	Dates	Priority	Avancement	Bloqué par
First Presentation (Methodology)	Finished		December 8, 2025 → December 15, 2025	High	0.00%	

+ New project

We clearly divided our game into different tasks managed by different members of our team. For each task, we can see who is responsible for what and who is the main substitute, even if all the team participate in each of the tasks. Moreover, all the tasks are subdivided into smaller tasks which help to see our progression and encourages us to continue as we can see which tasks are done and which ones are left to do which can be indicate thanks to the state menu. Each task have a priority level which helps us to organise ourselves, and prioritise our work regarding the different deadlines and progression. This means that the priorities can be changed and reviewed based on what we thinks fits best.

Task Name	State	Responsible	Substitute	Priorité
AI Enemies	Todo	Damien	Ruben	Low
▼ Map & Level Design	Todo	Sophie Mendez	Ruben	Medium
▶ Game lobby	Todo			Medium
Sounds	Todo			Low
+ New sub-item				
UI / Menu / HUD	Todo	lilou tardif	Ruben	Low
▼ Gameplay Mechanics	Todo	Ruben	lilou tardif	Medium
▶ gameplay	Todo			
+ New sub-item				
▶ Storyline	Todo	Sophie Mendez	lilou tardif	Medium
▼ Assets & Animation	In Progress	lilou tardif	Sophie Mendez	Medium
▶ Characters	In Progress			
▶ Random	In Progress			
▶ boat	Todo			
▶ islands	Todo			
+ New sub-item				
▼ Core	In Progress	Ruben	Damien	High
▶ Multiplayer/Network	In Progress	Damien	Ruben	Medium
▶ Combat	Todo	Ruben	Sophie Mendez	High
▼ Core Core	In Progress	Ruben		High
▶ Cool movement	Done	Ruben		High
▶ Effects	In Progress	Ruben		Low
+ New sub-item				

Notion is a great tool for collaborative work as we can all contribut together simultaneously on the document. It’s also the platform where we share all our common documents such as the reports or our presentations for the soutenances(mainly).

Technical choices

▼ Game engine

We chose Ursina (a Panda3D wrapper / engine) for its simplification of 3D, making it easy for us to develop in 2.5D. As it is just a wrapper, we can also interact directly with Panda3D if we so choose, for advanced features such as calculating matrices for shaders.

It also simplifies the networking side of things by giving documented, pre-built functions that do all the low-level work.

▼ Rogue-like combat

Creates a repeatable game loop that can easily be expanded once the framework of the game is completed, giving us leeway in case of delays.

Individual Work

Map and level design (Sophie)

As the map, level, and design manager, I was in charge of defining the game’s storyline and overall narrative structure. I developed this storyline to support the core gameplay mechanics while remaining simple enough to be easily understood and implemented within the project’s technical and time constraints.

The pirate theme seemed ideal to a storyline focused on exploration, combat, progression and treasure hunting. For this reason, I designed a storyline centered around a crew of pirates sailing across the sea, encountering enemy ships and islands in order to collect treasures and upgrade their ship and weapons. I also deliberately kept the storyline lightweight and non-linear. That means that instead of a complex scripted narrative, the story emerges through gameplay events, such as battles and discoveries. In this way we can avoid

long dialogues or cutscenes, which would have been more time-consuming to design and implement, and could also bore or disturb the players while t,hey are communicating with each other.

In addition, the storyline was designed to promote teamwork by placing all players on a single ship with shared objectives, instead of making a competitive game. For this, the storyline is imagined in a way that the game will encourage communication between players. Indeed, they will constantly need to choose between fighting or avoiding enemies, as well as deciding how to use the collected treasures.

Another advantage with our pirate themed game in terms of development and project constraints, is that it can be expanded or simplified depending on our advancement and the time frame we have to work with.

Core (Ruben)

Some prerequisite Ursina-specific concepts:

- `self.animate(str:name, float:target_value, float:duration, function:setattr_function, function:curve)`
  Calling an Entity's *animate* function creates an object with the variable name of the parameter 'name' concatenated with '_animator'. This object will be dynamically changing the value of *self.{name}* every frame, reaching *target_value* at the end of the duration. Additionally it does so non-linearly if given a *curve* argument.
  The generated object has the `pause()` , `resume()` , and `kill()` methods.

Player class

Each active Player controls their character with their game's local instance of `N3Player()` . Input is handled by the Player object directly and updates the player position based on states, and velocity (Vec3). This class handles no multiplayer aspect, as all position updates are done, as will be explained in the next section, by RPCs.

Local collision handling allows for very good modularity in multiplayer and reduced server load.

