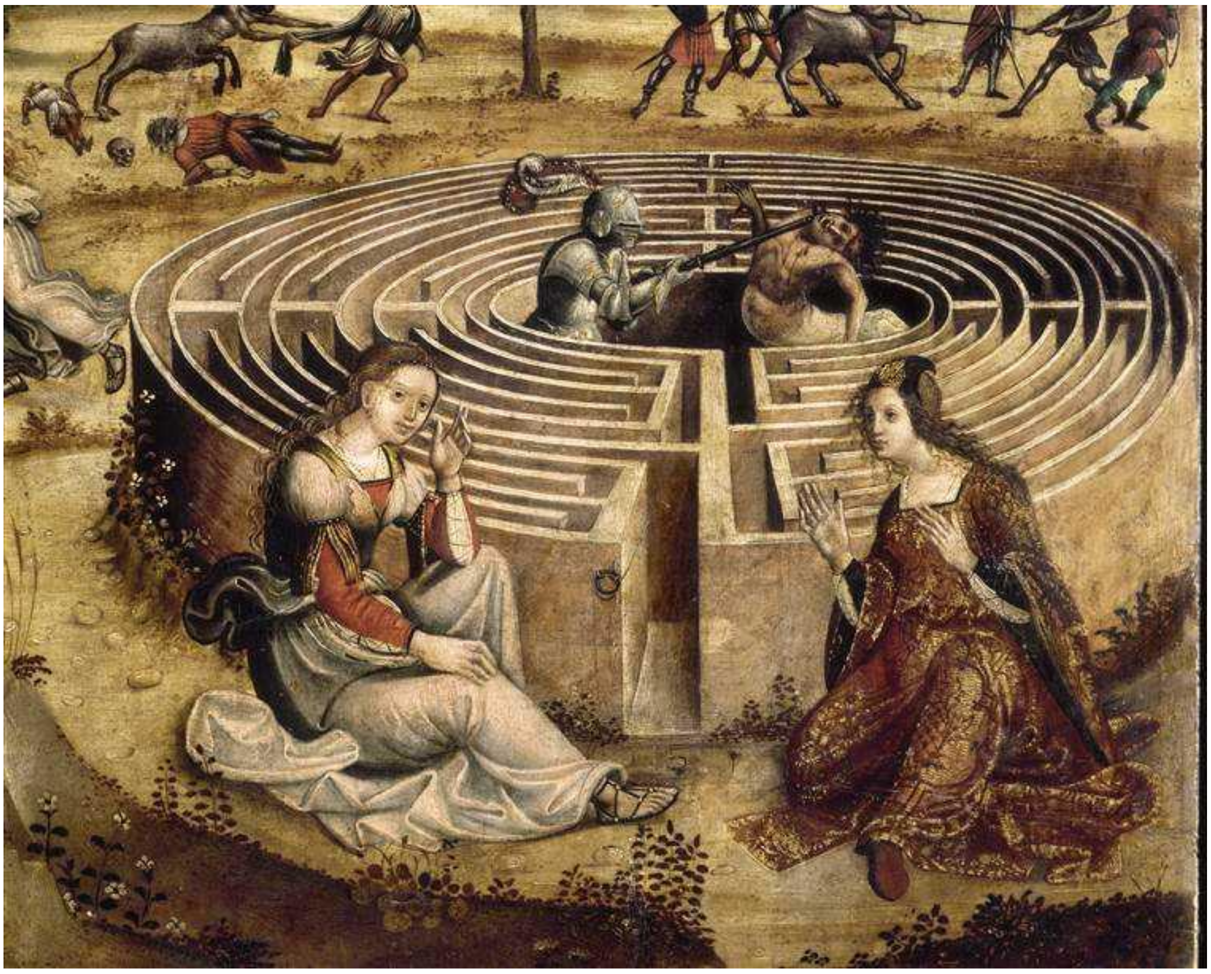
PG111 & LC103 - Bill of specifications: maze in C language



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 part of the students' academic training. Before getting to the heart of the matter, it is important to define what a maze is.

A maze 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 program should support the following features: generating, displaying, browsing and solving maze. In computing, the maze will be represented by 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 (Unified Modelling Language) 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, it can be concluded that there is only one actor in the system, which is the Player.

The use cases of this actor correspond to the actions that the user can do in the 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:

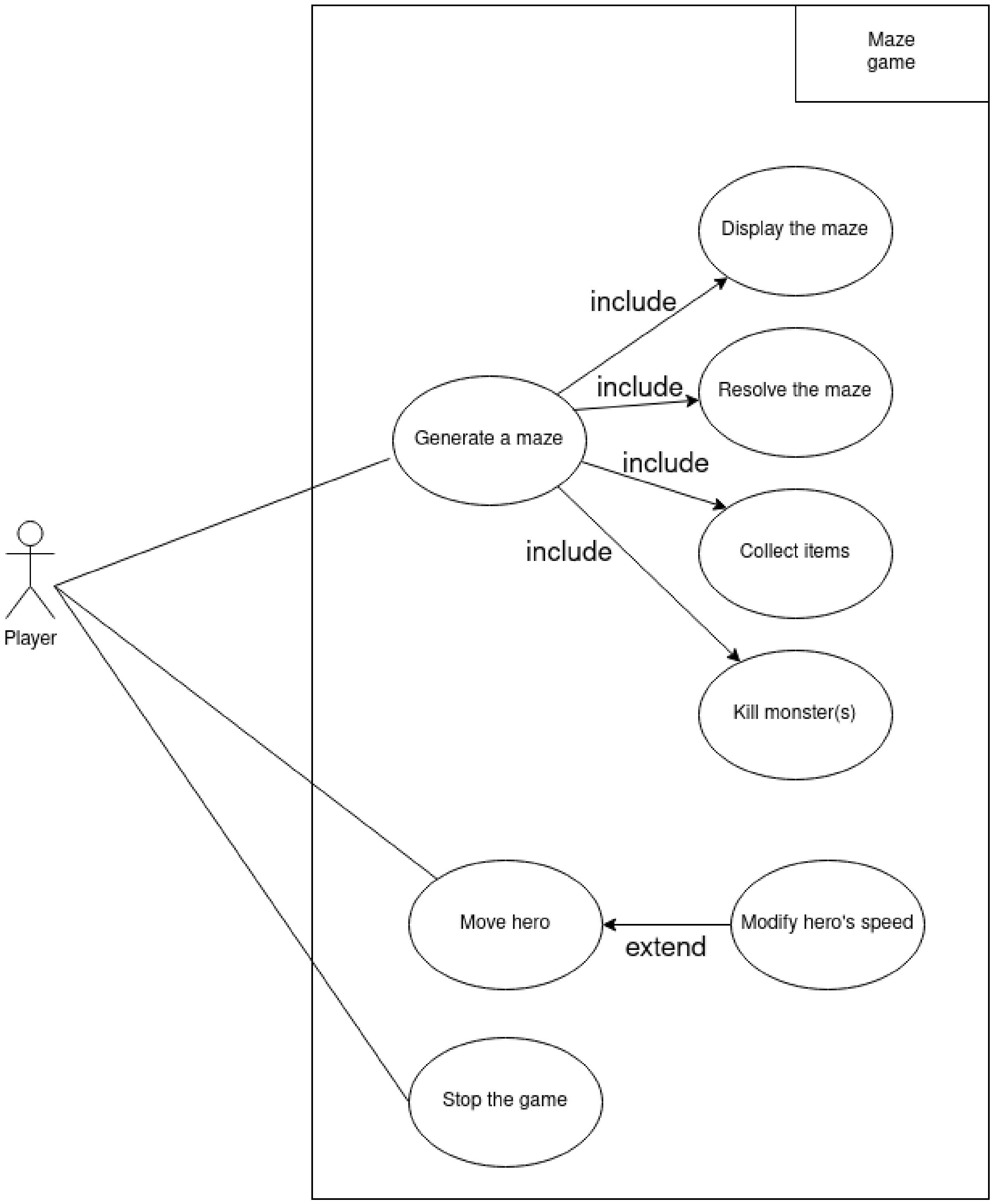


Figure 1: Maze game use cases diagram

This diagram provides a visual understanding of the functionality available in the application and the actors who can access it.

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

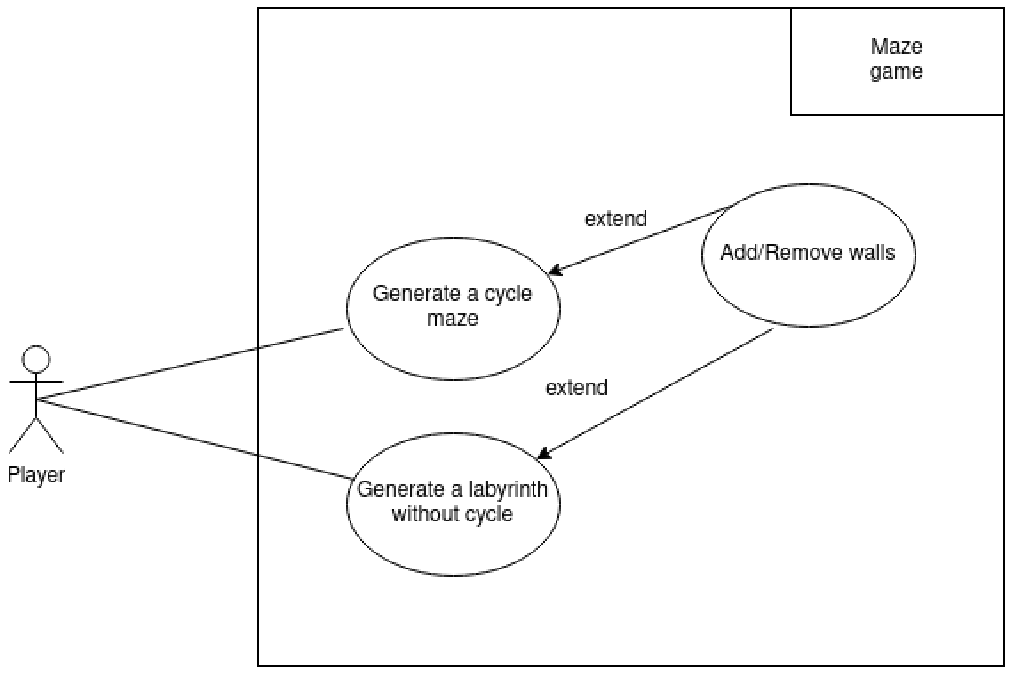


Figure 2: "Generate a maze" diagram

The user will be able to generate labyrinths, with and without cycles.

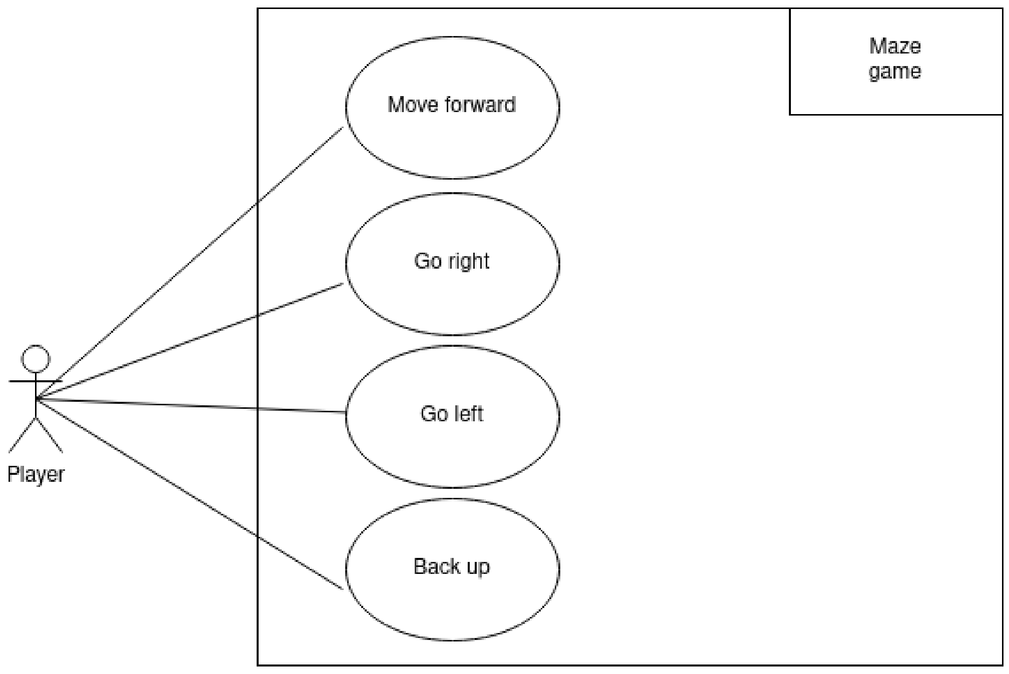


Figure 3: "Move hero" diagram

When he moves, the hero can move forward, go to the right, go to the left.

