ROOFTOP RIVALS

RAPPORT DE SOUTENANCE FINALE – SAE J3D

SALVAN ACHILLE - BENNJAKHOUKH RAYAN – CLEMENT GREGORY-LUCAS – HADJAB MEHDI – PUIJALON MILAN

MAY 2026 / EPITA PARIS

SUMMARY

Table des matières

Introduction

In order to meet our final assignment, each group was supposed to create a game from scratch. In other words, the game project was much more than just coding; rather, it was a chance to experience all stages of developing a software product from start to finish – from generating ideas to solving issues that arise during the process until finally presenting the results of our efforts. Right from the beginning, we had to seek the necessary knowledge, as well as learn to cooperate as a team during our months of laborious effort.

The development process started in November 2025. We had created the first prototype of the game in December and improved on that prototype in March, but this last report signals the end of the whole project. When reflecting on the journey we have undertaken since the beginning of the process until now, it can be seen that a lot of ground has been covered.

In our game entitled “Rooftop Rivals,” we developed a three-dimensional chase game that involves two players. The game concept is quite straightforward but very efficient. One of the two players acts as a runner while another person acts as the tagger. On top of the screen, there is a timer that displays how much time the player spent being the runner. With the increasing number of seconds that the player spends running, the chances of winning become greater.

But we did not want to develop the game where players chase each other back and forth on the plain field. At the very beginning of our development, our aim was to create a game that would be dynamic and strategic. To make this happen, we designed a system of power-ups where the player picks up a drone that is spread all around the map. Once the drone is picked up by a player, he can have various abilities for a short period of time, including a speed boost, jump boost, invisibility, an opponent’s radar, a shield, or slow effect.

This rooftop context emphasizes the theme of the design. The map is vertical and spacious, created in Blender, consisting of several floors, platforms, and having a clear cyberpunk aesthetic. Moving throughout the space is essential for this game experience; one should be able to move from building to building, not to fall down, and to utilize the geometry itself to either escape or get closer. This complicated the technical aspect of the project greatly, as now the map had to serve both visual and mechanical purposes, have correct collisions and physical behavior, etc.

We used Python to develop the game through the Panda3D engine. We received an opportunity to have a genuine 3D engine at our disposal that would support models, textures, lightings, collision detection, sound effects, and UI components. However, this engine brought its own share of problems as well. The engine has been designed for 3D rendering, yet doing certain tasks, such as creating smooth 2D menus with depth effects, was not natural and required hacks. On a larger scale, making a game in 3D using Python language is a rather audacious thing to attempt within the first year course. However, we knew about this from the very beginning.

One of the areas where this drive manifested itself was in the process of graphical development. The entirety of each three-dimensional object in the game, including the map, the player avatars, the drones, and the navigation mesh utilised for AI purposes, were crafted using Blender from the ground up. None of us came into the project with any knowledge of how to work with three-dimensional models before. Being able to master Blender to the degree necessary for creating objects capable of being properly imported and exported to and from Panda3D proved to be among the hardest tasks of the entire year.

Artificial Intelligence was yet another hurdle to tackle. The AI is responsible for controlling the drones which travel on their own through the terrain and interact with other players. It was necessary to create and implement a custom navigation mesh for proper path finding throughout the multi-layered game map, which took up a considerable amount of time in the latter stages of our project.

When it came to presenting the game in March, we were somewhat lagging behind in terms of map development and artificial intelligence. Thus, for the remainder of the project, our main priority was to try and make progress in those two areas, as well as perfecting all other aspects of the game. It was not an easy task. We are dealing with a fairly sizable

project, coordination between five people is no simple thing, and we had a full course load at the same time, including exams and practicals in C.

Though these were some of the hurdles that we faced while developing our game, our objective has been consistent all through – that is to provide you with a game that is fun to play and is technically sound, visually appealing, as per our own standards and according to the assessment guidelines provided by the school.

This document is an account of the entire journey from our first conception back in November to the final version that you will read right now. From our technical decision-making process to our internal team organization, from what we all did individually to everything that went wrong along the way, it will include everything. It is a true reflection of eight months of work, both its easy and difficult sides, the decisions we had to make, and what lessons we learned from this.This game, Rooftop Rivals, is not only an assignment but also a product of hard work, problem solving, and collaboration. Through this document, you shall see how we transformed a simple tag idea into a three-dimensional interactive game.

Technical Specifications Review

Initial Constraints and Requirements

Since the project has been validated at the end of the first bimester of November 2025, the technology used had been clearly specified. The game had to be coded in Python language, with Panda3D framework on Windows 10 and 11 operating systems. This was not an open-ended statement; rather, it limited our choices all along the year.

The functional specification at that point identified two systems. The first one was an artificial intelligence system where the drones will move around the game map by themselves, interact with the player, and offer bonuses once picked up. The second system was a multiplayer feature where two players can play against each other, with a common timer keeping track of how long they have been runners.

At this point, even the basic concept for our game, being a 3D version of tag in an environment featuring aspects of cyberpunk rooftops, was already established. We had decided what the basic game loop would

be like, how the power-ups would work, and what would the visuals look like in general. One thing that still wasn't clear to us was how difficult it would be to implement everything in 3D.

Three final copies of the specification document have been created during the development process of the project. The first one was delivered in December 2025 in conjunction with the first presentation. The second copy was revised in March 2026 after the second presentation took place, indicating what had changed and what was left to be done. Finally, the third version included in this report is the complete one that shows the state of the project as delivered.

Evolution of the Specifications

It soon became apparent how much we underestimated our original specification from November against what was required when developing the game. To develop a three-dimensional game using Python is not easy, and many aspects of the initial specifications were altered due to the constraints involved.

Certainly the largest evolution in our process was in graphical asset creation. From the outset, the visual elements of our game were known to have a cyberpunk style, with a rooftop map with multiple levels, and custom 3D character assets. What none of us anticipated was just how much work Blender would entail. None of us had used any sort of 3D model design program before this, and making sure that our assets could be exported and used correctly by Panda3D consumed far more of our time than was originally expected. Color retention, UV mapping, animation rigging, and creating our collision meshes all created complications we did not foresee.

Moreover, the artificial intelligence system has been significantly modified since its initial description. At the beginning of the work on the project, it was expected that movement for drones would be fairly straightforward. However, as the work continued, it became apparent that movement through a multi-level map with navigation, which is necessary to implement any autonomous movement, entails building a navigation mesh and a path finding system. This was among the most lagging aspects of our project, and this is what we worked on in the last stage of development.

Multiplayer played an essential role through all iterations of our game. The technical approach to implementing this game mechanic has been changed multiple times as well. At the beginning, the concept implied a fairly straightforward implementation of a local multiplayer mode. But since our game evolved and started incorporating additional mechanics (power ups, drones, role change, and a joint time counter), there became more information which needed to be synchronized between two players.

Another issue that came up and was not in the initial specifications was that the members were using different platforms for their work. Some worked on Mac, while some worked on Windows. This caused problems from time to time related to pathing of files, differences in library behavior, and differences in the way executables could be built.

Furthermore, there were a number of occasions on which the responsibility for carrying out the task changed during the execution of the assignment. The responsibility for the completion and debugging of the menu, for instance, was initially vested in one of our group members. Since, however, the complexity of some of the tasks exceeded our expectations, the responsibility was subsequently passed on to another member. It is quite natural that such changes take place when you have a team of five people.

Final State of the Specifications

When all was said and done, all the fundamental features that were outlined in the initial specification have been incorporated. The game will work on the Windows platform and is powered by Panda3D; it offers a full experience of playing a game of tag in a cyberpunk-style, 3D environment complete with power-ups, drones, artificial intelligence, user interface, and more.

The development of the specification through its different versions can be viewed as the learning process of the team in total. In each case, we had to take a critical look at what we thought was feasible and what was not and where we should concentrate our efforts. It was possible for us to adjust the plan while staying focused on the end goal, and this was an important element of success in creating a fully playable game.

Project Organisation and Management

Team Composition and Roles

The group consisted of five First-year students. Each of them had the principal responsibility of working on one or more aspects of the assignment, although in reality, their lines would overlap as the project progressed.

The position of project lead was not delegated but divided among team members. In fact, coordination was a joint function, with different team members assuming leadership roles depending on which domain needed to be considered. During the execution of the project, the division of labor kept on changing to accommodate various factors like personal workload, technical challenges, and changing priorities in the process of development.

These responsibilities were shared out in the following manner. The networking and multiplayer functionality would be coded by those who showed the most enthusiasm for systems programming. The artificial intelligence and pathfinding would be implemented by those who worked on game logic. The graphical assets, which consisted of the map, characters, drones, and nav mesh, were made using Blender by Rayan. The menus and the user interface were initially given to Rayan but eventually some aspects were done by Mehdi since there were problems that needed to be fixed at an engine level. Milan helped to make the character design initially, but then Rayan expanded on this.

Methodology and Tools

Since the start of our project, we tried to have a weekly meeting on Discord every week. This meeting acted as a regular check-in point where each team member would update us about their progress and also seek help, in case they were stuck somewhere. Such a pattern was critical to our success because we faced tough times in the latter part of our second semester.

We used Git as our version control tool and maintained our Git repository on GitHub. Every member worked on his own branch and merged the code to the main branch from time to time. This approach proved to be efficient in theory, but the problem arose during the merge process. Since more than one person had edited the common file or the

new file had been added that effected the other components of the program, a conflict would often occur and we had to solve it in meetings.

We employed the use of Discord in our daily communication other than during meetings. It enabled us to share screen shots, link documentation and generally communicate regarding any bug or need for coordination among other things. The period just before each presentation saw more intensive communication involving calls that would go on for many hours even extending from one day to another.

For the project, the programming languages used for the game were Python and Panda3D, while the software used for all 3D graphics was Blender. The software used for version control was Git and GitHub, while Discord was used for communication.

Project Timeline

The whole process took place during four bimesters, from November 2025 to May 2026.

In the first bimester, from November to December 2025, the concept of our project was developed, first specifications were drafted, and the construction of the first prototype was initiated. During the first presentation at the end of December, it became clear to us that our proof of concept is still non-existent. At that time, we could not play the game, and there was no visual component whatsoever.

In the second bimester period from January to March 2026, attention turned towards making the game functional. By the second presentation made in March, we were able to develop an operational map, operational character movement, basic power-ups logic, and menu system. Proof of concept was there but it was not fully developed because the map was of poor quality, the character design was of poor quality in some places, power-ups had glitches, and artificial intelligence was yet to be incorporated.

During the third and fourth bimesters, March to May 2026, the team worked on three major issues: the completion of the map and all 3D assets, the integration of the artificial intelligence program, and refining the other parts of the game. This stage was the toughest when considering workload and pressure regarding time management.

Lectures, tests, and programming classes in C language were being conducted at the same time, making it difficult to have lengthy sessions of working on the assignment. Nonetheless, the team still met regularly each week and communicated more often as the deadline loomed near.

By the end of May 2026, the game was done. Everything was functioning well, including all main features; the game had consistent visuals, and it was ready to present and submit.

How the Team Adapted

One of the major lessons that was learnt from this experience in project management is that the initial plans made in the beginning of a project can never stand up to reality. This is because tasks may take much longer than anticipated, system dependencies may create issues, and people may have certain conditions that impact their availability.

What enabled us to move forward was the ability to re-evaluate our priorities with each milestone completed and reallocate work as required. Following the first presentation, it was evident that making sure that we had an operational game was of utmost importance. After the second presentation, it became clear that the areas which needed to be focused on were the map and artificial intelligence. While this was often not an easy task, it provided us with honesty in knowing how to best allocate our efforts.

The group also reached a common ground of the understanding of the project, making coordination simpler. By the end of our project, we all had a clear idea of what other people were working on, thus finding it easy to tell who needed help, stepping in while wasting minimum time on figuring out what they need help with. This understanding was achieved thanks to our frequent meetings where each person presented an update about their work.

INDIVIDUALS CONTRIBUTIONS

Rayan BENNJAKHOUKH – Group Leader

Introduction

In this part of my project report I am going to describe my contributions to the Jeu3D project during the work done in second semester of 2025-2026. The majority of my work focused on the graphical identity of the game; namely all 3D assets created in Blender and all main menus. I also had a part coordinating role in the team.

