University of Dhaka

Department of Computer Science and Engineering

Project Report

Fundamentals of Programming Lab(CSE-1211)

Project Name

The Labyrinth

Team Members

Mahdi Mohammed Hossain Noki (AE-02)

Mohima Ahmed Joyee (SK-42)

Introduction

This report discusses the results of the work done in development of “The Labyrinth” on Windows platform using C++ programming language along with SDL2 library. It is a 2 dimensional (2D) single-player role-playing adventure game where the player moves the character through a labyrinth trying to escape it while overcoming obstacles along the way. The game is set in medieval Europe. It has different mazes for the player to play. It has intelligent computer-controlled enemies to challenge the player along with interesting objects like gates, switches and chests.

(N.B. This report uses “we” to refer to the project contributors. A clear distinction is to be made between the words “player” and “character”. The “player” is the human being interacting with the game from the real world, whereas, the “character” is the in-game player avatar that is being manipulated.)

Objectives

The principal objective for this project for the developers was to learn the basics of computer programming and development, utilize new libraries like SDL2 and SDL image etc. We created a fun action RPG puzzle-solving game for all ages reviving the classic gameplay styles of our childhood.
We intended to create modules that are easy to implement and are reusable. The game is simple but challenging, and thoroughly enjoyable.

Project Features

What makes this project interesting is that it utilizes some of the most interesting 2D action RPG game features.

The map is rendered entirely as tiles. Each square tile placed can be walkable, non-walkable or interactable. This map rendering gave the game some massive versatility when it comes to map design. The project includes a **map reader tool** that can read a map designed by a human and convert it into a file that the game can read to render the map in game.

The enemy is fairly smart and computer controlled. It is always on the lookout to establish a line of sight to attack the player.

The character can move in the 2D map anywhere that is walkable. Super smooth movement allows the player to expertly maneuver the character around the enemy to dodge attacks and position himself in an advantageous position.

One of the most interesting things that happens behind the scenes is how the game handles events. Every entity, before rendering itself, handles the events from the event pool. While actions like keyboard press or mouse tap generates events the computer can recognize, some actions like enemy getting damaged or player opening a door does not generate recognizable events. So the game exploits SDL's user defined event. Every action not recognized by the libraries generates a user event that the entities respond to.

Music in the game has its own channels and sound levels. Players can control the background sounds and SFX sounds separately. Player, enemy and every other sound generating entity has their own dedicated channel to play sound on, each with their own sound levels.

Some of the features the player experiences are as follows:

Menu and Pause Screens

Menu Screen

When the game is run, the menu is the first screen that the player sees. It contains five options for the player to interact. The start menu has an infinitely scrolling background with multiple layers, giving the illusion of distance. The logo and the buttons displayed are all custom-designed.

Figure: Menu screen

What each option has to offer is described below:

1. **Start**

   A new game starts. The game begins from Level 1.

2. **Levels**

A list of levels shows up and the player can select a level they want to play from the list.