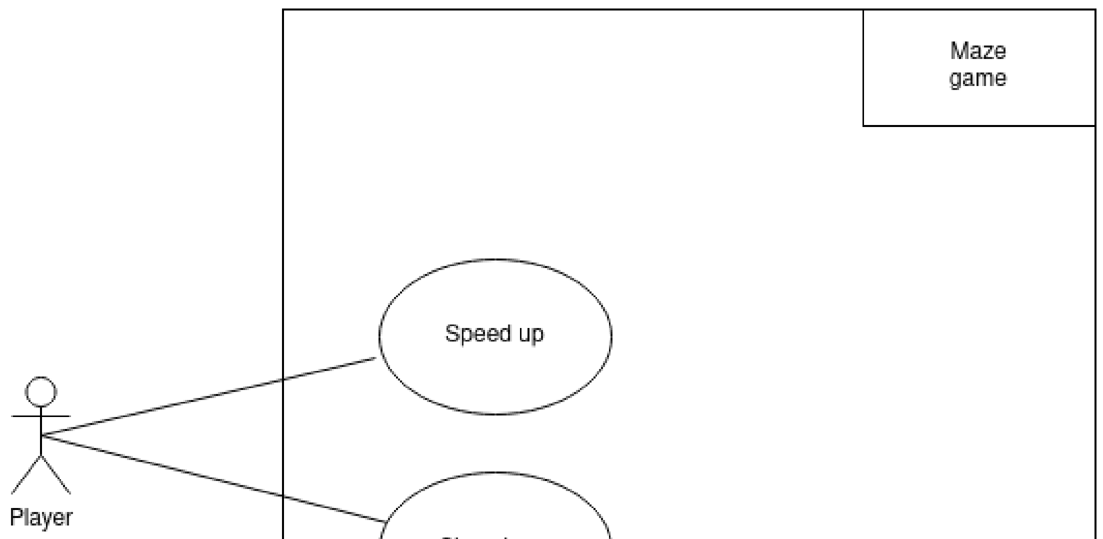


Figure 4: "Modify hero's speed" diagram

When the hero moves, he goes at a certain speed which can be increased or decreased.

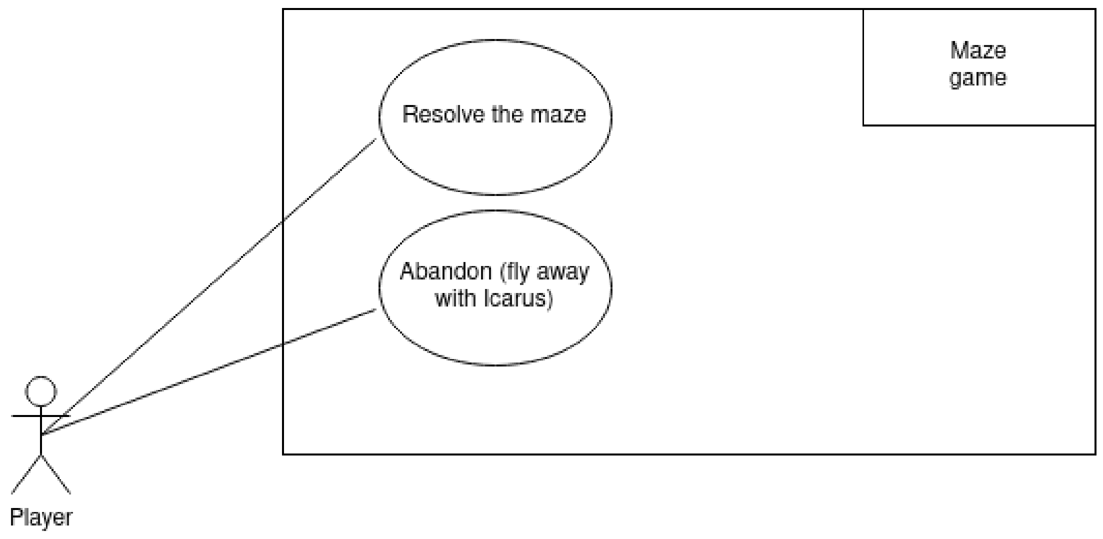


Figure 5: "Stop the game" diagram

To exit the application, the user solves the maze or presses Esc.

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

B. Associated functional tests

Some features of the program must be tested via unit tests.

For the maze generation functionality
- 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 which should be separate from the corners.

For the display functionality of the labyrinth:
- 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 (Graphical User Interface)

A. Scenarios

In this section, the system's responses to the user's actions are 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	

An example of scenario reading, which can be applied to others, is as follows:
- When the user presses the 1 key, the system should call the program generating the maze and the program displaying the generated maze.

- When the user presses the Escape key, the program should return to the home screen.

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	

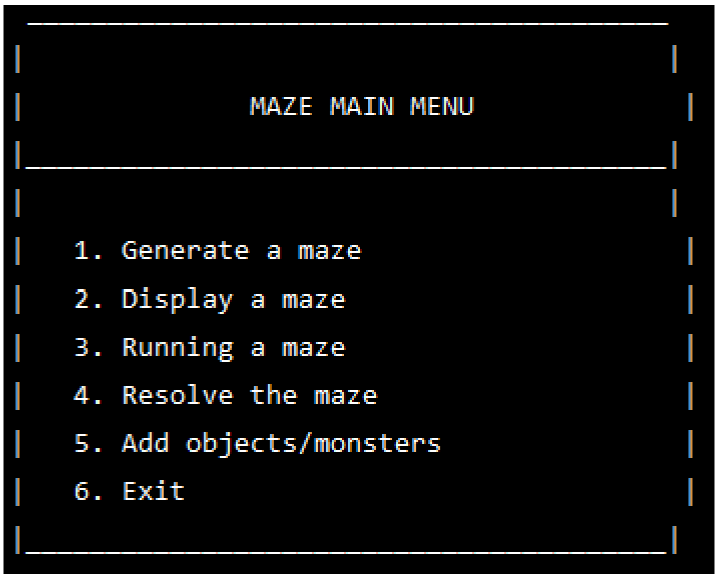
Use case	Resolve the maze	
Actor	User	
System	Maze Game in C	
Operations	System	User
1		Press “4”
2	Solve the maze using the algorithm	
3	Display the resolved maze	
4		Press “Escape”
5	Return to main menu	

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 report some graphical mock-ups of what the final interface will look like. These mock-ups are based on the different scenarios set up earlier.

