CentraleSupélec

ESSEC BUSINESS SCHOOL

Python - Project

17/12/2021

GOLD & GNOMES

Pixel art game designed by

Łukasz Pszenny,

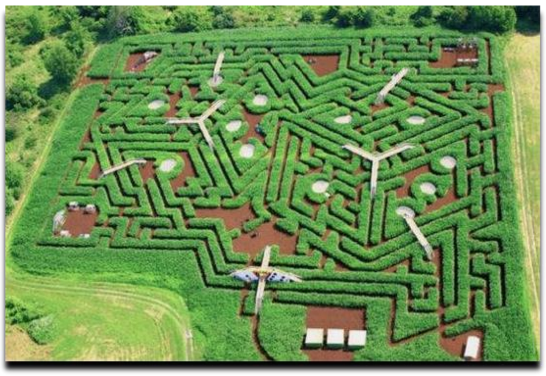
Paul Bédier,

Mengyu Liang

Table of Contents

Goals and objectives

The maze first appeared in ancient Greek mythology, symbolizing the eternal philosophical contradiction between freedom and reality. Humans have built mazes for 5,000 years. During this period, these peculiar buildings have always attracted people to walk hard along winding and difficult paths to find a way out, because they provide players with a joy to explore the unknown.



Davis Mega maze in England—one of the most famous mazes in the world

The maze game was then introduced and became a popular puzzle game, used to exercise people's reasoning ability, spatial perception ability and computing ability.

Our project was to design a typical maze game with randomly generated levels, allowing players to collect the coins and find the exit.

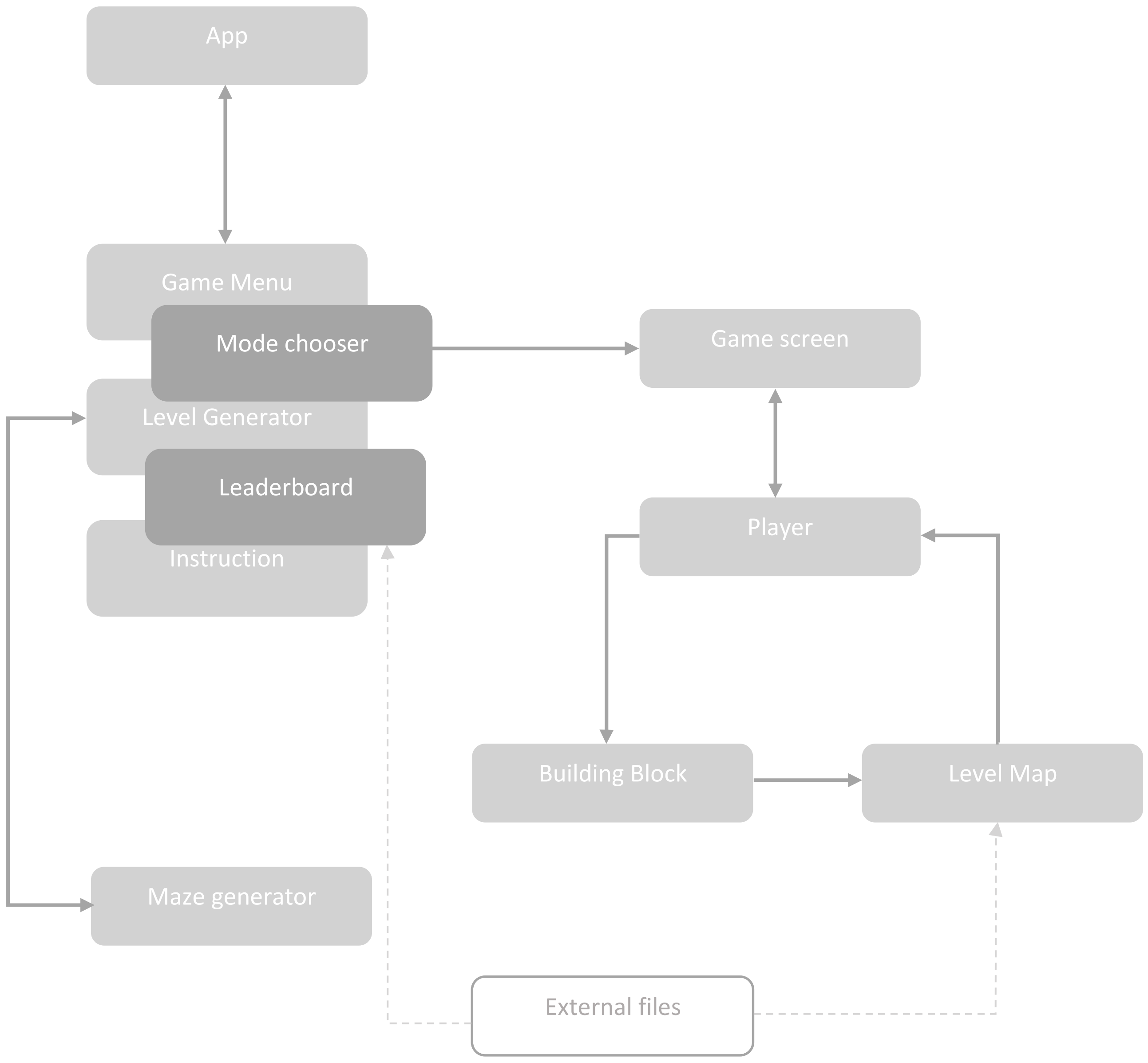
We were aiming to build a project that is easy to scale up (adding features) and self-sufficient (fixed number of assets, does not require large resources in terms of graphics and level design). To do so, we decided to focus first on the structure of the program and on building classes that would cover most needs. We also decided to rely on a maze generation algorithm so that we would not be required to design every level by ourselves. The goal was for the player to be able to play the game repeatedly without encountering the same levels, all the while limiting our level design work.