Some movement related-features are:

1. Realistic gravity when jumping

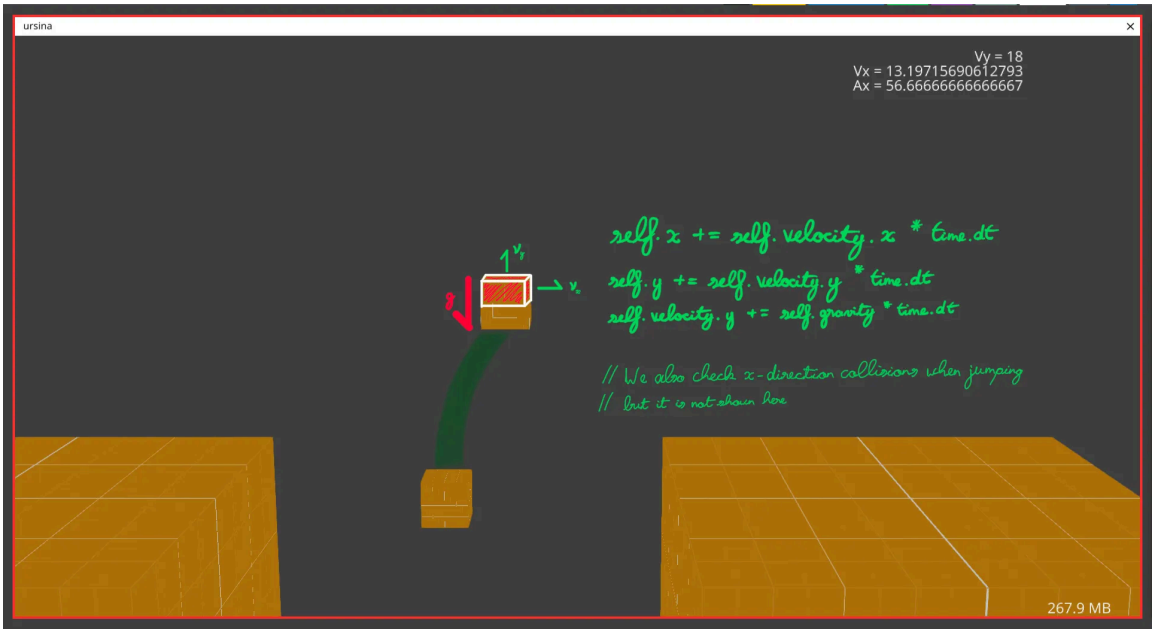
2. Non-linear x movement (friction and non-instantaneous velocity)
3. Slam (Instant terminal velocity reached, no allowed x-movement when slamming, boosted jump on land, no way to cancel slam until landed)

All the local states and values dictate how our player behaves:

```
# -- Movement --
self.walk_speed = 17
self.maxWalk_time = .3
self.stop_mult = 1.7
self.max_jumps = 1
self.slam_mult = 1.5
self.slam_timeFrame = .15
# -- -------- --
```

A full jump has two main states:

▼ **Jumping state**

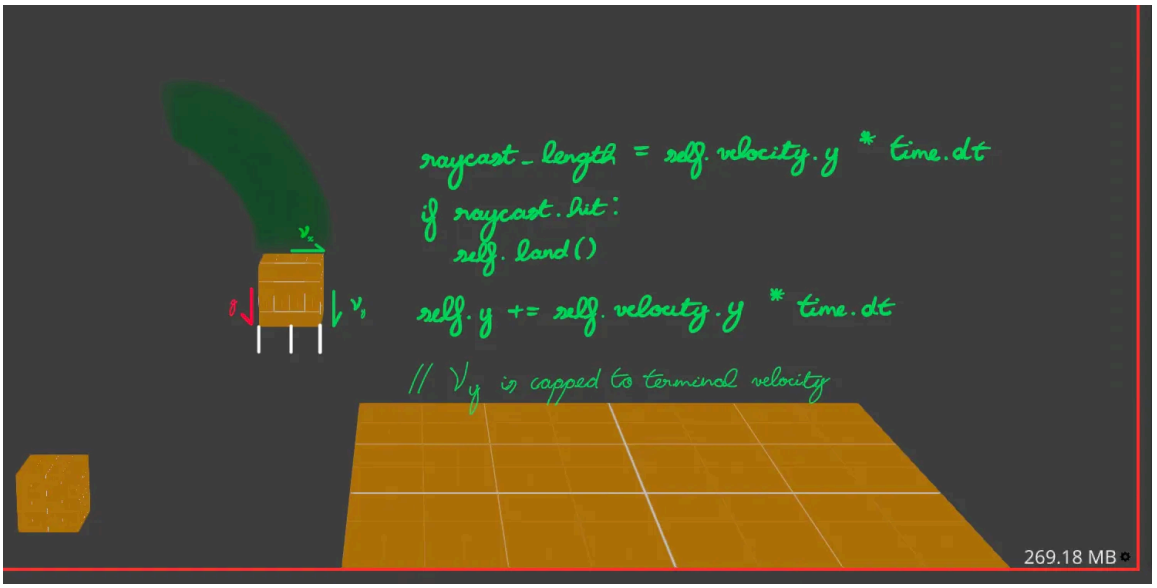


self.jumping

In this state, a fixed positive y-velocity is applied on our Player Entity, and the position 'animated' with a set duration, and target y-position.

An upwards facing boxcast stops our jumping state on a hit, killing the current instance of y_animator and starting the falling sequence.

