ReConnect

Project Report

Louis Harel
Yoann Balasse
Victor Tendron
Adam Franco
Maxime Houwenaghel

A1 EPITA Paris

LYVAM studio ✖ RECONNECT

Monday 26th May, 2025

Table of contents

1 Introduction

The following document is the final report of the ReConnect video game project, led by the LYVAM Studio. The project has been conducted from November 2024 to May 2025. This report contains all the information about the latter project. It includes a presentation of the ReConnect project, a description of the final product, and individual reports. Those are detailing each member's contributions and overall reflections.

1.1 The nature of the project

To begin with, ReConnect is a 3D singleplayer and multiplayer educational video game project. It takes place on an extraterrestrial planet named Edenia. The player is sent to this planet for communication maintenance. Their goal is to fix these communications and repair their ship so that they can return to Earth.

To reach this goal, the player will have to solve electrical puzzles to evolve in the game and progress. The further the player advances in the game, the more difficult the puzzles will be. At the end of the game, the player will have acquired electrical and electronic skills equivalent to those taught in high school.

1.2 Group presentation

Our group name "LYVAM Studio" comes from the names of its collaborators: Louis Harel, Yoann Balasse, Victor Tendron, Adam Franco and Maxime Houwenaghel.
To organize our work efficiently, we began by assigning tasks. Each task had a designated lead responsible for its completion, supported by another group member who could assist when needed.

Task	Responsible	Assistant
Story & Cutscenes	**Yoann**	**Victor**
Menu	**Victor**	**Adam**
HUD	**Adam**	**Victor**
Physics	**Yoann**	**Adam**
Tutorials	**Yoann**	**Adam**
Music & FX	**Victor**	**Yoann**
Level design	**Victor**	**Yoann**
3D modeling	**Louis**	**Maxime**
Interactions	**Louis**	**Victor**
AI	**Maxime**	**Louis**
Multiplayer & networking	**Adam**	**Maxime**
Website	**Maxime**	**Louis**
Communication	**Louis**	**Maxime**

Table 1: Task distribution between the members of the LYVAM Studio

1.3 Reminder of the previous defense

1.3.1 First technical defense

Our first defense took place the Tuesday 14th January, 2025. The features available were:

- A character with a third person camera, movements and animations.
- A multiplayer system with LAN and WAN capabilities and a distant online server.
- A main menu, not yet implemented in the game, with buttons to launch the game in solo, multiplayer, open the settings, and close the game.
- The map design on Blender was still in progress, using Voronoi diagrams for a random generation of the terrain.

The advancement was as follows:

Scenario	70%	55%
Menu	75%	75%
HUD	0%	0%
Physics	30%	30%
First tutorial	0%	100%
Music & FX	0%	100%
First level design	0%	100%
3D modeling	40%	50%
Interactions	0%	0%
AI	0%	0%
Multiplayer & networking	100%	100%
Website	100%	100%
Communication	0%	100%

Table 2: First defense advancement compared to the planed advancement

1.3.2 Second technical defense

Our second defense took place on Tuesday 11th March, 2025. The new features available were:

- A map made with random generation using Voronoi diagrams
- A main menu for launching the game in single-player, multiplayer and opening the settings and a pause menu for quitting the game and opening the settings.
- The electrical simulation system that permits to calculate the intensity and tension received by a target component in a given circuit.

We decided, with the agreement of the jury, to change our Gantt chart to a more realistic one that we would be able to follow. The new Gantt chart of the Reconnect project can be found below.

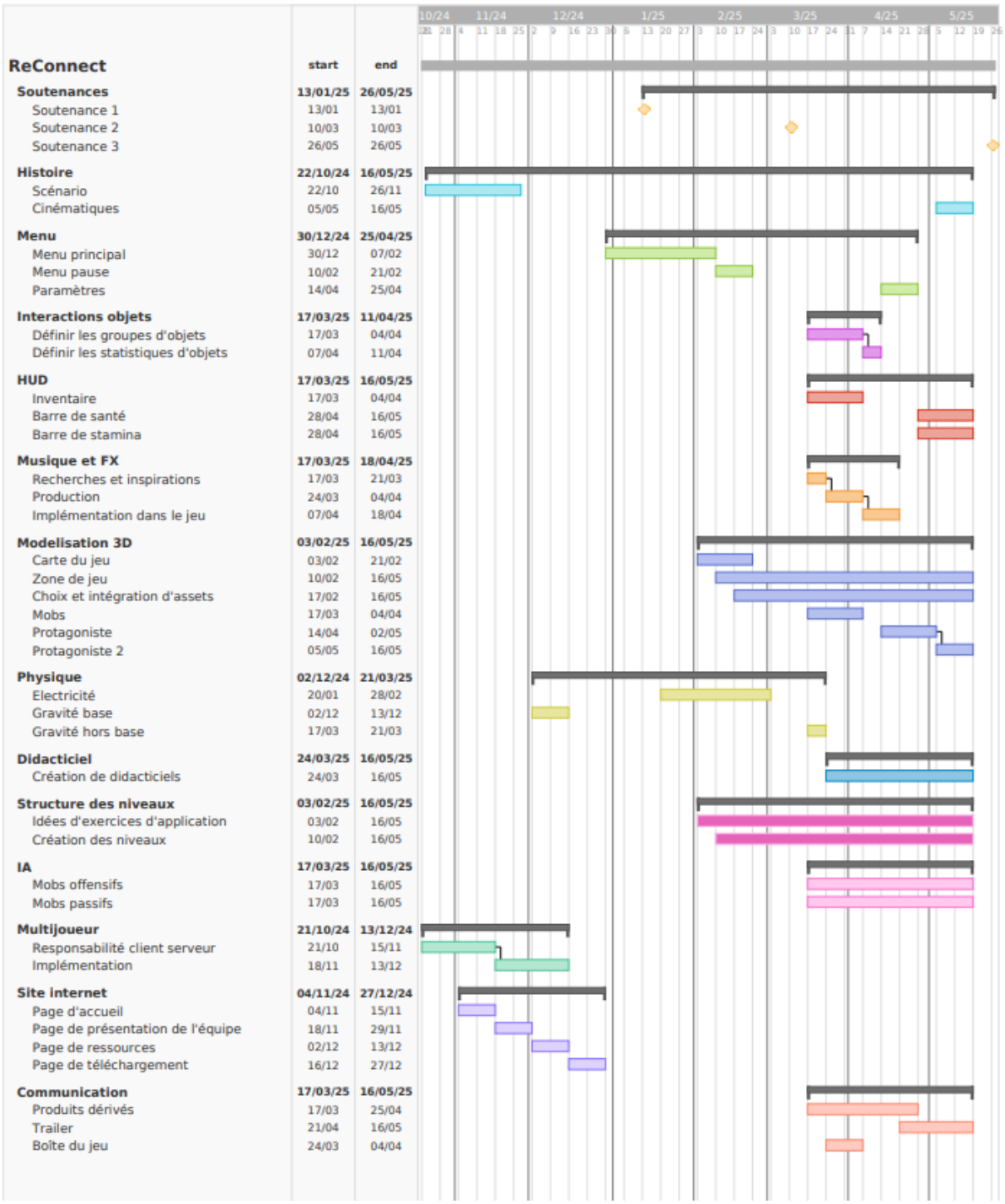


Figure 1: New Gantt Chart of the Reconnect project

The advancement was as follows:

Story & Cutscenes	80%	80%
Menu	75%	75%
HUD	0%	0%
Physics	80%	85%
Tutorials	0%	0%
Music & FX	0%	0%
Level design	30%	30%
3D modeling	40%	50%
Interactions	15%	0%
AI	15%	15%
Multiplayer & networking	100%	100%
Website	100%	100%
Communication	0%	0%

Table 3: Second defense advancement compared to the planed advancement

2 ReConnect Scenario

The game is set in a futuristic, science fiction-like universe. Earth, ravaged by the disastrous consequences of climate change, pollution and excessive resource has become an increasingly hostile environment for human life. Governments and governments and corporations invested massively in space research, giving in to technological solutionism. The aim of this research is to find a new habitable planet for mankind.

After several decades of research, a planet called Edenia was discovered and identified as being planet capable of hosting mankind.

Years after Edenia's discovery, the first colonization missions were launched. A human base is established on the planet to study the feasibility of setting up large-scale human societies. But suddenly all communication between the base on Edenia and Earth is cut off. Faced with this silence, a mission is quickly organized. It is led by a team of engineers, military personnel and scientists. to re-establish stable communications between the colony and Earth.

ReConnect's protagonist is a telecommunications engineer who helped design the communications systems systems used to link Edenia to Earth. He is responsible for leading a team of engineers and technicians to get the interrupted communications working again.

However, the shuttle carrying the team to Edenia crashes on the planet's surface due to violent atmospheric events. atmospheric events. Strangely, none of these weather conditions could have been foreseen. The only survivor of the crash crash, the protagonist must navigate the new world of Edenia alone, in an attempt to re-establish and return to his home planet.

3 Team members reports

3.1 Adam

3.1.1 Networking

The networking architecture of ReConnect is built on the Mirror networking library, a high-level API for Unity that facilitates the development of multiplayer games. This framework allowed us to efficiently implement a client-server model, ensuring synchronization of game states and interactions between multiple players.

The game supports both LAN (Local Area Network) and WAN (Wide Area Network) configurations. This flexibility enables players to connect within the same local network or remotely over the internet, depending on their setup and preferences.

In multiplayer mode, the game distinguishes between three roles:

- Server: Manages the authoritative state of the game world, including player actions and puzzle logic. For deployment purposes, the server can be run in headless mode, allowing it to operate without a graphical interface, which is particularly useful for dedicated servers and optimized hosting.
- Host: Acts simultaneously as both a server and a client. This mode is typically used for testing or for small multiplayer sessions where one of the players also manages the server instance.
- Client: Connects to an existing host or server to participate in the game session. Clients receive updates from the server and send their inputs for validation and execution.

This architecture ensures robust and consistent multiplayer gameplay, while also offering scalability and ease of deployment for different networking contexts.

3.1.2 Menus

A first version of the menus has been implemented by Victor for the first defense of January. However, they caused two issues. The first concerned the design. Victor used a default yellow design that did not match at all the colors of the game that have been defined in the MTE folder. The second was that the implementation of the menus in the game broke the multiplayer mode: we could no longer connect to a server or host one. Then Adam decided to remake them from scratch mainly to be able to test the multiplayer mode again.

3.1.2.1 Canvas and design

Technically speaking, the menus are Canvas objects in Unity. They can contain various user interface elements such as buttons, text or images. Adam created the menu canvas

by simply attaching the desired buttons and text for each one. The game contains 10 menus.

The main menu. It is the first thing that appear at the screen when the game is launched. It displays the name of the game 'ReConnect' in the ReConnect font. From this menu, the user can play in single-player mode, open the multiplayer menu, or open the quit menu.

The multiplayer menu. It is accessible thanks to the 'multiplayer' button in the main menu. From this menu, the user can play in host-mode which starts a client and a server at once. The port of the server can be chosen using the host port text input field. This port will be used by the other clients so that they can connect to the server. The user can also connect to a distant server that can be a dedicated server or a host server. To do so, they have to fill in the server address and server port text input fields. By default, the port is set to 7777 which is the default KCP port. The server address is set to 'reconnect.lyvam.studio' which redirects to a self-hosted ReConnect dedicated server. This also contains a back button to allow the user to go back to the main menu.

The quit menu. This menu can be triggered thanks to the 'quit' button of the main menu. It can also be accessed by pressing the escape key from the main menu. On this menu is displayed the text 'Do you want to quit the game?' and two buttons 'yes' and 'no' that respectively quit the game and go back to the main menu.

The pause menu. This menu is triggered by pressing the escape key while the player is in-game. It allows the user to unlock mouse pointer and quit the game. It contains two buttons: a resume button and a quit button. The resume button simply quits the pause menu to go back to the game, whereas the quit button goes back to the main menu stopping the network client and sever in case of host. It can also be escaped by pressing the escape key in the same way as it has been opened.

The connection menu. This menu is shown after clicking on the a button among 'singleplayer', 'host', and 'join server'. It is a temporary screen menu that is shown when the game is loading.

The connection failed menu. It is displayed when the connection to a remote ReConnect server has failed either because the server responded an error or after 3 seconds without any response. It contains the message 'Connection failed, please try again' and a back button that goes back to the multiplayer menu when it is pressed.

The game also has the 'knock out', 'lesson inventory', 'lesson viewer', 'mission brief',

and 'lesson unlocked' menus that are going to be described later in the report, in Yoann's personal report. There exists a virtual breadboard menu which is considered as a menu in the scripts but is not an actual one. The latter contains the instructions for the puzzles and is going to be discussed later on in section 3.14 Electrical puzzles.

Visuals of all the menus can be found in the appendix.

3.1.2.2 Menu manager script

A script controls these canvas. It is the singleton class MenuManager that inherits MonoBehaviour in order to be attached to a game object in the game scene. The fact that it is a singleton makes the class accessible from anywhere in the project using:

```
MenuManager.Instance
```

This script has as serialized fields all the canvas of the menus enumerated above so that it can easily access them to activate or deactivate them. This `MenuManager` is accompanied by a stack wrapper class called `History`. The general operation of the `MenuManager` script is as follows. There are two main methods: 'SetMenuTo' that takes a menu as argument and 'BackToPreviousMenu' that takes no arument. To change the current menu, the first method needs to be used. This method activates the given menu and deactivates all the others. Also, it pushes the previous menu into the history stack. When the escape key or a 'back' button is pressed, the second method is called. The latter function deactivates the current menu and activates back the one at the top of the history stack. It also pops the stack. This way of implementing the menus allowed the team to have a more generic code with less function, at least in comparison with a first version that consisted in having one method per menu transition.

3.1.3 Object interaction

There are two main scripts responsible for the global operation of the object interaction mechanism. The first one is the 'Interactable' abstract class that inherits from the 'NetworkBehaviour' class. This class defines the two following abstract methods:

```
public abstract void Interact(GameObject player);
public abstract bool CanInteract();
```

The first is directly responsible for the interaction behavior. When a player interacts with a 'Interactable' object, this function is called with the player that has interacted as argument. This function is an action, i.e. it does not return anything but modifies the state of various objects depending on the implementation. The second is, as its name suggests, a method to test if the player can interact with the given 'Interactable' object.

It returns the boolean value ‘true’ if the player can and the boolean value ‘false’ otherwise.

The other main script used in the object interaction operation is `PlayerInteractionDetecor`. This script is attached to the player game object in Unity and allows it to detect the ‘Interactable’ objects in its range. To do so, it uses a sphere collider component in trigger mode. This component calls pre-defined callback functions:

```
public void OnTriggerEnter(Collider other);
public void OnTriggerExit(Collider other);
```

These functions are automatically called by the Unity engine when another collider collides with the player’s sphere collider. The implementations of the latter functions try to get the ‘Interactable’ script attached to the ‘other”s game object. If it succeeds, this interactable is added to the list of ‘Interactable’ objects in the interaction range of the player:

```
private readonly List<(Interactable, Transform)>
_interactableInRange = new();
```

These ‘Interactable’ are coupled with their ‘Transform’ component to keep track of its distance with the player. Indeed when the player press the interaction key, it calls the ‘Interact(GameObject player)’ method of the nearest interactable object. Therefore, the distance of all the interactable objects in the player’s range have to be computed.

In the case where no Interactable has been found attached to the ‘other”s game object, the OnTriggerEnter method simply does nothing.

The `OnTriggerExit` method of the `PlayerInteractionDetector` script works similarly to the previous one. If the object has an Interactable script, then it deletes it from the `_interactableInRange` list.

The `PlayerInteractionDetecor` also manages the play of outlines of the interactable objects. The outline component comes from the Quick Outline Unity package.

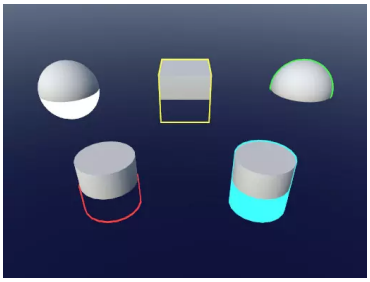


Figure 2: Demonstration of the possible rendering of the outline

When an interactable object enters the player’s range, the `PlayerInteractionDetector`

enables its outline component. This is used to guide the player and indicate him the objects they can interact with. This behaviour does not apply if its `CanInteract` method returns false: indeed this means that the player cannot interact with the object so it is unnecessary to highlight its presence. Also, when the interactable object becomes the nearest that the player can interact with, its outline is set brighter by increasing its width. It allows the player to see what object will be activated if they press the interaction key in the case of multiple interactable objects in their range.

In ReConnect, the Interactable script is used for two main things. The first is the breadboards. The breadboard are the interface where the electrical puzzles take place. The interaction mechanism of the breadboards is going to be described in detail in the following section.

