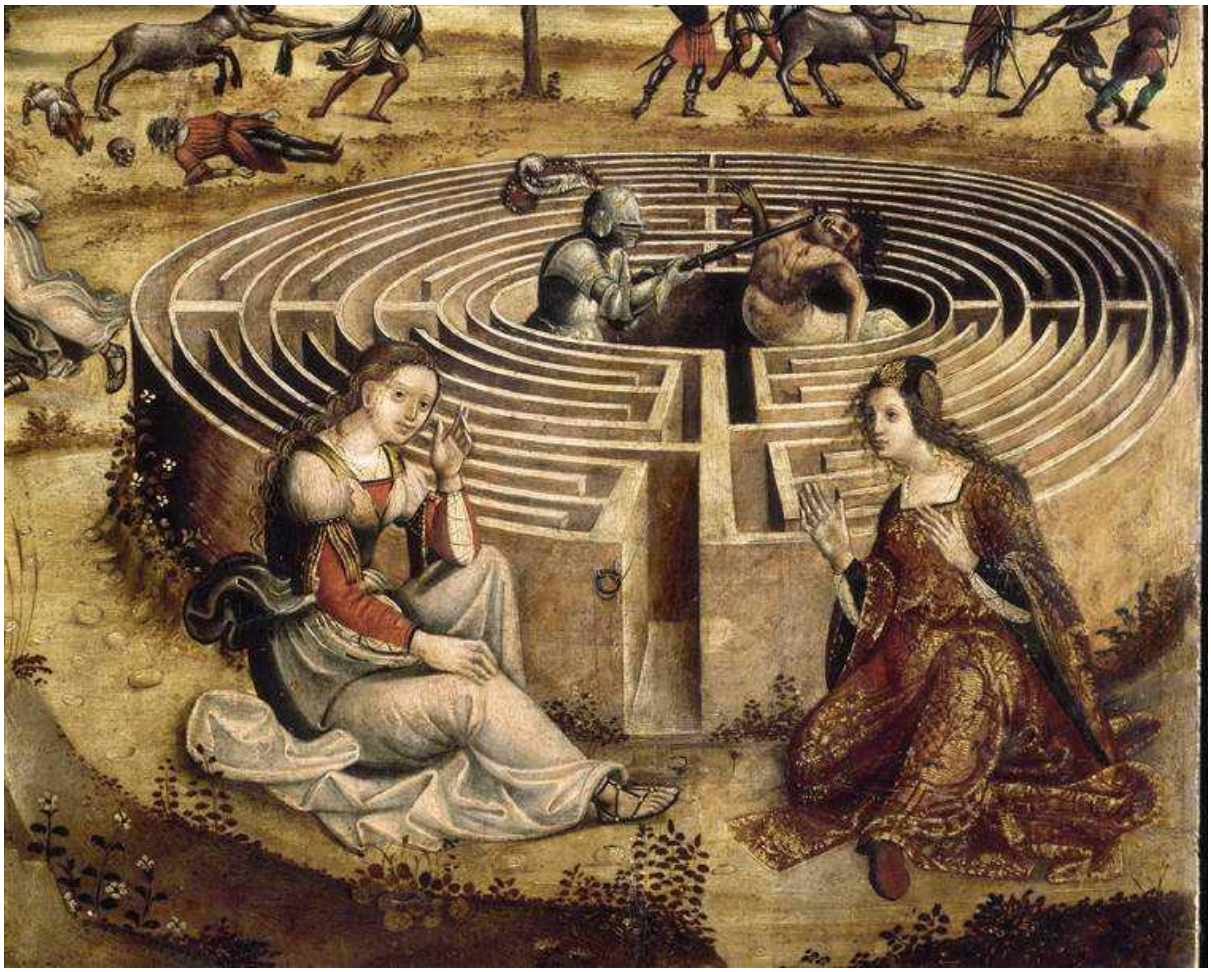
PG111 & LC103 - Bill of specifications: maze in C language



Student
ROYET Jules (Team n°2)
R&I 1A
ENSEIRB-MATMECA

Supervising professors
PAILLARD Emilie
JANIN David
2022-2023

Table of contents

Table of figures

1. Project context

As part of the C programming course of the first year of R&I at ENSEIRB-MATMECA, a project is to be developed to recreate the maze game in C language. This project aims to link theoretical skills such as writing a specification, with technical skills such as development which are modules of our training.

A labyrinth is a sinuous path, with or without branches, dead ends and false trails, intended to lose or slow down the one who tries to move through it. This motif, which appeared in prehistoric times, is found in many civilizations in various forms. In Greek mythology, the word refers to a complex series of galleries built by Daedalus to trap the Minotaur. According to the legend, only three people managed to get out of the Labyrinth: Daedalus because he was the architect, Icarus by flying away and Theseus helped by Ariane and his thread.

The project consists in developing a program in C language, which is a low-level programming language (not very expensive in terms of computer resources), that allows a number of functionalities on the labyrinth such as generating, displaying, browsing and solving labyrinths. Labyrinths are rectangular grids (in ASCII format) consisting of walls and corridors, with a single entrance, which is also the exit. The program will also support tree mazes (mazes whose paths can loop over other paths) and island mazes (mazes without loops), and can include optional treasures and monsters.

This report will explain the different functionalities to be developed for the project, with the help of UML diagrams, spreadsheet scenarios and graphic models. The multiple entities, methods, attributes and relationships will be detailed to be created. Finally, a provisional timetable will be done for the various tasks to be carried out for this project.

2. Requested features

A. Use cases

After analysing the topic, a conclusion is that there is only one actor in our system, which is the Player.

The use cases of this actor correspond to the actions that the user can do in our system and are the following:

- Generate a labyrinth
- Display a labyrinth
- Solve a labyrinth
- Change the hero's movement speed in the system
- Move the hero
- Collect items
- Kill monster(s)
- Stop the game

Use cases can be summarised via a UML use case diagram:

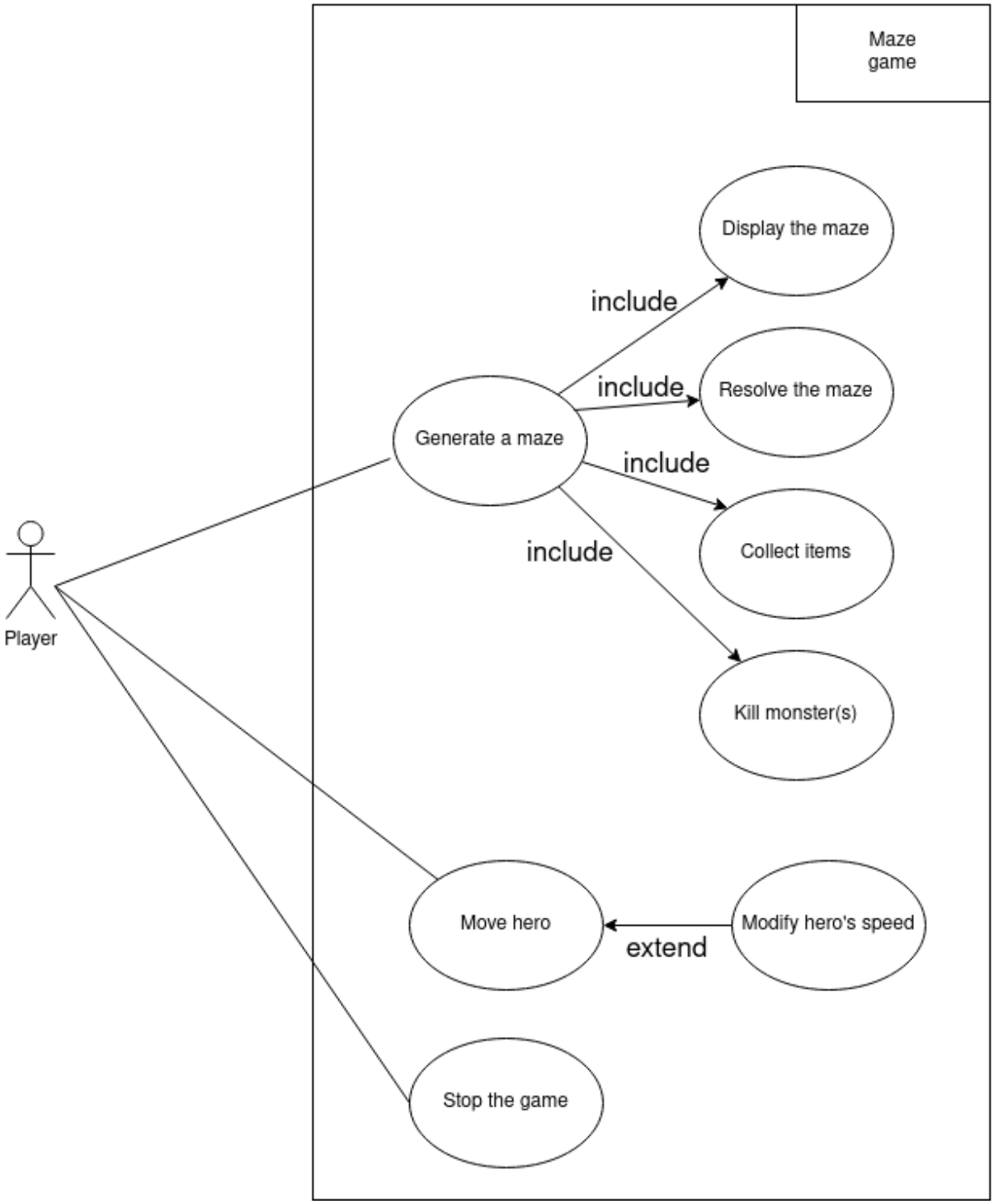


Figure 1: Maze game use cases diagram

Some use cases can be broken down into other use cases as follows:

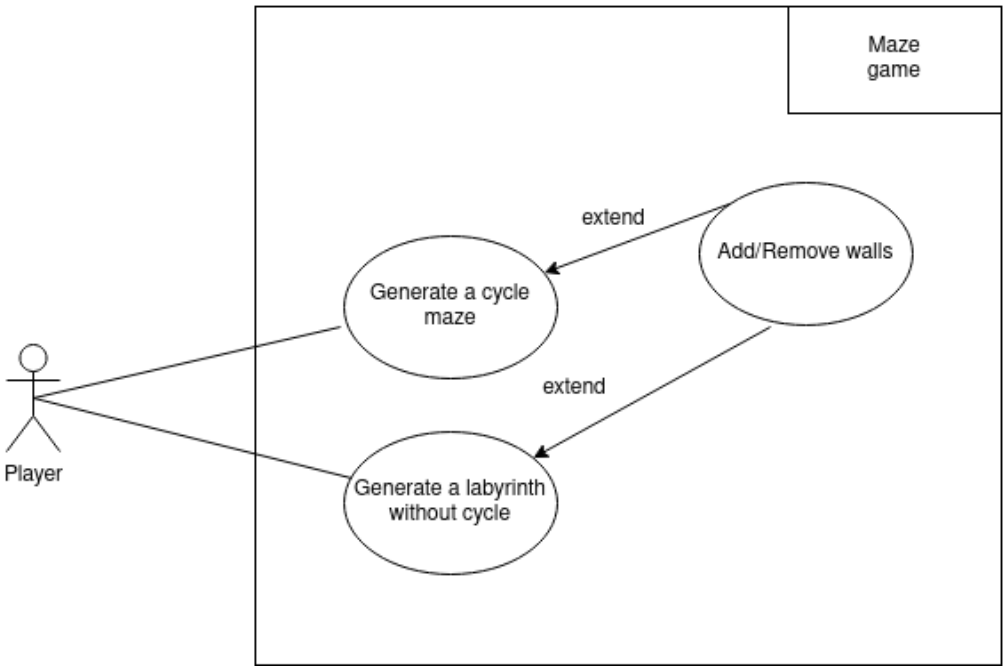


Figure 2: "Generate a maze" diagram

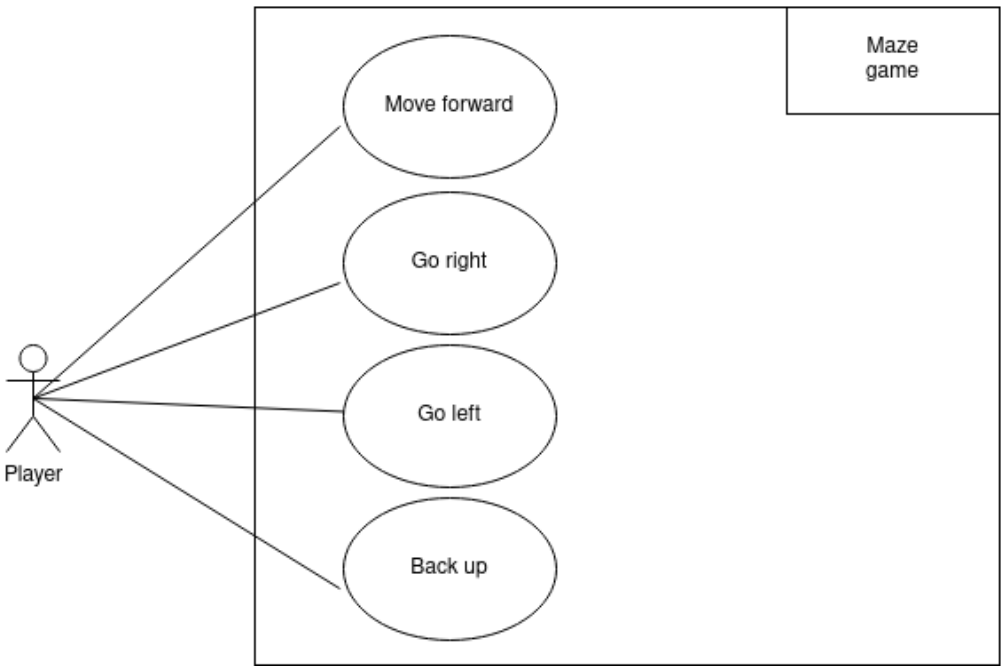


Figure 3: "Move hero" diagram

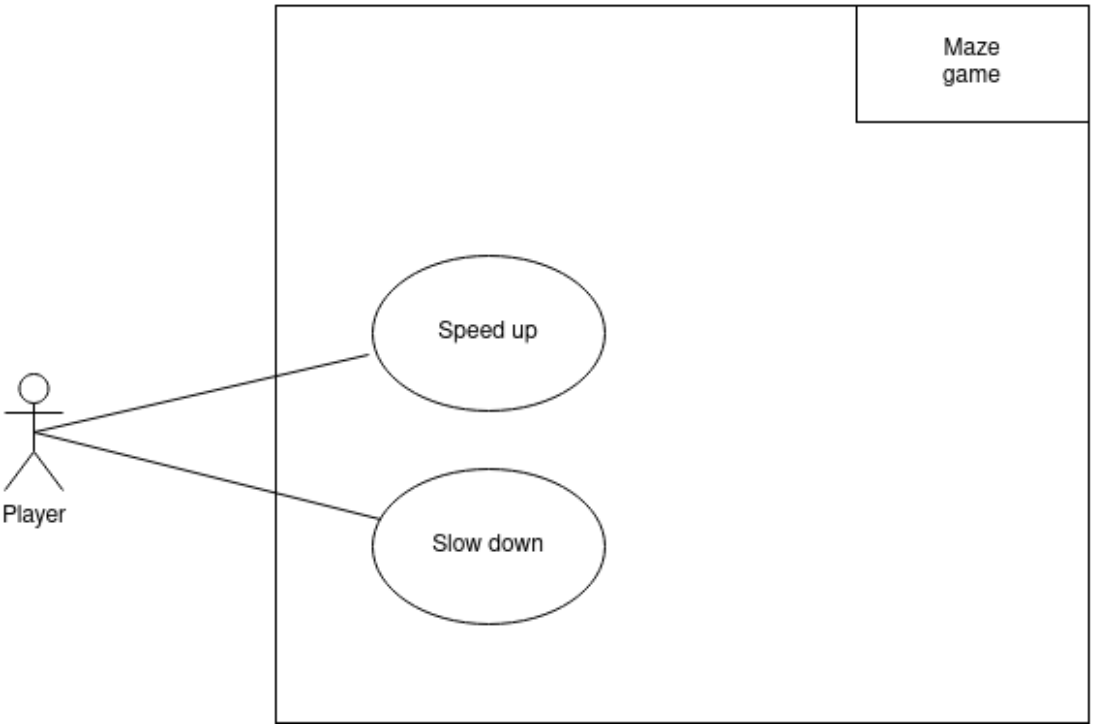


Figure 4: "Modify hero's speed" diagram

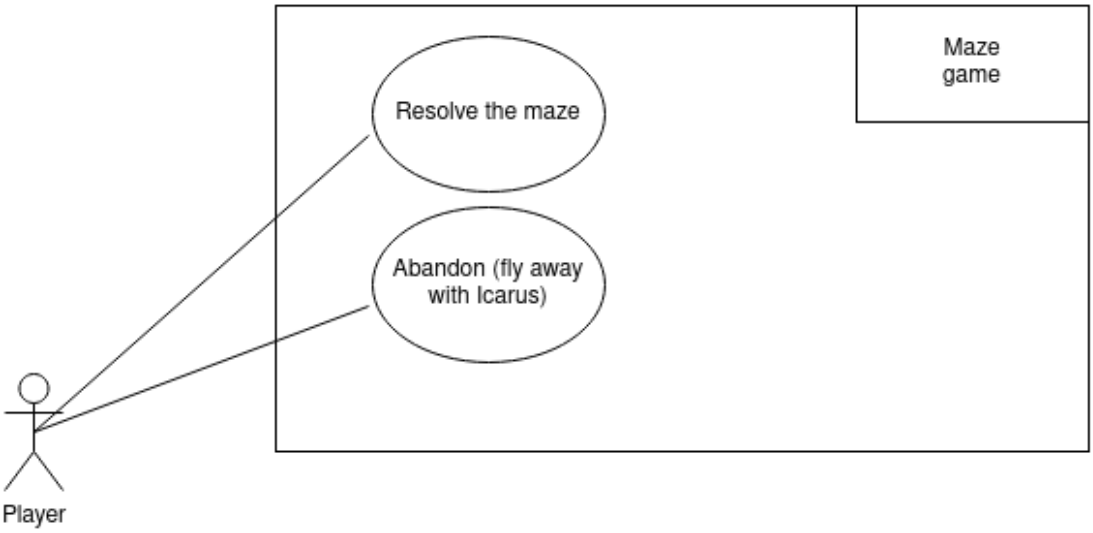


Figure 5: "Stop the game" diagram

Now that the different functionalities have been identified, tests can be associated in order to verify that they are working properly.

B. Associated functional tests

Some features of the program will have to be tested via unit tests.

For the maze generation functionality, we will have to:
- Check that the generated maze is valid (that all the boxes are accessible from the starting box).
- Check that the generated labyrinth is the right size.
- Check that the outer border consists of walls, except for the entrance (separate from the corners).

For the display functionality of the labyrinth, you will have to:
- Check that the labyrinth is correctly displayed, in black for the wall boxes and in white for the corridor boxes.

For the hero's movement functionality:
- Check that the direction keys on the keyboard move the hero in the desired direction.
- Check that the hero bounces off the walls if there is no directional command.

For the hero speed change feature:
- Check that the "-" key on the keyboard decreases the speed and the "+" key increases it.