▼ **Falling states**

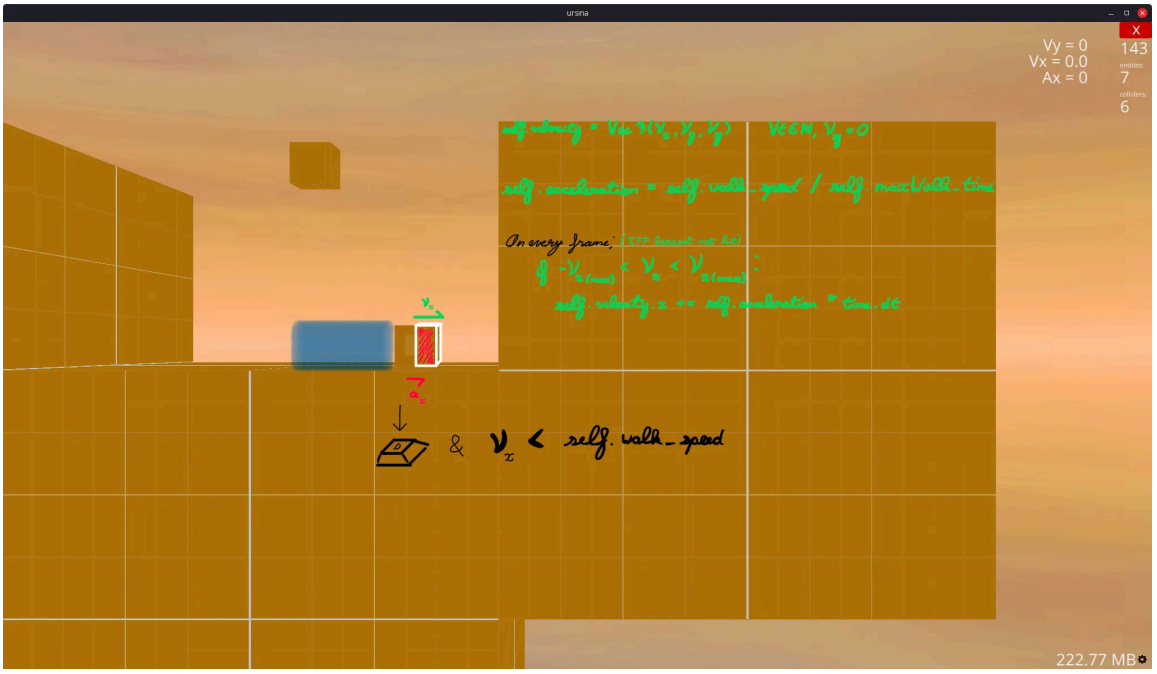


self.falling

The falling state is (similarly to jumping), animated using the `animate()` method until our *target_y* (terminal velocity is reached). Once terminal velocity is reached, if we have not landed yet, an additional linear_fall flag is set, making y-position updates be handled by manually adding the velocity each frame.
Collision detection is done with bottom-facing raycasts of dynamic length, initiating the `land()` on a hit of any of the rays.

x-movement has the following states:

▼ **Walking state**



self.walking

A walking state is triggered by a standalone ‘D’ or ‘A’ input, and removed when none of these keys are pressed. We set a fix acceleration based on our constants, triggered by the state, which increases the velocity every frame until the max is reached.

The case where both inputs are pressed at the same time simply sets `self.walking=False`

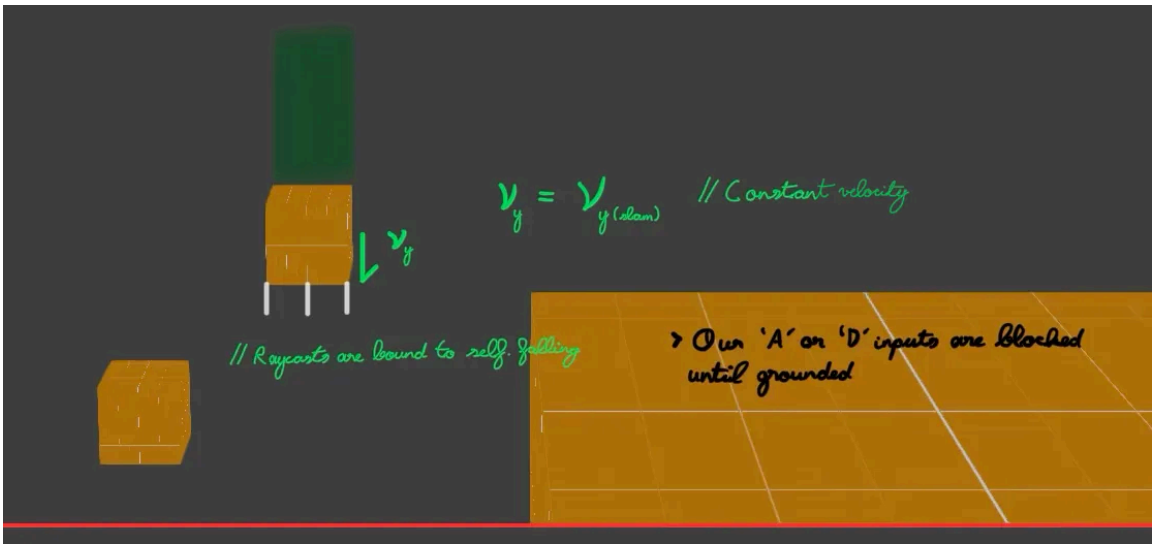
▼ Stopping state

self.stopping

A stopping state occurs when $V_x \; ! = 0$ and not keys are being pressed. In which case an opposing acceleration calculated from a given friction constant slows our velocity down until 0.