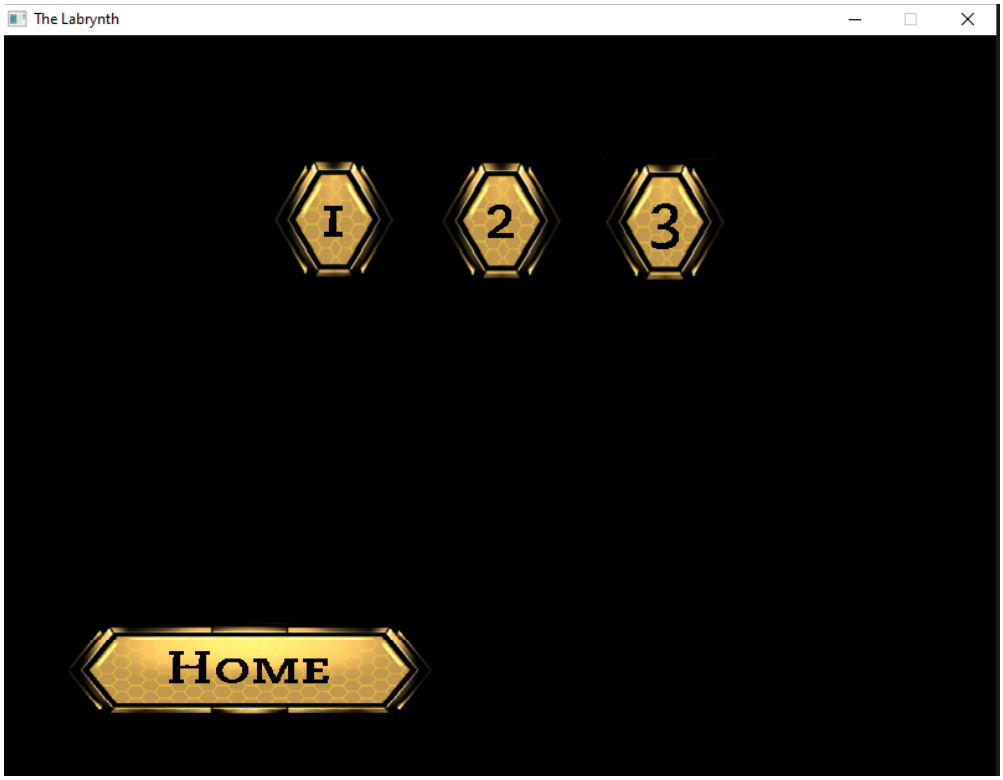

Figure: Level Selection Screen

3. **Options**

A list of options is displayed. There is an option to increase or decrease the volume of music, an option to increase or decrease the volume of SFX, and an option to toggle full screen. The player can access the instructions and credits from here.

The instructions screen offers a detailed set of instructions to play the game, also explaining various game features.

The credits screen scrolls up a list of the game developers and their credentials.