Development plan

	Objectives
Version 0	• identification of the basic elements of the application in such a way that its future development does not require a significant change in the structure of the program, but only the addition of new functionalities • creating a prototype of an application, containing GUI that allows basic interaction of the user with the application (designing front-end of application). • implementation of the basic functionality of the program in the form of a maze and the possibility of completing it by the user (designing back-end of application) • collecting graphic materials and defining the visual concept of the application • developing a method of storing levels available to the user
Version 1	• expanding the game module by introducing additional elements with which the user can interact, i.e. 1. implementation of coin blocks which collection is required to open the exit from the labyrinth 2. introducing a special type of block that prevents user from going forward until the appropriate action is performed • introducing a timer after which the game ends automatically • introducing a scoring system to encourage players to come back to the game to improve their scores • adding a way to display the results of the best players in the GUI (leader board)
Version 2	• adding a module that allows player to generate and save levels with given properties • adding a new game mode in which the user has to complete as many automatically generated levels as possible in a limited time • implementation of sound effects and background music

Program architecture

The following diagram shows a simplified structure of how the application elements communicate with each other in versions 0,1 and 2. However as we identified some redundancy during development of version 2 we changed the program structure a little by moving some functionalities into an additional class (Buffer) that is not indicated in this diagram and we made version 3. Further information about this change is provided in section describing version 3 on page 7 and 8. The main purpose behind the creation of each application element however wasn't modified between architecture of versions 0,1 or 2 and version 3.



App module is a controller responsible for displaying application to the user. It is always running and closing it, is equivalent to closing an application. Whenever user wants to change window for e.g between Level Generator window and Game Menu window, a request for a change is sent to an App which closes specific class and opens another. A Level Generator is responsible for collecting data from user and sending it to Maze Generator which performs computations to generate a specific level, then it sends this information back to Level Generator to display it to the user. In order to play a game, user needs to go through Mode chooser to specify parameters for Game Screen class that displays loaded level. Another task of Game Screen is to pass information to a Player who is able to interact with level by destroying block / collecting coins / finding an exit etc. Level Map and Leaderboard are two elements of an application that are able to get data from external files. The first one loads data in specific format into memory, the latter one displays data about result of previous players.