Our video game was made into a 3-D first-person shooter game based in a world of cyberpunk. The program was created with the help of Python and Panda3D. It is crucial that every single image seen from the very start until the battle area be consistent in terms of style – dark, lit up by neon lights, mechanical, and alive. Consistency throughout this program despite it not being initially created for 3-D graphics was incredibly difficult to accomplish.

Main Menu Development

Context and Initial State

During the start of the second semester, the class already had a working main menu that served its role in navigating through the pages, yet the look of the menu itself was quite plain. The menu didn’t embody the cyberpunk theme that we have established as a group during the first half of the year.

I was assigned the job of adding the newly designed menu on top of this background along with making it have a three dimensional look, which fits in with the rest of the game. However, this was easier said than done due to the fact that Panda3D is a game engine that specializes in three dimensional graphics, making it difficult to make 2D interfaces using it.

Main menu before and after redesign :

Implementation and Difficulties

The entire menu structure was created with the help of a new background, which fit the cyberpunk universe quite well. The main difficulty was to implement an interesting effect of the third dimension by creating a parallax illusion on the menu screen itself. To accomplish this task in Panda3D, one needs to know how to do it with 2D elements using the camera and nodes, which is not described anywhere specifically.

I had a number of bugs that appeared at this stage, almost all of them related to handling of two-dimensional elements with regard to the layering of such elements in connection with three-dimensional scene graph elements. After a lot of efforts, I did not succeed in coming up with a complete fix for those bugs, and therefore decided to transfer responsibility for fixing them over to my teammate Mehdi, who finished implementing these fixes within the engine.

Three-Dimensional Asset Production in Blender

Overview of Responsabilities

Blender made up most of my time in this second semester. I was tasked to create all the three-dimensional assets in the game from the map, the player models, the drone models, and the navigation mesh needed by the artificial intelligence in the game.

Blender was something new to me as the year began since I had not used the application before. However, getting acquainted with various options available on it, such as mesh sculpting, material assignment,

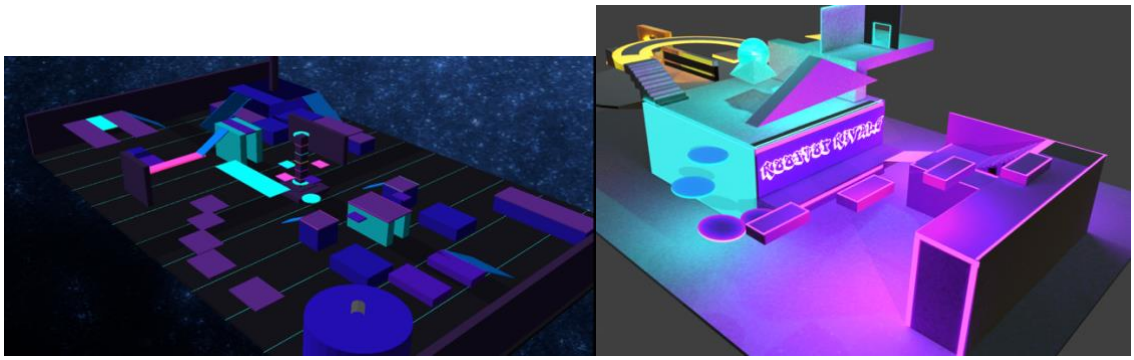
lighting configuration, handling the unwrapping of UVs, and exporting to Panda3D has been the most challenging process I have gone through thus far. Nonetheless, my familiarity with the application now means a lot of time spent experimenting.

The Game Map

The most labor-intensive resource is the map. The game takes place on the level of cyberpunk which is made of layers and contains geometric forms, raised surfaces, corridors, and direction lighting. It took several attempts to design a concept that will suit us.

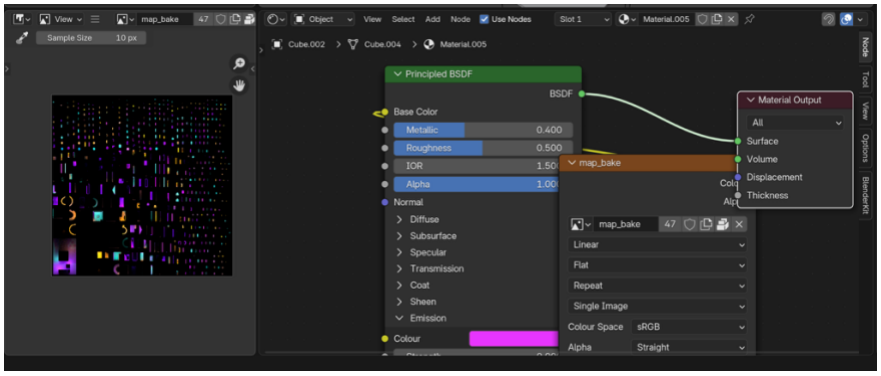
The first map design created for Milestone 2 was functional but had too few elements to fit the criteria. In the final version, the map was recreated to include more complex geometry, different levels of elevation, and also had a far more detailed system of lights. The purpose of this was to create an environment that reflected the neon-filled, dark world of the menu.

The map before and the map after the redesign :



However, what proved to be the biggest technical challenge in the creation of the map was related to the retention of color information when exporting the map from Blender. The program uses a node-based approach for materials; therefore, the data cannot be transferred into Panda3D automatically. Depending on the way that materials were set up, information about materials could be distorted. I needed to do a certain sequence of actions in terms of baking and material settings to force the engine to use my colors properly. In order to make the map

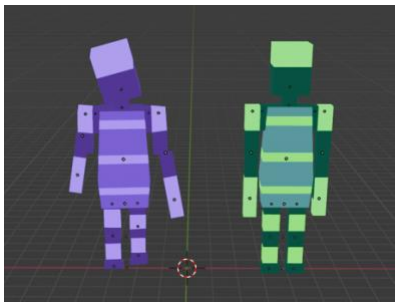
render neon colors of floors and walls and keep ambient occlusion for them, a solution had to be worked out through research and patience; several days were spent on it.



Players Characters

Two playable characters for the game were created by me. Both had to be legible from the size they would be appearing in the game, proportional according to the animation model, and cohesive with the cyberpunk universe that was created for them. For inspiration, I used the first sketches for the characters that were made by my teammate Milan.

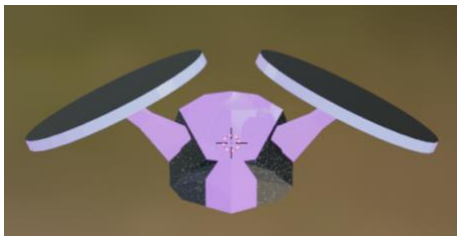
There are two distinct figures that vary only in color, with the first figure having colors in blue, and the second one using violet color shades. Each of the figures uses design cues meant to reflect a sense of mechanicalness without losing its humanoid nature.



Producing the characters was also challenging. Proportions that are right in Blender sometimes fail to look good when they have been placed into the game and moved around. Several edits were necessary when I noticed the results while running the game. In order for rigging of the character to go right and allow proper application of animation data, much weight painting was needed.

Drone Models

Along with the character models, I also created the model of the drone controlled by the artificial intelligence program. The design parameters were essentially the same as those for the characters; namely, that the model be mechanical, fit the theme of the game, and have a cohesive aesthetic style with the rest of the art assets. Creating a drone proved challenging in its own way because although it did not have to take a humanoid form, it still had to look realistic as a machine while being distinctly not a player.



Animations

Animating was one of the hardest technical aspects involved in the process of creating the assets. The animation system in Blender and the runtime engine in Panda3D do not interface seamlessly, and it was therefore challenging to align the animations from both ends.

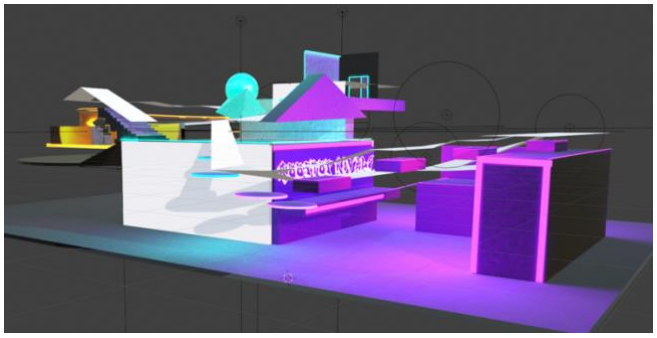
When it came to animating complicated movement cycles, I employed the existing rigs and data sets under open source licenses that were subsequently adapted to fit my own character riggings. The development of complete animation cycles starting from scratch for humanoid rigged characters is an elaborate process that would take a much longer time than what our project's timeline allows. There was also considerable effort in aligning the bone hierarchies, fixing the naming convention conflicts between the different rigs, as well as cleaning up any artifacts produced by the imported files.

Navigation Mesh for the AI

An example of one of the things I did, but not something that would have been as obvious, is creating the navigation mesh needed for the artificial intelligence. It is the artificial intelligence that manages the drones, and

for them to move around the map automatically, the engine needs a surface above the floor geometry.

This would be a relatively easy concept in theory. However, due to the fact that we have multiple levels, elevated platforms, slopes, and different geometrical forms at different elevations, creating an accurate mesh that allows the drone to navigate correctly without clipping through any walls and floating above any surfaces turned out to be a very exacting task. It took me about two days to fine-tune the mesh to perfection vertex-by-vertex in Blender, comparing each iteration to see how well the drone navigated upon importing to the engine.



Game Packaging

In addition to all these in-game assets, I have also created a visual package of the game in terms of its cover artwork. Here again, it was necessary to come up with a cohesive design concept for the product as a whole. In particular, the cover of the game matches the tone and theme of the rest of the visuals developed for it.

Team Coordinator

In addition to my technical contributions, I always tried to ensure that the project progressed efficiently. This involved sending tutorials and

references for those parts of the project that other people were finding difficult, assisting when somebody was stuck, and ensuring there was a collective understanding of where we were headed.

The truth is that this was among the hardest tasks during the project process. Managing a group of five people for eight months, having in mind the different motivational levels that each person could have, was not easy. There were times when there was very little communication, and the task that was to be completed failed to be fulfilled within the stipulated time frame. To overcome this challenge, I made sure that I was accessible all the time to my group, but there is only much one person can do. Despite all the challenges faced, we have successfully completed our project.

Personal Assessment and Skills Aquired

Blender ended up being the biggest software skill I learned as I went from having zero knowledge about it all the way to comfortably working in modeling, rigging, materials, and animations. The issue that I encountered was that the assets turned out to be quite extensive in scope and left me with little time to contribute code-wise, as I sometimes had to wait until my team members used the completed models. However, despite these problems, I still consider the product created by the team to be quite amazing. I now have a better understanding of game creation.

Achille SALVAN – Technical Lead

Introduction

As a core developer for *Rooftop Rivals*, my responsibilities encompassed the most critical, math-heavy, and technically demanding systems of the game. I fully designed and implemented the Peer-to-Peer multiplayer architecture, the synchronization logic between players, the physics-based player movement mechanics, the A* Navmesh Artificial Intelligence, and the event driven Heads-Up Display (HUD).

P2P Networking & Multithreaded Event System

Objective:

To create a seamless, low-latency multiplayer experience, I went for a decentralized Peer-to-Peer architecture using WebRTC (via the aiortc library) rather than a traditional authoritative dedicated server. The goal was to establish direct UDP data channels between the two clients to minimize latency.

Implementation:

Establishing a P2P connection requires a first step called "Signaling." Because two computers cannot naturally find each other on the internet behind routers and NATs, I implemented a WebSocket signaling server (`wss://rooftop-rivals.alwaysdata.net`). The process works as follows:

1. **Signaling connection:** Firstly, both clients create a WebSocket connection with the signaling server. It is this server which associates each player with UUID and lets players find each other's through party codes instead of IP.
2. **SDP Exchange:** The Host generates a Session Description Protocol (SDP) "Offer" detailing its media and data capabilities. This is sent to the signaling server, which relays it to the Guest. The Guest then replies with an SDP "Answer" which has similar content.

