UNIVERSITÉ PERPIGNAN VIA DOMITIA

Implementation Report

May, 2021

Creation and coding of a Ravensburger-like Labyrinth Game, using C language

Teacher in charge: ROSALIE Martin

Team: “Roman Empire” (made of REBIÈRE Marc and MEHAIBIA Rami)

Summary

Contents

1. What our game does first and foremost 3

1.1 How to play our game? 3

1.1.1 Before playing 3

1.1.2 The mechanics of our game 5

1.2 What makes our game different from the others? 8

1.2.1 Audio implementation! 8

1.2.2 Debug mode 8

1.2.3 Jukebox mode 9

1.2.4 Easter eggs 9

2. Discarded concepts 9

2.1 Age of the player 9

2.2 Vs computer 9

2.3 Unitary tests 10

3. Development phase and implementation of the code 10

3.1 Creation of the first textures 11

3.2 Beginning SDL 12

3.3 Encountering the first problem: Texture traces/tracks 13

3.4 Insertion texture decision 15

3.5 First matrix representations of the game board 15

3.6 Setting up the first iterations of movement and treasure get 19

3.6.1 Setting up movement rules 19

3.6.2 Allocating treasures to each player 19

3.6.3 Giving life to everything: The game loop 20

3.7 Insertion system and testing for more than 2 players 21

3.7.1 A nice (theoretical) plan 22

3.7.2 The (cruel) difference between reality and practice 22

3.7.3 Creating the insertion feature 22

3.8 Audio system nightmare 25

3.8.1 Enhancing the players' experience 25

3.9 Finishing touches 27

3.10 How our repository is organized 33

Conclusion 34

1. What our game does first and foremost

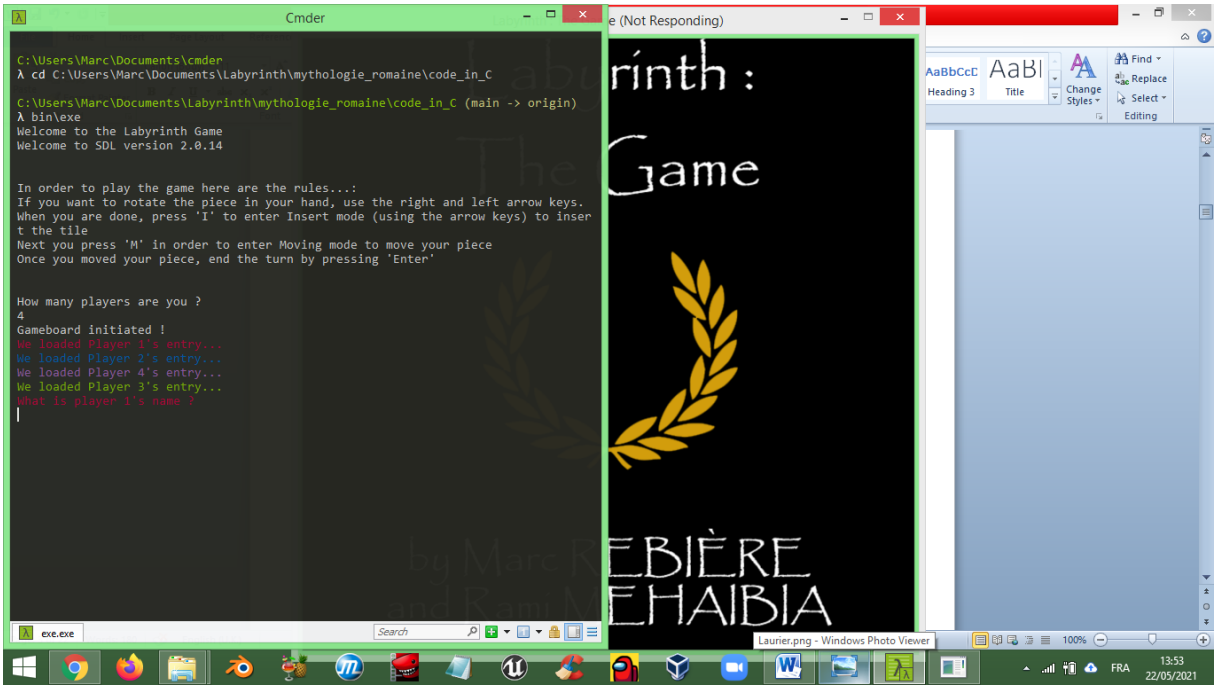
Well, be sure that it respects most of the constraints and objectives given. It does the job. So, let us begin with the actual mechanics of our Labyrinth Game. It has only one type of play: Multiplayer. The reason as to why we couldn't code a "computer mode" will come in a second. But first:

1.1 How to play our game?

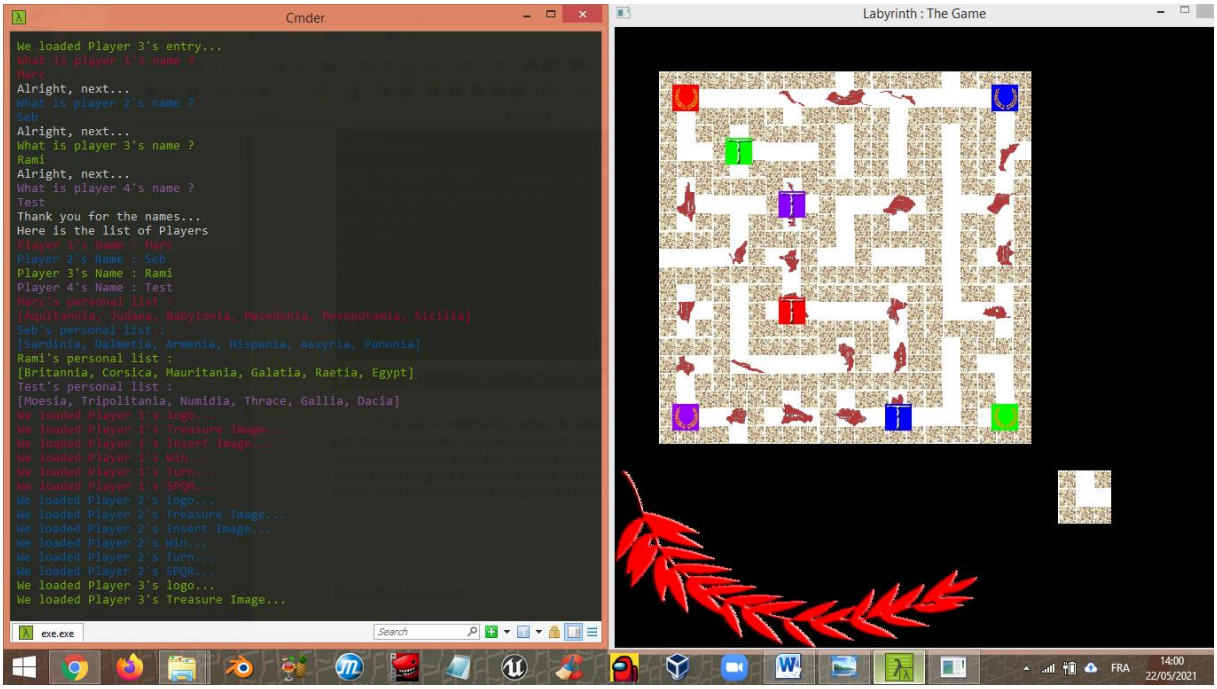
1.1.1 Before playing

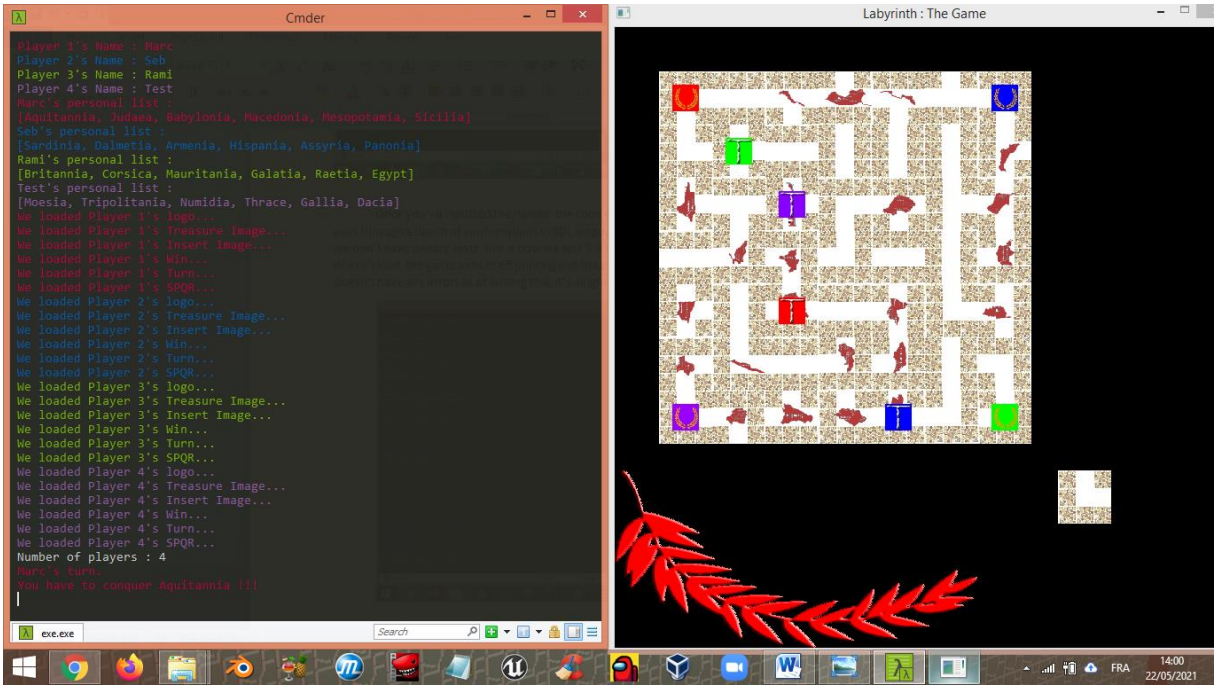
At the beginning, you are asked to input the number of players for the next game you want to play. You can only choose between 2, 3 or 4. If you put any other number it will ask again until you give a correct number.

After that, it will ask for the names of each person. A different colour will be given to each new player. The order will always remain the same: Red goes first, then Blue, then Green and finally Purple (they represent Players 1 through 4 respectively).



Once you've inputted the names, the code will load the players' texture and the game as well. It will pass through a bunch of confirmations in SDL language that the loading of a texture worked (since we don't have unitary tests, this is how we test if it actually works before appearing on the screen). In the eventuality that some texture doesn't load properly, the game exits itself printing out the error and its location in the code (but since it doesn't have any known errors as of writing the present report, it is fine).





Once everything is loaded, you can now play the game and enjoy it!

1.1.2 The mechanics of our game

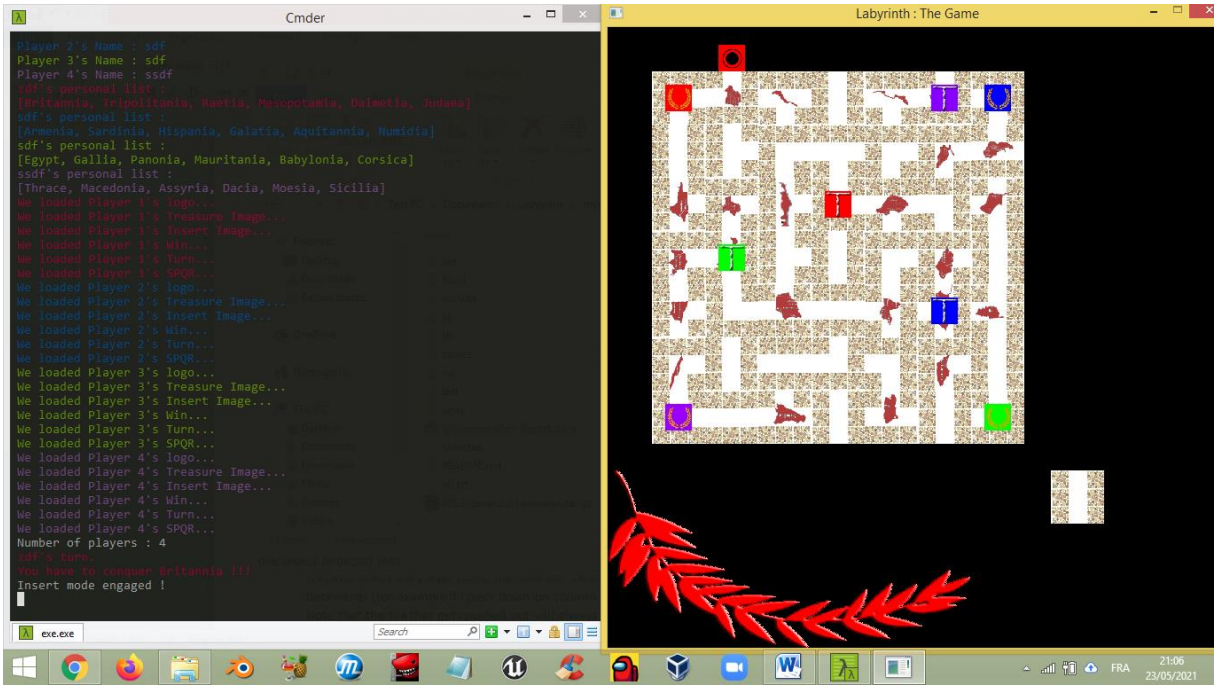
Each player appears on a different corner of the labyrinth board as a coloured square containing a laurel wreath. All the “treasures” to collect appear as the reddish silhouettes of Roman provinces to conquer, squattered randomly within the corridors of the labyrinth.

After this small setting up it is Player 1’s turn. By default, each player is in “rotating the piece” mode, where you rotate the outside piece. To do that you just press the right or left arrow keys. By default the Outside tile is either in the basic I shape, L shape or T shape.