The second use of the Interactable script in the game is the space ship's door which is located inside of the ship. This door is activated by the first level of the game. As soon as it is completed by any player, the door's internal interaction state goes from false to true: players are now able to interact with it. Therefore, the implementation of the `CanInteract` method is as follows:

```
public override bool CanInteract()
{
    return GameManager.Level > 1;
}
```

Where `GameManager.Level` is the level of the player, which starts at one and is incremented by one each time an electrical puzzle is solved.

When a player interacts with the ship's door, its position is set to another position at the surface of the outdoor map, next to the crashed ship's carcass. The player cannot go back inside the ship which is intended because the team wanted the player to explore the map and not to focus on the inside of the ship that moreover does not contain any point of interest other than the first puzzle. For practical reasons, the character controller component has to be disabled to set the position of the player. Thus, it is deactivated and activated back once the teleportation is done.

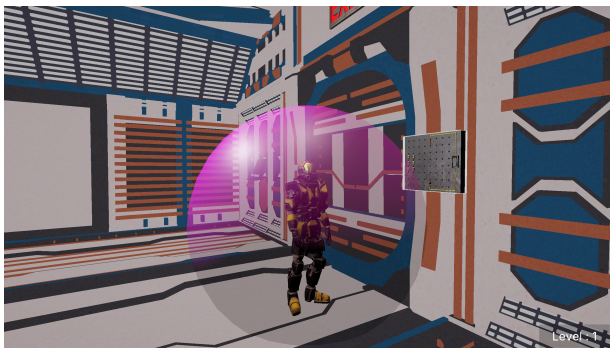


Figure 3: The player with the interaction range shown in pink

3.1.4 Electrical puzzles

3.1.4.1 The breadboard concept

A breadborad, also called a solderless breadboard or protoboard, is a construction base used to build electronic circuits. It is among other things, used in schools to teach electronics because it does not require any welding. It consists of a board with holes wired according to a standard pattern, in which wires or electrical components can be plugged.

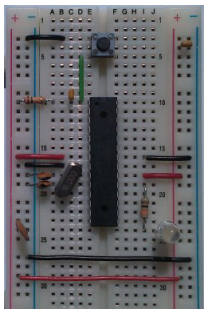
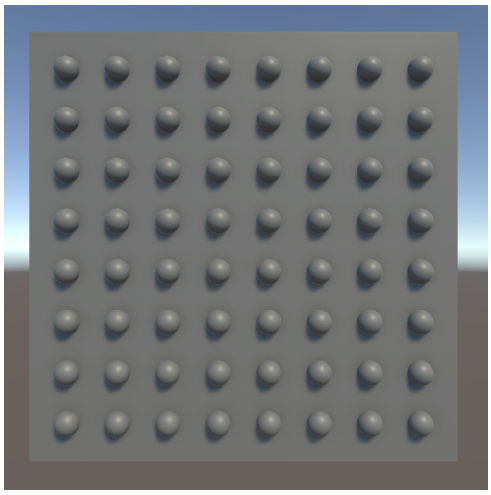


Figure 4: Example of a breadboard

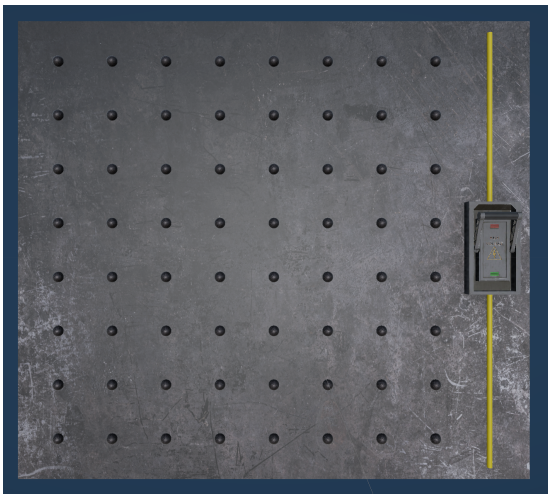
In the case of ReConnect, the breadboard is a crucial component of the game. Indeed, the core of the game consists of electronic puzzles that take place on a virtual breadboard. Therefore, Adam had to implement the easiest to use and most intuitive breadboard so that the player can easily manipulate it and explore the multiple possibilities.

Because of this desire to build the easiest to use and most intuitive breadboard interface in the game, Adam made our breadboard quite different from the real ones. The breadboard of ReConnect is made up of a solid plane on which 64 nodes, physically represented by spheres, are placed in a 8 by 8 grid. These nodes play the roles of multiple holes in the real breadboards: multiple wires, up to 8, and dipoles, up to 4, can be wired to it.

On the breadboard, on the right of the nodes, Yoann added a switch that plays the role of a submit button. This is explained in detail in Yoann's report.



(a) Screenshot of the first breadboard



(b) Screenshot of the final breadboard

Figure 5: Comparison of the initial and final unpopulated breadboards

3.1.4.2 Electrical components

The first thing to implement in addition to the breadboard was the wires. In the Unity project, a wire is a prefab object. This allows any script to instantiate and spawn wires in the scene. This property will be needed especially by the breadboard's script.

To create a wire, the player has to hold the mouse from one node to another. When the mouse is held and collides with a node, the breadboard script creates a new wire if a wire between the previous node and the current one can be created. A wire can be created between two nodes only if they are sibling nodes or diagonal nodes. To destroy a wire, the user can simply click on it.

In addition to the wires, there are two types of resistive dipoles that have been implemented in the game: resistors and lamps. As for the wires, they are implemented using prefabs, one for the resistors and one for the lamps. However, their behavior is slightly different from that of the wire.

First, the dipoles can neither be created nor deleted. They can only be moved and rotated. This implies that all the necessary components for a puzzle have to be placed on the breadboard. The team had planned to implement a player inventory that could contain dipoles. In that case, the components could have been collected on the map and then moved from the inventory to the breadboard. However, because of a lack of time and because it has been considered as non-essential to the game, the inventory feature has been abandoned.

Dipoles can be moved from one position to another with a basic drag-and-drop mechanics. When a dipole is released on the breadboard, it is automatically clipped to the nearest valid position. In the case where the nearest position is either already occupied by a dipole or outside of the breadboard, the dipole's position is set to the previous one. A

valid position for a dipole is between two adjacent nodes, either horizontally or vertically. Also, the position has to be empty: no dipole or wire has to be between these two nodes for the position to be considered valid. When the position where a dipole is released is not valid, this dipole's position is set to the previous position it had before being dragged. In the case where the dipole is dragged and released outside of the breadboard, its position is also set to the previous one.

More concretely, a resistor is a dipole that has a resistance value. In the circuit, it behaves as a real resistor. The lamp also has a resistance value. In addition to the resistor's behavior, a lamp can shine when they receive the right tension. More precisely, they shine when they are set as a circuit target and when the actual tension is equal to the expected tension. See the Loader paragraph or the Electronics subsection from Yoann's report.

(a) A wire (b) A resistor (c) A lamp (off) (d) A lamp (on)

Figure 6: The various components mentioned above

3.1.4.3 Puzzle interface

The breadboard is not the only component that was necessary to make the electrical puzzle interface. The interface of the electrical puzzles is actually composed of a user-interface panel on the left, a breadboard on the right, and a camera.

The panel explains to the player what is the purpose of the exercise. It contains a title and instructions for the level that can be text or images. For instance, in the case of the first level, the player has to reproduce a given circuit diagram on the breadboard in order to repair a door.

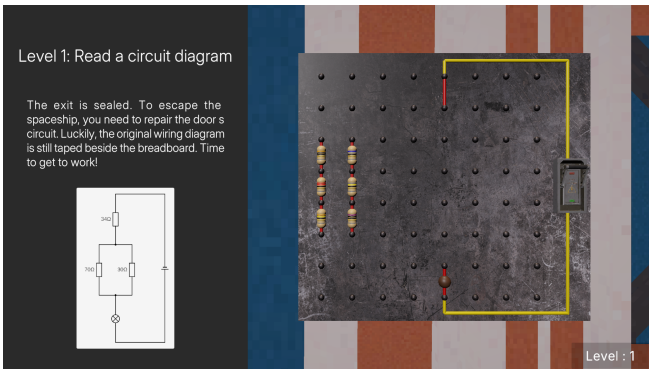


Figure 7: Example of a breadboard

The puzzle interface also contains a breadboard so that the player can build an electrical circuit and try to solve the puzzle. On this breadboard, all the components needed for the puzzle have already been placed. Often, there are more component that the required ones to force the player to think.

Finally, the puzzle interface also consists of a camera. Each electrical puzzle in the game has its own camera so that the player can see the breadboard from closer than they would using the third person camera.

In the game, the breadboards are interactable objects. The condition of the interaction is a comparison with the levels: the player can interact with a breadboard only if their level is equal to that of the breadboard. Indeed, the player starts at level one and therefore can only interact with the breadboard of the first level. This mechanism ensures that the player does the puzzles in the right order. This is important because the levels become harder the more the player advances in the game. Thus, the studio wanted the player to first learn the basics thanks to the first levels before going into harder topics.

```
public override bool CanInteract() => GameManager.Level == level;
```

Where `level` is a serialized field inherited from the `Interactable` class.

```
[SyncVar] public uint level;
```

The `Interact` function is slightly more complex, among other things, because the interaction has to handle the two cases where the player is outside the interface and wants to open it, and where the player is inside the interface and wants to quit it. In both cases, the interact function has to:

1. Toggle the visibility of the breadboard's outline. As with any interactable object, the breadboard has an outline component that allows the player to find it more easily and to know that they can interact with the object. However, when entering the breadboard's interface, seeing the outline is useless and even annoying. That

is why the `Interact` method disables it when the player enters the interface and enables it back when the player quits it.

2. Lock the player movements. The player should not be able to move when they are inside the breadboard interface. Their movement should be locked as long as they are in the interface and then unlocked when the interface is quited.

3. Hide the dummy model. As a remainder, ReConnect is a 3D third person game. Therefore, the player can see themselves. However, when the breadboard interface is open, the player's dummy model can hide parts of the breadboard, making the puzzle impossible to solve. Thus, the dummy model has to be hidden when the interface is opened and shown back when the player closes the interface.

4. Switch camera. As mentioned previously, the breadboard interface uses its own camera, which implies changing the camera. As the game uses the Cinemachine Unity package for the free-look camera, changing the camera programmatically has been achieved using the Cinemachine camera priority setting. The camera of the breadboard is set to 0 by default, whereas the free-look camera has a priority of 1 by default. Therefore, the free-look camera is used by default. When the player interacts with the breadboard, the priority of the breadboard interface integrated camera is set to 2. The camera is then changed with a smooth transition. Conversely, when the player quits the interface, the camera priority setting is set back to 0.

3.1.4.4 Loader

Adam and Yoann, who worked together on the electricity implementation, needed a way to load circuits without needing to place the components by hand in the scene. The first idea was to use a comma-separated value format that contained the type of the component and various other information depending on the component. For a wire, the format was `wire,h1,w1,h2,w2`, where `h1` and `w1` are the coordinates in height and width of the first pole, and where `h2` and `w2` are the coordinates of the second pole. For a resistor, the format was `resistor,h1,w1,h2,w2,resistance`, with `h1`, `w1`, `h2` and `w2` similar to that of the wire, and where `resistance` is the resistance in Ohms of the resistor. Finally, for a lamp the format was `lamp,h1,w1,h2,w2,resistance,nominal-tension` with `h1`, `w1`, `h2`, `w2`, and `resistance` similar to the resistor and where `nominal-tension` is the tension required for the lamp to turn on.

```
lamp,5,3,6,3,250,120
wire,0,3,1,3
wire,1,3,2,3
wire,2,3,3,3
wire,4,3,5,3
```

```
wire,6,3,7,3
resistor,3,5,4,5,250
```

However, as you may notice, this format was very unclear and very hard to both read and write.

Thus, Adam chose to use a yaml format and implemented a `Loader` script that can load a given circuit described in a yaml file on a breadboard in the game.

The first lines of the yaml file are used for the general settings of the circuit. The yaml keys are described below.

The field `title` of type string simply represents the title of the circuit.

The field `input-tension` of type float is the tension of the circuit generator. Indeed the generator is not a component that can be placed on the breadboard but is outside and is represented by the switch.

The field `input-intensity` of type float is the maximal intensity the generator can provide. It is not used in the game.

The field `target-value` of type float is the tension or intensity required at the terminals of the target component for the circuit to be considered to be successful.

The field `target-quantity` of type string indicates whether the tension or intensity, at the terminals of the target component, should be checked.

The field `input-x-pos` of type int is the x component of the position of the input point. As the circuit input is always in the top row, the y component is always equal to 0.

The field `output-x-pos` of type float: The x component of the position of the output point. As the circuit output is always in the bottom row, the y component is always equal to 7.

The field `components` of type sequence is the list of components on the breadboard represented as yaml objects described below.

Concerning the encoding of the components, wires, resistors, and lamps have most of their yaml fields in common. There is only the `resistance` yaml key that is common only to the resistors and the lamps.

The field `name` of type string is the name of the component. It is only used for debugging purposes and cannot be accessed from the game hence it is optional.

The field `x-pos` of type int represents the x coordinate of the first pole. The value must be between 0 and 7 included. A value of 0 means that the component is in the leftmost column of the breadboard.

The field `y-pos` of type int represents the y coordinate of the first pole. The value must be between 0 and 7 included. A value of 0 means that the component is in the top row of the breadboard.

The field `direction` of type string is the cardinal point indicating the direction of the second pole of the component with respect to its first. The value can be among `n`, `s`, `e` and `w` for a resistive dipole (resistor and lamp) and `n`, `s`, `e`, `w`, `ne`, `se`, `nw` and `sw` for wires since they can be placed in diagonal. For instance, a direction set to `e` would mean that the second pole of the dipole is $(x + 1, y)$, where x and y are the coordinate of the first pole.

The field `locked` of type boolean indicates whether the user can change the position of this component or not. It is optional and is set to `false` by default.

The field `target` of type int indicates whether this component is the target component for the circuit. Only one component can have this field set to `true` in a circuit and this component must not be a wire. It is optional and is set to `false` by default.

In addition of these fields, the resistor and the lamp have the field `resistance`. It takes a strictly positive floating point number. It simply corresponds to their resistance in the circuit.

Note the lamp no longer has a field `nominal-tension` in the new circuit format compared to the first comma separated value format because it has been integrated in the circuit settings through `target-value`.

For instance, here are two extracts of the first level encoded circuit.

```
title: Circuit diagram
input-tension: 150
input-intensity: 16
target-value: 40
target-quantity: tension
input-x-pos: 4
output-x-pos: 4
components:
- resistor:
name: R30
x-pos: 0
y-pos: 2
direction: s
resistance: 30
# [...]
- wire:
name: locked wire
x-pos: 4
y-pos: 0
direction: s
locked: true
- lamp:
```

```
name: light
x-pos: 4
y-pos: 6
direction: s
resistance: 20
target: true
locked: true
```

Although this format is longer, it is more readable and easier to write. Moreover, it allows for more customizations thanks to the circuit settings at the first lines, which was impossible with the old comma-separated value format.

3.1.4.5 Level 1, 2 and 4

Adam designed the first, second, and fourth levels of ReConnect. He based the levels on the lessons that Yoan had already created.

For the first level, the lesson topic was how to read a circuit diagram. Therefore, Adam decided to ask the player to draw a given circuit diagram on the breadboard. The circuit is a simple circuit composed of 3 resistors, including two in parallel and one in series. Concerning the scenario, the user-interface panel explains that the door of the ship has been broken due to the crash of the ship. The player has to repair it in order to escape the ship's carcass. Hopefully, the original circuit diagram is still there.

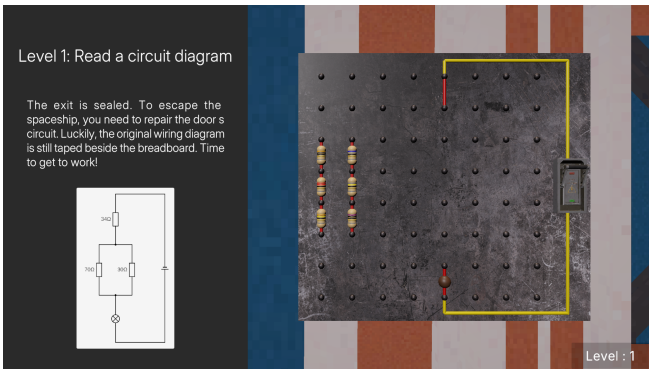


Figure 8: The first level interface

For the second level, the lesson topic was how to calculate equivalent resistance by combining resistors in series and parallel. Therefore, Adam designed a puzzle in which the player must select and arrange the available resistors to achieve a precise total resistance of 62 Ω before closing the circuit. The underlying network consists of a vertical string of four resistors that can be reconfigured to balance series and parallel paths, reinforcing the computational techniques introduced in the previous stage. In terms of narrative, the user-interface panel informs the player that they have just stepped into the pitch-black

laboratory, and restoring the lighting is critical to proceed with their investigation. By matching the exact resistance, the player brings the room back to life and experiences firsthand the practical importance of resistor combination in controlling circuit behavior.