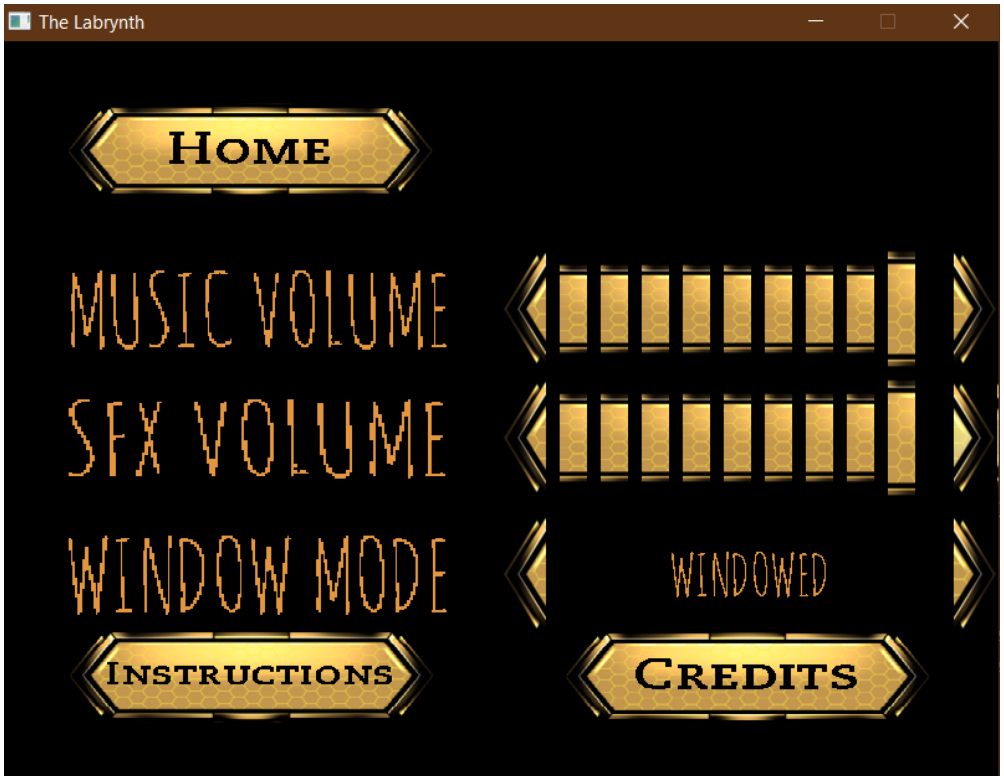

Figure: Options Screen

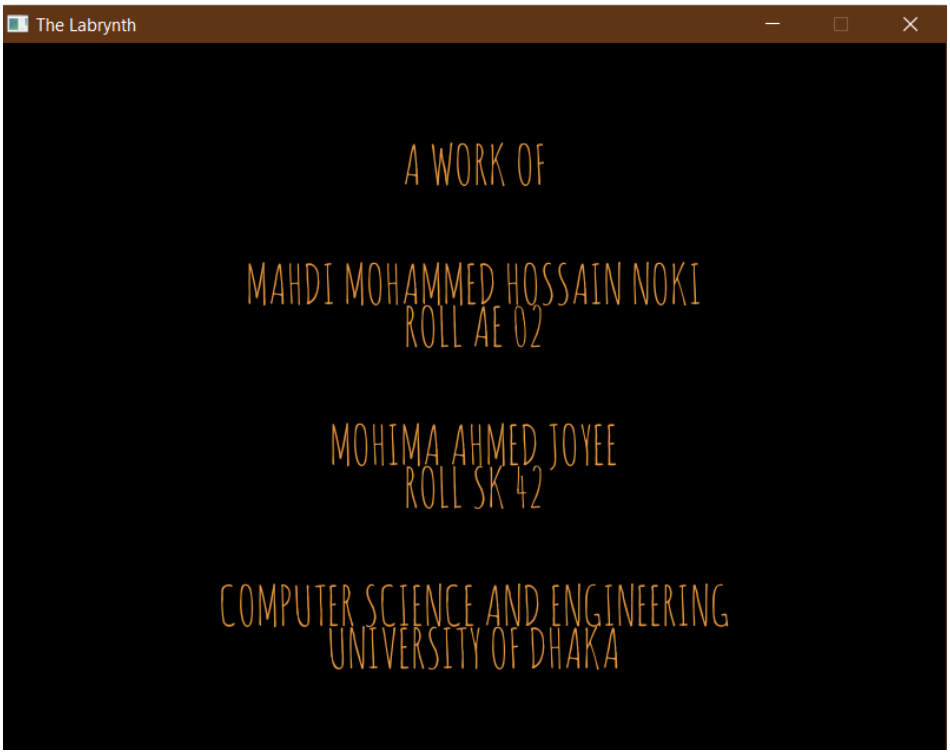

Figure: Credits Screen

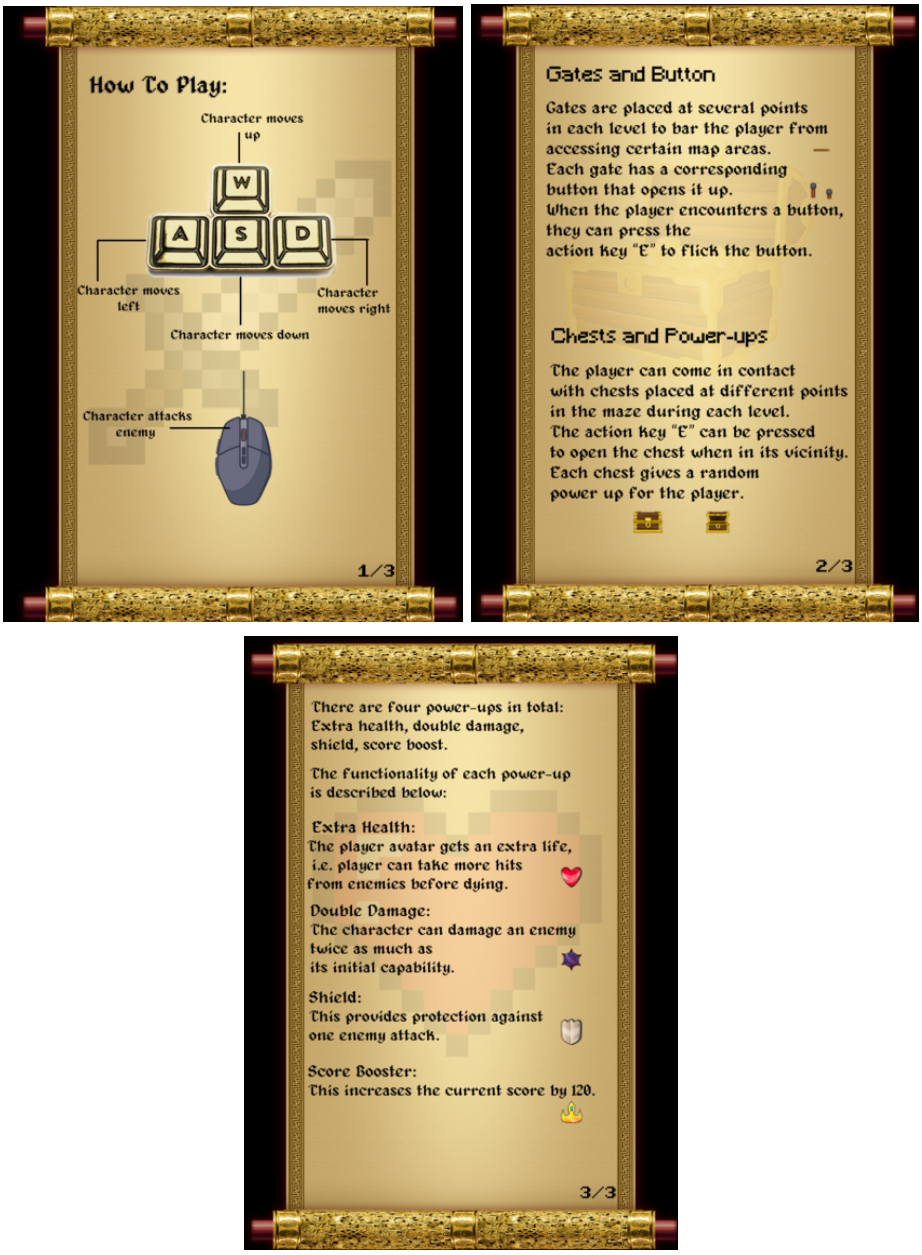

Figure: Instructions set

4. Highscores

The highscore table shows up. The player can see their level-wise highscores in this section.

Figure: Level-wise high-scores

5. **Quit**

   The application closes.

Pause Screen

Other than the startup screen, in between a game, i.e. when the player pauses the game, two options show up. The player can pause the game using the “Esc” button.