The slam mechanic:

▼ **Slamming state**



Particles class

This class generates particles around a point in a given pattern, using Physics-based entities. For now only a circular pattern generation is implemented.

```
class Particles():
    def __init__(self,
        origin=Vec3(0,0,0), # Relative to parent
        parent=None, # Required
        model="cube",
        scale=Vec3(.1,.1,.1),
        particle_velocity=Vec3(5,3,5),
        amount=10,
        variation=.02,
        texture=None,
        colour=color.brown,
        pattern="circular",
        auto_destroy=True,
        destroy_delay=2.5
        ):
...
```

A particle object is required to have a parent entity (in all cases for now; the player object) and will calculate and position generated particles relative to the parent.

Circular patterns are generated in the following way with *n* amount of particles requested:

$$\Delta\theta = \frac{2\pi}{n}$$

In the polar space, our delta between each of the particles' positions.

$$r = \sqrt{x^2 + y^2}$$

Where x and y are half the width and height of our parent entity accordingly.

$$(x, y, z) = (r\cos\theta\ ,\ 0\ ,\ r\sin\theta)$$

Computation of the relative position of each particle

Therefore on for each particle where $nth >= 2$:

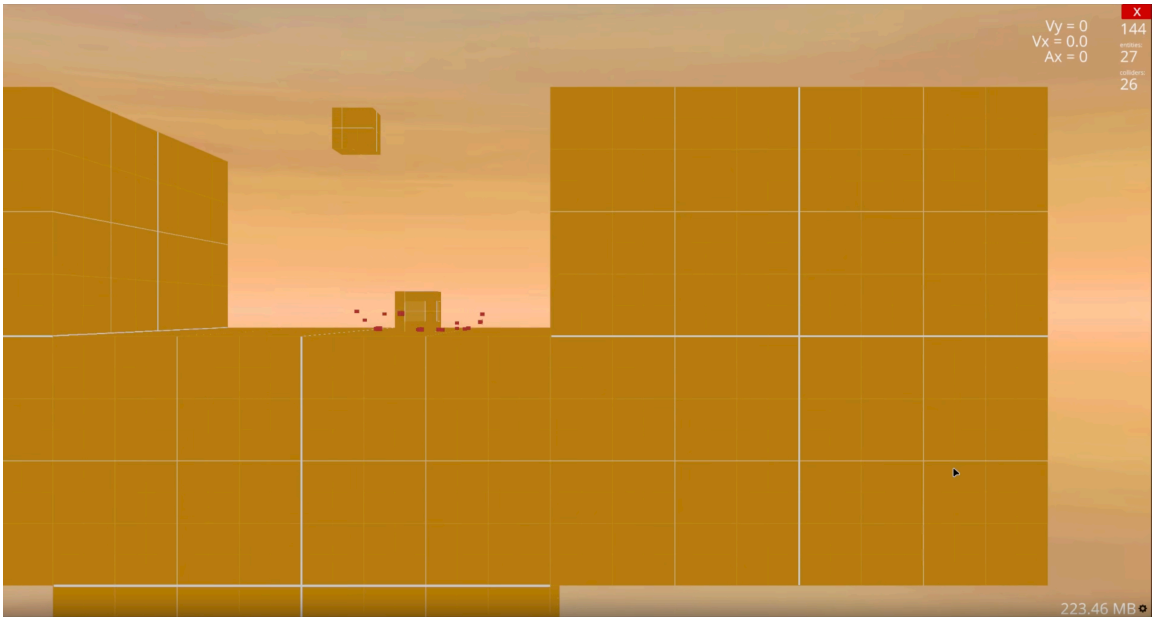
$$\theta \mathrel{+}= \Delta\theta$$
$$(v_x, v_y, v_z) = \alpha(r\cos\theta\ ,\ 0\ ,\ r\sin\theta) \times rand(0,1)$$
$$(x, y, z) = (r\cos\theta\ ,\ 0\ ,\ r\sin\theta)$$