Versions Description

Version 0

As described in the development plan section in this version, we wanted to create a fully working prototype with basic functionalities. We divided the classes into two types: the main task of the first group of classes is to display information to the user and to enable him to switch between displayed tabs. This first group is located in the **GUI** module. The **App()** class is responsible for embedding a tab in a window and controlling what is currently displayed to user. It has 2 methods in addition to initialization. **App.show_frame()** is a method that changes the currently displayed tab: game menu, game screen and instruction tab. Additionally, the start game button which is in **GameMenu()** class executes the **App.choose_level()** method which displays user the option to choose one of the levels in the levels directory. The **GameMenu()** class represents the game's main menu, which is the first tab displayed by the App class after running an application. Currently, the number of possible buttons to interact with is limited but being aware of the future developments of the project, we implemented it in such a way that it is easy to add additional options by creating separate classes for each tab and calling them in **App.show_frame()** method. After choosing one of possible levels a **GameScreen()** class is displayed for a user. This class connects abstract classes **Player(), LevelMap()** and **BuildingBlock().** It has a few methods that helps the game to run. **GameScreen().pause()** stops the game and prevents user from moving. **GameScreen().action()** method defines what happens when a player presses keyboard buttons. At this stage it allows only to move a player with w/d/s/a characters, but we will want to put additional functionalities here in a future. After every action user performs the method also triggers up to two additional methods. One is called **GameScreen().check_if_next_level()** where we define conditions that must be fulfilled to change a window to another one. So far it triggers **GameScreen().end_of_game()** method to display congratulations about finishing a level. This is a method where in the future we want to put conditions for alternative actions that happen after player reaches exit. The **GameScreen()** class has also very important method **GameScreen().draw_level()** that loads and displays initial view of the level with a player in it.

The other group of classes is focused mainly on game itself and has a variety of methods that doesn't change an appearance of a window until it is crucial. This group is located in the **gameplay** module (originally called modules.py) These are three abstract classes **Player(), LevelMap()** and **BuildingBlock()**. The most basic one is **BuildingBlock()** that stores data about properties of a block and his location. This is a class where we define if a block is accessible by player, if it is the exit from a maze, what are his coordinates etc. It has one method **BuildingBlock().draw()** that displays specific block (so only one, block with defined coordinates) on a given Canvas class. **LevelMap()** is class that is made from building blocks. One of its key attributes is list of all defined blocks that composes a level (beside player block). As we don't want to store all the possible levels at once, we need a way to load it from hard memory of computer. This is why **LevelMap()** has **LevelMap().load_from_file()** method that allows to open a file with a specific format (described in Level files section) and translate it into a **LevelMap()** class object. The last class **Player()** is a class that translates how the player interacts with a blocks, into changes in a specific element of a maze. It also stores data about player itself so what are his current coordinate and what direction is he facing.

All the GUI classes inherit from a tkinter Canvas class. In that way we are very flexible in terms of placing object in the window. However, this solution also has disadvantages, by doing it we aren't always able to position them so they are robust to the change of resolution. Therefore we decided to fix a window size and prevent user from changing its size.

Version 1

This version saw the addition of two block types, the coin and the obstacle. These features meant some new code in the **gameplay** group of classes. The class **BuildingBlock()** was mildly extended to deal with the exit block type which can have two states, open or closed. The class **LevelMap().load_from_file()** method was extended to implement coins and obstacles placement, as well as make the exit inaccessible to the player until all coins are collected. More interestingly, additional code was developed for the **Player()** class, namely a **open_exit()** method and a **destroy_block()** method. The first one simply switches the door open by switching the exit state to 'open' and giving the player accessibility to that block. The second one modifies the target block into an empty hallway block if target block is a breakable obstacle. Note that the **Player().move()** method was also extended to allow the player to collect coins which they walk over.