For the maze solving feature:
- Check that for each corridor square, a direction is indicated to exit the maze.

For the game stopping feature:
- Check that if the hero passes through the exit, the game stops.
- Check that if the player presses the "Escape" key on the keyboard, the game stops.
- Check that a monster that touches the player without a sword stops the game.

For the monster disappearance feature:
- Check that the hero can make only one monster disappear and only if he has a sword.

For the item collection feature:
- Check that collecting an item makes the item disappear from the maze.
- Check that the player has the item.

After defining the different functionalities and associating different tests, scenarios can be designed to act as user manuals.

3. Proposed GUI

A. Scenarios

In this section, the system's responses to the user's actions will be described via different spreadsheets representing action scenarios. These scenarios can be used as user manuals for each action in the application.

Use case	Generate a maze	
Actor	User	
System	Maze Game in C	
Operations	System	User
1		Press “1”
2	Generates the maze via the algorithm	
3	Displays the generated maze	
4		Press “Escape”
5	Returns to the main menu	

Use case	Running a maze	
Actor	User	
System	Maze Game in C	
Operations	System	User
1		Press any arrow
2	Change the current position to the cell indicated by the arrow	
3		Press “+”
4	Increases speed of the player	
5		Press “-”
6	Reduces speed of the player	
7		Press “Escape”
8	Return to main menu	
Exception		
	The next box is a wall	
	Bounce the player back one square	

Use case	Resolve the maze	
Actor	User	
System	Maze Game in C	
Operations	System	User
1		Press “4”
2	Solve the maze using the algorithm	
3	Display the resolved maze	
4		Press “Escape”
5	Return to main menu	

Use case	Manage objects/monsters	
Actor	User	
System	Maze Game in C	
Operations	System	User
1		Press any arrow
2	Change the current position to the cell indicated by the arrow	
3		Press “O”, “M”, “S”
4	Place the associated entity	
5		Press “Return”
6	Delete the entity on the cell	
7		Press “Escape”
8	Return to main menu	
Exception		
		Press “Return” on a cell without an entity
	Do nothing	

In order to provide a little more clarity, the following section will present some graphical mock-ups of what the final interface will look like. These mock-ups will be based on the different scenarios set up earlier.

B. Mock-ups

In this section, graphic mock-ups of our future programme will be created, based on the previous scenarios we have produced.