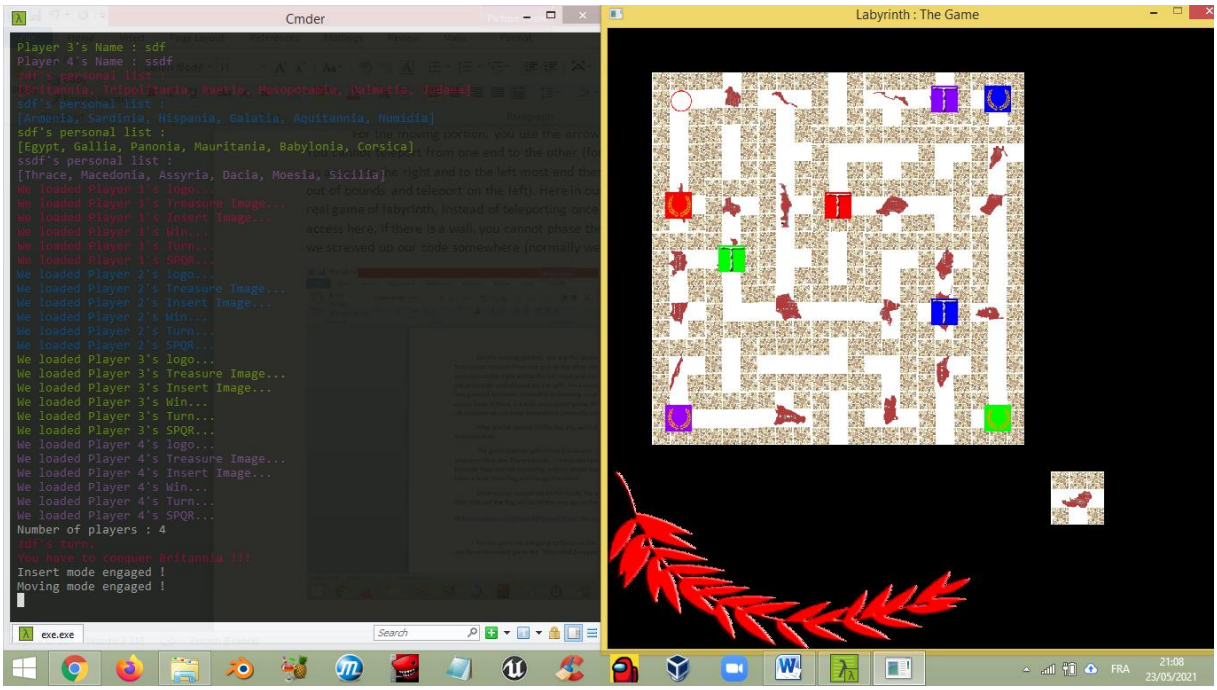
Once you choose your rotation, you can enter the “INSERT” mode by pressing the “**I**” key on the keyboard. A sort of cursor will be automatically set up on the top left of the game board which will bear the respective player’s colour.

In order to choose where you want to insert your tile, press the left and right arrow keys accordingly. When you have decided where to insert the tile, press the “**M**” key in order to enter “MOVE” mode.

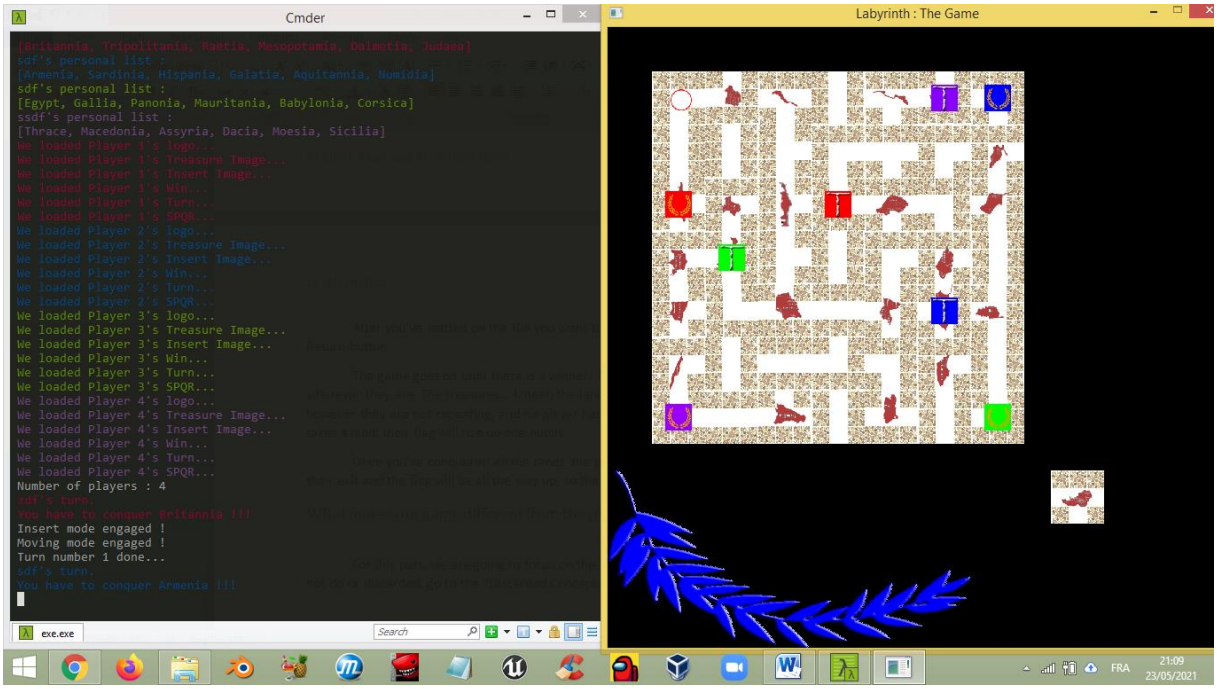
But here are some little titbits before moving to the moving section: If a player is pushed outside the boundaries of the game, no worries because he or she will reappear at the opposite end of where his or her pawn was pushed out; if a treasure is pushed out, you can still see the treasure on the outside tile; and finally the player doing the insertion cannot replicate the same move as the previous player backwards (for example if I push down on column 3, the next player cannot push up on column 3). Note that the tile that gets pushed out will always return to its basic shape (example: if we push out an L-shape oriented towards the right, it will be shown outside as a normal L-shape).



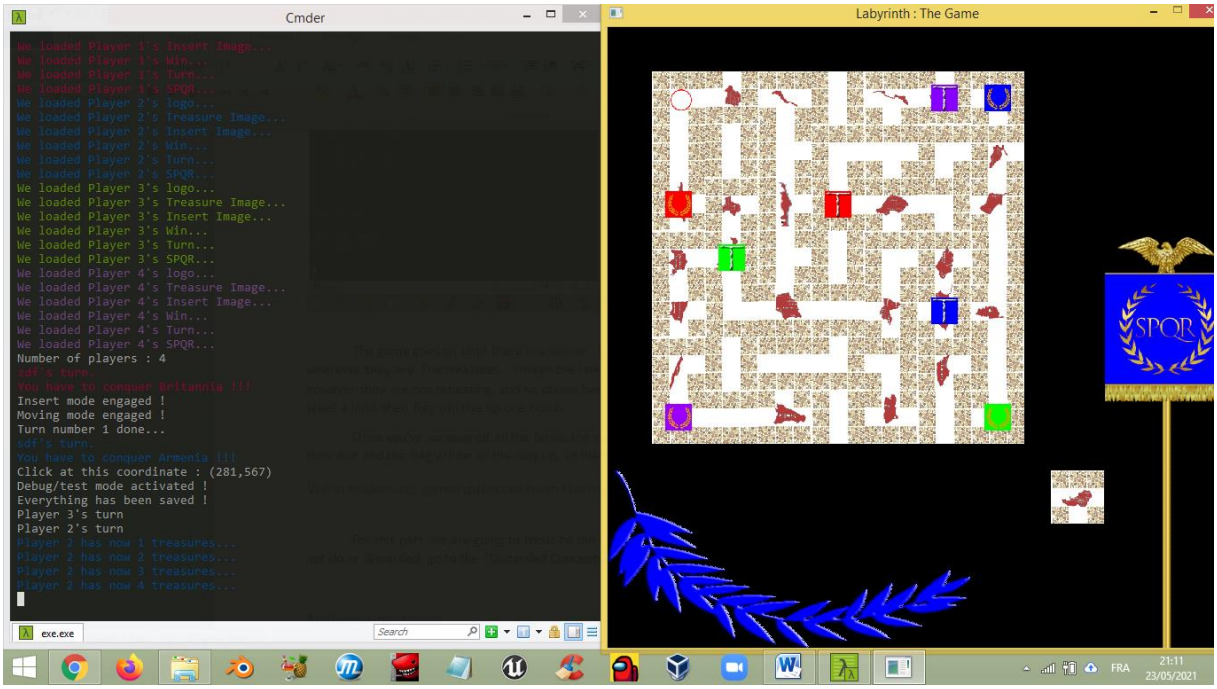
For the moving part, that is to say how each player can move its pawn along the corridors of the labyrinth, you use the arrow keys to navigate inside the map where you are allowed to. You may not teleport from one end to the other (for example if you are on a rightmost end tile with an access to the right and to the leftmost end there is an access to the left, you cannot pass by going out of bounds and teleport on the left). Here in our game you get to move the pieces, just like in a real game of labyrinth, instead of teleporting once there is an access to a treasure (like our teacher proposed, if there is an access to the treasure, we teleport there, if there are none, we stay where we are). It is all about access: if there is a wall, you cannot phase through it magically. But if you can, then that means that we screwed up our code somewhere (normally we did not).



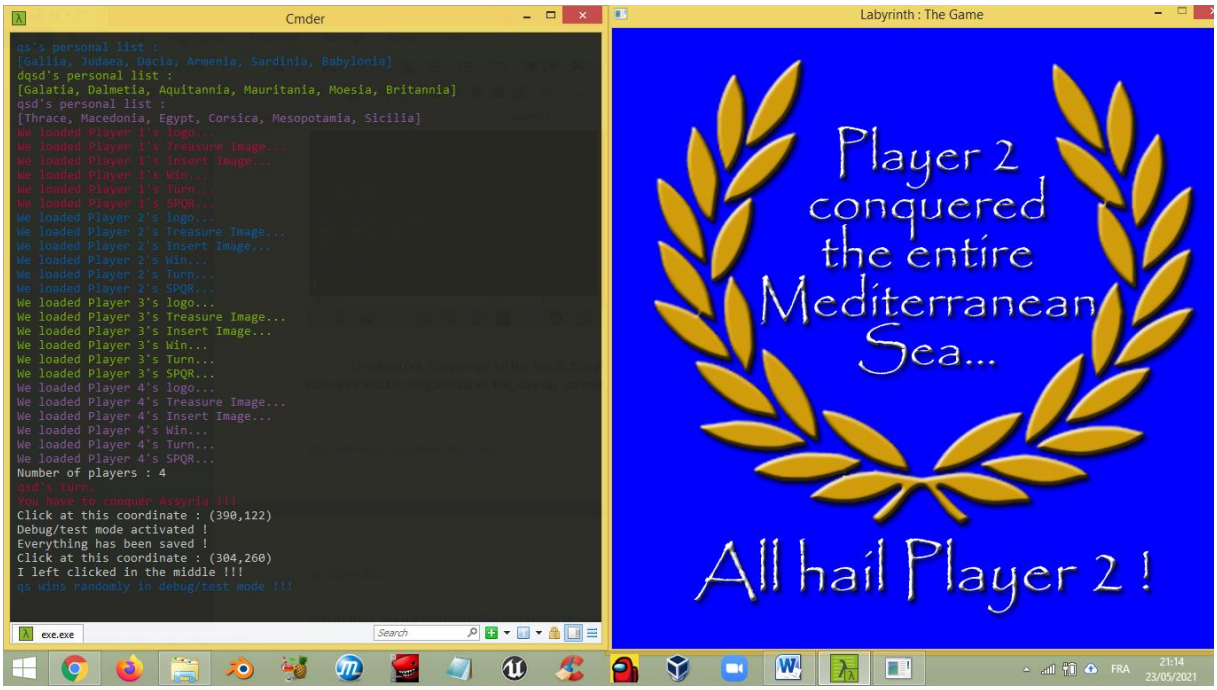
After you have settled on the tile you want to remain, you can end the turn by pressing the "**Return**" key.



The game goes on until the code detects that there is a winner. The treasures' positions and tolls will always update and they will show up wherever they are at each turn of play. The treasures… I mean the lands to conquer, are given at random to each player. However, they are not repeated, and no player has the same land to conquer. Also, when a player takes a land, his or her flag shown on the right side of the screen will rise up one notch. This little visual improvement will allow all the participants to quickly understand everyone's rank in the race at hand.



Once a player has conquered all the lands allocated to him or her, the player's respective treasure icon will appear on their exit and the respective flag will be all the way up, so that the player in question can go back home and tell his or her tales.



1.2 What makes our game different from the others?

For this part, we are going to focus on the things that we added, for the things that we could not implement or discarded from the code, go to the "Discarded Concepts" section.

1.2.1 Audio implementation!

I think we might be the only group who implemented that… If there are others, shame on us for not knowing. But here is the thing: all the audio was made manually, by yours truly.

For that, we used a nifty piece of software called "Famitracker" which is a NES music composer. In it we have created all the sound effects (that you can also find in the folder sfx, but that would be cheating).

It took one night without sleep to understand how SDL incorporates audio, but in the end we managed to put it in... Hopefully you will enjoy this feature to make the adventure even grander!

1.2.2 Debug mode

If you press the "**D**" key, you get to access our little **Debug mode**, which was used to check the flags rising up and the good implementation of the turn-based system (represented by the coloured laurels and the rising flag of each player).

When you are in Debug mode, all the game data is saved so that you may go back once you are done playing around with the flags or turns. Here are the tweaking keys:

- To raise or lower the current player's flag, press + or - on your number pad.
- To change the turn press * or / on the number pad.
- To go back to normal play, press D again.

1.2.3 Jukebox mode

It's exactly how it sounds like! A whole jukebox to play your favourite sound effects in the game, because... *why not*? To access this mode, press the "**J**" key.

Once in **Jukebox mode** you can press the numbers 0 through 9 and also the spacebar as well to listen to the magnificent and crisp audio to get that retro feel (Wow).

When you are done, just press "**J**" again.

1.2.4 Easter eggs

You thought that in this implementation report we were going to talk about every little thing? Well no, and that is where you come into play… Find out the other secrets in the game, or you know… just read the code and then try it out!

2. Discarded concepts

Now let us proceed to the discarded concepts from the analysis report up to now. From what we have gathered, there are 3 main things that we have not implemented.

2.1 Age of the player

In the end, it would have been too complicated to reorganize each and every time the players when they have put their names and ages, so we just abandoned the idea altogether. Instead, it is: "first come first served".

2.2 Vs computer

As you saw, we have a solid movement system in place, and that very same system is also the thing that does not let us implement a computer into our game. Why is that? Because in order to obtain this feature, we have to give random movements to a computer. That might be surely possible to code, but the main question is how long does the computer do that? Plus it may glitch or something worse, so the final result would have been disastrous. Instead we chose to develop something that would make our group stand out (hence the audio system). By the way, the choice of

implementing audio effects and music was at the core of our development drive after hearing that SDL was capable of handling this feature.

2.3 Unitary tests

Alright… The main discarded concept… Here is the thing: the SDL library provides functions that signal when the error occurred. In addition we always have "***if***" statements implemented to test out that if a function returns **NULL** or **1** (typically the error return values) we would print the area of the code where we had the problem, and we would exit the program. As for why we could not do it with "*Unity*", well it's because we would have to test out each function all over again, which we already test anyway whilst building the game. For you see, if the game crashes, we would know where and how it crashed. But since now it does not crash, that means that everything is going according to plan. Therefore we have decided that it would not really be of any use to test things that we already test while creating the game.

In conclusion we can say that we have discarded a few concepts along the way but we have basically applied the majority of what we wanted to do in our analysis report, and some more things along the way.

3. Development phase and implementation of the code

Now, for the moment you have been all waiting for… The DEVELOPMENT PHASE!

Let me tell you: it was hard, we worked 5 days straight on this one, at least 10 hours each day, and one night without sleep plus other nights where we worked until 4 in the morning. Those 5 days were true hell on Earth, from Saturday 24th of April to Wednesday 28th, for the main concepts, textures, sound effects, code implementation, trials and error and everything in between.