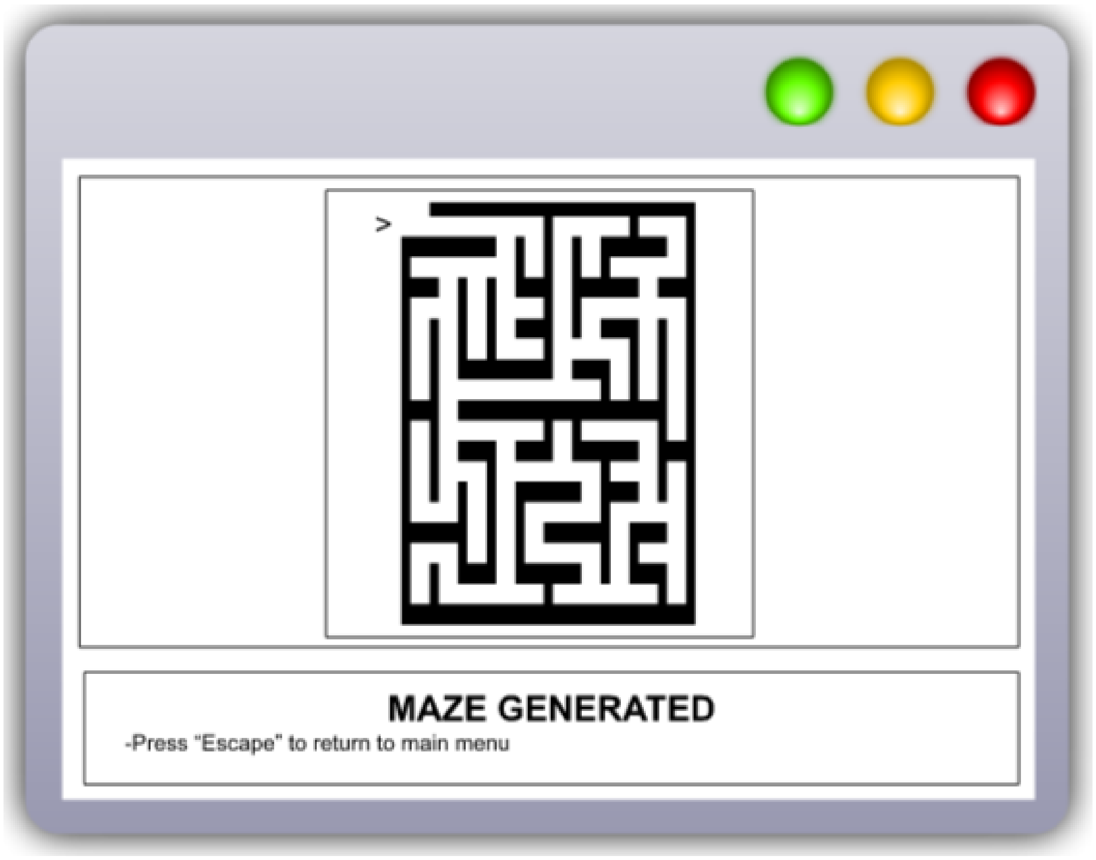
B. Mock-ups

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



```
 ________________________________________
|                                        |
|              MAZE MAIN MENU              |
|________________________________________|
|                                        |
|   1. Generate a maze                     |
|   2. Display a maze                      |
|   3. Running a maze                      |
|   4. Resolve the maze                    |
|   5. Add objects/monsters                |
|   6. Exit                                |
|________________________________________|
```