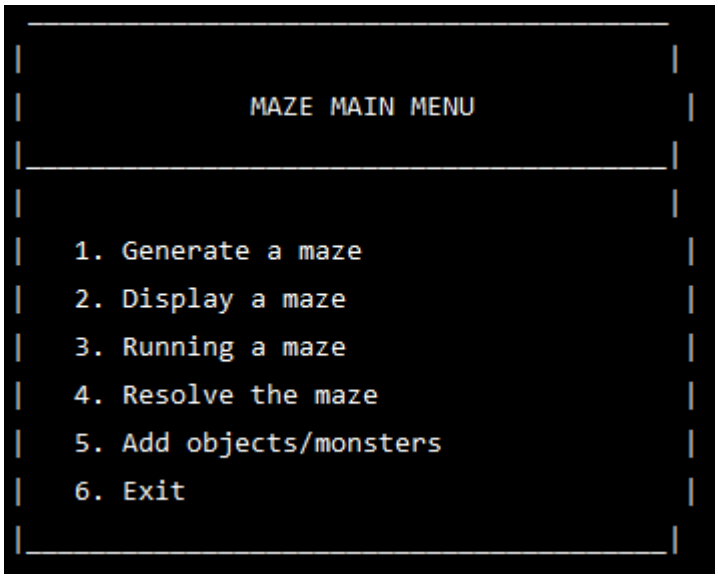


Figure 6: Game main menu mock-up

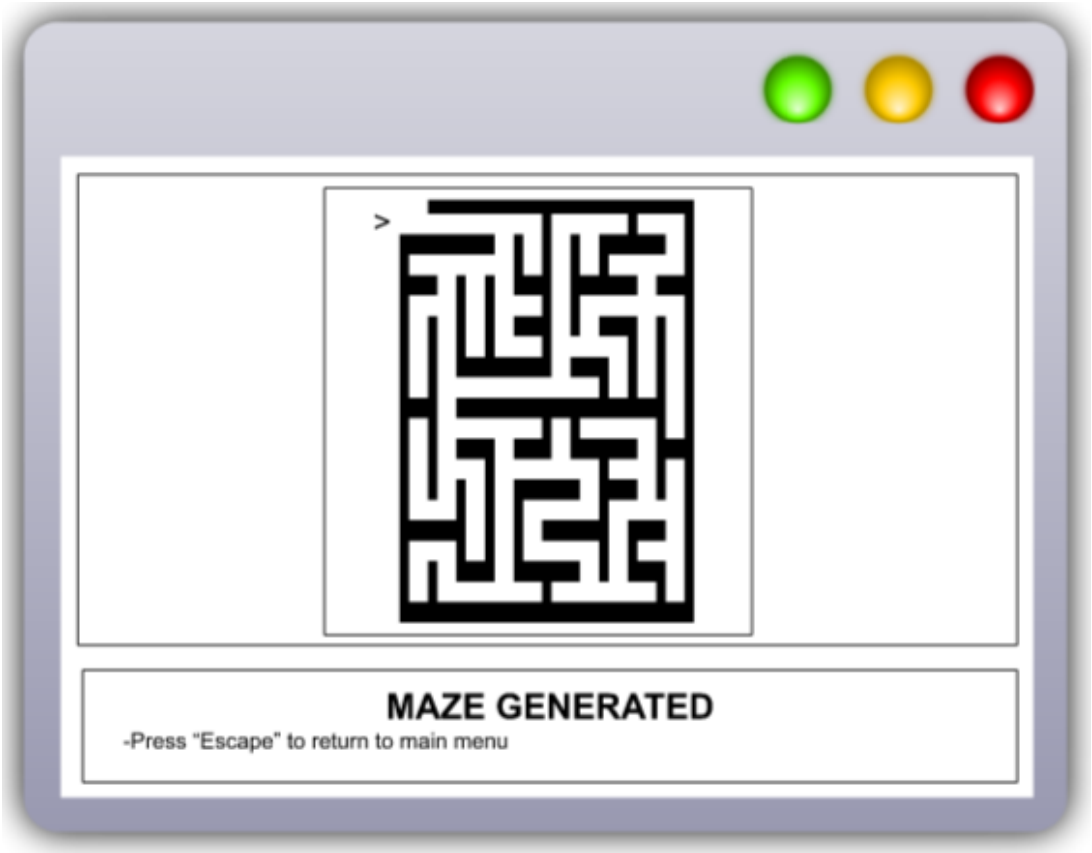


Figure 7: "Generate a maze" mock-up

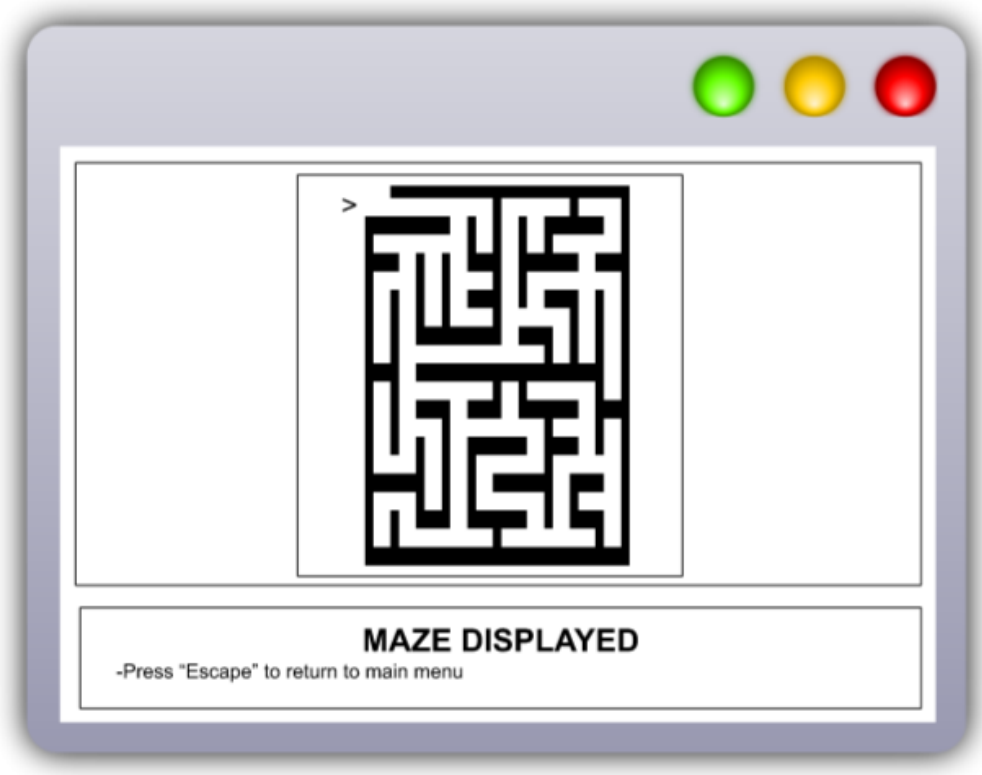


Figure 8: "Display the maze" mock-up

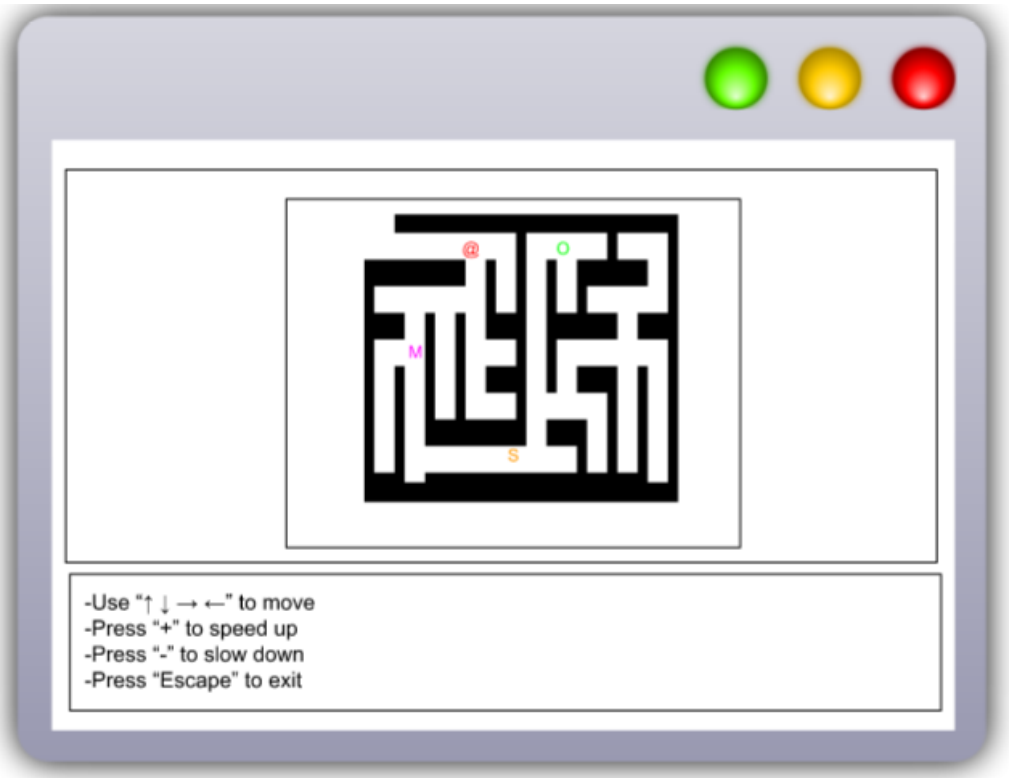


Figure 9: "Running a maze" mock-up

On this model, the player is represented by "@", the monsters by "M", the objects by "O" and the swords by "S".