1. **Home**

   This option takes the player to the main menu.

2. **Continue**

   This option resumes a paused game.

The player can see their current score when the game is paused.

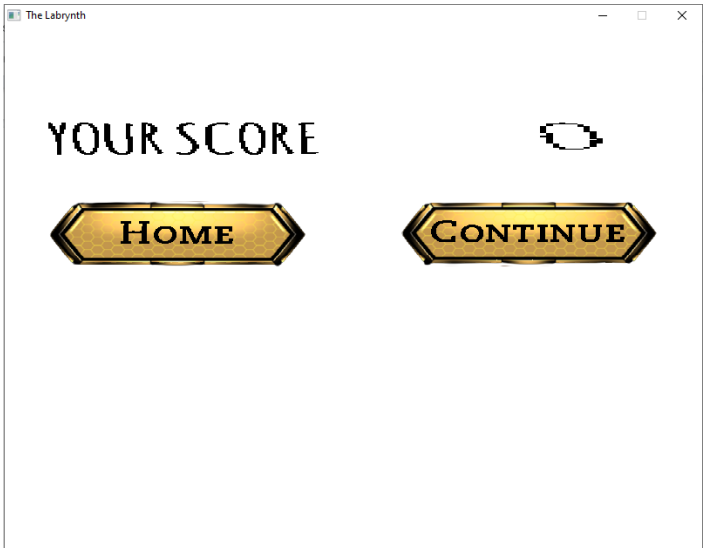

Figure: Pause Screen

Level Design

1. Camera

The game follows the main character wherever the player takes him. At any point during the game, the player can only see a small part of the entire map. Camera tries to keep the character at the center.