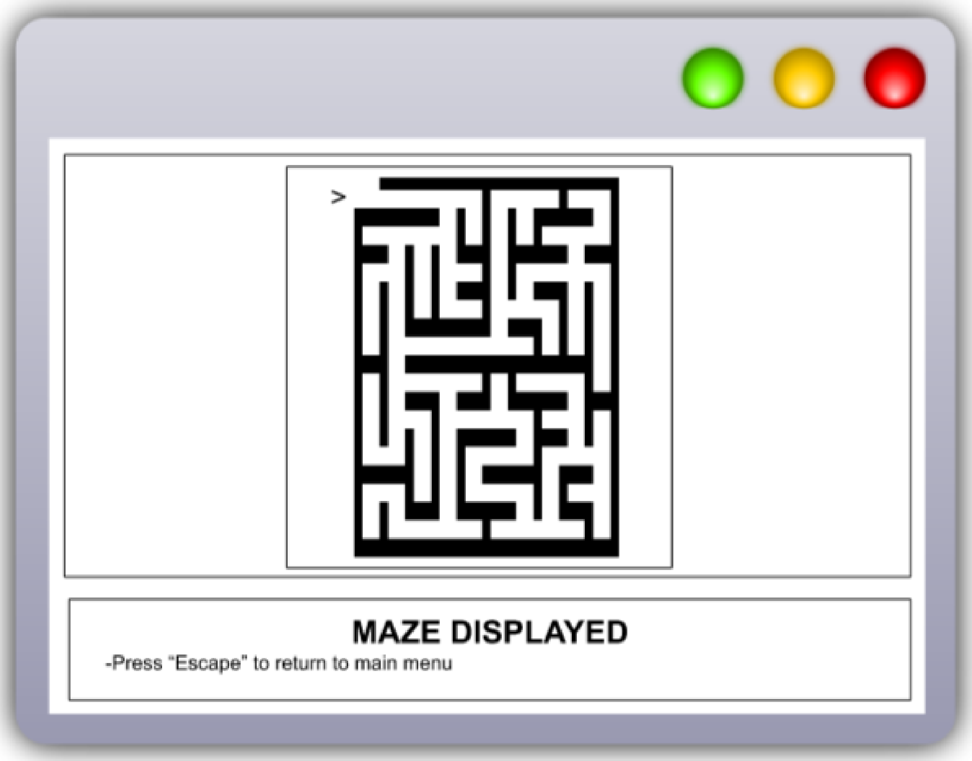
Figure 6: Game main menu mock-up



MAZE GENERATED

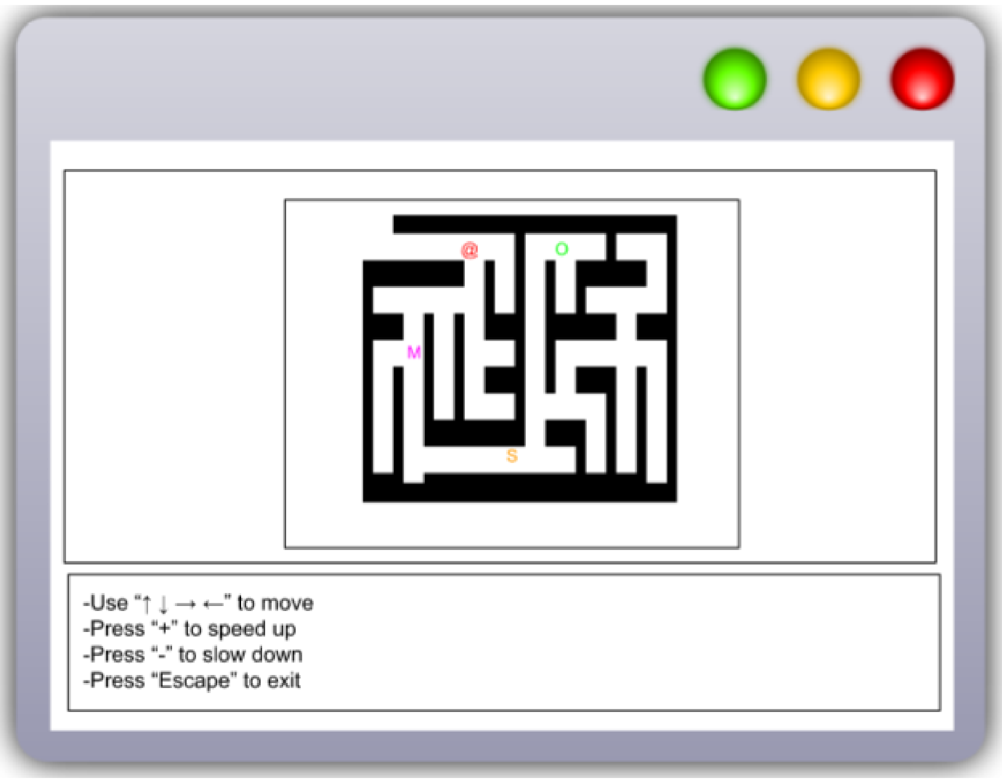
-Press "Escape" to return to main menu

Figure 7: "Generate a maze" mock-up



MAZE DISPLAYED
-Press “Escape” to return to main menu

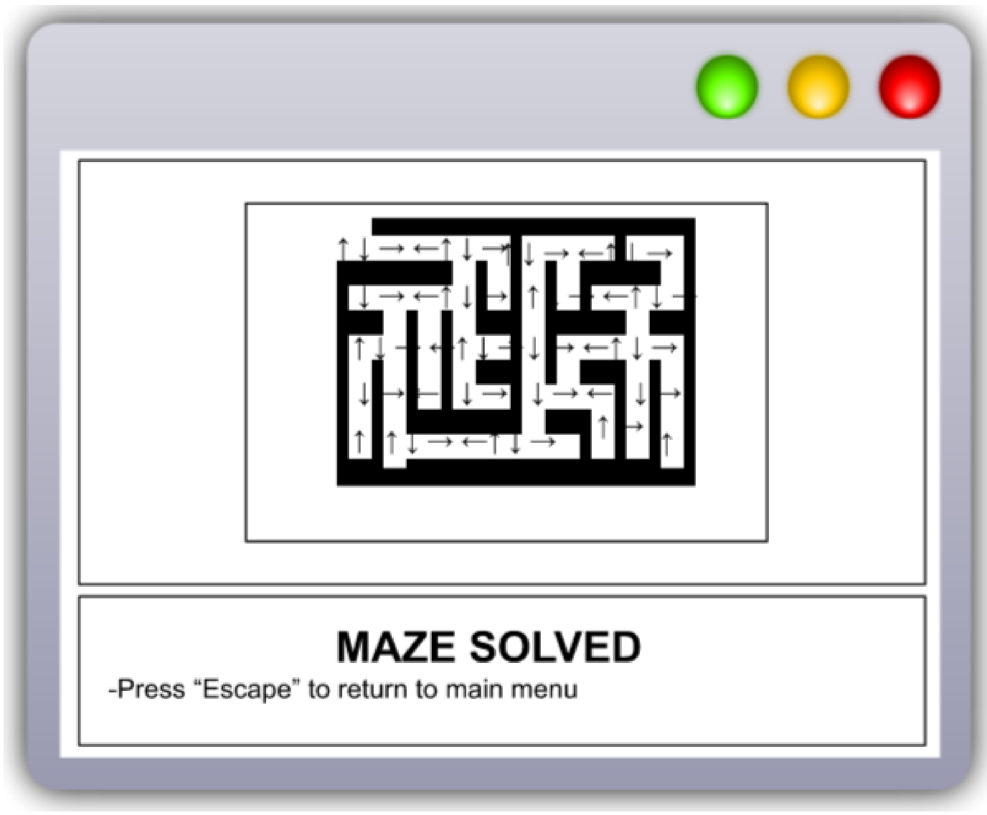
Figure 8: "Display the maze" mock-up



-Use “↑ ↓ → ←” to move
-Press “+” to speed up
-Press “-” to slow down
-Press “Escape” to exit

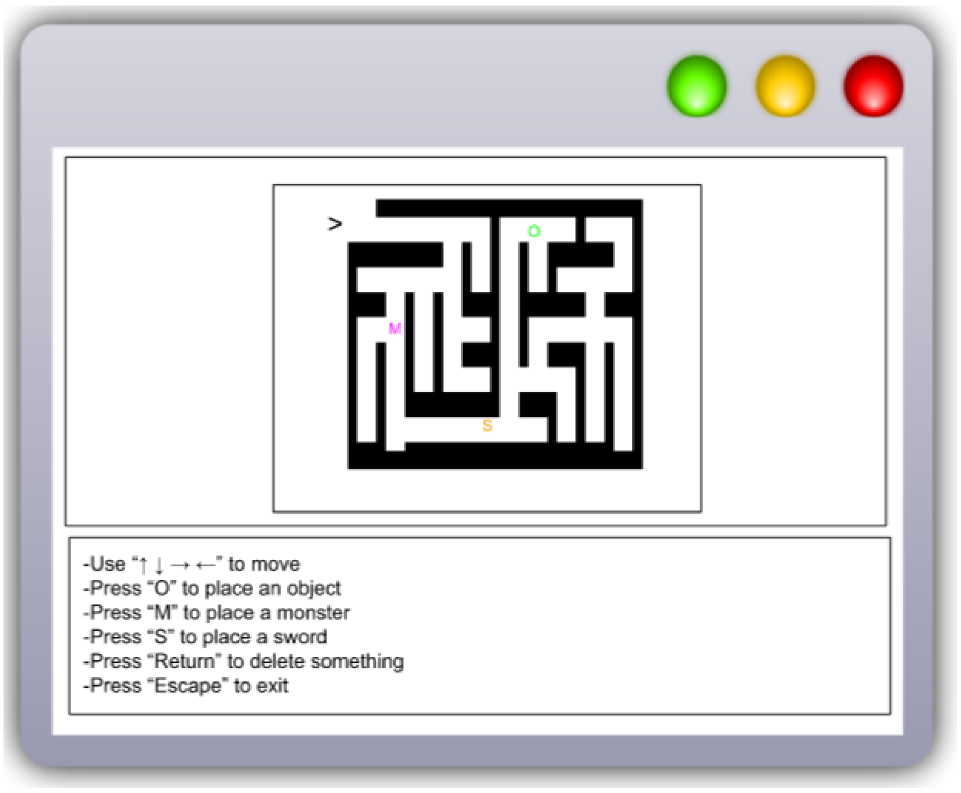
Figure 9: "Running a maze" mock-up

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



-Press "Escape" to return to main menu

Figure 10: "Resolve the maze" mock-up



-Use "↑ ↓ → ←" to move
-Press "O" to place an object
-Press "M" to place a monster
-Press "S" to place a sword
-Press "Return" to delete something
-Press "Escape" to exit