3. **ICE Candidates:** Simultaneously, both clients gather Interactive Connectivity Establishment (ICE) candidates. These are potential network routing paths (local IP, STUN server, public IP, or TURN server relays). They exchange these via the WebSocket until a direct path is successful. Once connected, the WebSocket is ignored, and game data flows directly between players via a WebRTC “DataChannel”. To maximize bandwidth efficiency, I completely avoid JSON strings during gameplay, serializing the game state into highly compact binary packets using Python’s “struct” library
   Exemple: packing a player's state into 22 bytes:
   “`struct.pack("B5fB",`
   ```
     msg_type,   # Indicates what is the packet content
     x, y, z,    # Player position
     yaw, pitch, # Player head rotation
     anim_stat   # Player animation state
   )”
   ```

Challenges Faced:

The most significant challenge in this section was not the network protocol itself, but multithreading. WebRTC relies on asynchronous operations, whereas Panda3D operates on a synchronous frame by frame main loop. Running them together initially caused massive frame drops and race conditions. I solved this by isolating the network into a completely separate thread (“`_run_async_loop`”).

To safely bridge the asynchronous thread and the synchronous game loop, I implemented an Event Queue system (“`queue.Queue`”). The network thread safely pushes incoming binary payloads into this queue, and Panda3D pulls from it exactly once per frame *via* a dedicated task (`self.update_network`). This guaranteed thread safety and eliminated micro-stutters.

Game Mode Logic & State Synchronization

Objective:

In a nervous tag game, latency can cause severe inconsistences between what the two players see. My goal was to design a synchronization model that resolves conflicts fairly while keeping combat perfectly responsive.

Implementation:

I designed a hybrid authority model within the “TagGame” class to handle synchronization:

- **Host Authority for Game State:** The Host controls the state of the game. The Host manages the main “runner_time” tracking, increments scores, and dictates round transitions. At specific intervals, the Host sends a "forced sync" packet to overwrite the Guest's timers. When a round starts, the Host generates random spawn coordinates and forces the Guest to initialize there.

- **Predictive Combat (Priority to Attacker):** For actions like tagging (attacking), waiting for a Host validation would make the game feel unresponsive and unfair for the Guest. I implemented a "client-side prediction" model. When a player presses attack, the local client immediately performs a raycast check. If the the attacker succeeds to hit the other player, the hit is validated locally. The local client immediately updates the roles (Tagger/Runner) and sends a message packet containing the Knockback vector (“vx, vy, vz”). The receiving client unconditionally applies this impulse to their physics controller.

Challenges Faced:

The main difficulty was handling simultaneous events, for example, when both players try to attack each other at the exact same millisecond. Because of the "Priority to Attacker" logic, both clients might briefly think they are the Tagger. I resolved this by enforcing the Host's overriding packet as the absolute truth. When roles are swapped, the host will send the other player a packet with its game state, and the Guest will update its game state. Desynchronization can't occur, because if it occurs, the Host immediately fixes it.

Advanced Movement Physics

Objective:

To create a fluid, momentum-based parkour system, simple coordinate manipulation was insufficient. I needed to interact directly with Panda3D's Bullet Physics engine, which is harder, less “Plug and Play”, but way more powereful. It allows me to manipulate gravity, velocities, and most importantly, apply impulses.

Implementation:

I separated the movement mechanics into distinct sub modules (“PlayerSlide”, “PlayerDash”, “PlayerWallrun”) managed by a central machine which has also smaller mechanics (movements, slopes, jump, and others).

- **The Slide:** Sliding relies on conserving and slowly dissipating initial momentum. To allow players to slide smoothly down slopes, I calculated the cross product of the slide vector and the ground's normal vector. I then took the cross product again to yield a perfectly slope-parallel velocity vector. When this is done, I just have to control the velocity of this vector.

- **The Dash:** When dashing, the controller temporarily disables gravity. To prevent players from exploiting the dash to fly infinitely upward, I extracted the camera's forward Quaternions and clamped the Z-axis vector. If the Z component exceeds a certain angle, it is capped, and the X/Y components are normalized and multiplied by a value which is the dash force. The impulse resulting to this creates the dash effect.
- **The Wallrun:** This was the most mathematically complex mechanic. It required so much testing and adjusting, that I had to first create it in Godot and test it there then translate the code to Python with Panda3D. I implemented a dynamic wall detection system casting 5 rays (from -45° to +45°) relative to the player's horizontal velocity vector. Once a wall is detected, the script applies to a highly reduced gravity. To keep the player from falling off, a continuous force is applied in the exact opposite direction of the wall's normal vector which mechanically sticks the player to the wall.

Challenges Faced:

A major issue with the Wallrun was determining when the player should detach from the wall. Initially, players would get "glued" to the wall until he jumped. So, I implemented an angle limit logic, the script calculates the dot product between the camera's horizontal look vector and the vector parallel to the wall. If the player looks away from the wall beyond a certain limit (45 degrees), the wallrun script instantly stops the wallrun, restoring standard gravity and allowing for a smooth stopping.

Navmesh Extraction and A* AI

Objective:

Distribute collectible power up drones across the rooftops and allow them to fly autonomously to new locations using pathfinding, avoiding obstacles and gaps.

Implementation:

Instead of relying on a grid, which is terrible for vertical 3D environments, I built a custom “NavmeshSampler”.

- **Geometry Extraction & Random Spawning**: The script reads the raw 3D geometry, decomposing it into individual triangles. Simply picking a random triangle to spawn a drone is mathematically flawed, as some places contain a lot of small triangles, and others few massive triangles. To fix this, I calculated the area of every triangle using half the magnitude of the cross product of its edges. I then used a weighted probability distribution based on the total surface area to select a triangle. Finally, the drone is spawned on this random triangle to ensure perfect uniform distribution across the map.
- **A-Star algorithm:** A* is a pathfinding algorithm which gives the shortest path from one point to another in a given graph. The graph nodes are the geometric centers of the triangles. Adjacency is determined by detecting if two triangles share at least two vertices. The A* algorithm calculates the shortest path using a “g_score” (actual distance traveled) and an “h_score” (heuristic) which is an estimation of the distance to the end. My heuristic is a strict Euclidean distance calculation from the current triangle center to the target triangle center.

Challenges Faced:

Processing 3D pathfinding is computationally expensive. If the drones recalculated their path to every frame, the game framerate would drop. I solved this by decoupling the logic. Pathfinding and target picking happen in “update_logic(dt)” which runs at a limited server tick rate. Meanwhile, the visual smoothing, which includes a mathematical sinewave bobbing effect and Quaternion Linear Interpolation to smoothly rotate the drone towards its target, is handled in “update_visuals()” every frame.

Dynamic HUD & Event-Driven Interface

Objective:

To build a working and clear HUD that communicates round states, timers, and ability cooldowns in real-time without hurting game performance.

Implementation:

The HUD features circular animated cooldowns for the Dash, Attack, and Power-ups. Rather than using a GPU shader, I pre-rendered an array of 150 image frames. The code simply maps the remaining cooldown time to an index in this array, setting the texture index instantly.

For UI animations, such as the role transitions, I utilized Panda3D's “Sequence” and “LerpColorScaleInterval” to create smooth alpha fade ins and fade outs, avoiding UI pops.

Challenges Faced:

The main challenge was performance. Originally, the HUD polled the player's variables in every single frame in a massive update function. This was inefficient. I refactored the entire HUD to use Panda3D's event system. The physics controller now broadcasts events only when a cooldown is actively ticking, or an action is triggered. This event-driven architecture drastically reduced the CPU load on the rendering thread.

Mehdi HADJAB - Game Manager

General Role and Project Context

During the development of **Rooftop Rivals**, my work was mainly focused on gameplay systems, user interface, player feedback, and the integration of several technical features into the main structure of the game. Rooftop Rivals is a multiplayer tag-based game where one player takes the role of the runner while the other player takes the role of the tagger. The main objective of the runner is to survive as long as possible, while the tagger must catch the runner before the timer reaches the limit.

My role in the project was not limited to one isolated feature. I worked on several connected systems that helped make the game more dynamic and easier to understand for the player. These systems included the **power-up system**, the **collectible drone system**, the **main menu**, the **server creation and joining interface**, the **in-game pause menu**, the **HUD**, and some parts of the **multiplayer flow**. I also contributed to technical work related to the Blender map, the navmesh, collision issues, and general bug fixing.

One of the most important goals of my work was to make the gameplay less repetitive. Since the game is based on a chase mechanic, it could become predictable if both players only relied on basic movement. To solve this, I worked on a system of temporary abilities called **SUPER Powers**. These powers give players short advantages during a match and encourage them to move around the map instead of staying in one place. Players obtain these powers by collecting drones placed in the environment.

At the beginning of the project, I had to fix several issues related to the power-up system. One of the first bugs I corrected was the problem where a power-up appeared completely shifted compared to the player after being collected. This made the mechanic visually confusing because the object did not behave as if it was properly attached to the player. I also fixed another bug where the drone stayed above the player indefinitely even after the power-up was inactive. This was important because the drone was supposed to be linked to a temporary gameplay effect, not remain permanently visible.

This work also required me to understand several parts of the Panda3D engine. I had to learn how Panda3D manages models, node paths,

collisions, transparency, rendering order, and task updates. Since Rooftop Rivals is built in Panda3D, many gameplay features depend on how objects are loaded, displayed, updated, and removed from the scene. Understanding these systems helped me debug issues more efficiently and connect my features correctly to the rest of the game.

I also contributed to the general structure of the game. The main application initializes Panda3D, configures model loading for `.glb` and `.gltf` files, sets up the camera, starts the music manager, creates the multiplayer client, and manages the transition between the menu and the game. Because of this, my work on menus, power-ups, and multiplayer logic had to fit inside a real-time game architecture rather than exist as separate scripts.

Overall, my contribution was mainly about connecting the player experience with the technical systems behind the game. I worked on what the player sees, what the player collects, how abilities are activated, how feedback is displayed, and how the game transitions between menus, matches, rounds, and lobbies. These contributions helped Rooftop Rivals become more complete, more readable, and more strategic.

Power-Up System and Collectible Drones

One of my main contributions was the implementation and improvement of the **power-up system**. The goal of this system was to give the player temporary abilities that could influence the match without completely breaking the balance of the game. These abilities were designed to create more variety during gameplay and make each round less predictable.

The power-up system is based on several classes. The generic `PowerUp` class stores the name of the power, the file associated with it, the player attribute affected by the power, a multiplier, and a duration. When the power is applied, the system saves the original value of the player's attribute, modifies it for a limited time, then restores the original value when the effect ends. This made the system reusable because different powers could share the same basic structure while changing different player attributes.

The first major power I worked on was **SpeedBoost**. This power increases the player's movement speed for a short period of time. At first, SpeedBoost became very strong because it allowed the player to

reach other drones quickly and chain several powers together. This created a balancing issue, so I proposed and implemented changes to prevent power-ups from stacking too easily. This was an important design decision because a power-up system needs to be fun without making one player unfairly overpowered.

I also worked on **JumpBoost**, which increases the player's jump speed and maximum jump height. This power was more complex than a simple speed boost because it had to interact with the player's movement and physics system. After testing and debugging, the JumpBoost became functional and added more verticality to the gameplay. This was useful in a rooftop environment where movement across different heights is part of the game's identity.

Other powers were also tested and improved. **Invisibility** originally made the player slightly transparent, but I later improved it so that the player became almost completely invisible. This made the power more useful because it allowed the runner to hide or reposition more effectively. **Radar** made the opponent visible through walls by changing the opponent's visual model and rendering it in a red color. This gave the player a temporary information advantage, which is very useful in a map with obstacles and vertical structures.

I also worked on **Shield** and **Stun**. The shield gives the player a temporary protection state and displays a transparent visual effect around the player. The stun power was later changed into a slow effect because a complete stun could feel too frustrating for the opponent. Instead of completely blocking the player, the slow effect reduces the opponent's movement speed for a limited time. This made the ability more balanced while still giving the player an advantage.