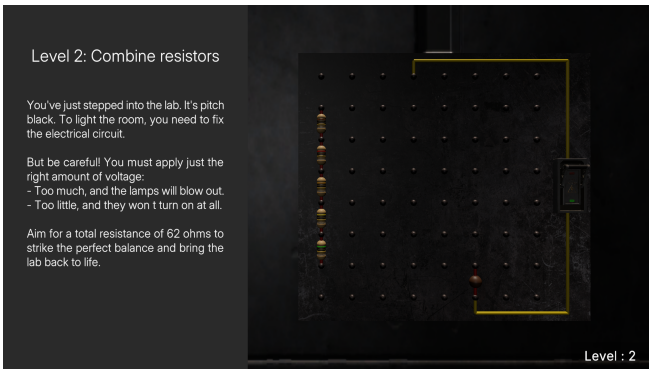


Figure 9: The second level interface

For the fourth level, the lesson topic focused on applying Ohm's law to set a specific current flow within a high-voltage circuit. Accordingly, Adam challenged the player to tune the same resistor assembly so that, when connected to a 200 V supply, exactly 2 A flows through a 30 Ω indicator lamp. This requires the student to calculate and configure the additional series resistance (100 Ω) necessary to protect the lamp while achieving the target current, thereby cementing their understanding of voltage, current, and resistance relationships. Within the game's story, the ventilation system in the crew quarters has been offline for years and must be reactivated to prevent suffocation. When the lamp glows at the correct current, the ventilation kicks back on, providing immediate feedback on the player's mastery of circuit intensity control.

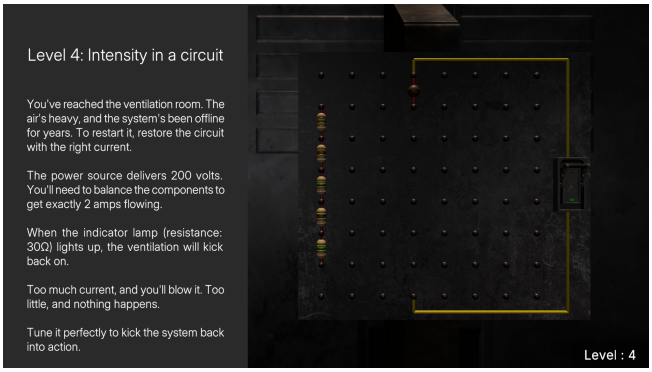


Figure 10: The fourth level interface

3.1.4.6 Wire being created feature

As a remainder, to draw a wire on a breadboard, the player must click on a breadboard node and hold the mouse button until the mouse pointer collides with a second node.

As the team tested the game, they realized that the player could not know clearly if they were tracing a wire or not. Indeed, they could only know that they had correctly clicked on the first node when the mouse encountered the second node. In the case where the first node has not been clicked successfully, the user should go back to the first node and try again. Although the problem seems little, it used to bring confusion and irritation to the player.

Therefore, Yoann and Adam had the idea to display the wire that is being created by the user in real-time. They wanted to see the wire between the first node and the cursor as long as the left mouse button is held. They called it the wire being created feature.

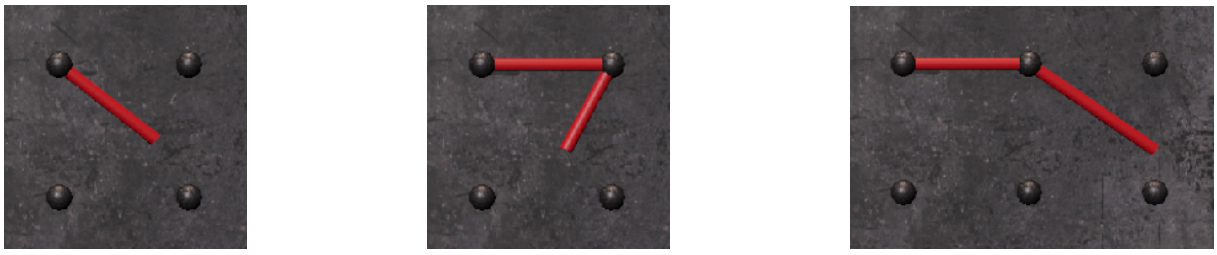


Figure 11: Previews of the wire being created feature

To implement this feature, Adam instantiated a wire prefab in the breadboard script's `OnStartClient` method. This ensures that every client has a wire prefab stored in a variable and avoids the instantiation of an object and the destruction of it each time the player creates a new wire. As this wire prefab is only used for visual purposes, its `WireScript` is disabled.

```
public override void OnStartClient()
{
    // [...]
    _wireBeingCreated = Instantiate(
        Resources.Load<GameObject>(
            "Prefabs/Electronics/Components/WirePrefab"),
        transform.parent,
        false);
    _wireBeingCreated.GetComponent<WireScript>().enabled = false;
    // [...]
}
```

Figure 12: Extract of Breadboard.cs OnStartClient method

Then, its position and rotation are managed in the `Update` function of the same script. This Unity event function is called every frame, which corresponds to the need because

the wire's position and rotation must follow the mouse pointer in real time. The update function can be summarized with the following steps:

1. If the player is not creating a wire, which is indicated by the Boolean state variable `OnWireCreation`, the function hides the wire being created prefab and returns. Otherwise, it executes the next steps.

2. Show the prefab using the `GameObject` method `SetActive`.

3. Then, its position is updated. Its position at any time can be calculated using the average formula: $\frac{firstNodePos+cursorPos}{2}$, where $firstNodePos$ and $cursorPos$ are 3-dimensional positions represented in C# by the `Vector3` class.

4. The wire being created prefab's orientation must also be set accordingly. To do so, Adam used the `LookAt` method of the `Transform` component of the prefab's game object. This method takes a 3-dimensional position, a `Vector3`, and make the given game object point toward this position. Thus, Adam used this function and gave the cursor position as argument.

   ```
   _wireBeingCreated.transform.LookAt(
       breadboardHolder.GetFlattenedCursorPos());
   ```

   Note that `breadboardHolder.GetFlattenedCursorPos()` is used to get the cursor position in world space instead of the screen space. Internally, this method written by Adam makes a ray cast using as direction the pointer, and as plane a virtual plane which has the same equation as the breadboard.

5. The prefab's scale must also be changed. The farther the pointer is from the pervious node, the longer the wire must be. In Unity, the scale is a `Vector3` object, which means that the scale has a `x`, `y`, and `z` component. However, only the length of the wire should be changed because the wire should not become larger as the wire is traced; hence, only the y component should be modified. The new value of the y component of the scale is given by the distance between the ends of the wire divided by two, which in C# is:

   ```
   float y = (_wireBeingCreated.transform.position
       - breadboardHolder.GetFlattenedCursorPos())
       .magnitude / 2;
   ```

6. Finally, the last step is to ensure that the wire is not too long. Indeed, the player should not be able to draw wires between two nodes that are neither adjacent nor diagonal. Therefore, if the length of the wire is greater than 1.8 units, one unit

being the distance between two adjacent nodes, the wire being created is hidden, and the player can no more create wires.

This feature improves the user experience and intuitiveness of the breadboard interface, which is crucial in the game because of its aim to learn by playing.

3.1.5 IA and mobs

Although Maxime did the base of the IA implementation, Adam reviewed his work and improved some parts of the code. The general idea of the mobs' behavior, both passive and aggressive, is as follows:

1. A random destination is chosen.
2. The mob move towards this destination avoiding obstacles thanks to the Unity NavMesh Agent component.
3. Once arrived to destination, the mob waits a random time before starting back at step one.

3.1.5.1 Mob script

As the two mobs of ReConenct, that are the passive jellyfish and the aggressive alien, have the a similar behavior, Adam chose to create a new script `Mob`, from which the two scripts `PassiveMob` and `AggressiveMob` created by Maxime inherit. This class contains the methods responsible for the spawn, the random destination selection and the pause handling of the mobs. It contains abstract methods to allow for more freedom in the implementation of the subclasses `PassiveMob` and `AggressiveMob`.
While creating this new script, Adam also improved the way the mobs chose their random destination. Maxime had implemented a system where two random numbers between -30 and 30 were added to the x and z components respectively. This implied that the range of destination was a square around the mob's current position which is unusual. Also, with bad luck, the mob could get a destination equal to its current position which would result in the mob not moving for a long time. Therefore, Adam changed the way the destination was chosen. His new implementation requires two random numbers: a radius between two given positive numbers and an angle in degrees from 0° to 360°. With these numbers and using the trigonometric function cosine and sine, Adam transformed the square into a annulus, which is a shape that can be described as a 2-dimensional torus or donut. This new method solved the two previously mentioned issues. The maximum distance that can be traveled by the mob is no longer dependent on the direction, which was the case in a square shape. Also, with this ne implementation, the mob cannot choose target that is closer that the minimum traveling distance. This minimum traveling distance corresponds to the lowest bound of the radius random range.

3.1.5.2 Passive mob

For the passive mob, which is a flying jellyfish, in addition to the x and z movements, Adam added a y movement so that the jellyfish goes randomly up and down. The x and z movements are handle by the navmesh agent component. However, it is not possible to use this component for 3-dimensional pathfinding. That's why Adam implemented it himself.
To do so, when the a jellyfish choose a new destination, it chooses also a new height. To get the new value, it adds to its current y position a random number. A maximization system prevents the jellyfish to go lower than the ground. Then, at each frame, the y position of the jellyfish is obtains by linear interpolation. This linear interpolation is based on the remaining distance to the target position in the xz-plane, as calculated by the NavMesh agent. Specifically, the total vertical offset, i.e., the random number added to the current y position, is applied proportionally based on how far the jellyfish still has to travel horizontally. At the moment the jellyfish selects a new destination, the total remaining distance in the xz-plane is stored. As the jellyfish progresses toward the target, its remaining distance decreases. The current y position is then calculated by interpolating between the original y value and the target y value, which includes the random offset, according to how much of the distance remains. For example, if the jellyfish has completed half of its path, the y position will be halfway between the original and the target y value. When the remaining distance reaches zero, which means that the jellyfish has arrived to the randomly chosen destination, the y value will match the target height. This method creates a smooth and continuous vertical transition that matches the horizontal movement, resulting in a floating, gliding motion that feels natural and organic.

3.1.5.3 Networking

The final step of the implementation of the mobs in ReConnect was the networking. To synchronize the mobs across the network using Mirror, each mob prefab is equipped with three core components: Network Identity, Network Transform, and Network Animator. The Network Identity registers the object with the networking system and ensures a unique network-wide ID, while the Network Transform automatically synchronizes its position, rotation, and scale over the wire. The Network Animator mirrors animation states from the server to all clients, so every player sees the same animations at the same time.

Originally, the mob scripts assumed local client authority. Each client managed its own agents' behavior. However, this system is not compatible with the networking. Adam refactored the scripts to server authority, meaning all decision logic including destination selection, height randomness, and state transitions now runs only on the server.

Under this model, the server remains the single source of truth: it calculates positions, synchronizes vertical interpolation, and transmits both transform and animation data to the clients. Clients consume these updates passively, resulting in a more robust system where mobs move and animate identically for every player.

3.1.6 Emotions and experiences

I found this project both engaging and intellectually stimulating. The concept of learning through play is, in my opinion, highly valuable and well-suited to a society increasingly influenced by gamification. Working on the game offered me an insightful opportunity to reflect on how the games I used to play are developed: what aspects are technically challenging and which are more straightforward.
However, I was not entirely satisfied with the overall management of the project. While I frequently collaborated with Yoann, whose assistance was greatly appreciated, I feel that the other team members did not contribute enough or began their work too late in the year. The project caused me significant stress and cost me several nights of sleep. The final presentation period was particularly exhausting.
Despite these difficulties, I remain proud of what we achieved: an educational multiplayer 3D video game that simulates electrical circuits. This accomplishment is especially meaningful given our limited resources: none of us had prior experience with Unity, yet we still managed to deliver a functional and ambitious project.

3.2 Yoann

3.2.1 The player

3.2.1.1 The camera system

For the camera system, we used the Cinemachine package, which provides advanced and highly customizable camera setups. We chose the FreeLook Camera, which allows the camera to orbit around the player based on three separate rigs: Top, Middle, and Bottom. Each rig defines a circular path around the player, where we can adjust both the height and radius to customize the camera's visual behavior and positioning.

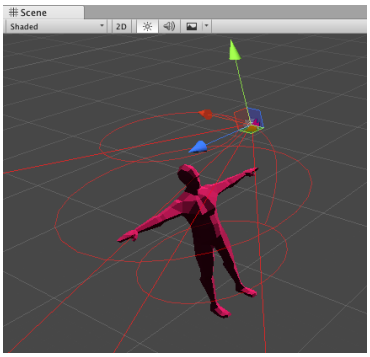


Figure 13: The 3 rigs that define the orbit of the camera

In addition, Yoann added several components to the player's camera to give it a more realistic appearance.
First, the Cinemachine Decollider component to prevent it from clipping through objects. This ensures that the camera pulls back when approaching walls or other environmental elements, avoiding unrealistic visuals like seeing through solid surfaces.
Yoann also added the Cinemachine Deoccluder component to maintain a clear line of sight to the player's character, preventing it from being hidden by obstacles. This ensures better visibility and control for the player.

He spent a considerable amount of time adjusting the camera settings to achieve a smooth and satisfying result.

3.2.1.2 The player movements and animations

The player movement task was not initially included in our Gantt chart or task distribution, but Yoann took the initiative to take on and complete it.

The player can move using the WASD keys (or ZQSD on AZERTY layouts). To translate these inputs into character movement, Unity's Character Controller has been used, which provides smooth and responsive motion while automatically handling collisions with the environment. One particularly useful feature is its ability to move naturally over small obstacles like stairs or steps, without needing to jump. This helps keep movement fluid and realistic, improving the overall feel of navigation in the game.

A jump feature has been integrated, for which Yoann had to simulate gravity manually. The Earth's gravitational constant, $g = -9.81\,\mathrm{m/s^2}$ has been used to create a realistic falling behavior when the player is inside the base or the spaceship.
Additionally, a custom gravity value of $g = -5\,\mathrm{m/s^2}$ has been configured when the player is outside, to simulate a floating or low-gravity environment. This contrast enhances immersion by making movement feel grounded indoors and lighter or more space-like outdoors. When the player is in the air, they are pulled downward over time by applying

the gravity value, scaled by the square of the frame's delta time.
To switch between the different gravity settings, Yoann created an "air-lock" system consisting of 2 planes, one on the inside, the other on the outside. The outdoor gravity is set when the player collides with the outer collider, and the inside gravity is set when the player collides with the inner collider.

Yoann designed the jump to reach a maximum height of 0.7 units, which corresponds to approximately 0.7 m in real life. To calculate the initial velocity needed to reach this height, the following formula has been used :

$$v = \sqrt{-2 \times \mathrm{g} \times \mathrm{h}}$$

Equation 1: Initial jump velocity to reach a given height h

Initially, the player could start sprinting or introduce new speed while in the air. This behavior was not realistic and was caused by improper handling of user input. To fix this issue, Yoann disabled the sprinting and crouching features when the player is not grounded.
Additionally, when the player is in the air, their maneuverability has been reduced to simulate more realistic physics, since in real life, we have limited control over our movement while airborne. To do that, we used Linear Interpolation with the function 'Mathf.Lerp' and a coefficient 'c' equal to 8 by default.

```
float Vx = Mathf.Lerp(
    _currentVelocity.x, desiredVelocity.x, c * Time.deltaTime);
float Vz = Mathf.Lerp(
    _currentVelocity.z, desiredVelocity.z, c * Time.deltaTime);
```

The Lerp factor is:

$$t = 8 * Time.deltaTime$$

Equation 2: Lerp factor depending on the frame rate deltatime

This means that if a player has a frame rate of 60 FPS, this means that the Lerp factor is $t = 8 \times \frac{1}{60} \approx 0.133$ so 13.33 %.

Different movement speeds have also been implemented for the player: a walking speed, a sprinting speed, and a crouching speed. This variety gives the player more freedom and

flexibility in how they control their character.

Finally, to enhance the realism of the gameplay and clearly represent each action to the player, Yoann implemented animations for all the different types of movement. These animations help make the character's actions feel more natural and visually responsive. The jumping animation was particularly challenging, as it had to be broken down into three distinct phases:

1. A jumping phase : the initial upward motion when the player initiates a jump.
2. A falling phase : the descent after reaching the peak of the jump.
3. A landing phase: the moment the player makes contact with the ground.