This version also saw the addition of a timer feature and a scoreboard. These features required additional code in the **GUI** group of classes. The **GameMenu()** class was extended with a new button for the leaderboard. The **GameScreen()** class was enhanced with an **update_timer()** method, which keeps the timer ticking every second, only when the game isn't paused. The timer is displayed on the canvas using a tkinter label. For the timer itself, we use datetime.time.time() method and we keep track of the difference between the starting time and the current time. A time limit is defined (currently 3min or 180 seconds) which when reached triggers the end of the game. The **GameScreen().action()** method is also updated to allow the player to break blocks by pressing <E> on the keyboard, while **GameScreen().end_of_game()** method is updated with a scoring function based on number of coins collected and time limit. To make it simple, the method checks whether final score is better than the top 9, and if yes, saves it at the right position in the leaderboard csv file. A new class **Leaderboard()** was created to manage the display of the csv file into the GUI of the game. It makes use of draw methods: **draw_position()**, **draw_name()** and **draw_score()** which respectively draw their attributed data on the canvas.

Version 2

This version's main feature is the addition of the random maze generation algorithm. For this, we created a new module called **maze_generating_function**. This module has only one class, the **MazeGenerator()** class, which manages the generation algorithm HuntAndKill. It relies on two methods, the **add_object()** method which can place coins or obstacles anywhere in the maze's empty hallways, and the **save_to_file()** method which can save the maze to a level txt file.

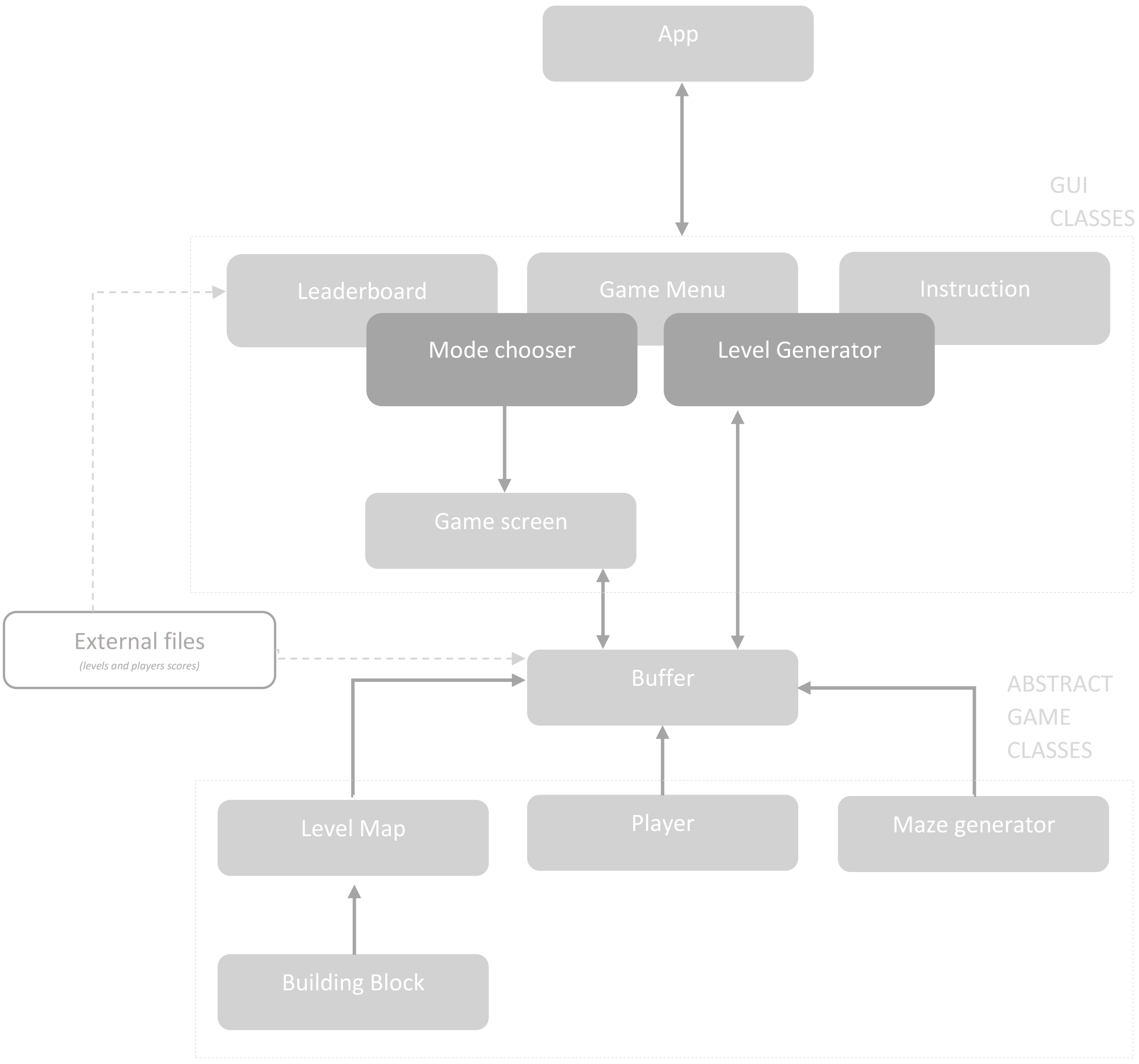
Next, the **GUI** module is also updated to add the **LevelGenerator()** class, which displays the screen to allow the user to generate a level using the **maze_generating_function**. It makes use of the following methods: **save()** simply saves the level to a txt file, **add_option()** adds the settings on the canvas, **update_number()** manages the transformation of setting numerical value into graphical displays, **change()** allows for the modification of setting numerical values by a given step, and **generate_and_display()** runs the algorithm and displays it on the side.