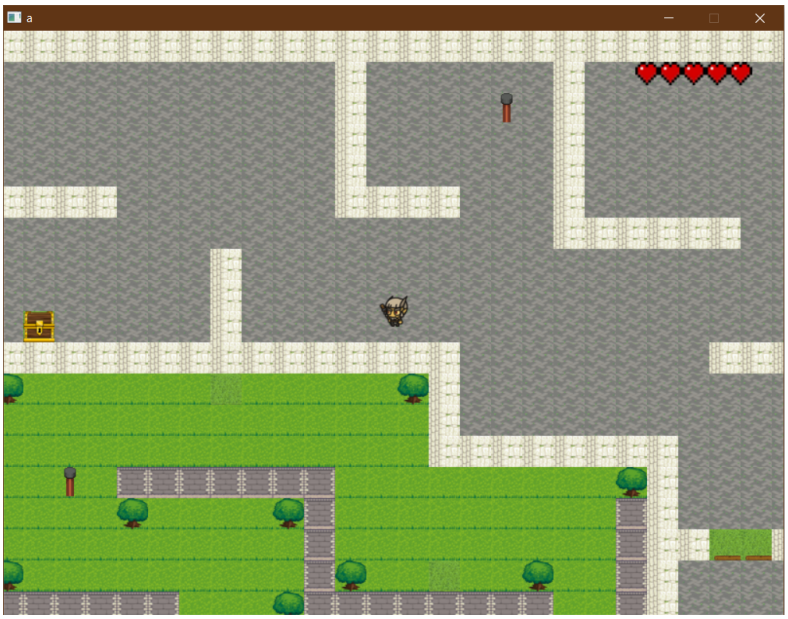

Figure: Camera positioning

2. Tileset

The game renders the whole map as individual tiles. The following atlas has been used for designing the mazes in each level. The top row contains the non-walkable tiles and the bottom row contains the walkable tiles. Tiles that are outside the camera range are not rendered. Tile data is read from a file when a level starts.

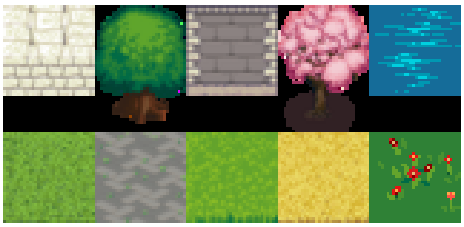

Figure: Tile Atlas

3. Gates and Button

Gates are placed at several points in each level to bar the player from accessing certain map areas. Each gate has a corresponding button that opens it up. Gates add a certain amount of difficulty to each level. When the player encounters a button, they can press the action key “E” to flick the button.

Figure: Button unpressed/pressed

Figure: Gate

Figure: Unpressed button

Figure: Pressed button

Figure: Gate closed

Figure: Gate opened

4. Chests and Power-ups

The player can come in contact with chests placed at different points in the maze during each level. The action key “E” can be pressed to open the chest when in its vicinity. Each chest gives a random power up for the player.

 Figure: Chest closed/opened

There are four power-ups in total: Extra health, double damage, shield, and score boost.

The functionality of each power-up is described below:

Extra Health: The player avatar gets an extra life, i.e. character can take more hits from enemies before dying.

 Figure: Extra Health icon

Double Damage: The character can damage an enemy twice as much as its initial capability.

 Figure: Double Damage icon

Shield: This provides protection against one enemy attack.

 Figure: Shield icon

Score Booster: This increases the current score by 120.

 Figure: Score Booster icon

Characters

1. Main Character

Figure: The main character

This character has five animation states: idle, walking, attacking, getting damaged, and dying. It can move in a 2D plane and attack enemies. It can also interact with certain map elements.

The following figure illustrates the inputs to control the character.

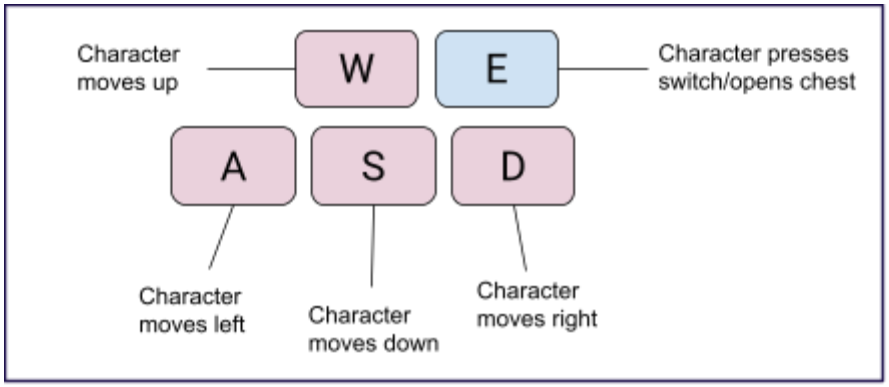

Figure: Keyboard Inputs

The character has a fixed number of hit points at the beginning of each level. It has five lives at the start, which is displayed at the top-right corner of the screen. It can only cause a fixed amount of damage to the defeatable enemies with each hit within melee range.