Note that the falling phase is also triggered when the player drops from a height without jumping, ensuring consistency in how airborne movement is visually represented.

Eventually, Yoann had to manage some conflicts between overlapping movements. For example, if the player is crouching and then tries to jump, the character will first exit the crouching state before jumping.

3.2.1.3 New 3D Model for the player's character

Initially, we used a temporary 3D model for the player's character to visualize the animations of the player while waiting for the final one.

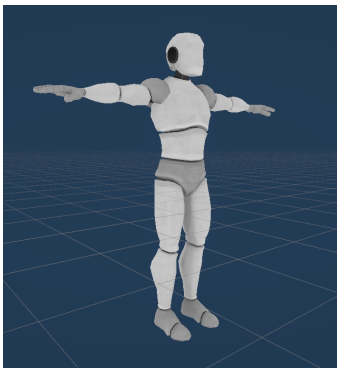


Figure 14: The character 3d model placeholder we imported until ours is ready

Since no custom character was designed, Yoann chose to import an existing model from the internet. The team wanted the player to have a futuristic appearance, equipped with armor to reflect their journey through space and the hazardous conditions on the planet's surface. The environment is not entirely safe, and the crash zone is inhabited by hostile creatures, making protective gear essential for the character.

The “Armored Man - Low Poly” model was chosen for its simplicity, low-poly design, and its perfect alignment with the vision we had for our player character. At the time of download from CGTrader, the model was free to use. However, as of the writing of this report, the original publication has been removed, and the model is now being sold for $5.

Figure 15: The new 3D model we added for the player character

One advantage of using an armored character is that the absence of a visible face allows the character to remain anonymous. This makes it easier for any player, regardless of gender or identity, to project themselves onto the character and feel more immersed in the game.

3.2.1.4 Knock out animation and visuals

The team had planned to create a combat system, but it was not wished to have a death of the player for various reasons. The game aims to be educational, and representing death is clearly not necessary to learn how to build electrical circuits. As the team already stated in the game’s specifications document “The player will have to acknowledge their death through their absence.” concerning the people that were previously on the planet before the arrival of the player.

Yoann proposed the idea of a knockout system instead of a traditional player death. In this approach, when the player is defeated, they are temporarily incapacitated and must wait a set amount of time before continuing. The duration of this wait needs to strike a balance: it should be long enough to make players cautious and respect the danger, but not so long that it disrupts the flow of gameplay or becomes frustrating. The knockout duration has been fixed to 10 seconds.

To represent the knockout state, a dedicated menu was created to inform the player of their condition and explain the reason for their incapacitation. This interface also includes a countdown timer, allowing the player to clearly see how much time remains before they can resume playing. The goal was to keep the player engaged and aware,

while encouraging patience during the knockout period.
The knockout interface functions more as an overlay than a traditional menu, especially when compared to the main menu or the pause menu. Its purpose is to represent the player's loss of consciousness while still allowing them to see what's happening in the game environment. To achieve this, a semi-transparent black background was used, creating a sense of fading out without fully disconnecting the player from the scene. During this time, the player's movement, camera control, and ability to exit the game are all disabled until the countdown ends.

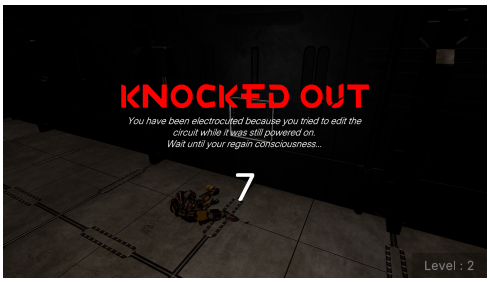


Figure 16: The knockout overlay/menu

Additionally, an animation has been added on the character to illustrate it's loss of consciousness. When the knockout is triggered, the character falls on the ground. When the countdown ends, with another animation, the character stands back up and only once the character finished to stand up, the player regains full control over their character.

3.2.2 Combat system : not integrated

Initially, the player had to fight back the aggressive mobs to survive their attack. Yoann started an implementation of the combat system permitting players to fight entities, making them loose health, and destroying them once they have no more health points.

To simplify the maneuverability of the player fighting mechanic, it was decided to apply damage to all entities within the attack range of the player, instead of targetting players in the looking direction of the character. Indeed, our game is played in 3rd person and it is less precise for fighting than 1st person.

To detect entities entering within the player's range, a Sphere Collider has been used on the player's character in IsTrigger mode. It's radius represents the attack range of the player and a material could be applied for debugging purposes.

The attack system consisted of 2 scripts :

1. **An abstract "Attackable" script :** Defines entities that can be targeted and attacked. It serves as a base class to be extended by specific types of entities, such as mobs or the player.

2. **An "AttackDetector" script :** Assigned to the attacker, this script manages interactions through Unity's collider trigger functions ('OnTriggerEnter' and 'OnTriggerExit'). It detects potential targets within range and handles the logic for initiating an attack.

The "Attackable.cs" script is inspired of the "Interactable.cs" script developped by Adam in the way it is called by the detector script when an attackable enters the player's attack range. The attackable script holds all the health data about the entity

```
public float MaxHealth = 100f;
public float CurrentHealth;
// Health points per second
public float RegenRate = 5f;
// Delay after taking damage before regen starts
public float RegenDelay = 3f;
public bool CanRegenerate;
```

Whenever the player decides to attack an attackable entity, a TakeDamage function is called with the damage that the player inflicts on the entity. This method is responsible for applying the damages to the health of the entity, and deal with a death scenario if it is the case.

A health regeneration system was also implemented, allowing entities to gradually recover health after a certain period without taking damage. This mechanism is checked every frame and becomes active only if the entity's current health is below its maximum and the required delay since the last damage has passed. Once activated, the player regains health at a rate of 5 HP per second. The regeneration is applied continuously, with the amount restored each frame calculated proportionally to the frame's deltaTime, ensuring smooth and consistent healing across different frame rates.

An abstract method named "KnockOut" is defined but not implemented in the "Attackable" script. It represents the action to be executed when an entity's health drops below zero. This method is intended to be implemented in derived classes, allowing for entity-specific behaviors upon defeat. This design enables the "Attackable.cs" script to be used for both players and mobs. In the initial implementation, only the "MobAttackable" subclass was defined. Its "KnockOut" method simply destroyed the mob, and could also be extended to trigger a death animation for the entity.

The "AttackDetector" script relies on the trigger methods of a Sphere Collider attached to the player. When another player enters this range, they are added to a list of targets within reach. Conversely, when they exit the collider, they are removed from the list. To visually indicate that an entity is attackable, an outline is toggled on or off around it.

When the player performs a left-click, the “OnAttack” method is triggered. This method first filters the list of targets to ensure that all references are still valid—removing any that may have been destroyed. It then applies the “TakeDamage” method to all remaining targets within range. The “AttackDetector” also stores the damage value dealt by the player during each attack.

Since the combat system was designed before the mobs were fully implemented, it was not possible for Yoann to develop the player-versus-mob attack mechanics in time. Ultimately, due to the incomplete state of this feature and the limited time remaining before the project deadline, the team decided to abandon the combat system.

3.2.3 Electronics

ReConnect is centered around electrical puzzles that the player must solve. Our goal was to give players full creative freedom in how they build their circuits, as long as the final result is correct. This open-ended design encourages experimentation and multiple solution paths, making the puzzle-solving experience more engaging and personalized. On the backend, this required the development of an algorithm capable of transforming any player-created circuit into a graph structure. From this graph, a series of transformations were applied to simplify the circuit while preserving its accuracy. This simplification made it possible to easily calculate key electrical values such as current (intensity) and voltage (tension) received by the target component. The conversion from the player’s design into an abstract graph was developed by Adam.

3.2.3.1 Electrical graphs

To perform calculations on circuits, it was necessary to represent them abstractly. For this purpose, Yoann decided to use the graph data type.

> *“ In discrete mathematics, particularly in graph theory, a graph is a structure consisting of a set of objects where some pairs of the objects are in some sense ”related”. The objects are represented by abstractions called vertices (also called nodes or points) and each of the related pairs of vertices is called an edge (also called link or line). Typically, a graph is depicted in diagrammatic form as a set of dots or circles for the vertices, joined by lines or curves for the edges. ”*

Figure 17: Definition of a graph (Wikipedia)

A circuit consists of components connected between them. The current flows from the input of the circuit to the output of it. The graph structure if perfect to represent a circuit.

Yoann began by defining different C# classes that will be this data structure.
First, there is the Vertex that will represent a component connected to other. Here is an extract of the properties of this class.

```
public class Vertex
    {
        public List<Vertex> AdjacentComponents { get; }
        public string Name { get; }
    }
```

Figure 18: Extract of the Vertex class

By definition, a Vertex is an object from the graph connected to other objects from the graph. Thus, it has an "AdjacentComponents" list that represents the components to which it is connected.

Then there is the Branch. A branch is the part of the circuit between two nodes that can deliver or absorb energy, excluding short circuits. In our definition of a Branch, it has a list of components and a total resistance corresponding to the resistance of the components it contains.

```
public class Branch
    {
        public List<Vertex> Components;
        public Node StartNode;
        public Node EndNode;
        public double Resistance;
    }
```

Figure 19: Extract of the Branch class

Then there is the circuit graph. It is composed of a list of Vertex that represent the components from the circuit. The input and output of the circuit are also give. It permits to do the traversal from the input to the output and thus simulate the path taken by the current. Finally, there is the Target Component, this is the component that is going to be tested in the given exercise to check if it receives the right amount of current or tension. The Branches list is not defined at first but is populated by the traversal of the graph.

```
public class Graph
    {
        public string Name;
        public List<Vertex> Vertices;
        public List<Branch> Branches;
        public readonly CircuitInput EntryPoint;
```

```
        public readonly ElecComponent Target;
        public readonly CircuitOutput ExitPoint;
    }
```

Figure 20: Extract of the Graph class

The graph traversal is carried out in multiple stages. First, the branches of the circuit are identified and stored in a dedicated list. To detect these branches, it is necessary to enumerate all possible current paths from the circuit's input to its output. This step ensures that only the relevant parts of the circuit, those that will actually be traversed by current, are considered. The paths are discovered using a recursive Depth-First Traversal algorithm.

For each path found, its components are examined sequentially until a node is encountered. A branch is defined as the portion of the circuit between two nodes. Once identified, the branch is created and added to the list. To avoid duplicates, a check is performed before inserting a new branch, as some sections of the circuit may appear in multiple paths. This is especially common since all current paths originate at the EntryPoint and terminate at the ExitPoint, potentially overlapping along the way.

Finally, our goal is the get the global intensity traversing the circuit. For that, some initial values have been set up at the circuit loading. For instance, the Global circuit tension, the Target component of the circuit, its resistance and even the tolerance of the circuit computation. To obtain the global intensity, the association of resistors in parallel and series is used to simplify the circuit over and over until it is only composed of an equivalent resistor and the target component. Then, using the Ohm's law, the global intensity traversing the circuit can be computed.

$$R_{eq} = \sum_{i=1}^{n} R_i = R_1 + R_2 + ... + R_n$$

Equation 3: Association of resistors in series

$$\frac{1}{R_{eq}} = \sum_{i=1}^{n} \frac{1}{R_i} = \frac{1}{R_1} + \frac{1}{R_2} + ... + \frac{1}{R_n}$$

Equation 4: Association of resistors in parallel

The Global Intensity refers to the current passing through the Target component. This component is always placed in series with either the input or the output of the circuit,

$$U = R \times I \iff I = \frac{U}{R}$$

$$\text{With:} \begin{cases} \text{U} & \text{the tension in Volts} \\ \text{R} & \text{the resistance in Ohms} \\ \text{I} & \text{the intensity in Amperes} \end{cases}$$

Equation 5: Ohm's law

ensuring that all current flows through it. This design constraint was intentionally imposed due to the complexity of supporting arbitrary placements of the target within the circuit. By limiting its position, the system becomes significantly easier to manage and analyze.

As a final step, the current received by the Target component is compared to the expected value defined in the exercise specifications. A tolerance margin, also specified in the same document, is used to determine whether the result falls within an acceptable range. If the check is successful, the exercise is considered complete, and the Target component triggers its "DoAction" method. This method is specific to each type of component and defines the behavior that should occur when current flows through it. For example, a lamp will turn on once the circuit has been correctly wired.

```
public class Lamp : Resistor
    {
        public LightBulb LightBulb { get; set; }

        public override void DoAction()
        {
            LightBulb.Set(true);
        }

        public override void UndoAction()
        {
            LightBulb.Set(false);
        }
    }
```

Figure 21: Extract of the Lamp class with its DoAction method

3.2.3.2 Short circuit and bypass handling

To have a more realistic electronic simulation, the short circuits had to be handled. There are 2 types of short circuits.

- **The bypass :** by placing a wire in parallel of a component, the theoretic resistance of the wire is null, thus the current will flow entirely though the wire and effectively bypass the component.
- **The short circuit :** by making a zero-resistance path between the input and the output of the circuit. This causes a large amount of current to flow through the circuit, leading to overheating, fire departure among other threats.

To handle the bypass, Yoann added a condition in the part of the algorithm dedicated to associated the branches in parallel to detect this case. When any of the branches in parallel has a resistance of 0, the whole group is merged into a single resistor with a null resistance. This ignores then the other components and bypasses them.

Concerning the short-circuit scenario affecting the entire circuit, an additional condition was implemented at the end of the circuit transformation process, just before calculating the global intensity. If the total resistance of the circuit is found to be zero, this indicates a short circuit. In this special case, a Knock Out is triggered for the player as a safety mechanism, preventing them from continuing after having performed a potentially dangerous action.

This is a purely educational decision, as Yoann wanted the player to become aware of how dangerous such a mistake could be in real life. By triggering a knockout in response to a short circuit, the game reinforces the importance of safe circuit design through gameplay consequences.

3.2.3.3 Add a switch and animation to the breadboard

Yoann and Adam decided to trigger the computation of the circuit with a kind of submit button, instead of computing it every frame as it is a really heavy procedure. In the first demo version, a UI “Submit” button was used.

Finally, Yoann added an electrical switch located to the right of the breadboard. The player must click on this switch to validate their attempt at solving the exercise. Upon activation, a custom animation designed by Yoann is triggered, causing the lever to move downward. When the lever reaches the down position, the circuit traversal and evaluation are performed. Based on the outcome, the switch will either remain down, if the circuit is correct, or return to its original position if the attempt fails.

The 3D model used for the switch, titled “Dirty SciFi Electrical Lever Switch” was sourced from Fab.com and is licensed under the Fab Standard License.

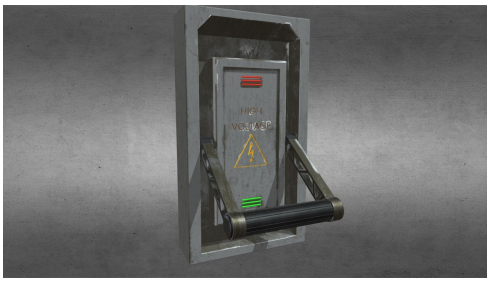


Figure 22: Model of the switch chosen to be integrated into the breadboard

Additionally, the switch is connected to the circuit’s input and output using yellow wires. Since these input and output points can be positioned anywhere along the first and last rows of the breadboard, a custom script was developed to dynamically adjust the position of the switch wires based on their actual locations in the circuit. This task was implemented by Yoann.

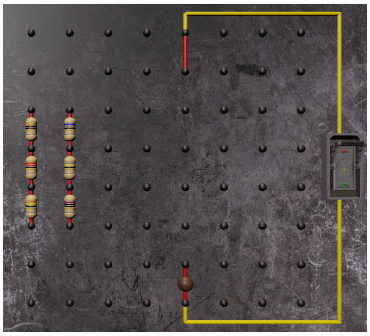


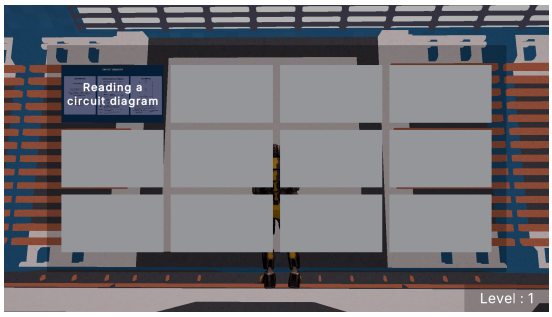
Figure 23: The switch integrated in the breadboard with the yellow wires

3.2.4 Menus

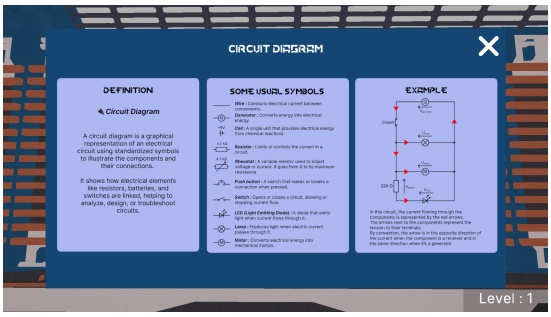
3.2.4.1 Lesson inventory

Some menus needed to be designed to display the educational content that Yoann created. To facilitate testing and integration with his lessons, Yoann decided to develop the necessary in-game menus himself.