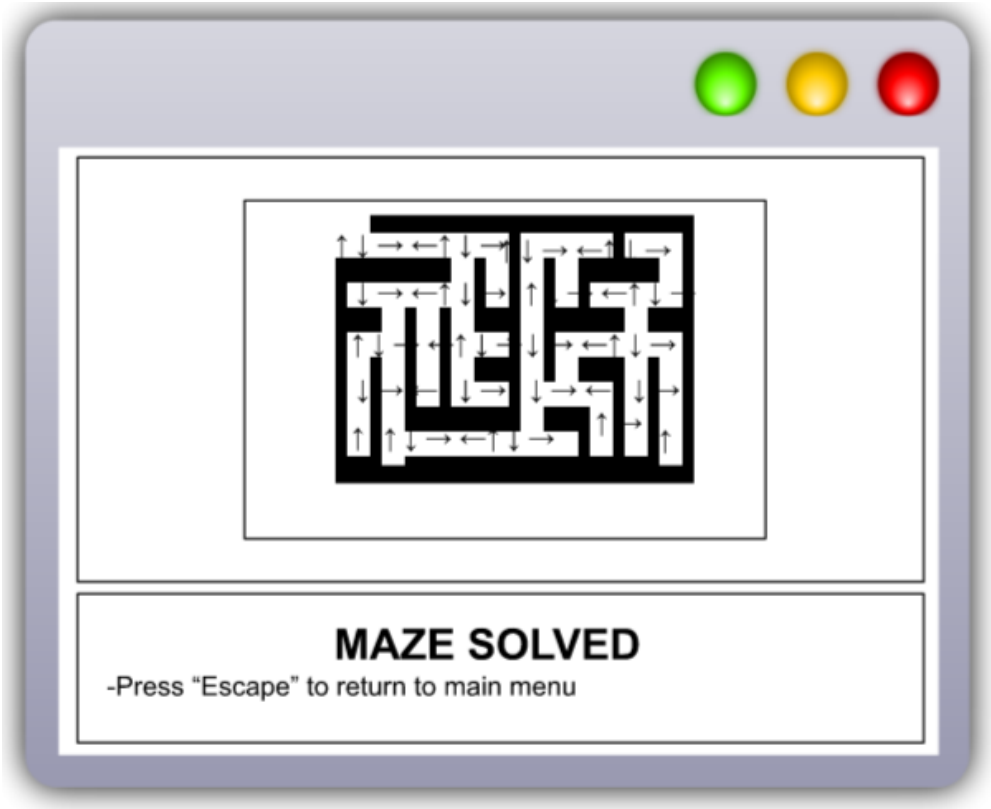


Figure 10: "Resolve the maze" mock-up

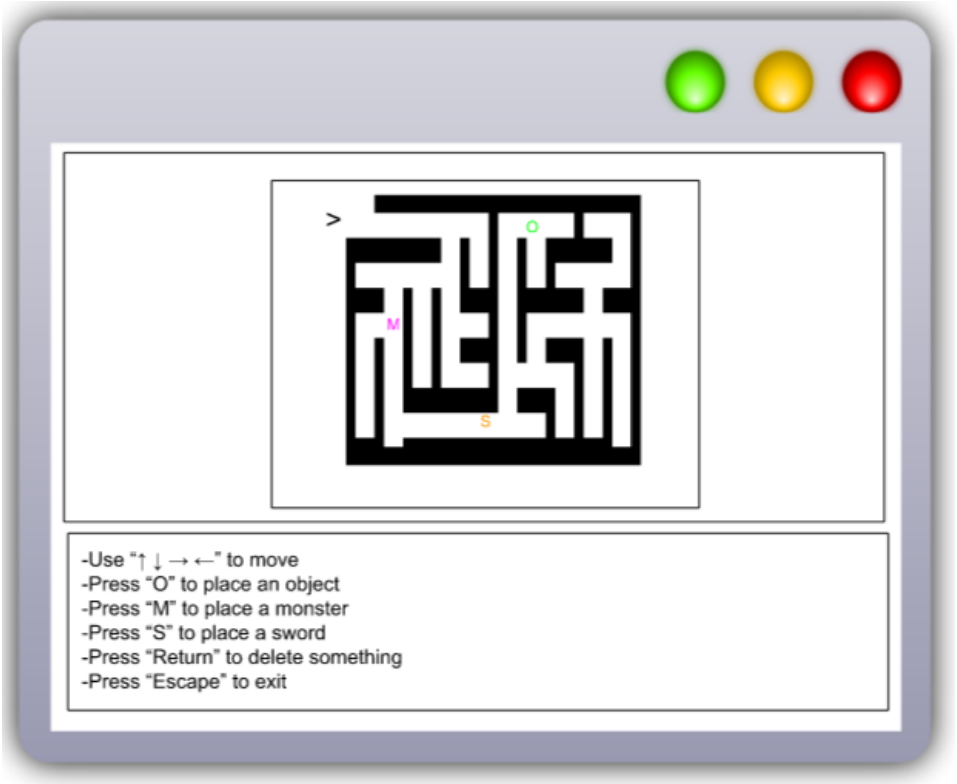


Figure 11: "Manage items/monsters" mock-up

On this model, the player is represented by "@", the monsters by "M", the objects by "O" and the swords by "S".

4. Structural analysis of the data-driven subject

Having studied the subject, producing a first data structural analysis is possible including the entities, attributes, methods and relationships associated with each of them.

A. Data structures

Since classes do not exist in C, all our entities will be struct types.
One of the most reasonable ways to represent a system with its entities, methods, attributes, and relationships is to represent it in the form of a UML class diagram, which is what is done below:

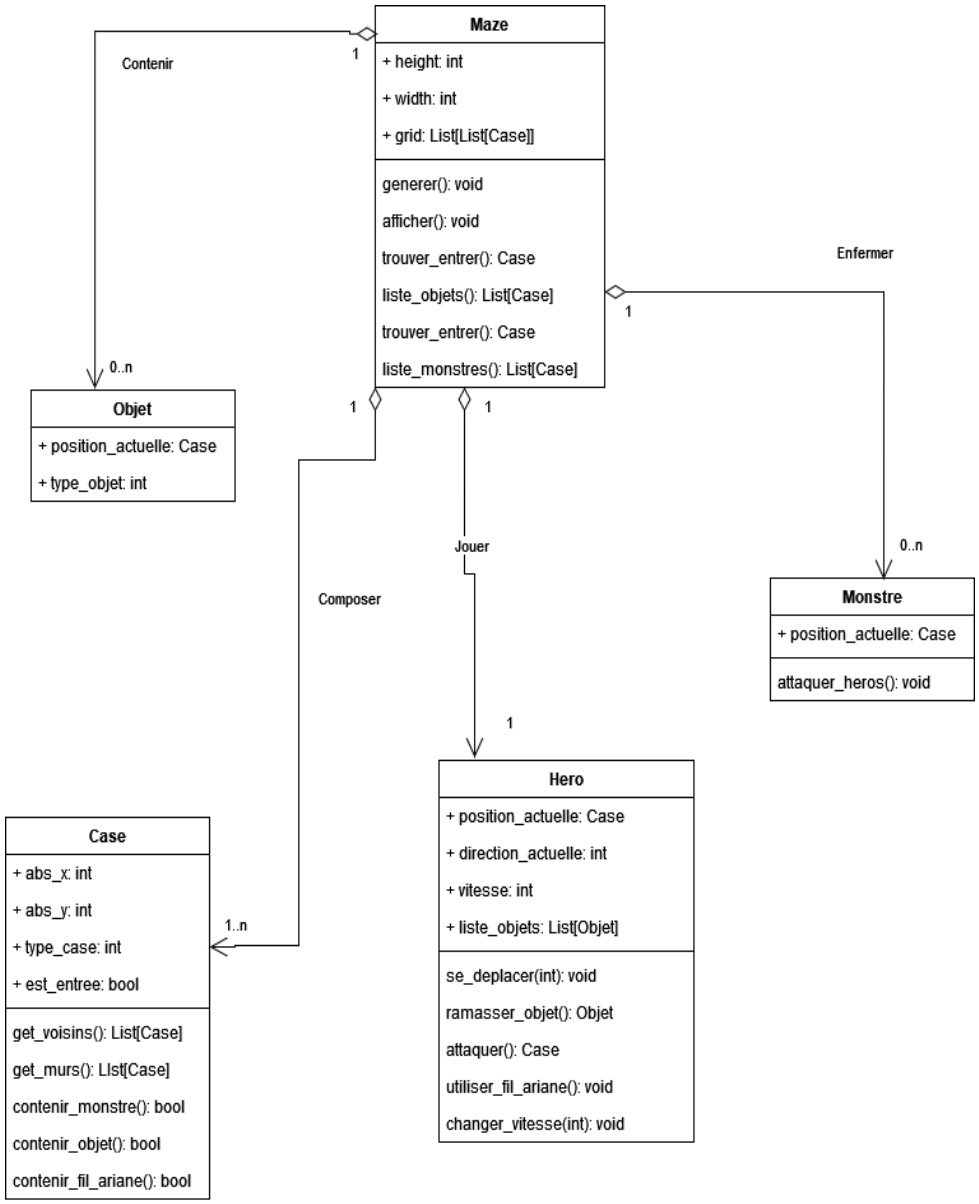


Figure 12: Class diagram of the system

There are good development practices. One of them is to name the different values used globally in the code via enumerated types and constants. This will be the subject of the following sections.

B. Listed types

Enumerated types are used to make sense of certain concepts in the code. Therefore, including them in the project is an appropriate decision.

typedef enum { TREASURE, SWORD, BONUS } object_type
typedef enum { WALL, CORNER } case_type
typedef enum { NORTH, SOUTH, EAST, WEST } direction
typedef enum { ACCELERATE, SLOW } speed

C. Constants

Constants allow for readability of the code, therefore the following will be used in the code:

define MIN SPEED: 10
define SPEED_MAX : 50
define MAZE_CHAR : "▉"

Constants make the code more readable. Still with a view to making the development phase as easy as possible, algorithms will be discussed in the next section.

D. Algorithms

One of the most important components of this project is the development of the algorithms, as these will enable us to develop the functionalities.

➢ Maze creation algorithm

There are many algorithms for creating mazes in C. Since binary trees are a notion that has been covered during the algorithmic courses of this training, the algorithm for creating mazes via trees will be the one implemented in the project.

The principle of this algorithm is as follows:
- We start with a grid where all the walls exist.

- We go through all the cells of the grid one by one (starting at the top left, i.e. at (0,0)).

- For each cell thus traversed, we randomly destroy either the South wall or the East wall.

NB: Be careful that if a cell is located on the right-hand edge of the grid, it is the South wall which will be destroyed. Similarly, if a cell is located on the bottom edge of the grid, it is necessarily the East wall which will be destroyed.

- No wall of the cell located at the bottom right will be destroyed.

➢ Maze solving algorithm

Again, there are several methods to solve a labyrinth. Since the display algorithm is based on the path of a tree, it may be interesting to use the same method for solving the maze.

The solution of the labyrinth will thus be based on a path in width of its tree.

Two cells are defined as adjoining if there are no walls between them.

- A function to obtain all the adjoining cells will be made.

- A function to obtain a matrix containing the distances of each cell from the entrance will be done.

- A function that takes this new matrix as input and places an arrow in the direction of the n-1 of the current square (where n is the value of the current square).

In this way, the maze can be solved.

Once the technical aspects of the development of the programme have been considered, it is important to organise the progress of the project, in particular by planning the different tasks, the necessary resources and the deadlines.

5. Provisional schedule

Different deadlines for this project have been given, which are as follows:

- specification report (March 1 / 10th of march),
- final report (13th of april),
- code (13 April).

Based on these deadlines, a Gantt chart can be produced that allows us to graphically represent our planned schedule and also clarify the various internal and external milestones:

10 March 2023:

- **External milestone:** Submission of the specifications.
- **Internal milestone:** Re-reading to validate the specifications.

10 April 2023:

- **External milestone:** Submission of the final report in English.
- **Internal milestone:** Proofreading to validate the final report in English.

13 April 2023:

- **External milestone:** Submission of the final report in French and the code.
- **Internal milestone:** Proofreading to validate the final report in French and verification that the code works properly and is accessible.

