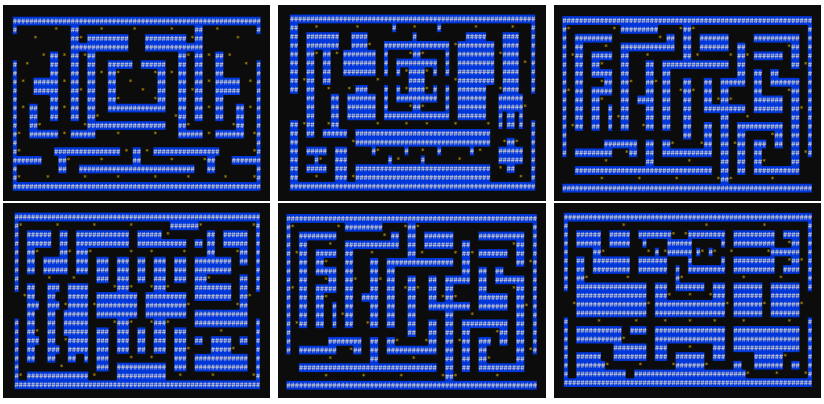
Eternal Run: developing a game in C++ with the ncurses library

1 Introduction

Eternal Run is a video game developed in C++ language that uses the ncurses library to create an interface playable via terminal. The player plays the protagonist who must run through maze-like maps, collect coins, defeat enemies and reach the next level.

2 How the game works

The game features a graphical representation in ASCII format and is composed of a total of 6 maps, which are repeated in a random order during the game.



The 6 maps of Eternal Run.

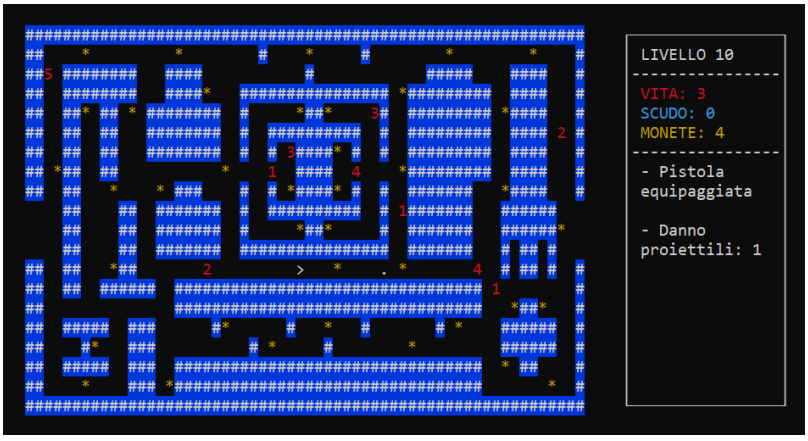
The objective of the player, who plays the protagonist, is to collect coins, defeat enemies and reach the next level. The game also includes market levels through which it will be possible to increase the hero's abilities, making him stronger and more prepared to face the challenges of the labyrinth. Among the various upgrades that can be purchased, we find the increase in the quantity of lives, up to a maximum of 3, the increase in the value of the shield, the acquisition of a gun, with which crossing the levels will become easier, the Increased bullet speed, which can only be upgraded once, and the damage they cause to enemies, who have an amount of life based on their level.

Market level.

The graphic interface is made up of two windows: one is the main one, in which the maps with the coins, the enemies and the protagonist who must cross the labyrinth are shown; the other is the information window, which contains the current level, current statistics, including life, shield and coins, and various information on the upgrades obtained.

Upon contact with enemies, the protagonist will lose a life and will be forced to restart the level with a new randomly generated map. However, it is possible to avoid this situation by purchasing shields from the market, which allow you to come into contact with enemies without suffering loss of life.
If all lives are lost, it will not be possible to continue and a game over screen will appear, from which it will be possible to decide whether to abandon the game or start a new game.

The player also has the ability to return to previous levels and markets by going through the entrance doors. This allows the protagonist to collect coins and defeat the remaining enemies, in order to increase his economic availability on the market.



In-game example.