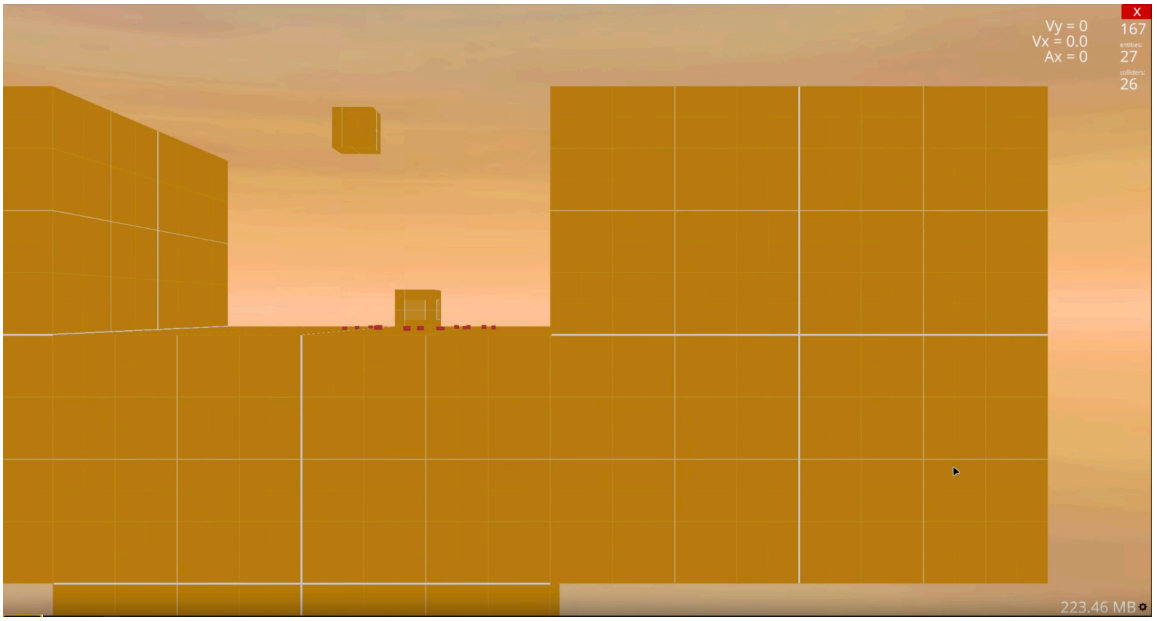
Where alpha is the particle velocity parameter (Vec3).

So overall: calculate delta theta, find radius, then convert into cartesian for every particle, applying the delta theta on a new particle.

Thanks to Ursina and Panda3D, all physics calculations are abstracted from us, meaning we only have to apply their velocity once, then they do the rest by themselves. After `delay`, they are destroyed using Ursina's prebuilt `destroy()` method for entities.

▼ **Circular particles**





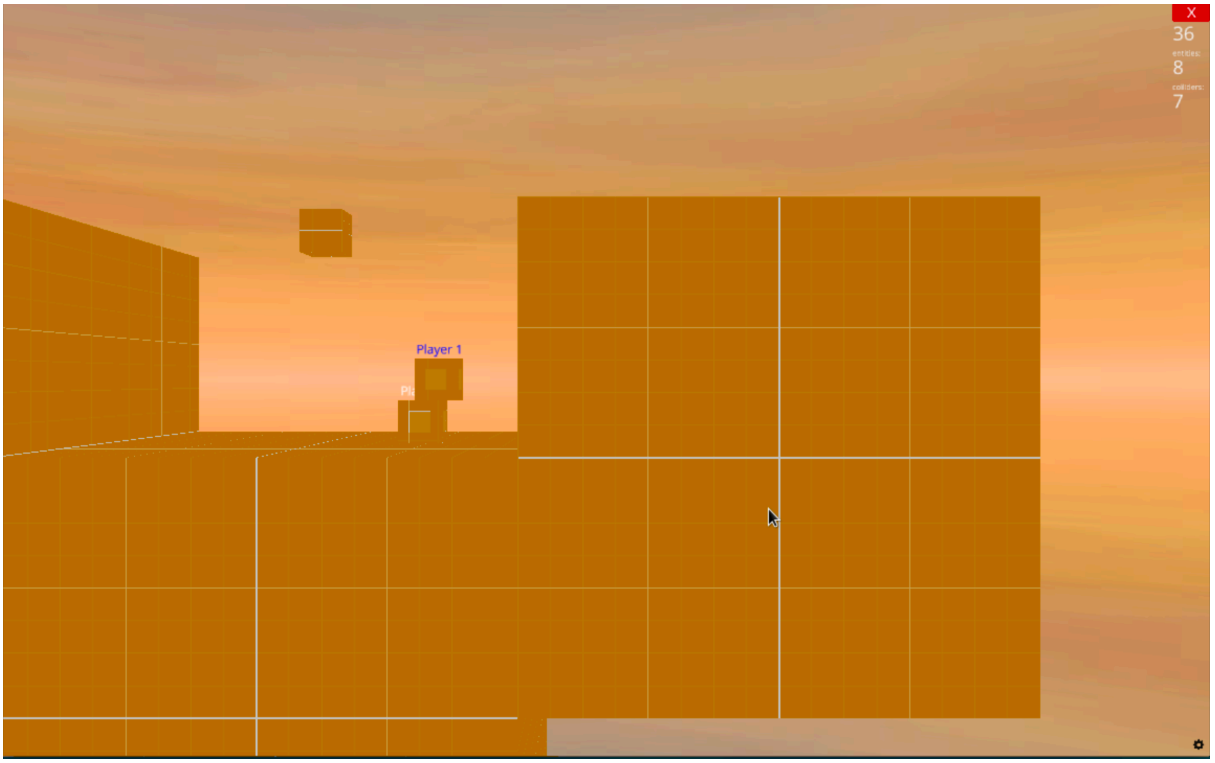
Effect called on land() when Vy ≥ Vy_terminal

Network (Damien)

Overview of the Networking System

My main contribution for the project is the implementation of the multiplayer networking system. The objective was to create a functional client-server architecture allowing multiple players to connect to that same session and interact in real time.