This menu functions as a kind of inventory or digital notebook, allowing the player to access lessons at any time during gameplay. It displays a grid of cells: gray cells indicate empty slots, while filled cells show a thumbnail and title representing a lesson. Players can click on a lesson to open and view it in a nearly full-screen format, providing an accessible and immersive learning experience.



(a) The lesson inventory menu



(b) Lesson viewer

Figure 24: Lesson inventory menus

These two menus are built upon the menu system designed by Adam. The lesson menu can be opened by pressing the L key (for "Lesson") and closed either by pressing L again or by using the ESC key. In the lesson viewer, a close button (a cross icon in the top-right corner) is also available for exiting the view, in addition to the ESC key.

To place the new lesson into the inventory, a "LessonInventoryController" script has been created. It contains a list of slots and a method "AddItem".

```
public void AddItem(string itemName, Sprite sprite)
```

This method enumerates sequentially all the slots until a free one is found and then gives the sprite and then name to the slot that will handle the display of these informations.

3.2.4.2 New lesson pop-up

When the player succeeds a level, it gains a level point. On each level passed, a new lesson is given to the player and the next exercise will be about this lesson and possibly the previous ones. Yoann designed this "pop-up" menu to inform the player that a new lesson is available in their library while reminding how to access the lesson inventory.

When the level changes, the "GameManager" script loads the new lesson menu by calling a method of the "MenuManager" script. The "GameManager" script contains a list of lessons organized in the order of their corresponding levels that can be give to the menu method so that it can load the right image.

```
private static readonly List<Sprite> LessonsByLevel
```

This menu is called by the "MenuManager" script using the method "SetMenuToNew-Lesson".

```
public void SetMenuToNewLesson(uint level, Sprite lesson)
    {
```

```
        LockPlayer();
        newLessonController.LoadImage(lesson);
        newLessonController.SetTextToLevel(level);
        FreeLookCamera.InputAxisController.enabled = false;
        SetMenuTo(MenuState.NewLesson, CursorState.Shown);
    }
```

This methods sets the image and the level and sends it to the NewLessonController that will display the menu accordingly. It also handles the locking of the player movements and camera while the menu is active.

3.2.4.3 Missions briefs : visual

Another menu that had to be implemented is the mission brief, which is displayed whenever the player levels up. It presents the objectives and lore for the next mission. This menu, along with the entire level progression system, was designed by Yoann.

The mission brief functions as a pop-up window, similar to the lesson menu, and is divided into four distinct sections:

1. **Header :** This section displays the title of the mission and a short summary sentence that quickly outlines the task at hand.
2. **Illustration :** Positioned on the left side, an image visually represents the mission's target, helping the player understand the goal through a quick visual insight.
3. **Description :** On the right side, the mission is explained in more detail. This section includes lore elements, context for the mission, and a clear explanation of what the player is expected to do.
4. **Footer :** The footer explains how to reopen the mission brief later in the game. Players can view it again at any time by pressing the "M" key (for "Mission").

Figure 25: Mission brief pop-up for the first level

It's important to distinguish the mission brief from the circuit instructions, which are focused on how to complete the specific circuit puzzle. The mission brief provides overall context and objectives—for example, a mission could be to restore the lighting system, while the circuit instructions describe how to wire the components correctly to achieve that result.

3.2.5 Tutorials and level design

3.2.5.1 Electronic courses

Our game aims to teach players through interactive gameplay. To achieve this purpose, it was necessary to create some teaching materials for the player. Yoann was responsible for this task and he created 5 lessons.

1. Lesson 1 : Reading a circuit diagram (deifnition, symbols, ...)
2. Lesson 2 : The resistance in a circuit (definition, ohm's law, association in parallel and series, ...)
3. Lesson 3 : The tension in a circuit (definition, loops law, series, parallels, ...)
4. Lesson 4 : The intensity in a circuit (definition, node's law, ...)
5. Lesson 5 : The resistance color code (definition, band layouts, color code values, ...)

For each lesson, there is a "Definition" part where the notion is defined, and its purpose explained. For the resistance, tension and intensity, an analogy to water is used to help the player understand what these physical quantities.

These teaching materials are condensed, and only give the necessary tools to the player to solve the exercises that we give to them. Further help may be needed for people that have difficulties to understand the notions taught. Our game is clearly not meant to replace teachers, it is just an alternative way of introducing electricity to children in schools.

The lessons designed can be found in Appendix A.2

3.2.5.2 Circuit instructions : levels 3 and 5

Yoann designed the circuit and instructions for the levels 3 and 5, respectively about tension in a circuit and the resistance color code.

For the circuit of the level 3, Yoann decided to ask the player to give exactly 62.4 volts to the lamp in the circuit in order to open a door. In order to do so, the player has 4 resistors available on the breadboard and they need to do some calculations by

associating them in series, or parallel to have an equivalent resistance and finally compute the tension received by the lamp.

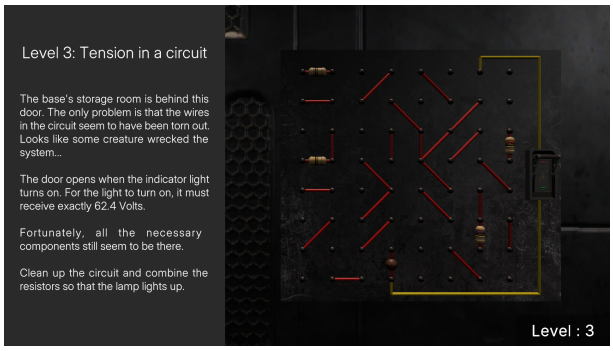


Figure 26: Level 3 : breadboard and instructions

For the circuit of the level 5, Yoann gave a circuit diagram with some resistors and their values, however, the tooltip permitting to know what is the value of a resistor on the breadboard has been deactivated for this level only. This forces the player to decode the value expected into the resistance color code and find the right resistors to complete the circuit.

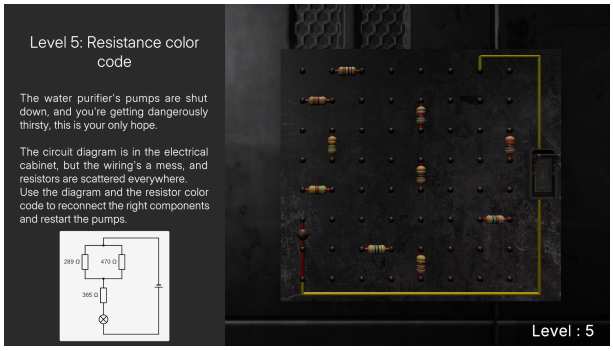


Figure 27: Level 5 : breadboard and instructions

3.2.5.3 Missions briefs : content

In the game, mission briefs serve as narrative and instructional guides that introduce each new level. These briefings are designed to blend immersive storytelling with clear gameplay objectives. As the player progresses, they are gradually introduced to more complex systems and environments, each one requiring problem-solving and electrical circuit-building skills. Yoann designed and implemented these mission briefs as part of the level system, ensuring each one fits naturally within the overarching story while teaching essential gameplay mechanics.

The first mission begins with urgency and danger: the player's ship has just crash-landed on the hostile planet Edenia. With rising temperatures threatening their survival, the immediate goal is to escape the wreckage. However, the ship's electrical systems have failed during the crash. The player is tasked with finding the power panel that controls the main door and restoring power to it. This introduces the player to the fundamental circuit mechanics and sets the tone for the survival-oriented gameplay.

In the second level, the player has successfully exited the ship but now finds themselves in a dark and unfamiliar base. The lighting system is offline, and visibility is extremely limited. The mission brief instructs the player to locate and repair the power panel for the base's lights. This challenge not only escalates the gameplay by introducing a new environment but also reinforces the importance of understanding circuit connections and troubleshooting errors in a realistic setting.

By the third mission, the player's basic needs come into play with hunger. The only way to access the base's food supplies is through the storage room, whose entry is blocked due to a destroyed power circuit. The player must reconstruct this circuit to open the door.

The fourth mission adds a layer of discomfort and urgency with a ventilation failure. The air inside the base is stale and putrid, indicating that the fans have not been operational for a long time. The mission brief directs the player to locate the fans on the upper level and repair their circuits. This introduces vertical exploration and pushes the player to apply their circuit-building skills in a more spatially complex environment, while reinforcing the survival theme of the game.

Finally, the fifth mission brings attention to hydration. The barren environment of Edenia leaves the player at risk of dehydration. The mission involves restarting two massive water pumps, part of a larger water purification system, to provide clean drinking water. This task challenges the player to apply their knowledge of the resistor color code to identify the correct resistors needed to replicate the circuit shown in the provided diagram. By interpreting the color bands, the player can determine the resistance values and select the appropriate components, reinforcing their understanding of a fundamental concept in electronics.

Overall, these mission briefs are more than just story devices. They are core gameplay elements that guide progression, teach the player electrical principles, and immerse them in a sci-fi narrative of isolation and resilience. Each one was carefully designed to scale in difficulty and deepen the player's engagement, both emotionally and intellectually.

3.2.6 Resistors 3D modeling and resistor color code

The team aimed to create a realistic game capable of simulating circuit design, much like it would be done in real life. To achieve this, it was important to represent real components, such as resistors, and teach players how they work. Typically, students focus mostly on theoretical calculations without seeing the physical components involved.

To address this, a realistic-looking 3D resistor model was needed, along with a visual explanation of how the resistor color code works. Yoann handled both the visual design and the coding of the color code system.

The 3D model was made using two spheres for the rounded ends and a connecting cylinder as the body. The cylinder was slightly thinner to improve the resemblance to a real resistor.

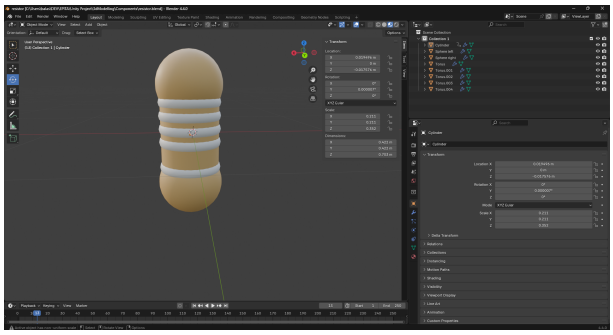


Figure 28: The resistor designed in Blender

The resistor designed is a 5-band resistor. This means it has 3 digit bands, 1 multiplyer band and 1 tolerance band. Each band is colored using the color corresponding to the value that must be written. This table looks like this :

Color name	Color	Digit	Multiplier	Tolerance
Silver		–	$\times 10^{-2}$	±10%
Gold		–	$\times 10^{-1}$	±5%
Black		0	$\times 10^{0}$	–
Brown		1	$\times 10^{1}$	±1%
Red		2	$\times 10^{2}$	±2%
Orange		3	$\times 10^{3}$	±0.05%
Yellow		4	$\times 10^{4}$	±0.02%
Green		5	$\times 10^{5}$	±0.5%
Blue		6	$\times 10^{6}$	±0.25%
Violet		7	$\times 10^{7}$	±0.1%
Grey		8	$\times 10^{8}$	±0.01%
White		9	$\times 10^{9}$	–

Figure 29: The resistor color code : association between color and values

The color bands are dynamically applied at runtime in Unity based on the resistance value assigned to the resistor when the circuit is loaded. First, the three most significant digits are determined, along with the appropriate power-of-ten multiplier. The tolerance value is predefined during the circuit loading process. Corresponding colors are then applied to each band. Whenever the resistor's value changes, the script automatically updates the bands to reflect the new colors.

The OnValueChange hook was useful to sync these colors over the multiplayer with Mirror's "[SyncVar]" since the loading process is done server-side but the color code has to be executed client-side.

3.2.7 Multiplayer

3.2.7.1 Synchronize of the breadboards

At first, Adam and Yoann prioritized creating a fully functional breadboard without considering multiplayer compatibility. It was only after the system was working correctly in singleplayer that the necessary adaptations for multiplayer use were implemented by Yoann. This way of doing was clearly not the best and creating the breadboard with the multiplayer compatibility in mind would have helped reduce the time spent debugging, adapting code, rewriting everything to handle Network authority, non-serializable objects, etc.

First, Yoann added the necessary Mirror components to synchronize the objects across the network. This included the "NetworkIdentity "and "NetworkTransform" components, which ensure that the GameObject's transform is consistently synchronized between all clients connected to the same server.

The loading of the circuit is done by the server in the method “OnStartServer”. Since it is executed by the server, it can directly spawn prefabs that will be then spawned in the client’s instances.

The first issue arose when it became necessary to set the “inner” parameter of a component, which corresponds to its representation within the circuit graph. This required synchronizing custom C# classes using “[SyncVar]”, a feature that Mirror does not support. To resolve this, a unique identifier (UID) dictionary was implemented server-side. Its purpose is to store non-serializable data on the server, allowing clients to request access or perform actions involving that data via server commands. As a result, custom data structures were replaced with UIDs integers that Mirror can serialize and transmit across the network.

```
public static class UidDictionary
    {
        private static readonly Dictionary<Uid, object>
            Dictionary = new();
        private static uint _nextId = 0;

        public static Uid Add(object item)
        {
            AssertIsServer();

            Uid id = new Uid(_nextId++);
            Dictionary[id] = item;
            return id;
        }
    }
```

Figure 30: Extract of the UidDictionary class with the Add method

Visual changes are transmitted from the server to the clients using [ClientRpc] and [TargetRpc] methods. For example, when a circuit is correctly wired, the target component must execute its DoAction method. To ensure this is visible to all players, a [ClientRpc] is used to trigger the DoAction on every client. This allows everyone to see, for instance, a light turning on when the circuit is activated.

To comply with Mirror’s authority system, only the player prefab is allowed to send commands to the server. As a result, most of the networking functionality had to be implemented within the PlayerNetwork script, which is attached to the player prefab.

Additionally, ReConnect is designed as a cooperative game, making simultaneous interaction with the breadboard essential. Yoann made this possible by enabling all players to collaboratively edit the breadboard in real-time, allowing them to draw and remove wires, move and rotate dipoles, and even submit the circuit together.

Overall, this was a particularly demanding task that required extensive rewriting and restructuring of the code to adapt to Mirror's networking model. Yoann devoted a substantial amount of time debugging the features across singleplayer, host, and client modes, with valuable assistance from Adam.

3.2.7.2 Level system and multiplayer syncing

The level system was developed by Yoann. To store the level data, a singleton script called `GameManager` was created exclusively on the server side. The current level is synchronized using a `[SyncVar]`, allowing the value to be shared from the server to all connected clients.

When a player completes a circuit, the validation is processed server-side, and the level is then incremented on the server. This ensures that all players advance together in the cooperative environment.

To facilitate testing and development, cheat codes were also implemented to manually trigger level progression without the need to complete the corresponding circuits. This allowed for quicker validation of level-dependent features during development.

Each breadboard in the game is also associated with a specific level. This level defines the exact player level required to interact with it. As a result, players can only manipulate breadboards that correspond to their current level, ensuring progression happens in the intended order and maintaining the game's pedagogical structure.

Upon level-up, a `[ClientRpc]` is used to notify all clients of the new level. This triggers the display of the corresponding UI elements, including the "new level" notification and the mission brief, across all connected players.

In addition, each level has a dedicated function that is invoked upon progression. These functions handle the specific in-game consequences of mission completion. For example, when the lighting system mission is completed, a function is triggered to activate a predefined list of lamp components in the environment, visually representing the player's success and altering the game world accordingly.

3.2.8 Trailer

To promote the game on the official website, a trailer was created by Yoann to highlight the game's core mechanics, visual identity, and cooperative gameplay. The aim was to produce a short, engaging video that could serve as an effective introduction to ReConnect.

Cinematic sequences were recorded within the Unity editor using Unity's Timeline feature. Yoann animated the camera through keyframed transforms to create smooth, controlled shots that emphasized key elements of the gameplay and environment. Each sequence was crafted to present the different features of the game clearly and aesthetically.

Once the cinematic scenes were completed, they were assembled in sequence to form a cohesive narrative. The Unity Recorder package was then used to capture these animations at runtime, allowing for high-quality video output.

The final editing was done in Adobe Premiere Pro, where Yoann selected the best takes, adjusted timing, added transitions, and synchronized the visuals with background music and texts. This resulted in a polished promotional trailer that effectively presents the gameplay experience and is now ready to be showcased on the game's website.