3 Game instructions

At the start of each level, the player is placed inside the front door. To start the game, you need to click on the directional arrow corresponding to the direction in which the protagonist is initially located. Once the game is activated, it will be possible to use the 4 directional arrows on the keyboard to move the character within the map.

Once the gun has been acquired from the market, it will be possible to use it using the space bar as a trigger, allowing the protagonist to shoot the enemies encountered along the way.

Furthermore, the player always has the option to exit the game at any time by pressing the escape key. This will bring up a screen asking for confirmation that you want to exit, ensuring that the player does not accidentally lose their progress.

In the market and in the various menus of the game, it is also possible to navigate between the various items using the up and down directional arrows, and select the desired options by pressing the enter button.

4 Implementation Choices

The video game is organized within a folder in which we find various files and subfolders.
Among these there is a README file, which provides a brief description regarding the use and execution of the video game.
We also find a makefile that allows you to compile the source code using a simple "make" terminal command. In this way an executable file with the name "eternal_run" will be created, which will contain the complete game which can be started by double clicking on the executable or through the terminal command "./eternal_run".
There is a folder called "storing_files", which will be used to store the save files generated during the game. These files will allow you to return to previous levels and continue the game where you left off.
Finally, within the game files we find the "src" subfolder, which contains all the source code divided into individual files. These are organized to ensure an orderly structure and easy understanding of the code. The structure is as follows:

- **main.cpp**
  In the game's main.cpp file, only the main() method has been retained to maintain the readability and maintainability of the code.

This method takes care of preparing the ncurses library, which provides textual input/output functionality for the game interface, and showing the splash screen, i.e. the initial screen of the game. Subsequently, the 2 game windows are created and the game loop is started, which represents the heart of the game itself. The methods used to do this will be explained later since their declaration is present within the other files.

- **graphics.hpp** and **graphics.cpp**
  Interactions with the ncurses graphics library are handled through the graphics.hpp and graphics.cpp files.

  Here we find the global constants and methods that allow us to interact with the ncurses library. In particular, the constants include those that concern the pair indices, i.e. the color combinations used within the application. These constants are critical to ensuring that the colors used are consistent when running the software. Furthermore, there are also constants that describe the indices of the market menu options, allowing easy management of the options offered to the user.

  The methods in the files are divided into four main sections.
  The first section includes general methods, such as the one for creating colors, which are fundamental for correct graphic display within the software.
  The second section concerns the standard screen, where there are functions that work with the input/output within it. Among these we find, for example, the one that shows the splash screen, or the initial screen of the software.
  The third section includes methods that deal with the game screen, which is the main screen where the maps, player, enemies and projectiles are displayed, as well as the market level, where users can purchase upgrades. Many of these methods work by looping through a list of Entity class objects and checking their current x and y positions to display them correctly on the screen. There are also methods that allow the exit and game over screens to be displayed on the screen once invoked.
  Finally, the fourth and final section includes methods that interact with the information window. These, many times, refresh the statistics on the screen to ensure that they are always updated to the current values. For example, if the player collects a coin, the window must be refreshed to show the updated number of coins in possession.

- **entities.hpp** and **entities.cpp**
  The entities.hpp and entities.cpp files are fundamental for managing the dynamic elements of the game, namely the Entities.
  These files contain global constants, classes, structures and methods that allow you to manage objects that have the ability to move within the map.

Global constants are used to identify the four possible directions for the Entities, namely right, left, up and down. These constants are essential for the correct functioning of their movements.

Class management was implemented through a subclass system:
The parent class "Entity" allows you to create generic objects capable of moving on the map and has the y and x position attributes, in addition to the attribute that indicates the direction of the entity. This class was used as the basis for creating two subclasses, namely Player and Enemy.
The Player class represents the protagonist of the game and contains specific attributes and methods for managing the player's upgrades and actions.
The Enemy class allows you to create enemies and has specific attributes and methods for managing their power and interactions with the player.

Finally, the entities.hpp and entities.cpp files also contain two structures that allow you to create lists useful for managing enemies and projectiles, with methods that allow their addition and removal.

- **map.hpp** and **map.cpp**
  The map.hpp and map.cpp files are important components of the in-game map management system. They contain global constants, structures and methods that serve to represent and manage the 6 maps present in the game.