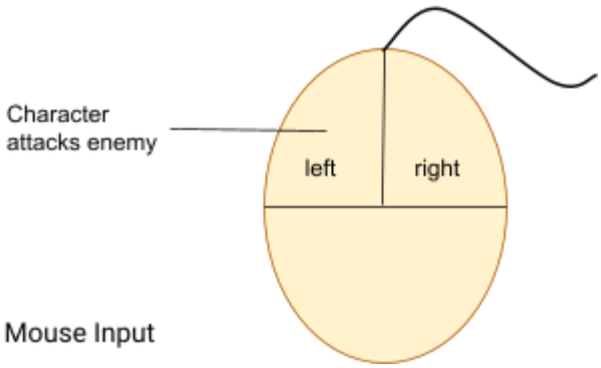

Figure: Mouse Input

Players can open doors and reward chests on the map. Player interacting with those generates user defined events that the tilemap handles.

2. Enemy

The enemy has five animation states: patrolling, attacking, getting damaged and dying. It moves in a straight line seeking to establish a line of sight with the character.

Each enemy follows a fixed route of patrol, i.e. either up-down or left-right. They each have a fixed number of hit points at the beginning of each level, a fixed patrol route length and a fixed attack range. It's health is displayed on top of its head.

The enemies are computer-controlled. They throw projectiles at the character at a constant interval upon establishing a line of sight. It uses Bresenham's line drawing algorithm to try to establish the line of sight. When an enemy projectile hits the player, the enemy generates a user defined event which the player handles.

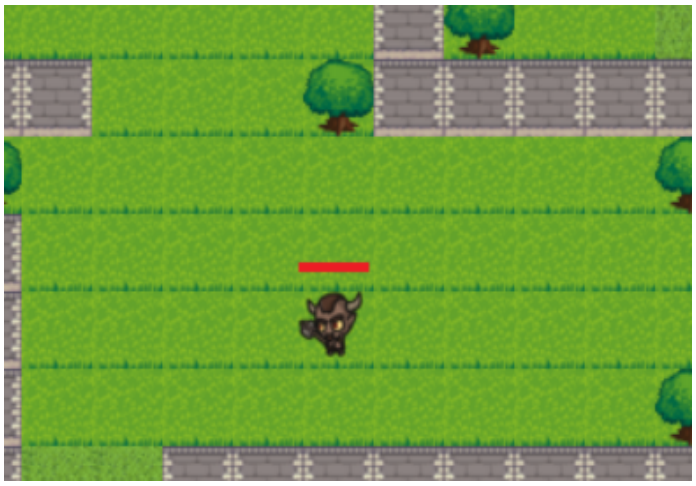

Figure: Enemy patrolling (health bar on top of its head)

Figure: Enemy throwing projectile upon establishing line of sight

Project Modules

---

1. game.h

This header file is inarguably the most important header file in the game. This consists of all the materials required to get the game running. It has two structures in it, Game and Score.

After declaring a Game variable, it must be initialized by the init() function. init() initializes SDL libraries with video and audio, SDL_image libraries with png, TTF and SDL_mixer libraries. It also initializes the window and a renderer to render on the window.

At the beginning of every level, level_end must be set to false and set_level_dimenstion(int, int) needs to be called to set the current level dimensions.

The struct has an enumeration list of screens, button IDs and a function to define what button actions will be, a list of sound channels and a list of levels completed.

The Game structure has an Events struct under it, defining user defined events. The Event structure has to use the create_event(Sint32, int*, int*) function to push an event in the event pool. The reset() function must be called after handling each event to clear the event.

The structure can have some TTF texts preloaded for showing in the pause screen. text_loader() loads these texts.

texture_loader() is used to create a texture from the surface and text_render() renders some TTF text.

The player can resize the window using the resize_window(int, int) function. It takes width and height as arguments.

read_score() and write_score() reads from and writes to the scoreboard saved on the player's computer.

The toggle_fullscreen() function is used to allow the player to make the game full screen.

camera_set() must be called before rendering player or tiles to make sure the camera stays on the player avatar and always renders it.