3.2.9 Emotions and experiences

I genuinely enjoyed working on this project, as the game concept was both original and educational, offering a new and engaging way to learn. I'm particularly proud of having developed a system that can simulate electrical circuits and programmatically calculate voltage and current in user-created circuits. While the implementation isn't fully universal, it remains flexible enough to support a wide range of use cases.

That said, the biggest challenge throughout the project was the team organization. I believe the final result could have been significantly better if everyone had contributed consistently and met the established deadlines. The lack of regular collaboration impacted the overall quality and progress.

Despite these difficulties, this project was a valuable learning experience for me. It was one of the first times I had to work in a team on such a complex and long-term task. The challenges that came with teamwork and deadlines gave me a realistic glimpse into what it might be like to work in a professional environment.

I dedicate this report to all the hours of sleep I sacrificed to finish the project on time.

3.3 Maxime

3.3.1 AI

In unity, Pathfinding can be done using a Unity component called "navigation mesh" abbreviated in "Navmesh". Navmesh is a data structure used to represent the navigable areas of a 3D environment.

3.3.1.1 Pathfinding of passive entity

At first, the implementation of pathfinding was relatively simple and rudimentary. The entity's movement was based on randomly selecting two numbers within a predefined range, which were then used as the next target coordinates. This method allowed the entity to move around the environment in an unpredictable manner, giving the illusion of autonomy. However, it lacked any real logic or awareness of obstacles, terrain, or goals. It served primarily as a placeholder to visualize movement and test basic mechanics, rather than offering any meaningful navigation strategy.

```
void Update()
{
    if (!_agent.hasPath || _agent.remainingDistance <= 3)
    {
        _agent.SetDestination(transform.position + new Vector3(Random.Range(-30, 30), 0, Random.Range(-30, 30)));
    }
}
```

Figure 31: Example of the first pathfinding implemented

After adapting the code for the network developed by Adam, and collaborating with Yoann to improve the realism of player movement, we made significant enhancements to the behavior of passive entities. A downtime system was introduced to simulate delays between actions, making the entities appear more natural and less robotic. Additionally, the codebase was enriched with new logic and conditions that allowed these entities to behave more like realistic non-player characters (NPCs). They no longer simply wandered randomly; instead, their actions followed coherent patterns, contributing to a more immersive and believable game environment.

```
Frequently called  0+4 usages  lomination +1
public override void OnStartServer()
{
    base.OnStartServer();

    MinMovementRadius = minMovementRadius;
    MaxMovementRadius = maxMovementRadius;

    _previousY = model.transform.position.y;

    _hasBeenInitialized = true;
}

Event function  lomination +2
private void Update()
{
    if (!isServer || !_hasBeenInitialized) return;

    // make the model spin
    model.transform.localEulerAngles +=
        20 * Random.Range(0.5f, 1.5f) * Time.deltaTime * Vector3.up;

    if (IsWaiting) return;

    if (HasArrived)
    {
        StartCoroutine(routine: PauseForSeconds(Random.Range(minPauseTime, maxPauseTime)));
    }
    else
    {
        // move the model on y-axis
        model.transform.position = new Vector3(
            model.transform.position.x,
            _previousY + (_targetY - _previousY) *
            (1 - Vector3.Distance(Agent.destination, model.transform.position) / _targetDistance),
            model.transform.position.z);
    }
}
```

Figure 32: Extract from code of passive entity

3.3.1.2 Pathfinding of aggressive entity

The pathfinding of the aggressive entity is similar to the passive one. It uses the first pathfinding and an algorithm to detect the presence of players around the entity.

```
Event function  lomination +1
private void Update()
{
    if (!isServer) return;

    var closestPlayerTransform = GetClosestPlayerTransform();
    if (closestPlayerTransform is not null)
    {
        StopCoroutine(nameof(PauseForSeconds));
        IsWaiting = false;

        Agent.speed = runningSpeed;
        Agent.SetDestination(closestPlayerTransform.position);
    }
    else
    {
        Agent.speed = walkingSpeed;
        if (HasArrived && !IsWaiting)
        {
            StartCoroutine(routine: PauseForSeconds(Random.Range(minPauseTime, maxPauseTime)));
        }
    }

    _animator.SetBool(_isRunningHash, Agent.velocity.magnitude >= 1f);
}
```

Figure 33: Extract from code of aggressive entity

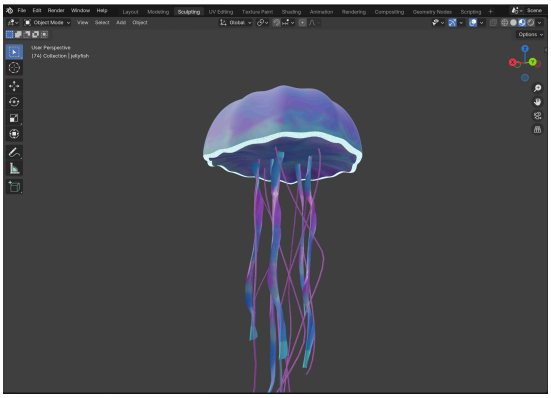
3.3.2 3D modeling

Living beings are evolving on Edenia. It has been decided to simplify wildlife to two distinct species : Medusa and alien. The first entity being a passive entity. The second entity being an aggressive entity against the player.

3.3.2.1 Medusa

The medusa is a passive entity that evolves in Edenia. The 3D model was find on the website "sketchfab.com" and was made by *Solcito Kawaii 3D*. To animate the 3D model, the bone and animation used come from Simple Jellyfish made by *RickStikkelorum* on the website "fab.com"

(a) View of the medusa from the game



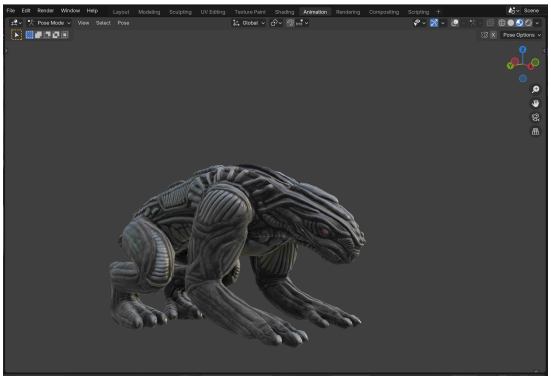
(b) View of the alien from blender

Figure 34: the Medusa

3.3.2.2 Alien

The alien is an aggressive entity that also inhabits the world of Edenia. Its 3D model was found on the website cgtrader.com and was created by *3DHaupt*. The model came fully rigged, with a bone structure and pre-made animations specifically designed for humanoid characters. This greatly facilitated its integration into the game, as we were able to directly import the model into Unity and use its animations within the Animator Controller. The existing rig also allowed us to add custom animations or blend existing ones to better match the alien's behavior and interactions in the game environment.

(a) View of the alien from the game



(b) View of the alien from blender

Figure 35: the Alien

3.3.2.3 Animation

To implement the animations of both types of entities in Unity, we used the Animator component. An Animator Controller was created for each species of entity, allowing us to manage their specific animation states. Within each controller, we structured the different animations in a state machine, defining transitions based on parameters such as movement speed, action triggers, or interaction events. These parameters and conditions determine when and how an entity switches from one animation to another—for example, from idle to walking, or from walking to performing a specific action—thus bringing the characters to life in a dynamic and responsive way.

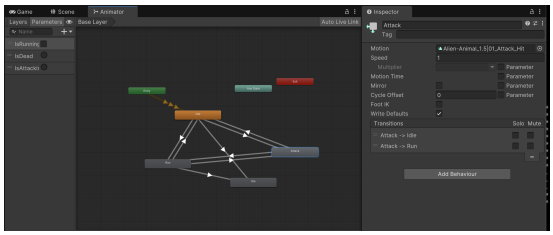
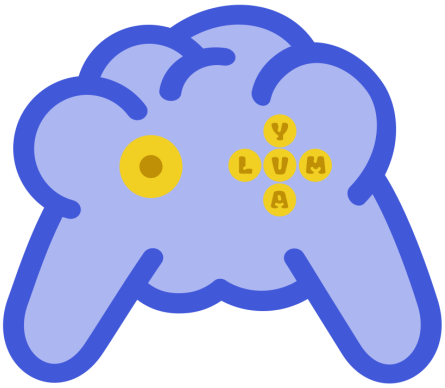


Figure 36: View of the alien from blender

3.3.3 Communication

3.3.3.1 Branding design

(a) Typographic logo



(b) Icon

Figure 37: LYVAM Studio logos

Both LYVAM logos were handmade on Affinity Designer. The first one (figure a) can be used in every context. Because it is a typographic logo, it is easy to understand the name of the brand without knowing the studio. However, the second logo (figure b) should be used with the figure a to emphasize the understanding of the brand.

3.3.3.2 Website

Today, a website is a key element in the communication and presentation of a video game. The website has undergone several updates to follow the evolution of the video game and the artistic direction of the project.

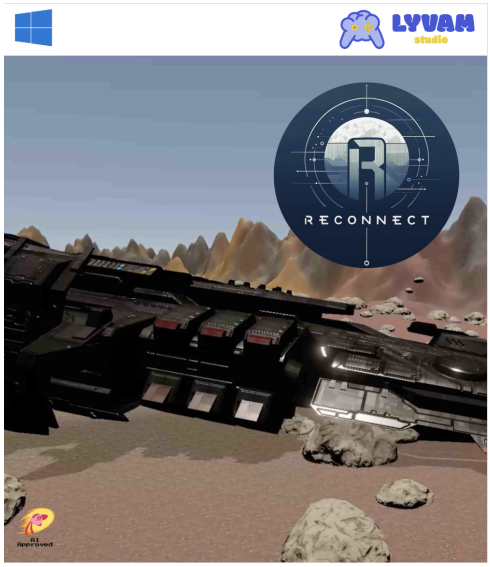
(a) Website V1

(b) Website V3

The third version of the website consists of 3 pages, including the home page, the reconnect game page, and the download page of the reports and the game. In the home page, a description of the studio can be found. In the game page, a trailer presents the game following by cards presentation that detailed the game. Then the credits closed that page. Finally, the download page allows the visitor to download the reports of the three defenses and each version of the game for windows, linux and macos.
The website is made with HTML and CSS. Hence it can be deploy locally on every computers. You can find the website on www.lyvam.studio. The choice of getting a domain name is also involved in communication of the studio. It has to be easy to memorize and fast to type. Moreover, the website is deployed on Vercel, an American cloud platform as a service. It offers the possibility to deploy online for free a website. Its particularity is, once the code is pushed to a git platform, Vercel automatically reload the website.

3.3.3.3 Box

The gamebox designed is also an important part of the project's communication. The box was inspired by playstation and xbox gaming boxes.
The design was made by hand with Affinity Designer. It quickly shows the different parts of the game.



(a) Recto

(b) Verso

Figure 39: Game box design

3.3.3.4 Sweatshirts

Last but not least, the communication also involves derivatives products as sweatshirts and a mascot. Using Louis's drawing, Maxime vectorized a first version of the mascot. A mascot is an easy way to attach the audience to your game. It is the main element of our sweatshirts. Hence, a new version of the medusa has been made to strengthen the attaching link with the audience. In this new version, the medusa is more realistic and tends to join the "kawai" era, that is to say the medusa tends to be cutest.

(a) Medusa V1

(b) Medusa V2

Figure 40: Evolution of the mascot

.

Figure 41: Overview of the sweatshirts

3.3.4 Emotions and experiences

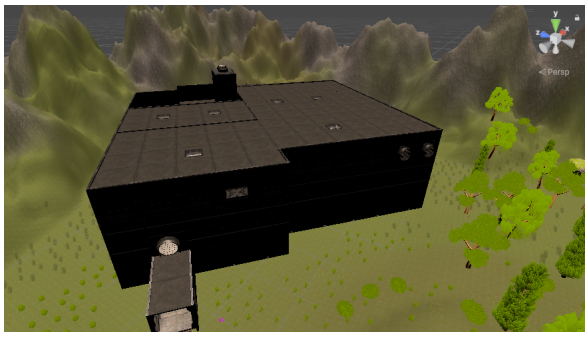
The idea of creating a video game as the first project at EPITA is really exciting. This has been rewarding in every aspects. Firstly, the team organization was a significant matter. Because of the different contributions of each one, the end of this project leaves me a bitter taste. A continuous work would have been the solution where working hand in hand would have allowed me not to be lost in the network implementation. In some way, it is the main teaching that I draw from this project. Secondly, it has reinforced my appeal for graphic design and brand creation. Already comfortable with design tools. I had to push my limits to find what best reflected the team's image. Thirdly, I discovered a new world in game design. With Unity at the center of the game, I was able to explore and learn how to use it. This will allow me to try future projects. Finally, I enjoyed working on this project despite the difficulties of organization.

3.4 Victor

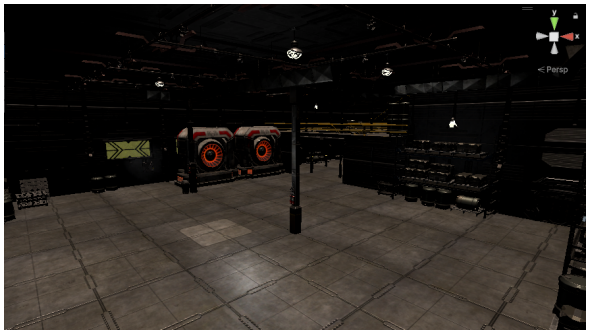
3.4.1 Level Design Implementation

Victor crafted the game's environments by leveraging a combination of Unity assets, each chosen for their unique features and suitability to the game's aesthetic and functional requirements.

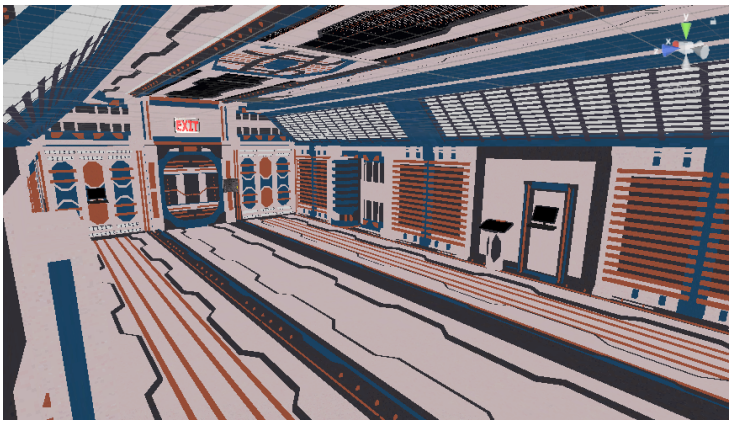
- **Sci-Fi Styled Modular Pack**: Used to build the interior of the crashed ship (Level 1). This free asset offers a set of modular walls, floors, and sci-fi props, allowing for flexible and efficient design of enclosed environments. Its detailed textures and compatibility with Unity's lighting system made it ideal for creating immersive spaceship interiors.
- **RTS Sci-Fi Game Assets v1 + Sci-Fi Construction Kit (Modular)**: These two asset packs were combined to create the laboratory spaces. The RTS pack provided external structures and props suitable for base layouts like the generators, while the modular construction kit was used for detailed and functional indoor spaces.



(a) Laboratory exterior using RTS Sci-Fi assets



(b) Main lab area



(c) Ship interior using Sci-Fi Modular Pack

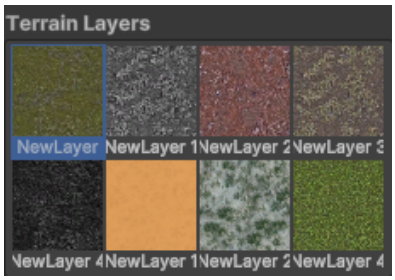
Figure 42: Key environment implementations

Interactive elements like the laboratory door were customized from base assets to add interactivity and realism:

3.4.2 Terrain Development

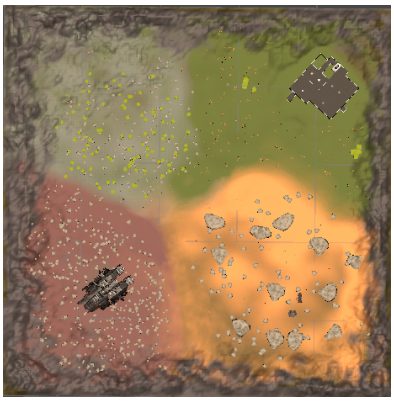
Victor's approach to terrain development focused on natural and immersive world-building by selecting asset packs with high realism and aesthetic compatibility.

- **Terrain Textures Pack Free**: Provided 4K PBR ground textures that enhanced the realism of the environment's surface. These materials blend seamlessly for varied terrain such as grass, dirt paths, and rocky ground.
- **Idyllic Fantasy Nature**: Offered vegetation, trees, and rocks to create an immersive outdoor landscape. The quality and stylized nature of these elements matched the intended visual tone of the world.