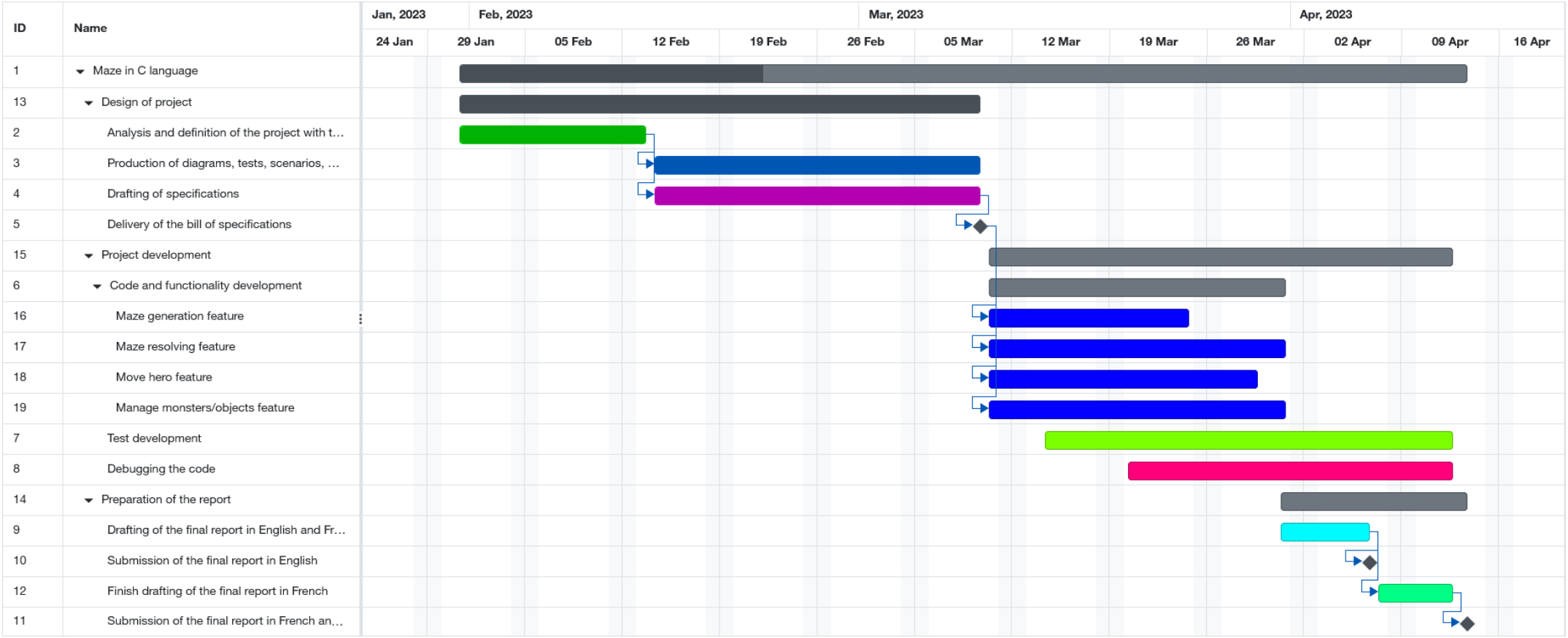


Figure 13: Provisional schedule (Gantt diagram)

ID	Name	Start Date	End Date	Duration	Progress %	Dependency	Resources
1	Maze in C language	Jan 31, 2023	Apr 13, 2023	53 days	30		Jules Royet
13	Design of project	Jan 31, 2023	Mar 09, 2023	28 days	100		Jules Royet
2	Analysis and definition of the project with t...	Jan 31, 2023	Feb 13, 2023	10 days	100		Jules Royet
3	Production of diagrams, tests, scenarios, ...	Feb 14, 2023	Mar 09, 2023	18 days	100	2FS	Jules Royet
4	Drafting of specifications	Feb 14, 2023	Mar 09, 2023	18 days	100	2FS	Jules Royet
5	Delivery of the bill of specifications	Mar 09, 2023	Mar 09, 2023	0 days	0	4FS	Jules Royet
15	Project development	Mar 10, 2023	Apr 12, 2023	24 days	0		Jules Royet
6	Code and functionality development	Mar 10, 2023	Mar 31, 2023	16 days	0		Jules Royet
16	Maze generation feature	Mar 10, 2023	Mar 24, 2023	11 days	0	5FS	Jules Royet
17	Maze resolving feature	Mar 10, 2023	Mar 31, 2023	16 days	0	5FS	Jules Royet
18	Move hero feature	Mar 10, 2023	Mar 29, 2023	14 days	0	5FS	Jules Royet
19	Manage monsters/objects feature	Mar 10, 2023	Mar 31, 2023	16 days	0	5FS	Jules Royet
7	Test development	Mar 14, 2023	Apr 12, 2023	22 days	0		Jules Royet
8	Debugging the code	Mar 20, 2023	Apr 12, 2023	18 days	0		Jules Royet
14	Preparation of the report	Mar 31, 2023	Apr 13, 2023	10 days	0		Jules Royet
9	Drafting of the final report in English and Fr...	Mar 31, 2023	Apr 06, 2023	5 days	0		Jules Royet
10	Submission of the final report in English	Apr 06, 2023	Apr 06, 2023	0 days	0	9FS	Jules Royet
12	Finish drafting of the final report in French	Apr 07, 2023	Apr 12, 2023	4 days	0	9FS	Jules Royet
11	Submission of the final report in French an...	Apr 13, 2023	Apr 13, 2023	0 days	0	12FS+1 day	Jules Royet

Figure 14: Provisional tasks (Gantt diagram

6. Non-functional requirements

For the non-functional needs of the project, the following tools will be used:

- Git and the enseirb repo manager to be able to host and version the documents (CDC, manuals) and the project code.
- Lucidchart to make the different diagrams of the CDC.
- GCC to compile the code that we will produce.
- Visual Studio Code to produce the code in C.
- A Makefile that will allow us to compile the project separately, so as not to recompile the different files of the project, especially those containing the structures, attributes and methods of our entities.
- Figma will allow us to produce the different mock-ups of our future C application.
- Discord will help us to communicate outside of class sessions.
- Google Drive will allow us to share the documents concerning the project.

7. Conclusion

Writing a specification for the production of a game such as the labyrinth in C is an essential step to ensure the success of the project.

We have included in our specifications a detailed description of the game's functionality, technical requirements, and performance specifications, as well as the time frame associated with the development of the game.
We also considered usability and user experience aspects to ensure that the game is enjoyable to play.
Now that this specification has been finalised, it can be used as a guide for the development of the game, ensuring that all requirements are met and that the game is as expected.

In conclusion, this specification is both a driving force for the start of the project design and an anchor for the rest of the project, both in the development of the code and in the production of the various manuals.

8. Bibliography

1. **UML Manual**

https://www.tutorialspoint.com/uml/index.htm

2. **Maze algorithm in programming languages**

https://rosettacode.org/wiki/Maze_generation

3. **C language tutorial**

https://www.tutorialspoint.com/cprogramming/index.htm

4. **Lucidchart**

https://www.lucidchart.com/pages/fr/uml-tutorial

5. **Scenarios**

Computer Science Specification's course of DUT of Bayonne

6. **Gantt schedule**

https://www.gantt.com/creating-gantt-charts

9. Glossary

UML: UML (Unified Modeling Language) is a graphical language for computer modelling.

Scenario: A scenario in computer science is a narrative that describes a set of possible interactions between users and systems.

Mock-up: A model is a partial graphic representation of a system or object.

Milestone: The milestone is a stopping point in the process allowing the project to be monitored. It is an opportunity for the team to make an intermediate assessment, to validate a stage, documents or other deliverables, and then to resume work.

Constant: A constant is a value that must not be changed by the program during its execution.

Listed type: A listed type is a data type that consists of a set of constant values.

Entity: Entities are computer objects that correspond to concepts or physical objects and need to be computerised.

Attribute: Attributes are the properties/characteristics of an entity.

Method: Methods are functions that act on the properties of an entity.

Class diagram: A class diagram provides an overview of a system by presenting its entities and the relationships between them.

Unit test: Unit testing consists of isolating a piece of code and checking that it works perfectly.

Use case: A use case defines the behaviour of a system under various conditions, in response to a user's request to achieve a goal.