Another important improvement was the **RandomPowerUp** system. Instead of always giving the same power, a drone can resolve into one random power from the available list. This makes the game less predictable because players do not always know exactly what they will receive before collecting a drone. However, the random power is resolved once and then stored, which prevents the power from changing later during activation or update. This helped make the system more consistent.

The power-ups are obtained through the **collectible drone system**. This system manages drones placed in the world and checks when the player touches them. Each drone contains an item linked to a random

power-up. When the player collects a drone, the drone gives the player a power and is removed from the active collectable list.

The drone system also uses a navmesh to find valid spawn positions. This is important because drones should not appear in unreachable locations or outside the playable area. At the beginning of the match, several drones are spawned on the map. Later, new drones spawn regularly every 20 seconds. This keeps the mechanic active throughout the match and encourages players to keep moving.

This system changed the rhythm of the game. Players now have a reason to explore the map, take risks, and fight for control of power-up locations. The drones are not just decorative objects. They are connected to the strategy of the match because collecting one can change the balance between the runner and the tagger.

HUD, Player Feedback, and Interface Work

Another major part of my work was the **HUD** and the different interface systems. A power-up system only works well if the player can clearly understand what is happening. For this reason, I worked on displaying useful information directly on the screen during gameplay.

The HUD shows several important elements: the player's role, the current game status, the score, the local player timer, the network player timer, progress bars, the crosshair, dash cooldown, attack cooldown, and the current power-up icons. This made the game easier to read during fast gameplay situations. Instead of guessing whether an ability is active or whether an action is available, the player can look at the HUD and immediately understand their current state.

The HUD also displays the **main power-up** and the **secondary power-up**. The main power-up appears as a larger icon, while the secondary power-up appears as a smaller icon. When the player receives or changes a power, the HUD loads the correct texture from the power-up assets. This visual feedback makes the inventory system clearer and helps the player know which ability is ready to be used.

I also worked on the cooldown displays. The HUD uses countdown textures to visually represent the cooldown of dash, attack, and power-up duration. Instead of only showing numbers, the system updates an image frame depending on how much time remains. This gives the player a quick visual understanding of when an ability will become available again.

The HUD is connected to gameplay events. It listens to events such as dash updates, attack cooldown updates, power-up updates, power-up cooldown updates, and crosshair changes. This event-based structure is useful because the HUD does not need to directly control the gameplay systems. Instead, gameplay systems send updates, and the HUD reacts by changing the correct visual elements.

I also contributed to the **main menu**. The main menu uses a cyberpunk-style background displayed on a 3D card. Instead of using a simple flat menu, the interface is rendered in a way that feels more connected to the visual identity of the game. The menu includes buttons for starting the game, opening settings, and exiting.

One technical part of the menu is the offscreen buffer system. The interface is rendered into a texture, then displayed on a 3D quad. This required creating a separate 2D camera, a render texture, and a picking system that converts mouse clicks into coordinates on the menu surface. This allowed the player to interact with buttons even though the menu is displayed inside the 3D scene.

I also worked on the multiplayer interface. The menu allows the player to create a server or join one using a six-character code. The code field handles keyboard input, focus, backspace, and error feedback. If the code is invalid, the border changes color to show the player that there is a problem. This made the connection process easier to understand.

The lobby system was also improved. When the host creates a server, the lobby displays the server code and waits for the second player. The start button remains disabled until the second player is connected. When a guest joins a server, the menu displays the connected state and allows the player to leave if needed. These screens made the multiplayer flow clearer and more complete.

Finally, I added an **in-game pause menu**. Pressing Escape opens a pause interface with buttons to continue the game, open settings, return to the main menu, or exit. When the menu opens, the game clears the player inputs and unlocks the mouse pointer. When the menu closes, the game locks the pointer again and resumes gameplay. This was important because the player needed a clean way to pause, adjust settings, or return to the menu without closing the whole application.

Game Loop, Multiplayer Flow, and Tag Game System

I also worked on systems connected to the main game loop and multiplayer match flow. Rooftop Rivals is not only a local prototype. It is designed as a multiplayer game, which means that player actions, positions, rounds, timers, and score changes need to be synchronized between two clients.

The `Game` class is the central structure that connects the main elements of a match. It creates the Bullet physics world, applies gravity, sets up lighting, creates the local player controller, creates the network player when needed, loads the world geometry, initializes the collectable drone system, and starts the tag game mode. This structure is important because all major systems need to communicate with each other during gameplay.

The game loop is separated into two update rhythms. Some elements update every frame, such as the local player, the network player, the visual animation of collectables, and physics simulation. Other elements update at a fixed tick rate, such as collectable logic and tag game logic. This separation makes the game more organized because visuals can stay smooth while gameplay checks run at a controlled frequency.

The multiplayer system uses binary messages to synchronize important actions. For example, player position, rotation, jump events, attacks, timer synchronization, respawn positions, round information, game over data, and slow effects can all be sent between clients. This was necessary because both players need to share the same match state.

I made small changes to the multiplayer functions in the main application. The application handles the transition between menu and game, checks when both players are ready, and starts the match only when the map has been loaded correctly on both sides. This prevents one player from entering the match too early while the other client is still loading.

The **TagGame** system manages the rules of the match. It stores both players, determines who is the runner and who is the tagger, manages the current round, updates the score, synchronizes timers, and handles the end of the game. The match uses a timer system where the runner

gains time while surviving. If a player reaches the required score, the game ends and the HUD displays either a win or lose overlay.

The round system is more advanced than a simple chase. In the first round, one player starts as the runner and the other as the tagger. In the second round, the roles are reversed. After that, the role assignment can become random. This gives both players a fair chance to experience both roles.

At the start of each round, the players are placed in starter spawn positions. The system also applies a temporary speed modifier of zero to create a countdown phase. The runner is blocked for a shorter duration than the tagger, which gives the runner a small head start. This is important for balance because the runner needs enough time to escape before the chase begins.

When the tagger catches the runner, the roles are swapped. The HUD updates immediately, and the host sends synchronization data so that both clients share the same role state. This prevents desynchronization, where one client could believe a player is the runner while the other client believes the opposite.

The game-over flow is also handled by this system. When the match ends, the correct overlay is displayed depending on whether the local player won or lost. After a short delay, the game automatically returns to the lobby. This creates a complete game loop: menu, lobby, match, round progression, game over, and return to lobby.

World Geometry, Map Integration, and Final Impact

Another important technical contribution was related to the **world geometry** and the integration of the Blender map into the game. The map is not only a visual object. It needs to be correctly loaded, centered, scaled, connected to the physics system, used for spawning, and linked to the drone placement system.

The world system loads the map from a `.glb` model. After loading, it calculates the bounds of the model, finds its center, and recenters the model in the Panda3D scene. It then applies a map scale so that the environment has the correct size in the game world. This was important because assets exported from Blender do not always have the correct scale, origin, or orientation when imported into a game engine.

The system also includes a fallback loading method for `.glb` and `.gltf` files. If the normal loading process fails, the code tries to load the model directly with a glTF loader. This made the map loading process more robust and helped reduce technical problems when working with Blender exports.

A major part of the world system is the collision mesh. The code extracts the geometry of the map and builds a static Bullet triangle mesh from it. This mesh is then attached to the physics world as a rigid body with zero mass. Because of this, the map becomes a real physical environment. The player can walk on it, collide with it, and interact with it through the physics system.

The world system also calculates safe spawn points. It uses the map bounds to choose random positions, then casts a vertical ray from above the map to find the highest valid ground position. This prevents the player from spawning under the map, inside the environment, or in the void. In a rooftop game, this is especially important because vertical positioning is central to the gameplay.

The starter spawn system also helps create fairer round starts. It tries to find two spawn points separated by a fixed distance and placed at the same height. This works with the countdown system and helps both players begin a round in a controlled situation.

The map is also connected to the drone system through the navmesh. The world system loads a separate navmesh file, aligns it with the map, scales it, and can display it in wireframe debug mode. This navmesh is then used by the drone manager to find valid positions for collectible drones. As a result, drones are less likely to spawn in unreachable or invalid areas.

I also contributed to visual atmosphere through the sky system. The world system loads a sky texture and attaches it to the camera as a background element. It is configured so that it does not interfere with depth or lighting. This helped reinforce the futuristic rooftop mood of the game.

Finally, I worked on cleanup and transition issues. When the game ends or returns to the menu, the system needs to remove the map, lights, physics bodies, collectables, network player, HUD, navmesh, and sky. Without proper cleanup, old objects could remain in the scene or memory and create bugs when starting a new match. This was important because the game includes transitions between menu, lobby, match, and return to lobby.

In conclusion, my work on Rooftop Rivals helped transform the project from a simple chase prototype into a more complete multiplayer game. I worked on the power-up system, collectible drones, HUD feedback, menus, multiplayer flow, tag game logic, map integration, collision setup, spawn logic, and several bug fixes. These systems are connected together: drones give powers, powers update the HUD, the HUD helps the player understand the match, the match is controlled by the tag game system, and the world geometry supports movement, spawning, and collisions.

Through these contributions, I helped make the game more dynamic, more strategic, and more polished. The final result is a multiplayer rooftop chase game where movement, timing, power-ups, roles, and map exploration all work together to create a stronger gameplay experience.

Milan PUIJALON – *Creative Director*

Project Report - Music Creation, Integration into a Video Game, and 3D Modeling

Project Objectives

The project had three main objectives:

1. Create a high-quality soundtrack that faithfully reflects the spirit and atmosphere of the game.
2. Integrate this music into the video game using functional Python code.
3. Design characters that fit the game's universe and could be implemented in a chase-and-pursuit gameplay context.

More specifically:

- The soundtrack had to convey a cyberpunk universe: a bustling futuristic city dominated by technology, while also expressing a certain nostalgia for the human condition in the face of increasing robotization. To capture both sides of this world, the soundtrack needed to include both calm and melancholic pieces, as well as more aggressive and dynamic tracks.
- The characters had to look futuristic and robotic in order to fit the cyberpunk setting. They also needed to reflect the gameplay by representing the cat-and-mouse dynamic, and remain visually consistent with the final map.

Planning – How to Achieve These Objectives?

Music

Although I have been playing drums and piano since childhood, I had no prior experience in music composition. I therefore had to learn this discipline from scratch while simultaneously mastering a complex and versatile software. I chose Studio One.

Characters

I had no prior experience in 3D modeling. I had to learn how to use a professional modeling tool while experimenting on my own to build models from scratch. I also had to think carefully about the artistic direction of the characters.

Code

I needed to learn how Python's class system works and write efficient code, something I had never done before this year.

Project Progress

1. Music Creation

I began with music creation, as it was the only area where I already had some practical experience. However, this time I had to work within specific constraints related to artistic direction (the cyberpunk universe), as well as spatial and time limitations.

Given the cyberpunk theme chosen with the team, I decided to compose music in two main genres:

- Electronic music: for its futuristic and technological feel, as well as its dynamic quality, which emphasizes the tension during chase sequences between hunter and prey.
- Chill-hop: a relatively calm form of instrumental hip-hop that evokes the melancholy of the modern world in which the characters live.

For production, I used Studio One, a DAW (Digital Audio Workstation) that offers very interesting features but requires time and effort to master. Alongside composing, I watched tutorials and researched how the software works.

I compose primarily using a MIDI keyboard controller, which sends data to my computer and allows me to access a wide variety of sounds without needing to record acoustic instruments, a valuable advantage when working in a limited space.

Studio One interface:

To compose, I use virtual instruments such as Impact (built into Studio One) for the rhythm section, audio samples used as virtual instruments, and presets such as those provided by Analog Lab V (a preset suite included with an Arturia keyboard). These presets are then modified through a modulator called Presence, another tool bundled with Studio One, which allows the shape or frequency of a signal to be adjusted to achieve the desired sound.