(a) Ground texture blending example

(b) In-game vegetation implementation



(c) Top-down view of terrain layout

Figure 43: Terrain system implementation showing texture work and spatial planning

3.4.3 The Challenges

- **URP Compatibility**: Because we use Unity's Universal Render Pipeline instead of the default renderer, many sci-fi asset packs weren't ready for it. I had to run Unity's Render Pipeline Converter on most assets—and since it only worked about half the time, I often ended up reassigning materials by hand or scrapping the asset entirely.
- **Performance Optimization**: In the lab level especially, too many high-detail models and small props caused noticeable frame-rate drops. To fix this, I replaced heavy meshes with lighter versions, balancing visual quality against polygon count to keep the game running smoothly.
- **Finding Free Assets & Importing**: Unity's Marketplace has great paid models but limited free options, so I searched sites like Sketchfab for things like the crashed-ship exterior. Importing those 3D files isn't as simple as dragging them into Unity, each texture, UV map, and material has to be set up manually, and sometimes textures or normal maps need extra editing before they work correctly in URP.

3.4.4 Custom Door System

Victor implemented the door system using Unity's scripting tools, without relying on pre-made animations. This approach offers precise control over when and how the door opens—useful for triggering door interactions based on gameplay logic, such as puzzle resolution or quest progress (e.g., unlocking after completing Level 3).

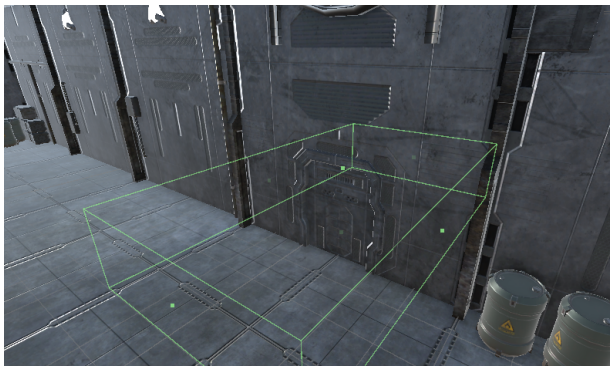


Figure 44: Door system with collision triggers

The door system is built around a `DoorController` script that moves the door's position over time and plays appropriate sound effects. Key features include:

- **Smooth door movement via code:**

  Instead of using an Animator component, the door slides open vertically using a linear interpolation. This gives full control over the movement and allows conditional opening later in the game:

```
Vector3 targetPosition = isPlayerNearby ? openPosition : closedPosition;
door.position = Vector3.Lerp(door.position, targetPosition, Time.deltaTime * speed);
```

- **Integrated sound system:**

  Sound feedback is handled using a shared `AudioManager`, keeping audio logic centralized and volume consistent across the game:

```
audioSource.clip = AudioManager.Instance.doorOpen;
audioSource.volume = AudioManager.Instance.ambianceVolume;
audioSource.Play();
```

This custom solution offers greater flexibility than animation-based systems, especially for expanding gameplay mechanics such as doors that open only after specific player actions.

3.4.5 Audio System – AudioManager

Victor developed a centralized `AudioManager` class to handle all in-game audio logic, including user interface sounds, ambiance, player movement, and music. This approach ensured consistent volume management and simplified integration across the project.

3.4.5.1 Simple Sound Playback Functions

To make audio integration easy and maintainable, Victor created individual playback functions for each type of sound. These functions abstract the logic of setting volume and

playing clips, allowing other parts of the game (like UI menus or gameplay scripts) to call them without duplicating code. For instance:

```
public void PlayButtonClick()
{
    PlayClip(buttonClick, interfaceVolume);
}
```

This method plays the standard button click sound at the defined interface volume level. Similar methods exist for errors, success sounds, ambiance (like doors), and level-up events.

3.4.5.2 Custom Footstep System

The only custom playback logic created was for footstep sounds. Because movement can vary (walking vs. running), the function randomly selects an appropriate sound and adjusts pitch and volume accordingly:

```
public void PlayFootstep(bool isRunning)
{
    AudioClip[] selectedClips = isRunning ? runningClips : footstepClips;

    int randomIndex = Random.Range(0, selectedClips.Length);
    AudioClip clip = selectedClips[randomIndex];

    if (!footstepSource.isPlaying)
    {
        footstepSource.clip = clip;
        footstepSource.volume = movementVolume;
        footstepSource.pitch = isRunning ? 1.5f : 1.0f;
        footstepSource.Play();
    }
}
```

This logic ensures steps never overlap and sound natural, adding immersion to movement-heavy gameplay sections.

3.4.5.3 Menu Integration

The `AudioManager` was directly integrated into the `MenuManager` script. This made it possible to provide audio feedback for every user interaction in the interface:

```
if (CurrentMenuState is MenuState.BreadBoard)
{
    AudioManager.Instance?.PlayBreadboardOpen();
```

```
}
if (CurrentMenuState is MenuState.NewLesson)
{
    AudioManager.Instance?.PlayLevelUP();
}
if (CurrentMenuState is MenuState.ConnectionFailed)
{
    AudioManager.Instance?.PlayError();
}
if (CurrentMenuState is MenuState.Main)
{
    AudioManager.Instance?.PlayMusicMenu();
}
else
{
    AudioManager.Instance?.PlayButtonClick();
}
```

This ensures each menu screen provides a unique auditory cue, enhancing user feedback and polish.

3.4.5.4 Challenges Encountered

Implementing the audio system revealed several technical challenges:

- **Missing Volume Controls:** There was no in-game volume slider or user-accessible control system, which made it difficult to balance the different audio sources. Some sounds, like UI clicks or ambient effects, were noticeably louder than others, and had to be manually adjusted in the code using hardcoded volume values.
- **Sounds Playing Twice:** Some UI events accidentally triggered the same sound more than once due to multiple overlapping menu transitions. Victor had to debug and ensure only one sound played per interaction by checking state changes more carefully.
- **Sounds Triggering on Load:** Some audio clips (like ambiance) were playing immediately when the scene loaded—even when the player had not interacted yet. This was fixed by delaying audio triggers using Unity variable or state checks to ensure sounds only play after specific player actions.

Overall, this centralized system made it easy to control, update and balance all sound in the game.

3.4.6 Emotions and Experiences

Working on this game project has been an incredible experience, filled with both challenges and rewarding moments. At the beginning of the project, I struggled with using Git for collaboration. I found it confusing to manage branches, make pull requests, and merge different parts of the project. However, as time went on, I gradually learned how to use these tools effectively. It was a valuable lesson in teamwork and version control, and I now feel much more confident when working in a shared development environment.
On the technical side, I also made a lot of progress with Unity. I became much more comfortable with handling the visual aspects of the game, such as importing and applying textures and managing assets. One of the most interesting parts for me was learning how to use colliders in Unity. I understood how to make objects interact with each other through collisions and triggers, which really brought the game to life.

3.5 Louis

3.5.1 3D Maps

Alternatively at the 3D Assets Louis wanted to create a map from zero. He mostly used Blender to create and make a render of the map.

He first sought to create a random ground regarding volumes (hills, hollows) and then covered it with an open-source earth texture. To ensure that the patterns weren't repetitive due to the map's large surface area, Louis used a Voronoi diagram to create random orientations with subspaces of the partition that is the plane.

He then modeled trees to arrange them on the map. Finally, he laid out various assets on the ground, in large numbers and with a random pattern, to create ground vegetation. He used a grass asset, a pebble asset, a flower asset and a mushroom asset.

The main difficulty was to implement the creation. In fact, once the map was uploaded, Louis had to create a Terrain collider which is an invisible shape used to handle physical collisions for a Terrain GameObject.

Two problems occurred, they have been fixed, however he is not satisfied of the result. Firstly, by building a most realistic map, he could not import it in Unity. Louis had to revise his design and degrade the quality. Fortunately, once the revision has been done, he was able to implement it in Unity. Secondly, because of an incompatibility of the assets, all the grass and flowers had disappeared during the importation in Unity.
Unfortunately, no solution has been found, which prompted us to make a second map. It also enabled us to optimize FPS, as the original map had too many assets due to the detail.

Figure 45: Map v1 on blender

3.5.2 Our Mascot

Firstly, Louis thought of something commercially interesting that could represent the game itself and tie the player into the ReConnect universe. Thus, he created a mascot: Squishy, a jellyfish. He first sketched it out, then send it to Maxime who simplified it as described above.

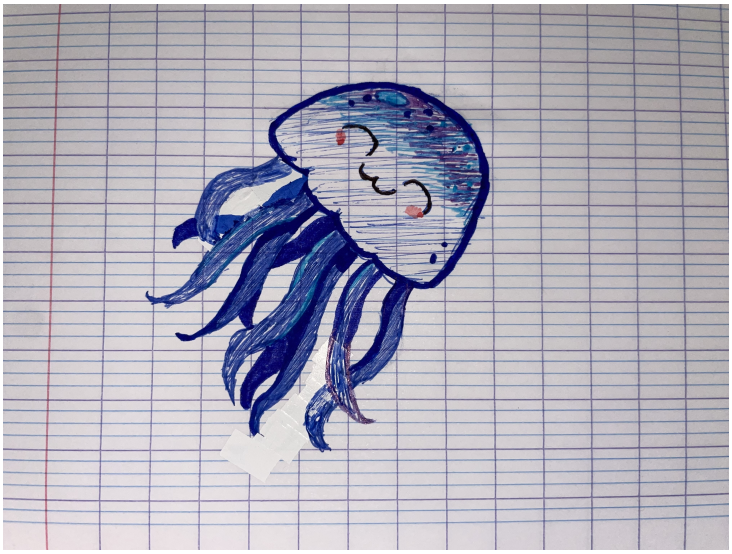


Figure 46: Squishy first sketch

3.5.3 Emotions and experiences

This project offered me a lot : way more humility, new Blender skills (Voronoï textures...), and a super team. Even if I have regrets about my abilities, I'm happy to have had the chance to meet such talented people and to have seen them at work during this year.

4 Conclusion

The ReConnect project represents the culmination of several months of intensive work, and technical exploration. Initiated in November 2024 and completed in May 2025, this project was much more than a simple video game development exercise: it was an opportunity for our team to engage in a multifaceted creative process, combining programming, game design, 3D modeling, and educational content development. As an educational game, ReConnect aims to teach fundamental electrical principles through interactive and engaging gameplay. It places players in a science-fiction setting where critical thinking and problem-solving are key to progression.

Throughout the development process, each team member brought unique skills and ideas to the table, as detailed in their respective sections of this report. From networking and interface design to artificial intelligence, 3D modeling, and level construction, every aspect of the game was built from scratch, often requiring to learn new tools and technologies on the fly. Notably, Unity was a new platform for the entire team at the start of the project, and mastering it was a significant achievement in itself.

Despite facing numerous challenges, including time constraints, technical limitations, and coordination issues, the team succeeded in delivering a playable multiplayer 3D video game with functioning electrical simulations and educational content. The experience was not without stress—particularly during key milestones such as the technical defenses and final presentations—but it was also deeply rewarding. The members learned not only how to build a game, but also how to manage a long-term project, communicate as a team, and overcome obstacles through perseverance and adaptability.

In retrospect, ReConnect has allowed to explore the potential of gamification in education. By embedding core scientific principles within an engaging narrative and interactive environment, the studio believes that it has created a product that is both instructive and enjoyable. While the game is not without its imperfections, the team is proud of what it has achieved with the resources and time available.

Looking ahead, the ReConnect project has provided valuable experience that the members will carry into future academic and professional endeavors. It has confirmed their interest in game development and interactive media as powerful tools for learning and expression. Above all, it has demonstrated what is possible when a team commits to a shared vision and works collectively to bring it to life.