Volume control functions are used in the options menu to control game volumes. music_volume_control(char) and sfx_volume_control(char) takes 'i' or 'd' as arguments to modulate volume.

The collision(SDL_Rect*, SDL_Rect*) returns whether two rectangles collide(AABB collision).

The Score structure is used to render the score of the player at a given time. It has to be initialized with a Game variable and the score to render. set_height() can be called to change the height at which the score gets rendered.

Two functions, render() and render_countup() render the score on the screen. The second one starts counting up the score from 0.

2. button.h

This handles events and renders a button. Buttons have their own tasks as moderated by the Game structure. This header file has the structure Button that holds the texture for the buttons. This structure can detect whether the player is hovering the mouse over the button or has clicked the button. Each button has three states - default, hovering and clicked. If the user has clicked a button, this structure asks the Game structure to execute the action the button was supposed to do.

3. enemy.h

This is the most complex header in the game. It only has the Enemy structure in it. Enemy has 5 animation states and a predefined route to patrol. A variable under this structure has to be initialised with enemy width, enemy height and each of a game object, a player object and a tile object.

After declaring an Enemy variable, loadFromFile() must be called to load the animation states, the projectile weapon and the sounds. set_spawn() must also be called to set the enemy spawn point in the map.

In each render cycle, the handle_event() and the render() function must be called in order.

handle_event() handles the events the game throws at the enemy. The most complex event is to determine the line of sight with the player. The structure has another structure in it called LineOfSight to implement Bresenham's Line Drawing Algorithm. LineOfSight::established() returns a boolean value - whether line of sight is established.

handle_event() also checks for events such as damage from the player and makes the enemy attack player when prompted. Enemy launches a projectile that travels in a straight line. There is a Projectile structure storing the weapon's information. handle_event() calls the function attack() when line of sight is established, which in turn

calls launch_projectile() or move_projectile() to launch or move the weapon in a line. Enemy attacks persistently with an interval of 4s. If the projectile collides with the wall or travels a certain distance, it stops and teleports back to the enemy. If the projectile collides with the player, it raises a player damage event which the player handles.

handle_event() function can also call move(), get_damaged() or die() functions to animate and move enemy is corresponding states,

The render function renders the enemy onto the screen.

4. background.h

This header file has only one struct in it, Background. This is a pretty simple module. It can load upto 3 layers of background and scroll them infinitely. Different scrolling speeds allow an illusion of distance. This is only used in the menu screen.

Any Background variable must be called with the loadFromFile(), set_vel(), set_rect() and set_width() functions. Only the loadFromFile() takes 3 file paths as arguments to know the paths to three layers of scrolling backgrounds. The other functions set the velocity, render rectangle and width of the background respectively.

5. init.h

This header only has the header files included and some preprocessors defined.

This must be included anywhere any C++ or SDL functions are to be called.

6. player.h

A very important header hosting the Player structure. Any Player variable must be initialised with player width, player height and Game and Tile objects. The loadFromFile() and set_spawn_point() functions must be called to load in the sprite sheets and set a spawn point for the player on the map.

Player has functions rect_collissions(SDL_Rect) to detect AABB collisions with other rectangles and center_rect_collission(SDL_Rrct) to detect whether it’s center point is inside some rectangle. Both returns boolean.

In each render cycle, handle_event(SDL_Event) and render() must be called in order to allow the player to handle events before rendering. handle_event() detects whether a player is walking, standing, attacking, getting damaged or dying and makes the renderer

render the corresponding animation state. When the player attacks, this raises an event for the enemy to handle.

render() renders the player onto screen.

7. screen.h

Game has a number of screens - the UI screen, instruction screen, levels screen options screen, high score screen and credits screen. This header file renders the screen the main function asks it to. It takes Game* as an argument for each screen rendering.

This function also renders the game levels when called.

8. tile.h

This header has the Tile struct. This makes the map render as individual tiles, allowing incredible versatility to the map and the developers. It has to be initialized with tile width and height, and the Game object.

the loadImageFromFile() and loadInfoFromFile() must be called to load the tile atlas and also load information about what tile to render and where.

It has various collision detection functions - tile_gate_wall_collission, tile_chest_collission, etc with intuitive names, each satisfying their own requirement.