Since we decided to add the Adventure game mode, we also created a **ModeChooser()** class which is used when clicking on <START GAME> in the main menu, and allows the user to pick the game mode. This happens in a new canvas. This class also uses the **choose_levels()** method which used to be in the **App()** class since version_0.

The **gameplay** module is left almost unchanged in this version.

Version 3

During the development of version 2 we discovered that the boundary between abstract game classes and GUI classes is a little blurred, therefore implementation of further functionalities may be difficult as one may not know where new methods should be written. Since expandability is one of our objectives, we decided to restructure the program architecture a little so as to make the distinction between these two groups more visible. We decided to add a new class - **Buffer()**. Its main purpose is to move all the calculations made regarding the current game into one class that is not responsible for displaying information. In that way **GameScreen()** class sends requests to **Buffer()** to perform certain actions like level generation / player movements / block destruction etc., then **Buffer()** applies these changes and allows **GameScreen()** only to display current state of the game. **Buffer()** class also combines the abstract gameplay classes (**Player(), LevelMap()**) so there is no need to store data about level map or block size in **Player()** class. This way the **Player()** class is more universal. We also identified further redundancy in our code. For example similar level generation was performed in **GameScreen()** class as well as in **LevelGeneration()** class so we decided to standardize level generation and perform it in **Buffer()** with the same method, which is in turn used by both classes. By implementing **Buffer()** we made it easier to implement changes in the future in case we need to change the way of displaying levels without changing how the game itself operates. Final program architecture was described on a following diagram:



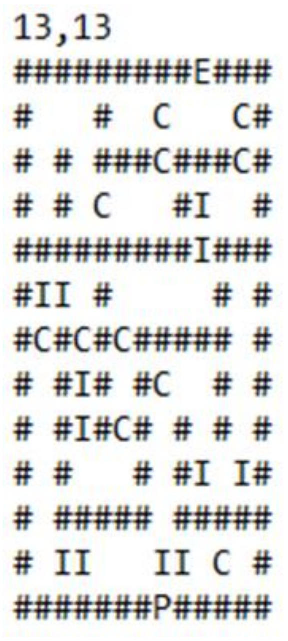
Level files

Levels are stored in text files for simplicity and ease of manipulation by any users. All the levels are placed in /levels/ folder in working directory of an application. Here is how to read a level file:

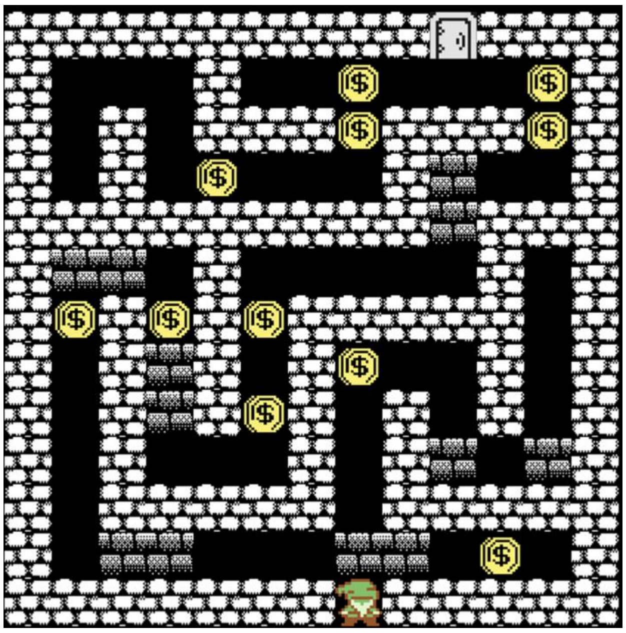
- the first line tells the application the size of the maze in width x height (number of blocks), divided by a coma
- then the maze itself is represented line by line, column by column
- '#' represents walls (blocks that can't be broken)
- ' ' represents the empty hallways in which the player can move
- 'P' represents the player's starting point
- 'E' represents the exit door's location
- 'I' represents the obstacle blocks that can be destroyed

- 'C' represents the coins that the player needs to collect

Example of how simple text file is displayed by an application:



```
13,13
#########E###
#   #  C   C#
# # ###C###C#
# # C   #I  #
#########I###
#II #     # #
#C#C#C##### #
# #I# #C  # #
# #I#C# # # #
# #   # #I I#
# ##### #####
# II   II C #
#######P#####
```



Instructions

When launching the application, you'll end up on the Main Menu. From the latter, you can access the following game features by clicking on their respective buttons:

- <START GAME>
- <Instructions>
- <Generate Level>
- <Leaderboard>
- <Exit>

<START GAME>:

Clicking on <START GAME> leads to the game mode selection page, where the user can choose between two ways to play the game.

1. Adventure Mode: this mode challenges the player to complete as many levels as possible in 3 minutes. When a level is finished, the application will generate a new one automatically and the player can play until the timer runs out. The goal is to collect as many coins as possible.
2. Solo Mode: this mode lets the player play a specific level of his choosing by loading from files. It still tracks time and computes a score.

When clicking on either mode, a new window pops up allowing the player to enter their name for the leaderboard, as well as loading up a level from file if Solo Mode was selected. If no name is selected the score will be saved as "unknw". Clicking <Start Game> will launch the actual game screen.

In the game screen, the player is represented by a green gnome sprite. The player starts at the maze's starting place, which depends on the level. The goal is to move around the maze, collecting every coin in the level in order to open the door and exit the level. The commands allowing you to control the game are the following:

- <W>, <A>, <S>, <D> keypress: move up, left, down and right respectively (use QWERTY for ease)
- <E> keypress: break a destroyable block
- <P> keypress: pause the game

The coins are automatically collected when the player goes over them. You can exit the game at any time by clicking the return button at the top left corner.

A timer is displayed on the top right corner and counts down from 3 minutes down to 0 (units are displayed in seconds). Pausing the game will stop the timer and prevent the player from moving.

Once you collect all coins in a level the door opens and the player can exit. If Adventure mode is being played, a new level is automatically generated. If Solo mode is being played, the end window pops up and informs the player, displaying their score and allowing them to go back to main menu. Once the timer runs out, the end window pops up as well in both mode.