As of writing this, we worked a bit here and there to polish the final beta a little more, but nowhere near the amount of work that we did in one sitting during that period. Even though I remember the dates of work, I do not remember what we did on each day with great precision . However I secured dozens of print-screen images of the development in the chronological order of what we did.

Therefore this part will be cut in several subsections in order to explain how we did things (visually speaking, instead of showing the code and leaving you to decrypt what we did). Also as a quick mention, in the end we coded 4 C files for this project:

1. **audio.c** which has one audio function that we'll describe later on,
2. **begin.c** which has functions that take care of creating the players,
3. **middle.c** which has functions that take care of creating the game board,
4. and finally **main.c** which is the code that has the most number of lines… more than 2000 to be exact. Why is that? Because essentially everything happens in there:
    a. the creation of the application,

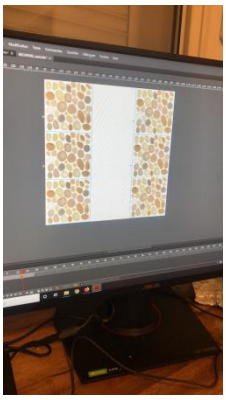
b. the building of the textures,
c. **the game loop**,
d. the freeing of everything after you are done playing.

Everything is done there in one go. It is no joke to say that we stayed a long time doing this game.

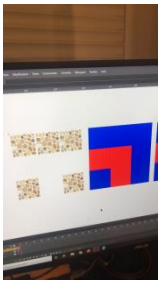
Now we can get onto the development phase!

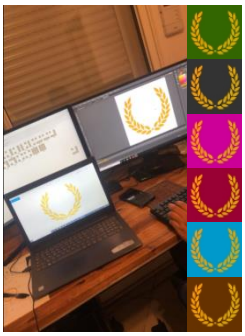
3.1 Creation of the first textures



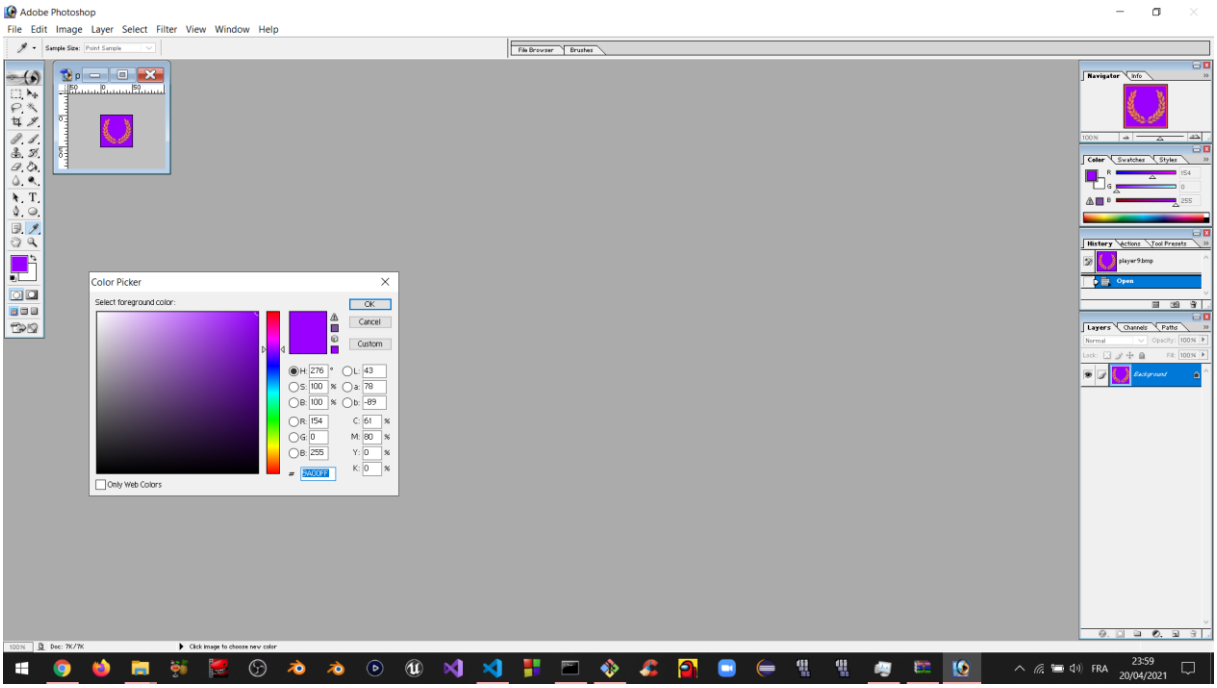


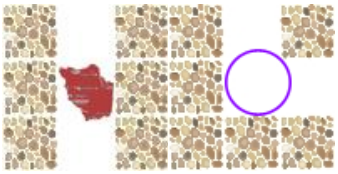
We had a pretty slow start to begin the project but in the end we began to think up our game design first and foremost. We went through multiple concepts for the tiles before we decided on a final design (which was in the end something reassembling a paved antique road). We even had one that we lost, which was basically a T-shape with the walking path represented by a red carpet with yellow limits and the walls were purple. Note that the majority of the textures were done by Rami while I was figuring out how SDL worked later on.





As for the players it was almost instantaneous: we knew that we had to represent the player as a “Roman Emperor” while still keeping the design simple and basic. There are still some unused player colours though, because we chose multiple colours to test out which were eye-catching. In the end we chose to go for the classic RGB with my little touch of Purple for the fourth player.





After that, we moved onto the special tiles: the “land to conquer” tiles and the “exit” tiles. Since we did not know that I-shaped tiles couldn’t have treasures we made 24 more than expected and are still in the folders, and there are therefore unused assets. As you can see, all the tiles are represented by default in their default shape, because we change their orientation in-game, thus minimizing the amount of sprites we need to create.

3.2 Beginning SDL

At the same time, I was beginning to figure out and learn how SDL worked. I understood several things:

1. How to initialize SDL
2. How to create a window and a render
3. How to set up a colour
4. How to draw a point
5. How to draw a line
6. How to draw a rectangle (and at the same time how to make one)
7. How to close a render and free everything
8. How to check for errors
9. How to present the render (because you have to present it every time you make a change or when you add something at some coordinate it doesn’t magically show up, you have to present it after the change)
10. How to delay the render
11. How to make a game loop (even though there was no game yet)
12. And finally, how to import a BMP file in the render (and yes, only bmp files work apparently)

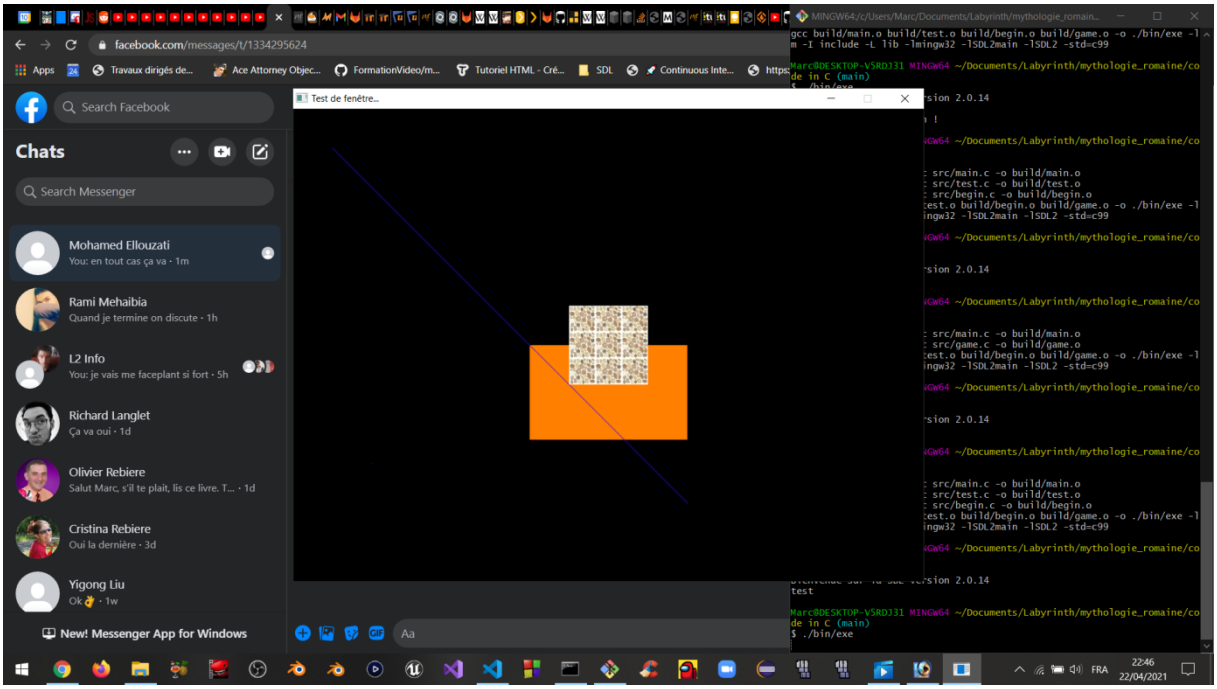
To actually import a **bmp file** on your render you have to first create a rectangle. The SDL rectangle is a **struct** that has at least four main variables to know by heart: **x**, **y**, **w** and **h**.

- **w** and **h** stand for width and height and we will cover them in a second.
- **x** and **y** are the coordinates but they begin from the top left of the render window thing. So if you put x=0 and y=0 well your image will be rendered on the top left but not centred, but by its top left corner as well.

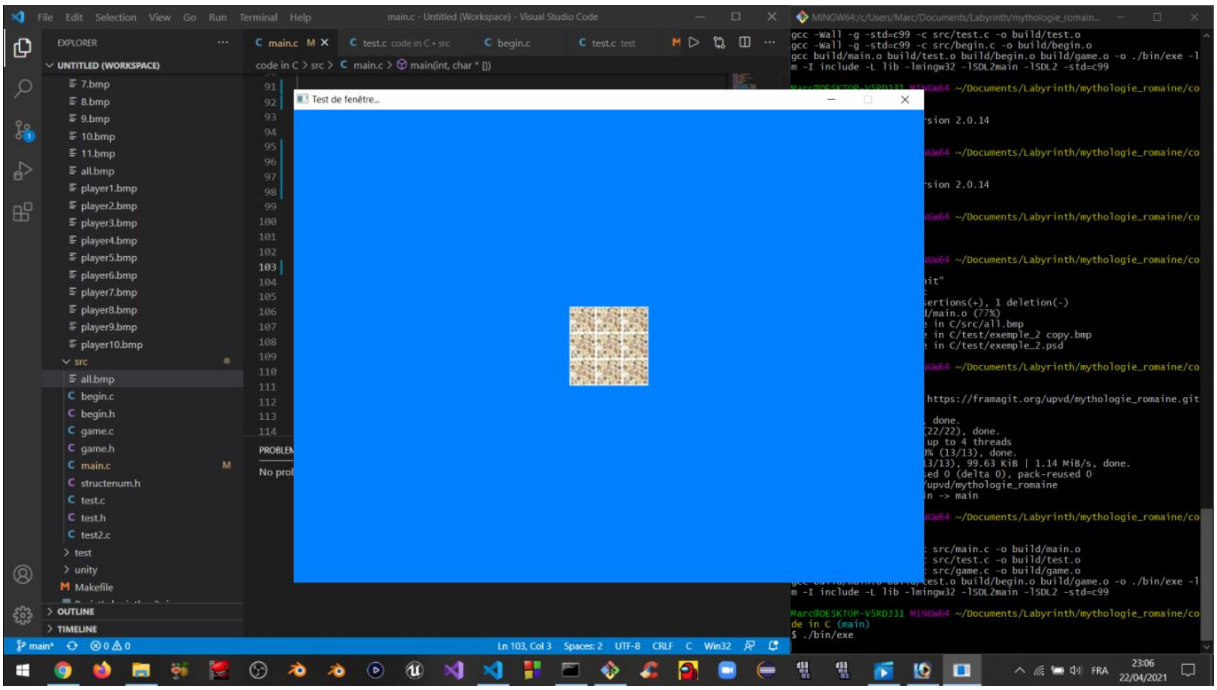
After you have made your rectangle, you need to create a surface and a texture. A surface is essentially a **struct** that will let you import the **bmp file** and the texture is going to take the said surface so that we can manipulate it as we wish. After we have attributed our surface to the texture, we do not need the surface anymore. The texture will then pass onto the rectangle with its width and height attributed with a function (when you see the code you will figure it out) and “*voilà !*”: we can now manipulate the rectangle! Now I know, this is just the tip of the iceberg but that is how it works in broad terms.

After messing around a bit with the functions (first image), we decided to have a fresh start and a blue background (second image). By the way, in SDL, you can also rename the window (we just

named it “Test de fenêtre…” because we had no inspiration at the time, and later named it “Labyrinth: the Game”).



Also you see... **game.c** was the file where we tried to make a functioning labyrinth on the terminal. In the end we just abandoned it and deleted it further down the line.



3.3 Encountering the first problem: Texture traces/tracks

After having succeeded in importing the first test texture (void of logic), now was the time to get started onto the great battle: (actually) MAKING THE GAME!

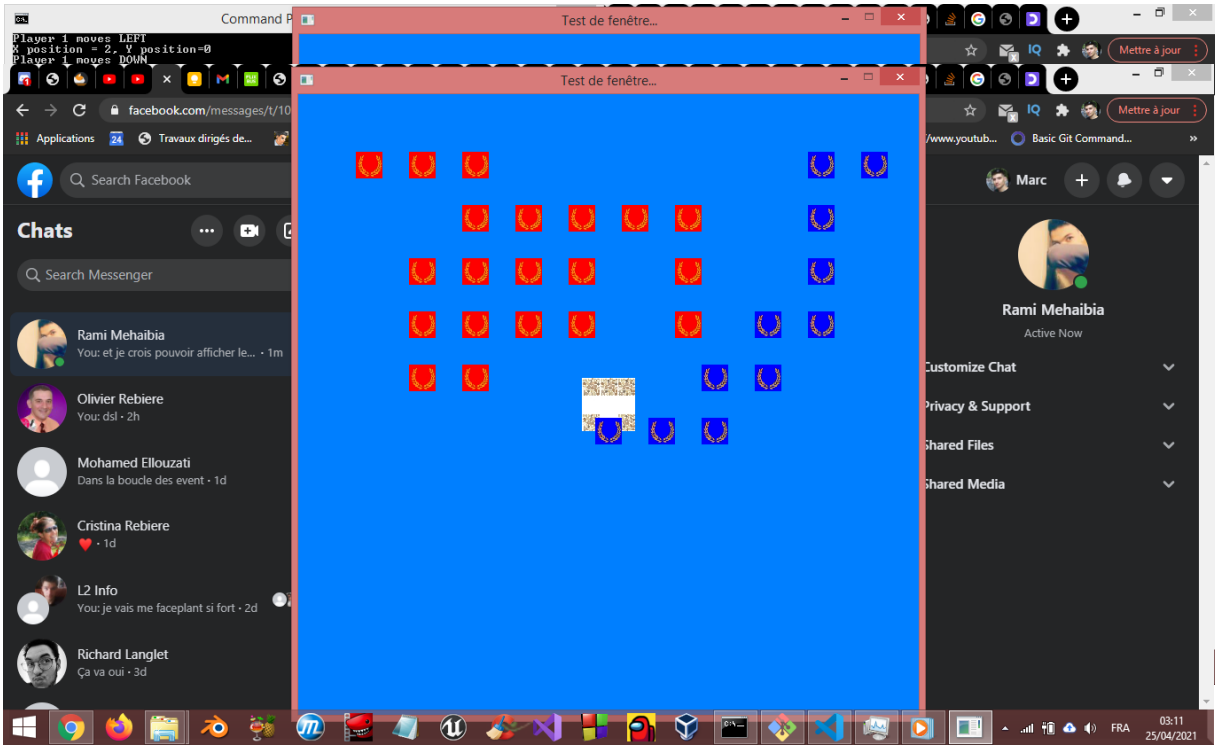
First and foremost we had to implement the players (and the amount of them would depend on how many you wanted between 2 and 4). After that we had to make them move on the game board somehow, and with limitations (and yes even though there was no game board present at that time). Therefore the first rule that the player must have automatically is: *Do not move more than 7 spaces (please)*. And you see, each player (which is also a struct by the way) had a position:

- Player 1 would spawn at (0,0) (but then converted into the coordinate system of the render),
- Player 2 at (6,0) (because x,y and not i,j), and so on and so forth.