Impact:


IMPACT XT
Perc Sandy
Trigger
One Shot
Loop
Toggle
Layer Mode
Choke
Quantize
Follow Tempo
Offsets
Start
End
Reverse
Normalize
Perc Radio
All Notes Off
Reset
A
B
C
D
E
F
G
H
Shaker
Perc
Snare Top
Ride
Hat Short
Perc Rain
Hat Op
Cymbal
Kick Top
Perc Sandy
Hat Cl 1
Hat Cl 2
Kick Low
Rim
Snare Low
Clap
Pitch
Transpose
Tune
Env
Vel
Filter
Cutoff
Res
Drive
Punch
Soft
Amp
Gain
Pan

Presence:


Instruments
1 - Presence
Telecaster Slide FNVol
None
Auto: Off
Compare
Copy
Paste
Midilar
LFO 1
Rate
Sync
Key
Free
Delay
LFO 2
Filter
Soft
Drive
Punch
Res
Vel
Key
Cutoff
Amp Env
A
D
S
R
Global
Volume
Velocity
Poly
Mono
Glide
Telecaster Slide FNVol
ElectricGuitar
Program:
Fender Telecaster
American solid-body, single-cutaway electric guitar
Velocity Switch, Open->Slides
Probability
No Efx Time
FN Level
FretNoise
Voices
64
36.85 MB
Env 2
Sample Start Mod
Pitch Fine Tune
Sample shift
Transpose
MOD/FX
PRESENCE XT
FX A
FX B
Mod A
Mod B
Modulation
Chorus
Flanger
Phaser
Mono
Delay
Speed
Width
Depth
Reverb
Pre
Damp
Size
Low
High
Mix
C1
C2
C3
C4
C5
2 Bend 2
Bend
Mod

Finally, the process moves on to mixing and mastering. During the mix, I apply effects to certain tracks and balance the frequencies and audio levels, ensuring that all sounds are clearly audible and that the listening experience is comfortable. Mastering involves applying compression to individual tracks and the overall audio file, as well as adding finishing effects, with the goal of enhancing the overall sound quality.

Here is what the compressor used during these two stages looks like:

2. Character Creation

Before starting, I needed a quality 3D modeling application. I chose Blender and began by watching tutorials. However, I quickly realized that Blender has a steep learning curve: the controls are not intuitive, the range of possible operations is vast, and many keyboard shortcuts are required to work efficiently. I often felt lost, and without consistent practice, I kept forgetting the commands.

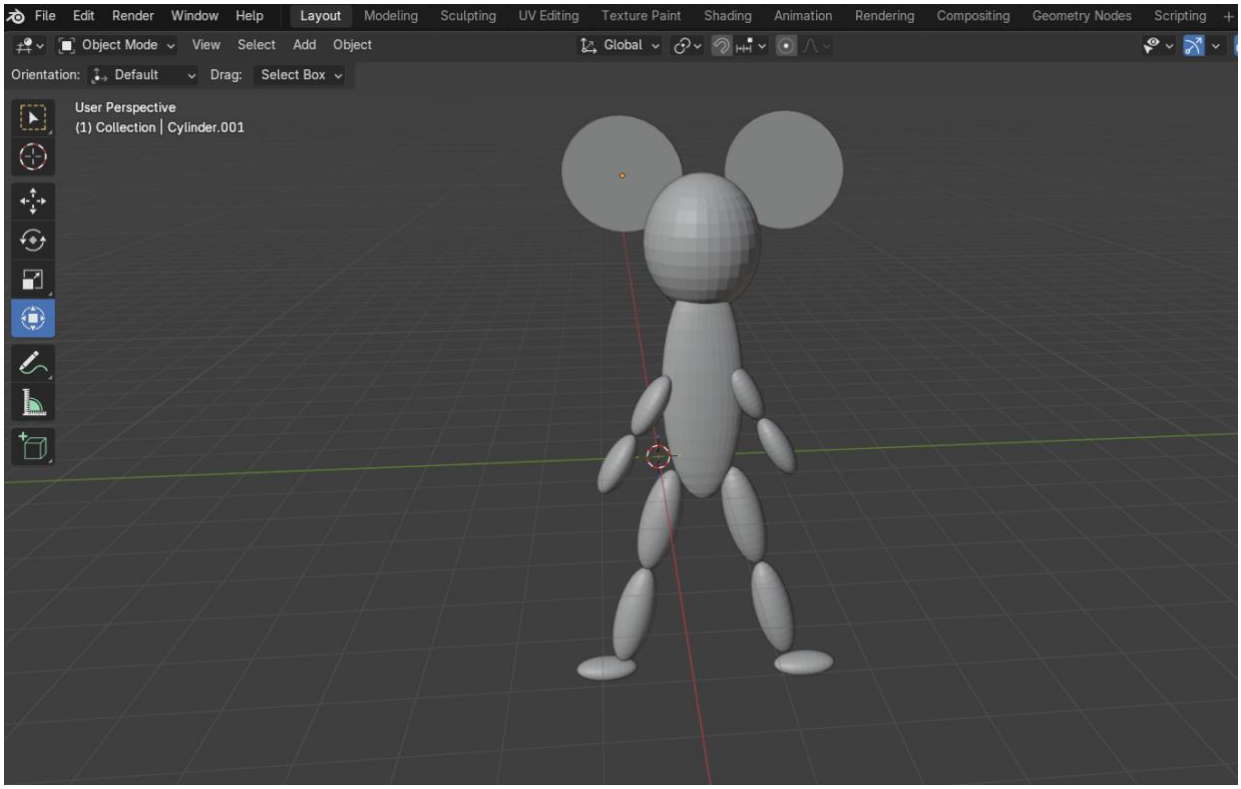
I therefore decided to start by creating simple shapes and performing basic operations on them, while continuing to learn Blender in parallel. This hands-on approach helped me retain information more effectively. After some time, I began assembling basic shapes to build characters.

I started with spheres, stretched them into the desired forms, then rotated and repositioned them to combine them together. The result was a reference body that outlined the future appearance of the characters.

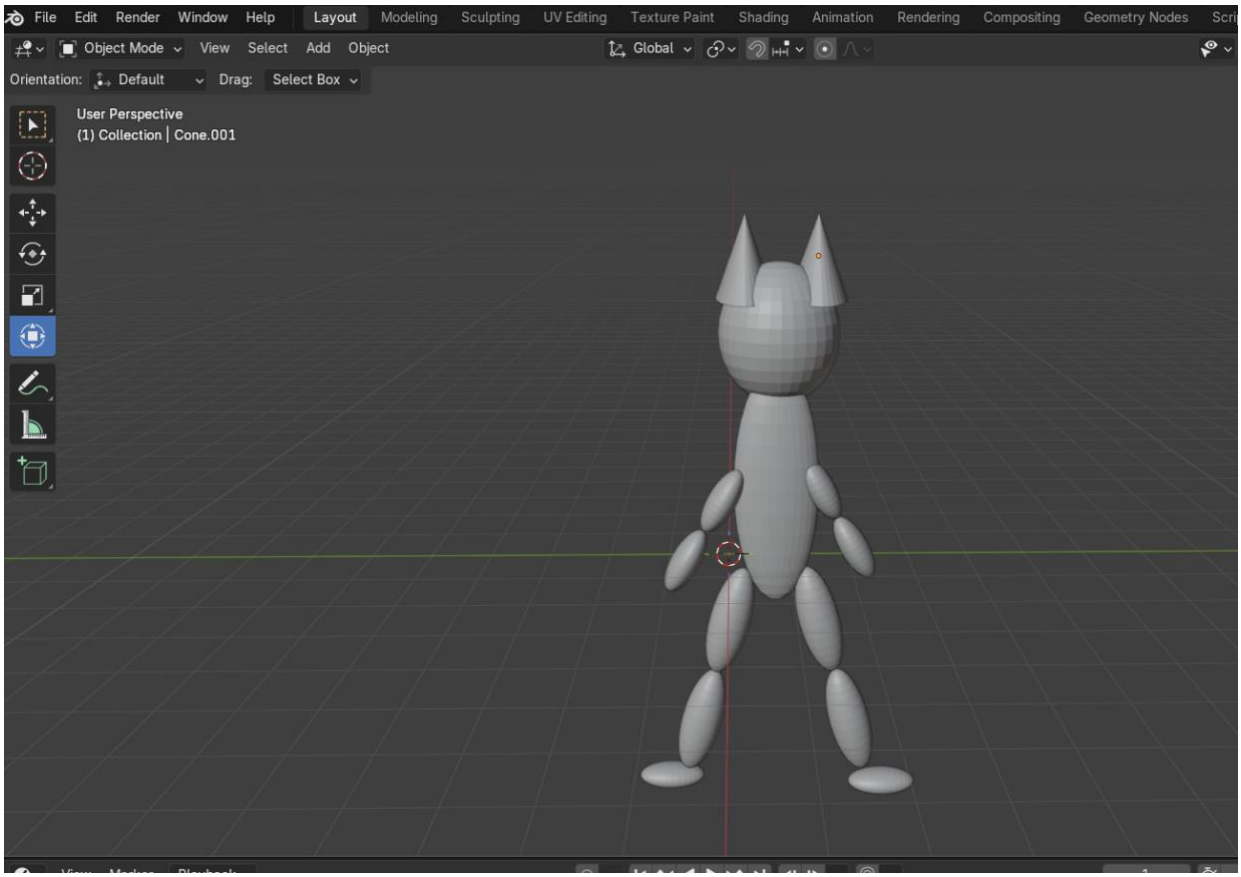
Given the concept of the game (a tag game, or cat-and-mouse), two distinct and easily recognizable characters were needed: a mouse (the prey) and a cat (the hunter).

Images of the first 3D models:

- The mouse



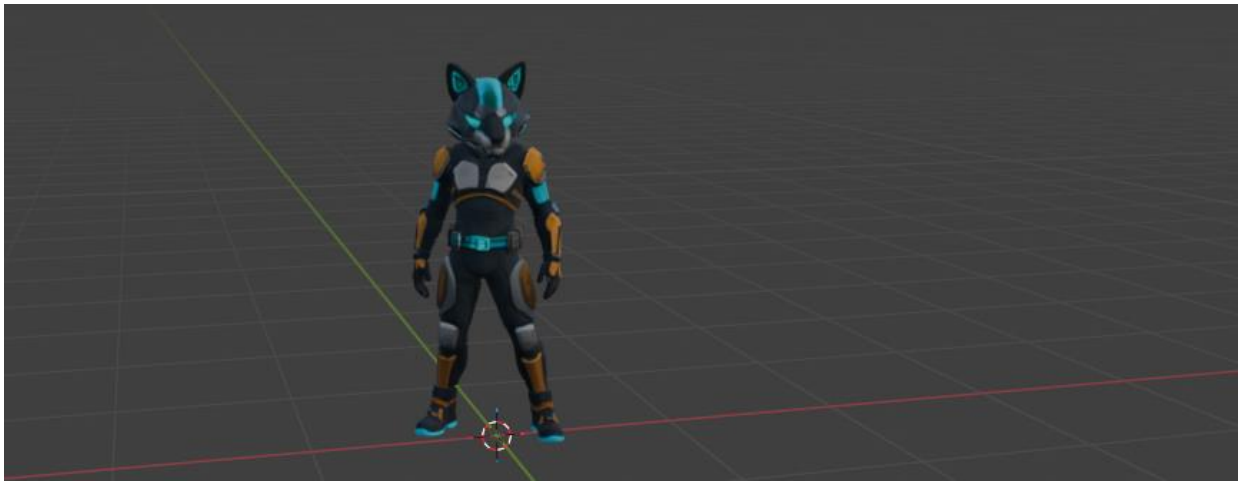
- The cat



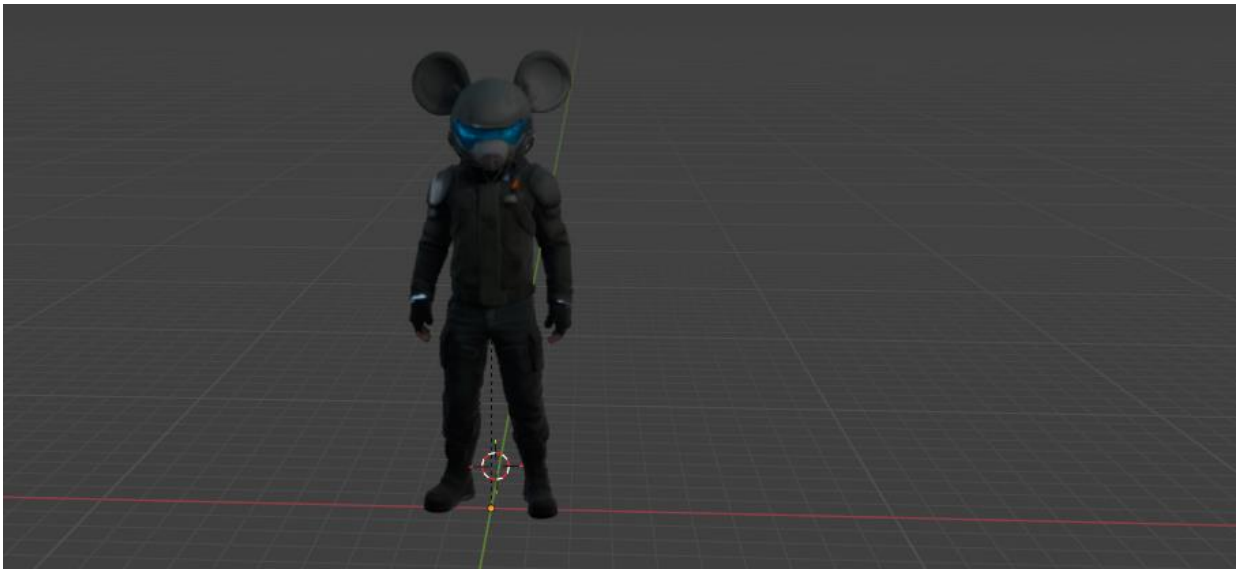
Subsequently, I improved the characters by adding more texture and a more convincing pose, for example:

After adding colors, here are the final renders:

- The cat:



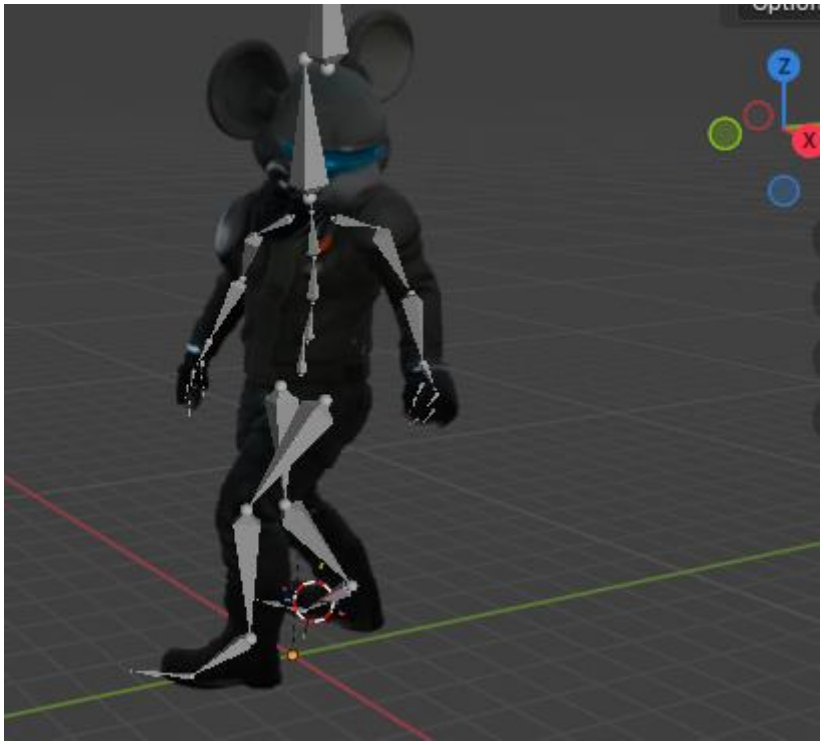
- The mouse



All that remained was to animate the characters to make their movements believable within the game. I created three animations per character: running, jumping, and idle (standing still).

Images of the animations in Blender:


root



3. Decision Not to Integrate the 3D Characters – Reasons

Ultimately, the team decided not to include these characters in the final version of the game. There were two main reasons:

- Hardware limitations: the 3D models and animations were too resource-intensive, causing significant slowdowns, especially since two players needed to be displayed on screen simultaneously.

- Artistic direction: the style of the characters did not match the low-poly aesthetic that the final game and its map had adopted. After a team discussion, we concluded that the characters were incompatible with both the visual direction and the performance limitations of a non-gaming computer, particularly in multiplayer mode. We therefore decided to change direction. Rayan, who had already built the map in this low-poly style, was well-positioned to create consistent, better-performing characters that could be properly integrated into the game.

4. Music Integration – Code and Implementation

Technical Implementation A music management system was implemented to enable random playlist playback during games.

Creating the Music Manager The file game/music_manager.py was created. It contains the MusicManager class, which does not inherit from ShowBase in order to avoid conflicts with the game's main class. A reference to the main application is passed to the constructor to access Panda3D's loader and taskMgr.

The manager holds a playlist of paths to audio files. It selects a track at random, avoids playing the same track twice in a row, and periodically checks whether the current track has ended in order to automatically move on to the next one. Methods are provided to start, stop, and adjust the volume.

Integration into main.py

Four changes were made in main.py:

- Import of MusicManager at the top of the file.
- In **init** of MainApp: initialization with self.music_manager = MusicManager(self).
- In start_game(): a call to self.music_manager.start() before the game instance is created.
- In destroy(): addition of self.music_manager.stop() to stop the music when the application closes.

No new functions were created; existing functions were simply extended.

Resource Organization A musics/ folder was created at the root of the project to store the audio files. The .ogg format was chosen for its strong compatibility with Panda3D. The MusicManager playlist was configured with relative paths pointing to these files.

Version control with Git

a feature/musique branch was created, and the changes were committed and pushed. A pull request was opened. A conflict arose in menu.py, which had been modified by another contributor. Since no local changes had been made to that file, I chose "Accept incoming change" to keep the remote version. The pull request was then merged into origin/main.

Coexistence with the menu music the menu.py file has its own independent music system, which plays a track on loop. Both systems coexist without conflict: the menu music plays at launch, then stops when a game begins. The MusicManager takes over during gameplay. Returning to the menu automatically stops the in-game music.

Summary of Completed Steps

1. Brainstorming on artistic direction.
2. Learning Studio One.
3. Beginning composition in Studio One.
4. Learning Blender.
5. First 3D models created.
6. Music progress:

    a. Menu music: calm, nostalgic, electro-futuristic (for the waiting screen).
    b. In-game music: more dynamic, to convey the tension of the chase sequences.
7. Improvement of 3D models (colors and textures).
8. Creation of animations (running, jumping, idle) for both the cat and the mouse.
9. Characters finalized → not integrated (for technical and artistic reasons).
10. Music finalized: 1 track for the menu, 3 tracks for gameplay.
11. Code written to integrate the music into the game.
12. Final version of the code pushed to the repository.
13. Testing with the team - final version validated.
14. Writing of the defense report and preparation of the oral presentation.
15. Rehearsal with the team.

Gregory-Lucas CLEMENT - Team Coordinator

HUD Design and Implementation

Objectives

My main objective for this presentation was to design and implement the HUD of Rooftop Rivals. In a chase-based game where every second matters, the HUD is the only channel through which the player understands, at any moment, who they are, what is at stake, how the score is evolving, and which abilities are available. A second non-negotiable goal was to preserve the cyberpunk identity built by the rest of the team: the HUD had to feel like a direct extension of Rayan's menus, not a grafted-on debug overlay.

Because the HUD lives on top of every other system, I knew it would not be built in a single pass. I approached it iteratively: ship a working first version quickly, identify what fails in real play, and rewrite the weak parts. This document follows that timeline, with six measurable sub-objectives used as a yardstick across both iterations:

* Display only actionable information for the player.
* Occupy less than ~10% of the screen surface.
* Stay readable on dark and bright neon zones of the map.
* Stay consistent with the menu's font, panel tone and accents.

* Update without costing frames - no per-frame work the HUD does not strictly need.
* Make the architecture trivial to extend when teammates ship new mechanics.

Research and Design Thinking

Before writing a single line of code, I studied how cyberpunk-adjacent games handle their HUD. Cyberpunk 2077 anchors dim semi-transparent panels in the corners with neon accents only on critical states. Mirror's Edge removes almost everything to keep the player focused on parkour. Deus Ex sits in between. From these references I drew three rules: the centre of the screen must stay nearly empty (only a crosshair belongs there), persistent information sits in corners against the edges, and colour must encode meaning, never decorate.

Combined with the six sub-objectives, this produced the early zoning diagram below: an identity strip at the top, two corner clusters at the bottom for active mechanics and secondary feedback, and a sacred central area for the crosshair.

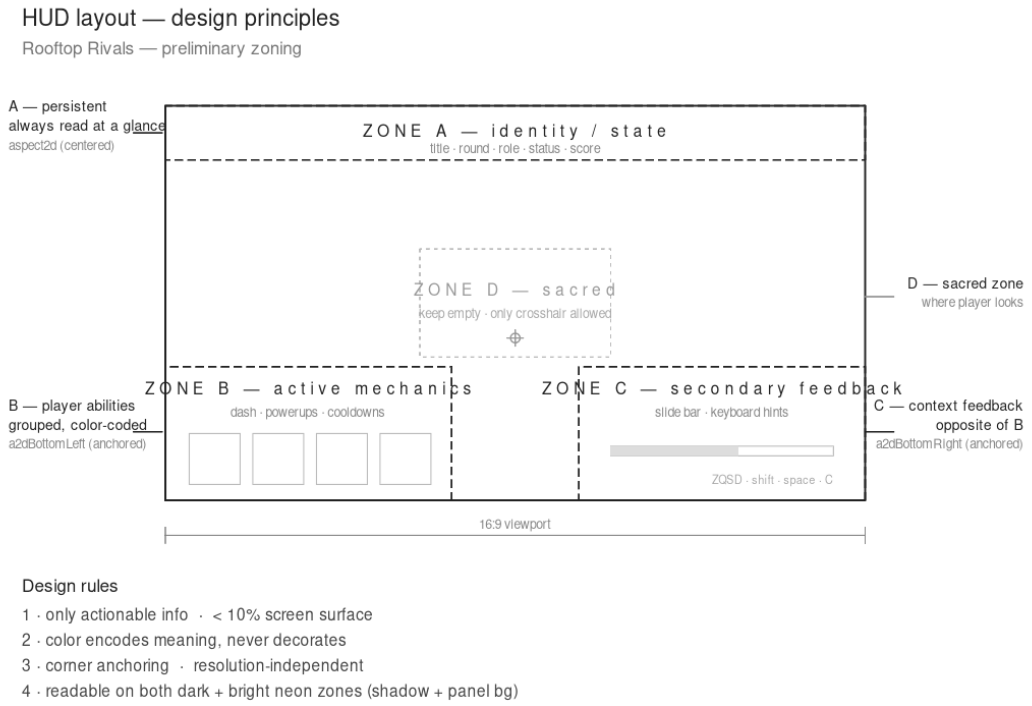


Figure 1 - preliminary HUD zoning, before any code was written.

First Iteration - Text-Based, Per-Frame HUD

The first version validated the design philosophy as quickly as possible, with zero asset dependency. Every element was a text element rendered with Panda3D's OnscreenText. Changing a colour, a position or a label

was a one-line edit, with no asset pipeline to maintain. The screen was divided into three regions: a top bar with title, round, role and status; a centred score line; and two bottom corners holding the dash icon plus five text power-up slots (SPE STU INV RAD SHI, the three-letter prefixes of each power-up name) on the left, and a slide bar on the right.

The HUD was structured as a hierarchy of self-contained widgets following the same three-method contract (init, update, destroy):

```
HUD  |-- PowerupHUD  |    |-- _PowerupSlot x 5  |-- SlideHUD
```

To avoid repeating OnscreenText configuration everywhere, I factored everything into a single _make helper. Each active power-up slot ran a small state machine on the remaining duration: above 15 seconds it used its own colour, between 5 and 15 seconds it turned amber, below 5 seconds red. The progress bar itself was rendered with a single Unicode character repeated, avoiding pre-rendered images for each fill level.