Two players connected to the same multiplayer session and rendered in the same game world. Player positions are synchronized through the client–server network system.



Two players interacting in the same multiplayer environment with collision handling enabled, demonstrating physical interaction between player entities.

Networking Model and Architecture

This project uses a client-server model, where :

- The server is authoritative and maintains the global game state.
- The clients only sends player actions and receive updates from the server.

Communication Protocol and Abstraction

The network layer is implemented using the **ursina.networking** module. Ursina's networking system is based on the **TCP protocol** rather than UDP.

This choice significantly simplifies development, as it removes the need to manually handle packet loss, reliability, and ordering. Given the cooperative nature of the game and the small number of players, TCP is a suitable choice for this project.

To further simplify networking, Ursina provides an abstraction based on **Remote Procedure Calls (RPCs).**

Remote Procedure Calls (RPC)

RPCs allow functions to be executed remotely on an other machine over the network.

It makes calling a remote function as simple as calling a local function because Ursina internally handles:

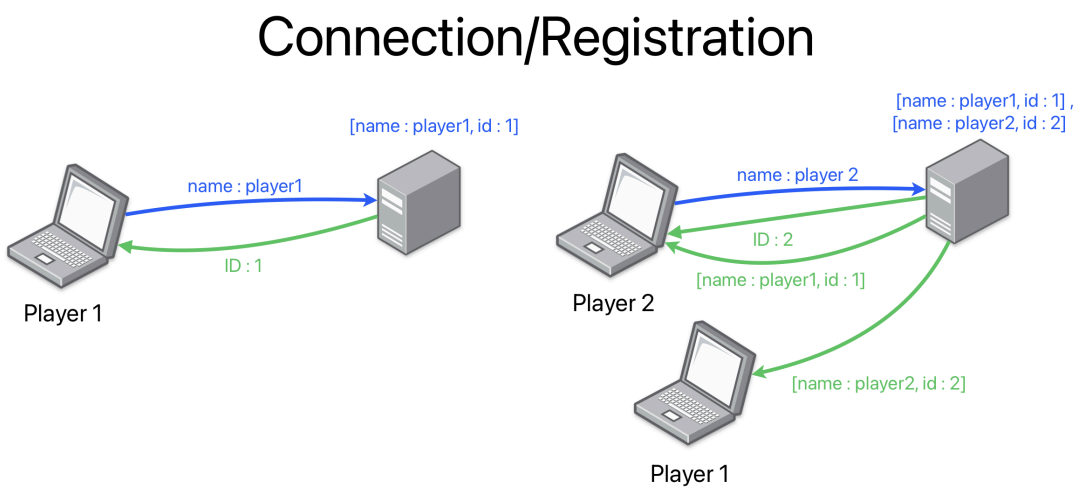
- Serialization of function arguments
- Transmission over TCP
- Deserialization on the receiving side
- Execution of the target function

Each RCP function explicitly defines its parameter types using type hints, which are required for proper serialization and deserialization.

This system allow a clear separation between Network logic and Gameplay logic and avoids direct socket manipulation.

Implemented Network Features

▼ Player Connection And Registration



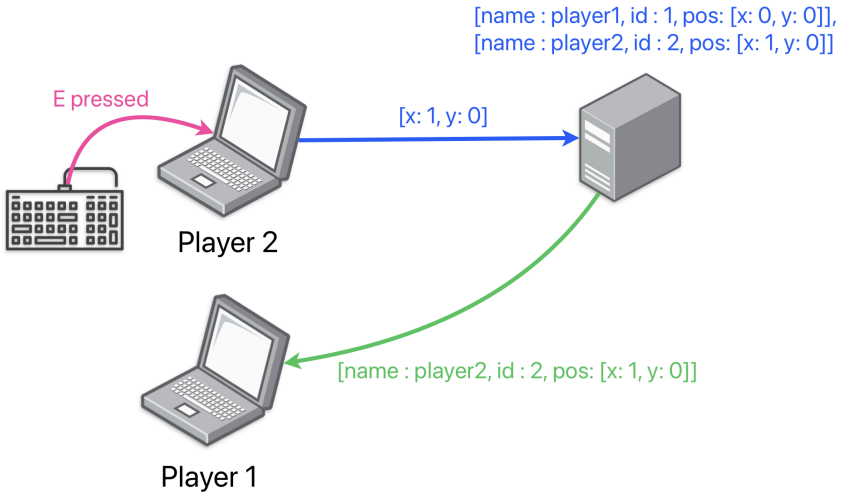
When a player connects to the server, it sends a registration request containing the player name. Then the server :

- Assign a unique player ID
- Stores the player state
- Notifies all connected clients of the new player
- Send the list of existing players to the newly connected client.

This insures that all clients maintain a consistent list of all players.