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 these 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 shown below:

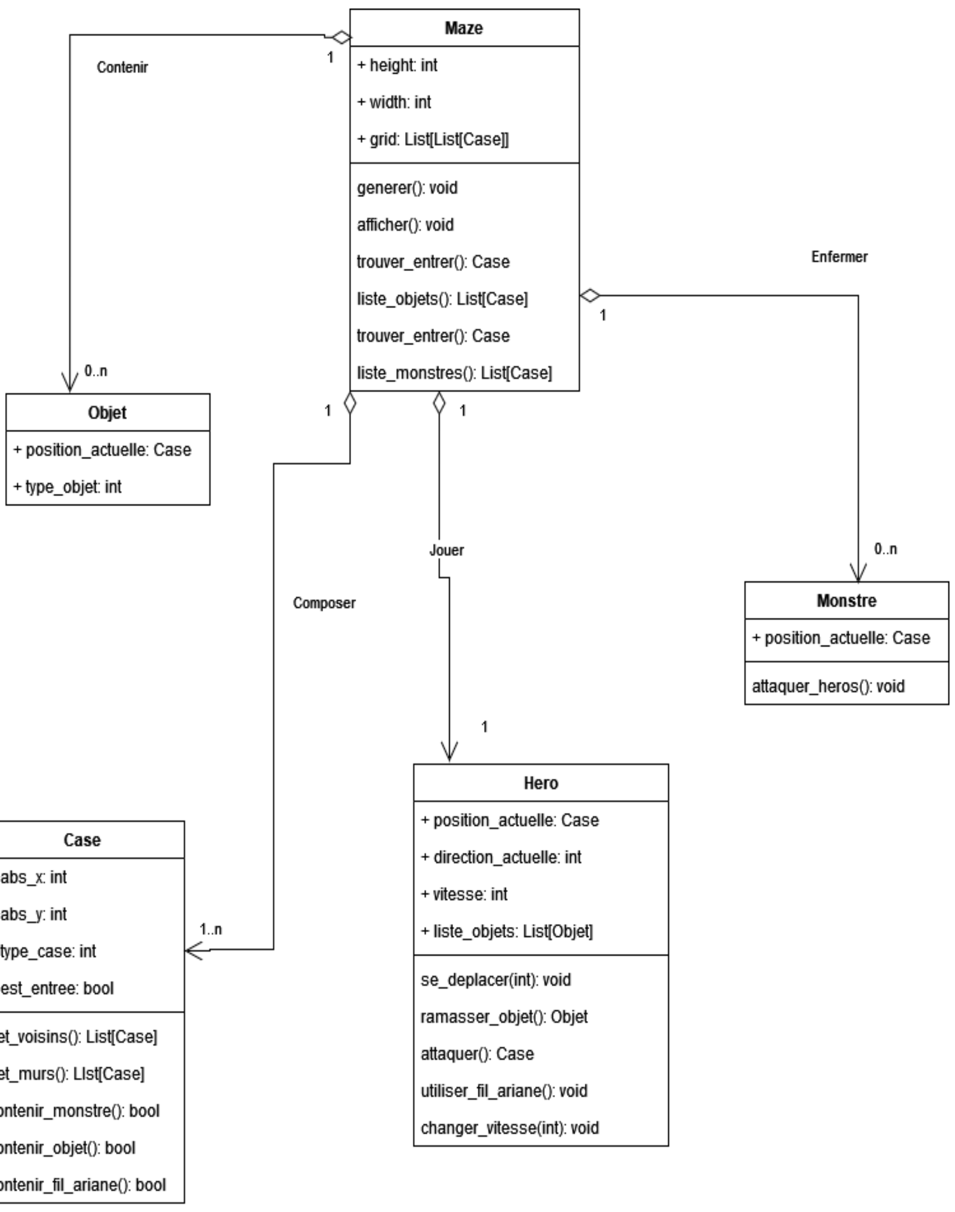


Figure 12: Class diagram of the system

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 is 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 are 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 are discussed in the next section.

D. Algorithms

One of the most important components of this project is to develop algorithms, to reduce the programming phase.

- Maze creation algorithm

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

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

- 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, a square is destroyed either the South wall or the East wall.

NB: It must be noted 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 width route of the tree (algorithm).

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

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

- A function to obtain a matrix containing the distance between each cell and 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 the programme development 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, the 1st / March, the 10th),
- final report (April, the 13th),
- code (April, the 13th).

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

10 March 2023:

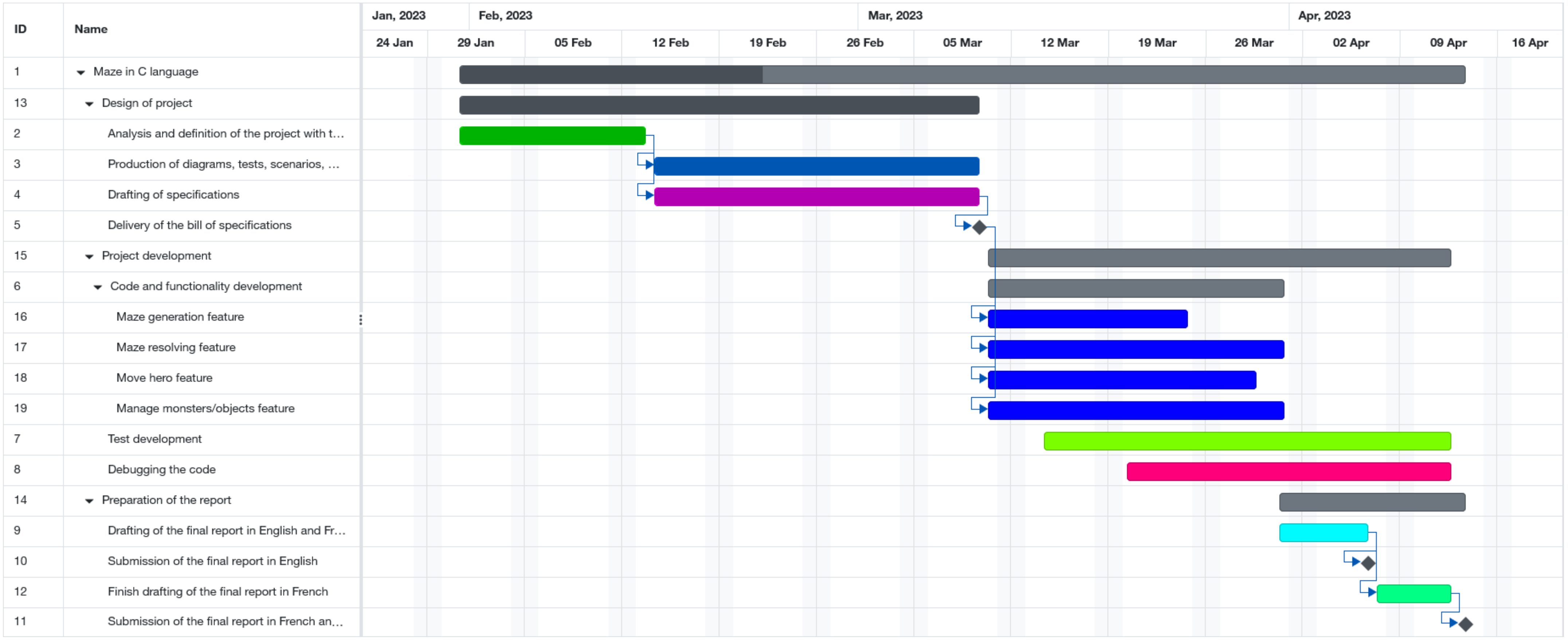
- **External milestone:** Submitting the specifications.
- **Internal milestone:** Re-reading to validate the specifications.