You see, the player would not be in the actual matrix of the game board. The player “hovers” over the game board with a coordinate system compatible with the matrix! Now, we have set the basic limits of the player to not move more than 6 or less than 0. That was nice and good…

But then, we would hit the main problem:

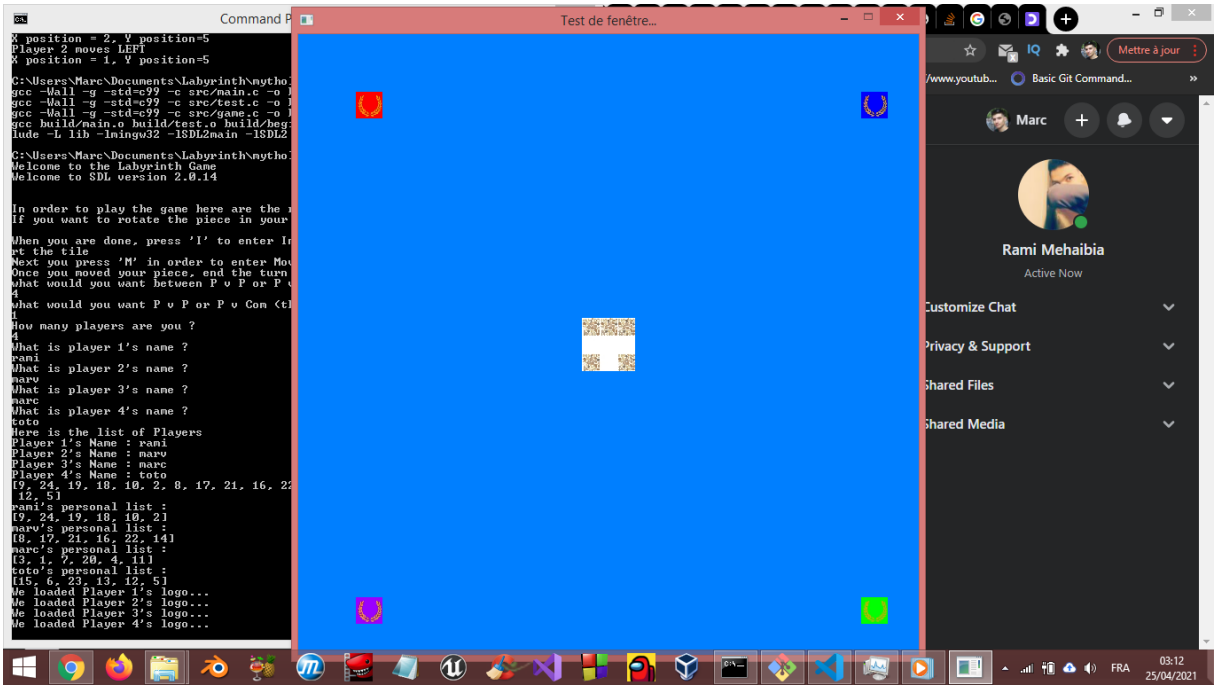
Before:



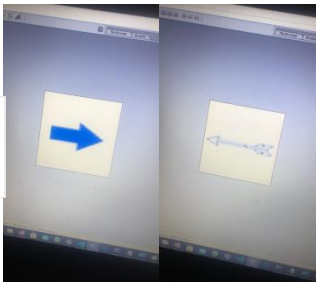
After: (also do not notice, please, the fact that I set up the limits wrongly)

So you see… the problem was that the render was like a piece of paper: you draw things but they do not magically go away: you have to present a new paper background taking into account the modification, and therefore render all the textures again in order (which was doable since we were able to make the game).

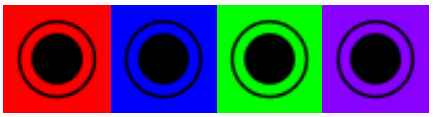
And that is what we did later on. Also below you may find an image of 4 players at once in the early… game I guess?



3.4 Insertion texture decision



Before we could render a huge matrix, we also had to settle on which insert texture we wanted to implement. For this purpose, we made several of them, but ended up choosing a sort of bullet point, so that we do not have to bother with where the arrow would point towards! Sometimes “keep it simple” is key!

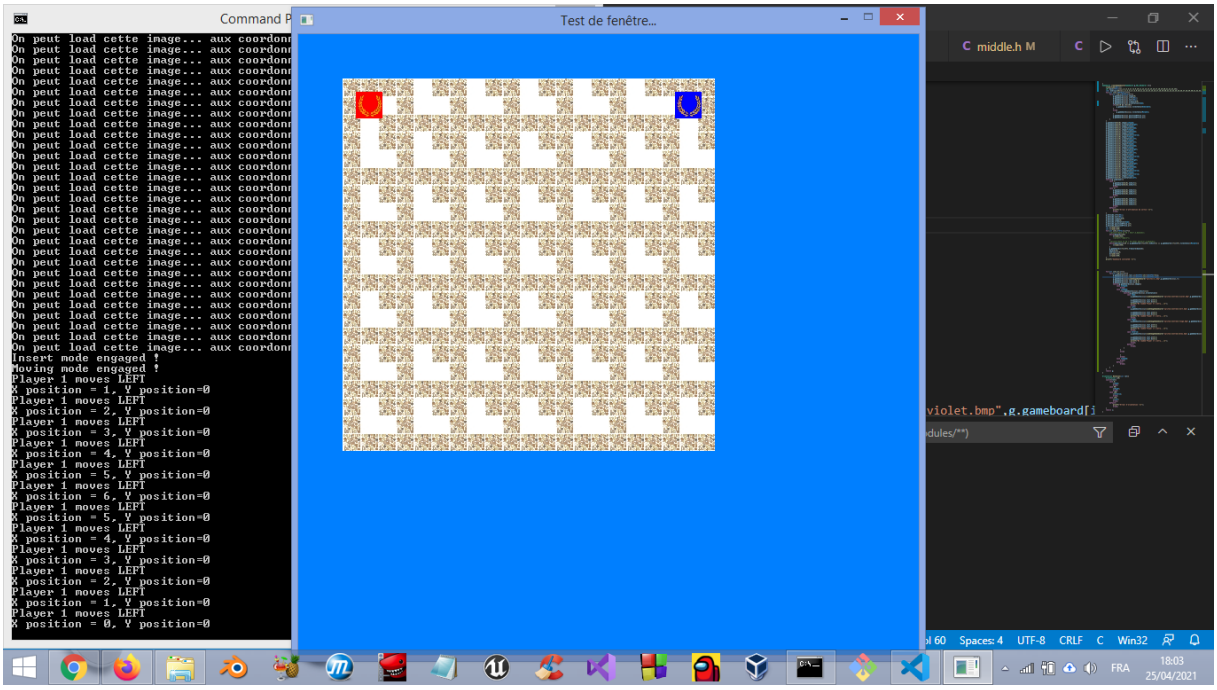


3.5 First matrix representations of the game board

Now for one of the more complex things… generating the game board itself.

Let’s first establish several things we prepared in the meantime. As we wrote in our analysis report, we wanted a structure for a tile, for a player and for a game board. And so we achieved just that!

As shown below, the player struct is separate from the other two whilst the game board structure has a 7 by 7 tile matrix. A tile structure contains rectangle, texture and surface variables. By default we loaded the **L** structures to test them out and they are placed on the render based on math that you can find in the code, but the result is – so far as we are concerned – impressive:

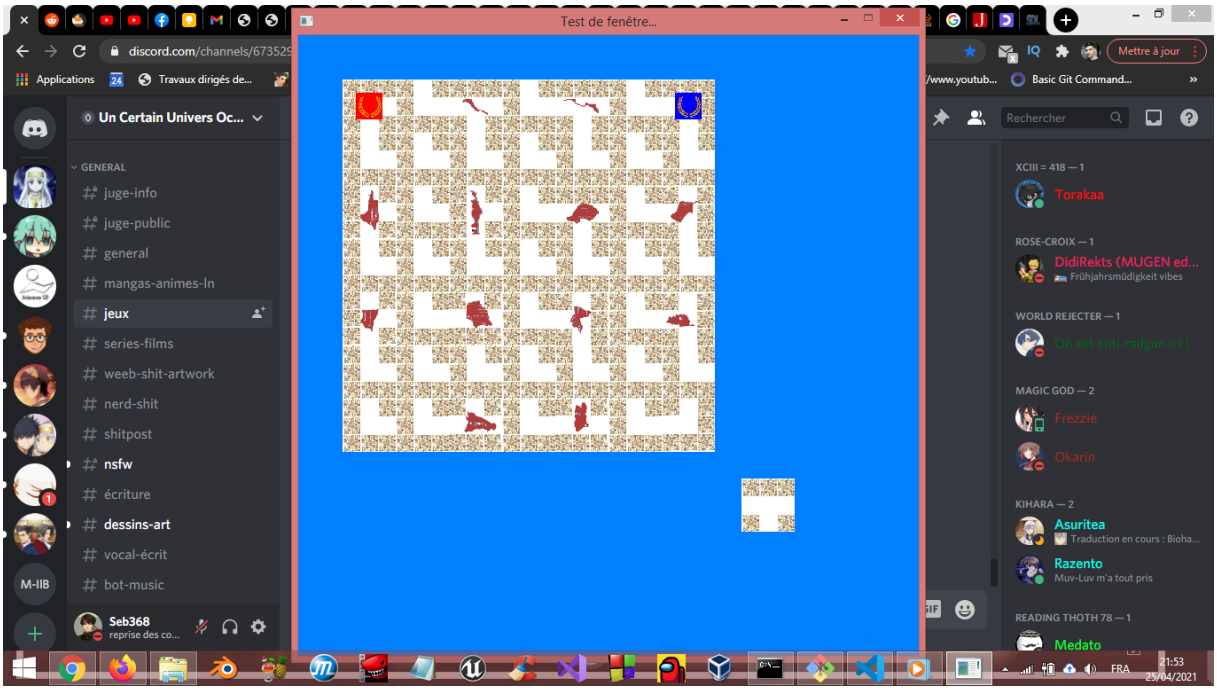


Yes, a matrix full of “L”s… That was only the beginning!

The next step in our plan was to set up the unmovable tiles. We took a page out of the Yugioh group in order to ease our process. The treasures 13 through 24 were going to be the unmovable tiles.

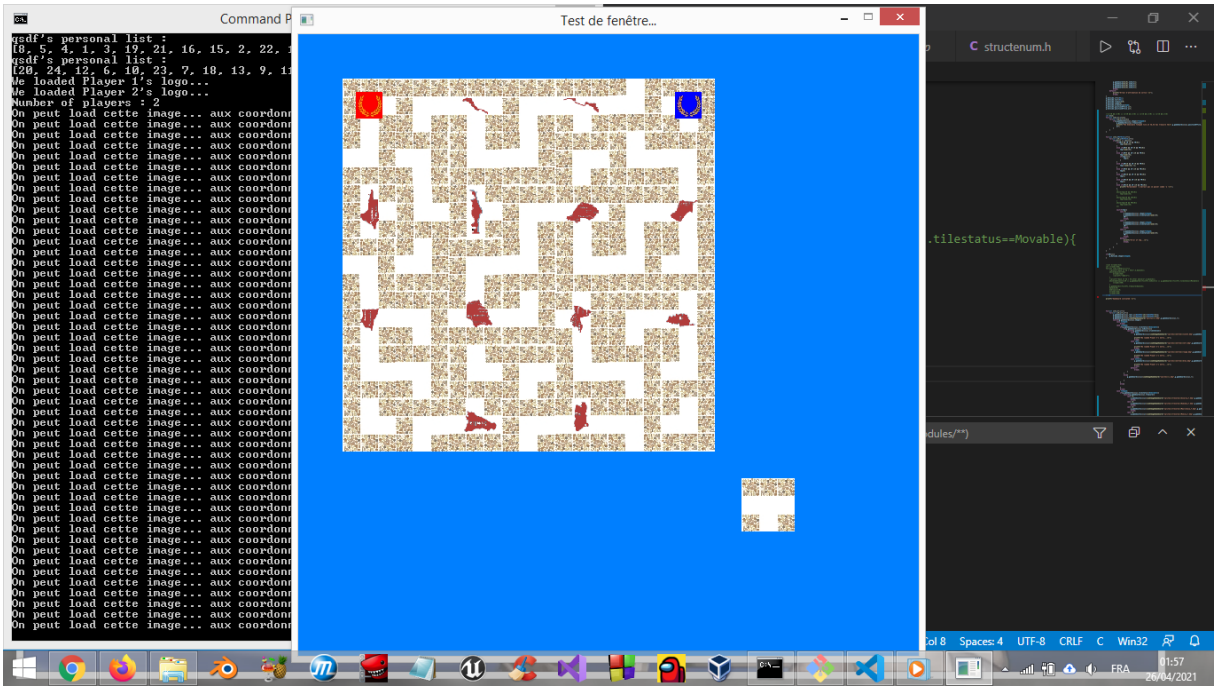
While we are on the subject of the Yugioh group, we were greatly influenced by them in the beginning because they did everything in SDL before us, and with their consent we looked on their code for clarifications on certain things, without copying of course, just to have a reference point on how to first set up SDL, liberate the stuff at the end, and this unmovable treasure setup. They were a great help and we thank them hereby from the bottom of our hearts, because we would not have made this much progress without their help. Of course, in return we helped them in other projects or works.