  A single map is represented by a structure called "map" which contains a two-dimensional 20x60 array, used to store the contents of each single block of the map, as well as the positions of the entrance and exit, which are fundamental for moving from one level to another. the next or previous one.
  The contents of the array are filled with numerical values ranging from -1 (empty block) to 3, in order to represent entry, exit and wall blocks. The use of an array and not a list to store the blocks was chosen to improve the performance of the game, since every time an entity moves within the map, it will be sufficient to check the next block in the array via its index, composed of y and x position, to determine if it is possible to move there, without the need to search through a loop for the corresponding block within a list.

  In addition to "map", we also find a structure used to build a list of coins, which can be collected by the player.

  The methods in these files include creating the 6 maps and 6 coin lists, as well as more specific methods for adding or removing individual blocks and coins from the list.

- **storing.hpp** and **storing.cpp**

The storing.hpp and storing.cpp files handle storing the state of previous levels, thus allowing the player to return to them by passing through the map entrance.

This functionality is made possible thanks to the "storing_files" folder, which is created when moving to the next level if it does not already exist. Three different files are saved within this folder for each level, each of which contains specific information about the coins, enemies and the corresponding map.
The first file contains the location of every coin remaining in the map. The second memorizes the level of the enemies remaining in the map, while their position is not saved as they are reinserted in a random location when you return to the previous level. Finally, the third file contains the index of the map corresponding to the level, which allows it to be correctly taken from the array made up of the 6 game maps present in the game.cpp file.

The two "storing" files contain several methods, including creating the storage folder and files, accessing the information contained in them, and deleting all files once the game is closed or the game is accessed over. These methods provide an efficient and well-organized solution for managing the state of previous levels, making it possible for the player to pick up their adventure where they left off.

- **game.hpp** and **game.cpp**
  The game.hpp and game.cpp files represent the backbone of the project, as they contain all the methods that allow the game to function correctly by connecting all the components already mentioned.

  In these files, global variables that record the progress of the game are saved, such as that of the player, the list of enemies and bullets, the 6 maps of the game and other variables related to statistics.

  The methods present include those for creating a new game, reloading a level from storing files, generating new levels randomly, managing the movement of the protagonist, enemies, projectiles and their collision.

  The most important method in these files is the one that handles the game loop, as it "brings life" to the game. This method consists of a while loop that runs until the user exits the game. A boolean variable is used as a guard for this loop. Within it, if the user has provided input, the latter is saved in a variable which is controlled via a switch to determine the next move to perform, such as changing the protagonist's direction or shooting.
  Subsequently, the positions of all the entities are updated and finally napms() is called to stop the execution of the program for several mil-

liseconds in order to obtain a smooth and manageable movement for the gamer.

```
while (!exit_game) {
      char input_char = wgetch(game_win);

      if (input_char != -1) {
          switch (input_char) {
              houses KEY_DOWN:
                  // move down
                  break;
              case KEY_UP:
                  // move up
                  break;
              ...
              case KEY_SPACEBAR:
                  // shoot
                  break;
              case KEY_ESC:
                  exit_game = true;
                  break;
          }
      }

      // move entities and refresh

      napms(80);
  };
```

Simplified game loop code

- **utils.hpp** and **utils.cpp**
  The utils.hpp and utils.cpp files contain methods that perform secondary tasks but help keep the code tidy and easily maintainable.

  Among these methods we find those that have the task of carrying out checks on the movements of the Entities, such as that for the possibility of moving within a certain block or that of calculating the next position of a certain object.
  There are also methods that allow you to clean the pointers within the lists, in such a way as to free the memory used by them once they are no longer needed, preventing them from accumulating in memory. This helps keep system memory free and available for other tasks.

5 Division of labor

- **Matteo Lombardi**: interactions with the graphics library, creation of maps, implementation of entities, implementation of game dynamics, writing of the report.
- **Enis Brajevic**: saving the state of previous levels, creating maps, implementing entities, implementing game dynamics, writing the README.
- **Lorenzo Molinari**: creation of entities (player, enemies and projectiles).
- **Christian Orsi**: Created market level, implemented upgrades.