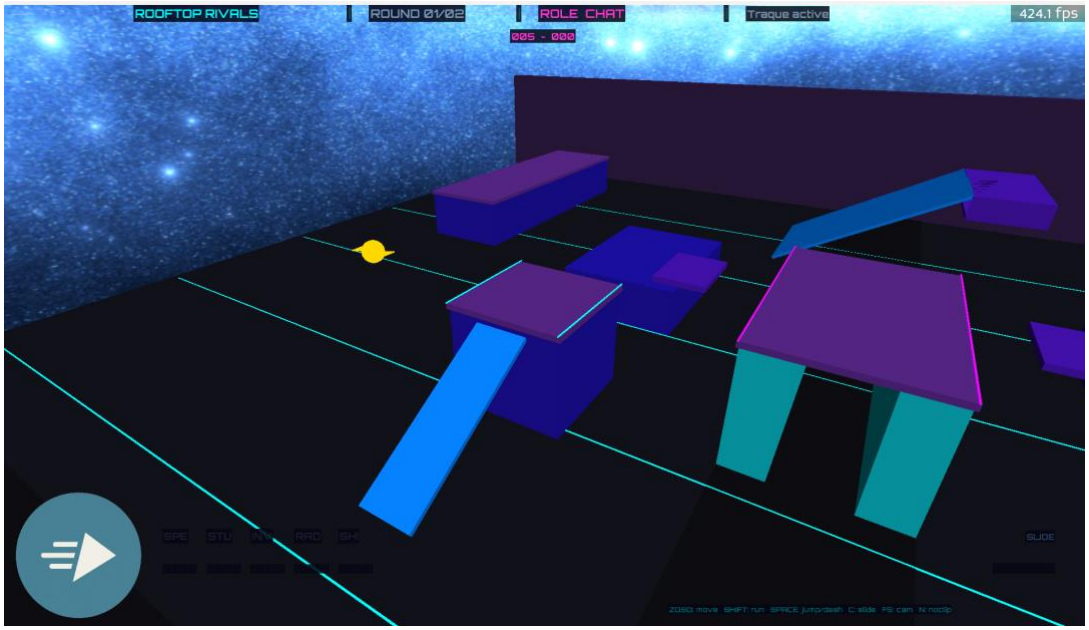


Figure 2 - first iteration in-game. Note the five grey power-up slots SPE/STU/INV/RAD/SHI bottom-left, even with a drone (yellow) in plain view.

That first version was visually complete and gave us something to play against very quickly, which was the whole point. But there was one important caveat that became obvious in playtests, and is directly visible above: the power-up column, while fully drawn on screen, never activated during play. The HUD exposed a notify_powerup method that was supposed to be called by the pickup site, but that call was never wired in collectable.py. The slots stayed grey for the entire match no matter how many drones Mehdi delivered. The display existed; the integration did not. This was the strongest argument for rewriting.

Identified Limitations

Once we ran real multiplayer playtests with Achille's movement and Mehdi's power-ups integrated, four problems surfaced that no polish on the first version would have solved.

Performance. The HUD called hud.update() from TagGame.tick on every frame, which Achille noticed was halving our frame rate. Every OnscreenText.setText triggers a re-layout of the underlying TextNode, expensive at 60 Hz across a dozen labels. This was not a tuning problem; it was an architectural mismatch.

Visual weight. In playtests the five-slot text column felt clinical. SPE STU INV RAD SHI read like a debug console rather than cyberpunk UI. We were missing icons, animations on pickup, feedback on role swaps.

Missing game moments. No countdown when a round started, no clear win/lose screen, no animation on role changes. In a tag game these are the three moments where the HUD should be loudest, and ours was silent.

Coupling - the power-up display did not work. The notify_powerup API required collectable.py to actively reach into the HUD, which created a hard cross-module dependency Mehdi and I were uncomfortable with. The slots stayed grey every playtest. The information we wanted to surface the most was the one the first iteration could not deliver. Patching with another direct call would have papered over the structural problem; we wanted a coupling model the rest of the team could live with.

Second Iteration - Image-Based, Event-Driven HUD

The second version keeps the same design philosophy but replaces almost the entire implementation. The top of the screen is now a horizontal score block: two timers flank a central score line, each paired with a DirectWaitBar that fills as the corresponding player accumulates runner_time. Blue for the local player on the left, red for the opponent on the right; the two bars literally race each other towards the win threshold. A semi-transparent top_overlay image anchors the block visually. The role and status labels move to the top-left corner so they no longer compete with the score.

A small crosshair sits at the exact centre, with two states: neutral, and target-locked when the local player is in attack range of the runner. The bottom-left cluster holds four OnscreenImage icons: main power-up, secondary power-up, dash, attack. Each has its own 150-frame

cooldown ring, reusing the technique Achille developed for the dash but applied uniformly to every mechanic.

Figure 3 - second iteration in-game. The top score block races two timers and progress bars (blue local, red opponent), the role label sits in the top-left, the crosshair is centred, and the bottom-left cluster shows the four image-based icons (main and secondary power-ups, dash, attack) each with its own cooldown ring.

The most important architectural change is that the HUD no longer polls. Each mechanic emits a Panda3D event when its state changes, and the HUD subscribes at construction:

```
self.tag_game.game.app.accept("update_dash_hud",
self.update_dash_cooldown)
self.tag_game.game.app.accept("update_attack_hud",
self.update_attack_cooldown)
self.tag_game.game.app.accept("update_main_powerup_hud",
self.update_main_powerup)
self.tag_game.game.app.accept("update_powerup_cooldown_h
ud", self.update_main_powerup_cooldown)
self.tag_game.game.app.accept("update_crosshair",
self.update_crosshair)
```

This solves the performance problem entirely: the HUD does work only when work is needed. The frame-rate regression Achille flagged is gone. More importantly, the messenger pattern fixes the power-up integration failure of the first iteration. Mehdi's pickup code does not need to know

the HUD exists; it just emits 'update_main_powerup_hud' and 'update_powerup_cooldown_hud', and the HUD listens. As a direct consequence, the power-up display actually activates in play for the first time. Slots that stayed grey the whole match in V1 now light up the moment a drone is collected, and the cooldown ring drains correctly until the effect expires.

The second iteration also introduces a real animation pipeline based on Panda3D's interval system. The start-of-round countdown (5-4-3-2-1-GO) is built as a Sequence of Func and Wait nodes. The role change overlay uses two LerpColorScaleInterval calls wrapped in a Sequence to fade in over 0.25s, hold one second, then fade out. Win and lose follow the same pattern. Every Sequence is paused in destroy, and every overlay goes through a _destroy_attr helper that falls back from destroy() to removeNode().

Four Panda3D coordinate spaces are now in play: aspect2d for the crosshair, a2dTopCenter for the score block, a2dTopLeft for the role labels, a2dBottomLeft for the ability icons. Each anchors children to the corresponding screen edge regardless of resolution. Image layering is controlled with setBin('fixed', N): icons at 5, cooldown rings at 6, score text at 10, timers at 10-15, crosshair at 20 - always on top.

Visual Identity and Coordination

The HUD shares Rayan's Orbitron-Bold font and sits in the same dark blue panel family as his menu's DirectFrame. The two backgrounds are no longer bit-for-bit identical (the in-game panel at (0.04, 0.08, 0.14, 0.72) is slightly lighter and more transparent because it sits on a busy 3D scene rather than a flat menu), but they read as belonging to the same world. The role colours are consistent across both versions: the tagger is warm and aggressive (red-pink), the runner is cool and evasive (mint green). Below is the current palette.

Colour	RGBA	Usage
Accent cyan	(0.08, 0.95, 1.0, 1.0)	Primary accent, echoes the menu title.
Tagger red	(1.0, 0.35, 0.35, 1.0)	Hunter role colour - aggression.
Runner mint	(0.35, 1.0, 0.85, 1.0)	Prey role colour - evasion.
Local / Network bars	blue / red	Two-bar race readable at a glance.
Panel background	(0.04, 0.08, 0.14, 0.72)	Same dark blue family as Rayan's menu panel.

Coordination with the team produced three concrete decisions. With Achille, every cooldown in the game (dash, attack) now reuses his 150-frame ring asset driven by HUD events, so every cooldown looks visually identical across features. With Mehdi, the HUD reads power-up state by listening on the messenger rather than calling a HUD method from his code, which makes adding a new power-up zero-cost on my side. With Rayan, the font and panel family are shared so the menu-to-game transition stays continuous.

Challenges and Difficulties

Three difficulties were the most consequential during the bimester.

Panda3D's coordinate systems. My first implementation placed every element in aspect2d, which worked on my test resolution but pushed power-up icons partially off-screen on a wider monitor during a team test. I assumed it was a scale issue and tried to compensate with multiplicative factors, which made things worse. The real fix was to re-parent elements to a2dBottomLeft and similar anchored sub-nodes, which makes positioning resolution-independent by construction. This lesson directly informed V2's use of a2dTopLeft and a2dTopCenter.

Per-frame updates. The first version updated the HUD on every tick, halving the frame rate. I initially tried to make update cheaper (caching last drawn values, skipping setText calls when unchanged), but the gains were modest. The real fix was structural: stop driving the HUD from a per-frame tick, let the mechanics tell the HUD when they actually changed. The messenger pattern in V2 closes Achille's TODO entirely.

Cleanup of running animations. V2's Sequences and LerpIntervals introduced a new failure mode: if the player exits while a Sequence is still running, the engine keeps dispatching events to a HUD being destroyed. The fix is a small but essential discipline in destroy(): pause every running Sequence, ignore every accepted event, and route every overlay image through a _destroy_attr helper that tolerates missing attributes. The _destroyed guard ensures cleanup runs at most once.

Conclusion and Future Work

The HUD now displays the two competing timers and bars, role and status, score, main and secondary power-ups, dash and attack cooldowns, and a state-aware crosshair. It animates the round start countdown, role swaps, and win/lose screens. It survives the multiplayer role swap and the brightness variations of the map, and pays nothing in frames per second outside the moments it actually has work to do.

Crucially, the power-up display - the main visible failure of V1 - now activates correctly in play thanks to the messenger architecture.

The methodological habit I am taking from this bimester is more valuable than any specific API: defining a small set of measurable rules at the start of the project, and rewriting the implementation when the original version failed three of them, was the right call. The first iteration was not wasted; it was the only honest way to discover where the design needed to push back. The second iteration exists because the first one exposed exactly what it could not handle.

For the next phase, three groups of work remain: polish the existing widgets (icon swap animations, score flash on tag, colour pulse on the closer progress bar), add an end-of-round scoreboard overlay reusing the Sequence pattern, and extend the HUD with new widgets the team discussed (opponent direction indicator when the Radar power-up is active, stun icon at screen centre when the local player is stunned). The event-driven architecture makes both additions a single accept and a small handler away, with no structural change required.

Collective Assessment

During the development of **Rooftop Rivals**, one of the most important achievements of our group was not only technical, but also organizational. At the beginning of the project, we were five students with different levels, different tasks, and different ways of understanding programming. We had to learn how to work together on the same codebase, how to divide responsibilities, and how to help each other when someone was blocked. For me, this was one of the biggest successes of the project.

One of the main challenges was learning how to collaborate using **VS Code** and **Git**. At first, working on the same project was not always easy. We had to deal with merge conflicts, commits, branches, and situations where one person’s code could affect another person’s work. Sometimes, Git created frustrating situations, especially when we had to go back to previous commits or fix bugs caused by different versions of the project. However, these difficulties also forced us to communicate more and to become more careful with the way we shared our work.

This collaboration helped us build stronger cohesion as a group. Each member had their own role, but we were not completely isolated from each other. When someone was stuck, another person could explain a part of the code, help debug a problem, or give advice. This was especially important because many systems in the game were connected. For example, the power-up system needed to interact with the player controller, the HUD, the drone system, the map, and sometimes even the multiplayer logic. Because of this, understanding only one file was not enough. We had to understand how our work affected the rest of the project.