But back to the topic at hand: we selected the treasures 13 through 24 and have imputed them within the matrix, and the result of this work is shown below:

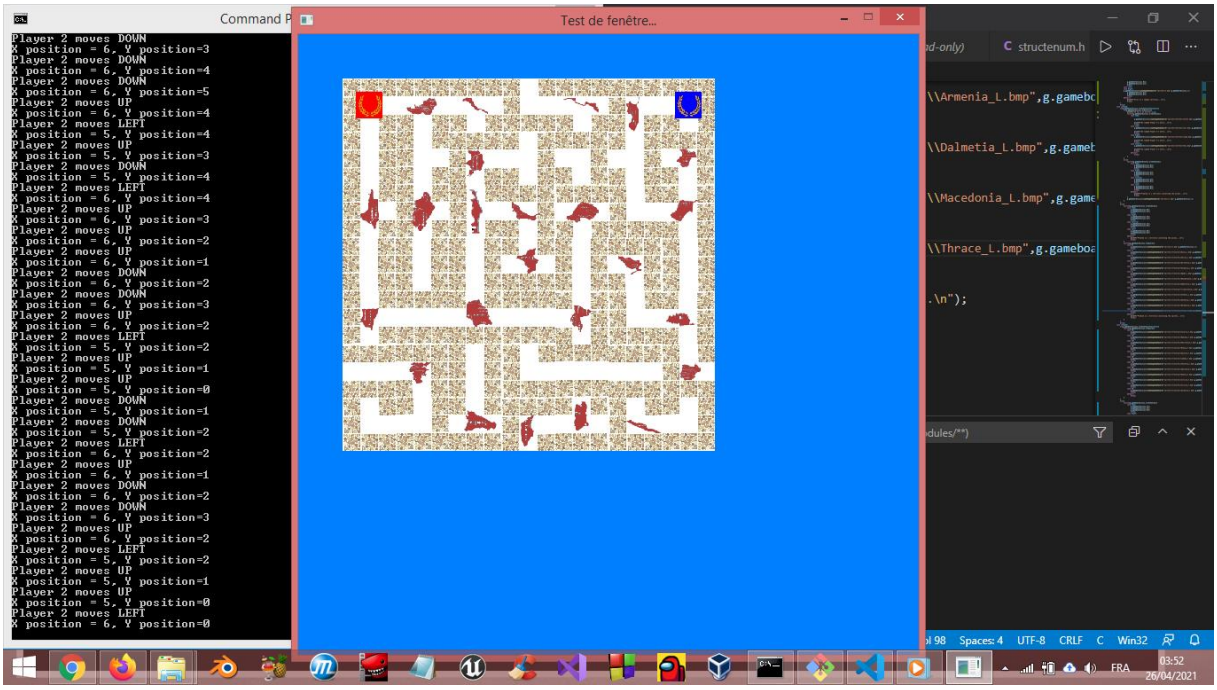


Please note that the tile at the bottom right is not the tile outside of the game! It was still the first test tile and we were trying to place the future outside tile in a place pleasing to the eye and to plan for it we tested our test tile in a pleasing position.

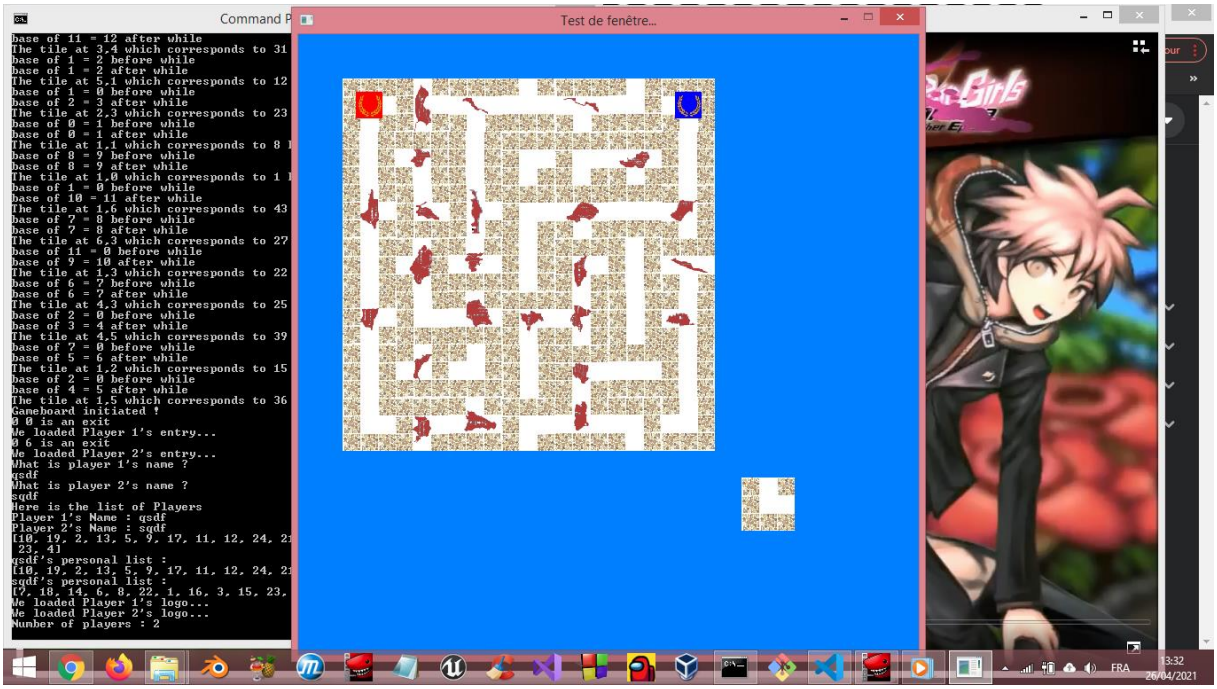
After this step, we made the necessary adjustments so that the movable pieces would have random orientations:



Once we were done with this process, we took out the placeholder for the outside tile, we put the remaining movable shaped tiles (with the same properties as before) onto the game board and then inserted the treasures 1 through 12 randomly (with the I-shaped tiles not containing any treasures). The results respected our plans.



Then, came the first iteration of the Outside tile, which by default was an L-shape for testing purposes:



3.6 Setting up the first iterations of movement and treasure get

After having succeeded in setting up the general game board (visually speaking) we needed to compose the logic behind the way of moving around. Let me explain: Right now we can "phase through the walls of the game" because we are only restricted by the earlier rule of "don't go over 6 or below 0".

3.6.1 Setting up movement rules

Therefore we created a switch on the beginning functions in **middle.c** that would define the first accessibility points for each tile depending on their shape and their orientation. For example if I have an L-shaped tile, the accessibility would be N=1, S=0, W=0 and E=1, meaning that we can go North and we can go East. And this process is done before we load the images, and the unmovable tiles are done manually beforehand while the movable tiles are done after we determined which type of tile goes where and what its orientation is.

After we did that, we needed to add a new rule for the player for each case (aka if he or she goes up, down, left or right). Let's take an example: the new rule added for going left is "*if we can go left from the tile we're on AND if the tile on the left can be accessed to the right, then we can go left*", coupled with the previous rule we had now an efficient way of going around the map without any bugs.

The next step to code properly was to implement the system of getting the treasure.

3.6.2 Allocating treasures to each player

Before dealing with getting the treasure we need to talk about how we actually attribute to each player their list of treasures to get. We have a function that generates a table of 24 integers from 1 to 24 and the code returns a table reorganizing those integers randomly without any repetition. These are in fact all the treasures (their ids) rearranged randomly.

After that we coded another function that takes that "reorganized" table of treasures and the list of players and then gives a part of that treasure table depending on the amount of players in the game. The cases are processed as follows:

- If we have 2 players, we give the first 12 reorganized treasures to Player 1 and the 12 next to player 2.
- For 3 players, it's the first 8 for P1, second 8 for P2 and third 8 for P3.
- The same goes for 4 players.

So, as a matter of fact: instead of actually having the treasures themselves we have their numbers/ids given to each player. Furthermore, each player is also given two special integers in their struct:

- the first one is the treasure number going from 0 (yes 0 not 1) to the total amount of treasures they have,
- the second one is the total amount of treasures they have to obtain.

Each time the player gets a treasure, the treasure number goes up and it identifies the next treasure the player has to get. Once the treasure number is equal to the total amount available, the player has to go back to his or her starting point.

How does that tie back to the game board, then? Well here is the thing: I already mentioned that the unmovable tiles are given treasures 13 to 24 and that the movable tiles are given at random for the treasures 1 to 12, what I did not mention was that it was given to an integer which is also part of the struct of the tile. If the tile contained no treasure, then that integer would be equal to 0 or else it would have had the treasure number, and this data lasts for the entirety of the game.

3.6.3 Giving life to everything: The game loop

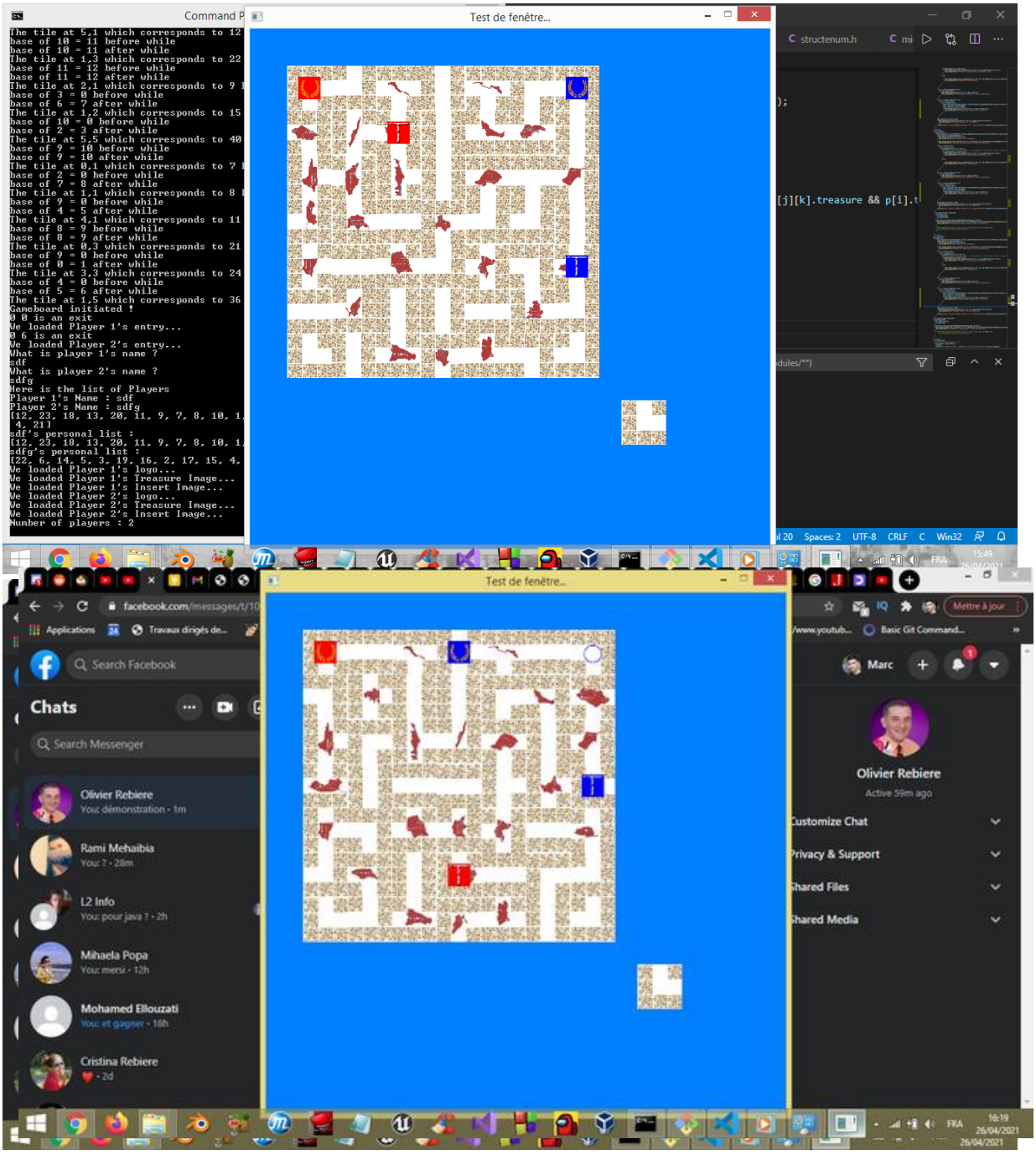
Now we can talk about the system itself. Every time the player moves, we would check each tile again and update the textures accordingly. We test several things within the code.

First we test if the player's treasure number does not equal the total amount. If it does then we check if the current treasure that the player has to get is equal to the one on the tile, then spawn the treasure image of the player at this tile specifically. Else do nothing. However, if the player's treasure number does equal the total amount, then the code spawns the treasure image at the starting point of the player.
That's for the checking, now comes the "get the treasure portion".

If when the player moves, and if his or her position matches the one of the treasure, we have to increase by **1** the treasure count in his or her possession, else do nothing. Later on we found out that this check-up system worked like a radar, and therefore had a bug if this scan was executed only once. At the beginning, the scan was done from up to down. If we arrived at the treasure then the next treasure would not necessarily spawn. It depended on the location. If the new treasure location was down from where the player is, then it was fine, it would spawn, but if it was up, then it would not spawn until the player moved again and the code restarted the whole check-up process again. That was because the scan feature already scanned the tiles up from where the player is. Therefore we needed to execute this check-up twice for each time the player moves, so that we would not miss the next treasure location.

Of course, if the number of treasures equaled the total amount, then it would spawn at the starting point of the player, but once the player reached that starting point, what would happen? Well in the first builds of the game it would just end the program and... that was it. In the later builds… it would have a victory screen and then close.

Below are three images showing both systems in action (the first being the treasure showing up and the other two the treasure getting by the blue player).



Blue took the treasure and it now spawned the next one

3.7 Insertion system and testing for more than 2 players

It is all well and good that we manage to move the player and get some treasure, but now we have to actually play the game, and therefore we need the insertion system to be put in place. We already have created the textures for it, we just need a clever way to use them.

3.7.1 A nice (theoretical) plan

To tackle this issue we need to implement multiple things:

1. The outside tile
2. The insert locations
3. The insert exception
4. The change of the tiles.

We will talk about them in order. First and foremost, the outside tile… How do we initiate it and how do we change its orientation? For the initiation portion, remember how with the remaining movable tiles we chose them randomly for their orientations and shapes? Well that was not quite true. We had to know how many of which shaped tiles there were in the game.

For the unmovable ones, there are:

- 12 T-shaped tiles (which all of them have treasures)
- 4 L-shaped tiles

For the movable ones, there are:

- 12 I-shaped tiles
- 16 L-shaped tiles (which could have treasures)
- 6 T-shaped tiles (which could also have treasures)

The unmovable tiles were initiated manually, but for the movable ones we set up a counter to know which tile remained and would be the one to be placed outside. The “gimmick” is that the outside tile would always be one without a treasure so as to not have problems later on.

3.7.2 The (cruel) difference between theory and practice

At this stage, we just needed to implement this feature into our code and then call it a day, right? Wrong! That was only the first step for creating our insertion system. We already had a good idea that we would first rotate our outside tile, then insert it and then move. Up until now we succeeded into moving the player and getting the treasures, essentially the “bread and butter” of the game. But now we need to focus on the other two aspects.

So we had that Outside tile set up. Here is the thing, the outside tile would always be in its normal orientation (meaning I, L and T) and that trend would continue for the later builds because it is way easier to bring something back to its original orientation and then decide on how you orientate it. But then came the part where we actually needed to orientate it properly. To achieve that result, we used some SDL functions that would rotate it for us at the press of a button. We just needed to think a little harder but we do not need to get into the specifics within the present report. We could now orientate it, great! Now we need to orientate it logically. So we’ve implemented the accessibility shenanigan from before for each tile shape and for each rotation.