The handle_event(SDL_Event) must be called before render() to allow the tile to handle events like button click or chest open.

render() renders the whole map as tile. The tile_type[][] array stores the information of tiles. To optimize the game, anytile outside the camera is NOT rendered. Only tiles the player can see are rendered on the screen.

Information about tiles is a 4 digit number. For gates and buttons, the first two digits indicate the unique number assigned to each gates-button pair. For other tiles the 4 digit number each corresponds to some unique tile. There are 5 walkable and 5 non-walkable tiles. Buttons and the reward chests are interactable tiles.

9. main.cpp

This initializes the Game variable and runs an infinite loop to render the screen that is required.

10. screen.cpp

This information about what to render on each screen is here. screen.h header file links to this.

11. mapreader.cpp

The map reader tool doesn't contribute to the game at runtime but is a great tool to generate maps. The Tile struct reads tile information from a text file. It would be rather hard for a human to generate a text file of large magnitude. So this map reader tool can take a human generated image and convert it to a readable file to open at runtime.

Team Member Responsibilities

Mahdi Mohammed Hossain Noki

1. Implemented the main character movement, animation and how the character interacts with other entities.
2. Implemented the enemies, i.e. how they move, different animation states and how they interact with other entities.
3. Implemented tilemap, i.e. maps are strictly rendered as individual tiles and maps are read from a text based file, which is in turn made using the map reader tool.
4. Implemented scoring system.

Mohima Ahmed Joyee

1. Designed all level maps, buttons and instructions set.
2. Added music and sound effects.
3. Implemented combat mechanics, i.e. how the main character attacks the enemies and how the enemies attack the main character upon establishing line of sight, and how power-ups affect character stats.
4. Implemented event mechanics, i.e. how different interactions between different entities in the game create new events.

Platform, Library & Tools

Platform: Microsoft Windows

Language: C++

Library: C++ Standard Library, SDL2 Library, SDL2 Image Library, SDL2 Mixer Library, and SDL2 TTF (Truetype Fonts) Library.

Limitations

---

1. The game does not have a loading screen. So the player has to look at some rendition of a previous screen while waiting for a new screen to load its components.
2. The game had limited access to game resources. Developers put more effort into game mechanics than resource creation.
3. The modules are intuitive to understand and implement but are not independent. Most modules are dependent on some other module.

Conclusions

Throughout this project, we have learnt a lot of new things and it has entirely been a new experience for us.

What we have learnt:

1. Basics of game development
2. A better understanding of the programming languages used and new header files
3. Using the SDL2 library to create an interactive application
4. Coordinating with a team
5. Creative thinking and paying attention to details
6. Debugging

Expectations we had in the beginning:

Initially, we expected to use axonometric projection as the camera perspective to create an impression of 3D with 2D assets, but the game was made with orthographic projection instead. The player was supposed to be able to move only on the nodes of the map graph, but the game ended up being a free roaming game.

Difficulties encountered:

1. Maintaining constant communication with team members and ensuring there is no blunder anywhere that might topple the whole project. We had to struggle to work as a team and divide work between ourselves.
2. Implementing modules created by other developers and utilizing its full potential.
3. Learning how new libraries work on the go.

Future plan

At present, this game runs on Windows platform. There are plenty of scopes for us to increase its functionality in the near future.

Some of the future plans include:

1. Making the game cross-platform.
2. Changing the artstyle of the game and customizing it.
3. Adding more levels and mini-games for bonuses.
4. Adding a variety of enemies.
5. Adding more collectables and hence enabling a wider range of power-ups.
6. Adding a player experience-gaining system.
7. Adding the option to change the player's weapon with different attack modulation
8. Adding friendly non-playable characters that can provide missions to the player.
9. Adding pets that will follow the player and help them in combat.

Repositories

GitHub Repository: *https://github.com/master-da/The-Labrynth*

Youtube Video: *https://www.youtube.com/watch?v=7fHzuEOvJrA*

References

We have used the following websites to learn more about SDL2 and to collect free game resources, i.e. soundtracks, background images, etc.

General References:

1. wiki.libsdl.org
2. documentation.help/SDL
3. lazyfoo.net/tutorials/SDL
4. www.geeksforgeeks.org

Game assets collected from:

1. craftpix.net
2. opengameart.org