10 April 2023:

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

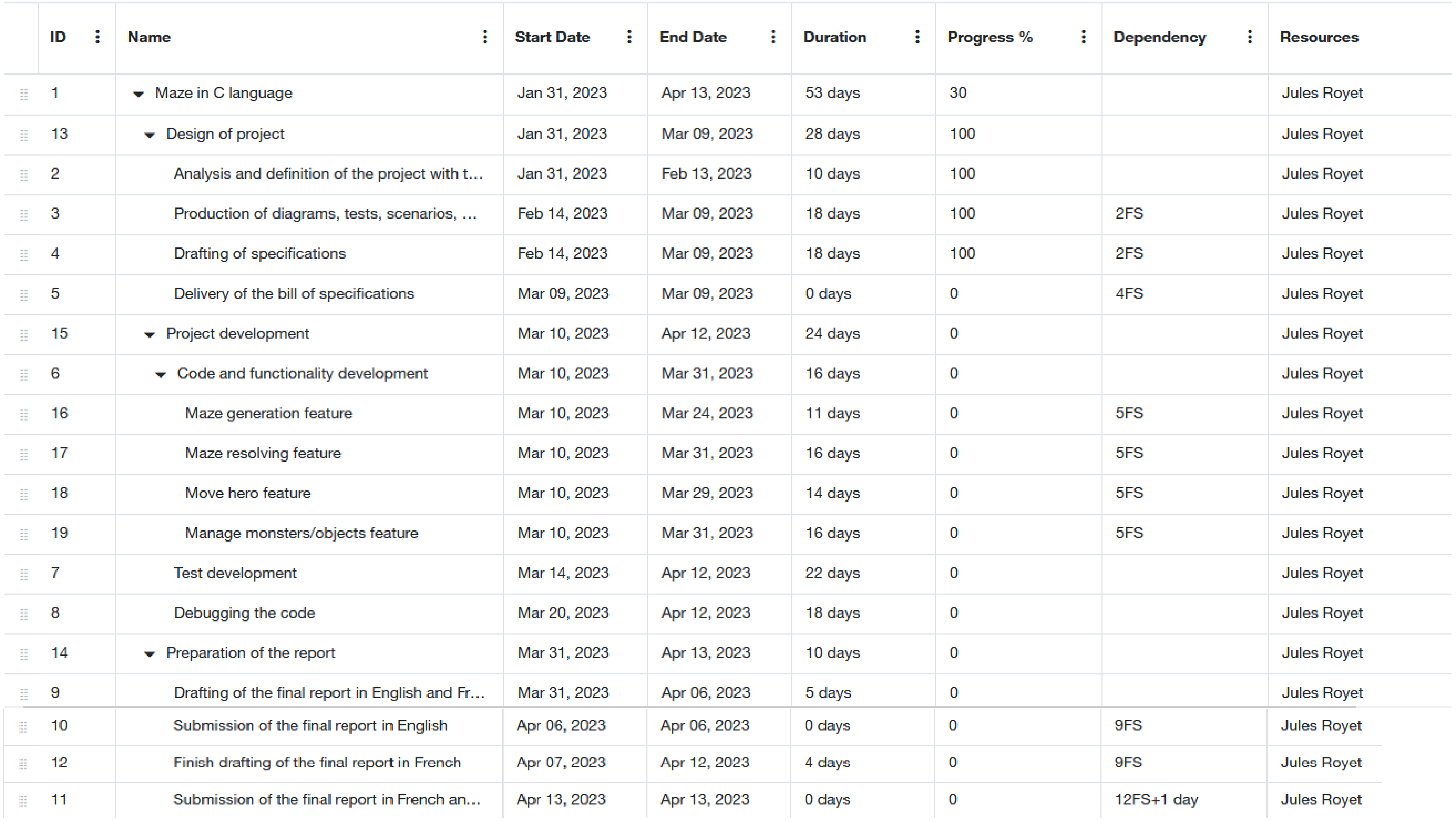
13 April 2023:

- **External milestone:** Submitting final report in French and the code.
- **Internal milestone:** Proofreading to validate the final report in French and verificating code works properly and is accessible.



ID	Name
1	Maze in C language
13	Design of project
2	Analysis and definition of the project with t…
3	Production of diagrams, tests, scenarios, …
4	Drafting of specifications
5	Delivery of the bill of specifications
15	Project development
6	Code and functionality development
16	Maze generation feature
17	Maze resolving feature
18	Move hero feature
19	Manage monsters/objects feature
7	Test development
8	Debugging the code
14	Preparation of the report
9	Drafting of the final report in English and Fr…
10	Submission of the final report in English
12	Finish drafting of the final report in French
11	Submission of the final report in French an…

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 produced.
- Visual Studio Code to produce the code in C.
- A Makefile 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 entities.
- Figma to produce the different mock-ups of the future C application.
- Discord will help us to communicate outside of class sessions.
- Google Drive to share the documents concerning the project.

Now that all the specifications are in place, let's look at the return on project.