3.7.3 Creating the insertion feature

Now for the *pièce de résistance*: The actual insertion! It worked in 2 parts.

The first part consisted in making a table of 12 integers initiated at 0. For what purpose, you may ask? To know if the current player cancels the previous player's insertion (because when you play a normal labyrinth game, you cannot do that). If it does, you cannot go into the moving mode, or else you can go to moving mode. Alongside that system of integers we integrated the insertion texture based on which player's turn it was. We also had to do some mathematical calculations to get the info where to place that texture, and succeeded in coding that process.

The second part was the insertion in of itself. The whole process could be broken down into the following successive steps:

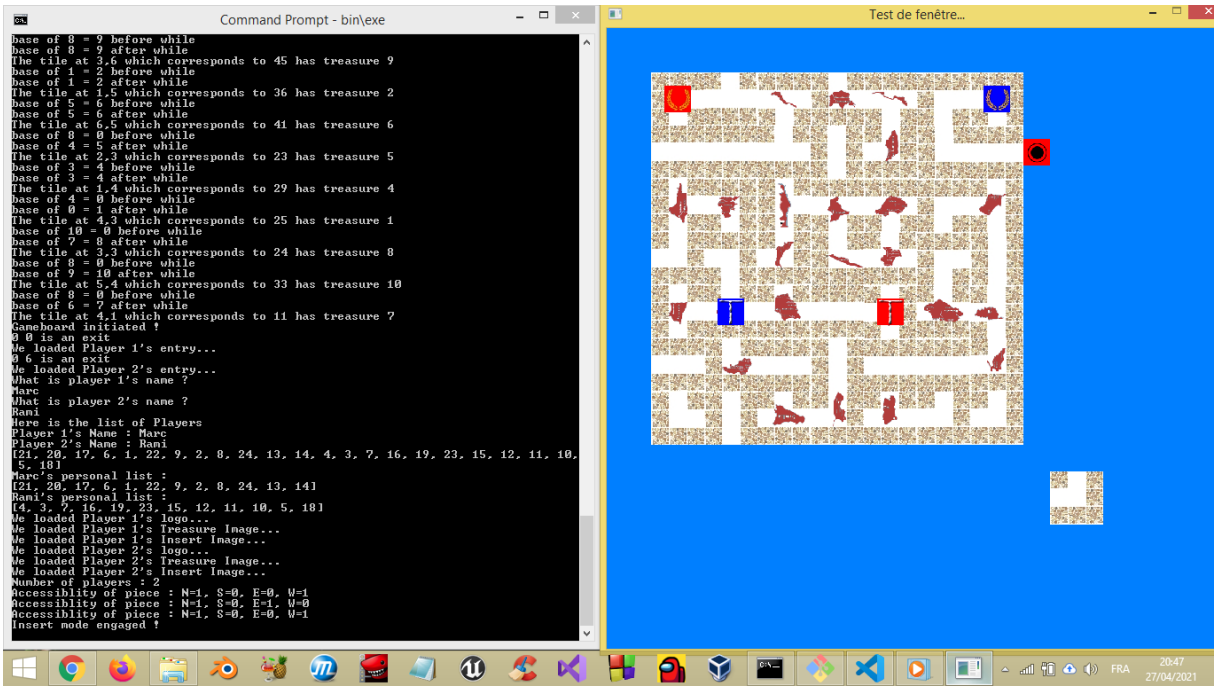
1. The outside tile would take the place of the first tile,
2. The first tile would become the second,
3. The second would become the third,
4. […],
5. The last one would be reverted to its basic orientation and be put outside of the game.

It was a kind of complicated process but that is how it finally worked.

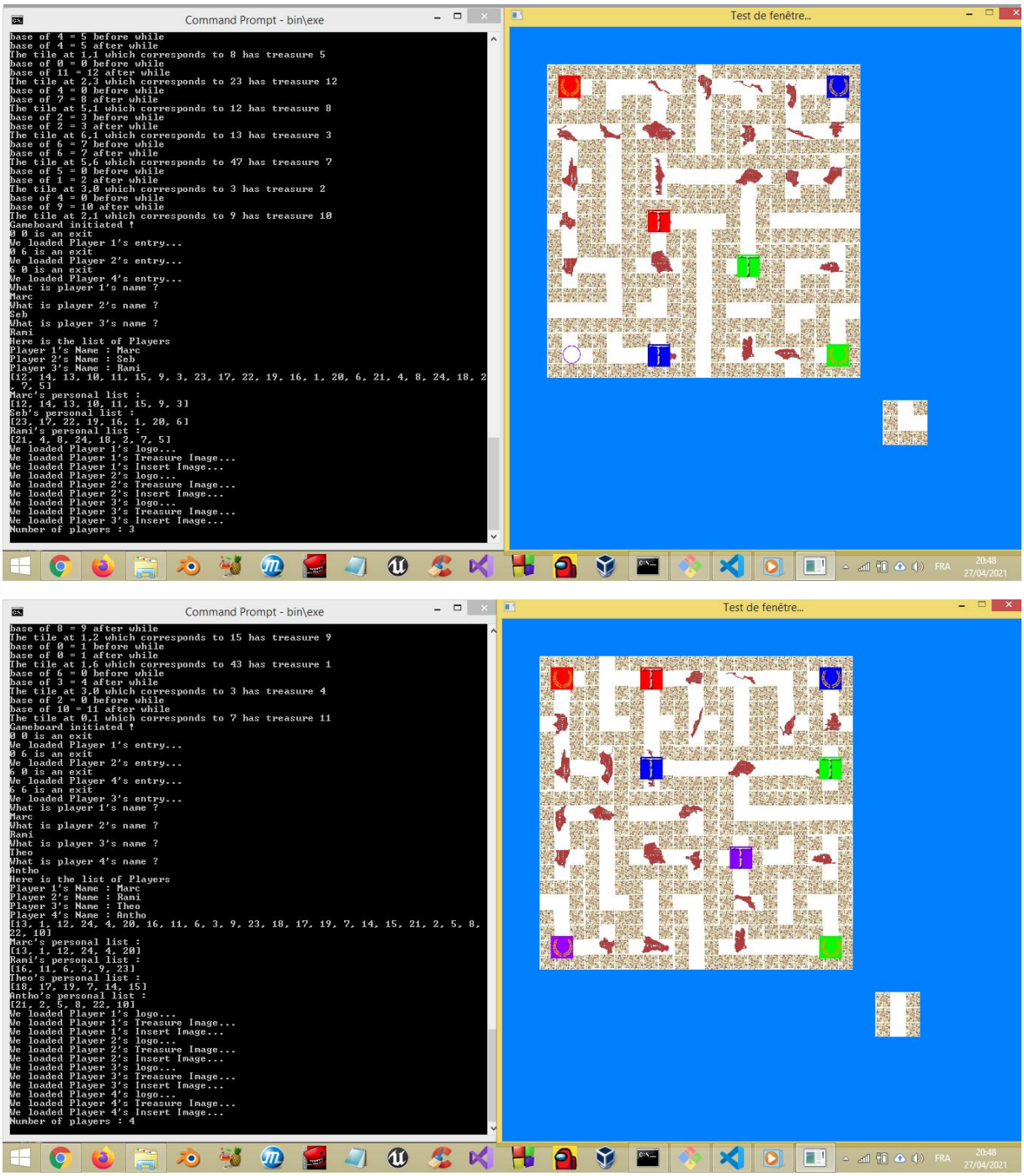
We also tested with more than 2 players and it worked, after some other bug fixes.

From then on, we had a working prototype of the game and we could technically play it with some bugs here and there (there are probably still some even as we write the present report) but the hardest challenge was yet to come. Before we talk about the next "roller coaster", here are some other development images of what we described in this part.

Default insertion position



Insertion after moving towards the right



Testing with multiple players at the same time

3.8 Audio system nightmare

This was the worst night I had. No sleep whatsoever. None. But I am so happy we could implement this feature without harming the whole coding process. Instead, we made it better.

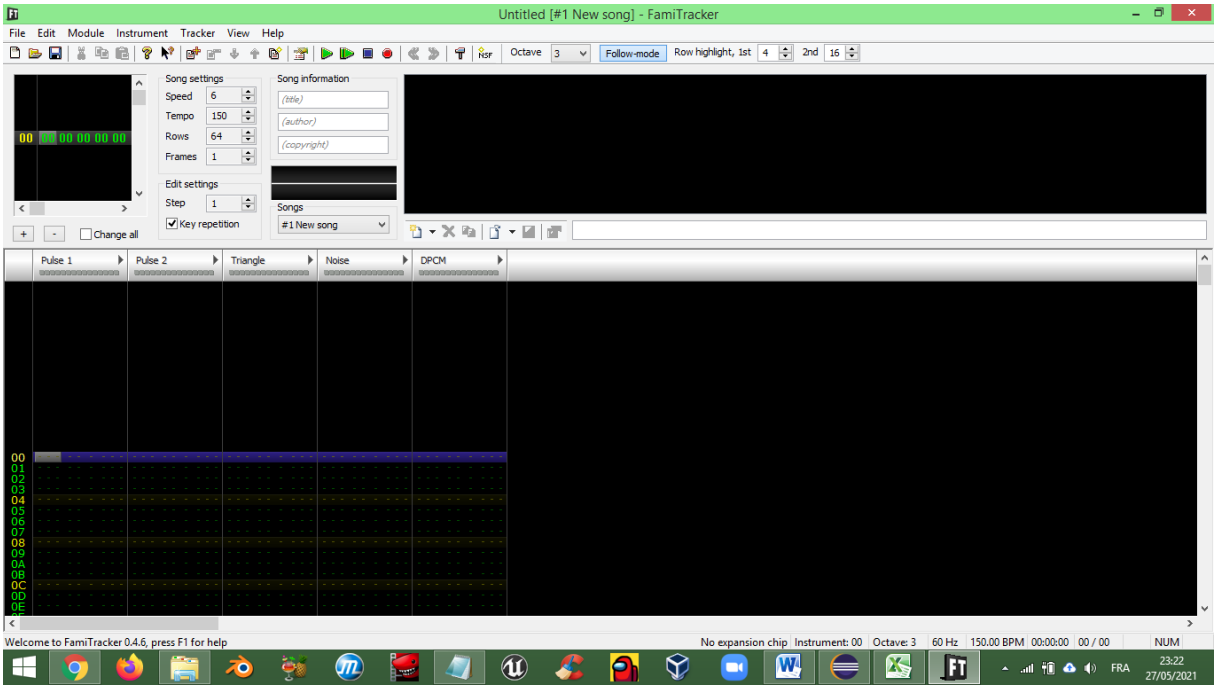
3.8.1 Enhancing the players' experience

This initial idea came into being by the sole fact that we could “Initiate Audio” in SDL. Turns out, it is only complicated to first grasp it, but then when you get the hang of it and improvise, the process in itself is quite handy.

But how did we manage to implement such a feature? There were three phases of work during the night of Monday to Tuesday (I think):

1. Creation of audio samples and sound effects
2. The “agony” of making it work
3. Implementation of every sound effect after it finally worked (yeah!)

Let’s start with the most enjoyable part: The creation of audio files. The idea we had in mind was to create a retro kind of Labyrinth game for the audio atmosphere. So I looked for a NES (Nintendo Entertainment System) sound creator and came across a nifty sound editor called Famitracker (http://famitracker.com/). In this application you have to manage a sort of timeline and 4 channels (2 Pulse channels, 1 Triangle channel and 1 Noise channel) where we insert notes (in the American system of **C D E F G A B C** which is the equivalent of **Do Ré Mi Fa Sol La Si Do**) and we can change the speed of the music, its tempo and create new instruments. At that time, I had only a small grasp on how it functioned, but it was just enough to generate decent sound effects for the game.



This application can process and export the music/sfx you create into **WAV** form (this detail will be important later on). With Rami we categorised which sound effects we needed and for what action. We also downloaded an audio file (which was the first easter egg) for testing purposes and we had it in both **WAV** form and **MP3** (just in case).

We produced all the sound effects but we then needed to figure out how to make them work into our project. My first attempt at incorporating the sound was by downloading a library extension of SDL called **SDL_Mixer**. It was supposed to be an easy incorporation but ended up being a waste of

time and resources. Maybe if we had more time to think about it we could have incorporated it (because apparently in the long run it was easier and could incorporate any audio format).

Therefore, after this timely setback we turned our heads towards the sub library already available in SDL, called **SDL_Audio**, which could only support WAV audio formats. Spoiler warning: it didn't work the first time. I followed a tutorial on how to make a sound come out of the computer with **SDL_Audio functions**. And when I tried to press the first button of testing (which was the **B** key, the same button where we tried to first have a response out of the computer earlier in our development), no sound came.

But instead of being discouraged, I just improvised. I checked how the functions worked and then changed a few pieces of code here and there, and it worked! We had an officially working audio feature!

3.8.2 Hitting the wall of all computers

Well actually, not quite. We indeed had created a game that could produce sound. But we had a limitation problem that we encountered after pressing repeatedly on the respective key.

It turns out that after a certain limit, the audio did not work anymore. And we found out that each computer contains "audio output canals" called "devices". Each time we executed our audio function, we would use up one of the computer's devices instead of reusing the one where the first sound came about. Having finally understood the cause of the problem, I then modified the function so that each time you use it (after the first time) the code would liberate the respective device and the previous audio and so it could then reuse the device all over again.

When that finally worked, it was already morning, I was tired as heck, I believe it was around 8 am, and the only thing I could do was to just take a nap, and so I did. I did not want to waste more than a day in implementing an audio system and put all my energies into it but it was worth it. Afterwards, I implemented the remaining sound effects and another easter egg for the teacher... and invented a so-called "**Jukebox mode**" described above.

3.9 Finishing touches

Alright, by this time we had a working game with audio! We could add some "quality of life" improvements to make it more appealing to the players.

3.9.1 Main colours and ergonomy

First we changed the background from blue to black, creating more contrast, in order to not strain the eyes of the players when playing the game. It was quite effective, because I was already feeling better!

Afterwards, we created the textures for the title screen and for each player to win. We just used our good ol' Photoshop for this task at hand. Note that we did not put the names of the players, because we did not know them, and we could not generate an automatic name creation for the win "hall of fame" screen. However the win would also be written onto the terminal and the name of the one who won.

Then, we had to implement something the teacher told us way back when we first started this project: ***coloured text***.