A Appendix

A.1 Menus

Figure 47: Main menu

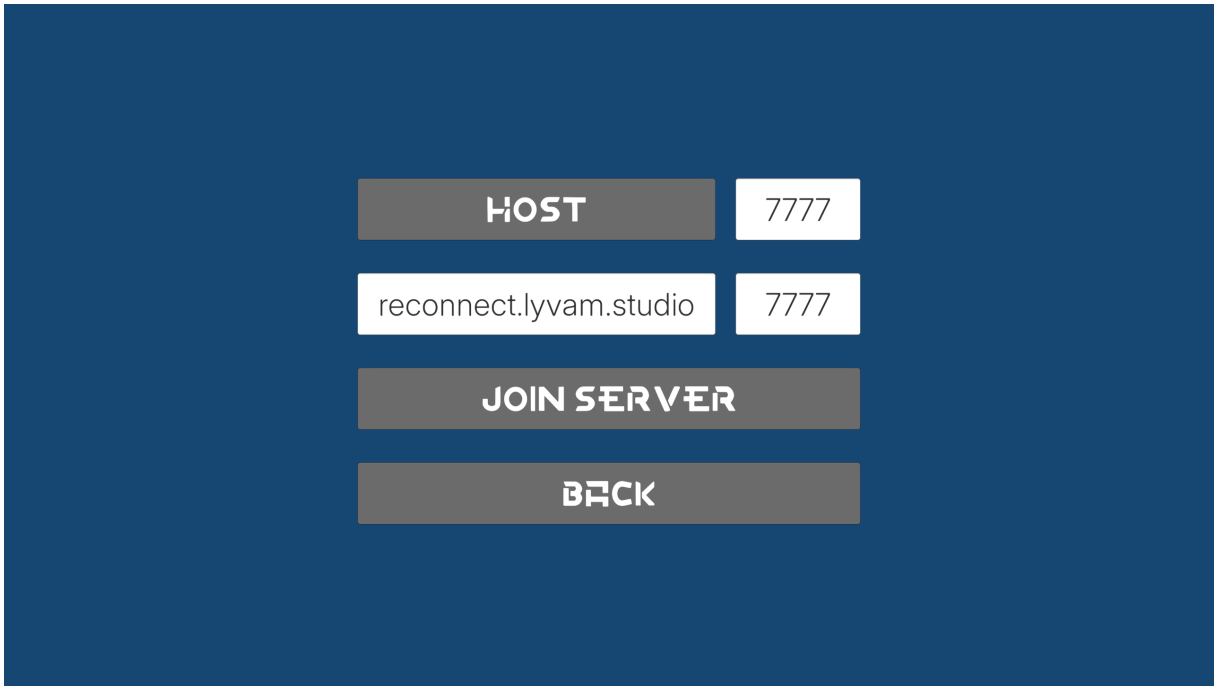


Figure 48: Multiplayer menu

Figure 49: Quit menu

Figure 50: Pause menu

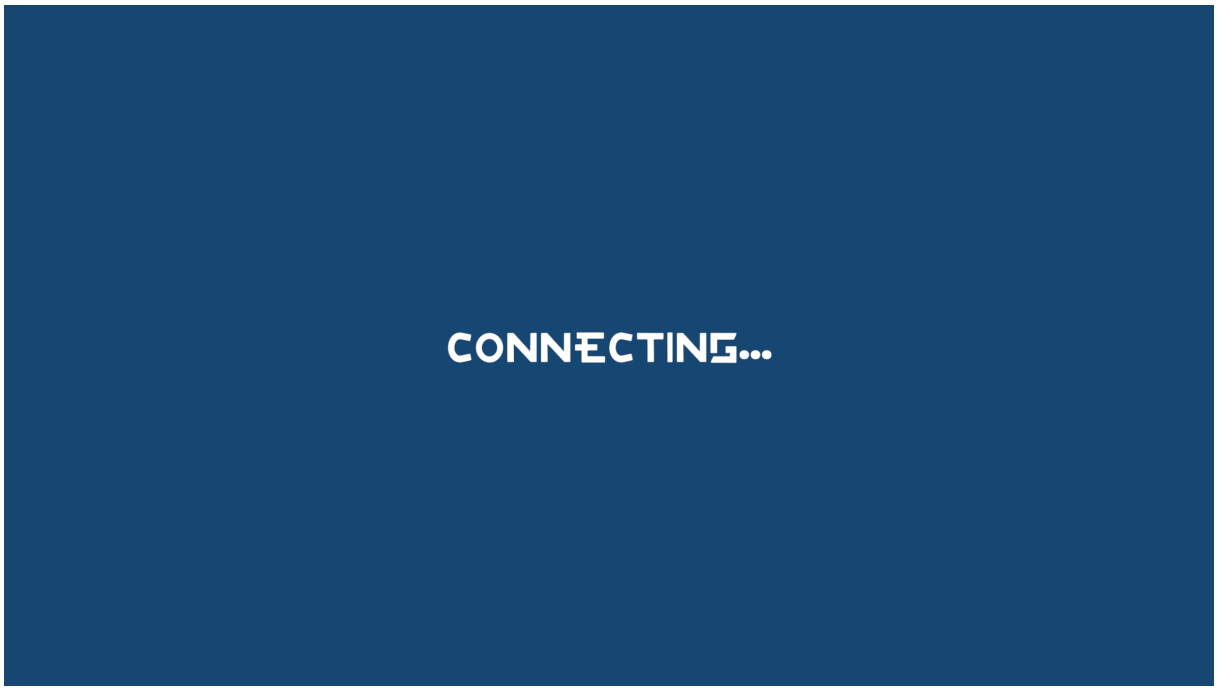


Figure 51: Connection menu

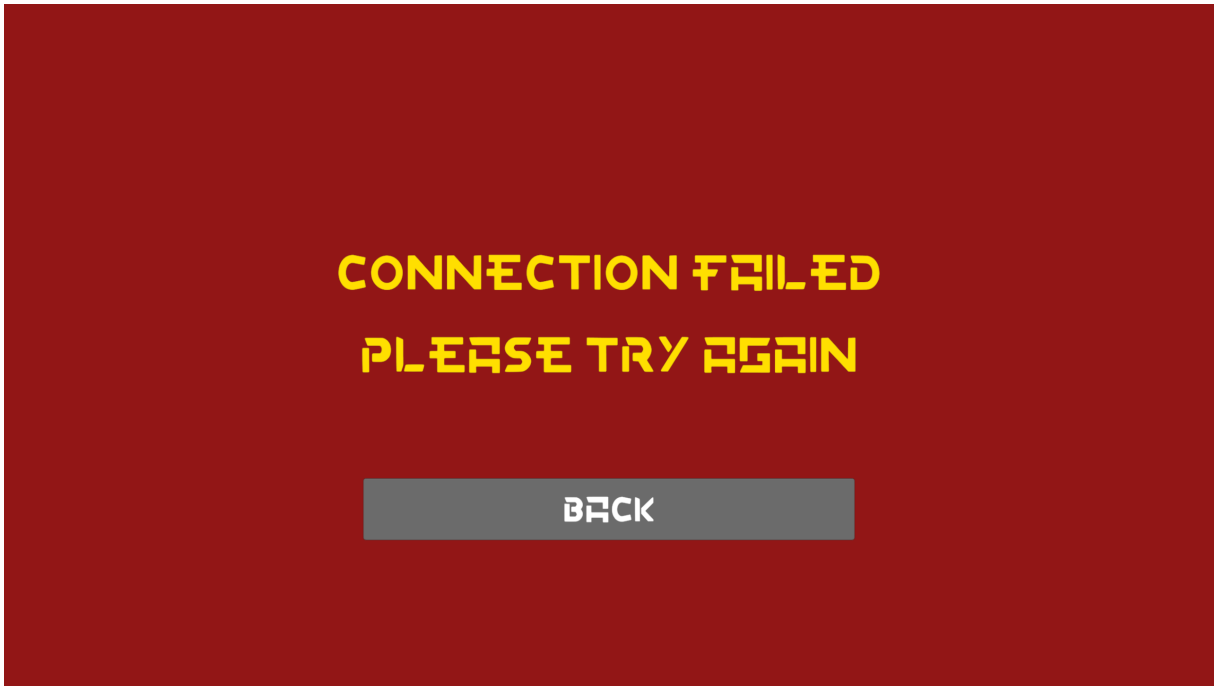


Figure 52: Connection failed menu

A.2 Lessons

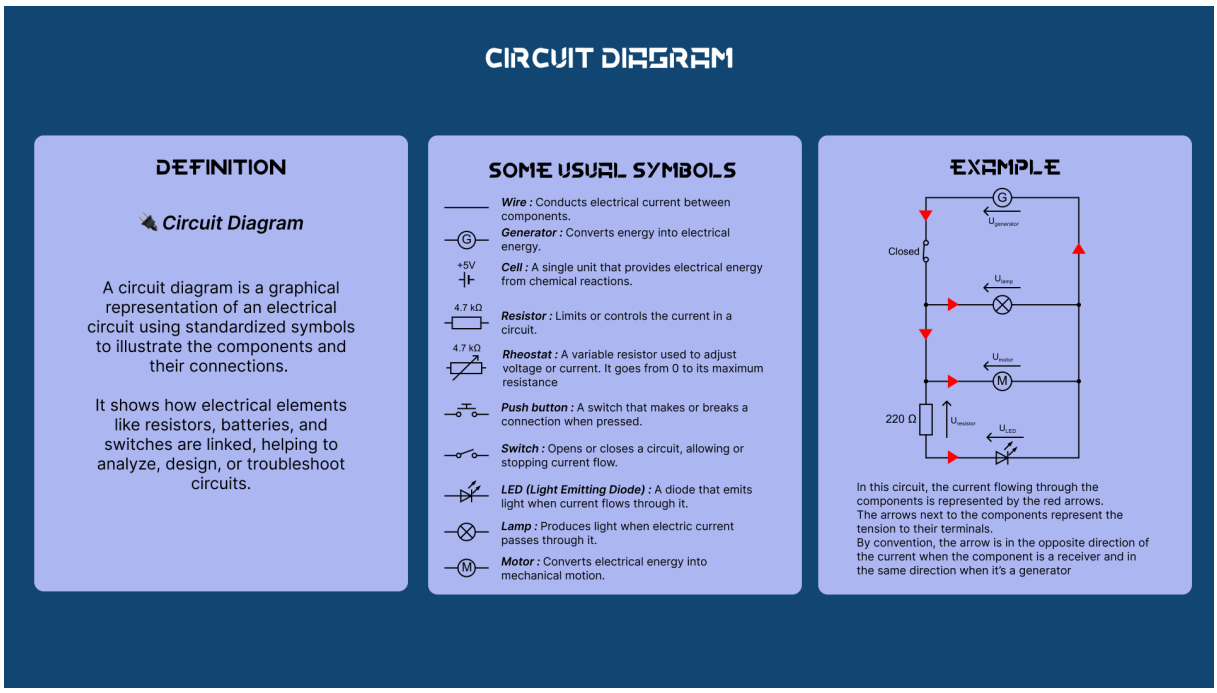


Figure 53: First lesson

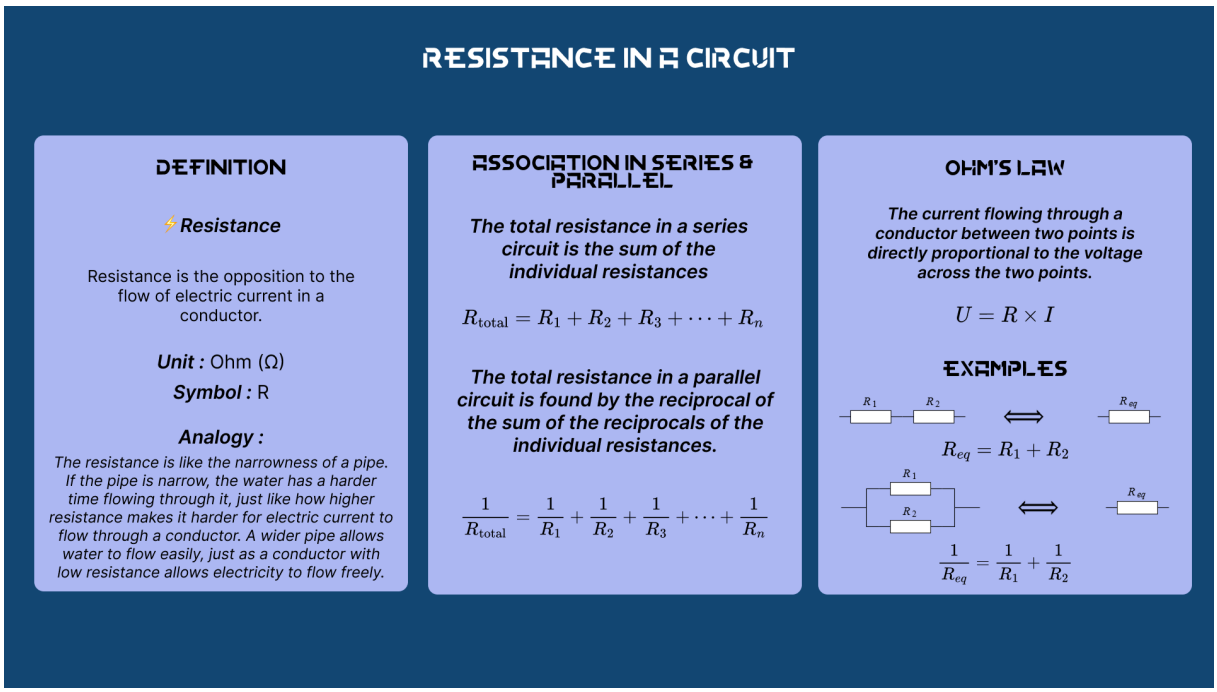


Figure 54: Second lesson

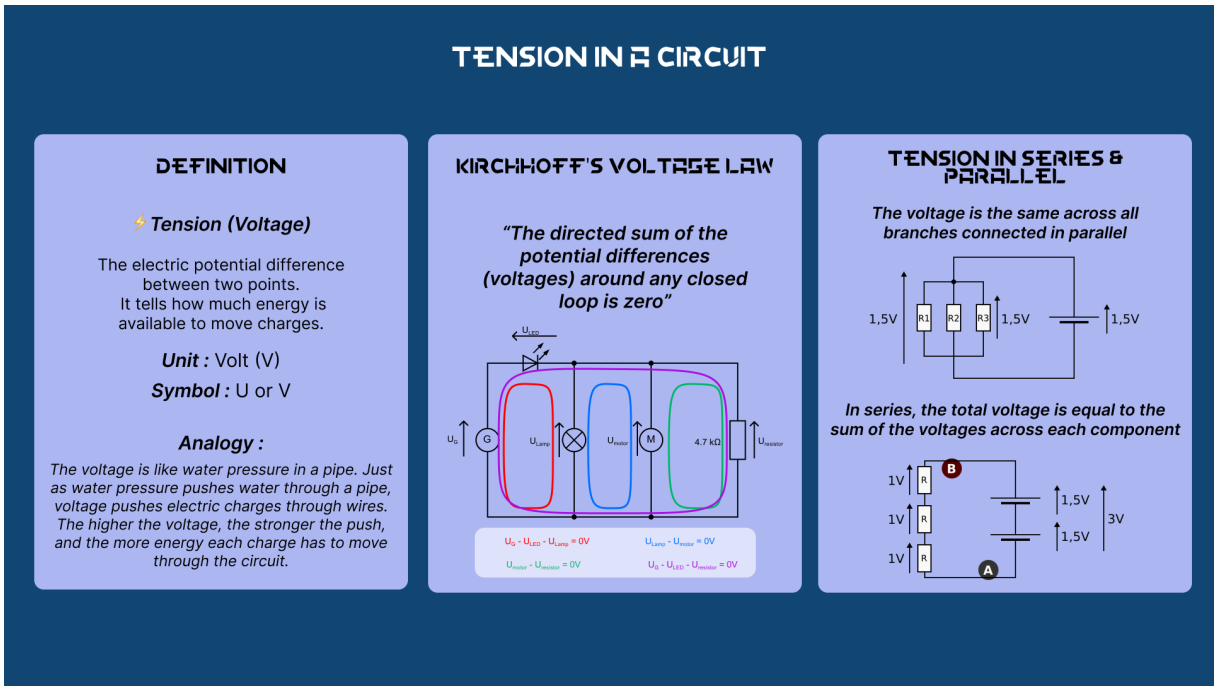


Figure 55: Third lesson

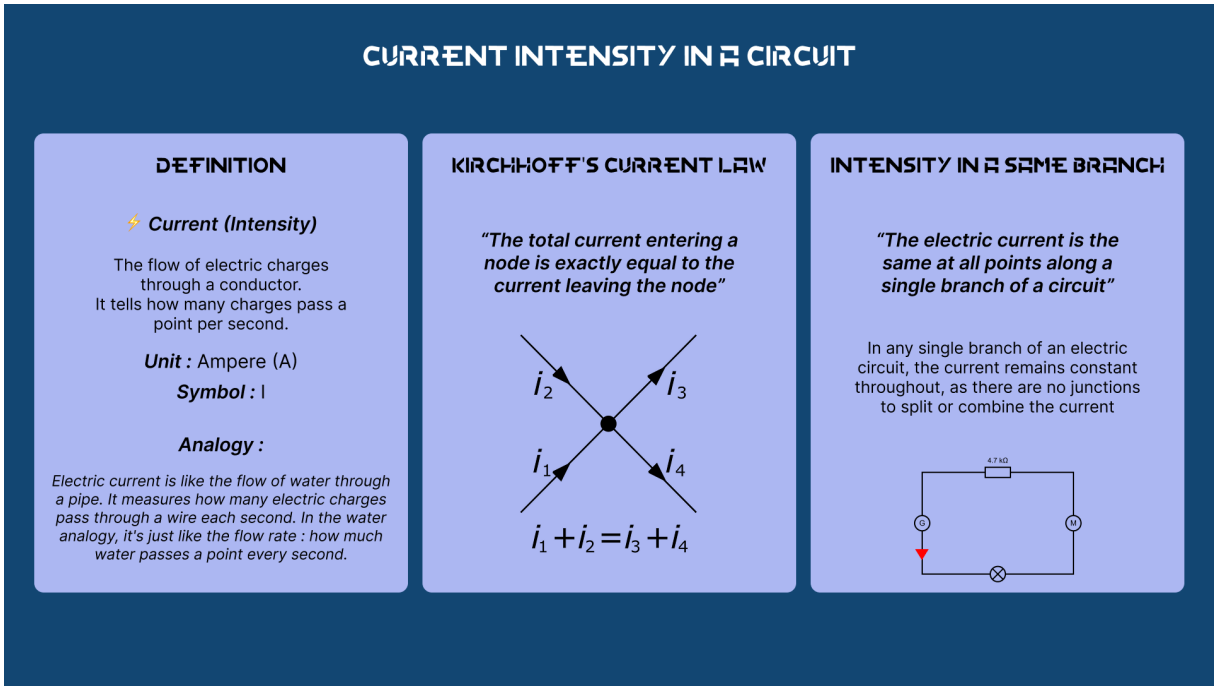


Figure 56: Fourth lesson

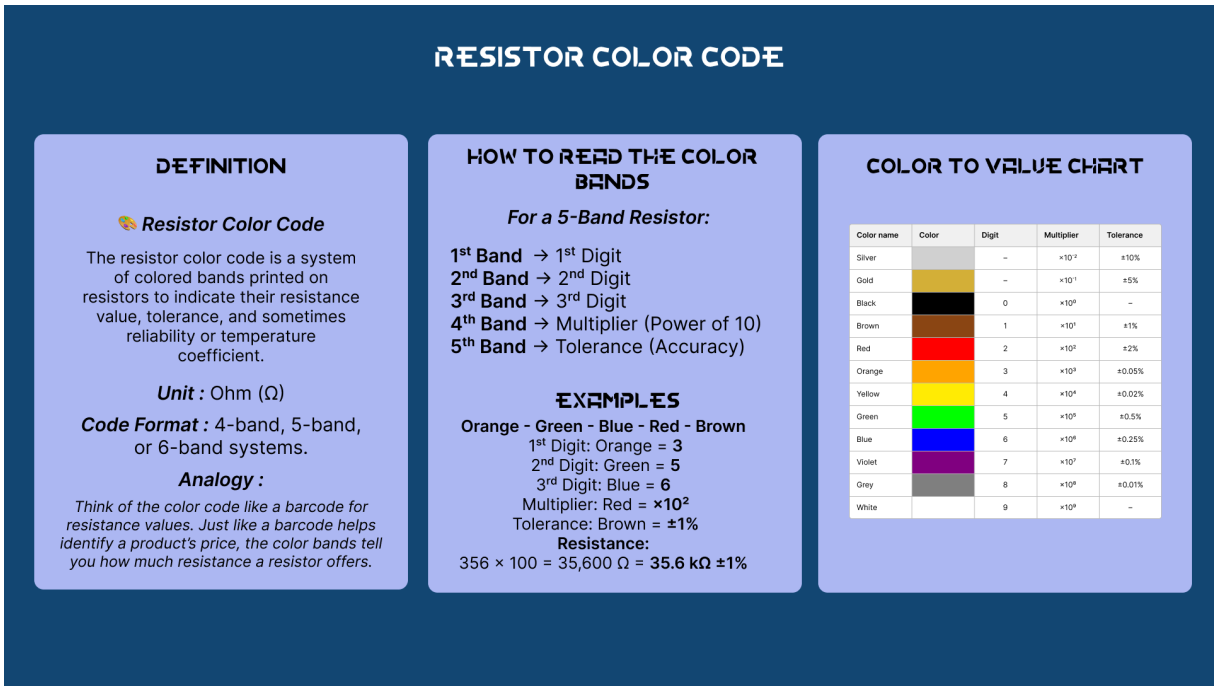


Figure 57: Fifth lesson