▼ Player State Handling

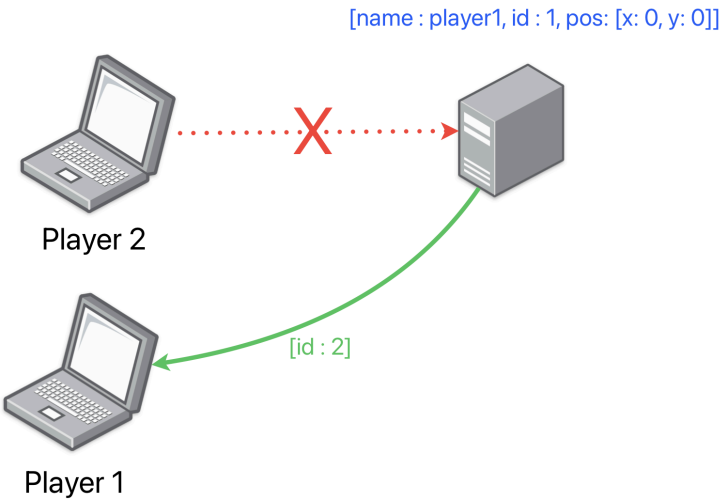
Player State Handling



Each client periodically sends its position to the server. The server updates the authoritative state and broadcast this information to other connected clients.

▼ **Disconnection Handling**

Disconnection Handling



Disconnection Handling is split between the server and the clients.

When a client disconnect, the server detects the loss of the connection, identifies the player associated and notifies all remaining clients.

On the client side, each client receives a notification containing the ID of the disconnected player. This client removes the corresponding entity from the scene as well as all associated data. This prevent “ghost players” from remaining visible after a disconnection.

Limitations of the Current Implementation

The current implementation has several known limitations:

- No client reconnection or session recovery is implemented
- The synchronization system does not include prediction or interpolation
- Security or cheating prevention is not addressed at this stage.

Assets and animations (Lilou)

Overview

My main contribution for our game is working on the graphics of it, such as the characters and the different elements needed for our game to look like a real pirate game. I am also in charge of animating the different assets created to make our game more interactive and professional-like.

Moodboard

As our game is a pirate theme based game, all our aesthetic and graphics will be pirate themed. I started to create a moodboard to have the overall vision of the atmosphere we wanted our game to have. We therefore chose to have a more childish and colorful pirate theme with the use of bright colors such as yellow or blue than the pirates mainstream which are perceived more violent and a lot less friendly with colors such as deep red, brown or black. In the moodboard I have also integrated some pictures of the style of graphics we want to have as well as some ideas for our own boat. As our target is everyone, starting from young children to adults, that’s why our game is bright and bubbly rather than realistic and deep(in terms of atmosphere and aesthetic).

Characters

In complement to that, we discussed as a team whether I should focus on pixel art or vector art for the overall drawings. It has been decided that the vector art would be easier to implement as well as being more fun and more in touch with what we wanted to have as an atmosphere. After that I decided to make our characters in a cartoonish style as realism or semi-realism would have been time consuming, harder to implement and animate as well as being less fun and childish than the cartoon characters.

Here is a sample of our first character. The proportions are deliberately not right and he is not handsome as a mainstream pirate would look like which therefore resembles a cartoon-style pirate.

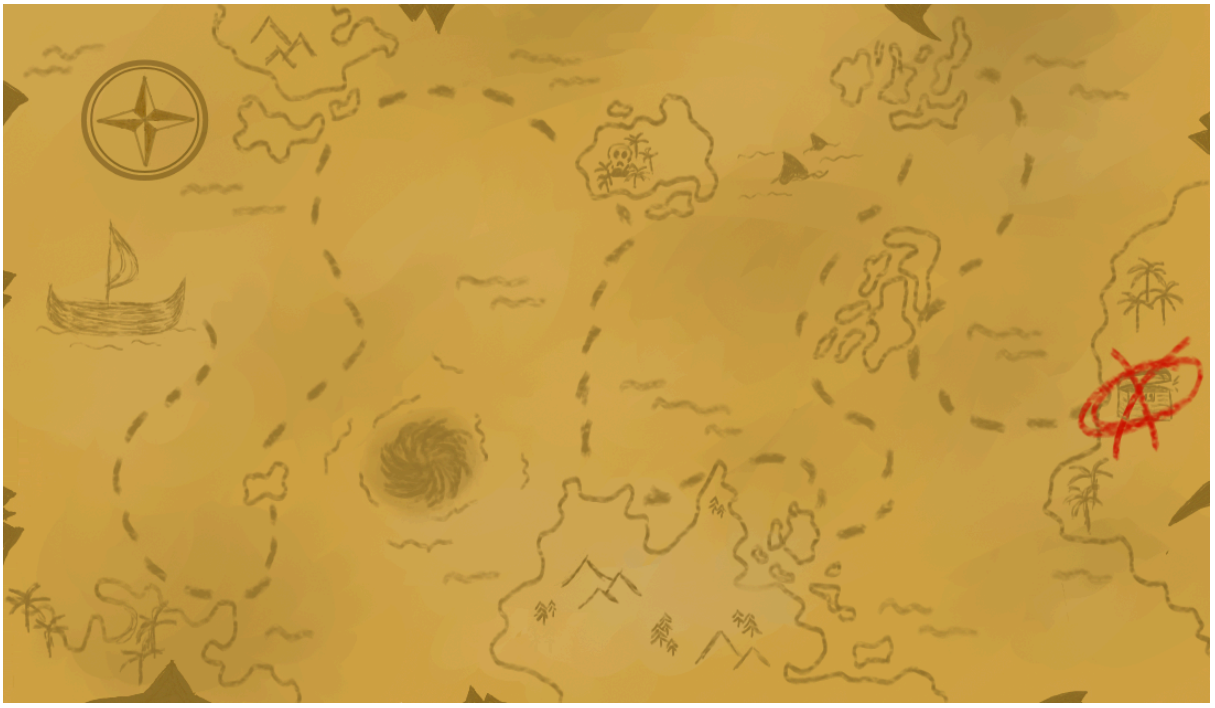
I will also have to draw several other characters that I will make more fantasy like(such as skeletons or marine creatures) as it will represent the enemies controlled by our game's AI. As there will have a choice of characters the AI will have to choose randomly which characters are getting to fight against our crew.