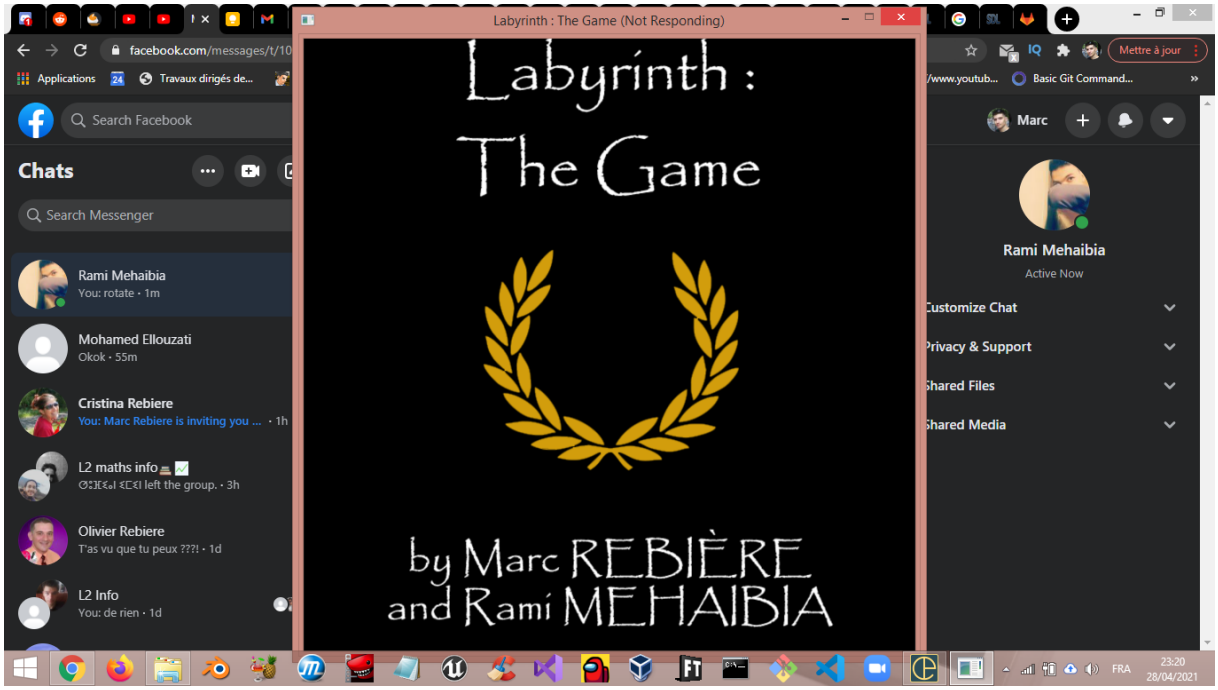
3.9.2 Coloured text and coding adversity

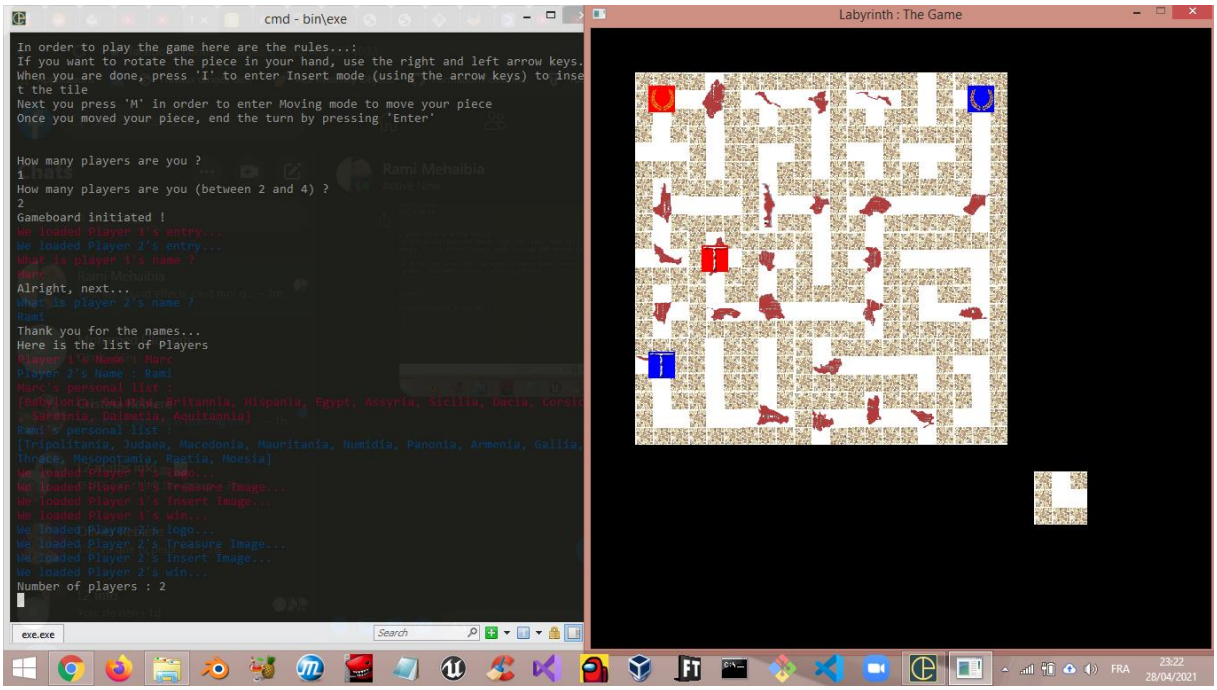
The thing is, we used the windows command prompt for Windows 10 or Windows 8 (depending on which computer we worked on at the time), and both cannot change their text colour locally, because it does not recognize some Linux commands fully and natively. The only thing it could do was to change the whole display colour.

And that is when we came across this wonderful terminal called **cmder** (https://cmder.net/) which could do all the Linux functions (as long as you had a good version of **mingw32**) and even use **git functions** (and so we could do git pulls and pushes from it). And to top it all off, we could change the colour of the text locally. So why did we bother ourselves to change the colour of the text? It was all for the purpose of having each player get their own colour (P1 Red, P2 Blue, P3 Green, P4 Magenta which was the closest to purple). And after tweaking a bit for each time we printed out something, the code would recognize how to colour the text depending on which player's turn it was.

Below are some relevant print-screen images illustrating the coloured text feature:


cmd - bin\exe
insert
moving_mode
move
endturn_mode
treasureget1
treasureget2
treasureget3
You pressed the B button ! Nice !
You got Rick Rolled !
rotate
insert_mode
insert
moving_mode
moving_mode
end
Click at this coordinate : (264,476)
C:\Users\Marc\Documents\Labyrinth\mythologie_romaine\code_in_C (main -> origin)
λ bin\exe
Welcome to the Labyrinth Game
Welcome to SDL version 2.0.14
In order to play the game here are the rules...:
If you want to rotate the piece in your hand, use the right and left arrow keys.
When you are done, press 'I' to enter Insert mode (using the arrow keys) to inser
t the tile
Next you press 'M' in order to enter Moving mode to move your piece
Once you moved your piece, end the turn by pressing 'Enter'
How many players are you ?
1
How many players are you (between 2 and 4) ?
2
Gameboard initiated !
We loaded Player 1's entry...
We loaded Player 2's entry...
What is player 1's name ?
Marc
exe.exe
(Not Responding)
rinth :
Game
EBIÈRE
EHAIBIA
Marc
Rami Mehaibia
Active Now
Customize Chat
Privacy & Support
Shared Files
Shared Media


Labyrinth : The Game (Not Responding)
Labyrinth :
The Game
by Marc REBIÈRE
and Rami MEHAIBIA
facebook.com/messages/t/10
Search Facebook
Chats
Search Messenger
Rami Mehaibia
You: rotate · 1m
Mohamed Ellouzati
Okok · 55m
Cristina Rebiere
You: Marc Rebiere is inviting you ... · 1h
L2 maths info
left the group. · 3h
Olivier Rebiere
T'as vu que tu peux ???! · 1d
L2 Info
You: de rien · 1d
Marc
Rami Mehaibia
Active Now
Customize Chat
Privacy & Support
Shared Files
Shared Media



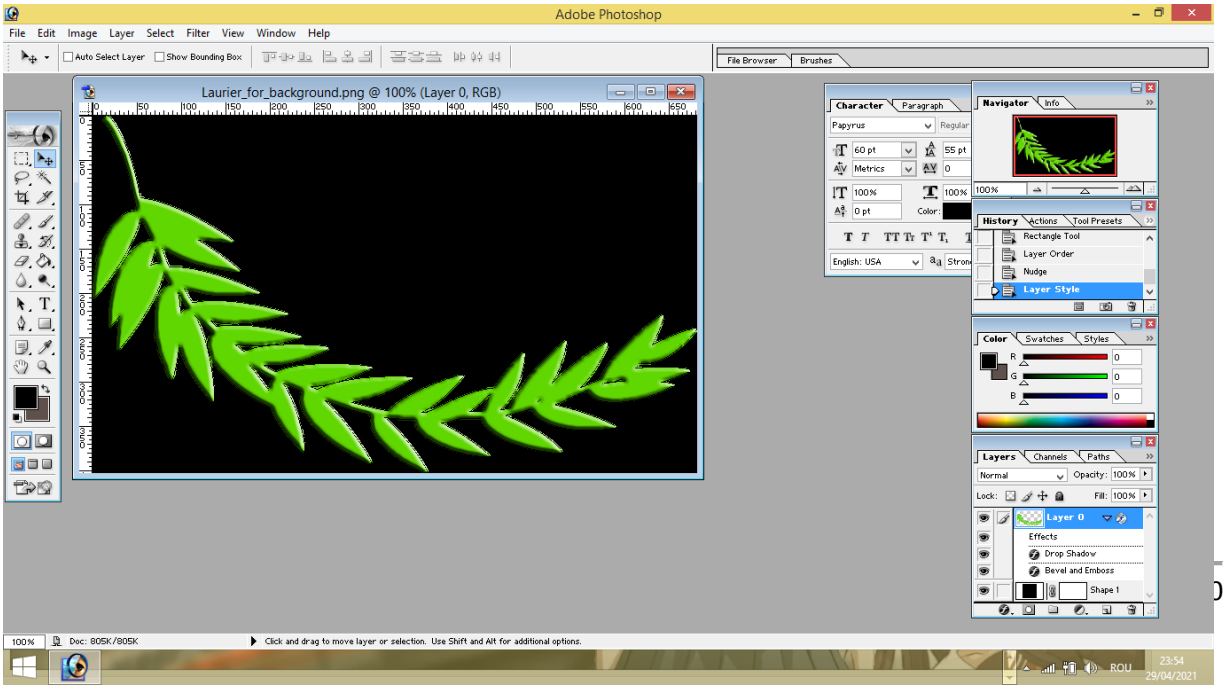
3.9.3 Easing the gameplay

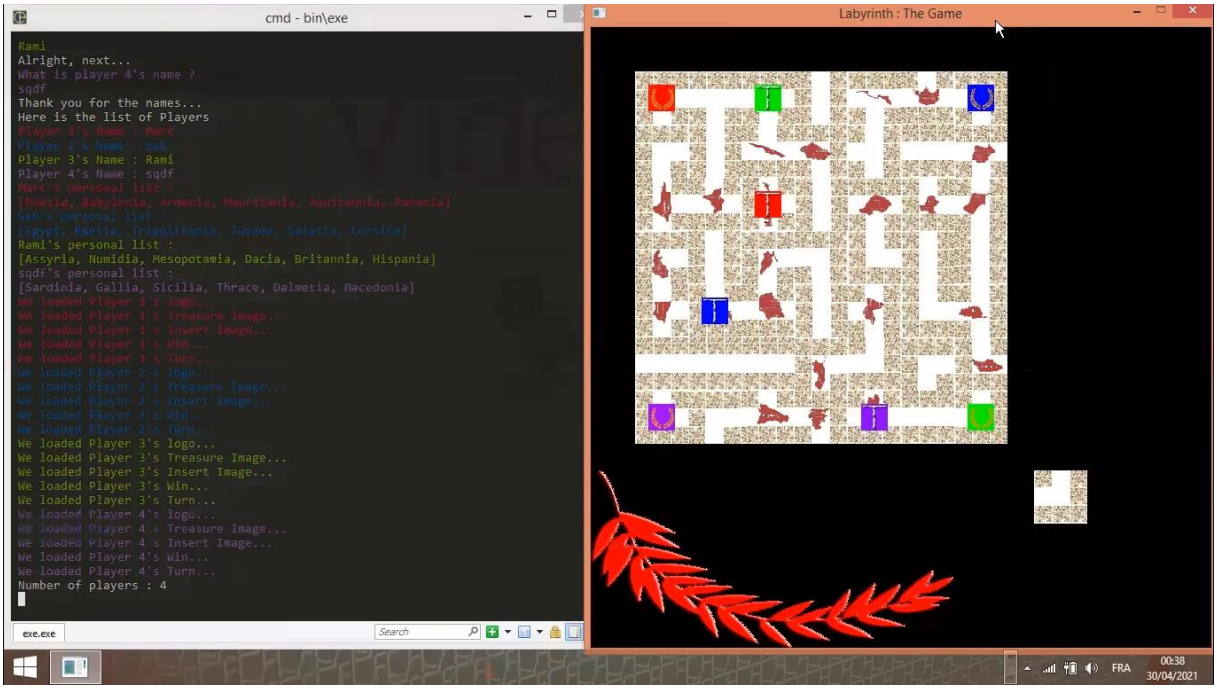
We also made a table listing all the lands to conquer and in the same order as the analysis report. And each time it is somebody's turn we would now tell which land they have to conquer (because we could convert the treasure at the current treasure number of the player into a printable word). Thus instead of printing "Player 1 must conquer 2", it says "Player 1 must conquer Dacia" for example.

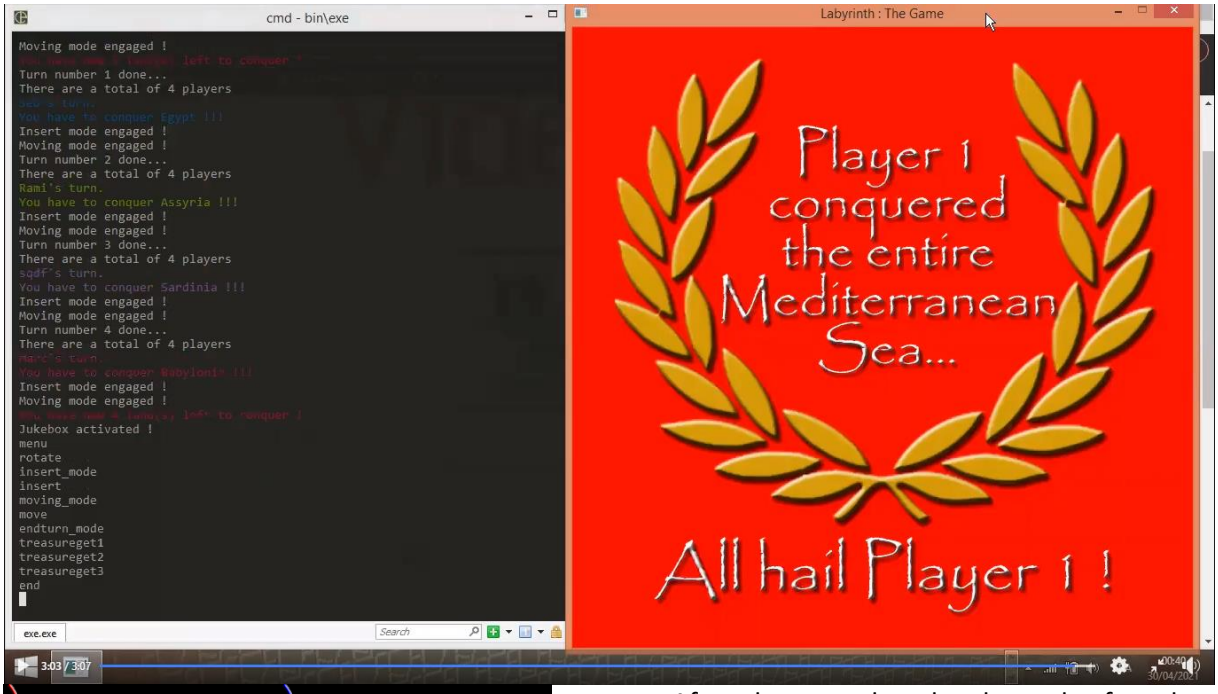
We also added a print for each time a player conquers a land by telling him or her how many lands are left and if there are none, we just tell him or her that they have to go back to tell their tale or something along those lines.