7. Project feedback

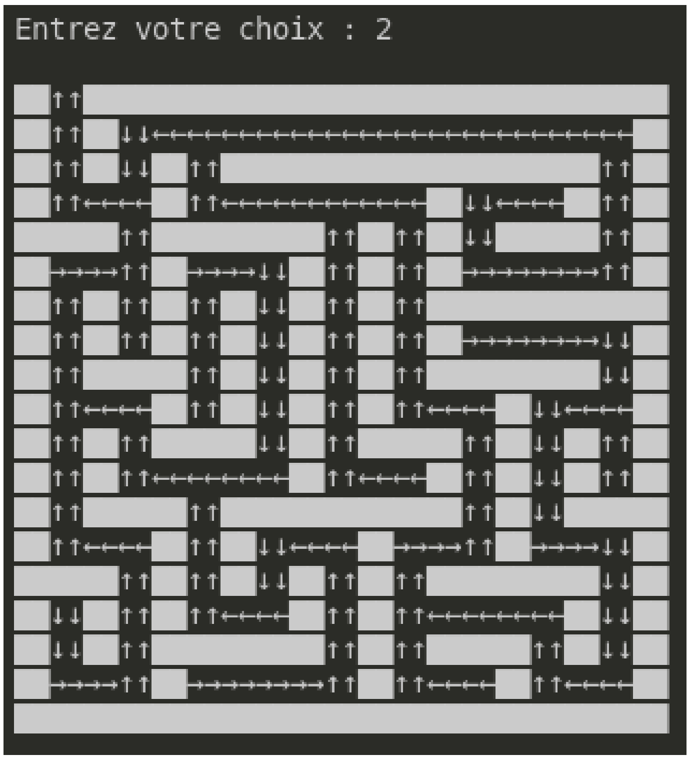
A. Fulfilment of project

Regarding the completion of the project, the expected functionalities have been developed, namely:
- the generation of random mazes
- display of mazes
- labyrinth solving

The optional functionalities have not been developed due to lack of time, namely:
- moving the hero through the maze
- the monster system
- the system of recoverable objects.

As for the respect of the specifications, the result is rather satisfactory, because the specifications were respected from start to finish. Indeed, the user interfaces designed resemble the models produced beforehand, just as the scenarios that include the user actions were implemented identically.

Here is the final rendering of the application with the example of a maze generated, solved and displayed:



```
Entrez votre choix : 2
```

Figure 15: Maze generated, solved and displayed

You can refer to figure 10 for comparison.

However, as previously mentioned, the functionality of moving the hero, monsters and objects has not been implemented. Although these are optional, they were considered in the deadline.

Furthermore, the code has been properly commented and documentation is available in the form of a small website in the project's "doc" folder.

Although the specifications were correctly met, it is important to look at the optimisation of the code.

B. Algorithmic complexity

As far as the algorithmic complexity of the code produced is concerned, it is 4n^3+n.

Indeed, such a complexity is obtained, because two loops are called in the function "digLaby" (to process all the cells), then a while loop is called in this function which, for each cell, is processed 4 times (to see if the cells are diggable in the 4 directions).
Also, there is a first loop which fills the maze with walls, hence the +n.

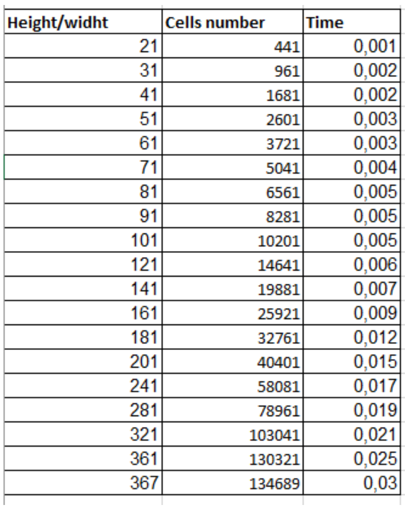
This complexity is not the best, on the other hand, it allows to have a simple and readable code and a very fast execution, if the maze generated is not too big as it will be shown in the next section.

C. Program benchmarks

It is important to specify that in this program, the generation and the resolution of the maze are done at the same time, the resolution is simply hidden from the user if he does not ask the program to show him.

Therefore, by using the time() function of Unix, a measure of the execution time for the generation and the resolution of the maze could be done, regarding the usr_time value at the end of execution.

19 maze sizes have been chosen to obtain an optimal result, knowing that these sizes must be divisible by 3. This gives the following result:



Height/widht	Cells number	Time
21	441	0,001
31	961	0,002
41	1681	0,002
51	2601	0,003
61	3721	0,003
71	5041	0,004
81	6561	0,005
91	8281	0,005
101	10201	0,005
121	14641	0,006
141	19881	0,007
161	25921	0,009
181	32761	0,012
201	40401	0,015
241	58081	0,017
281	78961	0,019
321	103041	0,021
361	130321	0,025
367	134689	0,03

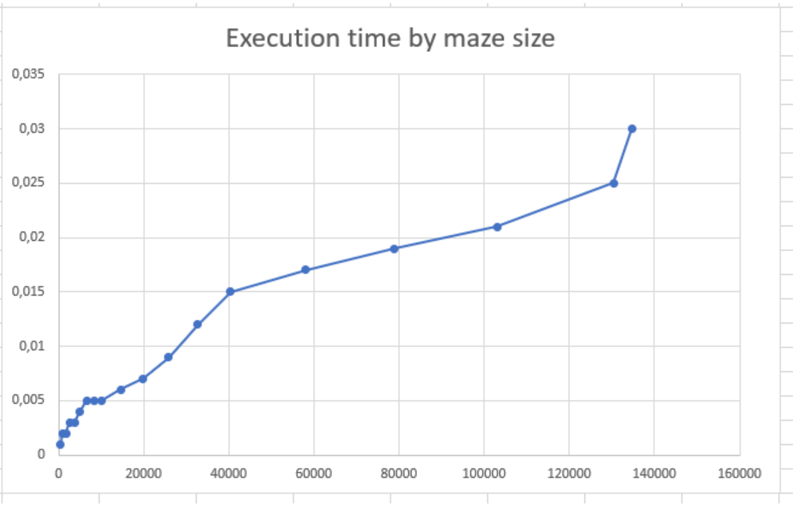


Figure 16: Graph of execution time versus maze size

It is observable that the program crashes at size 368 by 368. This is due to the maximum memory restriction of the dynamic allocation function malloc.

An important point in this project is also to discuss the various problems encountered.

D. Problem encountered & code smells

In the case of this program, the non-Linux compatibility is very bad. Indeed, after having tried to compile the program under Windows, it can be observed that the terminal does not interpret special characters such as the solid blocks constituting the walls of the labyrinth or the direction arrows.

Overall, the project was carried out in a rather clean manner. In addition to being properly architected and commented on, it is indented. However, some code smells could have been avoided here and there. Moreover, it would have been interesting to make our programme interactive with the notion of heroes moving around etc...

It is now time to conclude on the subject by outlining what has been retained from this project.

8. Conclusion

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

A game functionality detailed description, technical requirements and performance analyse have been included in the specifications as well as the time frame associated with the development of the game. Usability and user experience aspects have been considered to ensure that the game is enjoyable to play.
Now that this specification has been finalised, it can be used as a guide for developing 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 developing the code and producing the various manuals.

9. 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

10. 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 mock-up 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.

C language: low-level programming language which is not very expensive in terms of computer resources.

ASCII: *American Standard Code for Information Interchange* - Standardised code used for the exchange of computer data