<Instructions>:

Clicking on <Instructions> will display a screen summarizing the purpose of the game as well as the commands using the graphical theme. A narrative story introduction also displays on this screen and participates to creating the atmosphere of the game.

<Generate level>:

Clicking on <Generate level> leads to the level generation screen. The user can build their own level using desired settings, see what the generated level looks like, and save it in a file. To generate a level, the user can adjust the 4 following settings:

- Width: this is the width of the maze in number of blocks (limit: 9-51 blocks)
- Height: this is the height of the maze in number of blocks (limit: 9-51 blocks)
- Number of coins: this is the number of coins to be collected in the maze (limit 0-99 coins, but if there is not enough room the number will adjust down). If no coins are added, the exit door is already opened.
- Number of obstacles: number of obstacle blocks that can be destroyed by the player (limit 0-99 blocks, but if there is not enough the number will adjust down)

Once the user is satisfied with the settings, clicking on <Generate level> will launch the generation and display the result on the side. If user wishes to use this level, they can save it by clicking on <Save level> (the file will be saved with the following format "gen_lvl_date_time").

<Leaderboard>:

Clicking on <leaderboard> will lead to the leaderboard screen where a player ranking is displayed based on final scores. Only the top 9 scores are displayed.

The function used to compute the final score is the following:

$$Final\ score = Max[0, (Nb\ of\ collected\ coins \times 5) - (Time\ limit \div 10) + Nb\ of\ obstacles]$$

As the game is about collecting as many coins as possible, each coin is worth 5 points. However, the more time the player has, the more coins he will probably collect. Therefore, we included a time penalty of 1 point every 10 second. Finally, the number of destructible blocks makes it harder for the player to reach the exit and collect coins. Thus, the more blocks placed in a level, the higher the score.

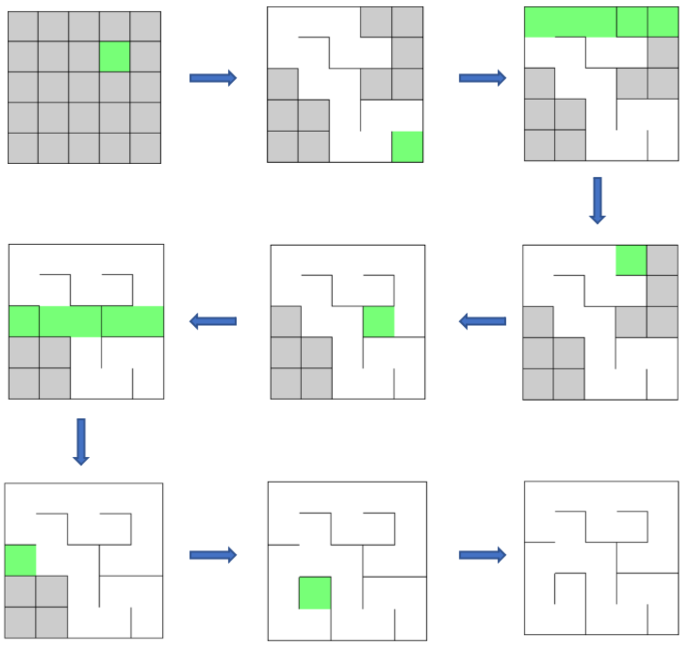
<u><Exit>:</u>

Clicking on <Exit> this will close the application.

Hunt and kill algorithm for maze generation

Similar to the Recursive Backtracker, the Hunt and Kill algorithm can be used to randomly generate a maze. In a nutshell, it works as follows:

1. Randomly mark a cell of the maze as the starting location.
2. Start in kill mode from this cell: perform a random walk by connecting and marking an unvisited neighbor, until the current cell has no unvisited neighbors.
3. Switched to hunt mode: scan the maze line per line, from left to right and from top to bottom, looking for an unvisited cell that is adjacent to the current marked cell. If found, connect the two cells, mark the formerly unvisited one and let it be the starting point of the next random walk of the kill mode.
4. The algorithm repeats these instructions until all the cells are marked.