Technically, one of the biggest successes of the group was the multiplayer system and the movement system. The multiplayer part, mainly handled by Achille, was a major technical achievement because Rooftop Rivals is built around a duel between two players. Without multiplayer, the main concept of the game would not really exist. The game needed to synchronize positions, roles, attacks, timers, rounds, and game states between two clients. Seeing this system work was one of the moments where the project started to feel much more serious and complete.

The movement system was also a major achievement. Since Rooftop Rivals is a rooftop chase game, movement is one of the most important parts of the gameplay. If the movement felt slow, rigid, or unpleasant, the whole game would suffer. The addition of dynamic movement mechanics helped make the game more enjoyable and more adapted to the rooftop environment. Running, jumping, dashing, wall-running, and moving through a vertical map made the chase more exciting and gave the game its identity.

For me, the moment when the project really started to look like a real game was when we had a first playable map, a good part of the movement system, and some powers that were already working more or less correctly. At that point, even if everything was not perfect, we could finally play something that looked like the game we had imagined. There was a real environment, a character that could move, and gameplay elements that made the match more dynamic. This was a very motivating moment because it proved that the project had a solid base.

After that first playable version, we decided to go further. We did not want to keep only a simple test map and unfinished mechanics. We worked on creating a more detailed map, improving the power-ups, adding new features, making the movement more dynamic, and

improving the interface. This progression was important because the project did not stay at the prototype stage. Step by step, it became closer to a complete game.

However, the project also came with major difficulties. The biggest collective difficulty was adapting to the development of a **3D game** using **Panda3D**. For most of us, this was a new environment. We did not only have to learn Python, but also how a 3D engine works. This included models, textures, cameras, lighting, physics, collisions, node paths, game loops, and imported assets. For beginners, this created many barriers at the start of the project.

The people working on Blender and 3D models also faced their own difficulties. Blender can be difficult to understand when you are not used to 3D modeling. Creating assets is already a challenge, but exporting them correctly and making them work inside the game engine is another one. Sometimes a model looked correct in Blender but caused problems once imported into Panda3D. This created issues with scale, orientation, collisions, or hitboxes. These problems were not always easy to understand because they were between artistic work and technical implementation.

To overcome these difficulties, we mostly relied on research, practice, and testing. We searched documentation, tried things directly in the project, observed what worked and what failed, and repeated this process until we understood more. When someone was blocked for too long, they could ask another member for help. This was how we slowly learned the patterns of the code and the logic of Panda3D. At the beginning, many parts of the project looked confusing, but after seeing similar structures several times, they became easier to understand.

Our organization was also important. Each member had a main area of responsibility. I worked mostly on the power-ups, menus, interface improvements, and some HUD-related parts. Achille worked mainly on movement and multiplayer. Rayan worked on the map modeling. Milan worked on music and player models. Gregory joined later and helped with the HUD. This distribution helped us move forward because everyone had a clear direction.

Most of our communication happened on Discord. When someone had a question, wanted to test something, or needed a meeting, we used Discord to talk about it. We also used a Google Doc to keep track of what had been done each day and what still needed to be done. This helped us maintain a clearer view of the project’s progression. Even if

everything was not perfect, I think we were generally well organized because we were able to follow our tasks and adapt when problems appeared.

Overall, the collective experience taught us that making a game is not only about writing code. It is also about communication, organization, testing, debugging, and adapting to other people's work. A game project has many moving parts, and each member's work can affect the others. This was sometimes difficult, but it also made the final result more rewarding. By the end of the project, we had not only built a game, but also learned how to work as a real development team.

CONCLUSION

In retrospect, we realized that we have gained so much more knowledge from this project than we originally expected. In eight months, we did not only gain knowledge on Python coding and 3D modelling, but we also have acquired skills in being able to work on a software project on our own and the magnitude of which is much larger compared to any that we have attempted before. From managing our own GitHub repository, solving merge conflicts, collaborating with others, adjusting priorities every time we give a presentation, everything that we had experienced is invaluable.

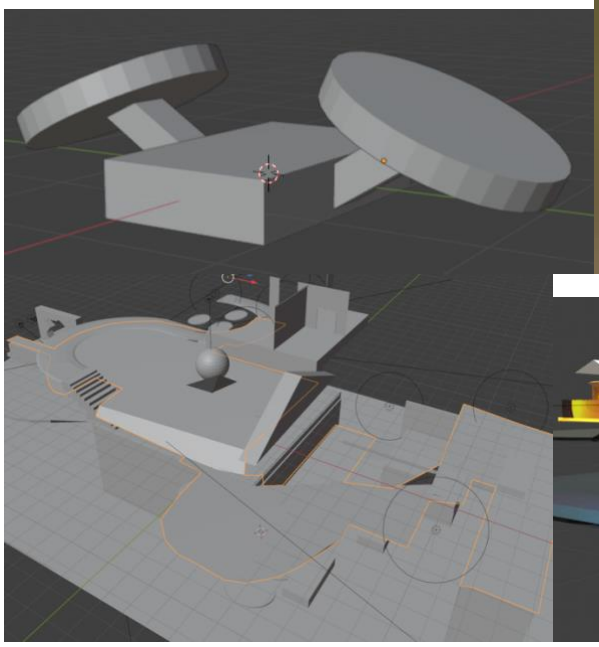
Apart from developing new skills, we were also given a chance to see how we handle adversity. We encountered various setbacks throughout the way and learned to adapt. Although the feedback that we received might not always be encouraging, it was always constructive, pointing us towards the good direction.

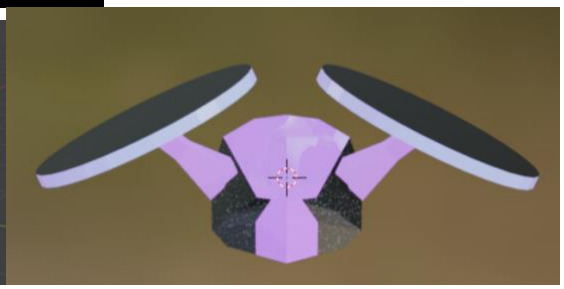
Overall, despite the difficult moments we've experienced, we are extremely proud of our videogame. Rooftop Rivals is a fully functional game with an intact 3D design, something that we created from scratch ourselves as rookies in our first year.

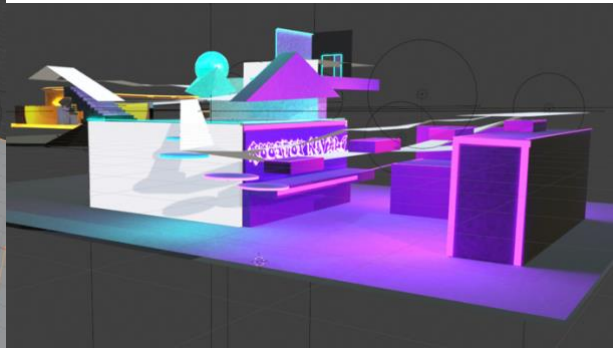
Appendix

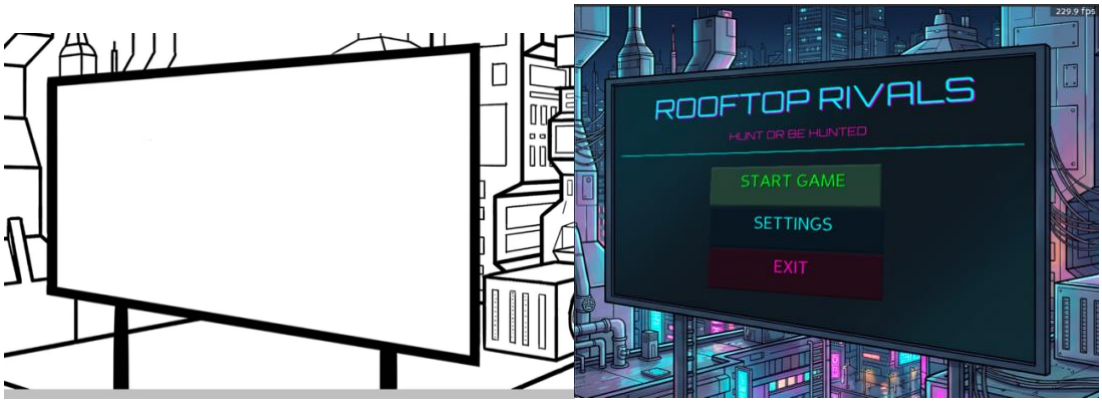
Game visuals :

Concept Arts :

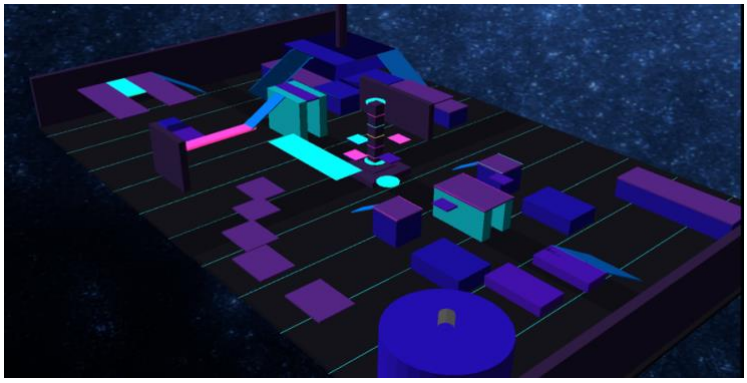




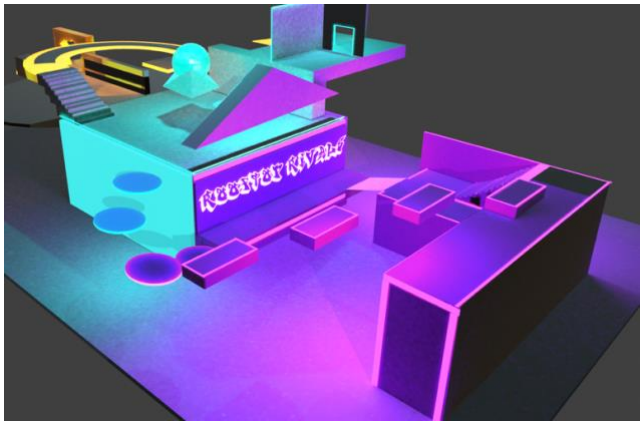




Test Map :



Final Map :



First CDSF : Second CDSF :

Tasks	Soutenance 1	Soutenance 2	Soutenance 3	Tasks	Soutenance 1	Soutenance 2	Soutenance 3
Movements	15 %	50 %	100 %	Movements	15 %	75 %	100 %
Menus	40 %	60 %	100 %	Menus	40 %	80 %	100 %
Multiplayer	80 %	70 %	100 %	Multiplayer	80 %	70 %	100 %
Sounds	40 %	65 %	100 %	Sounds	40 %	65 %	100 %
Map & 3D	15 %	60 %	100 %	Map & 3D	15 %	65 %	100 %
Website	33 %	75 %	100 %	Website	33 %	85 %	100 %
AI	0 %	50 %	100 %	AI	0 %	50 %	100 %
GameMode & HUD	30 %	60 %	100%	GameMode & HUD	30 %	50 %	100%

BIBLIOGRAPHY

https://docs.panda3d.org/1.10/python/index
https://www.w3schools.com/python/
https://www.youtube.com/
https://github.com/
https://stackoverflow.com/questions
https://www.youtube.com/watch?v=_5Js5pbvFSw
https://www.youtube.com/watch?v=1FGWgaCyE8E&list=PLuine2he2FmOY1ILTDc1OR9vHgIMBw4_W

https://www.youtube.com/watch?v=0beimTEHVSU

https://docs.panda3d.org/1.10/python/programming/gui/directgui/directoptionmenu

https://www.youtube.com/watch?v=zeV4_TNUFus

https://www.youtube.com/watch?v=z6SwiNHotLQ

https://www.youtube.com/watch?v=BvIz5XQcOEE

https://www.youtube.com/watch?v=CBJp82tlR3M

https://www.youtube.com/watch?v=aN8n-jyu1fU