Elements

Our game needs a lot of different elements, some that can be found in open source on the internet, such as vegetation or rubies. But others need to be drawn and personalised for our game. The Elements include the resources needed to upgrade our boat and to increase the health stats available in the treasure chests. It will also include all the elements the player will use during a combat such as the canons(that will have to be animated), the guns and the swords.

We wanted to include a treasure map to see the progression of our boat into the game, and to add to the lore. I did it first to put me into the pirate mood to

start with something not too hard even if it wasn't the most important element to create.



Animation

As we decided to use Ursina as our game engine, animated drawings are possible to implement. Indeed Ursina has a pre built-in class `Animator` which takes as argument a GIF or several images in order to implement it in the code. The implemented animation will be link to the player and will be seen when its argument is passed instead of the texture of the player's cube. The animation frame should 1×1 as to fit perfectly on the cube and the background should be transparent to have the impression that only the character is present rather than a cube.

Here are the first four frames to test how an animation works(where the animation would be available live on the presentation):

1st frame

2nd frame

3rd frame

4th frame

Possible 3D

As the requirements were to make a 3D game we are going to create and implement some 3D elements. To do so I will try to learn and use Blender which is a free application permitting to create 3D objects. My main objective will be for the object to be as light and simple as possible where the meshes will be as minimum as possible. As it will be quite long and difficult to do, only the main elements such as the boat and maybe the chests and other elements(we may choose later) will be produced.

Challenges

As it's a game necessitating a lot of elements and quite a few characters, it will be time consuming to design them and especially animate them. Implementing them in the core will also be a challenging task. Moreover, creating and blending some 3D elements in our game will be an issue I will have to face.

I will also start to focus more on the user interface of our game.

Problems encountered and current limitations

▼ *Degraded performance with particles*

When developing the Particles class, it was realised that keeping particles after a fall / slam, as satisfying as it may be, resulted in a decreased frame rate. So it was required to auto-destroy them after a set delay using the built-in invoke function on a delay.

▼ *Player goes through solid surfaces when traveling at a high velocity (y-axis)*

This was an issue when first developing our main Player class. Because collisions are handled on a local, and class-level basis, only the instance of our current player calculates any possible collisions. It did so at first by using set-length raycasts asynchronously from position updates, meaning sometimes the raycasts would generate at a wrong position.

Now however, we decided that for every frame update, a raycast of length equal to our dy will be calculated from our player and only allow moving at our y-velocity if there are no hits.

▼ *Player does not fully stop by itself*

This solved logic issue was caused by a stopping acceleration that would 'overshoot' V_x = 0

Simply adding a check where if d(V_x) + V_x goes past 0, we just set the velocity to 0.

▼ *Odd 'vibrations' of player entity on jumps when gravity is at specific values*

This might be due to Ursina's animate function and is yet to be solved. We might need to manually handle velocity calculations.

Upcoming work

All is detailed in our Notion. But in general we are going to improve all of what we already did. In terms of characters, they will all be produced(at least the four players) before the 2nd soutenance, at the same time, animations will be produced. Moreover some gameplay elements are going to be implemented in order to have an idea of the interface of our game.

Combat will be our next biggest priority, notably a basic sword, shield, and blunderbuss. Then add some simple server-side methods to distribute health values to clients on hits. Although each client will check for hits and be trusted the distribute correct new health values to the server.

On the networking side, we plan to improve the visual quality of multiplayer interactions. This includes adding interpolation for player movement to make remote players' motion smoother and less abrupt.

We also plan to move part of the collision handling to the server side. While player movement is currently computed locally on each client, the server will be responsible for validating collisions with other entities. This will allow the server to correct invalid positions and ensure a consistent and fair game state across all clients.

Conclusion

The core systems, especially movement and networking, are already functional and demonstrate that our technical choices are viable. The successful implementation of a working multiplayer prototype is a significant milestone and validates our decision to use a client–server architecture with Ursina.

Overall our main goals were achieved by the deadline, which is reassuring, and motivating the team to progress our game. As we communicate well in our team, the work is done smoothly, easily, and information is shared to make sure everyone is in touch with the actual state of our game.