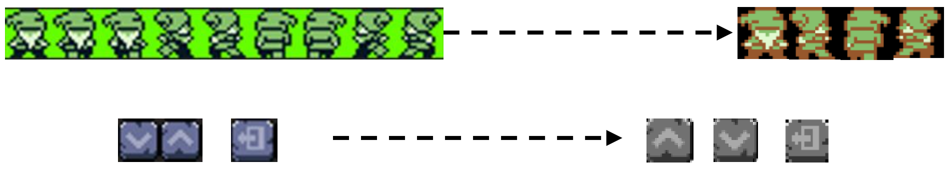
Reasons for choosing the Hunt and Kill algorithm:

1. It guarantees the integrity of the maze, which means that regardless the maze scale or the starting points, every cell in the plane will be included into the final maze.
2. The random walk phase tends to produce long, winding passages with fewer dead-ends compared to other algorithms.
3. Compared with the Recursive Backtracker algorithm that uses the stack, the Hunt and Kill algorithm does not use any data structure to record the connection relationship between points, which reduces the storage space required for the maze.

Graphics and sounds

Being aware of the graphic limitations that non-profit projects have and our lack of experience in this area we agreed on doing a whole game in pixel art style. In this way we had access to wide variety of free resources on internet that we mostly got on free license from itch.io web page and we created the rest of required graphics from scratch. In this way, we were able to adjust the graphics we found to the desired colour palette without investing overwhelming amount of time to this part of project. Every transformation was made in Adobe Illustrator software. Thanks to that we could kept the project files and easily change some part of already rendered graphics if it was necessary. We chose the black and white style, so most of the game elements are made in a two-bit colour scheme. However, to emphasize the important elements of the game and create a user-friendly interface, the most important elements, such as the player's figure or the coin model, were made using contrasting colours to make them easily visible on the screen.

Examples of colour adjustment done by our group:



Font used in game (MinimalPixel Font) was created by Mounir Tohami and was also acquired from above mentioned web page.

In the last version of the project, we also decided to add sound effects. Their presence increases the immersiveness of the experience, making the game seem more consistent, which coincides with one of our goals of creating an end-to-end project.

Similarly, sound effects were added to the game using a library of free license soundtracks from freesound.org, which are made by amateurs and enthusiasts. In order to limit search time for this part of the project, we decided to settle quickly for each specific piece out of a limited explored sample. This leads to all the sounds not having a lot of musical coherence between them, but for the scope of our project this was perfectly reasonable.

Finally, one piece of music was obtained from music composer Eric Matyas and his website of free soundtracks found at soundimage.org. A section of his website was aimed specifically at looping music, which is ideal for videogames. We selected the following track:

- Arcade Fantasy, by Eric Matyas -> used as actual game soundtrack

And the other piece of music was obtained from a free video game asset website at void1gaming.com. We selected the following track:

- Palm Tree Shade, by VOiD1 -> used for main menu soundtrack

Conclusions and ideas for further development

We are quite satisfied with how our project turned out. Indeed, most of our objectives were reached, especially the goals of making an easy to scale-up and self-sufficient project.

We have a few ideas on how we could improve the project, among which:

- Adding enemies that pursue the player in the maze. Hitting them causes the player to take damage, and upon a critical level of damage (3 hits for example) would lead to a 'Game over'.
- Configuration: allowing the user to change some options like the resolution of the application, the music/sound levels, the input method (AZERTY for example, or controller).
- Difficulty settings: allowing the user to choose a difficulty level, which would have an impact on the time limit and the number of obstacles (and number of enemies/amounts of damage given)
- Help functionality, where the user can be shown the path to the exit if confused (already implemented in mazelib, would only need to be translated in our program's structure and language)
- Other block types: traps (spikes, holes), treasure chests giving player health or other bonuses