3.9.4 More engaging visual effects

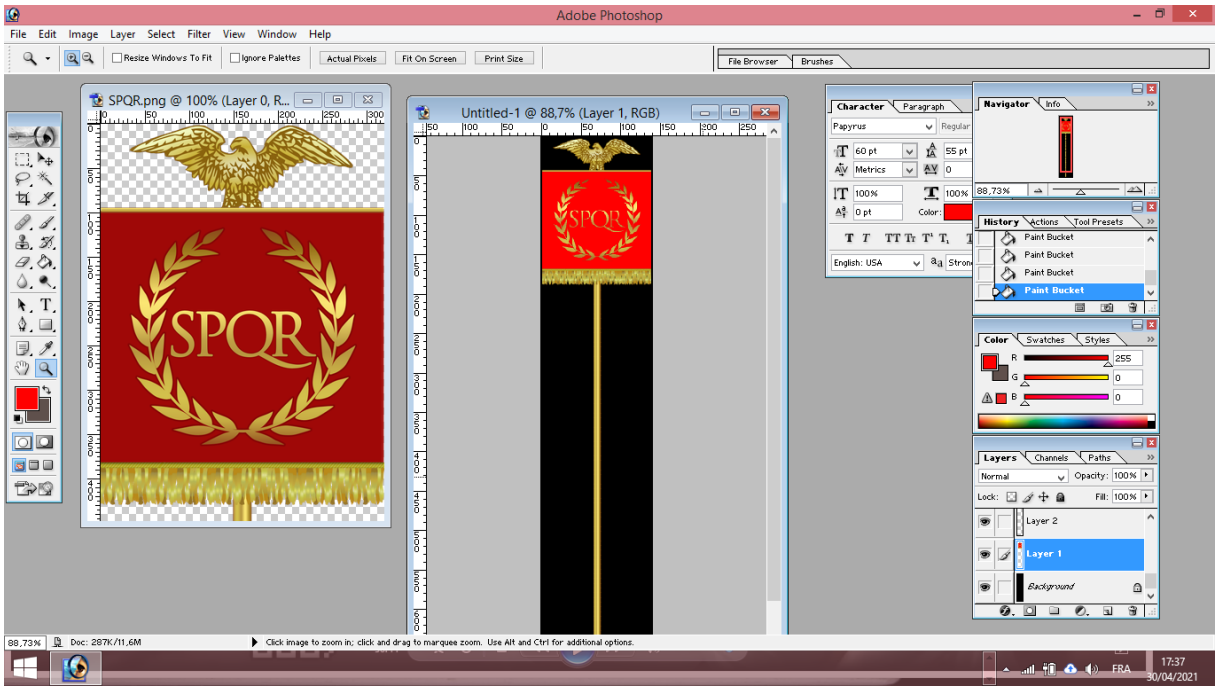
After that, we thought about how to tell visually which player's turn it was (even though we incorporated this system on the terminal, we went the extra mile for this), and the idea we came up with was to use another laurel coloured by the player's turn.

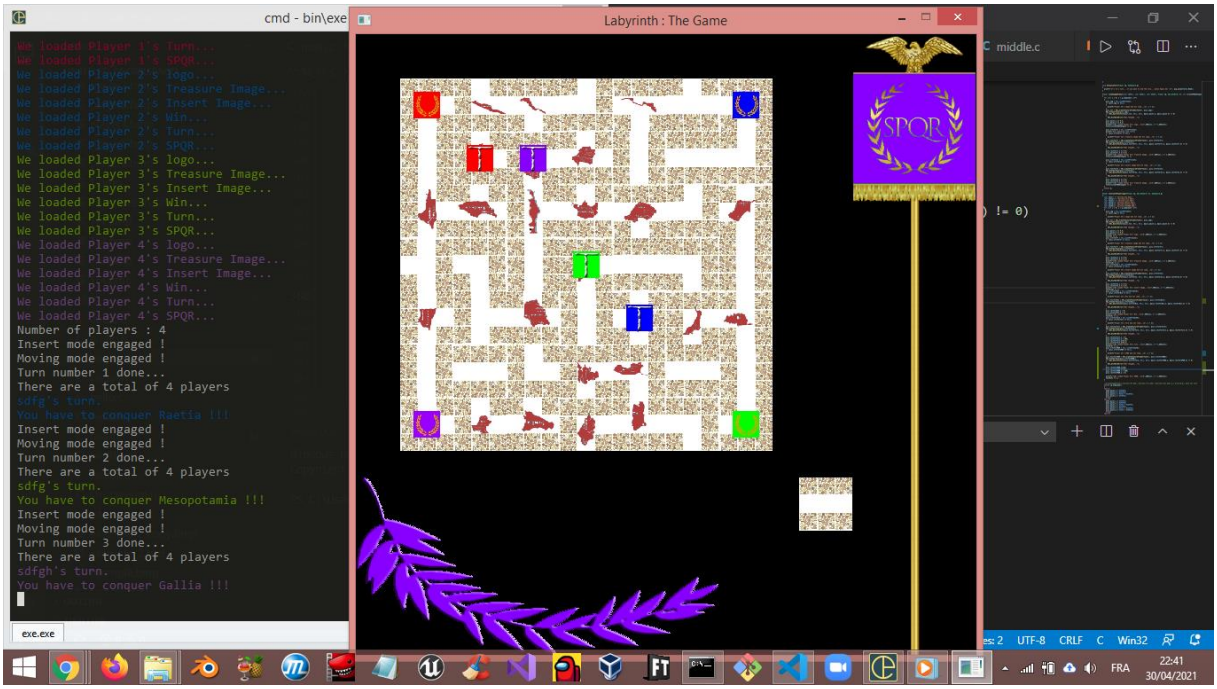






After that, we thought about the fact that the right side panel of the game board felt kind of "off" because it had blank space. So we decided to fill it up with a roman flag with the player's turn as a support to tell who's turn it was.

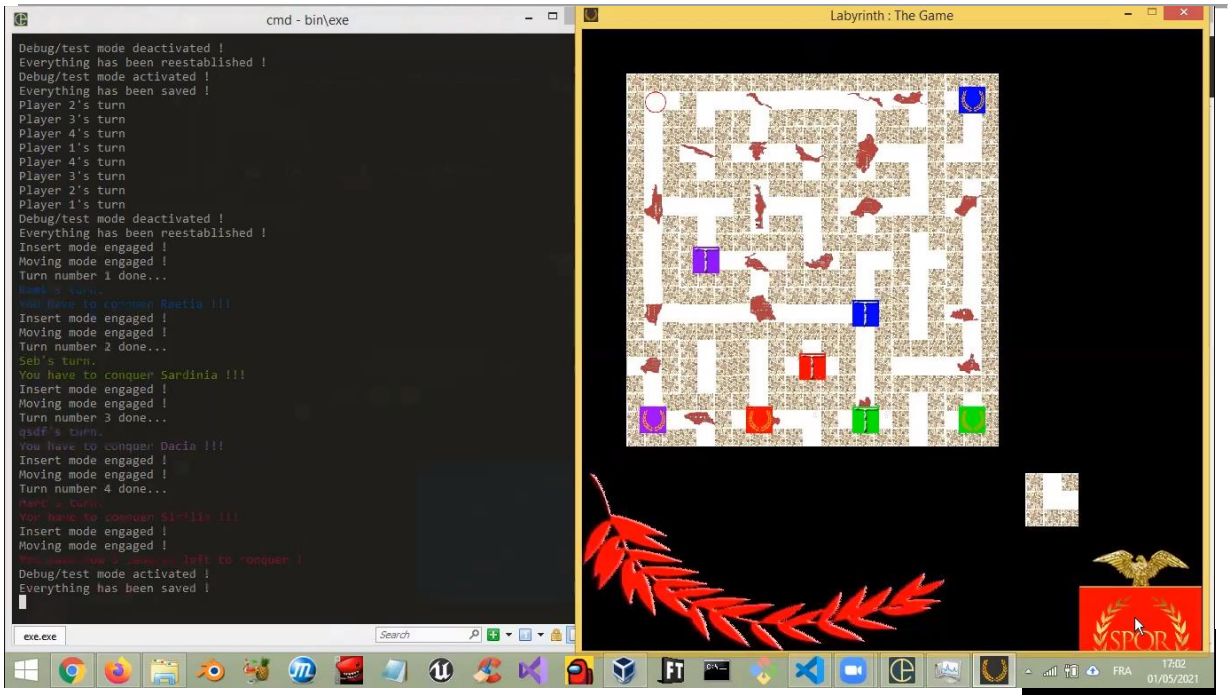


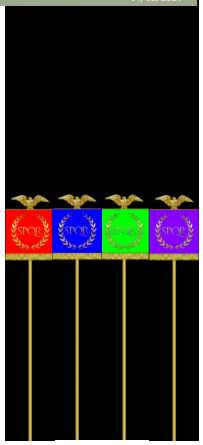


3.9.5 Visual effects enhancing gameplay

Then, we thought that these new-created flags could have another functionality, but we had to change them a little to make it work.

We had to actually make them bigger to make our new plan work. Those flags' role changed from telling whose turn it was into the visual gauge of how many treasures each player possesses in real time. Each time one player would get a treasure, his or her flag would rise a little more, and when it was fully risen, that meant that you had to go back to your starting point. Simple and efficient!





In that time we also developed a **Debug mode** to test out those functionalities while saving the turns of each player and their number of treasures.

And that pretty much sums up our entire development journey. We will be now talking about our current repository as of the time I am writing this report.

3.10 How our repository is organized

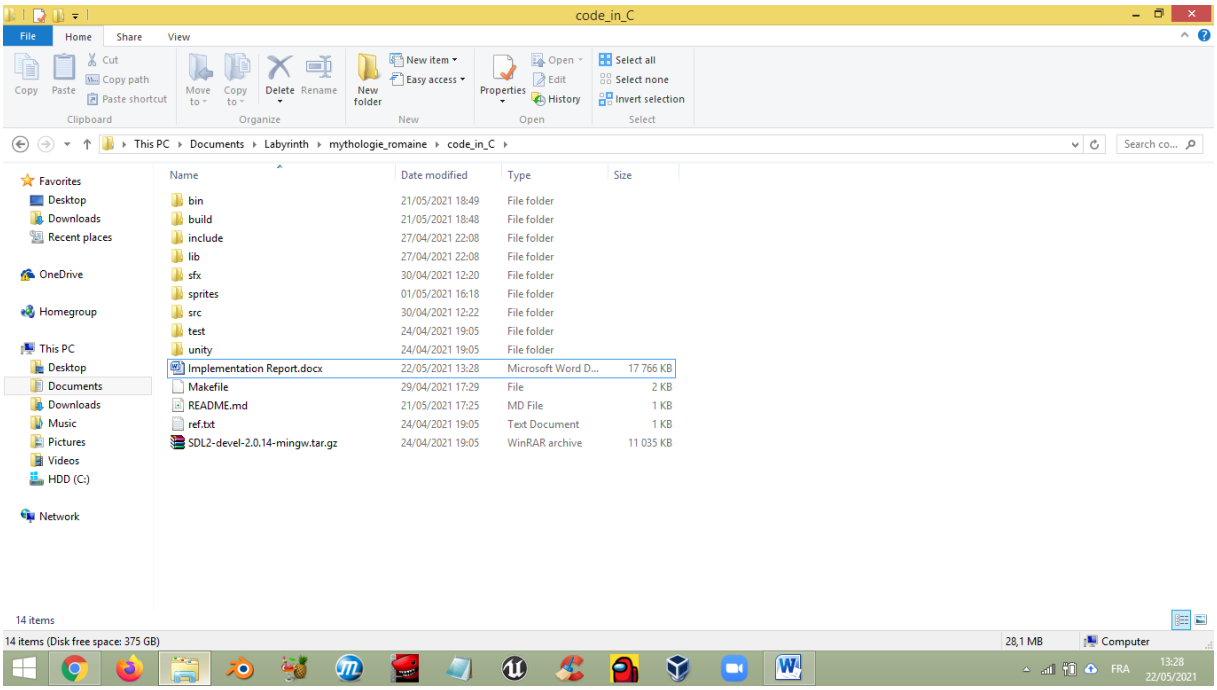
We have a bunch of folders that each serves a purpose:

1. **bin** : This is where our executable resides and where all the **dynamic libraries** of **SDL** and **SDL_Mixer** are
2. **build** : This is where all the **.o (object)** files reside to build our executable
3. **include** : This is where all the **.h (header)** files of **SDL** and **SDL_Mixer** reside
4. **lib** : This is where the actual libraires of **SDL** and **SDL_Mixer** reside
5. **sfx** : This is where our WAV sound files reside… no peeking!
6. **sprites** : This is where all our sprites/textures (aka the **BMP files**) reside.
7. **src** : This is where our source code resides (it is composed of **.c files** and **.h files**)
8. **test** : This is the folder where we tested before how unity worked
9. **unity** : This is where all the unity folders and files reside.

As for the files residing outside of the folders :

10. **Implementation report.docx** : Inception (there will also be a **pdf file** once we finish writing this report)
11. **Makefile** : This is a file where we write terminal commands and if we write make or mingw32-make on the terminal, it executes all the commands we have written in it.

12. **README.md** : This is a file where we have written what programs / software you need to install in order to play this game with links to said sources.
13. **ref.txt** : This is a file our teacher gave us as a reference on how to organize our repository
14. **SDLblablabla.gz** : This is the compressed folder where our SDL original files reside in case of an emergency



Conclusion

At first, we felt overwhelmed by the task at hand. How on earth could we make it? Stunned by all the layers of knowledge and coding required and somewhat discouraged, we felt that we had to begin slowly, but with perseverance. It was not easy, with many ups and even more downs, as an actual project is, in fact.

But in the end, we had a lot of fun creating and making this game, with some nightmares along the way, but it is alright because we learned (a lot of) new stuff.

We did it personally before during our distance-learning sessions on **Zoom**, but once again we would like to hereby thank our teacher for this very challenging project idea (although I wish we had chess, maybe for the guys next year?).

We are very well aware that the sanitary measures taken to fight the spread of the Covid-19 epidemics implied drastic changes for the “Via Domitia” University and for our teaching team. It was hard for everyone, teachers and students as well, but we are happy to see after more than a long year of “disruption” that, despite all forms of adversity, we overcame them and successfully learnt a lot through your efforts. We had to change our habits and our “functioning mode” as students. It was painful and sometimes discouraging. But we also think it is for the best. We feel stronger and wiser than before. Thank you all!

We hope that it will not be infuriating to rummage through our code and that you enjoyed reading this long report as much as we strived to write it, in a sometimes entertaining form…

Thank you for your attention, and have a nice day!